Design Document

John Kommala - Li Honglin - Masanari Doi - Hiroki Kumagai

Overview :

We decided to work the parts individually. As we have decided to work on this individually, our classes were individual. We made sure that we got the functionality right first then built our interface on top of that.

Design & Approach :

Part-1 :

Our Idea to solve this was to first setup the graph using the three input files while computing their individual cost based on the file location they were coming from. Then to implement the shortest path algorithm for the graph.

- **Graph**
  We chose to create separate classes to create a graph, one of the classes being ***Stop***, this class is used to store information from stops.txt file, it has individual attributes corresponding to column labels in stops.txt file. Another class as such is ***StopConnections***, this class can be predominantly considered as a Graph class. This class heavily relies on HashMap<Integer, ArrayList<ConnectionNode>> *adjacencies*, We read stops.txt using *readStops()*, in this method we break down the information from stops.txt into a Stop variable and store these as keys in the adjacencies hashmap. Then we use *readTransfer()* to read transfers.txt file and compute associated cost and use the information from and to stop IDs and the cost with *makeConnection()* adding a directed edge corresponding to the information. We use another class called ***ConnectionNode*** to assist with this, this class stores the stop_id and the cost associated. So far we are done adding the direct edges for the graph, now we have to add consecutive edges between stops from stop_times.txt. To read and process information from the stop_times.txt we use another class called ***Trips***, which builds on another class called ***TripDetails*** which has individual attributes corresponding to the column labels in the stop_times.txt file. We read the stop_times.txt file and filter out invalid times and then add them to an arraylist of type TripDetails in the Trips class. Now in order to finish setting up the graph, we use a simple for loop starting from 1 to number of valid trips(based on the trip time). We compare all the consecutive trips in the loop and use *makeConnection()* to add a consecutive edge between 2 consecutive stops with the same trip id. This concludes our graph set up as we have created all the possible directed edges and consecutive edges.
- **Shortest Path Algorithm**
  We choose to proceed with ***Dijkstra's Shortest Path Algorithm*** as opposed to Floyd Warshal's considering assuming it would be time efficient to find the shortest path between two given stops as opposed to calculating the shortest path between all the stops. We did not consider A* algorithm as we felt that we didn't have enough information required for some of the functions to use A*.

Part-2 :

Our approach for part 2 was to create a TernarySearchTree class to store the stop names and then implement the search functionality and return all the matching stop details using a map.

1. **TernarySearchTree class**
   This class consists of *insert()* and *search()* that are most relevant to our problem, we use *insert()* to add in stop names and build out the whole TernarySearchTree. The *search()* relies on two other methods to return all the possible matching stops. These methods are *crawlToPrefixLastNode()* and *findAllSuggestions()*. Whenever we search for a specific stop name, the method *crawlToPrefixLastNode()* helps in acquiring the prefix root based on the word being searched for. We then use this prefix root in the *findAllSuggestions()* to find all the matching stops names from the TST, this method is recursive and searches based on traversing through the left, middle and right nodes from the prefix root. This class also relies on another class called ***TSTNode***, the node that makes up the TST. This node has the following attributes, data - to store the character, isEnd - boolean to mark the end, Three pointers to further nodes called left, middle and right, which are of type TSTNode.
2. **makeMeaningful(String stopName)**
   This method is a recursive method that checks if the stopName has the following keywords flagstop, wb, nb, sb, eb at the start of the stopName and adds them to the end of the stopName if any.
3. **Map<String, ArrayList<String>> createNameDetailsMap(File filename)**
   This method returns a HashMap in which the keys are the stopNames and values are the corresponding string array consisting of the related information from the input file stops.txt.
4. **main()**
   First we store all our stopNames into a TernarySearchTree, using a method called *getStopNames()* that reads the stops.txt file and returns an arraylist of the stopNames, we then make sure that they are meaningful using *makeMeanigful()* and then add them to TST. Then from the user input we use TST.search() to obtain all the matching stop names. Finally using the map generated from *createNameDetailsMap()* we obtain all the details for the matching stops and present them as output.

Part-3 :

Our interpretation of part 3 was that we would need to fetch information from stop_times.txt file and return them based on their arrival time in a sorted order, while filtering the file for valid times, i.e. no more than 23:59:59. We used the following functions to do that,

1. **boolean isTimeValid(String time):**
   This function takes in input time in the form of a String and checks if it is valid or not by breaking the input time into hours, minutes and seconds and checks whether those are under maximum limits and return a corresponding boolean, i.e true if the time is valid and false otherwise.
2. **ArrayList<String> getValidTimes(File filename):**
   This function takes in an input file, in our case stop_times.txt, uses BufferedReader paired with FileReader to read the file lines in the file to be specific. If both the arrival and departure times are valid we add that corresponding line to an Arraylist<String> to be returned and we return this when reading the file is complete.
3. **Map<String, ArrayList<String>> createSortedArrivalTimeMap(ArrayList<String> validTimes)**
   This function takes in an ArrayList validTimes, which contains lines of data from the file stop_times.txt if they have valid arrival times and departure times. In this function we create a TreeMap, the key of the map being the arrival time as we have to fetch all the trips on a given arrival time. The idea behind using a TreeMap was that it would sort information based on the keys while being created. To this map we would add information based on the arrival time as the key, while adding the whole line to the value to be returned. We return this map once we're done.
4. **sortTripsBasedOnID(String[][] tripsData)**
   This method sorts this 2d array based on trip id which is stored at index 0 in every sub-array using bubble sort and returns a sorted array.
5. **main()**
   This is where we bring all of the above methods to get the functionality of part3. We pass in the file to *getValidTimes(File filename)* and then create a map using *createSortedArrivalTimeMap(ArrayList<String> validTimes)* and then use this map to fetch all the trips at a given time and then finally use *sortTripsBasedOnID(String[][] tripsData)* returning full details of all trips matching the criteria (zero, one or more), sorted by trip id.

Part-4 :

Our approach for UI was to use the methods and functions from the previous parts and check for all the possible errors, fix them and provide front interface enabling selection between the above features. We have managed to do this, our application consists of 4 screens, one of them being the home screen and the rest being one for each of the previous parts.