ADS Design Document

Cian Jinks, Vitali Borsak, Ajchan Mamedov, James Cowan

Overview

To create our project we decided to implement the following data structures and functionalities:

Graph Class

A graph class was needed to store the entire bus network of the city. This graph needed to be directed and edge-weighted as per the specification. Each node in this graph represents a given bus stop using the bus stop ID, and using an adjacency list we represent the various routes and transfers within the network. An adjacency list was chosen over other graph representations such as an adjacency matrix as there were a large number of stops so this would lead to lots of memory usage.

Immediately there was a problem as bus stop IDs are not increasing from 0 to 8758, they are instead seemingly random. This means the adjacency list needed to be implemented as a *HashMap* instead of an array. Doing so means we could still keep near O(1) lookup times. To keep things simple this adjacency list used bus stop IDs to reference each stop and if more information was needed about a given stop a lookup table was built into the graph class to retrieve a BusStop class for a given stop ID. This was also implemented using a *HashMap* for the same reason as before.

To store the edges between the nodes in the adjacency list we created a class called *NetworkNode* which simply acts as a pair tying an edge weight to a stop ID.

Shortest Paths Algorithm

The first functionality of the project required the ability to find the shortest path between any two bus stops in the network. To implement this we chose to use Dijkstra's Shortest Path algorithm. The reason we did not choose A* was that we did not have enough information to create a meaningful heuristic function. Floyd-Warshall would also not have worked as it has a much longer runtime than Dijkstra and does not record the path taken. Using a small modification we optimised Dijkstra to break as soon as it found the shortest path to our target stop and return the distance plus list of stops taken along that path.
This functionality is built into the graph class using the function *BusNetwork::getShortestPath()*.

Ternary Search Trie Class

A Ternary Search Trie Class was required to store the names of different stops and all bus stops associated with those letters. This class has an *insert*() function which inserts a *String* (name of the bus stop) and *Integer* (stop ID for that stop) into the

Ternary Search Trie, and also a *search*() function which can be used to search through the Ternary Search Trie and return an *ArrayList<Integer>* of stop IDs matching the search.

A private Node class was also created to store the character and an *ArrayList<Integer>* of all stop IDs of stops where their name matches the characters up until that point (the char and the stop ID's list are updated in the *insert()* function).

The functionality for searching for a string in the trie as the specification requires is provided by the *BusNetwork* class under *BusNetwork::searchTrie()*. This function makes use of the *search()* function discussed up above but uses it along with the lookup table to convert the bus stop IDs to *BusStop* classes.

Reading in Stops and Transfers + BusStop Class

In order to read at the bus stops, we created a file-reading algorithm that iterates through the lines with the *Scanner* class. Inside the while loop, the line is split at each comma in order to fetch its values using built-in parser methods. Additionally, the *stopName* is adjusted for WB, NB, SB and EB labels being moved to the end. These values are then translated into a constructor for our *BusStop* class which stores the stop information in a way that is accessible throughout our code. This class is then added to the *BusNetwork* and *searchTrie* as a node so that it can be accessed by the UI.

For the transfers, another *Scanner* class is used to iterate through the lines of the file. Each line is split at the comma so that its parts may be parsed using built-in parser methods. If the *transferType* is *0* or *2* then the transfer is added as a connection, which represents an edge, in the *BusNetwork*. If it is a different transfer type, it throws an error, as this should not occur with the sample input.

The *BusStop* class serves as a holding class to store information about stops in the network. It holds stopIDs, descriptions, latitude and longitude, and other useful information about stops that is utilized in our program.

Reading in Routes + Route Class

We needed a class that read and store the information from the stop_times.txt file, so we created a *TripDatabase* class. The *TripDatabase* stored an ArrayList of *Trip*s which each are made up of *TripSection*s. The *TripDatabase* has a method called *readTheStopTimeFile* that reads the information from the file line by line, then splits it by the commas and stores it into the *Trip* list accordingly. We use the *split()* function instead of *hasNextInt* because the information in the *stop_times.txt* file isn't homogenous.

We also added a *TripTime* class which parses and stores a *String* from the form of (HH:MM:SS) into corresponding variables, this class also has a *validate* function that

checks if the hour is not above 24 hours and that the minutes & seconds are not above 60 and returns true or false. Lastly, it has a *compare* function for comparing other times to it.

Searching for Arrival Times

The third functionality of the assignment was to add the ability to search for trips by arrival time and return all corresponding trip sections ordered by their trip ID. To implement this we chose to sort the trips by trip ID ahead of time when they were loaded in from the stop times file. As there is a very large amount of trips (>60000) we opted to implement quicksort for this to provide the best performance. One problem with this is quicksort lacks stability. This would cause a problem when displaying the trip sections in order of time so we could have used mergesort. However, luckily we circumvent this problem by the way we store each trip as a collection of *TripSection*s in one *Trip* class with an associated trip ID.

With the trips loaded and sorted by trip ID, we could then loop over each trip and search for the requested arrival time. After a quick test, we discovered that the individual sections of each trip were ordered by their arrival and departure times already. This made it possible to make use of binary search to significantly speed up the locating of a given arrival time as opposed to using linear search.

This functionality can be found within the *TripDatabase::searchForArrivalTime()* function.

User Interface

For our UI, we decided to use a popular java UI library called JavaFX. We implemented a preloader that displays a window that has a message notifying the user that the program is loading, during this time, all the necessary information is being read in from the files, parsed and stored. Once loading is complete, you will be prompted with the Home Page, which contains the name of the app, description, three buttons that bring you to the functionality outlined in the assignment specification and finally an about button, which displays the information about the team and their contributions to the project.