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.

Immediately there was a problem as bus stop IDs are not increasing from 0 to 5100, they are instead seemingly random. This means the adjacency list needed to be implemented as a HashMap so 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 is built into the graph class to retrieve a BusStop class for a given stop ID. This was also implemented using a HashMap.

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* because we did not have enough information to create a meaningful heuristic function. 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) 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 its lookup table to convert the bus stop IDs to BusStop classes.

Reading in Stops and Transfers + BusStop Class
readTransfers - add each as edge
readStops - add each as a node

Reading in Routes + Route Class
We needed a class that read information from the stop_times.txt file, so I created a RouteSection class. RouteSection has a method called readTheStopTimeFile that reads the information from the file line by line, then splits it by the commas and puts the information into the corresponding variables. I use the split() function instead of hasNextInt because the information in the stop_times.txt file isn't homogenous. The information then is added to the ArrayList in the form of Object of the class TripSection(a private node class) in the method called addRouteSection. 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 which checks if the hour is not above 24 hours and that the minutes & seconds are not above 60 and returns true or false.

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 such trips 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. With the trips loaded and sorted by trip ID we could then loop over each trip and search for the requested arrival time. Luckily, after a quick test, we discovered that the individual sections of each trip were ordered by their arrival and departure times. This made it possible to make use of binary search to significantly speed up the locating of a given arrival time.
This functionality can be found within the TripDatabase::searchForArrivalTime() function.