CSU22012: Algorithms and Data Structures II Group Project Design Document

---

Group Members: Aaron Readman-19334583, Brian Bredican-18328825,
Ralph Swords-19335541, Tom Roberts-19335276.

1. Introduction

1.1. Project Specification

This project is an implementation of a bus management system based on Vancouver Bus system data. Using input files obtained via TransLink Open API.

Required Functionality:

1. **Shortest Path** - Finding the shortest path between 2 bus stops, given by the user, and returning a list of the stops en route, as well as the associated "cost".
2. **Ternary Search Tree** - Searching for a bus stop either by full name or by the first few characters, using a ternary search tree, returning the full information for each stop matching the search (which can be zero, one or more stops).
3. **Searching all Trip**s - Searching for all trips with a given arrival time, returning full details of all trips matching the criteria (zero, one or more), sorted by trip id.
4. **Front Interface** - front interface enabling selection between the above features or an option to exit the programme, and enabling required user input, with error handling.

---

2. System Design

2.1. Design Overview

Since there were four functional requirements, and there were 4 members in our group, we decided to assign a requirement to each individual member.

2.2. Shortest Path - Ralph

In this part of the assignment, the program must find the shortest path between two stops given by the user. This is clearly a graph problem, where each stop is a vertex, and the routes and transfers between stops are the edges.

Having established that this is a graphing problem, we now must consider the various graphing algorithms we have encountered. Depth First Search can establish if two stops are connected, but it doesn't guarantee a shortest path. Breadth First Search also is not appropriate in this case since the edges are weighted. Since the graph is likely to contain cycles, Topological Sort is not appropriate. This graph represents bus routes, so it will not contain negative cycles. Therefore, Bellman-Ford is not necessary.

Since this problem is in the form of single-pair search, Greedy Best-First Search or A* might seem to be the best options. However, these algorithms require heuristic functions, which we do not have. Therefore these algorithms can not be used.

That leaves Dijkstra and Floyd-Warshall. Both algorithms, given two stops, would be able to find the shortest path between them. Dijkstra performs better than Floyd-Warshall in terms of time and space, with a time-complexity of O(E logV) compared to O(V^3), and a

space-complexity of O(V) compared to O(V^2). Therefore, I decided to use Dijkstra's algorithm as the basis of my solution.

2.3. Ternary Search Tree - Tom

The next feature we had to implement was an option to search for a stop using a ternary search tree. Since we were required to use a ternary search tree to implement this, there wasn't and choosing that we needed to do when it came to data structures.
The stop data was saved as an arraylist of Stop classes, which consisted of a stop's individual data, and we populated the ternary search tree in the initialisation portion, at the start of our program.
Ternary Search Trees are efficient at both searching for and inserting nodes. The ternary search tree operations have a similar time complexity to the binary search tree operations. e.g. the time it takes to insert and search is proportional to the height of the ternary search tree. The amount of space required is proportional to the length of the string that will be stored.
Both the search and insert operations have an average time complexity of O(logn), with a worst case time complexity of O(n).

2.4. Searching all Trips - Brian

In this part of the assignment we have to search for all trip information given an arrival time from the user and return all the relevant trip information. We also must delete all information from the list which contains invalid arrival time information. First we created a class, StopTimes that parses information from a line of the list separating all necessary information in array format. They then each are given a get method which makes them accessible in other classes.

In the class ArrivalTimes we read in the stop_times.txt file and put them into ArrayList format. In the method tripInfo we start by looping through the ArrayList and deleting the invalid arrival times that don't fit the correct format eg 26:00:00. This loop runs through the list line by line giving it time complexity of O(N).

Now we loop through the list using our get method for the list's arrival times and comparing them to the user input. When they match we use our get methods for all other information and adding them to a string then adds them to the method's ArrayList return. As this is looping through all values again making it O(N) meaning the method as a whole runs as O(2N).

2.5. Front Interface - Aaron

The User Interface was designed to be accessible through the command line. This was done to make it more accessible to people as it can then be used in various consoles in IDE's or they can just compile and run it via command line. The only important requirement for it is that java is installed on the user's machine and all the .txt files are in the same directory as the program.