CSU22012 Algorithms & Data Structures II

Design Document

Group Members	**Student ID**
Bernadine Lao	19333627
Merlin Prasad	19333557
Filip Kowalski	19334250
Darren Aragones	19335456

Introduction

We created a system to help passengers riding the Vancouver public transport system. We ran our code on Eclipse version 2019-06 (4.12.0). No additional dependencies were required.

We were asked to provide the following three functions:

1. Finding the shortest path between 2 bus stops (as input by the user), returning the list of stops en route as well as the associated “cost”.

2. Searching for a bus stop by full name or by the first few characters in the name, using a ternary search tree (TST), returning the full stop information for each stop matching the search criteria.

3. 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.

We decided to use a terminal based user interface that handles erroneous input from the user and gives them feedback on what went wrong. It was a simple but effective way to provide the user access to the necessary functions. The user is given simple instructions on how to navigate through the console.

Finding the shortest path was implemented with an adjacency matrix and Dijkstra. We asked the user to input the stop_id of the two stops they wanted to find the shortest path between.

Searching for a bus stop was implemented by a ternary search tree. The program suggests every possible bus stop name that matches the user’s input. An array list of results is returned in alphabetical order.

Searching trips by arrival time was implemented using an ArrayList. An object StopTimesInfo was created. A function getStopsInfo() takes in arrival time preferred in the format hh:mm:ss as parameter and returns a list of stops with the given arrival time as well as the information related to it sorted in an ascending order by the trip_id.

Shortest path between two stops

Data structure: Adjacency Matrix

- Space complexity: O(N^2)
- Access time: O(1)

An adjacency matrix represents all paths. We decided to use this over an adjacency list as it did not require any additional classes to be built. An adjacency matrix has a space complexity of O(N^2).

xThe search time to access an element in the array is O(1) making our algorithm very fast for repeated searches on the same graph. Searching in an array is much quicker than searching in an adjacency list consisting of lists of variable length.

We traded space efficiency for faster search access time. This is why we decided to use an adjacency matrix to represent the graph for this method.

Algorithm: Dijkstra

- Time complexity : O(N^2)

The graph contains no negative weights as we are measuring distances nor should any one user need to know the distance of more than a few bus routes. We used the Dijkstra algorithm to find the shortest path between two nodes. It has a time complexity of O(N^2). This is much quicker than some of the other algorithms we considered such as Floyd which has a time complexity of O(N^3). Given the large data set we are dealing with time efficiency is crucial. Dijkstra is one of the most optimum algorithms to find the shortest path between two nodes in a graph.

Searching for a bus stop

Data structure: Ternary Search Tree (TST)

- Search hit: O(L+ln(N)) average case
- Search miss: O(ln(N)) average case
- Insert: O(L+ln(N)) average case

The ternary search tree (TST) algorithm takes advantage of a limited radix when searching for strings. The stop names are extracted from the stops.txt file and used to construct the tree, with some string manipulation beforehand to remove the common prefix of ‘WB’ and ‘EB’. As we are not simply finding whether the user input is recognised by the tree, a mixture of tree traversal types are implemented.

The ***search*** function works by traversing the tree according to the user’s input and finding the last letter of the user’s input that is represented in the tree. If there is no such letter then there are no prefix matches. A function named ***match*** then runs on the node representing that last letter returned by search, which finds every node with a final value and thus a full word. **Search** uses char key comparisons while **match** uses inorder traversal of a tree given a root that is not necessarily the root of the entire search tree.

Search all trips with a given arrival time

Data structure: ArrayList

- Space complexity: O(N)

Algorithm : Collections.sort (Merge sort)

- Time complexity : O(NLogN)
- Space complexity: O(N)
- Stable : Yes
- In-place : No

We used the Collections library sort which implements merge sort to sort the data in ascending order. The prevailing reason was because merge sort is stable. This ensured the output was in proper chronological order as its other variables [e.g. start time] are preserved after sorting compared to Quicksort which is not stable even if it has a better space complexity.

Collections.sort also works on ArrayLists which is why we did not use the Arrays.sort function.

Merge sort also has a guaranteed worst case runtime of O(NlogN) compared to quick sort which has a worst case time of O(N^2). Merge sort is optimum for sorting an array of objects and this is why we used it to implement the sorting.