Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Project Design Document

Algorithms & Data Structures II

April 2020

Michael Makarenko

Adam Bewick Mulvihill

Prathamesh Sai

Daniel Ilyin

Design decisions for Part 1

For the first part, we used the busStopMap object to create an edge weighted digraph representing the routes between stops. Reading in files had to be fast since this part read in all three input files, so we used a BufferedReader instead of a FileReader or Scanner. To make it easier to create the digraph, as well as to make it easier to actually work with the digraph, this object also has three maps containing key-value. These maps were used to get a stop's ID using its name as a key, get the stop's vertex in the digraph using its ID as a key, and to get a stop's name using its vertex number as a key. Initially we tried using an ID indexed array for getting the vertex, but this wasted space as some IDs did not have associated stops, and getting a stop's name from its vertex proved too slow. We decided to use HashMap over TreeMap as these maps did not need any specific order for this part of the project. The digraph itself was implemented using an adjacency list of WeightedEdge objects, where each WeightedEdge contains the vertex it comes from, the vertex it goes to, its weight. For finding shortest paths, we initially tried using Floyd-Warshall, with the hopes that we would be able to find the shortest paths between all pairs at the start, which meant it would not have to be run again. However, with a worst-case asymptotic running time of $O(N^3)$, where N is the number of vertices in the graph, we found that Floyd-Warshall was far too slow. Instead we decided to use Dijkstra with a priority queue that used a binary heap, meaning it had a worst-case asymptotic running time of $O(E \log V)$, and found it was far faster, even when having to run it each time we searched for a shortest path. Since most of the methods in BusStopMap relied on being passed a valid stop name, all methods make sure that the name they are passed is an existing stop, and throw an IllegalArgumentException if the stop does not exist. These exceptions are handled in mainApplication. Dijkstra was called from a method in BusStopMap called makePaths, which took the starting stop's name as an argument and, using the HashMaps, got that stops vertex number and passed it to Dijkstra, as well as a reference to itself so Dijkstra could use methods to access the adjacency list and the number of vertices. From there, the method getCost simply accessed the distTo array in Dijkstra. The getStops method called a method in Dijkstra called getPath, which starting at the destination vertex used the edgeTo array to return an ArrayList of every vertex on the path. getStops then uses the returned vertex numbers to create an ArrayList of the stop names on the path by using the appropriate HashMap. These are then printed out for the user.

Design decisions for Part 2

For the second part, we used the stopName object to format stop names, populate the Ternary Search Tree and hold a reference to the TST. We used a **BufferedReader** to read each line of stops.txt, which is more efficient than using a simple FileReader. We didn't use a Scanner as we had no need to parse every token, only the stop name. So we split each line and then split each name, into arrays of Strings. We then converted the name into a **List** of strings, converted into a corresponding **LinkedList**. A LinkedList is faster at manipulating data (which is what we are doing, removing and inserting) while an ArrayList is better for storing (less space) and accessing data. Since our LinkedList is short-lived, space isn't a priority, and we aren't accessing anything beyond index 0.

We implemented the TST faithfully, using the minimum number of member variables necessary for both the TST class and the TSTNode subclass. The methods use simple String and reference manipulation. The allNames list is used to store and allow us to retrieve the line numbers of every

entry in stops.txt, because this list is almost as long-lived as the TST itself, and we must quickly be able to access every element, it is most efficient to implement as an **ArrayList**.

When case 2 is selected in mainApplication.java we use the scanner to input the users search query and then make a new stopName object. We then pass the users input into the get(searchQuery) method which will determine if there is one result in which case we return the last node, or if there are many results meaning we need to use the recursive method Traverse(node,string) which when passed node and a string as arguments, will follow the left, mid and right subtree to the end and add the value of every node that isn't null, to the ArrayList allNames. Then the get(searchQuery) method will return a value, if greater than 0 will print all the matching stops using the printStopNamesMatchingCriteria() method, and if less or equal to 0 will print out that no matching stops were found.

Design decisions for Part 3

For the third part, we decided to turn each line in stop_times.txt into its own **stopTime object.** To store those we decided that a List was the easiest way to do so, as we could easily use the **add()** function with an O(1) time complexity to add an element to the end of the list. This was much easier to understand and implement than using a resizable array where we had to copy the elements and make a new array with O(N) time to append an element to the end of the array. Also, we needed a data structure to hold those lists of stopTime objects according to their arrival time. We originally had Array-Lists and HashMaps in mind, and we thought of the pros and cons of both, taking into consideration both space/time tradeoffs as well as readability and simplicity. In terms of space complexities, HashMaps and Array-Lists had very little differentiating factors. The space complexity of both is normally O(N), and if you look deeper into the Java code for both classes, they are internally represented as arrays either way. Therefore, there was not much of a difference here. Hence, we then looked at time-complexities for both data structures. For Array-Lists, to access an element with **get()**, it is an O(1) operation however if we treated the indexes as the arrival times we realized that it would quickly get complicated. Furthermore, if we tried to add the modified list of stopTime objects to a particular index with **add(index, element)**, it runs in average O(N) time. We quickly realized that this was not viable, so we decided to use HashMaps instead. Using HashMaps, we were able to simply use the key as the arrival time, and let the type of mapped values be the lists of stopTime objects with that corresponding arrival time. This gets rid of the problem of using indexes, and also provides an amortized O(1) **getOrDefault()** function for accessing the list of stopTime objects with a particular arrival time and an amortized O(1) **put()** function for storing a new list of stopTime objects back into the HashMap. In terms of sorting algorithms for sorting the matching trips by trip ID, we had mergesort, insertion sort, and quicksort in mind. But quicksort was unstable, and we wanted to ensure the algorithm we used was stable so the stopTime objects were still in order. We also realized that top-down merge sort was slightly more efficient than bottom-down merge sort for caching reasons so we decided to go for that. **Top-down mergesort** was stable, and had a worst case asymptotic running time of O(N), but was not great for smaller arrays. Therefore, we decided to use **insertion sort** for the cases when we had to sort smaller arrays - a time when insertion sort thrives with it's O(1) asymptotic worst case running time. The mixture of both merge-sort and insertion sort led to our sorting algorithms being extremely efficient.

Design decisions for Part 4

One of our error checking methods is validTimeFormat() which is implemented by parsing 2-character substrings as integers to ensure 24-hour digital time format is being followed (as well as the : characters), without any unnecessary data structures nor loops to ensure big-O constant time. Another one, yesNo(), features just one user-dependant loop as it deals with parsing user input and avoids needless algorithmic or structural complexity. The method getStop() works similarly, completing in O(u) time, where u is the amount of time the user takes to enter a correctly formatted query. Here we use a regular expression to efficiently determine if the user enters a single integer. An earlier, less efficient and more buggy alternative was to use the scanner to manually parse every single token the user enters but we later found that taking the entire user input line as a String and parsing that one string as necessary is far more time-efficient, although we sacrifice a miniscule amount of space on storing said String.

In terms of the UI as a whole, we used ASCII art and a couple Unicode characters (such as in the progress bar) to make the most of the command line interface and make the instructions to the user as clear as possible to reduce chances of user error.