Niall Sauvage

20334203

Final Project Design Document

CSU22012 – Algorithims and data structures ii

# 1)

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

In order to efficiently complete this task, I decided to use Dijkstra’s algorithm, as it is not only relatively simple and easy to implement, but it will find the shortest path between only two vertices whereas Floyd-Warshall uses a cost matrix from every vertex to every other vertex. In order to represent my graph, I am using an adjacency list, which is an array of size V (where V is the count of vertices in the graph) with each entry representing a unique vertex. The array itself stores Linked Lists at every entry, which themselves store edge objects, representing the edges leading out from that vertex. Additionally, for convenience, I created a LinkedList which stores all stop (vertex) objects in the graph. I also used a hashmap so that I could look up a stop ID and have a stop object returned in 0(1) time. Finally, to represent the path itself, I used two arrays of size V, one which stored doubles called distTo[] and the other which stored stop objects called prevTo[]. distTo[v] stores the cost of the shortest path to v from the stop we’re starting from, while prevTo[v] stores the stop previous vertex to v along the shortest path from the source. By simply following this back and then reversing it, we can find the shortest path to v.

The user is first prompted to enter the ID of the stop they wish to go from and the ID of the stop they wish to go to, then we retrieve those stops using the Hashmap in time, as mentioned above. Next, Dijkstra’s algorithm is executed, which returns a Path class, which contains the cost of that path and an ArrayList of path stops. This is then passed to a method in FinalProject.java, which prints the path and associated cost.

The time complexity here trends towards as we are selecting the vertex v with the least distance to the source as it must select the minimum from a list of (at worst) v vertices, then we are relaxing every edge adjacent to that vertex, for every vertex in the graph. Thus, it is .

The adjacency list was created (read in) at the very start of the programme, to allow for the user to search multiple times, with data residing in memory for the duration of the programme, thus allowing the user to get results rapidly. Each vertex is given a unique ID by the programme, so that the adjacency list will not have empty entries (i.e. the first stop read in is 1, second is 2, etc). In the code, this is referred to as map\_id. It allowed for far easier indexing of the prevTo and distTo arrays.

# 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 (which can be zero, one or more stops)”

This necessitated the implementation of a Ternary Search Trie (TST) as it was the most cost-effective way to implement a String Symbol Table, with L + ln(N) time complexity for a search hit and a trivial way of searching for prefixes (recursively return all nodes in the trie that are children of the node representing the last character in the prefix). The same time complexity exists for insert (both in a typical case), thus it is efficient to both build and search the TST.

Similarly to the search for a shortest path, the Ternary Search Trie is created at the start of the programme to allow the user to search the same Trie multiple times without having to wait for HDD/SSD accesses.

# 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.”

To do this, I created a priorityQueue sorted by a comparator which would order the elements of the queue based on their trip\_id. Subsequently, I iterated over the queue and returned elements which had the same arrival time in an Array List, without the need for sorting as they were already sorted due to the comparator. This will run in O(N) linear time. An alternative method would be to sort via the time and return all entries (M entries) with the same arrival time. Thus, we could use binarySearch to find the first example, then iterating until we reach a time that is not the same as that being searched for. Afterwards, however, we would have to sort these items by trip\_id, leading to a time complexity of (binary search) \* (iterate for all values with same time) \* (mergesort on returned list) or log(N) \* M \* Mlog(M) which multiplies out to:

As such, there is a tradeoff: Should , i.e. small, then the second method is faster, however, it requires O(M) extra memory as mergesort is not in-place.

Once more, similarly to the previous two implementations, the priorityQueue is created when the programme starts in order to allow the user to search multiple times quickly. As touched on previously, this is a tradeoff between programme start time and search execution time – if the user is only making one request, it is much faster to read in after each request occurs as it does not need to read in all files, but if multiple requests are made, it is far more efficient to read in at the start.

The user is asked to input the arrival time they wish to search for, which is then read in through a Scanner object and the string is converted to a Time object, with an IllegalArgumentException being thrown, should the time inputted be invalid/in an incorrect format. The user is then asked to repeat the process and enter a valid String.

The output is generated by using a StringBuilder class, which dynamically adds spaces in order for the output to appear in a neat table format under the correct headers.