Vancouver Bus Management System Design Document

Luke Rabbitte (lrabbitt@tcd.ie)

Trinity College Dublin

CSU22012

April 10th, 2022

Finding the shortest path between two stops

(Input Files: stops.txt, transfers.txt, stop_times.txt)
(Runtime Files: Main.java, FindShortestPath.java, DijkstraSP.java, EdgeWeightedDigraph.java, DirectedEdge.java ShowScrollingText.java)

My program instantiates a graph of directed, weighted edges based on the adjacency list representation discussed in lectures. This means we maintain an indexed array of bags of directed edges. Each array index corresponds to a unique vertex in the graph. In my program I initially wanted the index of the array to correspond to the actual Stop ID. However, I quickly noticed an array overflow error. This happened for the following reason. We encounter 8757 stops. Because some Stop ID integers, for example 10205 on Hastings St, exceed the array boundary of index [8756], it is obviously not a good idea to directly match the array indices of our adjacency list representation to Stop IDs. Instead, I created a hash map which matches Stop ID keys to another variable called mapIndex, which is incremented every time a new vertex is added to the overall graph. I also found it necessary to maintain a reverse hash map which stored information in the opposite direction, from mapIndex keys to Stop ID values. Each of these maps will take O(V) space to store, where V is the number of vertices. However, once a hash map is implemented, we can insert new values and perform random access in O(1) time, which makes it a highly efficient option for the many get and put operations required by our program.

Once the graph is instantiated, we then add edges from transfers.txt and stop_times.txt, respectively, according to the specification sheet. Adding an edge to a graph with adjacency list representation takes O(1) time, as we are performing a constant-time array access followed by a constant time adding to a linked list. Each vertex-indexed array position holds a linked list of edges, so the adjacency list graph takes up O(V+E) space in total.

When given two valid Stop IDs from the user, my program checks that these Stop IDs correspond to vertices in our graph, and outputs an error message if not.

During the shortest path algorithm, I use a string builder to create the output of each stop along the shortest path. This is where I use my reversed hash map, as at this point in the program we need to be able to search Stop ID values by mapIndex key, rather than the original way of searching mapIndex values by Stop ID keys.

I chose to use Dijkstra, a single source shortest path algorithm. This is based on a min-heap priority queue and runs at a time complexity of O(V + ElogV), where V is the vertex count and E is the edge count. I considered using the A* algorithm which matches a single source vertex to a single sink vertex, as this could have been more efficient than the single source behaviour of Dijkstra. However, A* requires a combination of Dijkstra and a heuristic. I considered creating a heuristic function based on the latitudinal and longitudinal positions of the stops but decided that for the purposes of this assignment this was superfluous, as Dijkstra is still a very efficient algorithm.

Finding stop information based on part or all of the stop name

(Input Files: stops.txt)
(Runtime Files: Main.java, SearchForStop.java, ShowScrollingText.java)

This program reads stops from the stops.txt file, uses a string builder to move keywords to the back of the string, and then adds these strings to a ternary search tree. Ternary search trees have a maximum of 3 children per node, which is very space efficient when compared to the traditional trie or prefix tree which could store whatever the radix of the given alphabet is as the number of children per node. For example, if the alphabet is lowercase English characters, the number of children per node is 26. If the alphabet is decimal numbers, then the number of children per node is 10. Thus, the TST is a more space-efficient option that suits the purposes of our assignment well.

Each node in our TST stores the adjusted stop name string as the key, and length 10 arrays containing the full stop information as the value. The TST class contains a method that allows us to search for keys with certain prefixes, which caters for the required behaviour of searching by the first few characters of a stop name. Inserting into and searching a TST will take an amount of time proportional to the height of the tree (which could be O(n) in the worst case, where n is the amount of string keys to be stored) . However, the average case of insertion will be Θ (logn). The TST itself will take space proportional to the amount of stops that need to be stored.

It is worth mentioning here that my ShowScrollingText class in nothing but a convenient way of passing a long string and displaying a scrolling output pane, with credit given to the online tutorial I roughly based my design off.

Finding trip information based on a given arrival time

(Input Files: stop_times.txt)
(Runtime Files: Main.java, SearchByArrivalTime.java, ShowScrollingText.java)

This part of the program takes and error checks user inputted times, and stores as Date objects. Only valid times will be accepted. I created an array list of string arrays which contains all the parsed information about every possible trip. The goal with this piece of functionality was to return a subset of this larger array list containing only relevant string arrays whose arrival time at the arrival time index matched that given. This also had to be returned in ascending order of Trip ID.

I noticed that in the original file stop_times.txt, the Trip IDs appeared to already be in something that very closely resembled ascending order. Accordingly, I chose to implement a custom version of insertion sort, which is the most efficient algorithm for a set of integers that are already nearly sorted. The insertion sort algorithm will make O(n^2) compares based on the input set of size n (trip ID variables) but will only actually perform a swap if the two current elements are out of order. Its worst case is therefore still theoretically O(n^2), however in practice, if the elements are already basically in order this time complexity approaches O(n).

When searching through all trips to identify trips with matching arrival times, I used a simple for loop that runs in linear time through all options. Other search options such as binary search exist and have a better theoretical time complexity of O(logn) rather than O(n), however binary search mandates that the search array is already sorted. In this case, we could indeed have sorted all trips by their arrival time, and then created a binary search tree that goes recursively left if the time key given is less than the current node, and right if greater than the current node. However, the overhead of sorting all trips to begin with didn't seem worth it to me for this assignment.