Assignment 1 – Stage 1

Assignment 1 focuses on the implementation and manipulation of a train network. Using data read in from a CSV file, we have been tasked with completing a number of methods that will enable us to analyse and understand the data provided better. Furthermore, the focus of this assignment is on our ability to understand and write graph algorithms.

Stage 1 focused on being able to find the shortest relative distance between a set of two nodes. There were 5 methods to write – with two focusing on distance, two focusing on number of stops, and a fifth to calculate the distance in a set of stations. Given the description of the methods, it was obvious that we should use some path-finding function to be able to find the shortest path between the two given nodes. This meant that it would likely require an exhaustive search of the nodes in order to find the best path between the two nodes as we would need to iterate over the set, finding the shortest path between the set of nodes.

For each of the first four methods, I implemented an altered form of Dijkstra's *shortest path algorithm*. The algorithm has a set of values that track the distance of each node relative to the destination node and stops when we have exhaustively searched for this node, beginning from our origin node. It is a greedy implementation that operates by looking at all nodes in our graph, and consistently choosing the next best node available to it. This criteria of 'best' is defined as the node with the shortest distance relative to itself. We can say it is greedy due to this.

routeMinDistance

As described above, my implementation for this function was based on Dijkstra's shortest path algorithm. The algorithm does an exhaustive search on all nodes in the list of stations that have been read in. The algorithm, and my implementation of it, can be best described in a number of stages.

The first stage of the function is initialising all the values. All station nodes are set to 'unmarked' so that the algorithm knows they have not been visited yet. Then, all station names are placed into an ArrayList, *dist*, so that the algorithm does not have to check if they already exist as this is enforcing them to. Furthermore, I have implemented an ArrayList, *parents*, that has the purpose of storing the best previous station for each station. This will be better explained as the algorithm is explained.

The algorithm then initialises the distance from the starting node to 0 as the distance from the source node to the source node is 0. From here, we iterate through each station in our provided list that was read in from the CSV file. The algorithm then searches for the next best node relative to its current position. However, it is also ensuring that the next best node has not been touched yet as well. If we have already found the best path for a node, we don't want to do this again.

After finding this node, we then enter another loop that goes through each of the station adjacent to this *best* node. Each node is checked with a number of conditions before we can set any values that will alter our shortest path thus far. Firstly, we ensure that the node is not marked as we only want to visit unmarked nodes. We then want to ensure that our new distance to this node is not larger than the value that we already have to get here. If these conditions are met, we then update our *dist* and *parents* ArrayLists to reflect our newly found shortest path to this node.

The other methods that I have written are similar to the one described above, however, this one focuses on the distance to the path and uses the distance between adjacent stations to calculate a total distance.

Furthermore, a helper method was written that enables me to not reuse code in many of the other functions that I write for this section. It simply iterates through a HashMap that is passed as a parameter. This HashMap has a String mapping to another String – the relationship being that the first String is the closest station to the second String. This function starts at a passed origin node and continues until it reaches the final destination node, returning an ArrayList that can be returned in the main function.

For the implementation that incorporates any failures in the network, I simply added the following check at each adjacent node:

```
if (failures.contains(adj.getName())) {
     continue;
}
```

This line checks to see if the adjacent node is in the failures set that was passed. If it is, we utilise the `continue` keyword that will skip this iteration and ensure that a shortest path does not include this node.

routeMinStops

Similar to the previous method, my routeMinStops functions are heavily based on an implementation of Dijkstra's *shortest path algorithm*. In fact, these algorithms were mostly the same. Instead of checking for the distance in the path, the path was storing the shortest number of stops to that node. This was done using the ArrayList's `size()` function.

This algorithm was an exhaustive search to try and find the shortest number of stops between the two nodes. The Dijkstra implemented, as mentioned above, is a greedy implementation that looks for the next best fit when trying to select the best node to perform on.

Just like the previous method, there were minimal changes between this version and the version involving the failure set. All my second implementation did was check for the inclusion of the adjacent node in the failure set. Should it be included, it skipped this node to ensure it was not involved.

findTotalDistance

This algorithm was described to find the total distance in a set of stations. The function takes in a single parameter being an ArrayList of station nodes. Using this, we are able to calculate the total distance in the list. Being a simple algorithm, it incorporates a loop, a variable to track the previously visited station, and the running total.

In each iteration of the loop, the function checks if the previous item has been set yet. If it has not it sets it so that we can check the value of the next item. It should only not be set in the first iteration of the loop. Once it is set, it gets the distance between the current station in the loop, and the previous, looking up the distance between the two stations. The algorithm then adds this to the running total. Once it has exhausted all items in the list, it returns the value.

As described, my implementation is exhaustive in that it iterates through every single item in the list to be able to add their total to the running value.