Note for this that I will be using both directions of the tube.

Q2.1.

For This, I implemented DFS, BFS, and USC by implementing a Best-First Search with different evaluation functions. I have implemented separate functions for all of these, all using the base Best-First Search with the correct evaluation functions.

For the BFS, DFS, and UCS function, you need to pass the graph (station_dict) with the start node and goal node. The output from these functions is of the form (cost in avg time, solution path, nodes expanded). The entirety of part of the assignment is all on the same Jupyter Notebook.

The state is represented by a node in the search tree. The is encoded in this form (priority, second priority, (name, cost, children, actual cost). The priority is the priority number in the priority queue for that node. The second priority is the no. Of expansions up to and including that node and it is used to ensure Python's priority queue works correctly. The name is the name of the node. The cost is the cost w.r.t to the evaluation function (I.e., the cost so far). The children contain the actions of that state and the cost of each action. The actual cost is the cost w.r.t. the actual cost of each action, in this case, the average time is taken.

2.2.

I have run all my algorithms on all problems on the state space. From this, I see that USC has the smallest Average Time Taken with BFS closely following. We see that DFS is lagging with an Average Time Taken of nearly double the other two.

I have also printed out the solutions for (Euston, Victoria), (Canada Water, Stratford), (New Cross Gate, Stepney Green), (Ealing Broadway, South Kensington), (Baker Street, Wembley).

I will use this data to perform the analysis for this question.

So, looking at the Average Time Taken on all problems in the state space, we see that UCS has the lowest cost, followed closely by BFS. DFS is behind with nearly double the cost of the other two. Based on the problems I also solved above, I see that UCS always outperforms or is equally as good as the other two. This is expected as UCS is Optimal Cost. On most problems, BFS is close behind UCS. DFS is usually far behind with average time taken.

I will use the Route (Ealing Broadway, South Kensington).

The path costs for BFS is 20 with 135 Nodes Expanded and 9 nodes visited. The path cost for DFS is 51 with 970 nodes expanded and 25 nodes visited. The path cost of UCS is 19 with 141 nodes expanded and 9 nodes visited.

BFS chooses the next node in its frontier by the depth of the node. So, a small number of nodes visited, like 9, is expected as such. But this is not the optimal path cost as BFS will select the first node that is a solution and as such is not optimal where the actions have different costs.

DFS chooses the next node in its frontier by the negative of the depth of the node. This means that it will expand down a particular route and if it finds a solution, the path to the solution will most likely have a larger cost than the optimal path. So a large number of nodes visited, like 25, is expected. DFS is not optimal and will choose the first path it encounters that leads to the goal state. A path cost for DFS is expected.

UCS chooses the next node in its frontier by the cost of the smallest action from the start. The nodes visited by this is really that predictable as it depends on the path costs. But UCS is Optimal and as such, this is the best solution.

Both BFS and DFS do not consider the action costs, only the depths, so they are not optimal in this case.

For this part, I also used the route (Ealing Broadway, South Kensington).

I am unsure as to what this question is asking. I am interpreting it as, reverse the order of the elements in the frontier with the same priority (I.e., for [(1,"Euston"),(1,"Victoria")] I will take this as [(1,"Victoria"), (1,"Euston")]).

Looking at BFS, the path taken is exactly the same. This is expected as BFS expands by depth and will stop when it expands the goal node. So, in this, the path will be the same as BFS expands the same values no matter the order. Note that the number of nodes expanded is less in the normal direction with 135 compared to 139.

Looking at DFS, the path taken is different. This is because of the nature of DFS. DFS will expand as far as possible down a particular path until it reaches the goal or a dead-end. So, by changing the order of the elements at the same priority in the priority queue, different elements are popped and expanded. This will most likely result in a different path (unless that path is a dead-end). In this case, the reverse direction has a larger cost with 57 compared to 51 in the normal direction.

Looking at UCS, the path is the same. This is expected behavior, as UCS expands the nodes that have the smallest by the action cost from the start to that node by the shortest path. Also, UCS is cost optimal and as such the total path cost should be the same no matter how it is expanded. In this case, the reverse direction has expanded less nodes with 122 compared to 141.

I overcame the loop issue by using a graph search rather than tree-like search. This entails keeping track of the nodes and checking for redundant paths, which includes loops. More specifically, in my code, I add a child to the frontier if it is not in the reached nodes or the next cost to that node is less than the one, I have stored. This prevents loops by simply checking to see if a node has been expanded and only adding it to the list to be expanded (frontier) if it hasn't been expanded or we have a shorter path to that node.

2.3.

I have implemented a Best-First Search using line switches. I then used this to create a UCS with line changes. I assumed that the time to change lines was 2 units (I.e., minutes). The route I chose to do here was (Marylebone, Shepherd's Bush). In this, the standard UCS has a cost of 13 and makes 4 line switches. The new UCS with Lines has a cost of 16 but it only makes 1 line switch. This has drastically changed the path to minimize line switching.

Looking at BFS and BFS with Lines, we see that the path is the same, but the average time taken for the journey is much higher for BFS with Lines. This is because the BFS algorithm doesn't account for this extra cost due to line changes as it is only concerned with the depth of the node and not the action cost. This is the same for DFS and DFS with Lines with the same reasoning.

2.4

$$h(n) = \left|mean\big(zone(n)\big) - mean\big(zone(goal)\big)\right|,\ if\ n\ is\ a\ number.\ Else\ h(n)\ =\ 0$$

For my heuristic, I took the absolute value of the difference between the zone of the node and the zone of the goal.

If a node had 2 zones, I took the mean of them. If a zone had a non-numeric node, I discarded the Heuristic all together and set it to 0 as this gives no information so it is unusable.

I chose this, because it uses the zone information in a very simple manner. But also steers the search in the right direction if it is moving away from the goal.

For this, I looked at problem (Shepherd's Bush, Mile End). Comparing my BFS with heuristic only, I see that the cost is much higher than that of UCS. But the nodes expanded in the Heuristic is much lower.

3.1

The secret phrase for me is: L7F9SSp24R

3.2

The state is encoded as a string over the finite alphabet of Capital Letters, digits, and underscore. I further encoded the state to a numerical representation from 0 to 36 to better work with the data.

Selection was done based on the criteria that the parents must have a larger closeness score than the mean of the population at the start of that iteration.

Crossover was done randomly selecting the Kth character from either parent 1 or parent 2, where they both have an equal chance of being chosen, for all K up to 10 (the length of the word).

Mutation was done by looking at each numerical code for the Crossover and adding a [0,1,-1] with weights [8,1,1]. So, there is an 8/10 chance that no mutation occurs on that character and there is a 2/10 chance that a mutation does occur.

3.3

```
Population size:  4
Reproductions till convergence:  939
Population size:  10
Reproductions till convergence:  248
Population size:  20
Reproductions till convergence:  237
Population size:  40
Reproductions till convergence:  45
Population size:  80
Reproductions till convergence:  53
Population size:  160
Reproductions till convergence:  48
Population size:  320
Reproductions till convergence:  27
Population size:  640
Reproductions till convergence:  25
Population size:  1280
Reproductions till convergence:  24
Population size:  2560
Reproductions till convergence:  15
Population size:  5120
Reproductions till convergence:  13
Population size:  10280
Reproductions till convergence:  16
```

I looked at a population size of 100 which I ran 100 times. I got the mean reproductions to conversion as 60.94 with a standard deviation of 25.6876.

3.4

Looking at the population (as shown above). The reproduction till convergence is very high with a small population but then it decreases and then increases again slightly.

Looking at the mutation rate, a small mutation rate results in a lot of reproductions till convergence and increasing the mutation rate decreases it. But as the mutation rate increases beyond this point (0.4), the reproduction till convergence increases again.

```
Mutation rate:  0.001
Reproductions till convergence:  551
Mutation rate:  0.05
Reproductions till convergence:  103
Mutation rate:  0.1
Reproductions till convergence:  108
Mutation rate:  0.2
Reproductions till convergence:  78
Mutation rate:  0.4
Reproductions till convergence:  71
Mutation rate:  0.6
Reproductions till convergence:  197
Mutation rate:  0.8
Reproductions till convergence:  2786
```