ECS759P Assignment 1

Dimitrios Mylonas - 180326363

2 Agenda Based Search

2.1 Implement DFS, BFS and UCS.

In this part of the assignment the optimal routes from a station to another were found using 3 different algorithms: Depth-First Search, Breadth-First Search and Uniform Cost Search.

The data given to us was in a .csv file format and gave us a list of station connections, time taken between them, line, and zone information. To use this data a Pandas data frame was implemented. In this data frame names for each column of our file were created. Furthermore, zones that are labeled as [‘a’, ‘b’], ‘c’, and ‘d’ were relabelled as 7,8 and 9 in accordance to the London map tube. The structure of the resulting data frame is shown below in Fig.1.

	Start	Next	Line	Distance	Zone	Secondary Zone
0	Harrow & Wealdstone	Kenton	Bakerloo	3	5	0
1	Kenton	South Kenton	Bakerloo	2	4	0
2	South Kenton	North Wembley	Bakerloo	2	4	0
3	North Wembley	Wembley Central	Bakerloo	2	4	0
4	Wembley Central	Stonebridge Park	Bakerloo	3	4	0
5	Stonebridge Park	Harlesden	Bakerloo	2	3	0
6	Harlesden	Willesden Junction	Bakerloo	2	3	0
7	Willesden Junction	Kensal Green	Bakerloo	3	3	0

Fig. 1. Beginning of data frame

This was used then in conjunction with the networkx framework to create a graph of nodes and edges. The graph consisted of 271 nodes and 374 edges.

The first implementation is the DFS algorithm. This algorithm works by exploring nodes that go as deep as possible in the branch before looking at different branches. To implement this in our London Underground problem the functionality of the method “my_depth_first_graph_search()” along with the construct_path_from_root function given to us in Lab 3 was extended to work for our dataset and produce the time taken for each path found. This function was named calculate_time_taken() and worked by backing up the path taken and adding each ‘time’ value from the graph attributes. The main dfs function works by keeping track of which nodes we have visited and storing nested dictionaries that we can backtrace the path from. To make

these we iterate through the neighbours of our starting node and append their connecting stations to our nested dictionary if they aren't in our explored nodes. This appending occurs on the right of the list and so it is a LIFO. When we reach the goal node we return this dictionary. Using the construct path function we can backtrace through the dictionary to return a path and using the calculate_time_taken() we also return the total time of the path. Nodes expanded are also printed out.

Next, the BFS algorithm is implemented. Similar to the DFS, the functionality of the function given to us in Lab 3 is extended to handle our dataset and the calculate_time_taken() function used to find the time taken. Unlike DFS, BFS explores all neighbours in the same depth before exploring deeper in the branch. This is done again by keeping track of all the explored nodes and creating a nested dictionary that stores the path created by children that do not appear in the explored nodes.

Lastly, the UCS algorithm was implemented. This algorithm was created by using the skeleton of an implementation of UCS in a simpler problem by Richard Ketchersid which can be found at https://cocalc.com/share/public_paths/4bc0213188739dfe3da59b3ffd6c1c82d9eff225. My implementation uses a priority queue as a frontier. Nodes in the frontier are sorted by the weight of the node to the start node. The one selected for expansion is the one with the lowest weight. The weights are created in accordance to a cost function, which in this case is simply the time between stations. The algorithm also keeps track of a back pointer that keeps track of all the node expansion that can later be used to reconstruct the path. In this implementation the number of nodes expanded and the time taken for the minimum path is displayed.

2.2 Compare DFS, BFS and UCS.

Next, the 3 algorithms were compared for a number of different routes and the reasons for their effectiveness were explored. These routes can be seen in Table 1, along with the path time and nodes expanded by each algorithm for the same start-end stations.

The results show that in general BFS and UCS outperform DFS at finding better paths. Other than the route from Canada Water to Stratford, where UCS has found a path that is faster by one minute, the BFS and UCS algorithms find the same exact optimal paths. However, where BFS succeeds over UCS is at how long it takes to find that optimal route. BFS consistently expands less nodes and finds the same optimal answer as UCS.

Start-End Stations	Path Time			Nodes Expanded		
	DFS	BFS	UCS	DFS	BFS	UCS
Canada Water-Stratford	15	15	14	7	24	54
New Cross Gate-Stepney Green	8	8	8	10	8	9
Ealing Broadway-South Kensington	57	20	20	180	46	52
Baker Street-Wembley Park	14	14	14	4	8	83
Old Street-Ruislip	70	43	43	136	224	248
Rayners Lane-Kensal Green	46	41	41	237	164	171

Table 1. Table of path time and node expansions for our 3 algorithms for a number of routes.

These results can be explained by considering how the London Tube is structured. Due to the many connections between stations there is usually an answer not far from the root of the tree (as changing lines can help you reach your goal very quickly). This means that a BFS approach would be better than a DFS as solutions are usually not located very deep in the search tree. BFS has another property that lets it perform as good as UCS in finding optimal path times. Due to it searching every edge from a node before moving in a level deeper to do the same, we can almost guarantee that it will find the shortest path (or very close to it) since if there was a shorter path it would have found it already in a previous less deep level of edges.

The times where DFS does perform better seem to be attributed to luck as it only does so when the destination is very close to the root. This can be seen in both Baker Street-Wembley Park and Canada Water-Stratford where DFS get's lucky and finds the path only with 4 and 7 expansions respectively. However its luck runs out when considering larger routes, as it not only expands more nodes but also finds suboptimal routes to the goal.

Why does BFS outperform UCS however? In most cases in our graph the 'weight' of our edges is in the range of 1-3 minutes. If all the weights are the same in a graph UCS and BFS essentially act as the same algorithm. This small range of weights means that BFS usually selects the optimal ways anyway, and it does so with expanding less nodes.

We can therefore conclude that the best algorithm to use out of the 3 is the BFS as it performs as good or better than the other in less time.

2.3 Extending the cost function

Taking only the times between stations for our cost function does not give a full representation of what a real journey would be like, as the time taken to change lines can greatly affect what the best path is. To address this, the cost function of the UCS was extended to include a "penalty" in the cost function for node stations that required a line change. This penalty began with a value of 5, which is close to how long it would take for an actual line change in the London Underground, given walking and waiting for the next train. A comparison of our new UCS function with the old is seen in Table 2 below.

Start-End Stations	Path Time		Nodes Expanded	
	UCS	UCS_improved	UCS	UCS_improved
Canada Water-Stratford	15	15	55	47
New Cross Gate-Stepney Green	8	8	9	7
Ealing Broadway-South Kensington	20	20	52	54
Baker Street-Wembley Park	13	13	83	51
Old Street-Ruislip	43	43	248	224
Rayners Lane-Kensal Green	41	41	171	164

Table 2. Table of path time and node expansions for our 2 UCS algorithms for a number of routes for penalty value = 5.

As can be seen in the table the improved UCS outperforms the normal UCS as it almost always expands less nodes and always finds the optimal paths. This can be explained by the fact that due to the great connectivity of the London Underground, most stations can be reached with very few actual line changes. This means that choosing to expand nodes that are on the same line can be beneficial as we don't 'waste' expansions looking at stations in every possible line.

Furthermore, this idea was explored further by increasing the penalty to an even greater number, not modelled by the minutes it takes in real life. This way we are telling to the algorithm that it should almost always try to stay in the same line if possible. The value for the penalty used in Table 3 is 100.

Start-End Stations	Path Time		Nodes Expanded	
	UCS	UCS_improved	UCS	UCS_improved
Canada Water-Stratford	15	15	55	49
New Cross Gate-Stepney Green	8	8	9	7
Ealing Broadway-South Kensington	20	20	52	57
Baker Street-Wembley Park	13	13	83	47
Old Street-Ruislip	43	43	248	215
Rayners Lane-Kensal Green	41	41	171	180

Table 3. Table of path time and node expansions for our 2 UCS algorithms for a number of routes for penalty value = 100.

Here we can see that at some cases the number of nodes expanded are even smaller, than before. However, for some other routes, this is not the case. This means that at times the algorithm tries to force staying in the same line where it is not optimal. This means that a balance must be struck between choosing to ignore line changes and accepting that a line change must be performed to reach the destination.

Next, we can see the optimum paths for each of our routes as found by UCS and BFS respectively. DFS often does not produce this optimal path and finds an alternative, longer time path.

1. ['Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green', 'Mile End', 'Stratford']
2. ['New Cross Gate', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green']
3. ['Ealing Broadway', 'Ealing Common', 'Acton Town', 'Turnham Green', 'Hammersmith', 'Barons Court', "Earls' Court", 'Gloucester Road', 'South Kensington']
4. ['Baker Street', 'Finchley Road', 'Wembley Park']
5. ['Old Street', 'Angel', "King's Cross St. Pancras", 'Euston Square', 'Great Portland Street', 'Baker Street', 'Finchley Road', 'Wembley Park', 'Preston Road', 'Northwick Park', 'Harrow-on-the-Hill', 'West Harrow', 'Rayners Lane', 'Eastcote', 'Ruislip Manor', 'Ruislip']

6. ['Rayners Lane', 'West Harrow', 'Harrow-on-the-Hill', 'Northwick Park', 'Preston Road', 'Wembley Park', 'Finchley Road', 'Baker Street', 'Edgware Road', 'Paddington', 'Warwick Avenue', 'Maida Vale', 'Kilburn Park', "Queen's Park", 'Kensal Green']

2.4 Heuristic Search

These algorithms discussed were all uninformed search-the algorithm doesn't have any information about the goal. Next we propose a heuristic that will help guide the search towards the goal and hopefully expand less nodes in the process. Given that we know the zone of our goal station, we can aid the search by adding a heuristic to our cost function. Since the zones are in number the heuristic function created here was a "squared difference" between zones. This means that for each node, a difference is found between the zones and then squared. So for a station at zone 6, if the goal is at zone 3 then the cost added to the cost function is (6-3)^2 = 9. This means that stations closer to the goal zone are prioritised in the priority queue as they have a smaller cost.

This Heuristic search was compared with our improved UCS in the Table 4 below. It can be seen that it mostly finds the optimal paths at an improved time. In two particular examples Baker Street-Wembley Park and Canada Water-Stratford it cuts the nodes expanded to around a third, and for the others it also decreases the amount of nodes expanded. The algorithm chooses to ignore expanding nodes that go away from the zone of the goal station due to the heuristic function. Normal UCS would expand nodes simply because they have a low cost between stations, incorrectly moving away from the goal station.

Overall the Heuristic is successful at reducing the time taken for the search to find the optimal path. This was explored further by squaring the heuristic function again, overall putting the difference to the 4th power. This resulted in even better numbers for the nodes expanded as can be seen in Table 5.

Start-End Stations	Path Time		Nodes Expanded	
	Heuristic	UCS_improved	Heuristic	UCS_improved
Canada Water-Stratford	15	15	17	47
New Cross Gate-Stepney Green	8	8	7	7
Ealing Broadway-South Kensington	25	20	51	54
Baker Street-Wembley Park	13	13	15	51
Old Street-Ruislip	43	43	178	224
Rayners Lane-Kensal Green	41	42	157	164

Table 4. Table of path time and node expansions for our squared difference heuristic algorithm against improved UCS for a number of routes.

Start-End Stations	Path Time		Nodes Expanded	
	Heuristic 2	Heuristic 4	Heuristic 2	Heuristic 4
Canada Water-Stratford	15	15	17	9
New Cross Gate-Stepney Green	8	8	7	7
Ealing Broadway-South Kensington	25	25	51	39
Baker Street-Wembley Park	13	14	15	10
Old Street-Ruislip	43	43	178	156
Rayners Lane-Kensal Green	41	42	157	139

Table 5. Table of path time and node expansions for our squared difference heuristic algorithms and to the power of 4, for a number of routes.