EECS759P Coursework 1 Report

Name: Bheki Maenetja
Student Number: 230382466

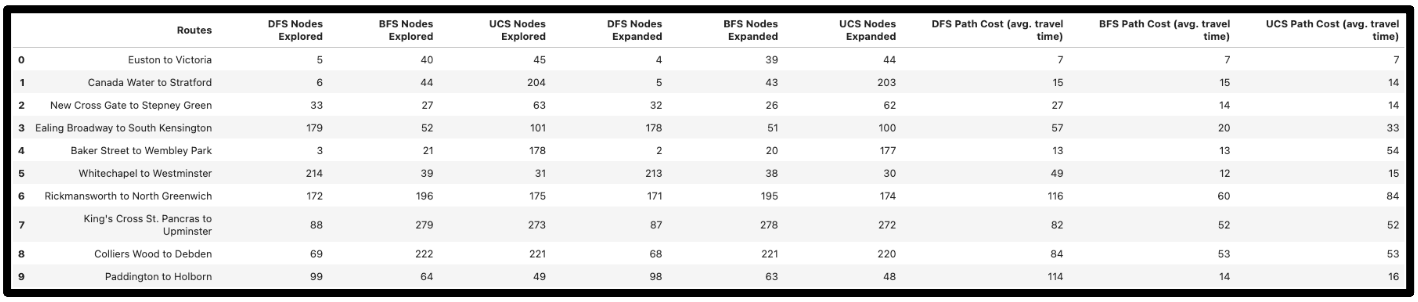
Agenda-based Search

DFS, BFS and UCS Implementation

The implementations of the DFS, BFS and UCS algorithms are modifications of the iterative version of the general search algorithm. States are represented by stations and the line on which they sit. For instance (Whitechapel, District) is a different state to (Whitechapel, Hammersmith & City). The states are encapsulated by the Node class (see code block 5 in the notebook). Each Node object has 4 properties: name (of station), line, parent and cost (representing the average travel time from the parent to the node on a given line). To run the algorithms (dfs_search, bfs_search or ucs_search) simply enter the name of the start and goal stations. Set the "print_steps" parameter to "True" to see all nodes that have been explored. To get the route found by the function, pass the goal node returned by the function to the build_path function; this function will return an ordered list of all the nodes on the path as well as the total path cost.

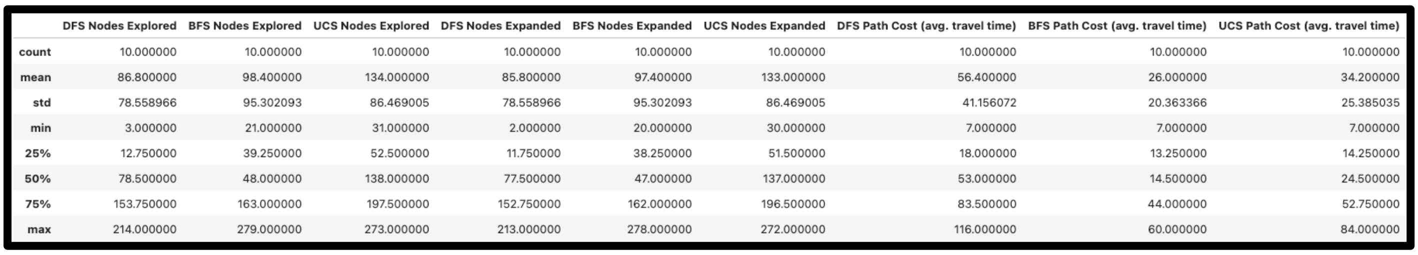
Comparison of DFS, BFS and UCS

Looking at the performance of the 3 algorithms on 10 selected routes (see tables 1 and 2 or code blocks 16 and 17 in the notebook) we observe a fairly mixed picture when it comes to their relative advantages and disadvantages. When looking at both path cost (avg. travel time) and nodes expanded/explored no algorithm is consistently better than the other 2. But, in terms of optimality, BFS and UCS tend to perform better in finding low-cost routes. DFS tends to find routes with much higher path costs; this is due to its tendency to follow one branch all the way to the end before looking at another. Although, for numerous routes, does expand/explore far fewer nodes than BFS and UCS.



	Routes	DFS Nodes Explored	BFS Nodes Explored	UCS Nodes Explored	DFS Nodes Expanded	BFS Nodes Expanded	UCS Nodes Expanded	DFS Path Cost (avg. travel time)	BFS Path Cost (avg. travel time)	UCS Path Cost (avg. travel time)
0	Euston to Victoria	5	40	45	4	39	44	7	7	7
1	Canada Water to Stratford	6	44	204	5	43	203	15	15	14
2	New Cross Gate to Stepney Green	33	27	63	32	26	62	27	14	14
3	Ealing Broadway to South Kensington	179	52	101	178	51	100	57	20	33
4	Baker Street to Wembley Park	3	21	178	2	20	177	13	13	54
5	Whitechapel to Westminster	214	39	31	213	38	30	49	12	15
6	Rickmansworth to North Greenwich	172	196	175	171	195	174	116	60	84
7	King's Cross St. Pancras to Upminster	88	279	273	87	278	272	82	52	52
8	Colliers Wood to Debden	69	222	221	68	221	220	84	53	53
9	Paddington to Holborn	99	64	49	98	63	48	114	14	16

Table 1 Performance of DFS, BFS and UCS on selected routes.



	DFS Nodes Explored	BFS Nodes Explored	UCS Nodes Explored	DFS Nodes Expanded	BFS Nodes Expanded	UCS Nodes Expanded	DFS Path Cost (avg. travel time)	BFS Path Cost (avg. travel time)	UCS Path Cost (avg. travel time)
count	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000
mean	86.800000	98.400000	134.000000	85.800000	97.400000	133.000000	56.400000	26.000000	34.200000
std	78.558966	95.302093	86.469005	78.558966	95.302093	86.469005	41.156072	20.363366	25.385035
min	3.000000	21.000000	31.000000	2.000000	20.000000	30.000000	7.000000	7.000000	7.000000
25%	12.750000	39.250000	52.500000	11.750000	38.250000	51.500000	18.000000	13.250000	14.250000
50%	78.500000	48.000000	138.000000	77.500000	47.000000	137.000000	53.000000	14.500000	24.500000
75%	153.750000	163.000000	197.500000	152.750000	162.000000	196.500000	83.500000	44.000000	52.750000
max	214.000000	279.000000	273.000000	213.000000	278.000000	272.000000	116.000000	60.000000	84.000000

Table 2 Summary statistics of DFS, BFS and UCS performance.

The Looping Problem

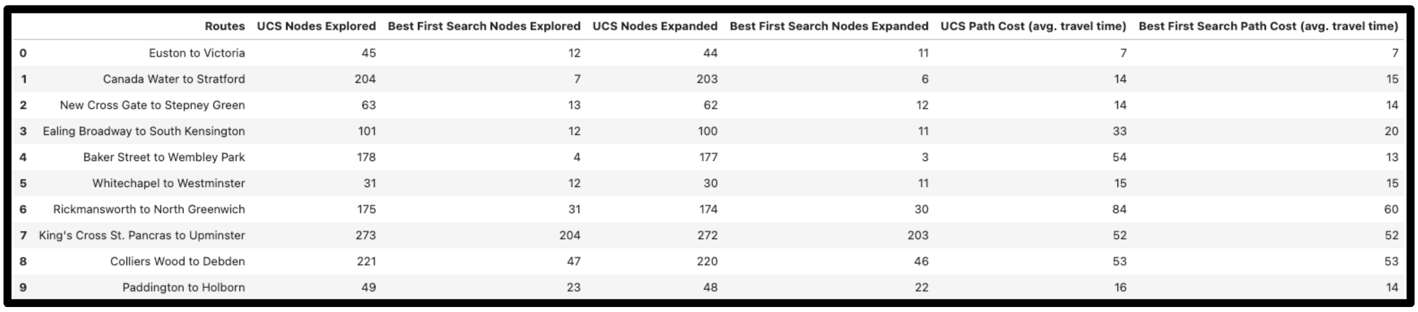
To overcome the "looping" issue the algorithms track every station that has been explored in a list. The list stores the name of every station that has been looked at – if the name of a new node is already in the list, it is not added to the frontier. Given that nodes can have the same station name (but different lines) only the names of the stations are added to the list; this is because on any given route, a station only needs to be passed through once.

Modified UCS

The modified UCS function applies a penalty of 2 minutes whenever changing lines at a station; this penalty is added to the cost of all child nodes who have a different line to their parents. When the algorithm is applied to the route “Euston to Victoria” we observe that it expands 36 nodes: far more than DFS (4) but less than BFS and the original UCS algorithm (39 and 44 respectively). However, the optimal path (Euston to Warren Street to Oxford Circus to Green Park to Victoria) has a higher path cost for this algorithm than it does for DFS, BFS and UCS. This is because the penalty is applied once when changing lines (from Northern to Victoria) at Warren Street.

Heuristic Search

The heuristic search algorithm, implemented in code block 22, is a best-first search algorithm that sorts states in the frontier based on a heuristic function. Obviously, a station that is in the same zone or a zone close to the goal station’s zone has higher utility than one that is in a zone farther away. However, even if a station is in the same zone, the goal may not be located on the lines that pass through that station. Therefore, the heuristic used by this best-first search algorithm considers the zone that a station is in as well as the line(s) that it is located on. The heuristic returns 0 if the name of the node matches the goal station. If it does not, then it is given a baseline utility of 1; added to that utility is the absolute difference between the zone of the goal and the zone of the node (all zones are mapped to numerical values) and a value indicating if the current node is on a line that passes through the goal station (0 if true, 1 otherwise).



	Routes	UCS Nodes Explored	Best First Search Nodes Explored	UCS Nodes Expanded	Best First Search Nodes Expanded	UCS Path Cost (avg. travel time)	Best First Search Path Cost (avg. travel time)
0	Euston to Victoria	45	12	44	11	7	7
1	Canada Water to Stratford	204	7	203	6	14	15
2	New Cross Gate to Stepney Green	63	13	62	12	14	14
3	Ealing Broadway to South Kensington	101	12	100	11	33	20
4	Baker Street to Wembley Park	178	4	177	3	54	13
5	Whitechapel to Westminster	31	12	30	11	15	15
6	Rickmansworth to North Greenwich	175	31	174	30	84	60
7	King's Cross St. Pancras to Upminster	273	204	272	203	52	52
8	Colliers Wood to Debden	221	47	220	46	53	53
9	Paddington to Holborn	49	23	48	22	16	14

Table 3 Performance comparison of uniform cost search and best-first search.

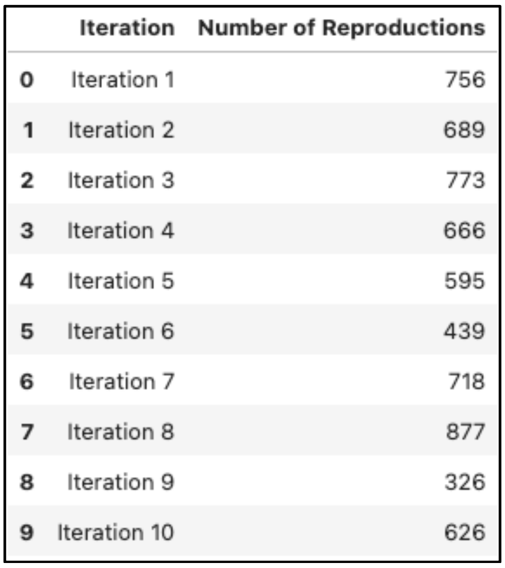
The algorithm performs quite favourably in comparison to UCS (see table 3). For each of the 10 selected routes the algorithm expands/explores far fewer nodes and, in all but 1 case (Canada Water to Stratford), finds a path with a cost that is lower or equal to the cost of the path found by UCS.

Adversarial Search

Genetic Algorithm Implementation

In the implementation of the genetic algorithm (see code block 28 in the notebook), populations are represented as lists of strings with each individual being a 10-character string consisting of randomly chosen capital letters, numbers or underscores. Selection (both mating and replacement) is done by selecting the highest ranked 50% of individuals in the population – this includes parents and offspring. The algorithm makes use of 1-point crossover to splice chromosomes and uncorrelated local mutations to create new individuals. The function has a default crossover and mutation rate of 0.5 and uses a population size of 1000. It also has a parameter called “n_gen” which specifies the maximum number of reproductions that can be done before the program terminates.

Number of Reproductions



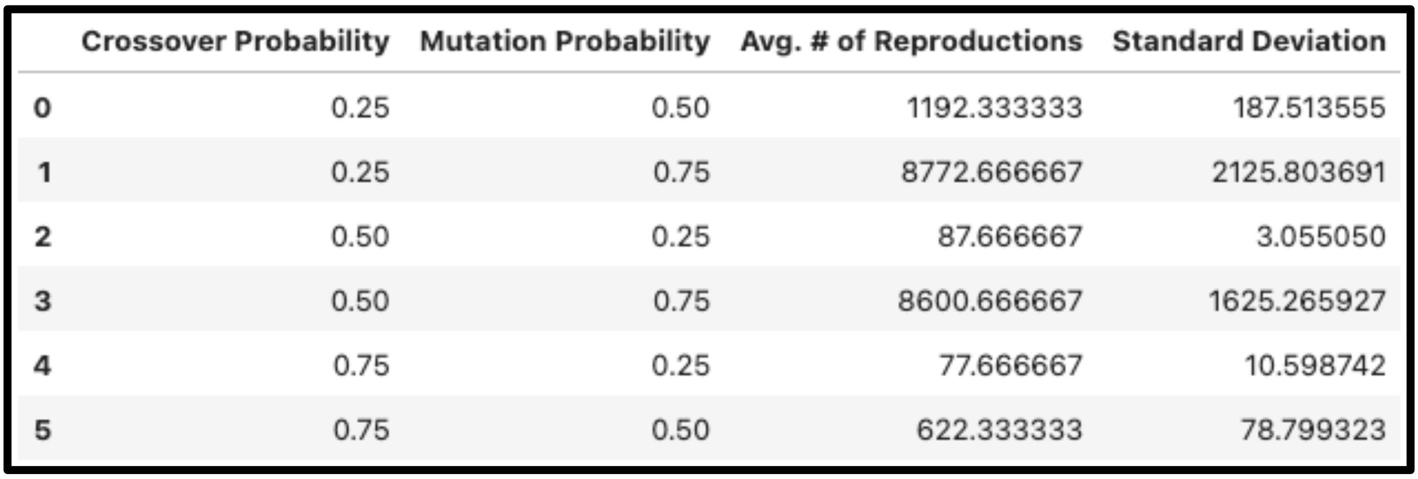
	Iteration	Number of Reproductions
0	Iteration 1	756
1	Iteration 2	689
2	Iteration 3	773
3	Iteration 4	666
4	Iteration 5	595
5	Iteration 6	439
6	Iteration 7	718
7	Iteration 8	877
8	Iteration 9	326
9	Iteration 10	626

Table 4 Number of reproductions needed for GA function to converge on solution.

Running the GA function on the default parameters (see table 4), we observe that the average number of reproductions needed for the algorithm to converge upon the solution ("ZS09_JG55Z" produced from username "ecs23496") is 647; the standard deviation of the results above is 162.4. For most iterations (see table 4) the algorithm converged after about 600 to 800 reproductions.

Effect of Hyperparameters

Looking at the effect of the crossover rate and mutation rate on the algorithm, we observe that there is great variability in the effect that these hyperparameters on the number of reproductions needed for convergence. The results (see table 5) show that lower mutation rates significantly reduce the average number of reproductions needed for convergence. The mutation rate seems to have a much greater influence on the number of reproductions than the crossover rate.



	Crossover Probability	Mutation Probability	Avg. # of Reproductions	Standard Deviation
0	0.25	0.50	1192.333333	187.513555
1	0.25	0.75	8772.666667	2125.803691
2	0.50	0.25	87.666667	3.055050
3	0.50	0.75	8600.666667	1625.265927
4	0.75	0.25	77.666667	10.598742
5	0.75	0.50	622.333333	78.799323

Table 5 Effect of selected combinations of crossover and mutation rate on the number of reproductions needed for convergence.