Artificial Intelligence – ECS759P
Coursework Submission 1

Piyush Lohokare – 230238819

IMPORTANT:
The implementation of BFS, DFS, UCS, Best First Search and the Genetic Algorithm are in the same .ipynb File. The .ipynb file can be opened using JupyterLab or Google Colab. The complete code can be run by clicking on the 'run all' option from the 'run' tab. For all the Search Algorithms to work, we will need to have the tubedata.csv file in the same directory as the .ipynb file is saved in.

2.1 Implement DFS, BFS and UCS

I'm creating 3 dictionaries. Details of these dictionaries can be found in the .ipynb file under the title "Declaring the Dictionaries"

2.2 Compare DFS, BFS and UCS:

• Is any method consistently better?

- As per my observations, some algorithms are performing better in some circumstances while others are performing better in others. Some start and goal stations are best suited for the DFS Algorithm and so we are getting a better cost and exploration for it. In some cases, the UCS works best.

• Report the returned path costs in terms of the count of the visited nodes for one route below (or any route of your choice) and explain your results based on the knowledge of costs each algorithm considers.

- Here, I'm considering the path Euston to Victoria

- The count of visited nodes for the algorithms can be viewed in the **Table 2.2a - Count of visited nodes** of the Appendix

- In some cases, the goal station will be much more easily accessible for the DFS algorithm, as is for Euston to Victoria, than the BFS algorithm. However, if we consider the stations Ealing Broadway to South Kensington, DFS does 179 explorations compared to 50 explorations of BFS. So, depending on how the data is entered in the dictionary, in some cases, DFS will give better results than BFS and vice-versa in other cases
- The number of explorations for UCS is approximately an average of the number of explorations needed for BFS and DFS.

• Report the returned path costs in terms of the average time taken for one route below (or any route of your choice) and explain your results based on the knowledge of costs each algorithm considers.

- Here, I'm considering the path Euston to Victoria

- The path costs of visited nodes for the algorithms can be viewed in the **Table 2.2b - Path costs** of the Appendix

- DFS and BFS do not consider the cost of travel from Euston to Victoria. However, we can observe that BFS performs better in terms of the path costs than DFS.
- As BFS is a variation of UCS, we can observe that the path costs of both these algorithms are same or in a range of ± 1.

• Report the returned path costs in terms of the visited nodes and the average time taken for one route below (or any route of your choice) for two different orders to process the nodes (direct and inverse order of explored nodes at each level) and explain your results for the three algorithms.

- The path costs and number of nodes explored for the BFS and DFS algorithms can be viewed in the **Table 2.2c - Normal and reversed path costs** of the Appendix
- As observed, the DFS with nodes considered in order performs better than the DFS with reversed neighbors in terms of the number of nodes explored and the travel cost.
- Inversely, the BFS with reversed neighbors performs better than the BFS with nodes considers in order in terms of the number of nodes explored and the travel cost.
- Reversing the neighbors will have no effect on UCS as we consider the nodes with the lowest costs first.

• Explain how you overcame the loop issue in your code.

- The looping issue where we generate the neighbors of the current station (say Station A) and add them to the list occurs when we select the first neighbor (say Station B) of the current station and generate its neighbors. In this newly generated list of neighbors, the previous station (Station A) is also included.

- To overcome this, I have used the condition from the Lab 2 solutions of BFS and DFS where we are adding each explored node to the explored_nodes set. I'm checking for each station in the neighbors if it is in the explored_nodes list. If the station is in the explored_nodes list, we are not appending it to the LIFO_list , FIFO_list, UCS_list dictionary in each algorithm. If the station is not in the explored_nodes list, we are appending it to the LIFO_list , FIFO_list, UCS_list dictionary in each algorithm.

2.3 Extending the cost function

- The BFS and DFS are not affected by the implementation of line change function as DFS and BFS do not consider the cost of travel.
- However, if we compare the UCS without line change implementation and UCS with line change implementation, we can see that the path costs for the UCS with line change implementation is more than the UCS without line change implementation, as is expected.

Here, I'm considering the path Euston to Victoria:

- The comparison can be viewed in the **Table 2.3 - UCS without Line Change and with Line Change Implementation** in the Appendix

- If there is a line change, we are adding 2 to the total path cost. So, as observed, we are changing lines while coming from Euston to Victoria once. As a result, the cost for line change was added to the path cost and the final path cost became 9 instead of 7.
I believe that the line change implementation will change depending on how the data is added in the station_dict. If there are two lines from the current station to the neighbor with the same cost, the algorithm will pick the line that was added first.

2.4 Heuristic Search:

- I'm creating a list of all the zones: list_heu = ['1', '2', '3', '4', '5', '6', 'a', 'b', 'c', 'd']. The other details are mentioned in the second code block under the title **"Declaring the Dictionaries"** in the .ipynb file
- The details about the zone usage to find the heuristic are explained in the .ipynb file under the title **"Best First Search"**

- For the calculation of the heuristic of the current node and returning it, please refer to the code cell right below the title **"BEST FIRST SEARCH"**

My motivation to use this heuristic:

- Details of these dictionaries can be found in the file Piyush Lohokare AI CW1 230238819.ipynb under the title "*My motivation to use this heuristic*"

Here, I'm considering the path Euston to Victoria:
- The comparison can be viewed in the **Table 2.4 - Comparison of UCS with Line Change and Best First Search** in the Appendix
- As can be seen from the above example, the Best First Search Implementation is performing better than the UCS with Line change Implementation by comparing their Number of Explorations.

2.5 Extra: Peer Review: Zone-based heuristic which assigns a constant time of 10 minutes for travel within one zone and a constant time of 20 minutes for travel across zones.

- As we are considering only the zone-based heuristic which assigns a constant time of 10 minutes for travel within one zone, there is a strong possibility that the algorithm will travel within the same zone as much as possible to reach the goal, even if the travel time and cost is more than what it could have been if it followed the actual shortest path.
- As this is a Zone-based heuristic and not station-based, the path to reach the goal might be shorter than the station-based

3.1 Implement a Genetic Algorithm:

- True password: _5OW0EHQYX
- Password Found: Password generated: ('_5OW0EHQYX', 1.0)

3.2 Genetic Algorithm components - Describe the chosen state representation and the methods you chose to perform selection, crossover and mutation.

- Chosen State Representation: The state representation in this case is the length of the password, which is 10, and the list of all permissible characters for the password, which are all uppercase Alphabets, the numbers 0 – 9 and the underscore symbol '_'.
- Selection:

The selection procedure has been implemented and explained in the code block for **"def gen_offsprings(pool, pMu):"** as well as in the **"def gen_algo()"** line number 39 where I am selecting the population with the highest fitness of half the size of the original population and using it as the mating pool.

Crossover:
The crossover procedure has been implemented and explained in the code block for **"def crossover(p1, p2):"**

Mutation:
The mutation procedure has been implemented and explained in the code block for **"def mutation(o1, o2, mr):"**

3.3 Genetic Algorithm Number of reproductions

- The number of reproductions I needed to converge to the password differed every time I ran the code. But, generally, the number of needed reproductions was between 13 Generations to 20 Generations. For this iteration, the number of generations I got was 19:

- After running the code 10 times, the average and standard deviation I received is 15.8 and 1.5362291495737217 respectively. But this will change each time the algorithm is run.

- The full output of this iteration can be found in the following link: https://privatebin.support-tools.com/?07198a3d6250e23a#S00Jo2Iq5c5PEgK2R+VqmcEcUGKAoQlTXFBYxizE1D0=

3.4 Hyperparameters

- If I change the population size to a very low value of 200 and the generations to 400, I get the correct password very few times and the other times, the loop runs for the generation size and ends without giving the correct password. This happens because the algorithm does not get enough population elements to work on and as a result, the fitness value plateaus.

- The comparison table for the original, changed population size and changed generation size can be found in **Table 3.4 - Hyperparameter change comparison**. When I take the population size as 200, the iterations run for the "Number of Generations (1000)" and the algorithm stops. When I changed the Number of Generations to 400 and reverted the change to the population size (1000), the passwords matched for each iteration. Since I am not passing the population size and generation size as parameters to the functions, I have commented on the hyperparameter testing changes in the code title **"Declare Hyperparameters"**. If the Number of Generations is less (400) and the population size is 1000, the passwords are matching but the standard deviation is increasing to almost double the amount.

The parameters that worked for me can be viewed in the code block below the title **"Declare Hyperparameters"** in the .ipynb file.

The above parameters worked for me, as when we increase the population size and generation size, the algorithm gets more population to do the selection, crossover and mutation on. Due to this diversity, the probability of finding the correct password increases as well.

APPENDIX:

List of Tables.

Table 2.2a - Count of visited nodes

	DFS	BFS	UCS without line change	UCS with line change	Best First Search
Count of visited nodes - Euston to Victoria	25	35	30	30	21

Table 2.2b - Path Costs

	DFS	BFS	UCS without line change	UCS with line change	Best First Search
Path Costs - Euston to Victoria	13	7	7	9	7

Table 2.2c - Normal and reversed path costs

	Count of visited nodes - Euston to Victoria - Regular	Count of visited nodes - Euston to Victoria – Reversed Neighbors	Path Costs - Euston to Victoria - Regular	Path Costs - Euston to Victoria – Reversed Neighbors
DFS	25	86	13	49
BFS	35	27	7	7

Table 2.3 - UCS without Line Change and with Line Change Implementation

	Start Station	End Station	Number of Explorations	Path taken	Cost of Travel	Lines Used
UCS without Line change Implementation	Euston	Victoria	30	['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']	7	NA
UCS with Line change Implementation			30	['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']	9	{'Northern', 'Victoria'}

Table 2.4 - Comparison of UCS with Line Change and Best First Search

	Start Station	End Station	Number of explorations	Path taken	Cost of Travel	Lines Used
UCS with Line Change	Euston	Victoria	30	['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']	9	{'Northern', 'Victoria'}
Best First Search			21	['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']	7	{'Northern', 'Victoria'}

Table 3.4 - Hyperparameter change comparison

	Population size: 1000 Generation size: 1000	Population size: 200 Generation size: 1000	Population size: 1000 Generation size: 400
Average	15.8	NA	16.0
Standard Deviation	1.5362291495737217	NA	2.23606797749979

REFERENCES

.ipynb file Code Block	Reference
SEARCH ALGORITHMS – BFS, DFS, UCS, BEST FIRST SEARCH	
3	COURSEWORK 1 UNDIRECTED_MAP.PY FILE
4	COURSEWORK 1 UNDIRECTED_MAP.PY FILE
5	LAB 2 SOLUTION FROM THE FUNCTION "def construct_path_from_root(node, root):" UNDER THE HEADING Solution DFS -- variation 1
6	LAB 2 SOLUTION FROM THE FUNCTION "def my_depth_first_graph_search" AND ADDED A FEW OF MY OWN LINES OF CODE LIKE THE WAY IN WHICH I AM CALCULATING THE NEIGHBORS.
7	LAB 2 SOLUTION FROM THE FUNCTION "def my_breadth_first_graph_search" AND ADDED A FEW OF MY OWN LINES OF CODE LIKE THE WAY IN WHICH I AM CALCULATING THE NEIGHBORS.
11	LAB 2 SOLUTION FROM THE FUNCTION "def construct_path_from_root(node, root):" UNDER THE HEADING Solution DFS -- variation 1 and tweaked it as needed
GENETIC ALGORITHM	
15	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
16	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
17	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
18	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
19	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
20	COURSEWORK 1 PASSWORD_FITNESS.PY FILE
-	YouTube video reference to understand the Genetic Algorithm working (First 4 videos of the playlist) - https://www.youtube.com/playlist?list=PLRqwX-V7Uu6bJM3VgzjNV5YxVxUwzALHV