A.I. Submission 1

By: Gargeya Sharma (220278025) MSc. Artificial Intelligence

2. Agenda-based Search

2.1 Implement BFS, DFS and USC

To run the implementation, you need to just run the python cells in the notebook in the given ordered sequence and that is pretty much it.

A state in my algorithm is defined as a dictionary with keys:

(<current node name>, <parent node>, <actual cost (in average time) to reach from current to parent node>)

This formulation helps in keeping track of the path being followed while performing the search. When the solution is found, the parent node and cost can be used to traverse back from the goal node to the start node and generate the sequence of nodes in the solution as well the cost incurred while following that solution.

2.2 Compare the BFS, DFS and USC

- If I must decide which one is consistently better than I will select Breadth First Search. It's not the best with all the measures every time but relatively from DFS and USC, the number of explored nodes and Average time taken to travel with the solution path is lower.
- The path I will discuss on is (Canada Water to Stratford). So, in terms of the counts for the visited node for this route:
    - BFS: 30
    - DFS: 159
    - USC: 201

  These results can be comprehended with the simple logic of how these algorithms work. In the case of BFS (Breadth First Search), we got the solution after 30 explorations because the goal state is looked for level by level at each depth. Every node at a certain depth is exhausted before moving deeper in the tree. Because we have branches in a search tree, exploration cost is defined as branching factor (b) to the power depth (d): $\mathbf{b^d}$

  So, increasing the depth (d) is causing an exponential increase with the number of nodes we have to traverse to find the goal, as can be observed with the number of nodes DFS has to explore: 159. **Note:** The location of goal state (formation of search tree) and sequence of exploration at each depth (direct or reversed) is crucial to change the efficiency of the algorithms. In our cases, the solution was rather not too deep in the search space hence, BFS found it easier than DFS to find the goal state and formulate a solution.

  With USC, it comes down to trying to minimize the objective function while performing the search. BFS is a special case of USC, as for BFS the objective function is to minimize the level of depth while finding the goal. Sometimes, we can find the solution with lower number of exploration than DFS and somewhat like BFS **OR** sometimes we can perform worse than both of them (in this particular case).

- The path I will discuss on is (Canada Water to Stratford). So, in terms of the average time taken for this route:

- BFS: 15
- DFS: 121
- USC: 14

In this case, the objective is different. Now, USC will give us the best result because it is programmed to find us a solution with that. Note: It is most definitely not necessary to have only a single solution path, so while DFS was exploring every branch till its' end, it found a solution while is rather large and completely un-optimal. So, the average time taken along with number of explored nodes both are high due to finding a larger route to reach to the goal. As mentioned above, It depends on case to case that the goal was not located down in the larger depth of the search tree hence, BFS was able to find the goal with the smaller average time taken.

- Path chosen is same as above: (Canada Water to Stratford)

BFS:

	Number of Nodes Visited	Average Time Taken
Direct	30	15
Reversed	36	15

DFS:

	Number of Nodes Visited	Average Time Taken
Direct	159	121
Reversed	224	20

USC:

	Number of Nodes Visited	Average Time Taken
Direct	201	14
Reversed	201	14

Changing the order for the child selection for the exploration of the goal is very influential for the DFS algorithm because just as you change the child, the entire sequence of the exploration will change. This has equal chances of either improving the outputs or worsening it. Usually, it works well to see both solution (direct and reversed) for DFS because one of them is definitely better than the other.

With BFS, it really doesn't matter that much if we reverse the list of children we get on each layer because it will eventually travel all of them irrespective of their sequence. So, the goal will be found sooner or later on that same path. That's why in our case, there was a slight increase in the number of explored nodes with reversed sequence, but the average time taken was the same because the solution path is the same in both the scenarios.

With USC, there is no bother even if we change the child exploration sequence at each level because the sorted function at the end of each iteration will give priority to nodes having lowest cost, so reverse operation is overruled by sorting function.

- There was NO loop issue in my code, as I am keeping a track for all the nodes traversed and every required value is systematically designed to be outputted robustly.

2.3 Extending the Cost Function

After Improving the cost function in DFS, BFS and USC by adding 2 in the cost, every time the Tube Line is changed (as the measure to track the amount of time to switch platforms). Path taken here:

(Euston to Victoria)

BFS:

	Number of Nodes Visited	Average Time Taken
Base	40	7
Updated	40	13

DFS:

	Number of Nodes Visited	Average Time Taken
Base	124	23
Updated	124	27

USC:

	Number of Nodes Visited	Average Time Taken
Base	37	7
Updated	53	18

Changing the cost function completely changed the mechanism of how USC will approach it. Trying to minimize a different kind of cost, now, will result in completely different solution than the base algorithm (before cost function updating).

Both BFS and DFS has nothing to do with how the cost function evolved in this case because their mechanism has no reliance on the cost function. Hence, their solution paths as well as the number of explored nodes is exactly the same as the non-updated one, only cost changed because whenever tube line was changed in the path of their solution, it added 2 in the final cost.

2.4 Heuristic Search

(Same as what I mentioned in my Heuristic Peer Review Submission)

Motivation:

When I opened the tube map attached in the document, I could clearly see how each station was assigned a region, termed as zones. Number of zones expands from centre to out. This gives me an inspiration to measure proximity of stations based on zones assigned to them.

Heuristic Function:

The first thing that I did was create a zone mapper, which means I converted each zone value from string to integer. There were few values as (a, b, c, d) in zones, so I mapped them as (7, 8, 9, 10) respectively. Using the *'zone_dict'* default dictionary created by the code snippet given by the professor, I created a *'heuristic_preprocessing(zone_dict)'* function that does all the groundwork for me. This function returns another dictionary *'zones_dict'* which will be used as reference data to calculate the heuristic between the parent and child node passed in as arguments.

Once I am done with getting values that I can mathematically operate on, I treated stations with a couple of zones as those stations whose zone is a mean of the values of their zones. The reason for going with this approach is to encode the fact that stations with 2 zone values are in neither of the zones completely. So, while they are somehow in between the regions of those two zones, their mean would depict the exact same thing in terms of distance between stations in a solution.

Another reason to take the mean of those values is to increase the range of outputs I will receive from the heuristic function. Now, it's not just integer values in terms of difference between the zones of starting stations and ending stations, but float values like 1.5,2.5,3.5 and so on. This increases the intricacy for the algorithm to give priority to the lowest difference (mentioned above).

While exploring nodes inside the 'while' loop, the frontier is being sorted at the end of the loop with respect to heuristic value provided as one of the features for the state representation while running best first search.

Summary:

The Main Idea is to explore those stations (nodes) first whose difference in zonal value is the least. Having the least difference ('0' in our problem due to possibility of having same zones) will search for nodes within the station's proximity inferred by heuristic function. That way, we can minimize the average time to reach from starting station to end station.

Comparison of Best-First Search with Uniform Cost Search

Path: (Euston to Victoria)

	Number of Explored Nodes	Average Time Taken
Best First Search	20	7
Uniform Cost Search	37	7

If you notice in the notebook, you will see that both the algorithms return the same solution path, but best-first search not only performed same if not worse in getting average time taken but also took 17 less exploration steps to reach the solution with respect to uniform cost search approach.

3. Genetic Algorithm

3.1 Implement a Genetic Algorithm

Answer string of 'ec22146': **WS99AXBQN1**

3.2 Elements of the Algorithms

- State encoding:
    - **FitnessMax** function that will create the objective to maximize my evaluation output
    - **Individual** is the list that will carry a single entity of my population comprising of certain genes (allowed values to construct an individual)
    - **Evaluator** function which encompasses the utility of given closeness function to check the fitness of an individual with final goal value (in my case: **WS99AXBQN1**).

    - **Get_value** function is the utility (named 'fill_value' inside the toolbox in the notebook) which I created to return a single chromosome that will be used to form an individual at random.
    - **Population** is the list with certain size that comprises of multiple unique individuals.
- Selection: I am using tools.selTournament with tournsize = 5
- Crossover: I am using tools.cxTwoPoint
- Mutation: Created my own mutation function to randomly mutate any chromosome inside an individual with a certain mutation probability. The function is names: mutation which takes an individual as a parameter and randomly changes the chromosome as mentioned earlier with certain (ideally) small probability. **NOTE:** Keeping the probability high would result in overpowering the gradual process of evolution and would result in unmonitored randomization that would take us nowhere near the solution. (It like how keeping a high learning rate for a neural network can bounce you off the minima and just randomly skips around the loss plane with no sense of direction)

3.3 The number of reproductions

It took me 100 generations (number of reproductions) to reach to the optimal solutions.

Ideally, I ran the code 5 times to encompass the uncertainty factor in genetic algorithms due to randomization. Additionally, to decide on which number of reproductions would give me the most consistent answer, I operated an experiment on multiple 'ngens' values (50,60,80,100). Here are how their mean and variance values during multiple runs looks like:

Number of Reproductions	Mean	Variance
50	0.98833	0.00011
60	0.99416	5.106382978723428e-05
80	0.99416	5.106382978723428e-05
100	1.0	0.0

Clearly, number of reproductions as **100** gives me the most consistent optimal fitness value with 0.0 variance.

3.4 Hyper-Parameters

I experimented with size of the population and size of the selection tournament. These two values are few of the many hyperparameters involved in this algorithm. Experimentation helps me get the values that are neither too large that they affect the performance of my algorithm nor too low to affect the efficiency of the output.

Size of the population will affect the capacity to hold different variations of individuals in a single go and evaluate my fitness score onto them to then further select the top-n performers out of them.

Size of the selection tournament will affect how many of those top performing individuals (mentioned above) will I choose from the population based on the fitness evaluation. Then take their chromosomes and perform mating on them to produce efficient offsprings and improve the quality of overall population.

My optimal values for them based on experimentation came out to be:

tournsize: 5 (selection hyperparameter),

n=300 (size of the population)

Here are some graphs to show experimentations on these values: