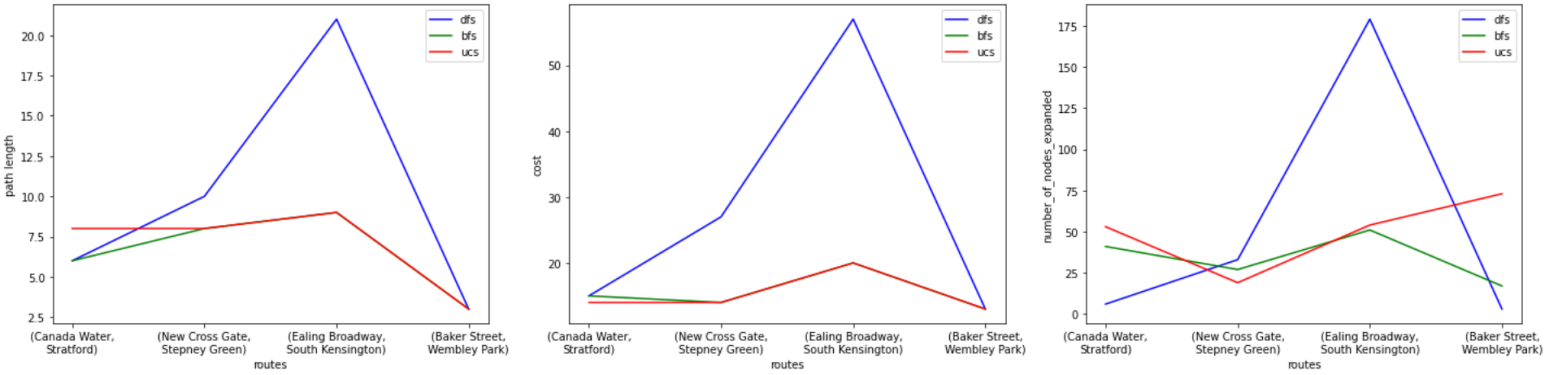
2.1 **Briefly describe how to launch the program and mention its main files in your report. Describe the state representation.**

There are two notebooks (Agenda-based search.ipynb and Genetic Algorithm.ipynb) for each part of the task. To run them, just run all cells in Jupyter Notebook.

To represent the state, I defined a class named State. This class represents the current state that we are in. So, by creating each instance we save our current station, line, path so far, and cost so far. The zone attribute of each state is initialized from zone_dict in the constructor using the current_station name which is passed to it as input. I implemented the get_neighbours function in the class which finds the children of our current state and calculates the path and cost from the current station to each child. After that, it creates a state object from each child and appends it to the current state children list. In the end, it returns the current_state children list.

2.2 **Make a detailed comparison of the three search methods. • Is any method consistently better?**



No, it depends on the route. For example, if our goal is in the depth of the graph DFS will find the answer faster but it does not guarantee the optimal solution. Otherwise, BFS and UCS can work better to find the goal in shallower levels of the graph. BFS guarantees optimal solution in terms of path length (it checks every level of the graph before exploring the next level) but it can have a big cost while UCS tries to be more efficient but it may explore more nodes. UCS also guarantees the optimal solution in terms of path cost.

As shown in the graph, we can see that none of the 3 methods is consistently better in terms of cost, nodes explored, and path length and it depends on the route. [In our tested routes, UCS is only better in terms of cost in average time taken compared to other methods, Since it always guarantees the optimal solution in terms of cost in average time taken which makes sense].

• **Report the returned path costs in terms of the count of the visited nodes for one route below**

Route Baker Street to Wembley Park

Method, path, the cost in the average time taken, the cost in the number of explored nodes

DFS:(['Baker Street', 'Finchley Road', 'Wembley Park'], 13, 3)

BFS:(['Baker Street', 'Finchley Road', 'Wembley Park'], 13, 17)

UCS:(['Baker Street', 'Finchley Road', 'Wembley Park'], 13, 73)

For this route, all 3 search methods, find the same path with the same cost in terms of the average time. The difference is in the number of the explored nodes until finding the solution. DFS has the lowest number of explored nodes which means that the goal is in the depth of the graph so it finds it faster (but does not guarantee the optimal solution) than BFS and UCS. BFS checks all nodes in the same depth at each level before exploring the next level and it guarantees the optimal solution in terms of path length. UCS is like BFS but it finds the optimal path (in terms of the cost in average time taken) according to the cost of exploring each neighbor. That is why it explores more nodes compared to BFS to find the optimal solution in this case.

• **Report the returned path costs in terms of the average time taken for one route below (or any route of your choice)**

Route Canada Water to Stratford

Method, path, the cost in the average time taken, the cost in the number of explored nodes

DFS: (['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 6)

BFS:(['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 41)

UCS:(['Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green', 'Mile End', 'Stratford'], 14, 53)

In this case, UCS finds the best path in terms of the cost in the average time taken but it explores more nodes (since it wants to find the optimal solution in terms of the average time taken).BFS and DFS find the same path with the same cost but DFS explores fewer nodes since the goal was in the depth of the graph and BFS first looks at neighbors in the same level in each step before going to the next level of the graph. So that is why BFS explores more nodes.

• **Report the returned path costs in terms of the visited nodes and the average time taken for one route for two different orders to process**

NORMAL ORDER

DFS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford', 'Mile End', 'Stepney Green'], 27, 33)

BFS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green'], 14, 27)

UCS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green'], 14, 19)
REVERSED ORDER
DFS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green'], 14, 268)
BFS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green'], 14, 40)
UCS:(['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green'], 14, 19)
We can see that in this route order of exploring children changes the DFS path. Meanwhile, the path of UCS and BFS don't change as expected and they find the optimal solution (since they guarantee to find the optimal solution (BFS in terms of path length and UCS in terms of path cost) and the order of checking children does not matter). The number of explored nodes changed in BFS since the intended nodes are closer to the top of the queue in the normal order in this case. DFS in reversed order finds the optimal solution (no guarantee) by exploring too many nodes compared to its answer in the normal order (which was not optimal). This is because the goal is in shallower levels of the graph so BFS and UCS find it with fewer explored nodes.

- **Explain how you overcame the loop issue in your code**

By implementing the explored nodes set, I always check if the child node has been explored before or not, and if it is I ignore it and check the next node in the neighbors list. Otherwise, I add it into both the explored set and queue.

2.3 **give an example in your report of how this new cost has changed the paths returned.**

Extended UCS Result:(['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 40)
DFS:(['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 6)
BFS: (['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 41)
UCS: (['Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green', 'Mile End', 'Stratford'], 14, 53)
We can see that adding the line change cost to the UCS algorithm has changed the path that it returns. It means that in this new cost calculation, the old UCS path has a bigger cost value so that is why the extended UCS has picked another path as the optimal solution (with fewer explored nodes). We see that in extended ucs we explore fewer nodes than the normal ucs in this case but ucs still has the best solution cost in terms of average time taken. BFS and DFS paths are also the same as the extended UCS path but DFS explored fewer nodes to find the same path (means that the goal is in the depth of the graph).

2.4 **Explain in the report the motivation for your heuristic and its formula**

bfs_with_heuristic:(['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford'], 15, 82)
UCS:(['Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green', 'Mile End', 'Stratford'], 14, 53)
It means that in this new cost calculation(only considering heuristic value), the old UCS path has a bigger heuristic value so that is why the bfs_with_heuristic has picked another path as the optimal solution in terms of less heuristic value. bfs_with_heuristic solution is shorter but it has greater cost and explored nodes compared to ucs.
For each current state, I calculate the heuristic between its zones and its children's zones:
Since the zone set has a maximum of 2 elements in our problem:
If parent and child zone sets are exactly the same, I return 0 heuristic. Otherwise, I find the different zone number between parent and child zone sets (different element). I calculate the absolute difference between common and different elements in sets multiply it by 5 and return it as a heuristic. So, it means that if two sets have an intersection but they are not exactly the same, it means that one of them has another zone. I calculate the difference between this new zone and the common zone to know how far is the new zone from the common zone (and multiply it by a coefficient to show its magnitude more which is not necessary) and return it as a heuristic. In this method, if the non-common zone number is farther the heuristic is bigger. If two sets don't have any common elements (which does not happen in our data because there is always a common zone number between parent and child due to the initial code) it returns 100 as a heuristic. So, my heuristic encourages the algorithm to pick the node with a closer zone number. for example: it picks {1} -> {1, 3} rather than {1} -> {1, 5} since zone 5 is farther than zone 3.

2.5 Extra: Peer Review

The zone-based heuristic: advantages: 1-It is so simple to code 2- since it assigns the same value for traveling across zones it is easy to compute and it needs fewer computation resources (the algorithm computation time is small)
disadvantages: 1- since near zones have close numbers to each other, when this heuristic assigns the same constant to every different zone it ignores how far zones are from each other so it is not that accurate 2- it has many answers with the same heuristic for every route which makes it less precise to use

3.1 **Report the true password.**

Password = RBPI951WUT

3.2 **Describe the chosen state representation and the methods you chose to perform selection, crossover, and mutation.**

There is a show_results and show_alerts option for each function which is set to False by default in order to reduce the size of the notebook.

State representation: I defined a class named Genome to represent each gene: This class has an attribute named password which is a string. In its constructor: you should pass the length of the password (in our case 10) to generate a password from accepted characters (using the generate_char function 10 times to pick a random choice from accepted characters). There is an option child_pass which is

set to none by default. This option is for debugging and for the cases where you want to generate a child genome when you know the password (for example in a crossover you generate passwords from parents and create children genomes from crossover passwords)
Class Methods:
Generate_char: pick a random choice from accepted characters and return it.
Mutation: for each character of the password we generate a random number and if this number is less or equal to the mutation probability, we generate a random character using the generate_char function and replace it in the password string. Also, if the mutation function does not change the password we print a warning for debugging.
__str__: for printing the object we use this method to show the password when the print function is called on the object of class Genome.
Crossover: For crossover, I have 2 nested functions **crossover** and **crossing_over**: In the **crossover** function, I generate 2 new Genomes from the given genomes a and b. First I check if the length of the passwords of parents are the same and if they are more than 2 chars which is not necessary in our case since all passwords have 10 chars by default so it is only used for debugging.
Then I generate a random number to find a position in the string to separate each genome password into 2 parts. I combine the first part of genome_a with the second part of genome_b and vice versa to create two children by crossover. After that, I check if the set of parents and children has common elements which means that crossover creates the exact same child as their parents and prints a warning message. In the end, I create two genome objects from crossover passwords and return them. In the **crossing_over** function, I iterate through the pool_to_cross list, and for each genome in the list, I generate a random number. If the random number is less or equal to the crossover probability I create a copy of the pool (others) and remove the current genome from it. Then I randomly pick the second genome for crossover from the others list and save its index. I call the crossover function to generate children then I check if the children have the same password as the other elements of the pool. If children are unique I replace them in the pool in the index of their parents. Otherwise, the crossover operation is refused since children are not unique.
Selection: we select half of the best parents and their children (children are generated by parents using crossover and mutation) for the next-generation population and we sort the population in descending order by calculating their fitness values. We repeat this process until we find the password (fitness of best genome (Genome in the 0 index of sorted list) =1).
To find the best parents we should sort the population by fitness function: we find the fitness of each member of the population and create a combined list (genome, its fitness). Then we sort the combined list in descending order so the fittest genome is in the index 0 and we return the sorted population.

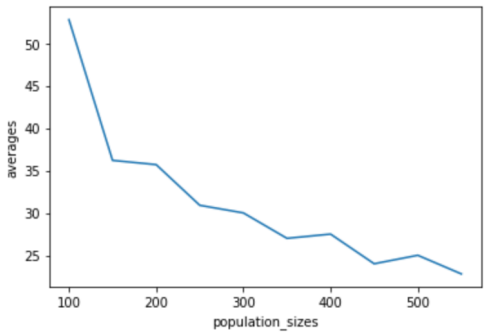
3.3 **Report the number of reproductions you had to perform to converge to the true password.**

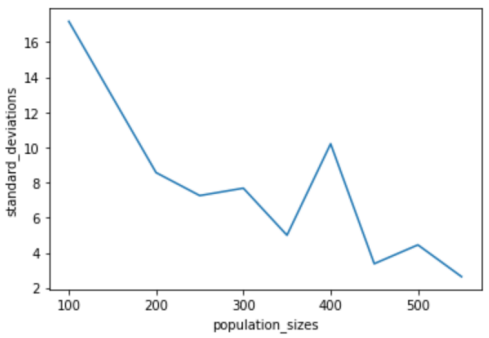
Population_size = 100, fitness_limit: int = 1, generation_limit: int = 400, pMuta = 0.05, pCO = 0.8

```
reproductions in 10 times running [79, 81, 45, 81, 57, 89, 80, 53, 57, 118]
average 74.0 standard_deviation 20.445048300260872
```

3.4 **Explore the effect of at least one hyperparameter on the performance of your algorithm.**

Size	Average	Standard Deviation
100	52.8	17.186040847152668
150	36.2	12.874781551544865
200	35.7	8.567963585356791
250	30.9	7.258787777583803
300	30.0	7.681145747868608
350	27.0	5.0
400	27.5	10.21028892833107
450	24.0	3.3763886032268267
500	25.0	4.449719092257398
550	22.8	2.638181191654584





Although there are some fluctuations in values, we can see that eventually, the average and standard deviation values of the number of reproductions decrease as the size of the population increases. This is because when we have a bigger population size (more variety), we find the password in less number of new generations. Also, since we have a bigger population size there may be higher chances to converge to the better solutions (higher fitnesses). Since the number of standard deviations also decreases as population size increases, it means that our results become more consistent and reliable.
So according to the table population size 550 has the lowest average number of reproductions to find the password. In our range of the population size, we can see that increasing population size by more **than 250-300** does not significantly change the average number of reproductions so maybe population sizes between 250-300 is our optimal value for this hyperparameter or it is near to it. Also, we can see the values of the standard deviation after this population size do not significantly change (compared to its changes from size 100 to 250).