ECS759P: *Artificial Intelligence*
Assignment 1: REPORT

Submission details

- **Course name**: Artificial intelligence
- **Course code**: ECS759P
- **Student name**: Pranav Narendra Gopalkrishna
- **Student number**: 231052045

1. Introduction

1.1. Obtaining the dataset

Here are the first few rows of the dataset

	0	1	2	3	4	5
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

1.2. Data representation for the search algorithms

We create the graph from the given data as the dictionary `station_dict`. We shall describe it in more detail in section 2.1.

`station_dict`:

```
Harrow & Wealdstone: [('Kenton', 3, 'Bakerloo', 5, 5)]
Kenton: [('Harrow & Wealdstone', 3, 'Bakerloo', 5, 5), ('South Kenton', 2, 'Bakerloo', 4, 4)]
South Kenton: [('Kenton', 2, 'Bakerloo', 4, 4), ('North Wembley', 2, 'Bakerloo', 4, 4)]
North Wembley: [('South Kenton', 2, 'Bakerloo', 4, 4), ('Wembley Central', 2, 'Bakerloo', 4, 4)]
Wembley Central: [('North Wembley', 2, 'Bakerloo', 4, 4), ('Stonebridge Park', 3, 'Bakerloo', 4, 4)]
```

For later reference and use, we also create another dictionary `zone_dict` that relates each station name to every zone it may lie in.

`zone_dict`:

```
Harrow & Wealdstone: {'5'}
Kenton: {'4'}
South Kenton: {'4'}
North Wembley: {'4'}
Wembley Central: {'4'}
```

2. Implementing agenda-based search

An agenda in general refers to a list (ordered set) of items to be processed in a well-defined order. In the case of search, agenda refers to the list of nodes to be explored in a well-defined order; the nodes chosen to be explored as well as the order of

exploration depends on the search method. The agenda-based search methods we are looking into are:

- Breadth-first search (BFS)
  - Agenda: FIFO queue of unexplored nodes
  - Order: First-come-first-serve
- Depth-first search (DFS)
  - Agenda: LIFO queue of unexplored nodes
  - Order: Last-come-first-serve
- Uniform cost search (UCS) (extension of BFS)
  - Agenda: Queue ordered by cost
  - Order: Least cost first

2.1. Implementing BFS, DFS & UCS

State representation

As previously mentioned, we create the graph from the given data as the dictionary `station_dict` . Each key of the dictionary is a station name (*string*), indicating the starting station of a single-stop route. This key, let us call it `start` , is mapped to a list of all neighbouring stations which a tube starting from `start` can reach without traversing any other station. Each element of the list associated with the key `start` is a tuple in the following format...

```
(end, time, line, zone1, zone2)
```

... where (considering `start` as the starting station name):

- `end` is the name (*string*) of a neighbouring station
- `time` is the average time taken (*float*) to travel from `start` to `end`
- `line` is the name (*string*) of the tube line connecting `start` & `end`
- `zone1` is the zone (*integer*) of the part of `start` connecting to `end`
- `zone2` is the zone (*integer*) of the part of `end` connecting to `start`

NOTE: A station can have two zones (indicating that it lies at the junction of two zones). Zones will be discussed further in the section on zone-based heuristics.

Given the graph above, a state is a station and is represented by the string denoting the station's name. Each neighbour of the station is a neighbouring station's name found within the tuple containing other attributes related to the path between the neighbours.

2.1.a. Breadth-first search (BFS)

Result of the BFS implementation for the path: *Euston to Victoria*:

```
path: ['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']
pathCost: 7
nExploredNodes: 26
```

As a search algorithm, BFS is both complete and optimal, i.e. it is guaranteed to generate all possible paths (if allowed to) and it is hence guaranteed to be able to find the optimal path eventually. However, in this implementation, we are stopping as soon as we reach a path without checking whether or not it is optimal.

2.1.b. Depth-first search (DFS)

Result of the DFS implementation for the path: *Euston to Victoria*:

```
path: ['Euston', 'Warren Street', 'Goodge Street', 'Tottenham Court Road', 'Oxford Circus',
"Regent's Park", 'Baker Street', 'Marylebone', 'Edgware Road', 'Paddington',
'Bayswater', 'Notting Hill Gate', 'Holland Park', "Shepherd's Bush", 'White City',
'East Acton', 'North Acton', 'West Acton', 'Ealing Broadway', 'Ealing Common',
'Acton Town', 'Chiswick Park', 'Turnham Green', 'Stamford Brook', 'Ravenscourt Park',
'Hammersmith', 'Barons Court', 'West Kensington', "Earls' Court", 'High Street Kensington',
'Gloucester Road', 'South Kensington', 'Sloane Square',
'Victoria']

pathCost: 69
nExploredNodes: 87
```

COMMENT ON THE ABOVE:
The above obtained path is in fact a valid path, but its absurd length (compared to the result of BFS) reflects a key fact about the nature of DFS; DFS traverses down a single branch, i.e. a single course of stations, until it reaches a deadend or the goal *without checking other possible branches from a given node* unless the branch it is currently expanding reaches a deadend. For this reason, DFS is considered non-optimal, i.e. not guaranteed to find the optimal solution. However, when we check for previously encountered nodes (as it is done in this implementation), we are guaranteed to find a valid path.

2.1.c. Uniform cost search (UFS)

Result of the UCS implementation for the path: *Euston to Victoria*:

```
path: ['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']
pathCost: 9
nExploredNodes: 25
```

2.2. Comparing BFS, DFS & UCS

The list of routes to search:

- "Euston" to "Victoria"
- "Canada Water" to "Stratford"
- "New Cross Gate" to "Stepney Green"
- "Ealing Broadway" to "South Kensington"
- "Baker Street" to "Wembley Park"
- "North Harrow" to "Stockwell"
- "Park Royal" to "Cheshunt" (*invalid path*)
- "Sudbury Hill" to "Swiss Cottage"

	route	function	pathCost	nExpandedNodes
0	(Euston, Victoria)	BFS	7.0	26
1	(Euston, Victoria)	DFS	69.0	87
2	(Euston, Victoria)	UCS	7.0	23
3	(Canada Water, Stratford)	BFS	15.0	26
4	(Canada Water, Stratford)	DFS	24.0	309
5	(Canada Water, Stratford)	UCS	14.0	34
6	(New Cross Gate, Stepney Green)	BFS	14.0	16
7	(New Cross Gate, Stepney Green)	DFS	28.0	357
8	(New Cross Gate, Stepney Green)	UCS	14.0	12
9	(Ealing Broadway, South Kensington)	BFS	20.0	47

	route	function	pathCost	nExpandedNodes
10	(Ealing Broadway, South Kensington)	DFS	89.0	183
11	(Ealing Broadway, South Kensington)	UCS	19.0	47
12	(Baker Street, Wembley Park)	BFS	13.0	11
13	(Baker Street, Wembley Park)	DFS	61.0	314
14	(Baker Street, Wembley Park)	UCS	13.0	31
15	(North Harrow, Stockwell)	BFS	36.0	131
16	(North Harrow, Stockwell)	DFS	154.0	169
17	(North Harrow, Stockwell)	UCS	36.0	115
18	(Park Royal, Cheshunt)	BFS	NaN	321
19	(Park Royal, Cheshunt)	DFS	NaN	375
20	(Park Royal, Cheshunt)	UCS	NaN	321
21	(Sudbury Hill, Swiss Cottage)	BFS	26.0	42
22	(Sudbury Hill, Swiss Cottage)	DFS	51.0	315
23	(Sudbury Hill, Swiss Cottage)	UCS	26.0	56

From the above, we get the following statistics for BFS, DFS and UCS:

function	pathCost-mean	pathCost-stdDev	nExpandedNodes-mean	nExpandedNodes-stdDev
BFS	18.714	3.379	77.5	34.878
DFS	68.0	15.460	263.625	34.235
UCS	18.429	3.398	79.875	33.857

UCS aims to explore lower cost paths first, which explains why it has the least average path cost among the uninformed search methods. BFS and UCS have similar statistics, which is more due to the nature of the domain than due to the similarity in search quality; in our situation, most stations are interconnected, which means exploring all possible options at every station is bound to lead to a relatively short path eventually. We also see that UCS explores slightly more nodes than BFS, but the difference does not reflect anything about the algorithms since it is a matter of chance whether following the less costly paths would lead to more deadends (thus more nodes visited) or not.

In general, for more dense (interconnected) graphs, BFS and UCS will tend to visit a high amount of nodes the further the starting point and goal are from each other, due to the high branching factor. However, due to the way these algorithms explore nodes, when points are relatively close to each other, both algorithms can reach the goal with a relatively shallow tree.

DFS is the worst performing algorithm here by far, with the highest mean path cost. The high standard deviation for the path cost suggests that DFS is relatively unstable, i.e. it is highly sensitive to its starting position and to the particular order in which the neighbours of each node are enountered. This is so because unlike BFS and UCS, DFS commits to one branch as far as it can rather than explore its surroundings completely. This approach is especially problematic in a dense and relatively big graph (i.e. a graph with many nodes and interconnections between nodes), because DFS's branch-focused policy may lead it to more convoluted paths even between points that are relatively close to each other, leading to the rise in the number of nodes visited.

As a special focus, consider the path...

```
Baker Street --> Wembley Park
```

These two stations are relatively close to each other (grids B3-C4 in the tube map: https://content.tfl.gov.uk/large-print-tube-map.pdf).

route	function	pathCost	nExpandedNodes
(Baker Street, Wembley Park)	BFS	13	11
(Baker Street, Wembley Park)	DFS	61	314
(Baker Street, Wembley Park)	UCS	13	31

Here, we see equally good performance of UCS and BFS in terms of path cost, although in this case BFS expands far fewer nodes (31 for UCS, 11 for BFS). However, DFS performs very poorly, exploring many more nodes (314) and resulting in a path with much higher cost (61). DFS comes up with a convoluted path despite the fact that the starting point and goal are relatively close (due to the reasons explained before), which means the branch it explores is very long; this is reflected by the number of expanded nodes, which is added along with the increasing branch length. UCS and BFS on the other hand visit all the surrounding nodes first, and thus, are more likely to hit the goal with a shallower tree. This reflects the fact that if the starting point and goal are relatively close, BFS and UCS perform very well due to their policy of exploring all immediate neighbours first.

Now, consider the inverse of the above path, i.e.

```
Wembley Park --> Baker Street
```

As the distance between the points are unchanged, BFS and UCS will arrive at more or less the same path cost due to the relative proximity of the points. However, DFS is more sensitive to its starting position...

route	function	pathCost	nExpandedNodes
(Wembley Park, Baker Street)	BFS	13	4
(Wembley Park, Baker Street)	DFS	20	13
(Wembley Park, Baker Street)	UCS	13	9

Here, we see that DFS has performed vastly better than last time (path cost 20 vs. last time's 61, expanded nodes 13 vs. last time's 314), even though the points are basically the same. This reflects the sensitivity of DFS to its starting position and the relative positions of its neighbours.

Overcoming the loop issue

A graph has no clear hierarchy, i.e. the connections between nodes are not unidirectional (ex. parent to child), which means nodes can be connected in a loop (i.e. a pathway returning to the starting position). This can be an issue when exploring the graph, since we may create pathways that overlap or that repeat themselves. Both these situations increase the number of nodes the algorithm visits, which in turn increases the time the algorithm spends looking for the goal. Furthermore, while BFS and UCS can eventually reach the goal even without correcting for loops, DFS is in danger of getting stuck in a loop indefinitely, since the basic version of DFS never backtracks unless it encounters a terminal node (a deadend or the goal). To solve this issue, the implementations simply maintain a list of explored nodes (called `visited` in the BFS and UCS implementations (*for reasons mentioned in the code comments*), and `expanded` in the DFS implementation). Whenever the algorithm expands from a given node, it always checks if the neighbour it is trying to visit already exists in the list of explored nodes, and whenever the algorithm successfully expands a given node, it stores this node in the list of explored nodes.

In my BFS and UCS implementations, I am using a method that requires me to store not just the explored node but the nodes just visited (i.e. the neighbours of the visited node). The reason for this is that in my implementations, the neighbour is immediately appended onto a path before it is expanded.

2.3. Extending the cost function

Considering line change time in the cost, we get the following updated costs...

	route	function	pathCost	nExpandedNodes
0	(Euston, Victoria)	BFS	9.0	26
1	(Euston, Victoria)	DFS	81.0	87
2	(Euston, Victoria)	UCS	9.0	25
3	(Canada Water, Stratford)	BFS	15.0	26
4	(Canada Water, Stratford)	DFS	30.0	309
5	(Canada Water, Stratford)	UCS	15.0	30
6	(New Cross Gate, Stepney Green)	BFS	16.0	16
7	(New Cross Gate, Stepney Green)	DFS	36.0	357
8	(New Cross Gate, Stepney Green)	UCS	16.0	11
9	(Ealing Broadway, South Kensington)	BFS	30.0	47
10	(Ealing Broadway, South Kensington)	DFS	113.0	183
11	(Ealing Broadway, South Kensington)	UCS	26.0	57
12	(Baker Street, Wembley Park)	BFS	13.0	11
13	(Baker Street, Wembley Park)	DFS	71.0	314
14	(Baker Street, Wembley Park)	UCS	13.0	18
15	(North Harrow, Stockwell)	BFS	40.0	131
16	(North Harrow, Stockwell)	DFS	176.0	169
17	(North Harrow, Stockwell)	UCS	40.0	119
18	(Park Royal, Cheshunt)	BFS	NaN	321
19	(Park Royal, Cheshunt)	DFS	NaN	375
20	(Park Royal, Cheshunt)	UCS	NaN	320
21	(Sudbury Hill, Swiss Cottage)	BFS	30.0	42
22	(Sudbury Hill, Swiss Cottage)	DFS	59.0	315
23	(Sudbury Hill, Swiss Cottage)	UCS	30.0	48

From the above, we get the following statistics for BFS, DFS and UCS:

function	pathCost-mean	pathCost-stdDev	nExpandedNodes-mean	nExpandedNodes-stdDev
BFS	21.857	4.01	77.5	34.878
DFS	80.857	17.647	263.625	34.235
UCS	21.286	3.874	78.5	34.184

Since BFS and DFS do not change the order of nodes to search based on path cost, it is clear that the `nExpandedNodes` statistics will be unchanged for them. Apart from the added cost of line change, BFS and DFS statistics are unchanged. UCS does change its search priority based on path cost, hence the new path cost function will try to minimise the number of line changes in a path due to the added penalty of line change to the cost function. Thus, we can expect changes for all the statistics of UCS.

As expected, the mean path costs for each search method has increased, but for BFS and UCS, the change is small (a change in cost of around 3) whereas the change for DFS is much larger. This is expected since in a relatively big interconnected graph,

DFS tends to come up with lengthier more convoluted paths, and the longer and more roundabout a path is, the more likely it is to have more line changes, leading to a higher overall line-change penalty.

The low effect of this updated cost on BFS and UCS can have the following reasons:

- UCS considers this updated cost as well, optimising with respect to it
- Most of the points here are relatively close, leading to BFS & UFS finding the goal with relatively shallow trees, which means closer to the neighbourhood of the starting point & thus fewer line changes in general
- The small penalty for line change (addition of 2 on the time cost)

2.4. Heuristic search

2.4.a. Understanding zones

The possible zones in this dataset are:

```
['1', '2', '3', '4', '5', '6', 'a', 'b', 'c', 'd']
```

In order, we can see that the available zones are 1, 2, 3, 4, 5, 6, a, b, c, d (*note that only a secondary zone can be assigned a zero, which only means that the secondary zone does not exist*). To see what a, b, c and d could mean, we use to following code...

```
a: ['Moor Park', 'Croxley', 'Watford', 'Rickmansworth']
b: ['Chorleywood']
c: ['Chalfont & Latimer']
d: ['Chesham', 'Amersham']
```

Upon comparing these station names with the zones given on the tube map, we see (keeping in mind that our CSV file's data is not up-to-date and may have discrepencies with the map):

- a is 7
- b is 8
- c is 9
- d is 10

Furthermore, from the map, we see that the zones are ordered in roughly concentric circles, with the inner-most zone (centered with respect to London of course) being zone 1. Now, for our convenience, we shall reassign `zone_dict` values as integer sets rather than sets of characters. We could have done this initially, but I have chosen to do this here due to take a more detailed look at zone values, which could only take place after the initial data input.

```
Harrow & Wealdstone: [5]
Kenton: [4]
South Kenton: [4]
North Wembley: [4]
Wembley Central: [4]
```

2.4.b. Creating a zone-based heuristic

Note that we will be using `zone_dict` as `zoneMap`, whose items are lists of the form `"station name":list_of_zones`. My zone heuristic works as follows:

- Obtains the absolute difference between zones 1 & 2 of the child node with respect to the current node as a
- Obtains the minimum of the absolute differences between the child node's zones & the goal node's zones as b
- Returns $a + b$

The motivation for the above is as follows:

- For a neighbour N_c of the current node N, zone 1 is the zone of node N whereas zone 2 is the zone of node N_c; thus, zones 1 & 2 together are the zones of the connecting points of the two nodes
- Travelling within the same zone has no extra charge (in terms of payment), whereas travelling between different zones does; thus, the difference in zones is important in evaluating the goodness of a tube network path
- The above condition also applies for the difference in zones between the goal node & the current node's neighbours; in general, we must aim to minimise the changing of zones, be it from the current node to the next or from the next node to the goal (as far as we can estimate)
- We need the heuristic when node N_c is not an immediate neighbour to the goal; thus, we do not have a clear idea about the zones involved when travelling from N_c to the goal & thus need to make an estimate
- To guarantee the admissibility of the heuristic, we take the estimate as the minimum zone difference between the possible zones of N_c and the goal (*one station can be in two zones*)

Applying this heuristic on the framework of UCS and getting results for multiple routes...

	route	pathCost	nExpandedNodes
0	(Euston, Victoria)	7.0	15
1	(Canada Water, Stratford)	14.0	25
2	(New Cross Gate, Stepney Green)	14.0	10
3	(Ealing Broadway, South Kensington)	19.0	36
4	(Baker Street, Wembley Park)	13.0	23
5	(North Harrow, Stockwell)	36.0	91
6	(Park Royal, Cheshunt)	NaN	271
7	(Sudbury Hill, Swiss Cottage)	26.0	41

Let us consider the path:

```
Euston --> Victoria Green
```

With the current heuristic (*heuristic 1*), while we do not find a better path (presumably because the previously obtained path was already optimal), we have reduced the number of nodes expanded from 26 (for BFS) and 23 (for UCS) to 15 for UCS with *heuristic 1*. This indicates that the pathfinding, at least for this path, was better informed and able to proceed (to a greater extent) in the direction of goal.

2.5. Comparing heuristics

From the results of the previous heuristic(*heuristic 1*), we get the following statistics:

- `mean(nExpandedNodes)` : 64.0
- `stdDev(nExpandedNodes)` : 28.888
- `mean(pathCost)` : 18.429
- `stdDev(pathCost)` : 3.398

We see that on average, compared the the UCS algorithm without the heuristic, adding the heuristic has decreased the number of nodes expanded (with a similar (*somewhat lower*) standard deviation maintained, thus indicating that the algorithm has not become more volatile). The final path cost is identical to UCS, indicating that it performs at least as well as UCS by not overestimating the heuristic cost so as to miss the more optimal paths. Hence, based on this test, we see the indication that this heuristic is overall an improvement in UCS, and does not take away from the goodness of the final path.

For the other heuristic, I have implemented the 1st suggested heuristic, which is explained the instructions PDF of this assignment as follows:

"*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.*"

Getting results for multiple routes...

	route	pathCost	nExpandedNodes
0	(Euston, Victoria)	7.0	17
1	(Canada Water, Stratford)	14.0	26
2	(New Cross Gate, Stepney Green)	14.0	11
3	(Ealing Broadway, South Kensington)	25.0	49
4	(Baker Street, Wembley Park)	13.0	22
5	(North Harrow, Stockwell)	36.0	89
6	(Park Royal, Cheshunt)	NaN	271
7	(Sudbury Hill, Swiss Cottage)	26.0	37

From the above heuristic (*heuristic 2*), we get the following statistics:

- `mean(nExpandedNodes)` : 64.625
- `stdDev(nExpandedNodes)` : 20.354
- `mean(pathCost)` : 17.6
- `stdDev(pathCost)` : 2.303

The results are very similar to the previous heuristic used, with a slightly higher amount of nodes explored on average (although that is not significant, as far as we know). However, the average path cost is notably lower (though by small amount) than UCS (as well as the UCS with heuristic 1), although this result is not significant with respect to what we know. The different heuristics are seen to perform differently for the different paths, and no one is very clearly better than the other. Furthermore, it is not clear yet how the selection fo different points would alter the above statistics of the heuristics with respect to each other.

With respect to the test, we get the following indications...

Advantages of heuristic 1:

- Clear logical basis
- No worse than UCS
- Reduces the number of nodes explored

Disavantages of heuristic 1:

- Relatively complex (especially due to the process of finding the minimum difference between neighbour & goal zones)
- Not clearly better than the much simpler heuristic 2
- Tends to underestimate the zone distance with respect to the goal

Advantages of heuristic 2:

- Better than UCS & heuristic 1
- Also reduces the number of nodes explored (to a similar level as heuristic 1)
- Simpler to implement & calculate than heuristic 1

Disadvantages of heuristic 2:

- Does not consider zone distance from the goal

3. Genetic algorithms

3.1. Implementation

We assign the fitness of a possible password by computing the string distance between the possible password and the actual password; the string distance is given by the sum of the square roots of the absolute differences between the Unicode values of each character of the possible and actual passwords. Finally, we normalise the distance function's output such that the final fitness lies between 0 and 1, with 1 being the highest possible fitness and 0 being the lowest. This allows us to see how close to the true value we actually are. Hence, for a candidate password $x = (x_1, x_2 ... x_{10})$ and given the actual password $y = (y_1, y_2 ... y_{10})$:

$$Distance(x, y) = \sum_{i=1}^{10} \sqrt{|Unicode(x_i) - Unicode(y_i)|}$$

$$Fitness(x) = 1 - \frac{1}{max_{Unicode}} Distance(x, y)$$

where:

$$max_{Unicode} = Distance(''0000000000'', ''\_\_\_\_\_'')$$

i.e. the maximum possible difference between two character strings under the constraints of this domain (*which will be discussed in the section **3.2.a** on state representation*).

IMPLEMENTATION NOTE 1:
By the way the distance function is defined, it is clear that both the characters involved as well as the order of these characters factors into the final fitness calculation. Hence, it is appropriate for our use.

IMPLEMENTATION NOTE 2:
In this implementation of the evolutionary search algorithm, the password is inaccessible by the evolutionary search function, and only the fitness can be calculated using a method visible to the password object. *The password visible only outside the evolutionary search function and only for testing purposes*.

Performing the evolutionary search with population size 25, probability of mutation (pMuta) 0.2 and probability of crossover (pCO) 1, we get the following results:

```
Max fitness of initial population: 0.5644908857404403

Search started!
............
Search complete!
Reproductions: 133

Goal: 68IO1W3QF9
Best 5 individuals:
#1.   68IO1W3QF9; fitness=1.0
#2.   68IO2W3QF9; fitness=0.9854135008502105
#3.   68IO2W3QF9; fitness=0.9854135008502105
#4.   68IO2W3QF9; fitness=0.9854135008502105
#5.   68IO2W3QF9; fitness=0.9854135008502105

We see that the true password is found as `68IO1W3QF9`, which is equal to the goal.
```

3.2. Algorithm components

3.2.a. State representation

In this domain, each state is a candidate password, which means each state is represented by a string of 10 characters, wherein the characters may be chosen only from the following:

- Uppercase English alphabet ('A'-'Z')
- Decimal digit characters ('0'-'9')
- Underscore ('_')

Random individuals as well as the initial population can be created by randomly sampling (ten times to obtain an individual) of the above list of possible characters.

3.2.b. Mating selection

The top four individuals are selected to mate with everyone else. The reason is that being the top individual implies more correct characters in the right places, which means it is beneficial for the top individuals to recombine their genes (without changing gene positions) so as to approach a potentially better set of individuals. However, these top four are also mated with the rest of the individuals that are not at the top so as to improve the diversity among candidate solutions.

3.2.c. Crossover

This implementation uses uniform crossover. The reason is that position of a gene is vital to its goodness, which means fitter individuals tend to have more genes in the right position and any superior positioning of genes must not be disrupted. At the same time, top individuals may have different genes at the right positions, which means recombining them without altering any gene positions can potentially lead to an individual with even more genes in the right positions.

The crossover between two individuals is done based on the following:

Given a chance of crossover p_{CO}, every individual has p_{CO} chance of crossing over with a random individual from the mating pool. Once the individuals are chosen for crossover, an offspring is created such that for every gene of the offspring, there is a 50% chance of getting it from one parent and a 50% chance of getting it from the other.

In this implementation, offsprings from the crossover do not immediately replace the existing population but instead are added to the existing population. This is to allow existing fit solutions to propagate to the next generation if they are fit enough. Such an approach is more beneficial in the current domain than one that favours more diversity right away, since fitness depends on very clear & definite parameters, namely the distance of correspondinng characters of the candidate solution from the characters of the actual password. In this context, the fittest parameters have a much higher chance of producing optimal or close-to-optimal solutions.

3.2.d. Mutation

For this implementation, the mutation function works as follows:

Given a chance of mutation p_{muta}, every gene of an individual has p_{muta} chance of mutating.

Mutation is done in this manner because mutating one gene per individual may be too slow for convergence, since a candidate password could be off from the true password at multiple different gene positions at once. Indeed, when testing with mutating one gene per individual, the convergence was far slower and more unreliable.

I have decided to only mutate the offsprings of the crossover, for the following reasons:

- Lower computational cost & time
- Preservation of most of the existing top candidate solutions

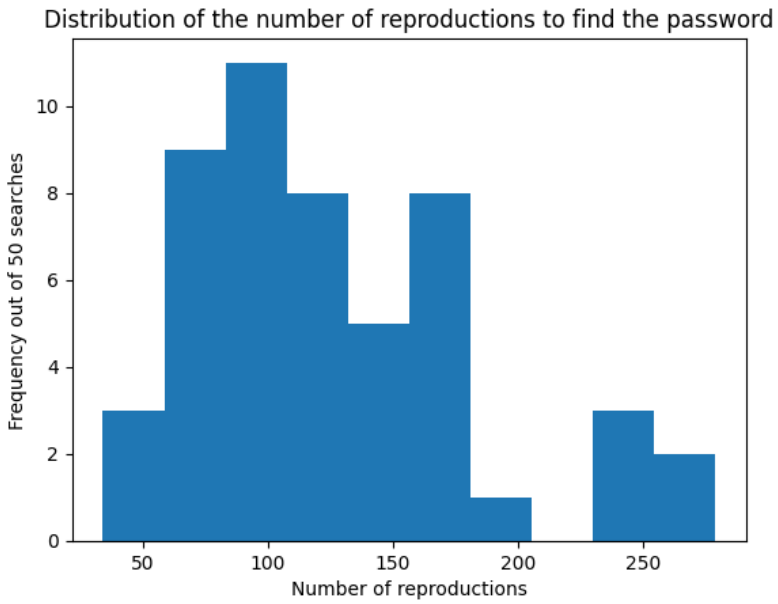
3.2.e. Environmental selection

The selection of the next generation after crossover and mutation is done by the principle of elitism, i.e. choosing the top (i.e. fittest) candidate solutions while maintaining constant population size. This maintains the best candidate solutions so far without recklessly replacing them with new candidate solutions.

3.3. Number of reproductions

Finding the average number of reproductions it takes to reach the desired result...

Obtaining the statistics...




```
Mean: 127.66
Standard deviation: 8.10780414168966
```

3.4. Effect of hyperparameters

Below are the results of experiments on three parameters:

- `pMuta` : Probability of mutation
- `pCO` : Probability of crossover
- `popSize` : Constant population size

For the hyperparameter, we defined the following list of values to test:

- `pMutas = [0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5]`
- `pCOs = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]`
- `popSizes = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]`

For each hyperparameter value, the evolutationary search was run 10 times (with the reinitialisation of the random initial population at every iteration to improve the sample's quality). When working with one hyperparameter, the others' values were kept constant:

- When iterating on `pMutas` , `pCO=1` & `popSize=20`
- When iterating on `pCOs` , `pMuta=0.1` & `popSize=20`
- When iterating on `popSizes` , `pMuta=0.1` & `pCO=1.0`

At the end of 10 iterations for each hyperparameter value, the following statistics were obtained:

- `nReps-mean` : Mean number of reproductions (*i.e. iterations of the search algorithm*) until convergence
- `nReps-stdDev` : Standard deviation of repetitions until convergence
- `maxFit-mean` : Mean of the maximum fitness of obtained (generally is 1)
- `maxFit-stdDev` : Standard deviation of the maximum fitness of obtained (generally is 0)

==========

Getting stats for parameter: pMuta

Number of unique values to check: 10

[.....]

	value	nReps-mean	nReps-stdDev	maxFit-mean	maxFit-stdDev
0	0.01	570.9	124.103219	1.000000	0.000000
1	0.02	398.5	48.318992	1.000000	0.000000
2	0.03	233.0	34.026754	1.000000	0.000000
3	0.04	216.9	25.864822	1.000000	0.000000
4	0.05	152.3	21.093151	1.000000	0.000000
5	0.10	131.9	31.165831	1.000000	0.000000
6	0.20	183.9	27.205312	1.000000	0.000000
7	0.30	256.2	41.656404	1.000000	0.000000
8	0.40	774.9	159.478303	1.000000	0.000000
9	0.50	1735.7	224.959865	0.998541	0.001384

```
==============================================
Getting stats for parameter: pCO

Number of unique values to check: 10
[.....]
```

	value	nReps-mean	nReps-stdDev	maxFit-mean	maxFit-stdDev
0	0.1	1239.5	143.280302	1.0	0.0
1	0.2	694.6	118.005949	1.0	0.0
2	0.3	352.4	30.946147	1.0	0.0
3	0.4	333.0	51.768523	1.0	0.0
4	0.5	259.1	27.336221	1.0	0.0
5	0.6	180.7	22.245921	1.0	0.0
6	0.7	125.9	12.566185	1.0	0.0
7	0.8	147.0	8.742997	1.0	0.0
8	0.9	150.4	14.708637	1.0	0.0
9	1.0	125.2	14.502276	1.0	0.0

```
==============================================
Getting stats for parameter: popSize

Number of unique values to check: 10
[.....]
```

	value	nReps-mean	nReps-stdDev	maxFit-mean	maxFit-stdDev
0	10	290.0	24.938324	1.0	0.0
1	20	153.5	13.302068	1.0	0.0
2	30	88.5	9.163242	1.0	0.0
3	40	62.9	11.091844	1.0	0.0
4	50	56.9	6.759364	1.0	0.0
5	60	43.2	6.131558	1.0	0.0
6	70	37.8	3.419942	1.0	0.0
7	80	39.1	4.371384	1.0	0.0
8	90	29.2	4.571214	1.0	0.0
9	100	29.9	2.851140	1.0	0.0

The above results indicate the following based on the values of `nReps-mean` and `nReps-stdDev` ...

1.

`pMuta` is optimal between 0.05 and 0.1; the convergence of the evolutionary search only reduces on either side of this range. Hence, the peak of the optimisation function (with respect to `nReps-mean`) of `pMuta` lies around $[0.05, 0.1]$. Note that `pMuta` $\in [0, 1]$.

2.

`pCO` is optimal at 1.0, and suboptimal for all values below 1.0. This means that crossover (*done in the method applied in this implementation*) is always preferable to skipping crossover. Note that `pCO` $\in [0, 1]$.

3.

The larger the `popSize`, the faster the convergence. We cannot comment on the optimality of `popSize`, since it does not have a finite range like `pMuta` and `pCO`. However, though `nReps-mean` (mean number of reproductions) almost monotonically decreases as `popSize` increases, it must be noted that a higher `popSize` results in greater computational cost which could offset the gain in convergence (with respect to the number of reproductions) after a certain point.

4.

For all parameters and their values, lower `nReps-mean` coincides with lower `nReps-stdDev`, meaning that a higher optimality of the above parameters with respect to the number of reproductions goes hand-in-hand with a greater reliability of the search algorithm (i.e. lesser variability in the performance of the algorithm).

Judging by the statistics presented here, we see that increase in `popSize`, i.e. population size leads to the greatest change in convergence. This could be understood as follows...

The evolutionary search's performance is limited by the diversity of its population and its ability to produce and filter a wider range of options. `pMuta` and `pCO` are hyperparameters that apply only to the existing population, and cannot achieve more than the existing population is capable of. Thus, with sufficiently well-performing methods of crossover and mutation in place, population size is the most significant hyperparameter in improving the accuracy and reliability of the evolutionary search algorithm.