Programming Assignment 1

Nupur Funkwal
2018A7PS0624G

Question 1 (8 Queens Problem)

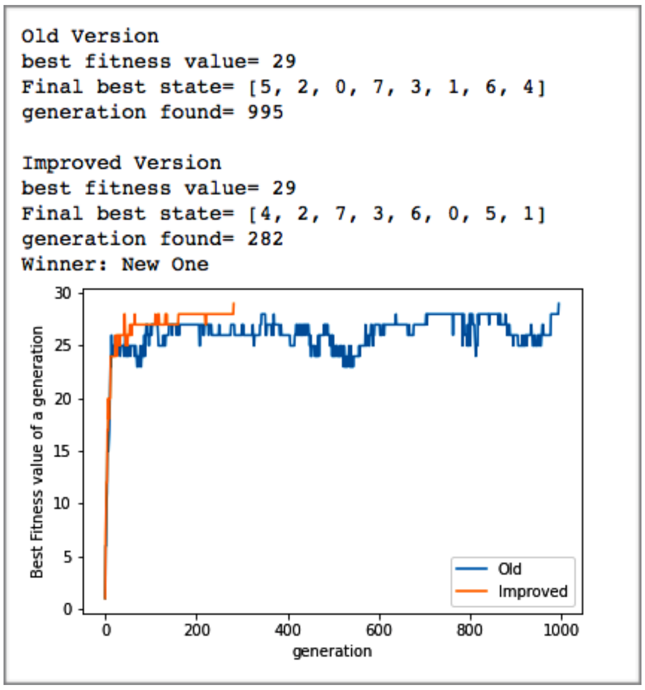
The state of the chess board is represented using an 8-element list (0-7), each of which tells the row position of the queen in that column. Eg. [1,2,1,3,5,6,7,0]

The old version of the solution for this problem uses a population size of 20. In order to bring more diversity in the population and avoid premature convergence, the population size was increased to **40**.
I used a new **Reproduce( )** function for improvement. In this, first the **fitness** of both the parents were calculated and then **probabilities** based on fitness (i.e. F(x)/F(x)+F(y)) for making a choice later were calculated respectively. The meaning of reproduction is to inherit from parents. So the idea was that the new child should represent either of the parents in a probabilistic manner. So if a value at a particular index is **same** in both the parents, it is assumed that the value is part of a good solution. So the same value is used at that index in the child as well. If at a particular index, the value is different in both the parents, then we choose one of the parents' value using probability (which was based on fitness of the parent) and put that in the child at that index. Therefore, in case both parents don't have the same fitness, the child will have more chances of being similar to the better parent and having improved fitness. This helps in getting better new population (as fitness of child has less chances of deterioration) faster compared to the old version.
Example of the crossover:

x = [0,1,3,2,5,7,2,4]
y = [0,3,5,2,6,7,1,0]
child = [0,1,5,2,6,7,2,4]

Here 0, 2 and 7 are common in both parents at respective indices, hence copied in child. There remaining indices are filled using probability from either of the parents.

Fitness function = 1 + no. of pairs of queens not in attacking position
Mutate( ) was kept same as old version (i.e. randomly changing the value at any index).
Probability of mutation = 0.2



```
Old Version
best fitness value= 29
Final best state= [5, 2, 0, 7, 3, 1, 6, 4]
generation found= 995

Improved Version
best fitness value= 29
Final best state= [4, 2, 7, 3, 6, 0, 5, 1]
generation found= 282
Winner: New One
```

In the plot shown, the new version performs better by finding the solution pretty early at generation = 282 while old version finds it at generation = 995. The code is run till generation = 1000 in worst case.

Question 2 (Travelling Salesman Problem)

Each state of population is represented by a 14 element list. Cities are labelled from 'A' to 'N'.
The old version has a population size of 20. The new version has population size of **30** to bring more diversity and avoid premature convergence. A distance of 1000 units has been used to represent infinity in the code.

Fitness Function = (1 / Distance(state)) + 1

It was observed that TSP is very sensitive to the selection of parents for reproduction. If a parent has bad fitness, the child turns out very bad. Therefore, when the code was run using the standard selection method for parents, in many cases bad parents were getting selected significantly giving poor results. Hence, we need to improve the probability of choosing a good parent. Since the distance is in denominator for fitness calculation, fitness values come out to be very small and probabilities come out to be quite close for all parents.To solve this, Min-Max scaling of fitness values was done. And then those values were used to generate probabilities for selection of parents.

The Reproduce( ) method in new is **same** as old version, where a random subset is chosen from the first parent and is added to the offspring. The missing values in child are then added from the second parent in the same order in which they are present in parent and must not be present in the child beforehand.

I have made an improvement in the **Mutate( )** function. The old version selects two random cities and swaps them. Steps taken in the improved version are:
Step 1 - A random subset (continuous) is chosen from the given state and is **reversed.**
Example

state = ['D', 'K', 'M', 'L', 'G', 'A', 'J', 'H', 'N', 'B', 'I', 'E', 'C', 'F']
mutated state after step 1 = ['D', 'K', 'A', 'G', 'L', 'M', 'J', 'H', 'N', 'B', 'I', 'E', 'C', 'F']

Step 2 - A random city (x) is picked from the state and the nearest city (y) to this city is computed from the graph. After this, we pick any random neighbour (z, which is not at infinity) of this nearest city (y) and **swap** it (z) with the initial city (x). This mutation is based on the idea of nearest neighbour cities.

So step 1 helps in bringing more diversity faster. In step 2, we are swapping two near neighbours of a city in a state in attempt to get better fitness. Overall this version performs much better than the old version by giving results in lesser time.

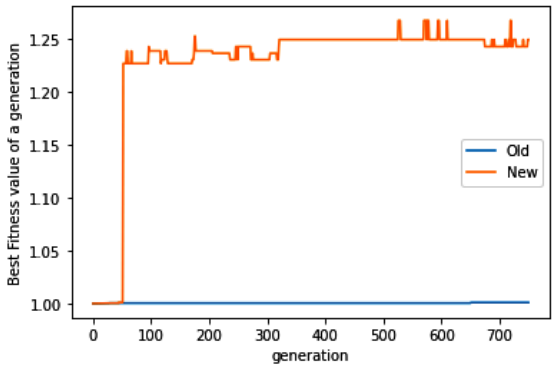
Mutation probability of new version = 0.20

```
Old Version

best fitness value= 1.0009953219866627
Optimal distance found= 1004.6999999999399
Final best state= ['D', 'C', 'E', 'G', 'A', 'L', 'J', 'H', 'B', 'I', 'N', 'F', 'M', 'K']
generation found= 659

New Version

best fitness value= 1.267379679144385
Optimal distance found= 3.7400000000000007
Final best state= ['J', 'A', 'G', 'L', 'M', 'K', 'D', 'F', 'C', 'E', 'I', 'B', 'N', 'H']
generation found= 526
```



This genetic algorithm is run for 750 generations and returns the best optimal distance found so far. From the plot, we can see that the new version is able to find optimal distance of 3.74 units while the old version finds optimal distance of 1004.69 units due to infinity taken as 1000 units.
Note: Some parts of the graph might appear as a straight line because Fitness function has distance in its denominator, so when the distance is in thousands, fitness function shows very little variation in the plot.

Mutation plays a very important role in this problem because the solution space is sparse, so more diversity in population can help in reaching a solution faster.