Task 1 – Report

1. Introduction

In this report, we'll discuss the algorithms and their performance as was tasked in Exercise 1.

In the first experiment, we'll attempt to solve the 8 queens problem and in the second experiment, we'll attempt to "guess" a given Shakespearean text.

For each experiment, we'll explain the algorithm we used to solve the problem, followed by the way we played with the hyperparameters and the results.

The programming language we chose for this experiment is **python 3.7** and the run environment is **Google's Colab** online editor.

2. 8-Queen:

2.1 Algorithm

We decided to represent the **chromosomes** in this form: [3, 2, 1, 5, 4, 6, 7, 8]

Each entry in this vector is a position for the queen in such a way that the queen cannot be attacked by another queen.

The **Fitness function** is the number of queens that threaten each other. therefore 0 is the optimal score and this is a minimization problem.

Those are the parameters in our code:

- Mutation Rate
- Crossover Rate
- Number Of Generations
- Population Size
- Extinction [How many of the lowest scoring chromosomes are removed from the list of potential parents]
- Elitism

2.2 Tunings

In **attempt #1**, we decided to give a high mutation rate and low extinction, which caused the problem to work for a long time.

It took the code to finish in 1 minute which is bad, so we understood something isn't good here, we need to change the hyperparameters.

The hyperparameters:

- mutation_rate = 0.4
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 100
- extinction = 10
- elitism = 2

```
[0:00:58.736033][Gen: 960 / 10000]      Average: 15.52  std: 4.924      Best: 2.0
[0:00:58.796485][Gen: 961 / 10000]      Average: 14.78  std: 4.698      Best: 2.0
[0:00:58.862880][Gen: 962 / 10000]      Average: 14.88  std: 5.233      Best: 2.0
[0:00:58.921534][Gen: 963 / 10000]      Average: 14.42  std: 5.398      Best: 2.0
[0:00:58.982536][Gen: 964 / 10000]      Average: 14.1   std: 4.677      Best: 0.0
Problem solved. breaking
```

This run was concluded in success after 964 generations and approx. 59 seconds.

In **attempt #2**, we decided to increase the extinction number, since we want to get better chromosomes for the next generations, therefore we increased extinction to 50. and we finished the task in 9 seconds which is better, but not good enough.

```
[0:00:08.549073][Gen: 138 / 10000]      Average: 14.64  std: 5.090      Best: 2.0
[0:00:08.617208][Gen: 139 / 10000]      Average: 13.84  std: 4.755      Best: 2.0
[0:00:08.678078][Gen: 140 / 10000]      Average: 14.44  std: 5.111      Best: 2.0
[0:00:08.737736][Gen: 141 / 10000]      Average: 13.74  std: 4.369      Best: 2.0
[0:00:08.817238][Gen: 142 / 10000]      Average: 13.78  std: 4.185      Best: 2.0
 0:00:08.884189 [Gen: 143 / 10000]      Average: 13.6   std: 5.154      Best: 0.0
Problem solved. breaking
```

This run was concluded in success after 143 generations and approx. 9 seconds. which is better, but not good enough.

In **attempt #3** we used the following Hyperparameters:

- mutation_rate = 0.1
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 100
- extinction = 50
- elitism = 2

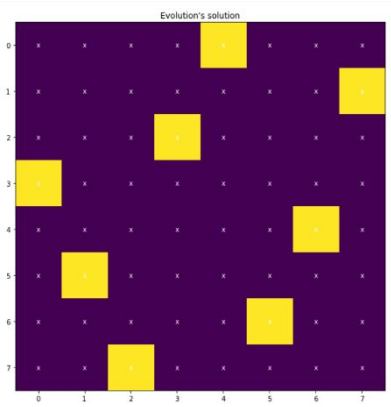
```
[0:00:00.049749][Gen:   1 / 10000]      Average: 15.42  std: 5.324      Best: 6.0
[0:00:00.109958][Gen:   2 / 10000]      Average: 13.1   std: 4.274      Best: 6.0
[0:00:00.167214][Gen:   3 / 10000]      Average: 12.3   std: 3.735      Best: 2.0
[0:00:00.231261][Gen:   4 / 10000]      Average: 12.84  std: 4.456      Best: 2.0
[0:00:00.293284][Gen:   5 / 10000]      Average: 12.08  std: 3.846      Best: 2.0
[0:00:00.353743][Gen:   6 / 10000]      Average: 11.82  std: 3.678      Best: 2.0
[0:00:00.410812][Gen:   7 / 10000]      Average: 11.38  std: 3.957      Best: 2.0
[0:00:00.467057][Gen:   8 / 10000]      Average: 11.34  std: 4.587      Best: 2.0
[0:00:00.525209][Gen:   9 / 10000]      Average: 10.8   std: 4.817      Best: 2.0
[0:00:00.585490][Gen:  10 / 10000]      Average: 10.16  std: 4.872      Best: 2.0
[0:00:00.642465][Gen:  11 / 10000]      Average: 10.06  std: 4.200      Best: 2.0
[0:00:00.700872][Gen:  12 / 10000]      Average: 10.52  std: 4.504      Best: 2.0
[0:00:00.763309][Gen:  13 / 10000]      Average: 9.38   std: 4.379      Best: 2.0
[0:00:00.823180][Gen:  14 / 10000]      Average: 9.14   std: 4.743      Best: 2.0
[0:00:00.879129][Gen:  15 / 10000]      Average: 7.52   std: 3.764      Best: 2.0
[0:00:00.935918][Gen:  16 / 10000]      Average: 6.26   std: 3.640      Best: 2.0
[0:00:00.994130][Gen:  17 / 10000]      Average: 5.56   std: 3.894      Best: 2.0
[0:00:01.062471][Gen:  18 / 10000]      Average: 5.32   std: 3.755      Best: 0.0
Problem solved. breaking
```

This run was concluded in success after 18 generations and approx. 1 second. We figured out a good hyperparameter for the model to quickly find a solution.

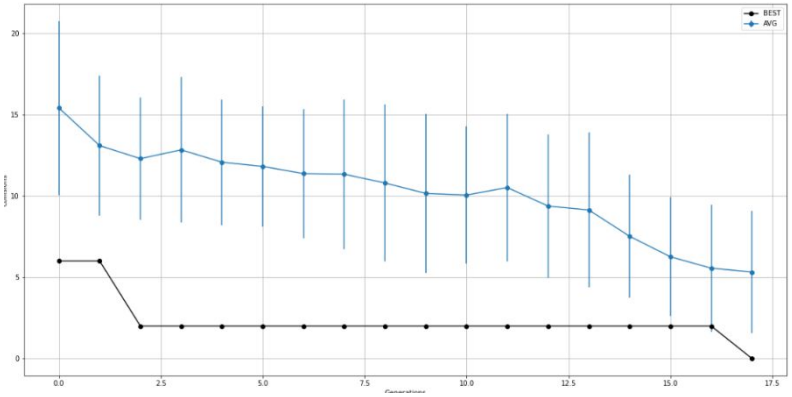
2.3 Results

We used the hyperparameters that have proven optimal in the previous section (2.2) to perform analysis.

The chosen chromosome is [3, 5, 7, 2,0,6,4,1]. Which represent the following scenario:



The following is a graph of the population fitness:



The blue graph is the average fitness of the population (with a line that represents the standard deviation in the population), The black graph is the score of the top runner of that generation.

The brute force algorithm for this method took about 03:36 minutes. which is significantly less efficient.

```
[0:03:36.471131][208686]          [7 4 2 4 3 3 2 4]          Score: 18.0
[0:03:36.472037][208687]          [1 2 5 0 7 3 1 6]          Score: 10.0
[0:03:36.472924][208688]          [5 3 0 3 5 4 1 7]          Score: 8.0
[0:03:36.473807][208689]          [3 3 7 2 4 6 0 7]          Score: 6.0
[0:03:36.474693][208690]          [2 5 3 0 7 4 6 1]          Score: 0.0
Problem solved. breaking
```

2.4 Discussion

it can be seen that the “best” score in the process is a strictly monotonic function (due to elitism) and the “average” graph might have slight up & down noise on it, but it is decreasing as generation advance.

It can also be seen that th brute force is not as effective for this problem.

3. Shakespeare's text:

3.1 Algorithm

We declared a vocabulary that consists of a-z, spaces and dots. ('abcdefghijklmnopqrstuvwxyz .'), then we started from random text and we decided to check whether a string is improving by checking if a character at index i match to Shakespeare's text at index i.

The fitness function is the number of matches between a chromosome and the original string.

Therefore, the algorithm will stop when the fitness function will return 300 since there are 300 characters in Shakespeare's text.

Those are the parameters in our code:

- Mutation Rate
- Crossover Rate
- Number of Generations
- Population Size
- Extinction
- Elitism

3.2 Tunings

In **attempt #1** we started with a low extinction rate compare to the population size, after running for **14:08 minutes** it reached to 10,000 generations and we stopped running, the code didn't coverage fast enough.

The hyperparameters:

- mutation_rate = 3 * 1.0 / length_of_text
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 400
- extinction = 50
- elitism = 10

In **attempt #2** we decided to cut off a large number of bad chromosomes **(extinction)** in order to generate a more stable generation, this impacted dramatically on the results and we managed to get to a result in **22 seconds** which is great improvement.

The hyperparameters:

- mutation_rate = 3 * 1.0 / length_of_text
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 10

In **attempt #3** we decided to **raise the crossover rate** and **elitism** to see what's happening, and as we guessed, there were too many crossovers that caused bad performances and we managed to finish the task in **21 seconds**.

The hyperparameters:

- mutation_rate = 3 * 1.0 / length_of_text
- crossover_rate = 0.95
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 50

In **attempt #4** we decided to **reduce the crossover rate** to see what's happening, we finished the task in **25 seconds**, and that happens because there are fewer crossovers between two parents, but there is a bigger number of elitism that takes good chromosomes.

The hyperparameters:

- mutation_rate = 3 * 1.0 / length_of_text
- crossover_rate = 0.7
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 50

In **attempt #5** we decided to **raise the mutation rate**, this causes a really bad performance because there were too many mutations and the algorithm couldn't coverage fast, this setting took us **3 minutes and 48 seconds**.

The hyperparameters:

- mutation_rate = 8 * 1.0 / length_of_text
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 50

In **attempt #6** we decided to **reduce the mutation rate**, this setting took us **29 seconds** solution, which is better, which means raising the mutation rate too much is really bad.

The hyperparameters:

- mutation_rate = 1 * 1.0 / length_of_text
- crossover_rate = 0.8
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 50

In **attempt #7** we decided **to raise a little bit the mutation rate** and we found a good hyperparameter which makes the model to output the best performance, this algorithm with those parameters finished the task in **18 seconds**.

The hyperparameters:

- mutation_rate = 3 * 1.0 / length_of_text
- crossover_rate = 0.87
- number_of_generations = 10000
- population_size = 400
- extinction = 300
- elitism = 10

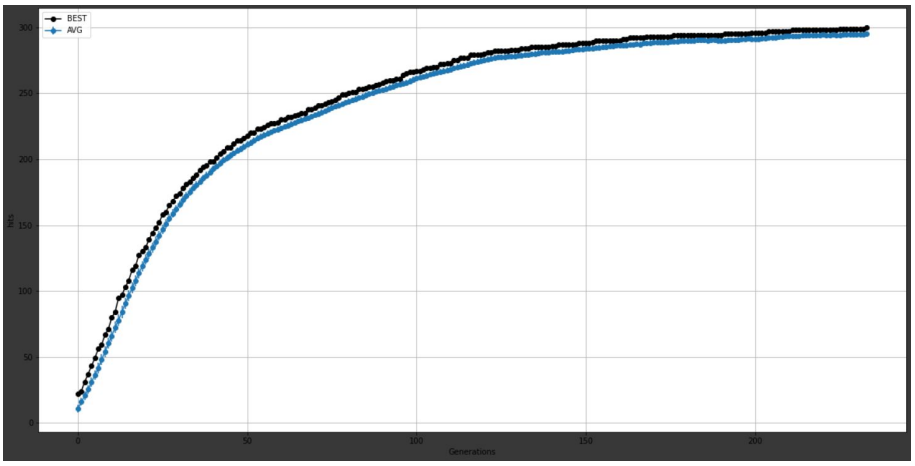
3.3 Results

We're going to take the last hyperparameters and use them in our model and plot the graph of it.

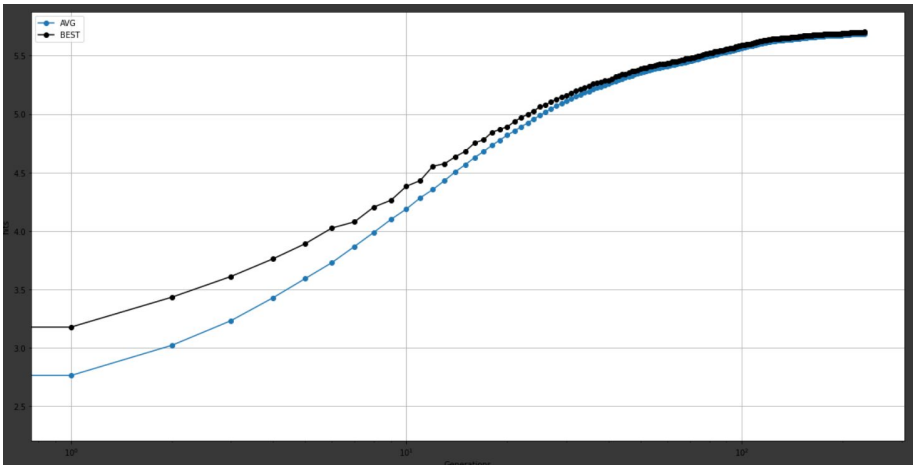
The black line is the best score of that generation and the blue is the average (the standard deviation was gathered, but can't be seen in the graph).

The X-axis is the number of generation and the Y-axis is the number of matches between a chromosome and the input text.

The best and average score of each generation on a **regular** scale.



The best and average score of each generation on a **logarithmic** scale.



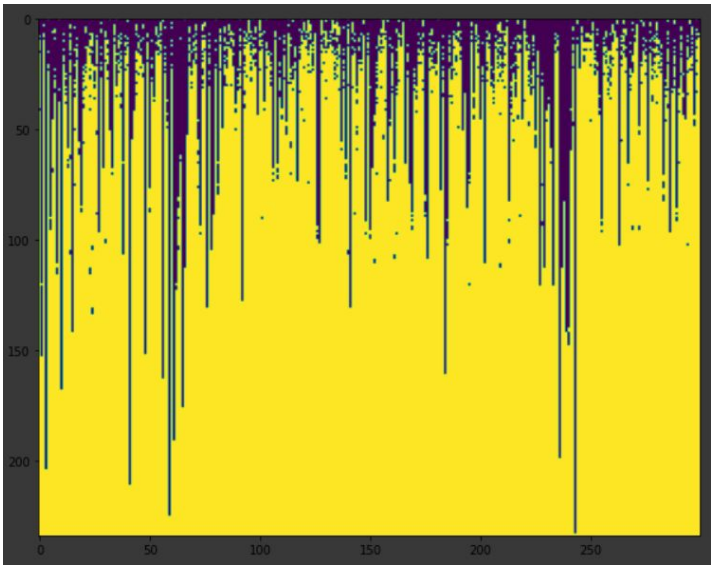
We have also created a graph to show the progress of the GA:

The Y-axis is the number of generation (0 is at the top).

The X-axis is the letter in the text.

if (x,y) is blue it means that in the best chromosome in generation Y, the X character is **not correct.** If (x,y) is yellow then in generation Y, the best chromosome Xth letter is **correct.**

It can be seen that in the first few generation almost all of the characters are wrong. very quickly most of the characters are correct and the majority of the work is just to align the last few letters. This also can be seen in the fitness graph from before. most of the work is done in the first few generations and most of the time is “wasted” on just reaching the 100% percent.



3.4 Discussion

Finding the best hyperparameters can be a challenging task, trying different variations of hyperparameters is critical because we saw how a little change impacts dramatically on the performance.

The brute force for this method did not converge in a reasonable time and therefore it can be determined that the GA is better for this task as well.

4. Conclusion

In this task we were asked to implement a genetic algorithm for 2 different optimization tasks.

The difference between these two tasked was mainly in three areas:

- The hyperparameter tuning
- The fitness function
- The chromosome representation.

The core was the same, a genetic algorithm.

In both cases the GA has proven itself better than brute force, and the wider the possible space for answer search - the more this difference was visible.