Genetic Algorithm for Image Optimization

part 1 and 2

What we get when we run the code .py file

1) Displaying the graph of the search space according to the number of pixels, the image type, and weather the image has been simplified *(2m 3s)*

2) Running the model of part 2 *(7m 48s)*

3) Running the first improvement of our model in part 3 *(16s)*

4) Graph of the evolution of the final fitness score when we increase the hyperparameter k (we run a model for each k) *(3m 52s)*

5) Our best model that approximates file_jpg *(24m 2s)*

6) Our best model that approximates file_png *(3m 9s)*

7) Part 4 : new model approximation for file_jpg *(3m 11s)*

8) New model approximation for myself1.jpg *(4m 13s)*

Our project started with a genetic algorithm which **was designed to optimize images by adjusting individual pixel values to imitate an image** which in this case were two: the Esade logo with and without color, which we did during Part 1 and Part 2 of the project .

The genetic algorithm use various essential functions for its correct implementation and running:

- **load_image:** in charge of the loading and resizing the images according to a specified maximum size but maintaining the initial aspect

- **simplify_color:** used for color simplification, reduces the color complexity of images by converting pixel values to either 0 or 255 creating a binary like image representation

- **compute_genotypes:** used for genotype calculation by calculating the unique colors present in the image and their frequencies setting the stage for genetic diversity

- **generate_random_image:** in charge of random image generation based on the calculated genotypes and their probabilities that contribute to the initial population in the genetic pool

- **compute_search_space:** used for search space evaluation where we assessed the complexity of our optimization problem with it, basically gives us the potential number of unique solutions

- **initial_population:** in charge of initial population creation, generating a diverse starting population of images, each with a potential to evolve towards the target image

- **selection:** used for parent selection, potential parent images were chosen based on their fitness scores based on their similarity to the target image

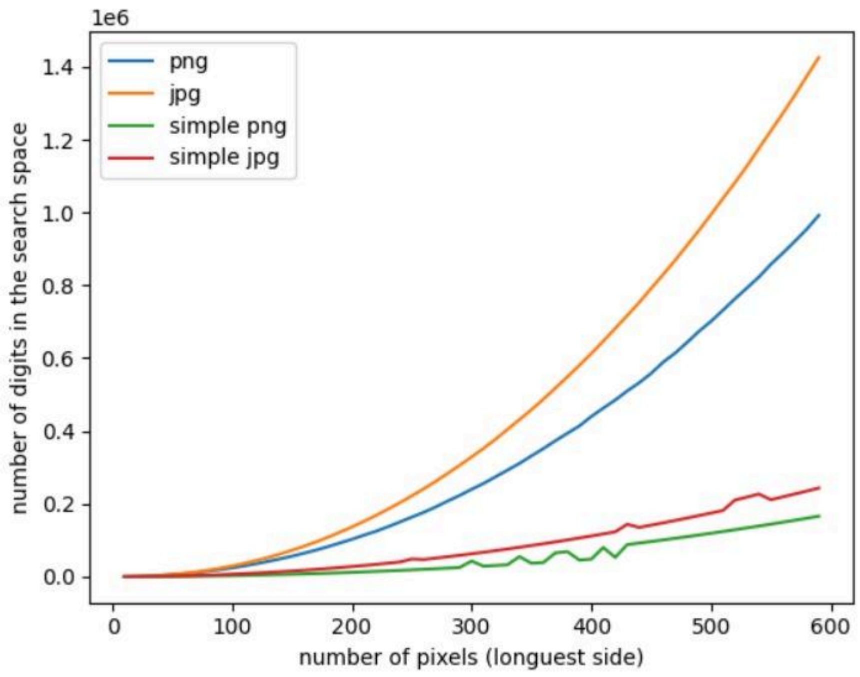
- **crossover and mutation:** we combine and altered the genetic information of parent images to produce various children

- **replacement:** in charge of population replacement, updating the population with new generations of images, discarding less fit individuals

During our project's progression, **we encountered another computational challenge while plotting the search space graph**, which involved handling very large numbers that the computer could not process meaningfully. **We discovered a solution through the "math" library's "log10()" function, which can return the logarithm base 10 of large numbers.**

By applying "log10()" to the search space calculation, we were able to represent the big search space values as their logarithmic equal value. **This enabled us to plot the search space graph on a log10 scale which means we could handle the exponential growth of the search space with increasing image resolution.**

The updated graph successfully visualizes the big differences in search space between the original image and their versions which are simplified. As the graph is scaled to a log10 basis, the values should be interpreted as the number of integers of the search space number.



This method not only resolved our plotting issue but also improved the explainability of the data, as it directly relates to the search space of the size of the images (because we don't use a wrong formula).

<u>**We can extract 3 pieces of information from the graph**</u>

- the <u>higher </u>the **number of pixels**, the <u>higher </u>the **search space** (exponentially)
- The search space of the **simplified** image is way <u>lower </u>than the one of the **original image**
- the search space of the **jpg** image is <u>higher </u>than the one of the **png** image

With this in mind, we move into Part 3 of the project using this information to better tune our hyperparameters regarding the number of pixels that we deal with as well as the image that we are trying to approach (png or jpg) : **enhancing the genetic algorithm's functions for better performance and efficiency**

part 3

1) Improvement of the mutation function in terms of time complexity

hyperparameters baseline :

- picture : file_png
- maxsize = 100
- number of iteration = 2000
- population size = 30
- cross = True
- replacement rate = 100%
- mutation rate = 0.1%
- k = 30

We understood by keeping track of both the total genetic algorithm time and the mutation time that mutation time is more than 95 percent of the total time of execution of the genetic algorithm. So we have put our main focus on this function.

The initial mutation function :

In this function

- We go through each pixel, if a random number between 0 and 1 is lower than the mutation rate, mutate the pixel with a randomly chosen genotype (that we randomly choose at the moment)

```
total time : 7m and 34s
total mutation time : 7m and 18s
```

The first version of the improved mutation function :

In this function :

a) we calculate first all the possible indexes of the image to get all pixels in the variable *all_indexes*
b) We take a sample from all these pixels (equal to the mutation rate times the number of pixels). This way, we have the proportion of mutated pixels that we want according to the mutation rate.
c) going through all the sample pixels to make them equal to a randomly chosen genotype (that we randomly choose at the moment)

```
total time : 7m and 59s
total mutation time : 7m and 42s
```

The first version of the mutation function improvement is not efficient. But its constitution will help us develop a very fast model

The second version of the improved mutation function :

first change (on a))

After a moment of reflection, we realized that there is no need to calculate the *all_indexes* variable for each image because it is the same across images and across generations (as it is just the list of the indexes of all possible pixels). So we had the idea of calculating it only one time at the beginning of the algorithm and put it as a parameter of the mutate function, and stop calculating it each time we call the mutate function. This has been a significant improvement in terms of time complexity as we went from 8 minutes to 5 minutes (with the baseline mentioned before).

second change (on b))

Because the mutation function is repeating itself many times, we want to reduce its job as much as we can, but there is no other work that can be done only one time outside the function. Then, we had the idea of playing with "fake" randomness, which means that we only generate one random pixel mutation sample (which is a bunch of indexes that indicate where the mutation should happen) for a generation, and then each image uses this sample differently. What does that mean? In fact, each image will add to the pixel mutation sample (which is actually represented by an array of integers) a number according to its position in the population.

ex : for *image number 5 we add 3*5 to the array (pixel mutation sample)*
*for image number 7 we add 3*7 to the array (pixel mutation sample)*
*for image number 13 we add 3*13 to the array (pixel mutation sample)*
(the number 3 is of course arbitrary)

This way, we don't mutate the same pixels for all images of the population.

A new real random pixel mutation sample will be created for the next generation (but then we repeat the same "fake" randomness process)

We have to note that in parallele of creating pixel mutation sample to indicate where the mutations should occur in terms of pixel location, we also create a random list of genotypes of the same length as the pixel mutation sample so that we can put the genotype for each pixel of the pixel mutation sample

Now we have the same effect as if we would generate a random array for each image of the population, but taking way less time.

Only the second version of the improved mutation function is in the .py file. its name is mutation_improved *third change (on c))*

The last change is just about using numpy broadcasting to put the genotypes in their right pixel place instead of going through a loop. (the goal is again computational efficiency)

The changes in the whole genotype algorithm resulted by the second version of the improved mutation function are implemented in the genetic_algorithm_improved1 function.

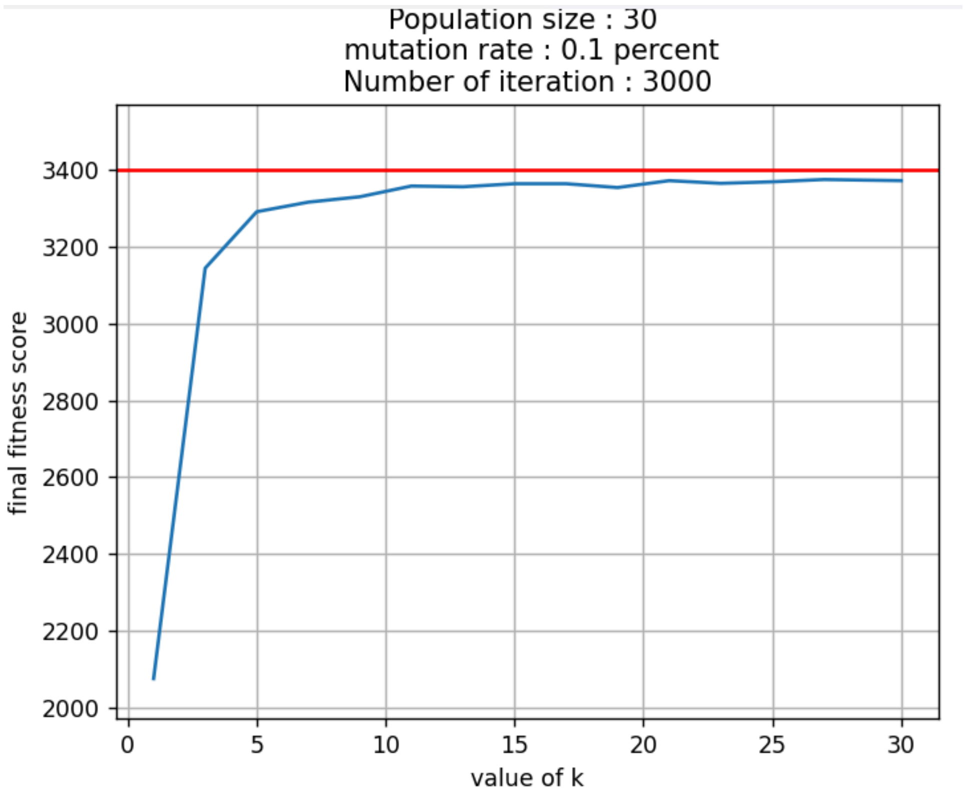
2) searching for the best hyperparameter k as a general rule (in terms of final fitness result)

By intuition, our hypothesis was that the optimal k is the *population size*, meaning that the best selection is just the best score image repeated *population size* times. This is because we think there is no need to explore less fitted images, as the fitness score is 100% correlated with the potential of the image.

To find out, we have run the genetic algorithm 30 times, each time with a new value of the hyperparameter k. Our results are represented in the graph below

hyperparameters baseline :

- picture : file_png
- maxsize = 100
- number of iteration = 3000
- population size = 30
- cross = True
- replacement rate = 100%
- mutation rate = 0.1%
- k = [1, 2, 3, …,30]



As we can see, the result score starts being constant from k=10 until k=30 increasing a little bit. We can conclude that our hypothesis is confirmed, that the best k hyperparameter is in fact the *population size* (in this case 30).

3) Improvement of the selection function in terms of time complexity

hyperparameters baseline :

- picture : file_png
- maxsize = 200
- number of iteration = 5000
- population size = 30
- cross = True
- replacement rate = 100%
- mutation rate = 0.1%
- k = 30

The initial selection function :

- In this function, for *population size* number of time, take *k* random images in the population and selects the one with the highest fitness score

```
total time : 0m and 58s
total selection time : 0m and 21s
total mutation time : 0m and 13s
total crossover time : 0m and 7s
```

The improved selection function :

As we noticed in the previous section that the best *k* for selection is *population size*, this means that the new generation of parents should be just *population size* copies of the best image of the old generation.

- To make this faster in case *k* = *population size*, the computer automatically takes the best score image and replicates it *population size* number of times to create the new parent generation

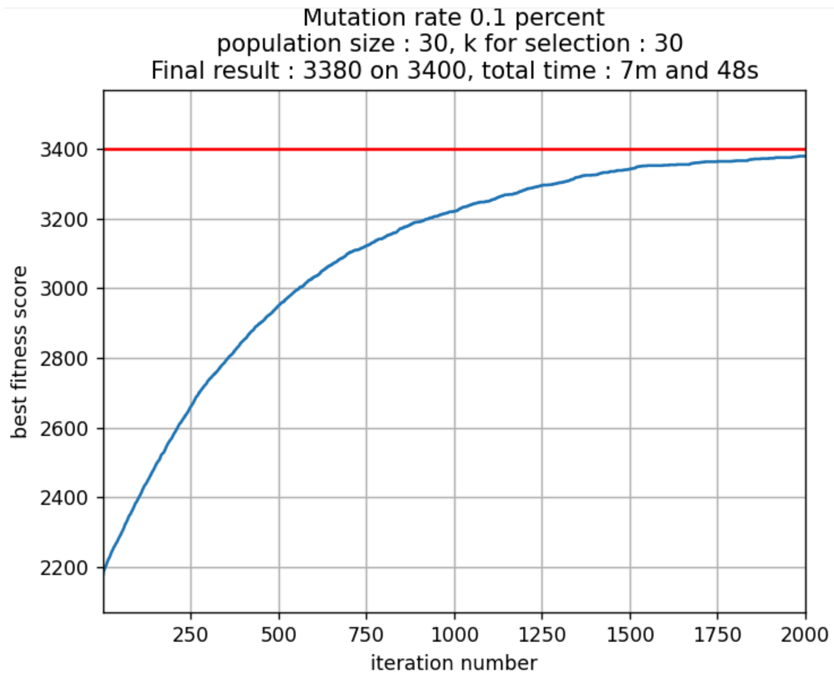
```
total time : 0m and 44s
total selection time : 0m and 8s
total mutation time : 0m and 12s
total crossover time : 0m and 7s
```

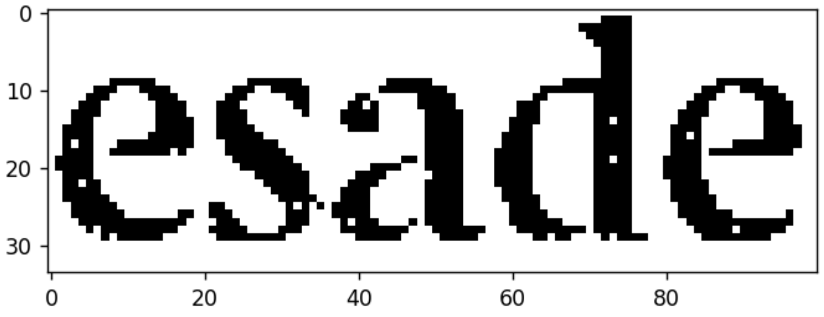
We have significantly improved the time complexity of the selection function (in case k=population size)

4) Difference between not improved and improved genetic algorithm

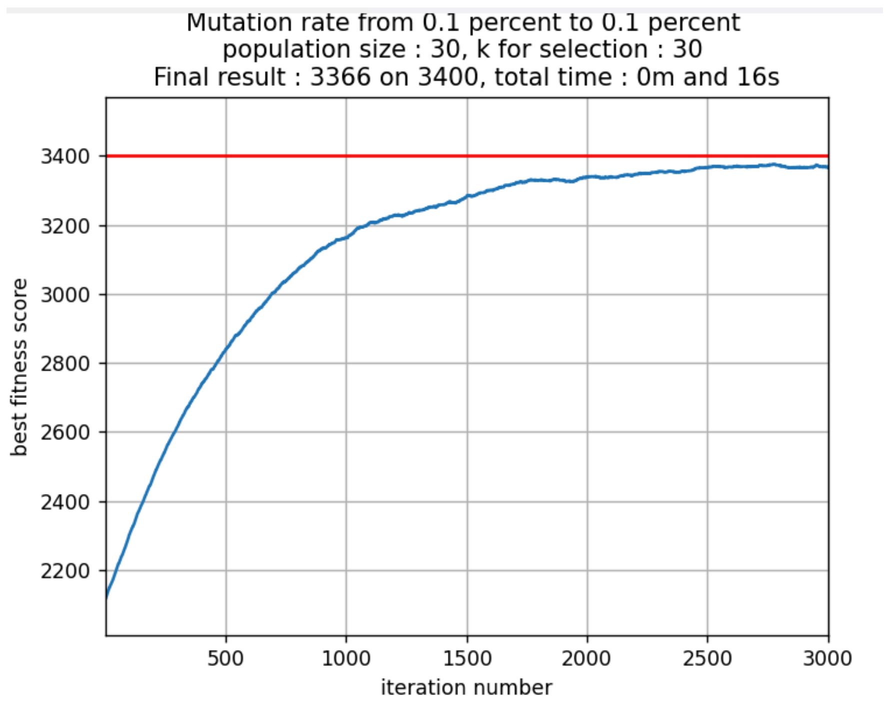
We have been doing enough change to look at the differences between our initial genetic algorithm (the one of part 2) and the current one in terms of execution time as well as result

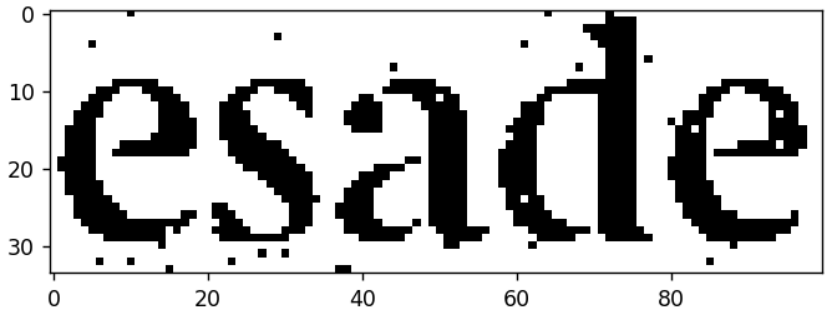
Genetic algorithm of part 2





Current genetic algorithm (using improved mutation function as well as improved selection function)





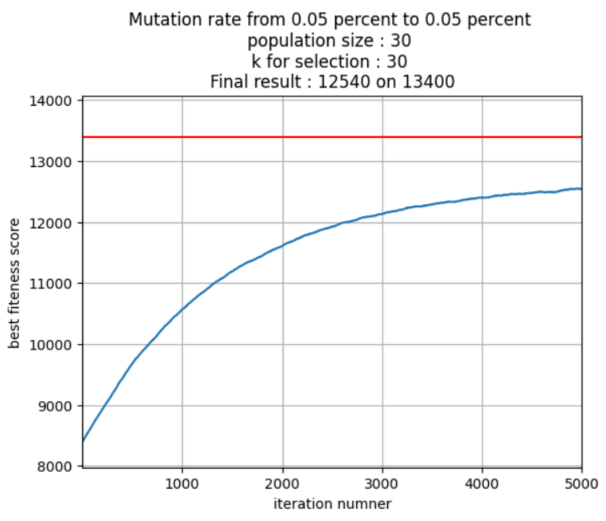
Basically we can do in 16 seconds what we used to do in 7 minutes and 48 seconds

5) Understanding the impact of the population size on execution time and final fitness result

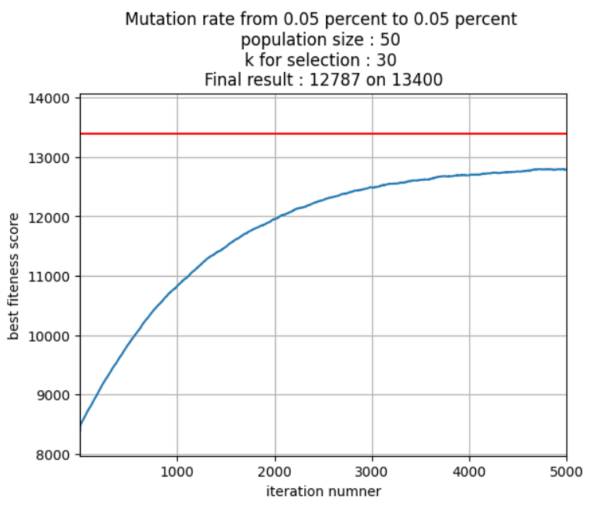
hyperparameters baseline :

- picture : file_png
- maxsize = 200
- number of iteration = 5000
- population size = [30, 50, 80, 120]
- cross = True
- replacement rate = 100%
- mutation rate = 0.1%
- k = population size

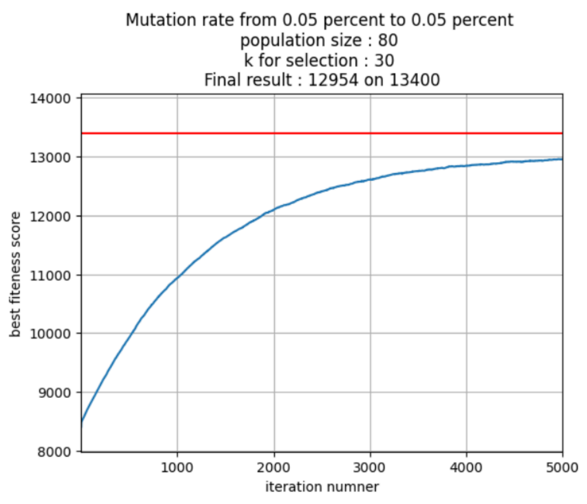
`0m and 45s`



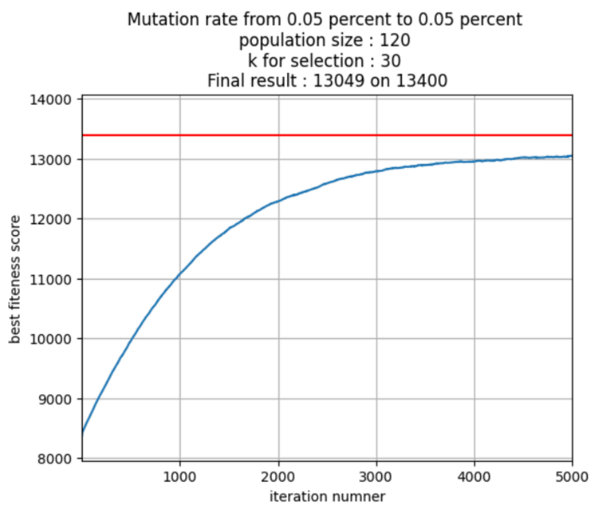
`total time : 1m and 26s`



`total time : 2m and 14s`



`total time : 3m and 19s`



Taking into account the computation time and the final fitness score result, we have a better idea of the behavior of the population size parameter. A higher population means usually a better final score but also a higher execution time

6) Searching for the best mutation rates (in terms of final fitness result)

first improvement (genetric_alogithm_improvement1)

better efficiency in terms of mutation time

developed in part 1 of this document (in the part 3 of the project)

Make the mutation rate change with time

In order to privilege exploration at the beginning and exploitation as we approach the final result, we had the idea of decreasing the mutation rate with time. So there will be a lot of mutations at the beginning (to privilege exploration), but they will decrease with time in order to exploit more the population that we have.

We have not performed a specific study to know which mutation rate to set. This is because it depends on too many parameters such as the goal image, the number of iterations etc. And the fact that we deal with changing mutation rates involves that we can not test out too much. So after making several tests we developed a small intuition on how to tune this hyperparameter according the the number of iteration (for example for a test with a very high number of iteration we would decrease more the mutation rate to favoritise exploitation as we would be approaching the result for a longer phase). This way we applied genetric_alogithm_improvement1 to *file_png*

breaking condition

We created a new parameter named breaking_condition, to make the algorithm stop when there is no more improvement in terms of fitness function. This way we earn time to make more test, this has been helpful especially in the context of finding the best parameters for *file_png*

second improvement (genetic_alogithm_improvement2)

Make the mutation rate change with fitness function (not with time)

Then we found a better way to make the mutation rate change during the process of the algorithm, but without worrying about too much parameters (r_mut at the beginning and r_mut at the end)

By intuition, in the process of a genetic algorithm, the farest the current score is from the goal score, the more we want to privilege exploration (more mutations).

And vise-versa, the closer we are from the goal solution, the more we want to privilege exploitation (less mutations).

By understanding that the distance from the goal score should be proportional to the mutation rate, we can build a model where the mutation rate changes according to the distance to the goal for each iteration. The proportionality between distance and mutation rate will be the variable distance_to_r_mut_ratio.

The distance will be expressed as a percentage of the goal score

A study have been made on the graph of a previous normal test to give a value for distance_to_r_mut_ratio, we found that 750 is reasonable

distance = 1-(current score / goal score)
distance_to_r_mut_ratio= 750
r_mut = distance / fitmut

This way, we calculate r_mut in each iteration

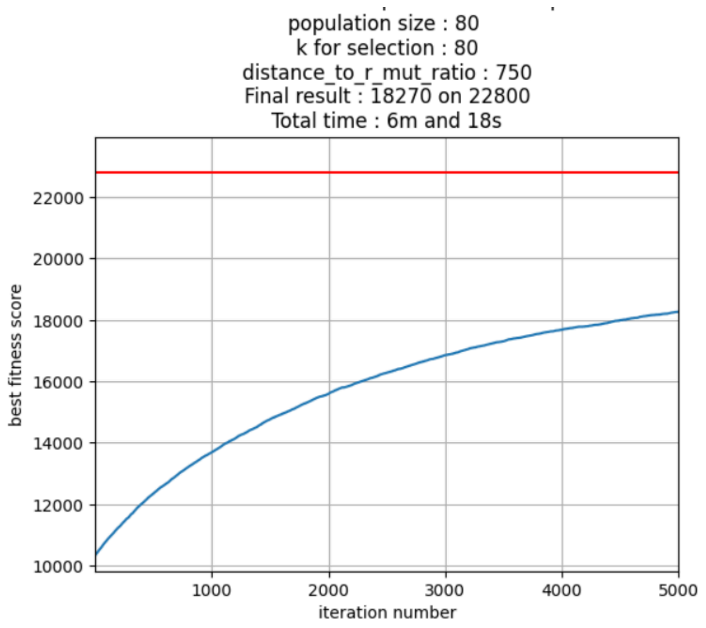
In order to have more exploitation at the beginning, we don't use the r_mut computations until the exploitation threshold is reached. In the iterations before the exploitation threshold, we explore according to a r_mut equal to 0.05%

This adaptive mutation rate has been especially helpful in the context of finding the best parameters for *file_jpg*. The results were satisfying without a very high time of execution

7) searching for the best hyperparameters for the final model we will apply to file_jpg

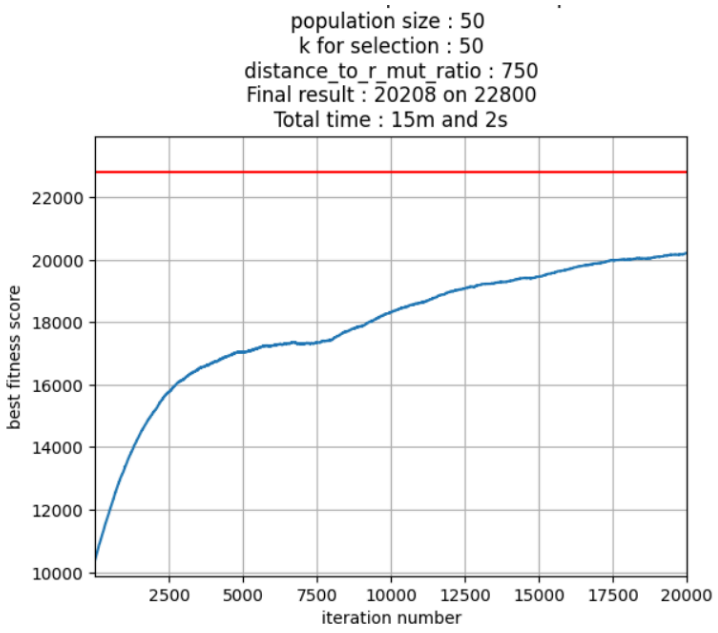
the resolution we chose was 200, of course we used the second improvement of genetic algorithm

<u>step 1</u>



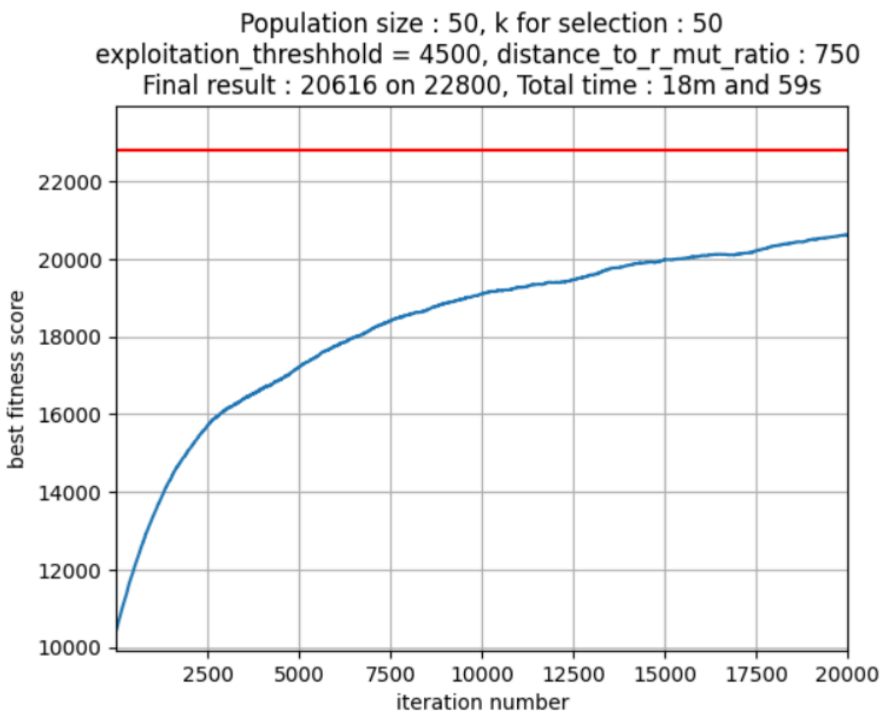
<u>step 2</u>

- reduction of the population from 80 to 50
- increase of the number of iterations from 5 000 to 20 000
- implementing an exploitation threshold equal to 8000



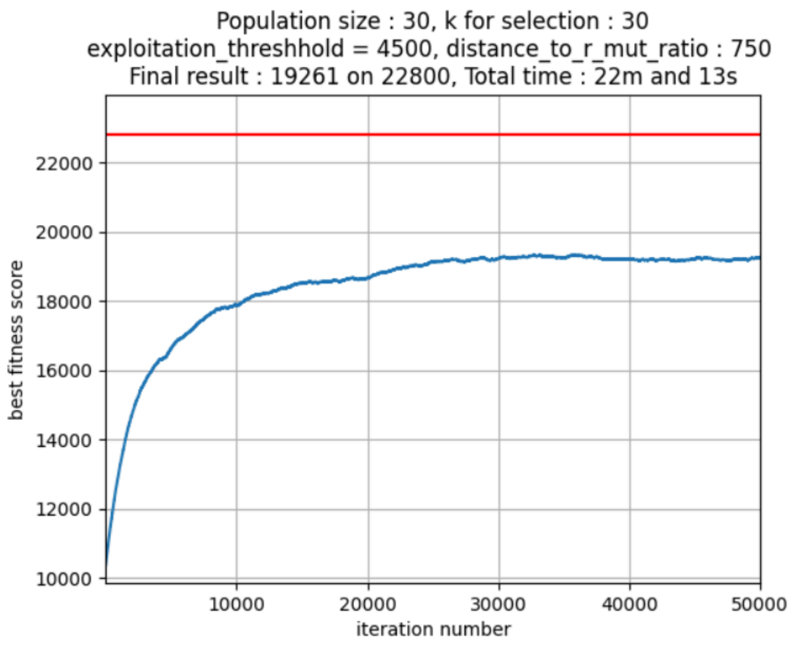
step 3

- changing the value of threshold of exploitation to 4500



step 4

- testing with a population of 30, but with more iterations (50 000), to compare fitness result



We will keep the population of 30 as we can have more iterations for about the same amount of time. However, the score starts being constant across generations.

step 5

- testing new distance_to_r_mut_ratio : 500 (higher r_mut → more exploration)

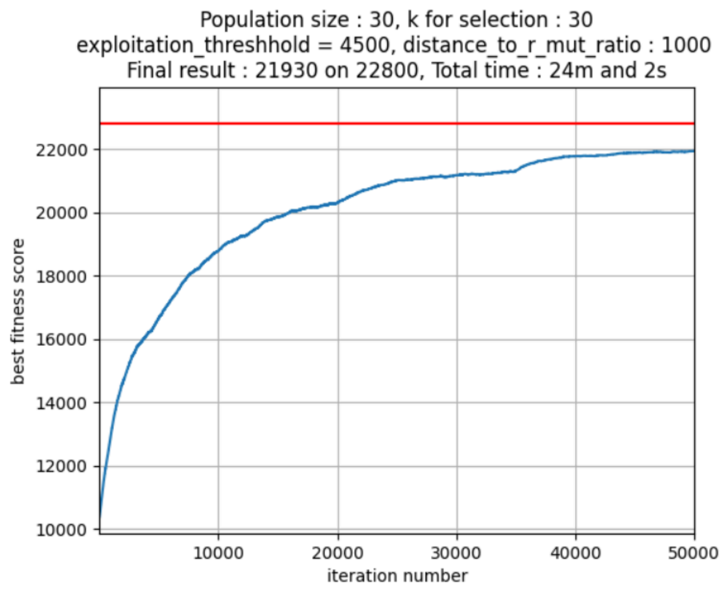
In the prints giving updates on the execution, we can see :

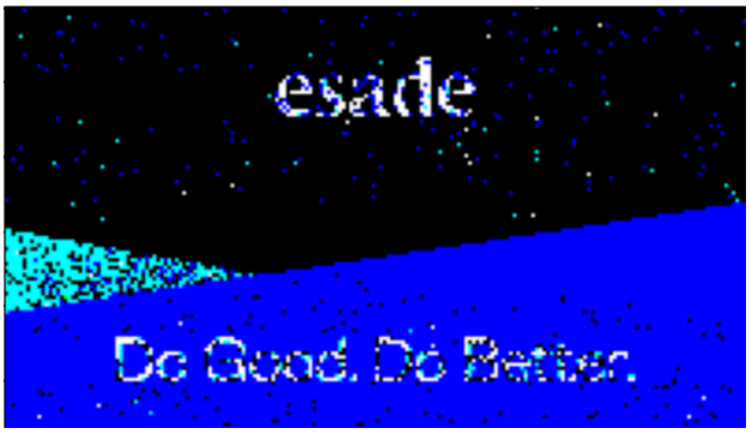
```
iteration : 11211 on 50000
best score : 16443, target : 22800
mutation rate : 0.0558 percent
```

As the score is very low for the 11 000th iteration, we will stop the execution and consider other changes

step 6 (final result)

- testing new distance_to_r_mut_ratio : 1000 (lower r_mut → more exploitation)



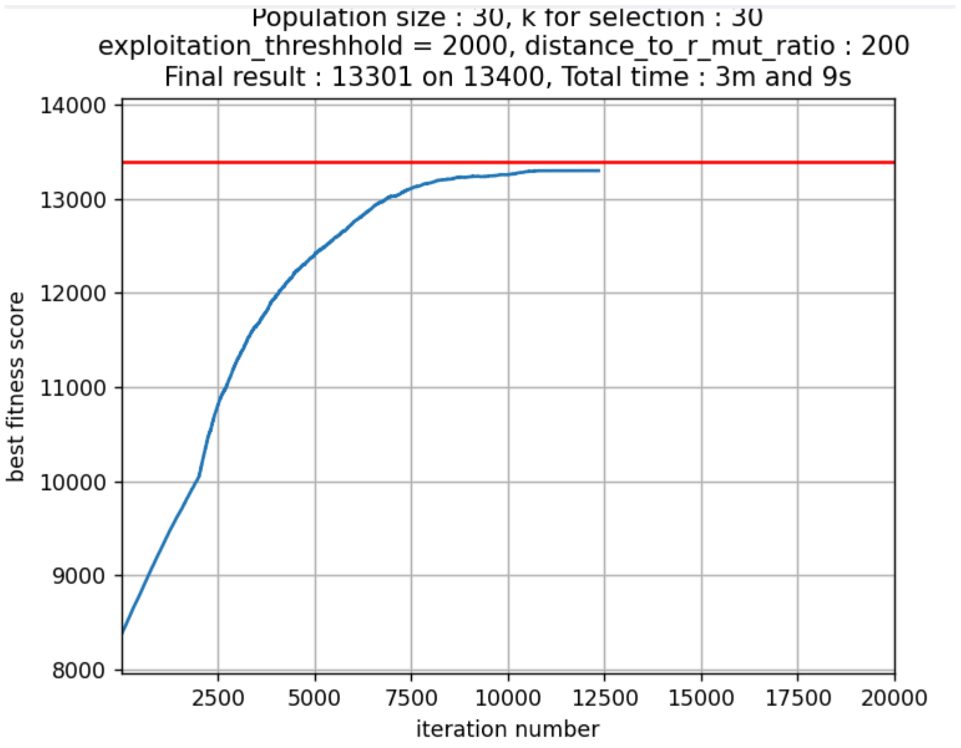


the we run the algorithm for one or two hours with the same hyperparameter baseline, we should get an image very close to the goal image (file_jpg)

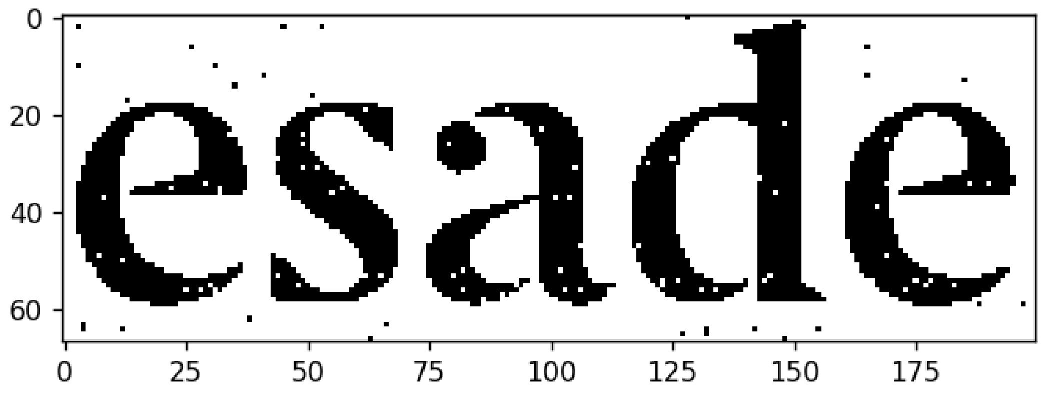
7) Running the model for file_png

Here, we tried to find the best hyperparameters for file_png the same way we did for file_jpg. We have been tuning the exploitation_threshhold as well as the distance_to_r_mut_ratio.

Setting the breaking_condition to “True” was very helpful as it stops the algorithm when there is no more improvement in the process. This way, we don’t lose time while tuning the hyperparameters.



The algorithm has stopped at the 12000th iteration because there was no more improvements (the breaking condition stopped the execution)



Group Management Document Group: Julia Vidal, Lucia Tortajada, Adam Aqasbi, Víctor Pérez and Gael Mensa

We, the undersigned members of the group for the Genetic Algorithm assignment, would like to report that our group work was conducted smoothly and efficiently without any major issues or conflicts.

Throughout the project, we always maintained communication active, which enabled us to identify and resolve any minor issues that arose promptly. We also established a clear plan for the project's progress and ensured that everyone was kept up-to-date with the latest developments.

Our group is satisfied with the results and believes that we have achieved our learning objectives. We would be happy to work together on any future projects and assignments as well.

· - Víctor Pérez has given all 10 to every member

· - Julia Vidal has given all 10 to every member

· - Gael Mensa has given all 10 to every member

· - Adam Aqasbi has given all 10 to every member

· - Lucia Tortajada has given all 10 to every member

Part 4

1) How does this new algorithm work

This new model is not actually a genetic algorithm, because there is no population and selection of the best individuals. However, it is certainly an evolutionary algorithm. How does it work ?

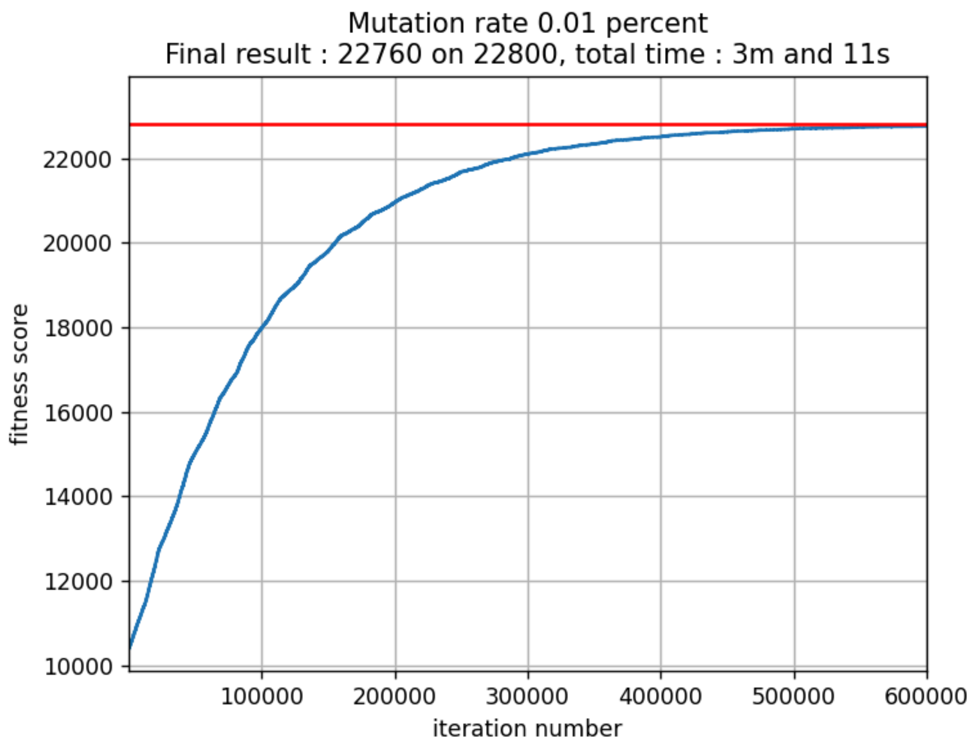
It is actually very simple :
we start with a random image

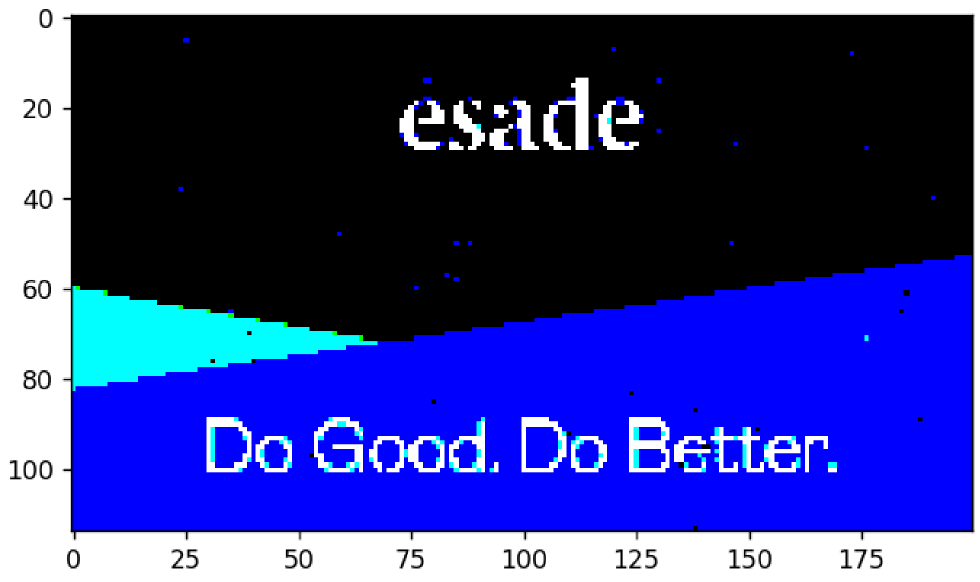
- We perform mutation on it.
- If the fitness function of this new mutated image is better than the fitness function of the image before mutation, we keep it. If not, We delete the new image and keep the old one

And we keep the process for n_iter number of iterations.

We can consider it as an evolutionary algorithm as the individual evolves by learning from its mistakes

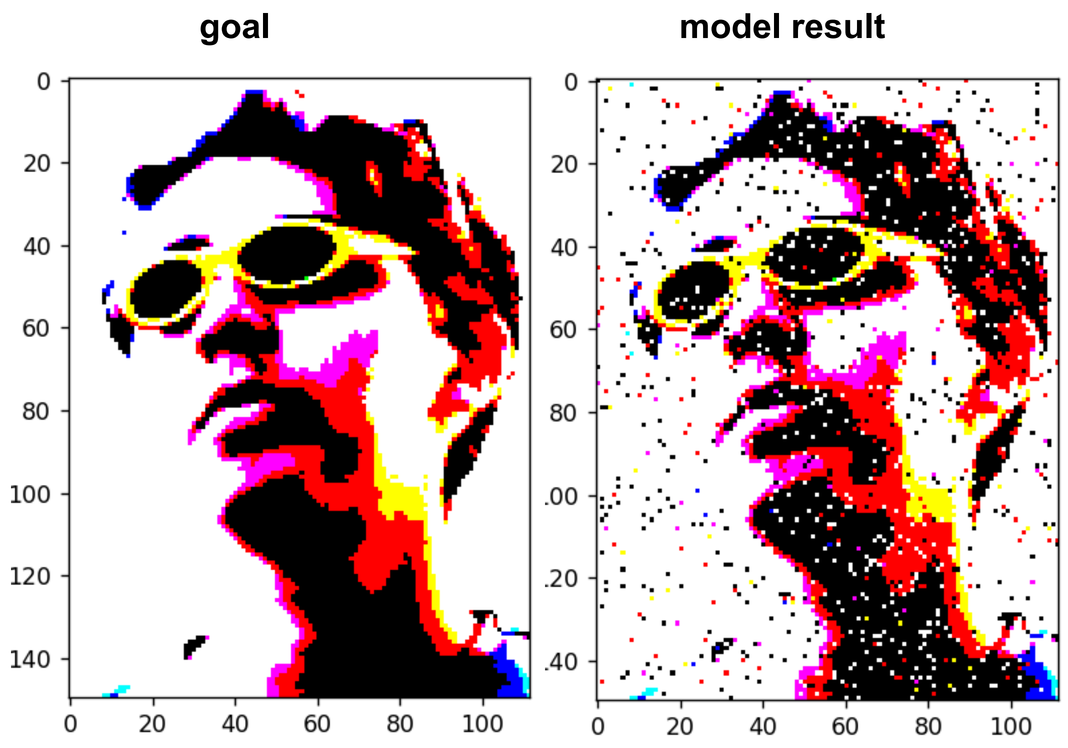
2) Implémentation

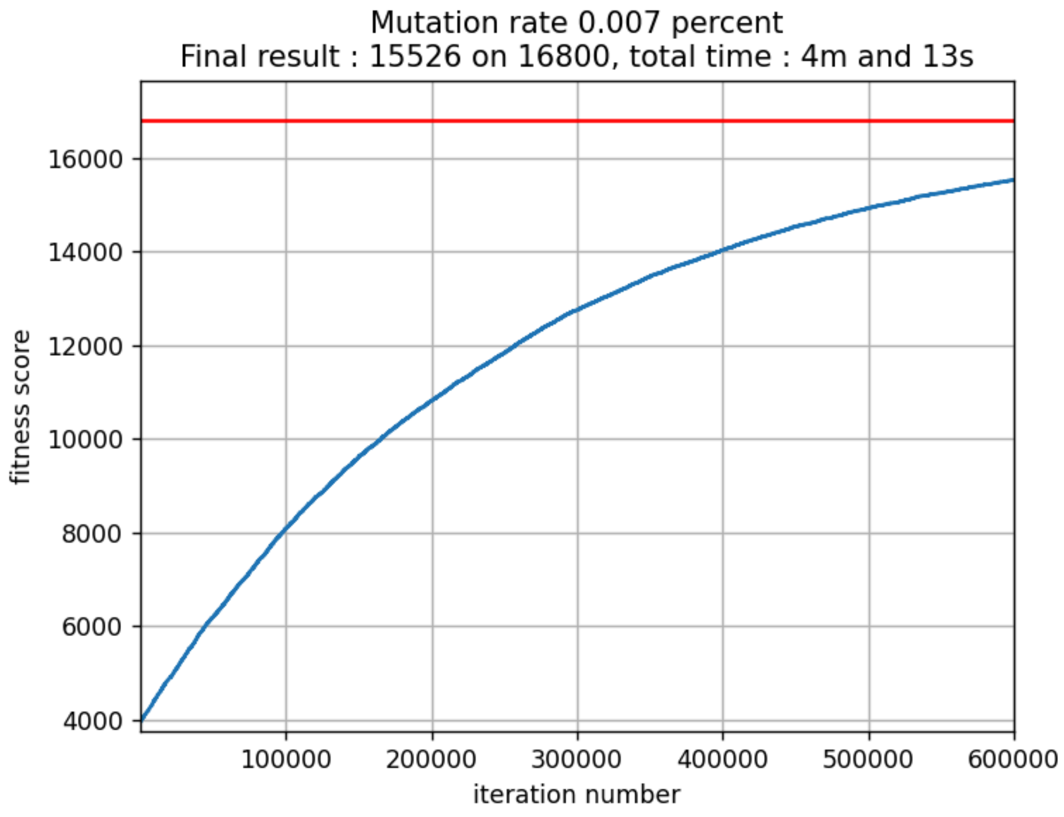




So we can archive a significantly better result than in part 3 (an almost perfect result in fact), in <u>3m and 11s</u> instead of <u>24m and 2s</u>

3) bonus test


goal
model result
0
20
40
60
80
100
120
140
0
20
40
60
80
100


Mutation rate 0.007 percent
Final result : 15526 on 16800, total time : 4m and 13s
fitness score
iteration number
16000
14000
12000
10000
8000
6000
4000
100000
200000
300000
400000
500000
600000