Computational Intelligence for Optimization

Image Recreation

Team Rocket

May 2024

https://github.com/Bwdo77/CIFO.git

Eduardo Costa, 20201536

Gonçalo Reis, 20201534

João Martins, 20201567

Rodrigo Martins, 20201583

Prof. Leonardo Vanneschi | Prof. Berfin Sakallioglu

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação

CONTENTS

1. INTRODUCTION

The Image Recreation problem is simply the recreation of an image having the original one, this can be achieved in 2 different ways. The first one is when we have all the correct pixels from the beginning, but they are scrambled, the objective is to try to put the correct pixels int their place. The second approach is to have a set of random pixels and slowly change those pixels until they converge to the colors of the original image.

We believe that the second approach was more interesting and would result in a more interesting visualization of the evolutionary process that occurs during this conversion of the image.

The only thing that was left was to choose the image that we wanted to recreate, for this we created a simple Pikachu sprite (20x20 pixels) [Figure 1] in order to apply the same image for all the various intermediate implementations that we will experiment with.

1.1. DIVISION OF LABOR

Instead of making a clear division of tasks we tried to do the entire project together and discussed through the project what would be a good approach at each step of it, together as a group. By doing so we would all have a good grasp of the project in its entirety and would participate in its various steps.

2. INDIVIDUALS

A clear definition of the individuals that compose the population is crucial in these problems, luckily for us, in our case it was quite simple to reach a clear and effective representation of the image. Each individual would be represented by a matrix of N * M pixels, being N the length of the image and M the height of it.

By doing so we were able divide the initial image into their pixels, which were simply represented by an array with length 3 with values from 0 to 255. These values would then represent RGB values associated with each pixel.

Therefore, an individual would be represented by a matrix of N by M arrays, each one of them containing 3 values.

Since our image had 20x20 pixels, it would have 400 arrays. Since each array has 3 values, each one of them with 256 different values, we would have a total of 400 * 256 * 256 * 256 = 6 710 886 400 different possible individuals in our population.

3. FITNESS ALGHORITMS

Regarding the fitness algorithms, we only found one plausible way of verifying if an individual was closer to the original image, and that was by using distances. Our idea was that we could use the representation of the RGB values as vectors, and therefore verify the distance between these vectors.

By doing so, we would be able to create a minimization problem were we would try to make the sum of all distances of the pixels that composed an individual, when compared to the original image, be 0.

3.1. EUCLIDEAN DISTANCE

The Euclidean Distance was our first approach on the problem, it is a widely well-known way of measuring distances, and it was simple to implement. The idea here was to simply sum the square of the differences of each RGB value and then make its square root.

3.2. SQUARED EUCLIDEAN DISTANCE

Then we had a good idea, to simply make the square of the fitness. This would be quite useful to penalize pixels that would be further apart, this will be useful for the selection algorithms that care about the differences in fitness, such as the Roulette Selection.

3.3. MANHATTAN DISTANCE

Another distance measure that we believed was interesting to try was the Manhattan Distance. Which would simply be the sum between the absolute values of the differences of each RGB value.

3.4. SQUARED MANHATTAN DISTANCE

Similarly to what we decided on the Euclidean distance, we will also be using a squared Manhattan for the same reasons stated above, to penalize pixels that are further apart more severely.

4. SELECTION ALGHORITMS

When it comes to the selection algorithms we tried 3 different approaches, each one of them having always in consideration the 3 main characteristics of them:

- Selection is probabilistic.
- If f(A) is better than f(B) then the probability of selecting A must be higher than selecting B.
- No individual can have a probability of being selected equal to zero.

4.1. ROULETTE SELECTION

In this selection we give the probability of an individual being selected based on the differences in their fitness. We can calculate it by making the sum of all fitnesses and then dividing the fitness of each individual by the sum of fitnesses.

However, this would mean that a higher fitness would have a better probability, but since our problem is a minimization problem, we must the exact opposite, a smaller fitness must indicate a higher probability of being selected.

We solve this issue by normalizing the probabilities, meaning that we will do 1 - probability for each individual and then divide that probability by the sum of all probabilities.

4.2. RANK SELECTION

Similar to the Roulette Selection, the big difference is that the fitness gap between individuals is ignored, the one factor that matters is if a certain individual has lower or higher fitness than his peers.

In here we will attribute a higher ranking to the individual with the lowest fitness and continuing doing so until we reach the individual with the highest fitness.

Then we simply divide each rank by the sum of the total ranks to get the probability of an individual being selected.

4.3. TOURNAMENT SELECTION

Our final selection algorithm is the Tournament Selection, the main idea here is to select K individuals randomly, and with repetition, from the population. Then we simply select the best individual for the new population.

An important factor to consider here is the size of K, tournament size, which when bigger the probability of selecting a bad individual becomes less probable, however it takes more time to make a tournament. This

phenomenon is called selection pressure, and our objective is to select a tournament size where we get the best individuals consistently without with the smallest K possible.

5. CROSSOVER ALGHORITMS

For the crossovers we tried to find different ways of merging two parents in order to find 2 new offspring from their original values, by doing so we will be able to make small changes to the previous generation in order to try to find better individuals.

5.1. HALF-CHANGE CROSSOVER

In this first crossover, we will give for each RGB value a probability of swapping it between the parents. The default probability is 50% which could, in some edge cases, could indicate a complete swap between the parents or the inexistence of the crossover, resulting in a replication of the parents.

5.2. SINGLE-POINT CROSSOVER

Here we have the classic crossover, where we exchange genetic material between the parents to create 2 new offsprings. The main idea here is to select either 1 or 2 as the crossover point and then make a swap of either the Green and Blue values (crossover point at 1) or just the Blue values (crossover point at 2).

This indicates that the Red value will always be the same, which for one side might be a small problem, since we will never change that part of the genetic material, but on the other side it guarantees that we always have a crossover that is not a total replication of the parents.

5.3. ARITHMETIC CROSSOVER

Finally, we have also decided to use an Arithmetic Crossover, where, based on a parameter called alpha, this parameter will be multiplied by the values of one of the parents, and the inverse for the values of the other parent, in order to create offsprings that are based on both values of the parents, similar to calculating a weighted average of their RGB values.

6. MUTATION ALGHORITMS

The mutation is a crucial step of Genetic Algorithms, in here we will try to make a more disruptive approach where we will complete change some individuals, in some way, in order to create completely new pixels for some individuals, that, in combination with crossover, should result in finding even better individuals.

6.1. UNIFORM MUTATION

For the Uniform Mutation we have the simplest algorithm, for each RGB value select a completely new, and valid, result. This implementation is extremely disruptive, but with some luck, we might be able to find for some pixels a better solution than simply using crossover, since we will be introducing completely new values.

6.2. SHUFFLE MUTATION

In the Shuffle Mutation we decided to go with a not so disruptive approach, we will completely shuffle the RGB values of each pixel, which will result in the same values that already existed but in different positions.

It is important to note that the shuffle might end up being exactly the same as the original, which would result in a replication of the parent, not only that, but we must also take into account that if we have a pixel with the same values for all of its RGB values, like [10, 10, 10] this kind of mutation will not be beneficial at all.

7. GENETIC ALGORITHM

Our first approach was to try all different combinations of fitness, selection, crossover and mutation, however we quickly realized that this would end up being very difficult for 2 main reasons.

The first being that we have 72 different combinations, and the second being that we must run these combinations multiple times, since the algorithms are probabilistic, we must average those results of N runs in order to have a correct grasp of their performance.

We ended up doing a single run of the 72 combinations and saving the results into an Excel file (Combination.xlsx), which can be found in the same folder as this report. This gave us a first idea of the values that we were getting and helped us to make sure that everything was working as intended.

Due to these problems, we ended up deciding to separate the problem into 4 smaller problems, which would be to verify the best combination for each of our Fitness Methods and then, by normalizing them, finding out the best overall model based on its progression across the generations.

In order to do this, we would use the same default parameters for all the implementations and then run each of them 10 times, in order to get more correct results. Therefore, we made 100 individuals per population across 1000 generations. We also always used elitism and decided to go with a tournament size of 20%, which would indicate a size of 20 individuals, and an alpha, for the arithmetic crossover, equal to 0.5, which would guarantee that both parents have the same weight.

Finally, in order to find the best combination for each fitness method we used an A/B strategy, meaning that we would keep the same parameters for everything except for one of them, starting with the selection algorithms, then the crossover and finally with the mutation. Due to this we had to start all implementations with the same crossover and mutation, which were the Single-Point Crossover and the Uniform Mutation, with an 80% and 1% probability of happening, respectively.

This strategy would exponentially reduce the time that it would take to analyze the various combinations, even though it would still take an enormous amount of time and give us very good insights into what combination could be the best.

7.1. EUCLIDEAN DISTANCE

Based on the plots [Figure 2] we can clearly see that the best selection was the Tournament Selection, for the crossover, on the other hand, they all ended up being very similar, but the Single-Point Crossover was slightly better, finally for the mutation, we can clearly see that the Uniform Mutation gave better results.

7.2. SQUARED EUCLIDEAN DISTANCE

Similarly to what happened before [Figure 3], the best selection was the Tournament Selection, and the best mutation was the Uniform Mutation, the big difference is in the crossover, where both the arithmetic and Half-Change presented better results. We ended up choosing the Half-Change since it had a worse initial population but was able to catch up with the Arithmetic Crossover.

7.3. MANHATTAN DISTANCE

Very similar to the Euclidean Distance [Figure 4], we end up using the Tournament Selection, Single-Point Crossover and Uniform Mutation.

7.4. SQUARED MANHATTAN DISTANCE

Once again [Figure 5], we use the Tournament Selection, Single-Point Crossover and Uniform Mutation, however, it is important to note that the results from Shuffle Mutation are way closer to the Uniform Mutation in this approach.

7.5. DOUBLE AND WEIGHTED TOURNAMENT SELECTION

Since we had such good results and all approaches used the Tournament Selection, we decided to try a couple variations of this selection.

The double tournament is a more robust tournament basically, we will do K tournaments to get K competitors, and then make one final tournament to get the best individual across all previous winners.

Even though it may sound like a good idea, due to the complexity of our problem, the time complexity associated with this approach made it unviable.

In the Weighted Tournament we apply a similar logic to the Roulette Selection, where we will give each individual a probability of being selected based on their fitness, instead of it being completely random.

7.6. BEST COMBINATION

We then verified the best individual of the best combination for each fitness, and the results were very good, in all 4 initial approaches we can clearly see the Pikachu [Figure 6].

However, this visualization of the final image was not enough to make a decision, therefore, we normalized the fitness values of all approaches and compared them [Figure 7]. AS we can see the Squared distances ended up being the best approach, and between them the Squared Manhattan seems to perform a little bit better. Another interesting thing to note is that the Weighted Tournament did not outperform the Squared Manhattan, this indicates that the randomness associated with the classic tournament is very important and not selecting the bets individual as often ends up being more beneficial in the long run.

7.7. OPTIMIZING CROSSOVER AND MUTATION PROBABILITIES

From the very start we have gone with quite small values associated with crossover and mutation; therefore, we tried to increase them and compare the final results to get the best parameters.

As we can see [Figure 8] small values of mutation and crossover are the best combination, we can also verify how disruptive a mutation is when compared with a crossover.

8. CONCLUSIONS

After testing various approaches we concluded that the best way of recreating an image using Genetic Algorithms is by calculating the fitness of an individual as the sum of the square of the Manhattan distance of all pixels to the original image, then using a Tournament Selection approach to select the individuals that will be present in the next generation, we should also use a Single-Point Crossover to guarantee that a exchange of genetic material occurs with 80% chance and also an Uniform Mutation with a chance of happening of 1%.

With all of this we made one final execution of this Algorithm with 20000 generations [Figure 9], and the final image as well as the progression of the evolution were extremely positive, picturing the Pikachu and making a clear distinction between the lighter and darker pixels.

This showcases that with more generations we would inevitably make a complete recreation of the original image.

9. APPENDIX

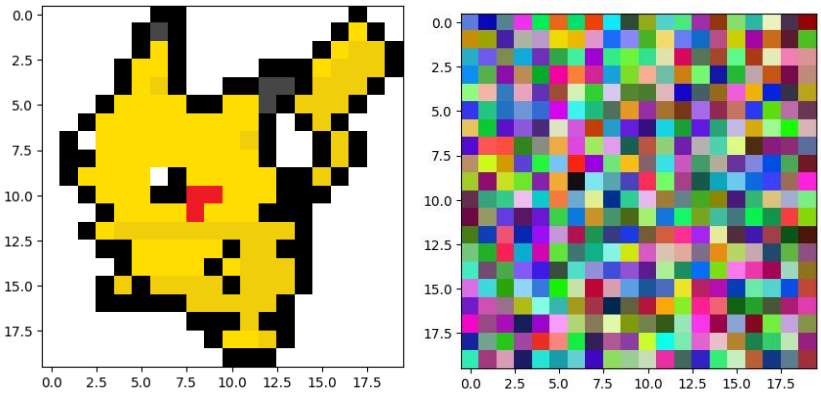

Figure 1 - Original Image and Initial Population

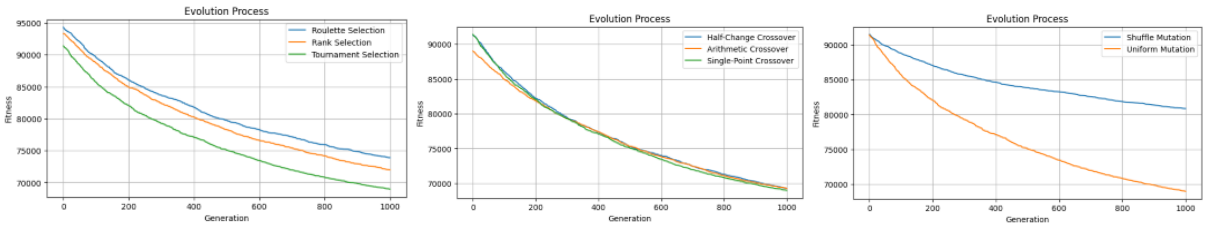

Figure 2 - Euclidean Distance Selection, Crossover and Mutation

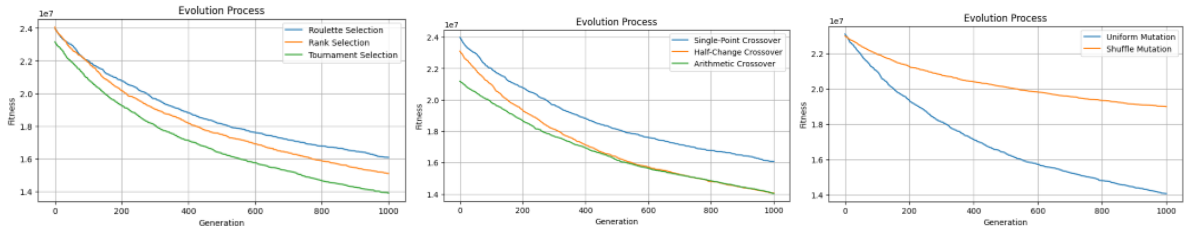

Figure 3 - Squared Euclidean Distance Selection, Crossover and Mutation

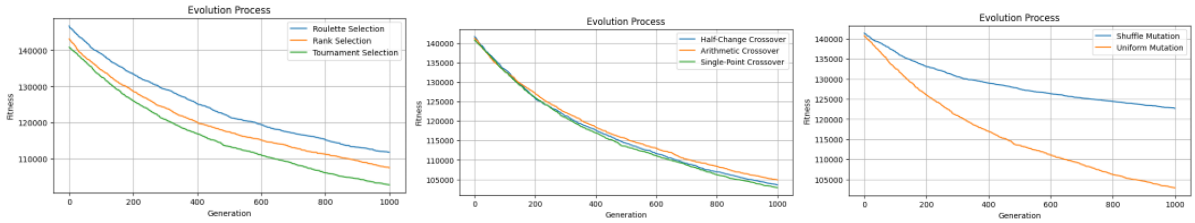

Figure 4 - Manhattan Distance Selection, Crossover and Mutation

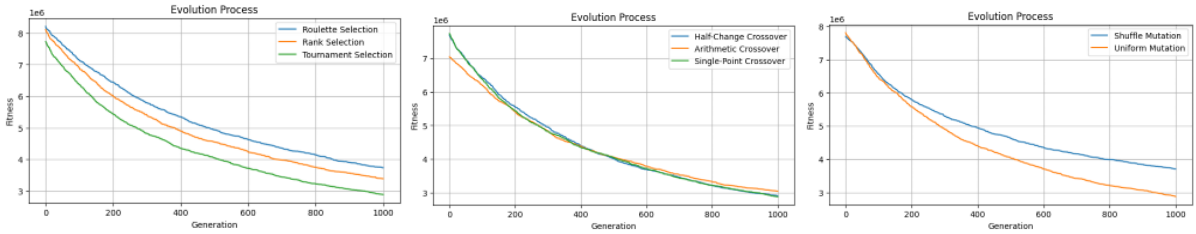

Figure 5 - Squared Manhattan Distance Selection, Crossover and Mutation

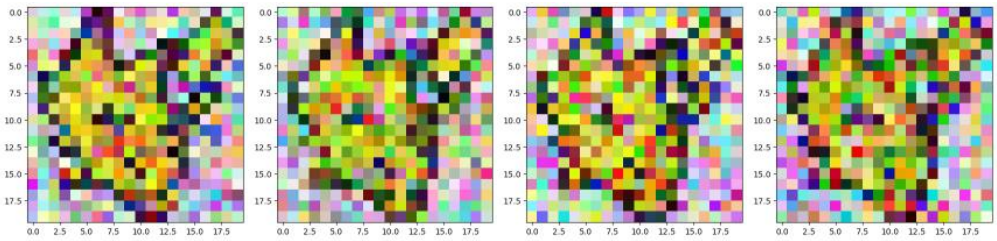

Figure 6 - Best Individual for Euclidean, Squared Euclidean, Manhattan and Squared Manhattan

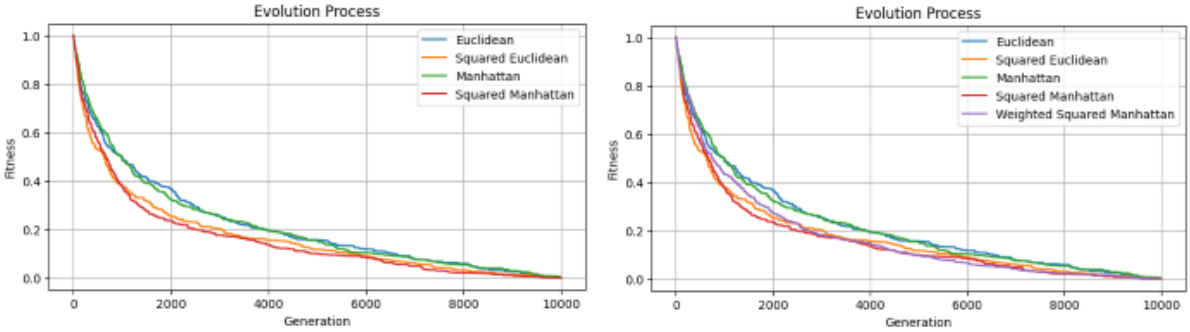

Figure 7 - Fitness Comparisons

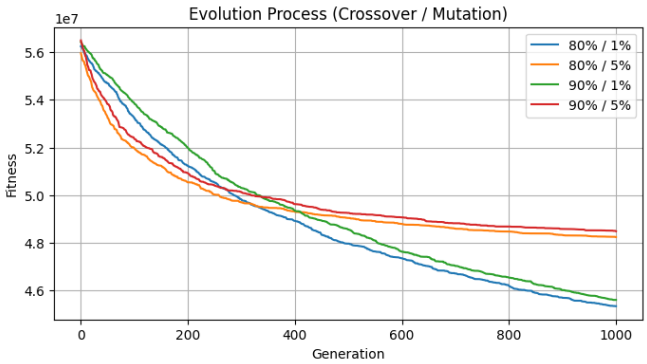

Figure 8 - Crossover and Mutation Probabilities Comparisons

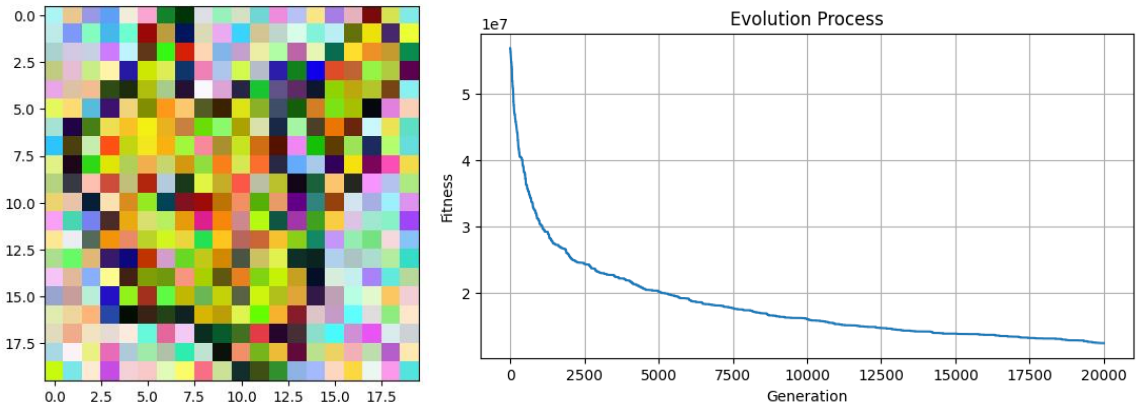

Figure 9 - Recreated Image and Evolution Process