# Image Recreation Using Genetic Algorithms

Ignacio Barquero Garcia
*Instituto Tecnologico de Costa Rica (ITCR)*
Cartago, Costa Rica
ibarqueroga@gmail.com

Paul Villafuerte Beita
*Instituto Tecnologico de Costa Rica (ITCR)*
Cartago, Costa Rica
paulvillabeita@gmail.com

***Abstract*—Genetic algorithms are inspired by the process of natural selection. These algorithms evolve populations with strategic actions until the target is reached, in this case, the algorithm is modifying and improving a group of initial randomly generated images until one of them reaches an acceptable fitness percentage in comparison with the target image.**

## I. Introduction

This document has the implementation of genetic algorithm in a program that tries to create a viable way to reach a target image from a image population. This all based in how genetic algorithms work, and how with the usage of simple mutation, selection and crossover actions, a random generated image will suffer strategic changes until it reaches the target image.

## II. Methodology

### A. Program Classes

*1) Fitness:* This class has 3 fitness algorithms related to the adaptability process and the comparison between each image with the target one. This class doesn't have a constructor, it was made mainly in order to have a well organized code. On ***"Part B, Fitness Functions"*** you'll find the implementation of these methods.

*2) Matrix:* Matrix class contains the representation of the initial random image (that would be modified in order to improve it) saved as a BufferedImage Java object [**?**]. This class has all necessary methods and attributes that will facilitate the image manipulation, as shown in the constructor

```
public Matrix(int _Width, int _height)
throws IOException{
    width=_Width;
    height=_height;
    matrix=randomMatrix();
    matrix=toGrayScale(matrix);
}
```

*3) Population:* This class has the necessary attributes for making operations on a population while the genetic algorithm is running. The **Mating Pool** is a Java ArrayList that allows making crosses. Basically, the elements of the actual population are inserted on the mating pool according to their fitness percentage, so, those images with high fitness are going to be inserted more times than the other ones. After generating the mating pool, two random images of it will be taken for a cross, and the fact that there are more fit images, increases the possibility of picking 2 of them.

```
public Population(int width,int hight
,int n) throws IOException{
population=new Matrix[n];
this.length=n;
generateRandomPopulation(width,hight,n);
}
```

*4) Target Image:* This class is very similar to Matrix class because it keeps stored as a BufferedImage the target image and the the function that converts it into a gray-scaled image.

```
public TargetImage(String path)
throws IOException{
    BufferedImage img =
    ImageIO.read(new File(path));
    matrix = toGrayScale(img);
    this.width=matrix.getWidth();
    this.height=matrix.getHeight();
}
```

*5) Useful:* This class has all the useful functions that the system need between the classes and we organize it in a class that could be used in any time that we need it. This class doesn't has a constructor because doesn't need it, because it doesn't represent an only object.

### B. Fitness Functions

*1) Euclidean Distance:* The Euclidean Distance use the two points to look for the distance between them and also with this we could look for the distance between the images looked as matrix.

```
public Matrix euclideanDistance
(Matrix img,TargetImage target){
double innerSum=0;
for(int i=0;i<img.getWidth();i++){
for(int j=0;j<img.getHeight();j++){
int aux1=img.getMatrix().getRGB(i, j);
int aux2=target.getMatrix().getRGB(i, j);
double resAux=Math.pow(aux1-aux2,2.0);
innerSum+=resAux;
}
}
double result= Math.sqrt(innerSum);
img.setFitness((int)(result));
return img;
```

```
}
```

*2) PixelIndividualComparison:* Algorithm adapted from the book *"The Nature of Code"* by Daniel Shiffman. It consists in comparing each pixel of the target image with the ones on the "in-process" image. If the pixel matches, the variable "score" increases by one. At the end, it returns the similarity percentage by multiplying the score by 100, and then, dividing it by the total of pixels of the image. Here is the implementation

```
public Matrix PixelIndividualComparison
(Matrix img,TargetImage target){
double score=0;
for(int i=0;i<img.Height();i++){
for(int j=0;j<img.Width();j++){
if(img.getRGB(j, i)==target.getRGB(j, i)){
        score++;
        }
    }
}
int totalPixels=img.Height()*img.Width();
score=(score*100)/totalPixels;
img.setFitness(score);
return img;
}
```

*Analysis*

$$(m+1*C)(n+1*C)(C^6)$$
$$\Rightarrow (m+1)(n+2)$$
$$\Rightarrow (m*n)(m)(n)(1)$$
$$\Rightarrow O(m*n)$$

*3) MyOwnFitnessFunction:* This function considers not only if the pixel on the same position in each image, but also the similarity percentage of the colors. This idea came out because for a computer two colors may be different, but for a human eye, this difference could being not be perceived. Keeping that in mind and for this particular case, there is no relevance with a minimum difference that the final user wont notice. This improvement, reduces significantly the number of iterations to reach an acceptable fitness percentage.

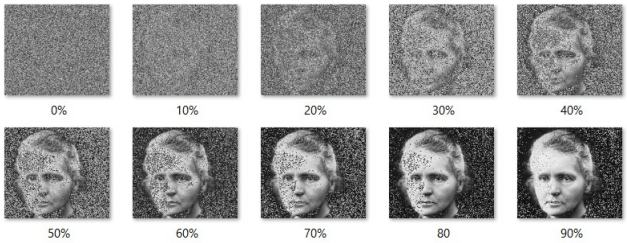
```
public Matrix myOwnFitnessFunction
(Matrix img,TargetImage target){
int windows=0;
int score=0;
for(int i=0;i<img.Width();i++){
for(int j=0;j<img.Height();j++){
Color c1=new Color(img.getRGB(i, j));
Color c2=new Color(target.getRGB(i, j));
double similitudeParent=
useful.colorSimilitudePercentage(c1,c2);
if(similitudeParent1 >=80){
        score++;
        }
    }
score=(score/(img.Height()*img.Width()))
score=score*100;
img.setFitness(score);
return img;
}
```

*Analysis*

$$(i+1*C)(j+1*C)(C^9)$$
$$\Rightarrow (i+1)(j+2)$$
$$\Rightarrow (i*j)(i)(j)(1)$$
$$\Rightarrow O(i*j)$$

## III. Experiments

### A. First experiment




*1) Algorithm:* Pixel individual comparison

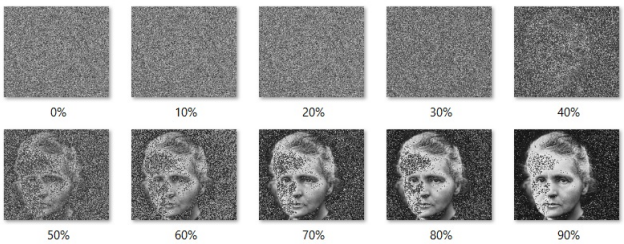
*2) Population:* 50

*3) Image Resolution:* 184x159. Despite this is a small image with low resolution the process has been slow and takes several minutes, but it reaches the expected results.

*4) Total Iterations:* Using this algorithm, it took 4719 iterations get a 90% fitness percentage.

*5) Iterations for fitness percentages:*

- 10%: 136 iterations.
- 20%: 285 iterations.
- 30%: 516 iterations.
- 40%: 759 iterations.
- 50%: 1366 iterations.
- 60%: 1962 iterations.
- 70%: 2620 iterations.
- 80%: 3557 iterations.
- 90%: 4719 iterations.



*6) Algorithm:* My Own Fitness Function

*7) Population:* 50

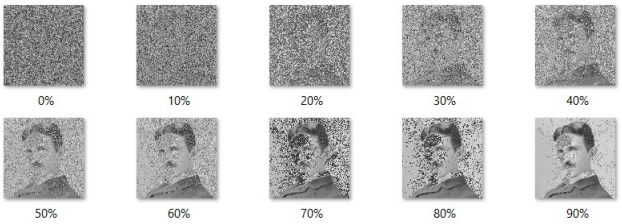
*8) Image Resolution:* The image resolution is: 184x159.

*9) Total Iterations:* The total of iterations that the image required to get to get 90% was 2416 iterations.

*10) Iterations for Comparison Percentage:* For all the comparison percentage the process took different iterations:

- 10%: 1 iterations.
- 20%: 2 iterations.
- 30%: 3 iterations.
- 40%: 110 iterations.
- 50%: 318 iterations.
- 60%: 557 iterations.
- 70%: 1005 iterations.
- 80%: 1461 iterations.
- 90%: 2416 iterations.

## B. Second experiment




*1) Algorithm:* Pixel individual comparison

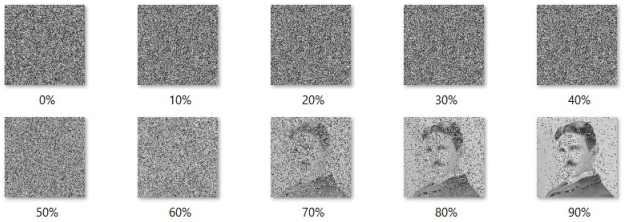
*2) Population:* 50

*3) Image Resolution:* The image resolution is: 100x100.

*4) Total Iterations:* The total of iterations that the image required to get to get 90% was 2416 iterations.

*5) Iterations for Comparison Percentage:* For all the comparison percentage the process took different iterations:

- 10%: 92 iterations.
- 20%: 137 iterations.
- 30%: 175 iterations.
- 40%: 228 iterations.
- 50%: 285 iterations.
- 60%: 372 iterations.
- 70%: 444 iterations.
- 80%: 556 iterations.
- 90%: 743 iterations.




*6) Algorithm:* My Own Fitness Function

*7) Population:* 80

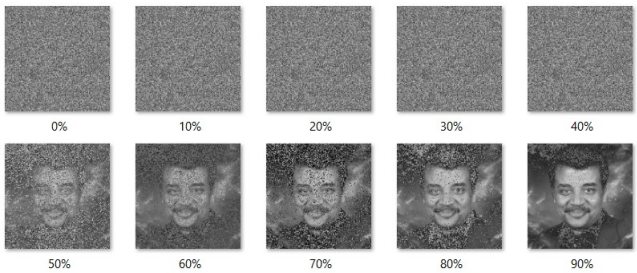
*8) Image Resolution:* The image resolution is: 100x100.

*9) Total Iterations:* The total of iterations that the image required to get to get 90% was 331 iterations.

*10) Iterations for Comparison Percentage:* For all the comparison percentage the process took different iterations:

- 10%: 1 iterations.
- 20%: 1 iterations.
- 30%: 3 iterations.
- 40%: 26 iterations.
- 50%: 126 iterations.
- 60%: 285 iterations.
- 70%: 444 iterations.
- 80%: 209 iterations.
- 90%: 331 iterations.

## C. Third experiment




*1) Algorithm:* Pixel Per Pixel Comparison

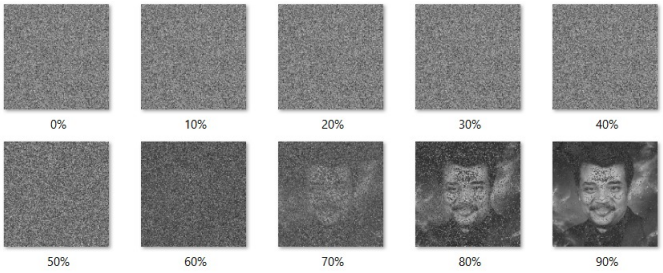
*2) Population:* 68

*3) Image Resolution:* The image resolution is: 200x200.

*4) Total Iterations:* The total of iterations that the image required to get to get 90% was 867 iterations.

*5) Iterations for Comparison Percentage:* For all the comparison percentage the process took different iterations:

- 10%: 61 iterations.
- 20%: 113 iterations.
- 30%: 164 iterations.
- 40%: 218 iterations.
- 50%: 282 iterations.
- 60%: 365 iterations.
- 70%: 471 iterations.
- 80%: 584 iterations.
- 90%: 867 iterations.




*6) Algorithm:* Own Fitness Function

*7) Population:* 68

*8) Image Resolution:* The image resolution is: 200x200.

*9) Total Iterations:* The total of iterations that the image required to get to get 90% was 867 iterations.

*10) Iterations for Comparison Percentage:* For all the comparison percentage the process took different iterations:

- 10%: 1 iterations.
- 20%: 8 iterations.

- 30%: 21 iterations.
- 40%: 117 iterations.
- 50%: 206 iterations.
- 60%: 258 iterations.
- 70%: 328 iterations.
- 80%: 444 iterations.
- 90%: 662 iterations.

## IV. Results

In all the experiments the result was very similar. Ee used images bigger than the recommended. All of them, we can appreciate that "PixelIndividualComparison" method lasts longer than "myOwnFitnessFunction" to reach a 90% fitness. We also can see that this algorithm, even when lasts less, hasn't a continuous evolution, it is not as normal as it should be. On the firsts generations there is no visible result because "myOwnFitnessFunction" is more flexible when comparing pixels, when the generations reach a 40%-50% fitness score. It is important to highlight that in half of iterations, the function achieves almost the same result than "PixelIndividualComparison".
We can also appreciate that for small images, the results are not very different, nevertheless, "myOwnFitnessFunction" is still faster.

## V. Discussion

Despite of the time that the program takes to solve the problem, it is achievable to get good results using these genetic algorithms .It is important to consider the population number, the fitness function used and the target image, thats because the execution time will increase considerably if you work with high resolution images. After the experiments, we can notice that the clue is the fitness function an well done mutations. We used a simple "crossOver" function and the results were really satisfactory. Also, we have to highlight that being permissive with the colors and count as the correct color the ones that are similar, but not the same, reduces significantly the iterations number and the generations.

## VI. Conclusions

After doing a lot of tests, experiments and changes in all the time that we have being working in this project, we can say that the algorithm requires that the processor make a lot of tasks and requires several minutes to reproduce an image that can relate to the target image, but the genetic algorithm work and gets to the image that we need to, taking the time that he needs.

## References


[1] Oracle Corporation. Class BufferedImage Documentation[Date unknown, accessed 2018 Agust 20 ] https://docs.oracle.com/javase/7/docs/api/java/awt/image/BufferedImage.html

[2] Oracle Corporation. Class Random Documentation[Date unknown, accessed 2018 Agust 20 ] https://docs.oracle.com/javase/7/docs/api/java/util/Random.html

[3] Oracle Corporation. Class Arrays Documentation[Date unknown, accessed 2018 Sept. 3 ] https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

[4] Melanie Mitchell, An introduction to genetic algorithms. The MIT Press, Cambridge, Massachusetts, 1998.

[5] Daniel Shiffman,The Nature of Code. December 1998.

[6] Oracle Corporation. Class ImageIcon Documentation[Date unknown, accessed 2018 Sept. 3 ] https://docs.oracle.com/javase/7/docs/api/javax/swing/ImageIcon.html