Evolutionary algorithm

Almir Mullanurov B17-06

April 2019

1 Information

Steps to run the project:

- Install the project to your computer.
- After that, if you do not have it, you need to install python3 and following packages for it: pillow, numpy, copy and random. You can do it using pip commands (pip install pillow/numpy/copy/random).
- Add the image that you want to be changed in the project directory (in the same folder as file **123.py**).
- Rename image to **input.jpg**.
- After that go to the directory of the project via terminal and run the following command **python3 123.py**. Or you can open it in PyCharm and run it from the application.
- Now you need to wait until the program will finish the generation.
- The result will be in the same folder with name **res.jpg**.

Github repository: github.com/Mirlan-code/Evolutionary-algorithm
Feel free to contact me via email almirka02261999@gmail.com or telegram @mir_lan

2 Description of the genetic algorithm

Genes in the algorithm is just RGB pixels. Because of the size of the picture, the simple gen is $512 \times 512 \times 3$. So I decided to do the initial gen as randomly generated colors (uniform distribution from (0, 0, 0) to (255, 255, 255) in the 512×512 picture. Due to the reason that we will not run it infinitely, the generated image will have the shade of chosen colors, so I chose the random colors nearly to some my favourite color(150, 150, 150).

In the algorithm I divided the initial page into blocks which will have their own generations. They are not dependent, so it will be much faster in this case. Due to the reason that pixels are generate the closest to the initial, we will have pixel art which is pretty similar to the input image (**update_rect** function). Also pixels will be more visible if pixels inside each block will have similar color (**update function**).

After that I find interesting to add more blocks with independent generations. Rectangles are random tuples (x_1, y_1, x_2, y_2) with uniform distribution of variables (x_1, y_1, x_2, y_2) with their own world and generations. Because of this we will see the new shapes of the picture with intersections between the rectangles. Each rectangle has its own pixel world so when they intersects, it means, that two worlds generations are somehow merged (the last one most probably captured the new one and will have domination in their intersection).

In the main part of the algorithm I have two functions called **update_rect** and update. They are using for creating the new generation of the world (In my algorithm each block is independent world). Each of these functions use their own fitness functions, but mutation is the same for both.

I decided to do that, because pixels of different sizes are really good for decorating the picture and viewing other sides of the art. With their intersection, the pixels will produce very interesting things such as images with something like blur or noise in it.

3 Fitness function

The function which evaluates the current generation is the fitness function. It returns the real value in range [0, 1]. The closest returning value to 1 is the best.

Due to the reason that the ideal world is the picture from the input, I decided to do the fitness function as a simple Manhattan distance between the current generation and ideal world. The Manhattan distance between them is just the absolute difference between the corresponding RGB pixels (**fitness_rect** function).

Also I did the second fitness function for the function **update**. Because we want pixels to be similar, but not equal color inside one block, fitness function evaluates the value calculating the Manhattan distance between the average color in the block and pixels color (**fitness_similar** function).

4 Crossover

My crossover function for the genes is not really hard and somehow similar to the our world. Crossover between two generations (mother and father) is just

the new genes which lay between the mother and father genes with (probably) some small deviations (**crossover** function).

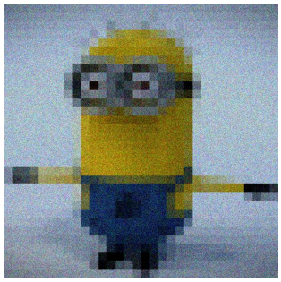

Figure 1: Mother

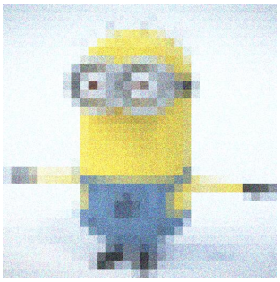

Figure 2: Father

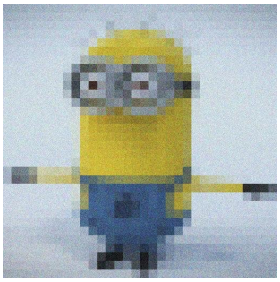

Figure 3: Child

My population (this variable in the algorithm equals to 10) is sorts via fitness functions. So I have two halves: first one is the generations, which will stay alive, and the second one, which will die. After that I pick two generations from the first half and using **crossover** function create new child for the next generations.

5 Mutations

Sometimes my algorithm stacks on the way to ideal image because of crossover between generations. So I used mutations to get out of equilibrium. Mutations is used to change some priority generation for other, which can not beat the best one, but can create another, probably most fittest child (**mutation** function).

The generation of the mutated generation happens with some probability (for my algorithm I chose 0.25). If it happens, then some childs will have this mutated generation as their parrent.

6 Examples

I want to show several examples of how my algorithm works.

- The first one worked too much because each rectangle has its own 40 generations. So the total amount of generations was $40 \cdot 512 \cdot 512 = 40 \cdot 2^{18} = 10485760$. And because each block has size of 16×16, then each generation was doing $16 \cdot 16 \cdot \mathit{populations} \cdot 3 = 7680$ operations.

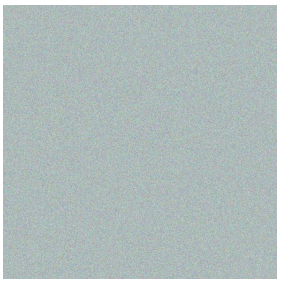

Figure 4: Before

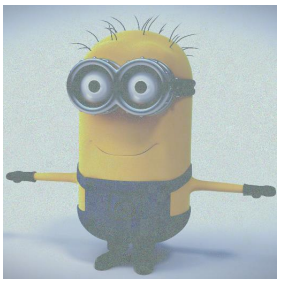

Figure 5: After

As we may see, the output is really close to the ideal image since the amount of iterations is big. Because of that I decided to have mutations more often and changed the amount of generations.

- The second example is used another initial generation – normal distribution with the expected value being average color of the block. Also amount of generations per rectangle was equal to 8.

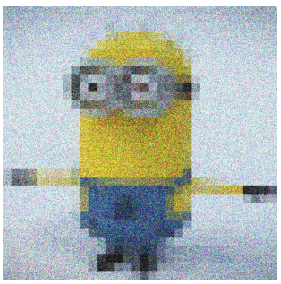

Figure 6: Before

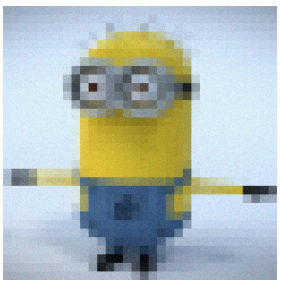

Figure 7: After

- The next few examples are the best one for my opinion that my algorithm produced.

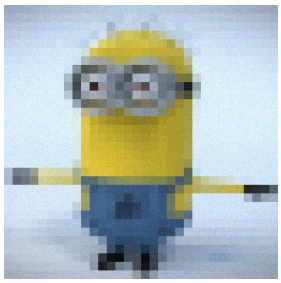

Figure 8: Initial generation has colors of the pixels nearly to the input

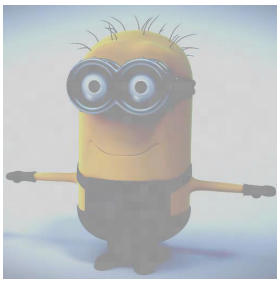

Figure 9: Initial generation was moved to the color(150, 150, 150)

Figure 10: Initial generation has colors as an average in the block

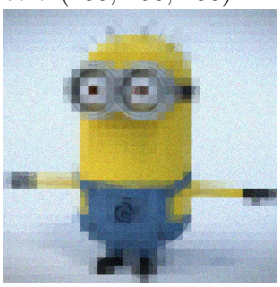

Figure 11: Initial generation has random colors and worked too much

These pictures gives us a good understanding that each parameter in the algorithm is needed. You may change one of them (for examples decrease the size of the block) and it will produce something which is really different from the other and that is the main art of the genetic algorithm. Pixels in the final images show the prototype of the real world generations and populations.

- And the other pictures

Figure 12: Earth before

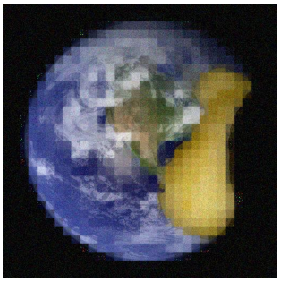

Figure 13: Earth after

Figure 14: Smurf before

Figure 15: Smurf after

As you may see, the eye of the smurf has some other colors inside it. This is because of many mutations in this region and due to the reason that amount of generations inside each block is fixed (it does not always go until it will be greater than **golden_number**).