Assignment II

Computers Can Do Art!

Dragos Strugar - 5 April 2019

VISIT https://strudra.github.io/genetic-algo/
to see the algorithm in action.

Final Results (source image on the left, result on the right),

Obtained after 4hrs of running the algorithm:

SOURCE IMAGE: MICHAEL JACKSON, THE "KING OF POP"

Introduction & Different Approaches

Using a programming language of your choice you will be given a several test input images of 512x512 pixels. You will produce via any evolutionary algorithm another 512x512 from a single test image your computational artist which will be presented to your peers and a distinguished judging panel!

The goal of this assignment was to come up with an evolutionary/genetic algorithm to produce an image with another 512x512px image as an input. All other parameters were left unknown, leaving the freedom to the writer.

To tackle the problem, I was thinking of using numerous methods, fitness criteria, and shapes. Some of them are: pixel-by-pixel, rectangles, triangles, ovals, lines, 2d/3d spatial nodes, and in general polygons. I have implemented some of these approaches, analysed the result and decided to finally go with the approach of polygons (more specifically, hexagons). Below is the rationale and reasoning on choosing hexagons as my main building block.

As the goal of this assignment is not necessarily to produce an exact replica of the source image, it was obvious to me that the pixel-by-pixel approach is not going to produce the most creative/artistic outcome there is. Especially when the colour sampled from the image provided as an input.

Similar thing happened when I tried with rectangles. I got a more interesting result, but the image looked very similar to the source image, and in addition, it was all in pixels, it looked unprofessional and not artistic at all.

2d/3d spatial node approach was very interesting to me. I wanted to try it out. The idea was to map the lines (straight or curve) to the 2d/3d space. Both ends of these lines would serve as nodes. Then other lines would get appended to the ends of previously generated lines. 3d case is also possible as we can randomly pick the 3rd, Z axis point for the picture and map our line to that point. After some time spent thinking about the implementation, I was not sure I could effectively implement this algorithm, especially in my language of choice - JavaScript. I wanted something a bit less performance-heavy, something that can be executed in a traditional web-browser, and finally something that works quite well - i.e. still produces an artistic outcome. It was too hard to implement.

Ovals (including circles) would have looked way better than the pixel-by-pixel & rectangle approach. I was thinking of using the BFS graph algorithm to determine the fitness criterion. Overall the solution in my opinion would look like a piece of art, but I choose not to go with this approach as the next one (polygons) looked way better and it is more straightforward to implement.

Polygons were my final choice for several reasons. First was the ease of implementation. Second, I liked in particular how hexagons looked like when combined with one another. And third, the image I chose as a source image has quite a bit of detail, and polygonal shapes model such shapes very nicely. This is not to say that my approach is exclusive to the input image, but rather that it fits the source image the best.

Fitness Function of the polygonal approach

The first thing I needed to figure out was how to determine what outcome is better than another. In the pixel-by-pixel example it was very easy. As each colour can be represented in the RGB spectrum, I basically just took the absolute value of the difference between the source image's red, green and blue values with the corresponding generated image's values. Smaller deviations means that the fit is better.

However, in the polygonal case, the case is a bit different. We still want to use colours to determine the fitness. I needed to sum all the pixels within the hexagon and sum them all up. Smaller total deviation means that it is the better fit. Here is the pseudocode for my idea:

```
fitness = 0
for i = 0 to width
    for j = 0 to height
          color C1 = get_pixel_colour(source_img, i, j)
          color C1 = get_pixel_colour(final_img, i, j)

          // get colour deviation
          del_red       = red(C1)    - red(C2)
          del_green     = green(C1) - green(C2)
          del_blue      = blue(C1)   - blue(C2)

          // euclidian distance
          double pixel_fitness = del_red * del_red +
                                 del_green * del_green +
                                 del_blue * del_blue

          // lower better
          fitness += pixel_fitness
    end
end

return fitness
```

It is a fairly simple idea. Just sum all of these in the polygon and we get our result. The crossing pixels have 50-50% chance of being included and excluded from the consideration.

Genetic Programming

After completing the fitness function, the next step is to come up with a complete genetic programming algorithm which would generate new individuals and compare them to the source using the fitness function.

The algorithm models a population, with each individual containing DNA that can be depicted using an image. We start off with a randomly generated gene pool, and compare each and every one of them to the source image. Then we apply the fitness function to the results, and we get the best one. Then, the one that beat all the others is naturally selected as the winner.

Essentially, the algorithm does the following steps:

1. Get and replicate the current individual's DNA sequence and slightly modify it (mutation)
2. Use the new DNA to actually render hexagons on the screen
3. Compare the obtained result with the source (reference) image
4. If the new result looks like the source more than the previous one (using the criteria of the fitness function), make the newly generated image the best one
5. Repeat the process.

Genetic Programming Parameters

We were also asked to explain the parameters that our program uses. These are parameters like chromosome function, crossover, mutation, and various constraints. The fitness function is explained above.

To make the implementation easier, I have made two data structures:

- Population
- Individual

Population has only 1 parameter - size, and has two methods: iterate and getFittest. Size determines how many Individuals are in the population. getFittest sorts the Individuals by fitness using the fitness function mentioned above. Iterate runs the genetic algorithm. Here are the steps:

1. Initialises the offspring to an empty array
2. Determines how many individuals should be created
3. Sorts the individuals by fitness
4. Makes a new Individual using the parent pool (crossover)
5. Fittest survives (natural selection)

Individual has two parameters - mother and father (their DNA). The main logic behind the rest of the code that is used to initialise the individual is doing mainly one thing — figuring out whether to use mother's or father's gene for some part (randomly). In addition, an Individual can mutate its genes by tuning the colour parameters (parents' DNA).

The major function that can be applied to the Individual is the `draw` function. It just uses JavaScript utility functions to draw the individual on the screen.

Overall Algorithm Parameters

The entire project is build s.t. a lot of parameters can be adjusted for the use case. Here is the exhaustive list of the parameters that an end-user can change before the runtime of the algorithm. Note that due to the shortage of time of for the assignment, the visual dashboard has not been completed and these values should be changed in the source code.

- Population size (# of individuals in population)
- Mutation chance (likelyhood of DNA string mutating during breeding)
- Selection cutoff (# of individuals that get selected to breed next)
- Fittest survive, i.e. natural selection (yes or no)
- Maximum # of polygons used to draw the image

These parameters can be change to better fit a specific use case. Some of them are used as halting conditions (combination of time elapsed,

Algorithm Analytics

In order to better understand the algorithm and how well it performed, I came up with a set of analytics items, like:

- Time elapsed (can be used to terminate the program)
- Current number of generations
- Avg. time per generation
- Avg. time per mutation
- Current overall fitness
- Highest fitness achieved
- Lowest fitness generated

These items may or may not be useful to the end user during the time of image generation. I chose to have them for development purposes, however one may use them for a more detailed analysis.

Implementation

The implementation of this algorithm is done in the JavaScript (JS) programming language. As I stated above, I chose it for a few reasons:

- My familiarity with the language and tools surround JS
- Straightforward image manipulation
- JS built with changes (including changes to pictures) in mind
- Easily deployed as a website so results can be shared with others (like you, the TA or the professor reading this)

Reference URLs are on the bottom of the review. This picture looks artistic to me because it is not a straightforward pixel-by-pixel comparison. The shapes are chosen randomly, and the goal throughout the process was not strictly set. I was not aiming to fully copy the image, but rather to make a creative version of that. The shapes I have chosen, polygons, I think have done the job nicely.

SOURCE IMAGE: MICHAEL JACKSON, THE “KING OF POP”

Reference Links

Link to the final implementation:
https://strudra.github.io/genetic-algo/

Link to the source code:
https://github.com/strudra/genetic-algo

Thank you,

Dragos Strugar, B17-05