Genetic Painting

Liangwang Ruan
Peking University
Email: ruanliangwang@pku.edu.cn
Student ID: 2101111577

***Abstract*—The paper proposes a genetic algorithm to let the computer learn how to draw like human. Given a reference image, the algorithm can generate a sequence of geometry primitives to mimic the strokes on the canvas. User can control the level of details of the painted image by adjusting the stroke size limit and stroke number.**

I. Introduction

Generative art refers to art that in whole or in part has been created with the use of an autonomous system, has been widely studied over the past decades. A specific branch of generative art uses computer algorithms to show(most visually) regularity of mathematical patterns and show variability from pseudo-random numbers, it's known as algorithmic art. A typical example of algorithmic art is fractal art generated by iteration algorithms in complex domain as shown in Fig. 1.

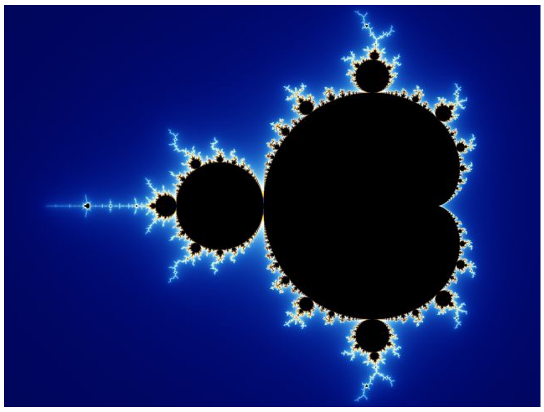

Fig. 1. Mandelbrot Set ©From Wikipedia

Intelligent optimization algorithms are especially designed to deal with NP-hard, non-convex, non-smooth optimization problems, include genetic algorithms, simulated annealing algorithms, ant colony algorithms, etc. Many of them are inspired from biological and natural behaviors, like the genetic algorithms are simulating the evolution theory of Darwin, the ant colony algorithms are imitating the foraging behavior of the ants. The complexity of the problem itself and the bionics nature of these methods make them can generate complex, fascinating visual arts. Examples include 3D animation [1], 2D drawing [2], image triangulation [3]. This paper is one of these works, trying to use the genetic algorithm to let the computer learn how to draw like human. To be more specific, the algorithm learns to draw one stroke of paint at a time, at last get a similar painting work like the given image. This work is **not** the first work to do this task, many other artists have tried the same task with other intelligent optimization algorithms [2] [4] [5], but this work is the first to apply genetic algorithms to this task.

II. Related Work

Roger Johansson is the first to use simple triangles to reproduce fine art [6]. Although he claims to use the genetic programming method, he truly uses the hill climbing algorithm to optimize the colors, shapes, and positions of a group of triangles. In his algorithm, he optimize 50 triangles for about 1 million steps to get a similar result of the face of Mona Lisa. Because he optimizes all triangles as a whole, the method is hard to scale-up. Inspired by Roger Johansson, Michael Fogleman creates a macOS application *Primitive* that can transfer an image into combination of geometric primitives [4]. He uses the hill climbing algorithm and the simulated annealing algorithm to optimize the shape and position of each primitive one by one. Unlike Roger Johansson's algorithm, *Primitive* doesn't optimize the color of each primitives, but samples the color from the reference image. Sam Twidale then creates another desktop application called *Geometrize* that makes *Primitive* cross-platform and easy to use [5]. Sebastian Proost uses a real genetic algorithm to optimize triangles to re-draw Van Gogh's The Starry Night [7], but like Roger Johansson's work, he optimize 150 triangles as a whole, so the algorithm is not flexible enough and hard to scale up. Anastasia Opara implement the hill climbing algorithm as Roger Johansson [8], but she introduces a sampling mask to help the algorithm to focus on different area of the reference image in different stages.

III. Algorithm Overview

In this section I will first define the problem we want to solve, then introduce the basic algorithm and two main techniques of our algorithm to get better performance.

To define the problem clearly, the input of the algorithm is a reference image(whether photographed or painted), we want the algorithm to output a series of semitransparent geometric primitives, by combining them one by one we can get a similar image as the reference. We call each of the semitransparent geometric primitives a "stroke". Also we want the algorithm to mimic the behavior of real human painting, i.e. first using large size stroke on the background, then small size stroke on the main body. To simplify the algorithm, we only consider the rotated ellipse as our strokes. Each stroke has 5 properties:

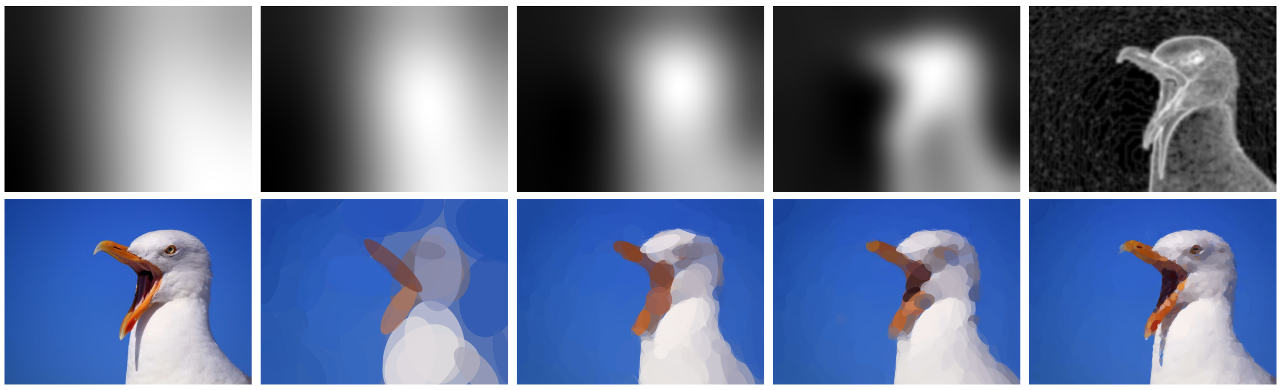
Fig. 2. Seagull: The top line is the sampling mask at stage 200, 400, 600, 800, 1000. The bottom left image is the reference image, the other four images are the canvas at stage 50, 200, 600, 1000.

1) color: (r, g, b) 3 channels
2) center position: (x, y) on canvas
3) size: (sx, sy)
4) rotation angle: $r \in [0, 180)$
5) opacity: $\alpha \in [0, 1]$

The basic algorithm we use is genetic algorithms. In this problem, every stroke is a DNA, many strokes construct a population. The fitness of the DNA is evaluated as the similarity between the reference image and the canvas after drawing the stroke on. At every stage of the algorithm, we first initialize the population randomly, then evolve the population using cross over and mutation for some generations, at last draw the stroke with the most fitness on the canvas. However, directly applying this genetic algorithm does not work well as shown in Fig. 3. The shape and color of the stroke is a bit random, and all the details are last. To improve the result, we introduce two main techniques as follow.

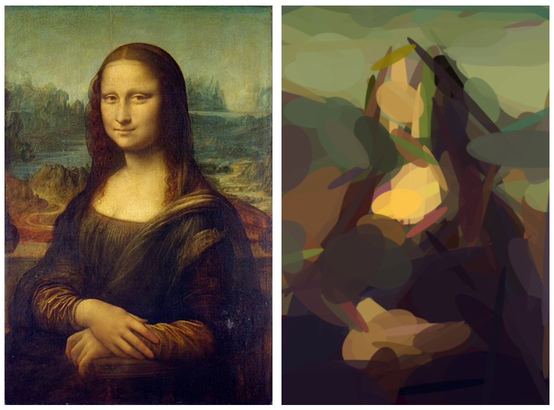
Fig. 3. Failure case: the left is the reference image - the Mona Lisa by Leonardo da Vinci, the right is the result of the basic algorithm after 300 iterations.

a) Space Reduction: Having so many parameters of one stroke, even we only optimize one stroke at one stage, the parameter space is still too large for the algorithm to find a global minimum. To solve this problem, one important observation is that we don't have to choose the globally best stroke at every step, because the stroke size is small compared with the canvas, there should have multiple plausible positions for one stroke. And even if we can't even find a plausible position, we can still paint the stroke on the canvas and hopefully it will be covered by other better strokes. As a result, we can randomly sample a small constraint area for the strokes at the beginning of each stage, and sample the population only inside the constraint area. And the same as *primitive* [4], we can sample the color from the reference image, rather than optimize it from random. Further more, we can constrain the maximum and minimum size of the stroke at different stages, and fix the opacity at 0.7. In this way we can largely reduce the searching space so the algorithm can easily find a good stroke.

b) Sampling Mask: When human paints a picture, we usually first paint the background, then paint the main body. The background has less information and is more smooth, the main body has more information and has more edges. To let the algorithm mimic the behavior of human being, we create a series of sampling masks at different stages as shown in Fig. 2. These masks are used to provide the position sampling probability in population initialization, and are created using the Gaussian blur of the gradient of the original image as in [8]. At first the Gaussian kernel size is big so the algorithm has more chance to explore background on the canvas, then the kernel size is decreasing so the algorithm can focus on the foreground. With these masks the algorithm can complete the details on the main body rather than wasting on the background.

IV. Implementation

Alg. 1 shows the scheme of our algorithm, I'll introduce the details of each steps in this chapter.

At first the canvas is cleared as the mean color of the reference image, then in each stage, the algorithm follows a traditional genetic algorithm routine.

a) Initialization: For the stroke size limit, we set the maximum and minimum size at the beginning of the program and fix it at the first few stages, after some stages the maximum size and minimum size both begin to decease, and at the last

Algorithm 1 Genetic Painting

```
Require: $I$                                        ▷ Reference image
Ensure: $S$                                   ▷ Painted image sequence
 1: $canvas \leftarrow \mathrm{mean}(I)$
 2: $S \leftarrow [canvas]$
 3: for $s < stage\_max$ do
 4:     ComputeStrokeLimit($s$, $stage\_max$)
 5:     CreateMask($I$, $s$, $stage\_max$)
 6:     $P \leftarrow$ InitPopulation($I, pop\_size$)
 7:     EvaluatePopulation($I$, $P$)
 8:     for $g < generation\_max$ do
 9:         $Q \leftarrow$ CrossOver($P$)
10:         $Q \leftarrow$ Mutation($Q$)
11:         EvaluatePopulation($Q$)
12:         $P \leftarrow$ BestPreservation($P, Q$)
13:     end for
14:     $p \leftarrow argmin_{loss}(P)$
15:     $canvas \leftarrow$ UpdateCanvas($canvas, p$)
16:     $S \leftarrow [S, canvas]$
17: end for
```

few stages the size limit is fixed again. The limit at last stages affects the detailed level of the painted image, but the size limit at the beginning does not affect the final image that much. For the sampling masks we follow a similar idea. At first few stages the mask is all black meaning uniform distribution. Then the Gaussian blurred mask is computed and the Gaussian kernel size is decreasing from image size to one pixel. To compute the mask, we first transfer the reference image into gray scale, then compute the gradient in two directions, then change it into polar expression, then normalize the magnitude into $[0, 1]$ and apply Gaussian blur. Apart from building masks from Gaussian blur, the difference between the canvas and the reference image is also used at the last stages helping the algorithm to finish the details on the main body. To initialize the population, we first randomly sample a point on the canvas using the sampling mask, and get the color of that point on the reference image. The stroke size is uniformly sampled between the size limit, the rotation is sampled based on the local gradient on that point as in [8]:

$$r = U(0, 180) \cdot (1 - |\boldsymbol{d}|) + angle(\boldsymbol{d}) + 90 \qquad (1)$$

This equation means: if the reference image has large gradient here, the stroke should better aligned with the edge, which is perpendicular to the gradient. But if the local gradient is small, the stroke can be drawn in random direction. By doing this, we get the first DNA in our population, and all the rest DNAs can be mutated from this DNA using the mutation operation introduced below.

b) Cross Over: The parents are sampled based on the fitness of each DNA. The fitness is computed as the negative of absolute difference f between the reference image and the canvas after drawing the DNA on. We use dynamic linear regularization method for computing sampling probability:

$$F_k = -f + f_{max} + \xi^k \qquad (2)$$

where k is the generation number, $\xi = 0.9$. After choosing two parents p_1 and p_2, we randomly let the child's position, size, and rotation angle from either p_1 or p_2 with equal probability. Continue this process until we get the next generation same size as the parent.

c) Mutation: We only mutate the position, size and rotation angle of the stroke. For the position, we let the stroke can move a little bit from the original point in two directions. For the stroke size, we let the stroke can shrink into half or extend double in either direction, and also need to satisfy the size limit. For the rotation angle, we just let the stroke to rotate freely. The three mutation operations have different possibilities depending on their influence on the result, for example we set the possibilities of them at 0.2, 0.4 and 0.4.

d) Best Preservation: For better convergence rate, we preservation the best individual in the parent generation. We compare the best DNA in the parent generation and compare it with the best DNA in the child generation, if the parent generation performs better, we replace the worst DNA in the child generation with the parent's best, otherwise we do nothing.

V. Results

I implement the algorithm in C++ using the OpenCV library [9]. The algorithm usually takes few minutes to converge on a regular image(around 500px). The differences between the canvas and the reference image at different stages are shown in Fig. 4, it shows our algorithm converges well. The stages

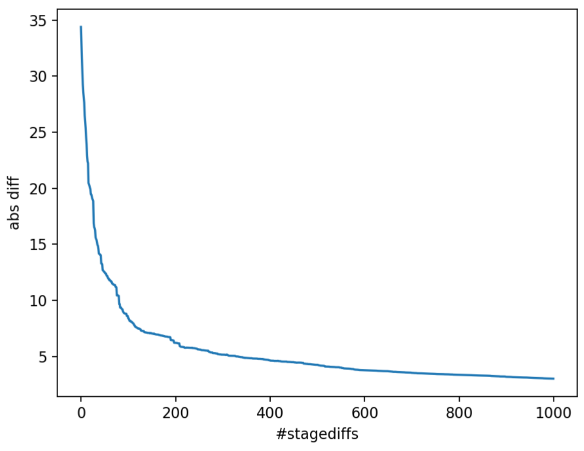

Fig. 4. Difference Convergence

needed to get a good image largely depend on the complexity of the image. For example, in the seagull example(Fig. 2) the algorithm can generate very detailed image like the reference in about 1000 stages, but in the Mona Lisa example(Fig. 5), the algorithm still needs more works on the face after 1000 stages. But we can see from the canvases in the early stages that the algorithm can approximate the reference image with very few strokes, this makes the algorithm able to generate stylized art works with simple geometries.

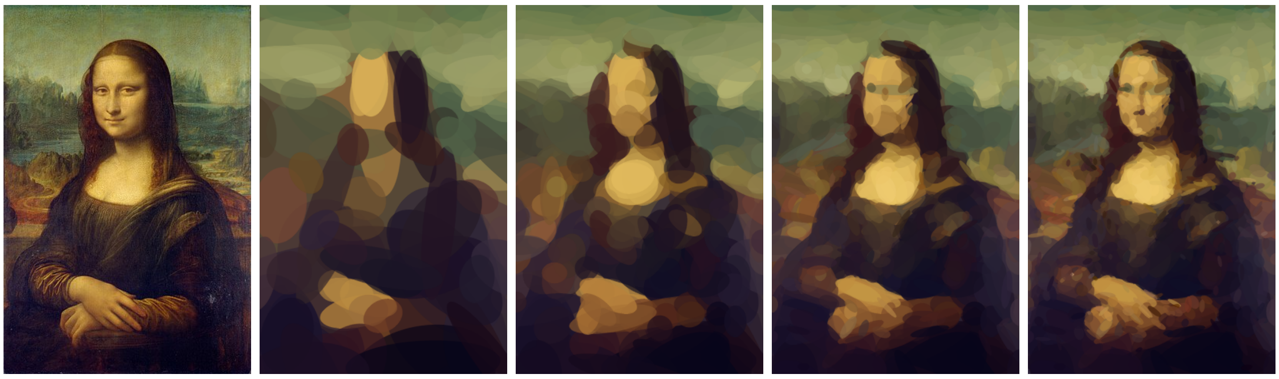

Fig. 5. Mona Lisa: The left most is the reference image, the rest images are the canvas at stage 100, 300, 600, 1000.

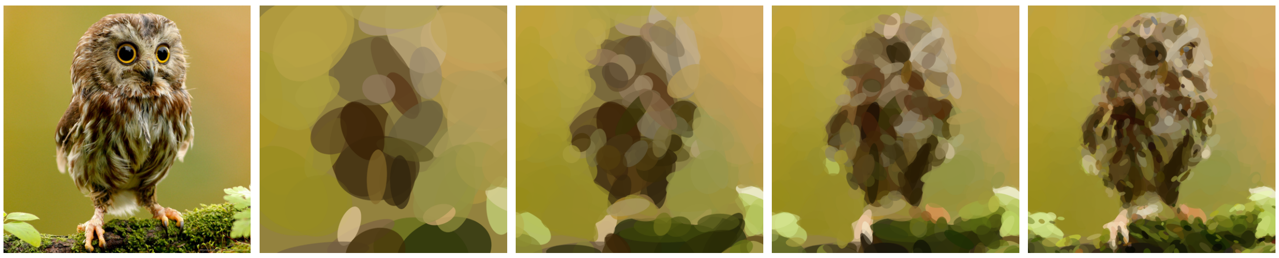

Fig. 6. Owl: The left most is the reference image, the rest images are the canvas at stage 50, 200, 400, 800.

VI. Limitation and Future Works

Take a closer look at the face of Mona Lisa in Fig. 5 and the eyes of the owl in Fig. 6, we can find the colors are preserved but the shape details are missing. When human paints a portrait, the painter usually spends a large portion of time on the face of the character. But for the computer, if the face only takes a small area on the canvas, the possibility of the algorithm to draw a stroke on the face is relatively small. To solve this problem one can try to use the face detection algorithm to create a sampling mask, force the algorithm to spend more time on the face of the main character.

Another drawback of this algorithm is efficiency, which can be improved by using parallel genetic algorithms [10]. At each stage this algorithm is a standard genetic algorithm, so it should be easily paralleled. At last the current implementation is not interactive enough, applications like [4] [5] can adjust the shape and size at run time and have better GUI support. But this implementation is good enough to show the efficacy of the algorithm.

References

[1] D. de Andrade, N. Fachada, C. M. Fernandes, and A. C. Rosa, "Generative art with swarm landscapes," *Entropy*, vol. 22, no. 11, 2020. [Online]. Available: https://www.mdpi.com/1099-4300/22/11/1284

[2] Drawing with ants: generative art with ant colony optimization algorithms. [Online]. Available: https://amydyer.art/wordpress/index.php/2020/01/01/drawing-with-ants-generative-art-with-ant-colony-optimization-algorithms/

[3] Triangula. [Online]. Available: https://github.com/RH12503/triangula

[4] M. Fogleman. Primitive. [Online]. Available: https://primitive.lol/

[5] S. Twidale. Geometrize. [Online]. Available: https://www.geometrize.co.uk/

[6] R. Johansson. Genetic programming: Evolution of mona lisa. [Online]. Available: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/

[7] S. Proost. Genetic art algorithm. [Online]. Available: https://blog.4dcu.be/programming/2020/01/12/Genetic-Art-Algorithm.html

[8] A. Opara. Genetic drawing. [Online]. Available: https://github.com/anopara/genetic-drawing

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[10] T. Harada and E. Alba, "Parallel genetic algorithms: A useful survey," *ACM Comput. Surv.*, vol. 53, no. 4, aug 2020. [Online]. Available: https://doi.org/10.1145/3400031