Layered Evolutionary Image Generation

Goedert, Samuel
goederts@msoe.edu
Milwaukee School of Engineering

CONTENTS

ABSTRACT

I present a new approach to image generation via Genetic Algorithm and demonstrate an implementation for image compression. By using layers of genetic populations that each contribute to the generated image, scalability is increased, and convergence becomes faster.

INTRODUCTION

The goal of this experiment is to test a new approach to image generation via a Genetic Algorithm (GA). The implementation in this paper has the goal of image compression, regenerating an image in a new format with the hope that it will take less storage. This GA is not limited to this application, but it is an appropriate way to communicate the algorithm. The first section of this paper covers a traditional GA implementation for image compression. The second section covers the new layered GA with the same objective as the first.

GAs are optimization algorithms that mimic natural selection to find minimums or reduce loss for some given function. Modeling computer programs after biological processes was initially done by Nils Barricelli and first published in his 1954 paper "*Esempi numerici di processi di evoluzione*" (Barricelli, 1954). GAs can outperform other algorithms because they do not have to exhaustively search through the entire solution space to find an optimal solution. By iteratively making random adjustments to a set of candidate solutions and removing ineffective solutions, GAs can converge to an optimal solution.

GA's use a fixed size population which is a set of entities. The entities are encoded in either chromosomal bit-strings or numeric attributes typically in the form of a vector or matrix. The bit-strings are used when the entities have Boolean attributes like the presence or absence of a feature/trait. The numeric attributes are used when traits are not absolute, when they can be within a range such as size or position (Sourabh, Chauhan, & Kumar, 2021). Each entity's data contains a potential solution to a proposed problem. GA's have a set of rules for what occurs to the population each generation, which varies between implementations. A fitness function is a function that evaluates the effectiveness of each entity within the population against the problem space. During each generation, the least fit entities, according to the fitness function, are removed by a constant amount. The empty slots in the population are filled by cross-over (mating) between surviving entities or exact cloning of survivors, with the option for random mutation. The surviving members don't have to be kept for the next generation if enough new entities are created. Through this generational process, the average fitness scores of the population converge to an optimal solution.

In this paper I use a GA to encode an image with a set of rectangles rendered onto a canvas. Each entity within the population is a set of rectangles competing to resemble the original image as closely as possible. To increase the efficiency of the algorithm a series of populations are created, each contributing a layer of rectangles to the image rendered. Layering the algorithm reduces the time for convergence and improves scalability.

All computations done for this paper were run on Milwaukee School of Engineering's supercomputer cluster named *ROSIE*. Having access to this level of computing power accelerated experiments and reduced computation time.

SINGLE POPULATION

This section describes a GA that is intended to compress a given image by encoding it as a series of rectangles. The GA is a simple implementation with a constant population size and each entity in the population has a numeric matrix that represents a constant number of rectangles that are used to render an image. The fitness function evaluates how closely a rendered image imitates the reference image (image to compress). The bottom two-thirds of the population are removed and repopulated using cross-over between the survivors with random mutation in the offspring. The surviving members are retained in the next generation without mutation. Cloning is not used in this GA.

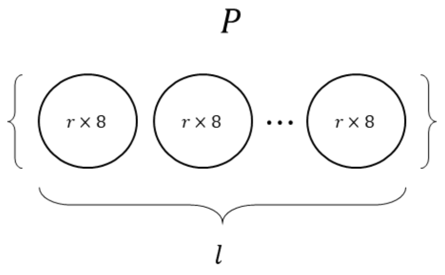

FIGURE 1: A visualization of the set P (the population) with l entities inside.

$$\begin{array}{c} E \\ r\left\{\begin{bmatrix} 0.42 & 0.25 & 0.41 & 0.10 & 0.26 & 0.45 & 0.77 & 0.71 \\ 0.29 & 0.47 & 0.73 & 0.12 & 0.98 & 0.21 & 0.01 & 0.42 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.04 & 0.33 & 0.64 & 0.29 & 0.60 & 0.98 & 0.44 & 0.78 \end{bmatrix}\right. \end{array}$$

FIGURE 2: An example of what an entity might look like, each row corresponding to a rectangle.

The population, P, is a well-ordered finite set with the hyperparameter l defining the count of elements (FIGURE 1). The elements in P are entities represented by a matrix E (an arbitrary element of P). E has the dimensions $r \times 8$, r being the hyperparameter defining the number of rectangles per entity. The values in E are floating points bounded by (0, 1). Each row vector of any given entity, $\vec{v} \in E$, has a length of 8 which represents a rectangle (FIGURE 2). The first four elements in $\vec{v}$ determine the bounds of the rectangle and the last four the color and opacity.

$$E_1 = \vec{v} = \langle 0.4, 0.2, 0.4, 0.1, \underbrace{0.26, 0.45, 0.77, 0.7}\rangle$$

#4472C4B3

FIGURE 3: An example rectangle from an entity and the color that the data represents.

The target image is encoded as a $x \times y \times 3$ matrix I, (x, y) being the resolution of the image and the third dimension holding the RGB values for each pixel. Each E can be rendered as an image using the function $g(E)$ which iterates over each row vector $\vec{v}$ in E. It creates a rectangle from each $\vec{v}$ layering them to create an image with the same dimensions of I. For any given $\vec{v}$ the function uses the first four values as percentages for the boundaries of the rectangle. The corner coordinates for each rectangle are $(x\vec{v}_1, y\vec{v}_3)$, $(x\vec{v}_2, y\vec{v}_3)$, $(x\vec{v}_1, y\vec{v}_4)$ and $(x\vec{v}_2, y\vec{v}_4)$ where x and y are still the dimensions of I. The points are not specified to a corner implying that $\vec{v}_1$ and $\vec{v}_2$ or $\vec{v}_3$ and $\vec{v}_4$ can be flipped without changing the resulting rectangle (FIGURE 4). The last four values of $\vec{v}$ are multiplied by 256 and rounded to integers to create a four-byte color value for the rectangle. Each $\vec{v}$ is placed sequentially on the same (initially blank) image to create the output of $g(E)$. For any single-color rectangle in the bounds of the reference image there exists a sequence of eight floating point numbers bounded by $(0, 1)$ that can create it with $g(E)$.

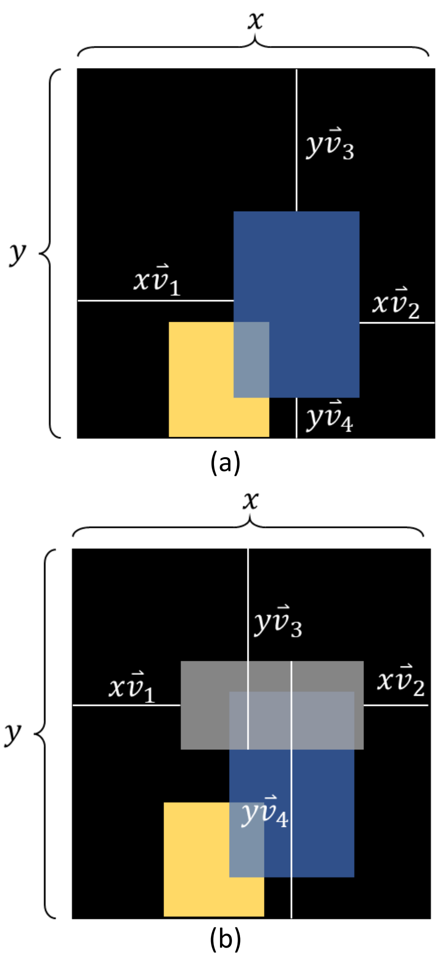

FIGURE 4: A rectangle being rendered onto a canvas with another rectangle already rendered (a) and a third rectangle being rendered on top of the previous one (b).

The fitness function, equation

(1), of the GA is defined as the sum of the absolute value of each element after an element-wise difference between the target (I) and the generated image ($g(E)$), returning a scalar. In similar situations it is common to choose mean square error or mean absolute error as a fitness function but I chose to leave out the reduction portion to remove the slight overhead of taking the mean. This approach works only because the function results are used relatively for sorting and are not used as factors in the changes. A vector containing the fitness values for each respective entity can be constructed using equation (2).

$$f(I, G) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{3} \left| I_{ijk} - G_{ijk} \right| \qquad (1)$$

$$\vec{u} = \langle f(g(E_1), I), f(g(E_2), I), \dots, f(g(E_i), I) \rangle \qquad (2)$$

Using $\vec{u}$, P is sorted by each entity's respective fitness value. The set S is the set of the top $\frac{l}{3}$ performing Es according to their new positions in P. Entities in S are randomly paired with each other without duplicates four separate times ($\frac{l}{6}$ pairs per pairing, $\frac{2l}{3}$ total pairs). These pairs go through a cross-over process that combines the row vectors (rectangles) from the two parents. For each row, there is a 50% chance that the row comes from the first parent, if not it comes from the second. All the entries in each newly spawned entity have a set random chance of being mutated. If a number is chosen to be mutated it is set to a random number in the range $(0,1)$. The bottom $\frac{2l}{3}$ performing E's are replaced by the new entities. The S subset of P remains unchanged through this process and is carried into the next generation. Due to the selection process evenly creating $\frac{l}{6}$ pairs the population size must be divisible by 6.

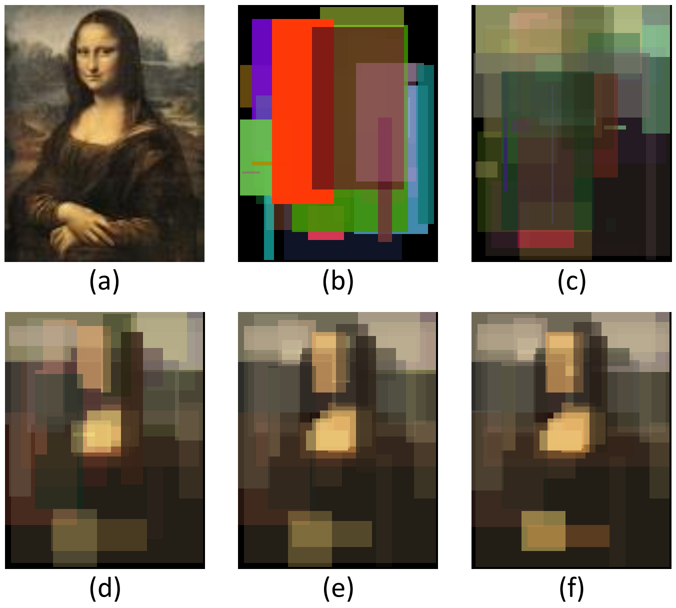

FIGURE 5: Reference image followed by GA result after 0, 100, 1000, 5000, and 15000 generations respectively with population size of 30, mutation rate of 0.002 and 35 Rectangles.

Passing this algorithm through many generations can create a final image with resemblance to the reference image after a few minutes. The storage cost of the image is magnitudes smaller than other means of compression, but a quick glance at Figure 5 shows this is clearly not an effective strategy for image compression.

LAYERED POPULATIONS

To improve the GA's results and to save on computational complexity I created an iterative layered version of the GA. This version is a series of the GA populations defined in the last section. Each layer (i.e., population) in the series is rendered on top of the image generated by the preceding layer. This approach lifts the computational burden by having smaller amounts of rectangles per entity and lower population sizes. This decreases the number of rectangles and images needing to be rendered each generation which is the most computationally intense part of the process. The single layered GA is restricted to simplicity since the algorithm scales poorly. As the number of rectangles increases the time for convergence increases by a large magnitude (the relationship cannot be accurately described with certainty since it is not due to the complexity of the algorithm but rather the nature of how it converges). This scaling does not apply to the layered approach which scales seemingly linearly with the increase of rectangles. The layered approach also fixes the issue of rectangles with a deeper ordering not having effects on the image since they are buried behind so many other rectangles.

The series of populations is a well-ordered set L with hyperparameter d populations (each population being the same structure as P defined in the previous section). The layered algorithm iterates through each population in L running the same GA as defined in the previous section with the difference being the image generation function. Each population overlays its rendered rectangles on top of the image created by the preceding generation. This is done with a modified version of $g(E)$ defined as $h(E, I)$ that renders the output onto the already existing image I. Take any layer $l_n \in L$, the image that l_n creates is $h\left(l_n, h\left(l_{n-1}, h(l_{n-2}, \dots)\right)\right)$ where the ellipsis continues until l_0 in which the original $g(l_0)$ is used instead of using $h(E, I)$ since there is no preceding image.

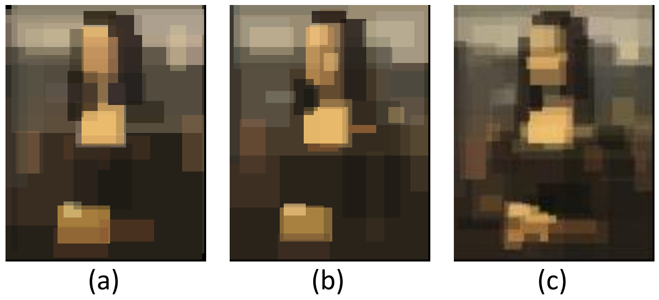

FIGURE 6: Mona Lisa images created with layered GA: 5 rectangles per 7 layers (a), 1 rectangle per 35 layers (b) and 2 rectangles per 35 layers (c). All with a population size of 18, mutation chance of 0.05 and 1200 generations per layer.

The algorithm iterates over the layers once, each layer going through generations using $h(E, I)$ to overlay on the cached image of the previous layer. Each layer converges much quicker than using a single layer and to a lower minima. In the single layer approach, it is common for rectangles to be “lost” behind others since they had a low ordering and had been overlaid by a higher one. The layered algorithm mitigates this slightly by evaluating the loss of a rectangle before the later ones are overlaid. Population layering makes the algorithm scale (with respect to rectangle count) linearly, converge faster, and converge to a lower minima. However, when the algorithm is scaled up it approaches the same storage cost as other standard compression algorithms without being accurate enough to compete against them.

CONCLUSION

There are many potential ways that this approach can be improved, using a more efficient way to store the rectangles is plausible. An approach could be made to have the populations use a different shape or a combination of shapes other than rectangles. Allowing for circles or curves of some kind would greatly reduce the amount of shapes used on sections of images that have curved edges. A possible efficiency improvement could be parallelizing the algorithm by iterating through the generations on multiple layers simultaneously. Another thing to test is initializing the colors of the rectangles to the average color of the reference image for the space that that rectangle occupies; this could greatly reduce the number of generations needed since the color will already be ideal. However, initializing all entities to the same values creates low genetic diversity in the population.

A known issue with both the single layer and multi-layered implementations is that rectangles may end up having zero width, height, or opacity. This would cause the affected rectangle to be completely invisible which would make it useless data that needs to be stored, which defeats the purpose of a compression algorithm. If a rectangle in the population is contributing to the error greatly and it receives a random mutation that causes one (or more) of the conditions that would make it disappear then the error will consequently go down. Since this is a likely case to happen the algorithm suffers greatly.

This algorithm will likely never find use in image compression but hopefully will find other uses. For example, assume there is a heuristic function that evaluates the usefulness of an image without the presence of an exemplar image. An image could be generated with a layered GA and then evaluated with this heuristic. Through genetic manipulation, an image could be found that satisfies the function. This could especially be useful if the heuristic function is not differential since it would not be possible to apply deep learning. Perhaps some implementation of a layered GA of a similar design may find use with an entirely different application outside of image generation. Since having multiple populations work together to solve one problem is computationally cheaper this could potentially find use with a problem that is unrelated to images.

REFERENCES

Barricelli, N. A. (1954). Esempi numerici di processi di evoluzione. In *Methodos* (pp. 45-68).

Sourabh, K., Chauhan, S. S., & Kumar, V. (2021). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications, 80*(5), 8091-8126. doi:http://dx.doi.org/10.1007/s11042-020-10139-6