Controlling Image Synthesis using Artificial Intelligence through Genetic Algorithms and Aesthetic Fitness Criteria

by Krish Jain
Advisor: Steve Tanimoto, Professor of Computer Science at the University of Washington
(Summer 2019)

ABSTRACT

In this project, a software was developed to create aesthetically pleasing images using computer vision and PixelMath, an image processing software. This project uses genetic algorithms, which is a process of applying mutations, crossover, and aesthetic fitness criteria over multiple generations, to create the most aesthetically pleasing images.

SUMMARY

Genetic algorithms are based upon Darwin's Theory of Natural Selection, which states that organisms that have a higher fitness, or ability to survive and reproduce, will have a higher chance of reaching the next generation. Through random events like mutation and crossover, some have higher fitness than others, and if a mutation happens to create a higher fitness individual, that mutation is likely to be passed down throughout generations. This type of AI process tries to mimic a natural occurrence, which results in individuals that adapt to the conditions over time. In this project, genetic algorithms and related techniques are used to automatically come up with images that are better and better from an aesthetic "fitness" criterion.

In this project, we first generate an initial set of images, each image being represented in as tree structure. Then a mutation is applied on all these images by choosing a random non-terminal from the original tree, calling the formula generation specifically on that non-terminal and replacing the subtree under the chosen non-terminal with a new formula. A similar process occurs with crossover except there are two trees (two parents) that combine to make one child tree. A fitness criteria is applied at each stage by getting the RGB values of each pixel and doing calculations such as standard deviation, mean, median, colors present in the photo, and mode. The highest fitness images are allowed to pass onto the next generation and this process repeats. Each of these processes will be discussed in more detail later

SOFTWARE

Program Flowchart

The flowchart displayed in Figure 1 shows the genetic algorithm process used in this software.

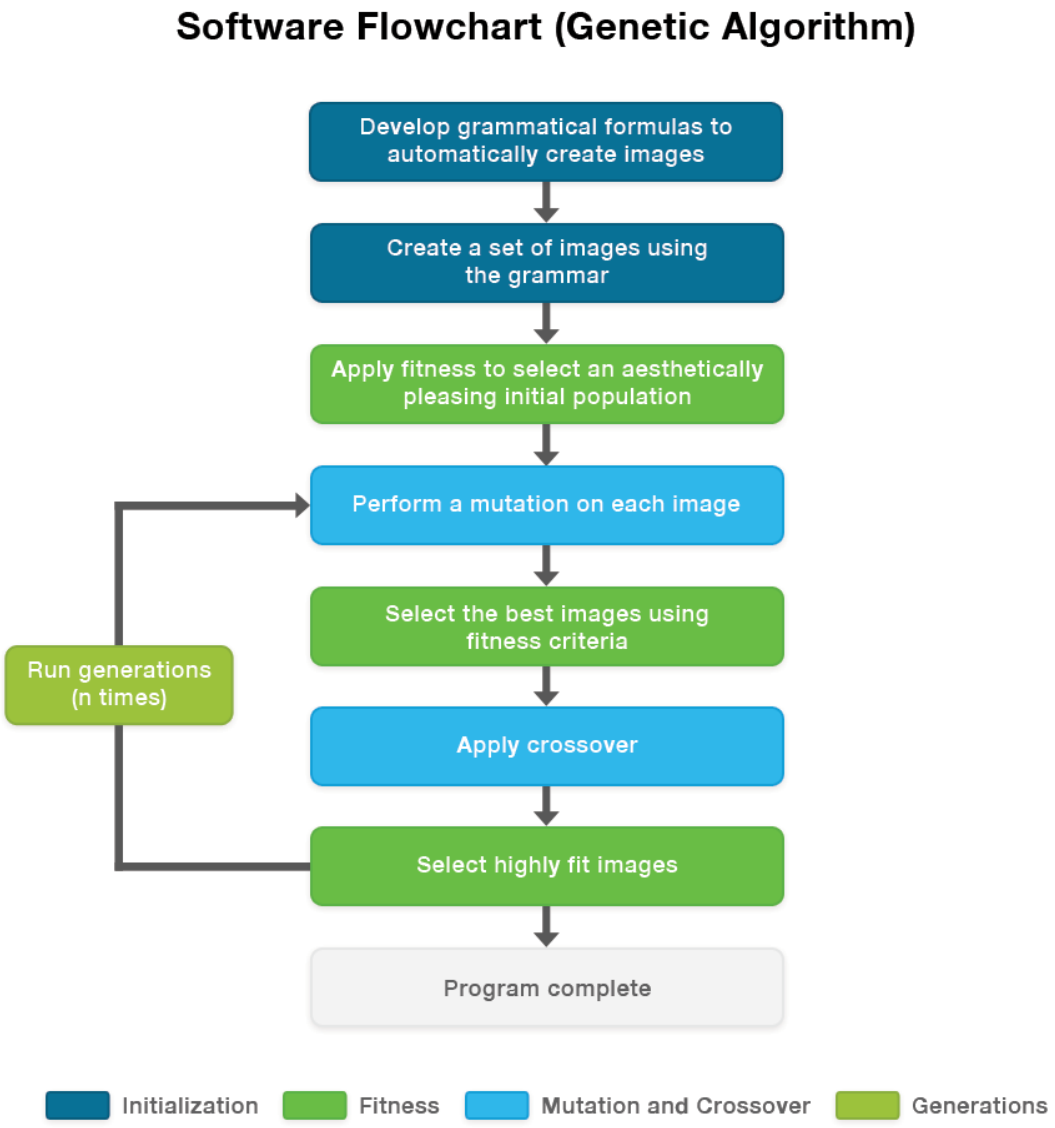

Figure 1: A description of how the genetic algorithm process is implemented in the program.

Data Structure

Each image has four critical parts that are used throughout the program:

1. **Tree Data Structure:** The tree is essential in order to successfully alter a formula through mutations and cross-overs. To avoid parsing through a formula and converting it to non-terminals each time, the formula is stored as a tree.

2. Formula: The formula is stored so that the image can be generated at any time. Grammar based formula generation is used where non-terminals (e.g. E225, P, E1P, EE, etc.) represent symbols that can be expanded on and terminals represent symbols that can't be expanded upon.
3. Dictionary: The dictionary is used to maintain the counts of each of the non-terminals. This is important when adding nodes to the tree because there could be multiple instances of a single non-terminal. Each non-terminal in the tree is stored with a number using the counts in the dictionary such as E1P 0 so that two instances of E1P, for example, can be distinguished from one another.
4. Fitness Value: The fitness value is stored along with this information so that deciding if an image has high enough fitness to move onto the next generation is easy because each image's fitness is easily accessible.

Initialization

Using the tools available in PixelMath and grammar-based formula generation, the initial set of images is created. Just like a sentence in English, each part of the formula can be written as parts, similar to how a sentence in English requires a noun phrase and a verb phrase. The parts that can be expanded on are called non-terminals. The non-terminals present in this code are P, E255, E1P, EE and EI, which either represent a terminal, such as 255/x or y, or lead to more formula generation such as E1P + E1P. Each non-terminal represents a function whose range will be within certain values. For example, E255 represents a function that only returns values in between 0 and 255.

First, a production string is defined which has a non-terminal on the left-hand side and its value on the right-hand side. There are multiple such entries per each non-terminal. One example entry would be E1P ::= abs(EE), where ::= is used to parse the left and right hand sides.

Now, formula generation will be discussed, which happens recursively. Among the choices for what each non-terminal is equal to, the formula generation randomly chooses one of these options. Next, for each element in the right-hand side equivalent of the non-terminal, it either calls the formula generation again with the non-terminal present in the right-hand side, or adds the terminal to the output items. While this is happening, each time that a non-terminal is detected, it is added to a tree that has its root node defined as a P. P means pictures and it is the root node because it is present in every formula. As the formula is being converted to a tree, a dictionary is being added to that contains each non-terminal as the key and its count as the value. There is also a depth counter, which restarts the formula generation if it goes on too long. At the end of this process, an image is fully initialized and is ready to run through generations.

One of the automatically generated image is shown in Figure 2:

Example Image (resembles aurora borealis)

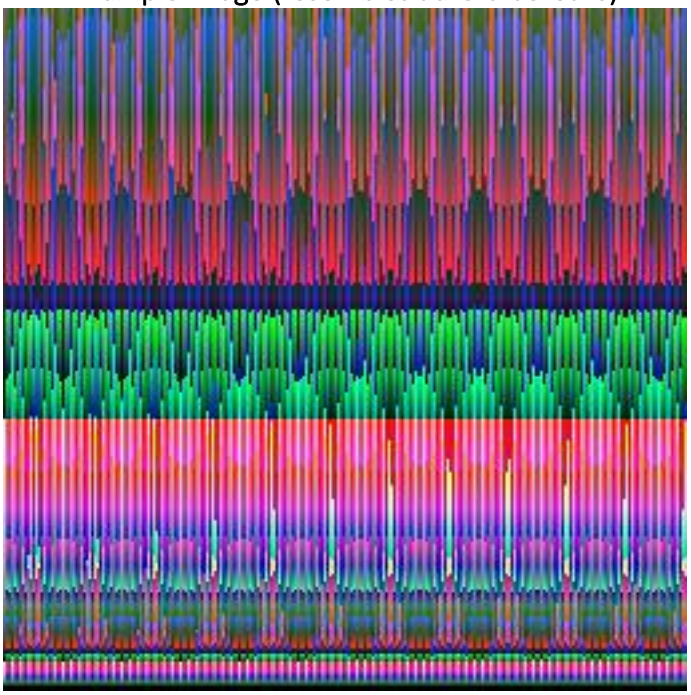

Figure 2
Formula: hsv(log(y), abs(cos(x)), log(y))
This is one of the formula generated image. The non-terminals were replaced by terminals in this formula. For example, the E1P in this generation was replaced by log(y), and so on.

Mutation and Crossover

The mutation function is necessary to ensuring that there is diversity between pictures. This allows for possible improvement in pictures.

First, a copy of the input tree is created to avoid changing the original tree. A non-terminal present in this tree is randomly chosen using the counts present in the dictionary and the random module. Then, the formula generation is called with that non-terminal, resulting in a smaller subtree that needs to be added to the main one.

The children nodes to the non-terminal in the copy tree are all removed. By traversing the tree, the nodes present in the subtree are all added to the main tree. Finally, the image is updated by adjusting the counts of each non-terminal in the dictionary and changing the naming of each non-terminal in the tree.

The cross-over function is very similar to mutation. The only difference is that there are two input trees. One of these input trees is chosen to be the base and the other to be the replacement. A new combined dictionary is created which has the counts of each non-terminal. Then, a random non-terminal is chosen using this combined dictionary.

Finally, that non-terminal is found in the second tree, and the process that happened in mutation is mimicked.

In Figure 3 is the original image synthesized by the formula generation. The mutated and crossover version of the same image are shown in Figures 4 and 5 respectively.

An experiment is also shown in Figure 6 where the picture is a solid lime green color with a very low fitness. As mutations occur, the image from the past generation and the mutated one are compared. The better one moves on and as is visible from the fitness in the title of the image, the fitness keeps on increasing.

Original Image

Figure 3
*hsv(log(y),cos(x)*sin(y),log(y))*
This is one of the original images generated by initialization.

Mutated Image

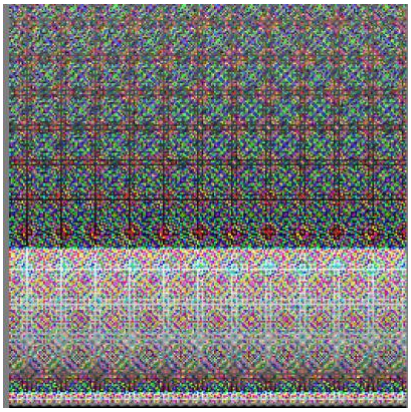

Figure 4
*hsv(sin(x)*cos(y), cos(x)*sin(y), log(y))*
*After a mutation is applied on the picture, the picture looks very different. The E1P non-terminal, which was log(y) originally was replaced by sin(x)*sin(y).*

Crossover Image

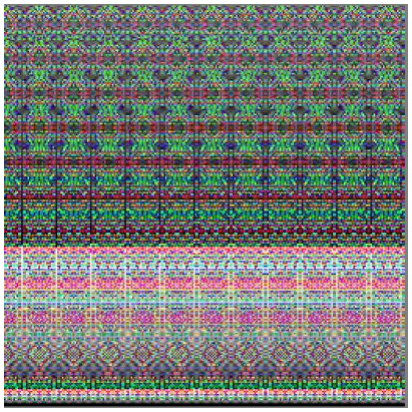

Figure 5
*hsv(abs(cos(y)),cos(x)*sin(y), log(y))*
In this crossover, the log(y) term is replaced by (abs(cos(y)). This term was present in the second parent and was therefore present in the child.

Effect of Mutating a Simple Image

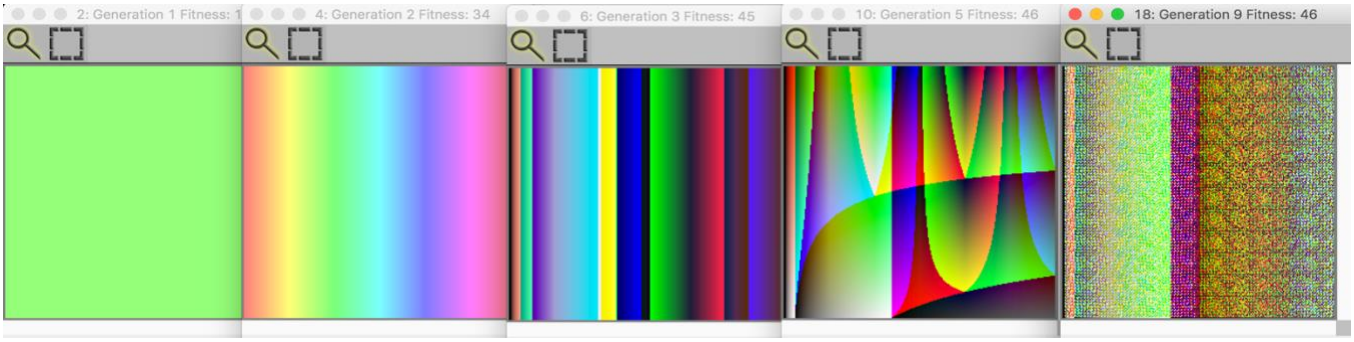

Figure 6

This was an experiment to see the effect of mutation. The fitness starts from 12 and goes all the way up to 46. After 9 generations, the image at the end has many more layers and much higher fitness than the original. Some generations in between resulted in bad mutations, or ones that lowered the fitness, which resulted in the same images twice. These weren't displayed so that the full transition is displayed.

Fitness

The fitness function is essential to any genetic algorithm. The fitness function takes an input list of images, computes the RGB values, and uses statistics from the fitness file created to give a numeric value to the beauty of the picture.

The statistics computed are mean, median, mode, most, standard deviation, and colors present. The mean is the average of all the RGB values of each pixel. The median returns the middle value in the dataset of RGB values when arranged in ascending order. The mode returns the element that occurs the most, and most returns the count of how many times that element appears. Standard deviation measures the amount of variation in the data set and is calculated through the standard formula. Finally, colors present measures how many pixels fall under each color range. Based on how many pixels are found in each range, the picture either contains the color or doesn't.

Based on the value that each of these functions returns, there is also a function that computes how much fitness value each statistic gives. For example, colors present was decided to have more weightage than mode and most, since they are related, so colors present accounts for about 40% of the fitness value.

Finally, the fitness function changes the fitness value associated with the picture (the 4^{th} element present in the list) to match the one it just computed.

Below in Figure 7 is a table of an experiment that was conducted where the program was run for 4 generations. In addition to the top 6 out of 20 (high fitness) images to move onto generations, the middle six images (average fitness) and the last 6 images (low fitness) were also taken. This experiment shows that over generations, no matter what the initial population is, the fitness increases and better images are produced. In the low fitness images, there was an 80-100%

increase in average fitness value after just 4 generations. The lower the fitness of the base images, the higher the percent increase in average fitness was achieved.

Average Fitness Values After 4 Generations for Varying Initial Populations

	Initial	Gen. 1 Mutation	Gen. 1 Crossover	Gen. 2 Mutation	Gen. 2 Crossover	Gen. 3 Mutation	Gen. 3 Crossover	Gen. 4 Mutation	Gen. 4 Crossover
Images 0-6 (high fitness)	38.66	44.33	45.00	45.66	46.00	46.33	47.00	46.66	46.66
Images 7-13 (average fitness)	28.33	30.00	31.33	32.66	35.33	35.66	36.00	36.33	36.00
Images 14-20 (low fitness)	16.33	19.66	29.66	32.33	32.33	33.00	34.00	33.66	33.00

Figure 7

This table shows the average fitness values for the mutated and crossover generations after 4 generations. There is a direct correlation between lower initial fitness value and a greater percent increase in average fitness in the last generation. There is some variation because the crossover doesn't necessarily result in a higher fitness after that generation. In crossover, the generation isn't compared to the previous so the average fitness can decrease.

Generations

It was decided to increase genetic diversity, the mutated version of a picture and the original picture will be compared. The higher fitness one will move onto the next generation. This is because sometimes, if a picture has a really high fitness, say 46, and the mutation causes it to drop to 42, it is still a good picture. The problem is then a different picture with fitness 40 is eliminated, and then all the pictures in the end look very similar since a mutation doesn't make too big of a change. A similar decision was made in crossover. It was decided that only one child of a base tree can move onto the next generation.

Also, initially, to ensure that the generations weren't applied onto low fitness images, a base set of images of 20 was refined down to 6. These were the ones that had mutations, fitness, and cross-over applied to them.

The user has the option to control how many generations happen. Also, after all the generations are done, the 6 original images and the 6 modified images are displayed, with an O and M in the titles signifying modified and original.

Example of an Interesting Piece of Code

Typically, when developing genetic algorithms, programming cross-over is the hardest part of the program. When I was thinking of the data structure to use to store formulas to be edited in mutations, I realized that a tree structure would be the most beneficial even if it would be hard to code at first. In the code for mutation, it allowed for choosing a random non-terminal and removing the subtree underneath it straightforward. As a natural extension, it made coding cross-over very simple, made the code understandable, and easy to read.

Now, I will analyze this code. Here I just want to display that piece of code from the mutation function:

```
[code begins]
random_choice_number=random_choice+" " + str(extra_number)
        for node in mutated_tree.traverse("P"):
                for child in mutated_tree[random_choice_number].children:
                        mutated_tree.remove_node(child, random_choice_number)
tree2=Tree()
        node2=tree2.add_node(random_choice_number)
        non_terminal_dictionary2 = {"P":0, "E1P":0, "EI":0, "EE":0, "E255":0, "COND":0}
        formula2=gen_phrase(random_choice_number,node2, tree2, non_terminal_dictionary2)
        tree2_list=[tree2, formula2, non_terminal_dictionary2, 0]
for node in tree2.traverse(random_choice_number):
                for child in tree2[node].children:
                        mutated_tree.add_node(child, node)
[code ends]
```

Here, random_choice represents the random non-terminal that was chosen to be replaced. First, a random number, extra_number, is added to random_choice to match the node that will be present on the tree. Next, while traversing the node, all the children of the random_choice_number, the random non-terminal, are removed.

Then, a new tree is initialized with its root node as the random non-terminal. It also has a dictionary created and then is called into formula generation with that non-terminal. This way, a new subtree is created which is our mutated version. This is what the original tree needs to be replaced with.

Finally, the subtree is traversed through and every node that is detected on this tree is added to the mutated tree that is returned.

To code cross-over, instead of generating the formula from the chosen random non-terminal, I used the non-terminal (and the subtree underneath) from the second parent as a replacement.

Key Learnings

This project enabled me to learn many invaluable lessons that I wouldn't have learned otherwise:

- Power of Artificial Intelligence: I learned about what you can do with AI and the limitations of traditional programming. The genetic algorithm that I implemented is a learning system because it takes information from one generation onto the next and is learning.
- Efficient program: At first, while running the fitness function, I was running a separate loop through all the pixels (65536 of them) each time a statistic had to be generated. I optimized this strategy by condensing everything into one loop and that made the program run much faster.
- Right data structure: I learned about data structures like trees and how choosing the right way to store information can make coding later much easier. Had I not chosen to use a tree structure, it would've been almost impossible to perform a cross over function correctly.
- Art is subjective: I originally created a second version of the fitness function, which had multiple changes such as giving 8 colors a lower fitness to avoid completely rainbow like pictures. This ended up creating worse pictures, and this taught me to remember that this is art and not a science. Sometimes creating a fitness algorithm that is correct from a mathematical standpoint may not translate into the most beautiful pictures.
- Importance of Documentation: I learned of the importance of documentation and readability, for the both the programmer and others. While reading code, it is sometimes hard to interpret exactly what the programmer aimed to do, and it is helpful to document exactly what happened at each step so it is easy for someone who knows nothing about your code to understand it. It is also helpful for debugging because the programmer can pinpoint which area of code is creating issues.

CONCLUSION

After running the program for multiple generations, I was ending up with some beautiful images. Below in Figures 8,9, and 10, I have some instances of running the program over several generations. In each set, the first set of pictures is the initial set of images, the next 6 are the last mutated generation, and the bottom 6 images are the final images after the last crossover.

In some cases, I observed that over the generations, some images remain the same. This is because the original image had such high fitness that the mutations and the crossover couldn't create an image with higher fitness.

I also noticed that over generations, the fitness value of some images kept on getting higher and higher, and eventually these images reached such a high fitness value that they wouldn't be replaced by its changed version.

Image Set #1 (Number of generations: 3)

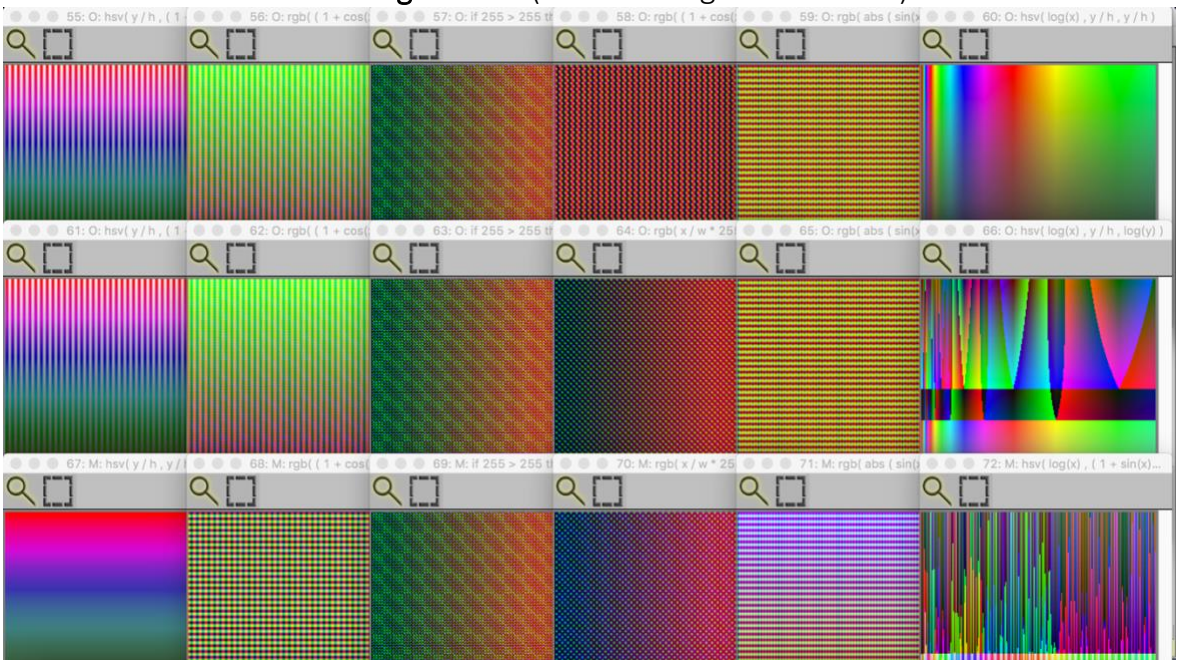

Figure 8

The third image in this set has all three of its forms the exact same, meaning that this image had a high fitness originally and the original picture was chosen over the other ones each time. The last image was consistently mutated and had cross-over applied that changed the picture, meaning that according to the fitness criteria, it is getting much better over generations.

Image Set #2 (Number of generations: 10)

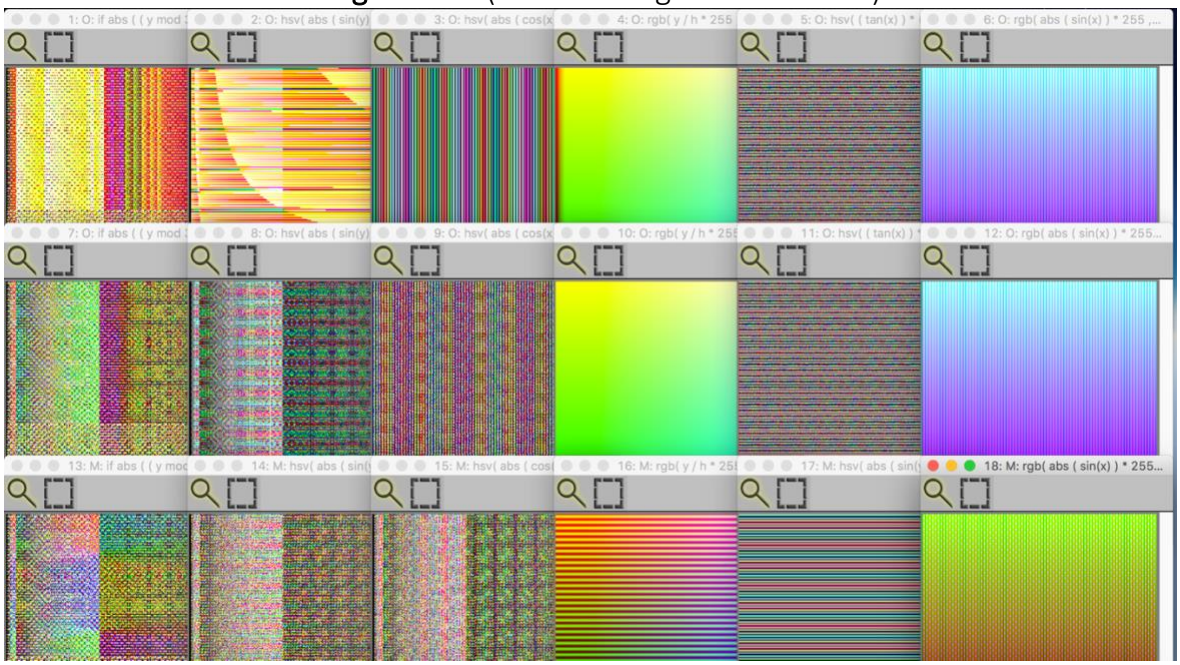

Figure 9

In the fourth image, the two colors present didn't give the picture a high enough fitness, so when a version with more colors came from the cross-over, that was preferred. First image was consistently mutated to create an end product that had high fitness. These images ended up to be nice because this was run for 10 generations.

Image Set #3 (Number of generations: 4)

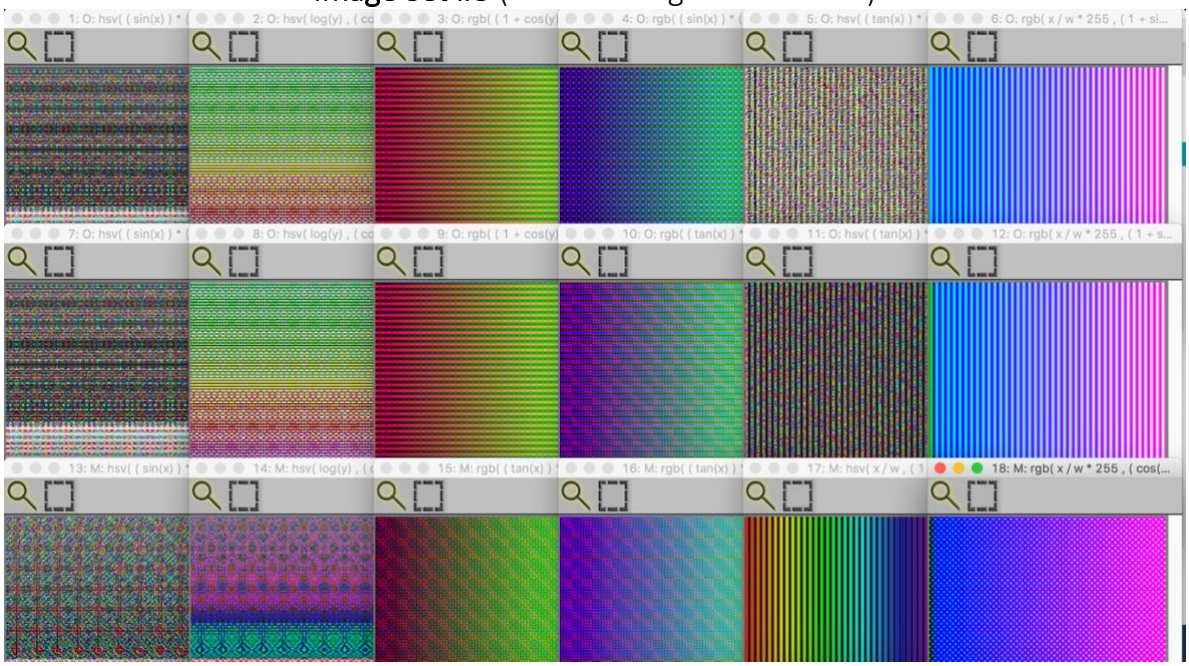

Figure 10

The third image demonstrates a good illustration of cross-over. The cross over image has the same gradient of the original, but the pattern was added from the other parent of the child. Some of the images like the fourth and sixth one remained largely the same, but the other three were mutated some each generation to create an end product that was much different than the original.

Cool Image 1:

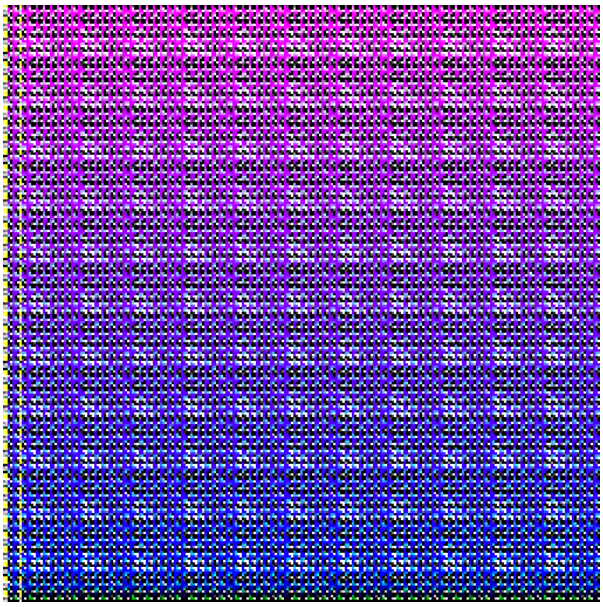

Figure 11

Formula: if log(x)>sin(x) then if tan(x)>cos(y) then 255(tan(x)*tan(y))+abs(cos(x)) else rgb(y/h*255, cos(x)*sin(y)*255, log(y)*255) else hsv(log(y), cos(x)*sin(y), sin(x) *cos(y))*

*This is one of the cool images that comes from a complex formula. The rgb and hsv contributed to the color gradient that was formed while the tan(x)*tan(y) term contributed to the cool triangular like pattern that was formed on the picture. The log function also allowed for the dark color in the bottom as log quickly accelerates and creates a different spectrum of colors.*

Cool Image #2

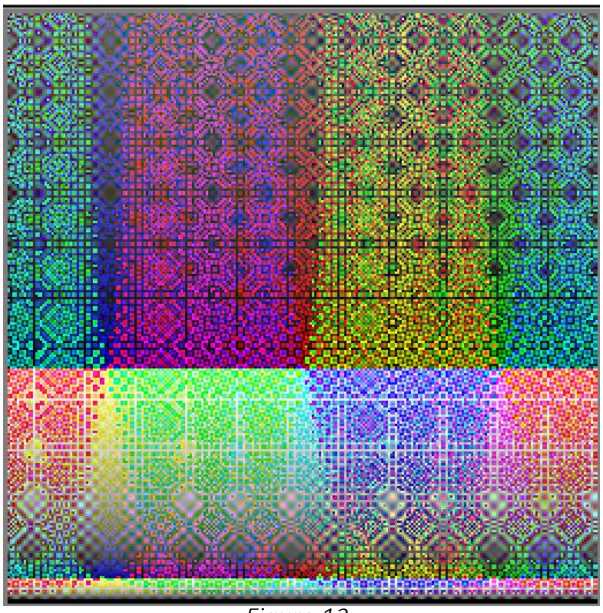

Figure 12

*Formula: hsv(x/w, cos(x)*sin(y), log(y))*

This picture is a good illustration of how a simple formula can lead to very cool looking image. The diamond like repeating pattern comes from the trigonometric functions and the wide variety of colors from the log.

REFERENCES

Cummins, Chris. “Grow Your Own Picture Genetic Algorithms & Generative Art.” Grow Your Own Picture, chriscummins.cc/s/genetics/.

Jain, Shubham. “Genetic Algorithm Introduction & Their Application in Data Science.” Analytics Vidhya, 25 June 2019, www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/.

Kromkamp, Brett. “Knowledge Management.” Java Tree Implementation, 31 Dec. 2016, www.quesucede.com/page/show/id/java-tree-implementation.

Tanimoto, Steven. An Interdisciplinary Introduction to Image Processing: Pixels, Numbers, and Programs. MIT Press, 2012.

Tanimoto, Steven. “An Interdisciplinary Introduction to Image Processing: Pixels, Numbers, and Programs -- Support Website.” University of Washington, 4 Nov. 2016, https://pixels.cs.washington.edu.