Graham Stuard
Wednesday, December 6th, 2023
CS 411: Genetic Programming

Highest Form of Flattery: Image Imitation and Generation in GP

ABSTRACT

Genetic programming (GP) struggles with image generation and manipulation programs, compared to other forms of artificial intelligence (AI), like neural networks. However, because they create far more readable individuals, image manipulation GP is easier to study, and interesting to see the techniques it can devise. In this way, image generation or matching can be treated like a problem that might be on PSB1 or PSB2[1], but with more unique stacks and instructions–its goal is just as much to use control flow and its executable stack (exec stack) instructions correctly, as it is to create interesting images.

The broad goal of this experiment is to test whether the current instructions are good enough to allow GP to mimic and modify images in an efficient and timely manner. If that proves true, this code base could be used for many subsequent experiments on image GP with very little modification to the overall instructions and system. I did this by running the program many times with the entire default instructions set, counting the quantity and placement of each instruction or literal, and by running tests with some of the most important instructions removed, comparing the final best individuals. One run of the first, used lexicase selection instead of tournament selection, which proved to have a small negative effect across the board, most likely due to the small list of training cases.

I find in my testing that the current instruction set is complete enough to be used for more image manipulation experiments in GP, although it relies far too heavily on two instructions,

[1] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 785-794.

`area_get_by_color` and `image_fill_area` (AKA `image_change_area_color`), especially for image combination.

1 INTRODUCTION

Image generation in GP can be frustrating, because it is so far behind other methods. Therefore, it is important to keep in mind that the goal is just as much to create an interesting program using control flow and reading, then responding to inputs, as it is to paint a picture.

With that being said, the world of image GP opens up three key questions about implementation and results. Those are:

1. How should something as abstract as images be judged on fitness? What is better–slowing down the program to give a more human-seeming fitness function, choosing something objective that can mimic a subjective fitness, or choosing an objective fitness function that cannot make an "artistic" result?

Although a potential different question that could be researched would be using a neural network as a fitness function, for this experiment all of the fitness functions were objective, as it focused more on testing which instructions are best, rather than creating art.

2. How much autonomy should the GP implementation be given when it comes to image drawing? How basic should or can the instructions be in order to guarantee that the algorithm has more expressability with the resulting images?

This was a real departure from the rest of the research done around image GP. Most other image based instruction sets have a lot of very structured techniques (such as random walks, drawing specific shapes or equations, or copying whole elements from other images). The instruction set in this implementation is entirely based around the image grid, and applying different colors to different locations–the most structured any single instruction can get is with the area stack, and everything else was left to the program to devise.

3. Should the GP be creating individuals that receive one image, multiple images, or none at all? And within that, should the individuals be programs that create unique images, programs that match the given images, or should the individuals be the images themself, modified with unique instructions?

For this project, again because the goal became to test different base methods of manipulating images with GP programs, it made the most sense to have images that receive multiple images and have the individuals be programs that match the images.

2 IMPLEMENTATION SPECIFICS

In order to handle image generation, five stacks and one library were added to the base GP system from October. The library was image-grid[2], a library created by one Tom Helmuth to manage images as a three dimensional vector of floats representing RGB values, and to use the mikeraz[3] library to display and handle the actual images themselves.

The new stacks were :float, :image, :color, :pixel, :area, :input and :boolean. Float and boolean function as normal in Push–they both have a slightly smaller instruction set (especially float, compared to int), because they are used for relatively few instructions within the program. :color is a stack of vectors with three floats that represent an RGB value, or basically a single cell within an image–this was created so that float values from other things did not get confused with color values from the images. The :area stack contains two dimensional vectors of :pixels, which are vectors of two ints, representing a location on the input image. Unlike float and color, integers and pixels can be transferred between each other.

Finally, the :image and :input stacks are similar but different. The input stack existed before, but it used to be a map, and now it is a stack that always duplicates its top element

[2] https://github.com/thelmuth/image-grid/tree/main
[3] https://github.com/mikera/imagez

whenever it would be taken in a Push instruction, and always has one element on the top of the stack. This is a safeguard so that the programs cannot simply take the input and put it onto the image stack, but so the image can be compared at all times. The image stack simply contains image-grids.

3 BACKGROUND & RELATED WORK

In my research, I decided on three main goals that an image generation GP system could have. Previous research has easily dedicated the most time to unprompted image creation.[4] The individuals in these systems are more often images than they are programs that return images. The other two are prompted image recreation, and prompted image combination. This GP system focuses on the first.

The difference between ‘prompted’ and ‘unprompted’ image-focused GP is on the inputs given to the system. In my GP system, which is a prompted system, depending on the fitness function selected, either one or two images will be given as inputs, and the fitness value of a given program is how closely the RGB values of each pixel align (in cases where there are two inputs, the fitness value is based on the image with more of a difference in pixel values). In unprompted image GP, images are created without an existing input image. Ultimately, the difference here is important, because unprompted image GP is usually focused on the output image solely–can the system create an interesting or good looking piece of art–whereas prompted GP can sometimes produce an interesting output image, but more often the goal is to use control flow and the given instructions correctly, creating a similar image as given, but solving the problem in an interesting way.

[4] E. M. Fredericks, A. C. Diller and J. M. Moore, "Generative Art via Grammatical Evolution," 2023 IEEE/ACM International Workshop on Genetic Improvement (GI), Melbourne, Australia, 2023, pp. 1-8, doi: 10.1109/GI59320.2023.00010.

4 ANALYSIS OF INDIVIDUALS

During informal testing, five categories of individuals were made clear. Each one gets distinctly better at solving the problem, and are characterized by their semantic diversity. In this section, each one will be explained and examples will be provided.

The first of these, and most obvious, are randomly generated programs that do not get any images onto the image stack. Using a population size of more than 10, (so once I wasn't actively writing code), these would appear for either just the first generation or not at all. This is a simple category, and they get removed quickly, because if even a single one of their test cases does not have an image, their error value on that case will be infinite (`##Inf`), and therefore the total error will be as well. Tournament selection removes these failing individuals faster than lexicase, as a result.

```
(1.0 area_get_by_color [1.0 1.0 1.0] integer_% 1.0 image_compare
float_* float_% [0.0 0.0 0.0] area_get_by_color
area_get_by_color integer_% float_- float_- [0.0 1.0 0.0] [0.0
1.0 0.0] integer_-)
```

After these obvious weak programs are removed from the population, the "static" group is formed. Most of this code is WORD FOR USELESS CODE vestigial, but somewhere in the program will be `image_create_uniform`, which is the only way images can be created by a program when there is none on the image stack already. The image stack was implemented without literals because having multiple of them would bog down the already large instruction set. These programs will do well or okay on the test cases that are monochrome, and often get perfect scores if the entirely black or white test cases are being used, but their total error is about as bad as any program can be that does not totally fail any test case.

```
(area_get_by_color integer_% image_fill_area image_compare
float_* float_% float_< area_get_by_color image_compare float_-
image_compare_pixel_float [0.0 1.0 0.0] image_create_uniform
color_average_grayscale pixel_get_color)
```

The change between static and dynamic programs is less noticeable, but usually the error vector will stay the same for some generations (5-20, depending on the population size and instruction set) until they all lower by around 1.0. (This change is mostly noticeable because some of the whole number error values will become decimals, and the printed report will get much longer–kind of a lucky implementation of warning the user when the GP has made progress). This is usually because of a single `pixel_get_color` and `image_change_pixel_color` pair.

The above step is often skipped (especially in runs with high populations). Similar to the above example, this significant jump in fitness comes from a single pair of instructions, usually `area_get_by_color` and `image_change_area_color`. Unlike the pixel pair, this is not just placing the exact value in one spot, but blanketing an area with more correct values. It takes longer, usually, to start to be used successfully by individuals, most likely because it requires more inputs to not no-op, and if the float value given is not large enough, the area taken might be too small/empty anyways.

Finally, the best phase of these individuals is when they begin to properly utilize exec instructions. This realization during the first round of testing is the reason that in the second round, the exec instructions were prioritized. As will be expanded on in the results section, this category has the most significant leap in fitness, and is the proof of the fact that the GP does somewhat solve the problem.

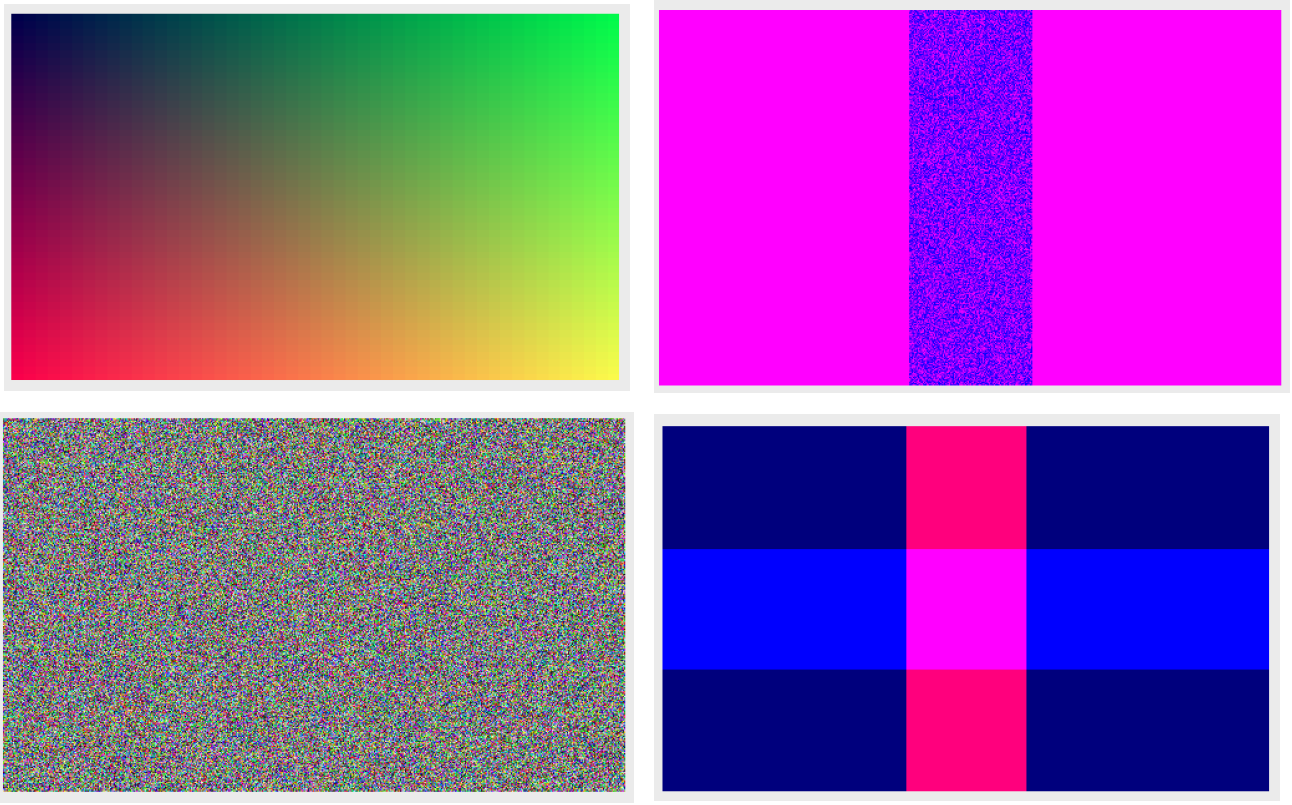
No program created by this GP was able to get a complete match on every test case, especially with the more complicated images like the random pixelation test case. However, it is possible for an individual to be generated that would, which would look something like this:

```
(integer_height exec_do*count (integer_width exec_do*count
(int_dup int_dup int_to_pixel pixel_get_color
image_change_pixel_color)))
```

5 EXPERIMENTAL SETUP

The first experiment consisted of three trials, two with tournament selection that are the same and one with lexicase. This was mostly to see if the programs could consistently glean significantly low error values on all images, not just the easiest ones (the monochrome training cases).

Below: the non-monochrome test cases, of varying degrees of difficulty.

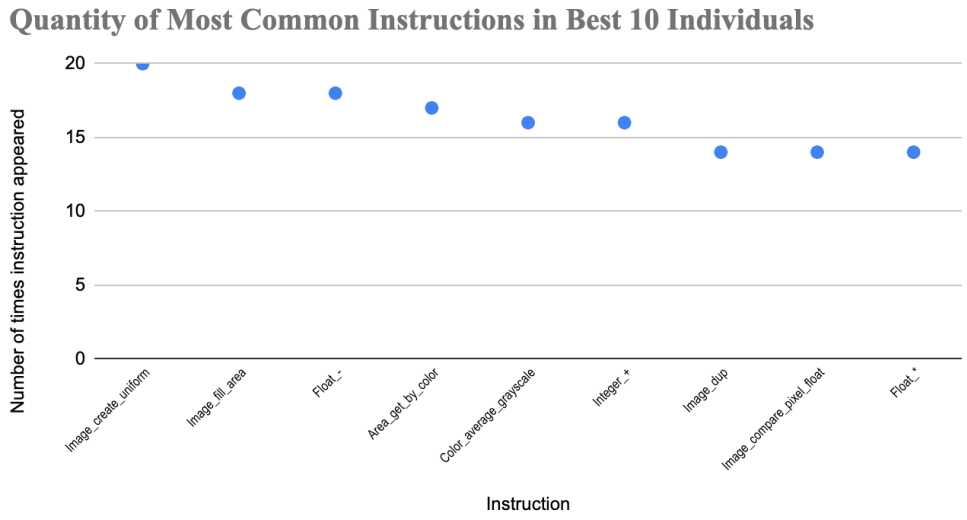


For these questions, all of the test cases were used, except for the second run of tournament selection, which was missing the gray, white and black monochrome test cases. For all of these runs, the instruction set was the default, the maximum generations was 65, the population size was 30, and the max initial Plushy size was 16.

6 RESULTS

For the first three runs, with the full instruction set and much larger hyperparameters, some pretty strong individuals were created for some cases, but none had very good solutions to the hardest test cases–the completely random test case and the gradient test case. Upon reflection, the random test case can really only be solved sufficiently by the exact program from section four, which encourages overfitting and is probably too high of a standard for only three trials to achieve. Hopefully more runs could potentially get an individual that comes closer.

With that said, the more useful data from these runs was the quantity that each instruction appeared. From each run the top ten individuals were compiled and the quantity of each instruction was compared to get the most common instructions:



While there may be some randomness here, as it would be surprising for any of the individuals behaviorally benefitted from Float_* more than they did from any of the pixel instructions, it does confirm the hypothesis that the first thing these individuals learn how to do and are rewarded for is creating uniform images of the same average color of the input image, and then finding and filling large areas that have a uniform or close to uniform color.

The second run of tournament selection created the best individual overall, with a fitness score of around 36,250 (the worst score besides ##Inf is 112,500, and an entirely random image would average a score of 56,250) on average per test case, however the top 10 individuals performed worse overall. This could be just a random occurrence, or it could be because this second run did not use the monochrome test cases, resulting in less individuals simply getting a decent fitness score because they solve those easy training cases.

The best individual, as mentioned above, was:

```
(integer_+ exec_dup (integer_% 0.0 integer_- image_fill_area
exec_do*count image_compare 1.0 1.0 float_+ 1.0
area_get_by_color area_get_by_color image_compare_pixel_float
float_- float_- [0.0 1.0 0.0] float_- color_average_grayscale
image_create_uniform color_average_grayscale
image_create_uniform image_fill_area))
```

The second round of experiments is worth mentioning because it was attempted, but it did not return any interesting results. The trial groups for this were:

1. Removing all but the core pixel functions (`image_compare_pixel_float`, `image_compare_pixel_boolean`, `image_change_pixel_color`, `pixel_get_color`, `int_to_pixel`, and a few exec and integer instructions)
2. Removing all but the core area functions (`image_fill_area`, `color_average_grayscale`, `area_get_by_color`, and a few exec and integer instructions)
3. Removing almost all the integer and float math functions (`integer_-`, `integer_*`, `integer_%`, `float_+`, `float_-`, `float_*`, `float_%`)

4. Removing all the core boolean functions (the boolean stack, `exec_if`, `image_compare`, `color_compare`, and `image_compare_pixel_boolean`).

Unfortunately, these were barely able to achieve lower average scores than 56,250 (the random cutoff), even with the monochrome test cases. This is most likely because these tests were too limited–for timings sake they ran with a max Plushy size of 9 and a population of 10–and because limiting the instruction set does limit the amount of the search space they can cover. Below are the disappointing results:

Core pixel functions	Core area functions	Remove int/float	Remove boolean
`54352.99`	`54352.99`	`41599.99`	`41599.99`

7 DISCUSSION

There were two big flaws in this testing over all–the time complexity of these runs, and the potentially flawed training cases. While they produced some interesting individuals, it may have been more interesting to test the overall program in a few, dedicated, longer runs, instead of trying to save time for a second experiment, and trying to do half the runs with a different training set than the other.

With that said, the ability especially of the area function to generalize on the gradient case, and get a similar looking image with a far better than random score is definitely interesting, and proof of the fact that this setup of an area stack has some merit for generating images. Especially for the potential combination of images, where there could be area functions testing where two areas are similar, knowing this was the productive section of the best individuals is useful knowledge.

8 CONCLUSION

Overall, the most time consuming part of building my code base was getting the different instructions to work together well without removing the autonomy of the GP system/making the problem too easy.

If I were to continue with image generation GP, my next experiment would focus entirely on different fitness functions–I would probably use upwards of 10, and test them based on their results. This would be interesting in multiple ways, as it would allow me to expand past just prompted image GP.

The most appealing of the potential fitness functions that I did not end up implementing is an algorithm that creates a fitness value based on the diversity of the program, the density of bright or colorful pixels/hot-zones in the image, and an algorithm that looks for certain in an image (giving a worse fitness score the more obvious patterns are found). The only change that would need to be made to my original system for this would be either:

- Implementing a color gradient function and a few mathematical-seeming area functions (such as random-walk or a fractal pattern), or
- Implementing a few more control flow patterns, so the existing image instructions would follow more patterns and change more pixels overall.

This would ideally create unprompted images that are interesting to the eye, and would only require a few more things to be implemented on top of the current system.

With all of this said, while only the few best individuals from the longest run of the full instruction list experiment were able to achieve visually significant results, that does prove that this implementation has the basic tools to easily test other image generation GPs, unprompted or not, without biasing the images with human built draw functions.