COSC 3P71: Evolving Mathematical Formulas to Reconstruct Images

Carson Cormier
COSC 3P71
Brock University
Thorold, Canada
ww23iu@brocku.ca — 7843469

Tye Surface
COSC 3P71
Brock University
Thorold, Canada
hc22sf@brocku.ca — 7757529

Massimo Romano
COSC 3P71
Brock University
Thorold, Canada
hf23lg@brocku.ca — 7854532

***Abstract*—This report outlines our implementation of a Genetic Programming (GP) system to create mathematical formulas that reconstruct target images. The system represents formulas using a fixed parametric structure, and applies evolutionary operators such as tournament selection, crossover, and mutation to evolve parameter values toward better solutions. We tested our program on simple images such as gradients and basic shapes, and analyzed how different parameters affect the evolution's results. We measured performance using Mean Squared Error (MSE) between the generated and target images. Our results show that the GP could successfully evolve formulas that capture basic image patterns, and does much better than random formulas. The way formulas are represented, and the sampling strategy that was used, were key to the algorithm's success.**

I. Introduction

This report analyzes our Genetic Programming system that creates mathematical formulas to reproduce an inputted image. The approach we used is a type of procedural generation that is often seen in games and computer graphics, and shows how evolution can find complex math relationships automatically. The goal was to evolve a formula that uses x and y coordinates to make an image as close as possible to a target image.

II. Background Review

Genetic Programming is an evolutionary algorithm inspired by natural selection in nature [1]. It evolves computer programs or math expressions using tree structures that grow or shrink over time. Genetic Programming was a computing method introduced by John Koza that expands on traditional genetic algorithms by allowing programs to have flexible lengths, rather than fixed-size chromosomes [2].

A. How Genetic Programming Works

Our Genetic Programming follows these steps:

- Load and normalize the target image.
- Set Genetic Programming parameters (population, generations, mutation rate, etc.).
- Make an initial population of random formulas.
- For each generation:
  - Evaluate each formula's fitness.
  - Select parents using tournament selection.
  - Apply crossover and mutation to create new formulas.
  - Replace the old population with the new generation.

B. Related Work

Research in the past has shown that evolutionary algorithms can be used for procedural content generation in computer graphics. For example, computer graphics artist and researcher Karl Sims used artificial evolution to create complex visual patterns [4]. Our work uses similar principles but instead applies them to the problem of image reconstruction through mathematical formulas.

This is also related to symbolic regression, where Genetic Programming is used to discover mathematical relationships from data. The difference is that instead of fitting individual data points, we are trying to match spatial patterns across the entire image.

C. Key Components

Our Genetic Programming uses:

- **Chromosome:** Expression trees for formulas.
- **Terminal Set:** x, y coordinates and random constants.
- **Function Set:** +, -, *, protected /, sin, cos.
- **Selection:** Tournament selection with K=3.
- **Crossover:** Subtree crossover.
- **Mutation:** Point mutation and subtree mutation.
- **Fitness:** Negative Mean Squared Error (MSE).
- **Sampling:** Evaluate on 20% of random pixels each generation.

III. Methodology

A. AI Technique and Justification

We chose a parametric approach with a fixed formula structure instead of using full genetic programming with free-form trees. We decided to do it this way to give us a good balance between flexibility and speed. The fixed structure still allows the formula to represent basic image patterns, but it evaluates much faster than unrestricted Genetic Programming, which helped us keep runtimes reasonable. Although we refer to our system as Genetic Programming, our implementation follows a parametric Genetic Algorithm approach with a fixed formula structure, as recommended for beginners in the project outline.

B. Formula Representation Design

Each image is generated by the formula represented as:
$f(m,n) = a_1 \cdot \sin(a_2 \cdot m + a_3) + a_4 \cdot \cos(a_5 \cdot n + a_6) + a_7 \cdot m + a_8 \cdot n + a_9 \cdot \sin(a_{10} \cdot m \cdot n) + a_{11}$

The parameters a_1 through a_{11} are evolved to adjust their values. This structure works well for smooth patterns such as gradients and circles, as was the main focus of our experiments.

C. Fitness Metric and Evaluation Strategy

To calculate the average squared difference between the target image and the generated image, we used a Mean Squared Error (MSE) as our main fitness metric:

$$\text{MSE} = \frac{1}{WH} \sum_{m=1}^{W} \sum_{n=1}^{H} \left(I_{\text{target}}(m,n) - I_{\text{generated}}(m,n) \right)^2$$

Pixel coordinates m and n were normalized to the range $[0, 1]$ before evaluating the formula at each location. To understand the values more easily, we converted this to a similarity percentage. To improve performance and efficiency, we only evaluate 20% of pixels each generation (random sampling), which reduces computation time while still giving a reliable fitness estimate.

D. Algorithm Design and Parameter Choices

Parameters used:

- Population size: 100
- Maximum generations: 500-800 (depending on target)
- Tournament selection K = 3
- Elitism: 1 individual
- Pixel sampling: 20% per generation
- Crossover rate: 0.8
- Mutation rate: 0.15

We chose these values based on various tests and selected the best results. A population of 100 gives enough diversity without being too slow. Tournament size of 3 provides good selection pressure. The 20% sampling was the key to making it run fast enough. And a crossover rate of 0.8 with a mutation of 0.15 gave steady improvement without too much randomness.

IV. Implementation

Our system was built in Python with these main parts:

- formula.py: Formula representation and evaluation.
- ga.py: Genetic Programming algorithm with tournament selection, crossover, mutation.
- fitness.py: MSE calculation with pixel sampling.
- operators.py: Selection, crossover, and mutation functions.
- images.py: Target image generation.
- main.py: Main evolution loop.
- visualize.py: Plotting and image display.

The code handles numerical issues like division by zero (protected division) and NaN values (replaced with 128).

V. Experiments and Results

A. Experiment 1: Horizontal Gradient

TABLE I: Genetic Programming Performance on Gradient Image

Method	Similarity	Generations	Formula
Random	15.3%	N/A	Random
Hand	100%	N/A	(x+1)/2
Genetic Programming Run	99.98%	300	242.1*x + 12.3*y - 4.7

For our first experiment, we tested the system on the gradient image, as it was the easiest target due to its smooth-changing pixel values. Random formulas performed very poorly, while the hand-written formula achieved a perfect match. The Genetic Programming evolved a formula that was almost perfect in only 300 generations.

B. Experiment 2: Circle - Standard vs Balanced Fitness

TABLE II: Circle Comparison: Standard vs Balanced Fitness

Fitness Type	Similarity	Result
Standard	97.8%	All white
Balanced	99.04%	Circle visible

The second experiment used the white circle with a black background, and it showed a big problem. This problem is that standard fitness gives a high score for just making the whole image white (which matches most pixels numerically), but looks completely wrong. However, switching to a balanced fitness function gives equal weight to black and white areas, so the circle is actually visible.

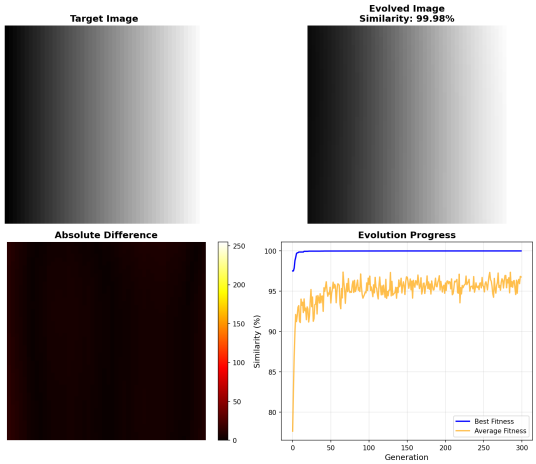
C. Experiment 3: Checkerboard Pattern

TABLE III: Genetic Programming Performance on Checkerboard

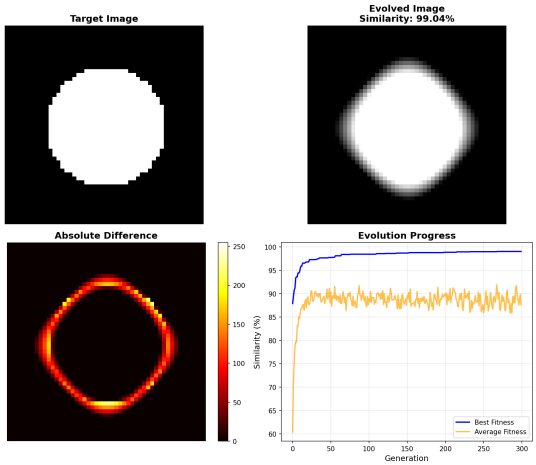
Method	Similarity	Generations	Visual Match
Random	15.7%	N/A	Poor
Hand-crafted	43.2%	N/A	Poor
Genetic Programming Run	75.63%	800	Fair

The third experiment was the checkerboard, which was the most difficult target. The Genetic Programming recognized the repeating pattern, but the edges were blurry. Moreover, the random and hand-crafted formulas both performed poorly, while the Genetic Programming system resulted in a very mediocre match of about 75.6 percent after 800 generations. Conclusively, the final formula was more complex than for the previous experiments, reflecting the higher difficulty of representing sharp, repeating patterns.

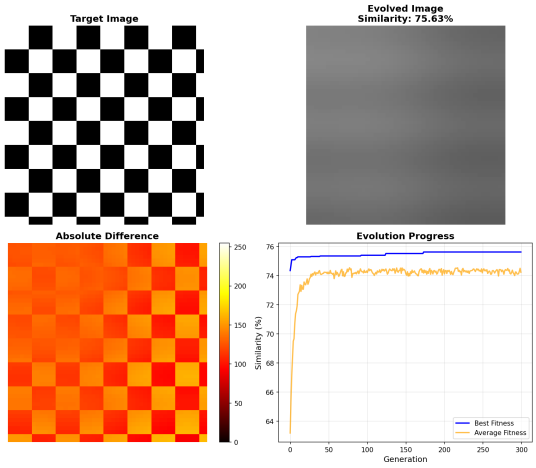
D. Visual Results



(a) Gradient evolution showing near-perfect reconstruction.



(b) Circle evolution showing target, evolved image, similarity, and fitness.



(c) Checkerboard evolution demonstrating pattern recognition.

VI. Advanced Features Implemented

We implemented the below advanced features to improve our system as required by the project:

A. Feature 1: Multiple Fitness Functions

We created and tested multiple fitness evaluation methods:

- Standard MSE.
- Balanced fitness (gives equal weight to dark/light areas).
- Weighted fitness (higher weight to less common pixels).
- Edge-aware fitness (focuses on image edges).

Multiple fitness functions addressed the problem where standard MSE gave high scores for images that looked wrong (seen in mostly white images like the white circle).

B. Feature 2: Computational Efficiency - Pixel Sampling

We used 20% random pixel sampling per generation instead of evaluating the full image. This reduced computation time by about 80% while still getting accurate results that worked well on the full image.

C. Feature 3: Convergence and Fitness Curve Analysis

We tracked and visualized:

- Best fitness over generations.
- Average population fitness.
- Convergence patterns for different target images.

This helped us see how evolution progressed and when the system was starting to plateau.

D. Feature 4: Multiple Target Images with Comparison

We systematically tested on 3 different targets with increasing difficulty:

- Gradient (simple)
- Circle (medium)
- Checkerboard (complex)

We compared results across all three with metrics and visuals, which allowed us to evaluate how well the system performed under different image inputs.

VII. Discussion

A. Strengths

Our Genetic Programming system was able to evolve formulas that recreate images. The results show that it works well for simple patterns such as gradients or simple shapes, but struggles under more complex target images. Concluding that our system's performance depends a lot on the target's complexity.

Moreover, using 20% pixel sampling was crucial to keeping the program running fast and efficiently. Without pixel sampling, the program would have to evaluate every pixel every time, which would be too slow.

Additionally, the balanced fitness function we created solved a major problem where standard MSE often failed for images with mostly one color.

B. Limitations

A major issue is that the standard MSE breaks if the image is mostly one color, sometimes giving high similarity scores even when the image looks completely wrong. For example, Standard fitness gives 97.8% similarity for the circle image, although it produced an all-white image (completely wrong). While balanced fitness gives 99.04% similarity and produces a visible circle (correct). This shows that the numerical metrics don't always match visual quality.

Next, through the checkerboard image, it was apparent that smooth mathematical formulas struggle to represent sharp edges. The results show that the repeating pattern was recognized, but its edges were blurry.

Another limitation is that more complex images need longer evolution times and still don't get as good results. For example, the checkerboard only reached 75.63% after 800 generations.

C. Observations

We observed that the system handle different types of patterns with varying difficulty. The evolution worked best on:

- Linear patterns (gradients) - very easy
- Smooth shapes (circles) - medium difficulty
- Sharp patterns (checkerboard) - hard

The evolved formulas' complexity also grew with the target's difficulty; simple gradients produced compact formulas, but the checkerboard needed a longer, more complex expression.

VIII. Conclusion

In conclusion, Genetic Programming works well for generating formulas for simple images and shows how evolution can be used for creative and automatic design. The balanced fitness function was important to getting good visual results on similar-coloured shapes like the circle image. In the future, our group will extend our research by trying to find new ways to simplify the more complex image formulas.

IX. Team Contributions

Tye Surface - Built the main Genetic Programming algorithm, formula trees, and evolutionary operators.

Massimo Romano - Ran the experiments, collected all the data and numbers, and assembled the GitHub.

Carson Cormier - Wrote the entire IEEE report, including background, methods, results, and analysis.

References

[1] J. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press, 1992.

[2] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach, Fourth Edition," Pearson, 2020.

[3] B. Ombuki-Berman, "COSC 3P71 Artificial Intelligence: Lecture Notes on Genetic Programming," Brock University, 2024.

[4] K. Sims, "Artificial Evolution for Computer Graphics," Computer Graphics, 1991.

X. GitHub Repository

`https://github.com/Massimo-R/image-formula-evolution`