Image Reconstruction using Genetic Algorithms with Triangular Tiling

Siaglov Ilia
Department of Computer Science
Innopolis University

***Abstract*—This paper presents an evolutionary approach to image reconstruction using a distributed Genetic Algorithm (GA). The objective is to approximate a target image using semi-transparent geometric primitives (triangles) within a strict time limit. To address the computational complexity of image processing, a "Divide and Conquer" strategy is employed: the image is partitioned into a grid of independent tiles, processed in parallel using multi-core architecture. The proposed method utilizes Root Mean Squared Error (RMSE) as a fitness function and employs OpenCV for high-speed rendering. Experimental results demonstrate that the algorithm consistently produces aesthetically pleasing "Low Poly" artistic approximations within 2-3 minutes on standard hardware, significantly outperforming sequential approaches.**

***Index Terms*—Genetic Algorithm, Image Approximation, Parallel Computing, Tiling, Generative Art.**

I. Introduction

Standard EAs face significant challenges when applied to high-resolution images (e.g., 512×512 pixels). Calculating the fitness function (pixel-wise difference) for the entire image is computationally expensive. Furthermore, as the number of shapes increases, the genome becomes large, making convergence slow and prone to local optima.

This project addresses these limitations by implementing a **Tile-based Distributed Genetic Algorithm**. Instead of evolving the entire image at once, we split the problem into 64 smaller sub-problems. This approach allows us to: 1) Utilize 100% of the CPU capabilities via parallel processing. 2) Simplify the search space for the genetic algorithm. 3) Achieve a distinct "mosaic" or "stained glass" aesthetic.

II. Methodology

A. Algorithm Flow

The system follows a standard genetic pipeline, enhanced by parallel execution. The process for a single input image is as follows:

1) **Preprocessing:** The target image is resized to 512×512 pixels (if it is not) and divided into an 8×8 grid (64 tiles of 64×64 pixels each) (this value can be changed).
2) **Parallelization:** Each tile is assigned and calculates as a separate CPU process.
3) **Evolution:** A local GA runs independently for each tile to reconstruct that specific segment of the image by iterating through translucent triangles of different sizes and colors.
4) **Reconstruction:** Upon completion, all evolved tiles are builded together to form the final result.

B. Chromosome Representation

We chose triangles as our drawing primitives because they can approximate both sharp edges and smooth gradients (via overlapping). But this figure can be changed, i.e to squares or circles. An individual represents one tile and consists of a fixed number of triangles (genes). In our experiments, we used $N = 80$ triangles per tile. The chromosome is represented as a flat array of floating-point numbers in the range $[0.0, 1.0]$. Each triangle requires 10 genes:

$$Gene_i = [R, G, B, \alpha, x_1, y_1, x_2, y_2, x_3, y_3] \tag{1}$$

Where:

- R, G, B: Color channels.
- α: Transparency (mapped to range $0.1 - 0.6$).
- (x_n, y_n): Coordinates of the three vertices of the triangle relative to the tile size.

C. Fitness Function

The fitness function determines how well rendered tile matches the target tile. We use the Negative Root Mean Squared Error (RMSE). RMSE is preferred over MSE because it scales linearly with pixel intensity errors, making it more interpretable and understandable for people. There was an idea to use SSE, but the values were too large (-20-40 mill).

The fitness F for an individual I is calculated as:

$$F(I) = -\sqrt{\frac{1}{W \cdot H \cdot C} \sum_{p=1}^{W \cdot H \cdot C} (P_{gen} - P_{target})^2} \tag{2}$$

Where W, H are tile dimensions, C is the number of channels (3), and P represents pixel intensity values. A fitness of 0 implies a perfect match.

D. Genetic Operators

To drive evolution, we use standard operators:

- **Selection:** We use *Tournament selection* with size $k = 3$. This method is efficient and prevents premature convergence by giving a chance to non-dominant individuals and at the same time does not let through those who are completely unsuitable.
- **Crossover:** *Uniform crossover* is applied. For each gene in the chromosome, the offspring takes the value from either Parent A or Parent B with a 50% probability.

- **Mutation:** We use *Gaussian mutation*. With a probability of 5% ($rate = 0.05$), a random value drawn from a normal distribution ($\sigma = 0.15$) is added to a gene. This allows for accurate tune of coordinates and colors.
- **Elitism:** The top 6 best individuals are copied to the next generation unchanged to ensure that the best solution found so far is never lost.

III. Setup

The algorithm was implemented in Python. The following external libraries were used:

- **OpenCV (cv2):** Used for rendering triangles. It was chosen over standard PIL/Pillow libraries because its C++ backend provides significantly faster rasterization, which is critical for calculating fitness thousands of times per second.
- **Multiprocessing:** Used to create a pool of workers corresponding to the number of CPU cores.
- **NumPy:** Used for high-performance matrix operations during fitness calculation.
- **MatPlotlib:** Used for plotting the statistical graphs

The parameters used for the experiments are listed in Table I.

TABLE I
GENETIC ALGORITHM PARAMETERS

Parameter	Value
Image Size	512×512
Grid Size	8×8 (64 tiles)
Population Size	60
Generations	300
Shapes per Tile	80
Mutation Rate	0.05
Mutation Scale	0.15
Elite Size	6
Tournament Size	3

IV. Results and Analysis

A. Visual Results

The algorithm successfully reconstructs the input images faster than the allotted time. The resulting images exhibit a stylized "Low Poly" effect while retaining the structural details of the original image.

B. Convergence Analysis

To evaluate stability, the algorithm was run 3 times for each image. Fig. **??** illustrates the convergence of the fitness function over 300 generations for one of the input images.

The X-axis represents the number of generations, and the Y-axis represents the Fitness value (-RMSE). The blue line shows the **Best Fitness** in the population, while the orange line shows the **Average Fitness**. The gap between them indicates that the population maintains genetic diversity throughout the run.

Fig. 1. Reconstruction result for Input Image. The 8x8 tiling structure is visible, adding to the final result artistic effect.

TABLE II
EXPERIMENTAL STATISTICS (IMAGE: INPUT1.JPG FROM TESTS FILES)

Run #	Time (sec)	Final RMSE	Generations
1	154.6	-16.33	300
2	141.9	-16.27	300
3	148.7	-16.34	300
Average	**148.3**	**-16.31**	**300**

C. Statistical Analysis

Table II presents the statistical data collected over 3 independent runs of one image.

The statistics show that the algorithm is highly stable. The standard deviation in execution time is negligible, and the final RMSE values are very close across all runs. The average execution time is approximately 2.5 minutes, which is well within the 10-minute constraint.

V. Conclusion

In this assignment, we implemented a distributed Genetic Algorithm for image reconstruction. By dividing the image into tiles and parallel processing, we achieved high quality approximations using semi-transparent triangles. The use of the RMSE metric provided a stable gradient for evolution. The proposed method demonstrates that dividing a complex evolutionary problem into independent sub problems is an effective strategy for reducing computation time while maintaining high visual quality.

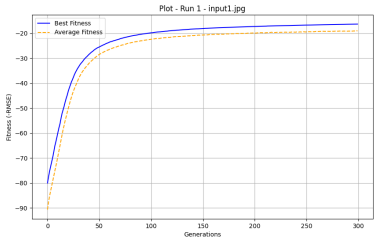

Fig. 2. Input 1: Run 1

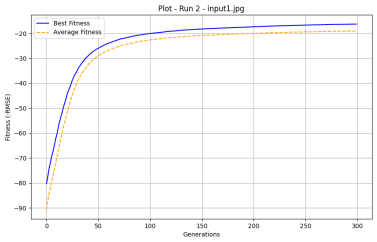

Fig. 3. Input 1: Run 2

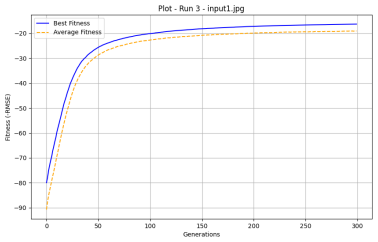

Fig. 4. Input 1: Run 3

Fig. 5. Convergence analysis for Input Image 1 across three independent runs. All runs show consistent improvement.

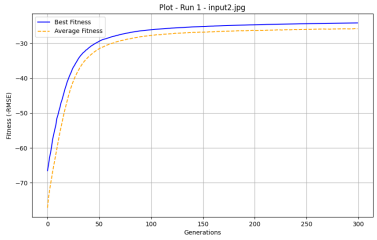

Fig. 6. Input 2: Run 1

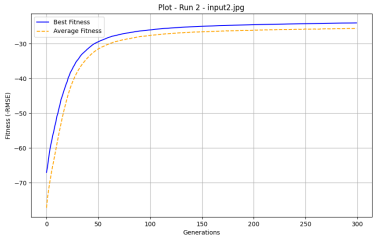

Fig. 7. Input 2: Run 2

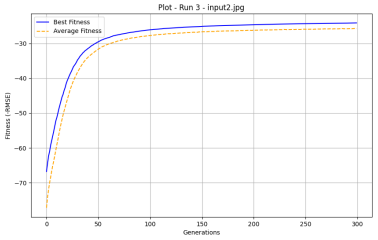

Fig. 8. Input 2: Run 3

Fig. 9. Convergence analysis for Input Image 2.

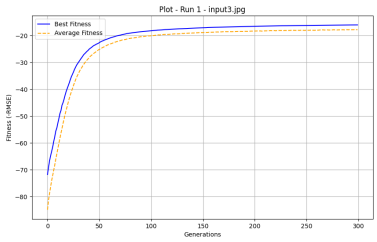

Fig. 10. Input 3: Run 1

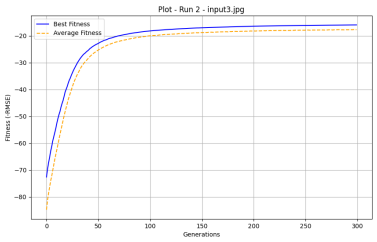

Fig. 11. Input 3: Run 2

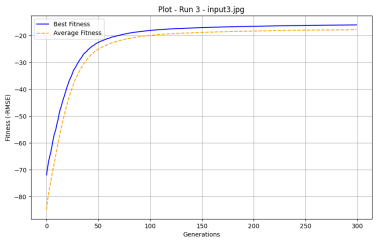

Fig. 12. Input 3: Run 3

Fig. 13. Convergence analysis for Input Image 3.

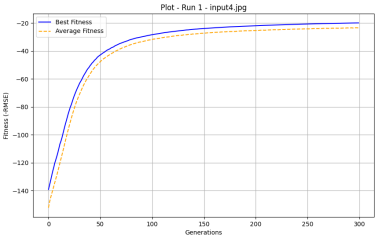

Fig. 14. Input 4: Run 1

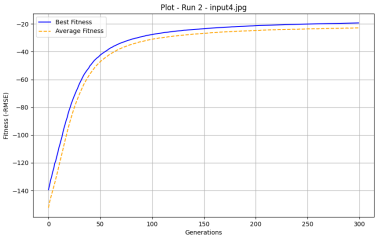

Fig. 15. Input 4: Run 2

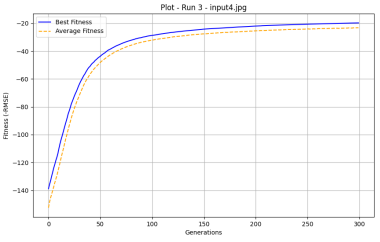

Fig. 16. Input 4: Run 3

Fig. 17. Convergence analysis for Input Image 4.

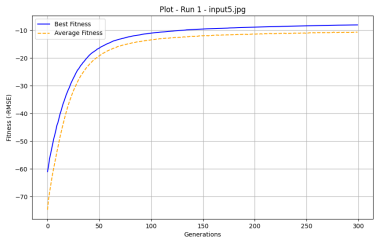

Fig. 18. Input 5: Run 1

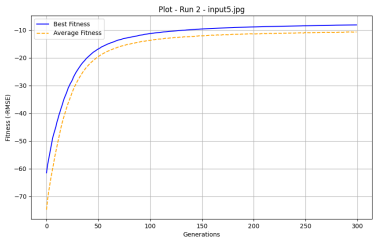

Fig. 19. Input 5: Run 2

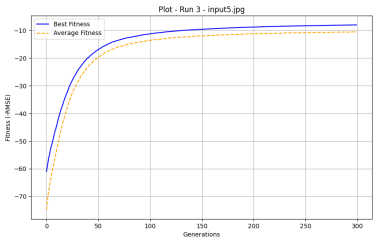

Fig. 20. Input 5: Run 3

Fig. 21. Convergence analysis for Input Image 5.