Evolutionary Reconstruction of Images Using a Letter-Based Genetic Algorithm

Arthur Aipov
CSE-05, Innopolis University
Email: a.aipov@innopolis.university

***Abstract*—This work studies a genetic algorithm (GA) for approximating arbitrary target images using a fixed set of letter-shaped primitives (A, R, T, H, U). The genome encodes the position, scale and color of individual letters that are rendered on a canvas. The fitness function is defined as the mean squared error (MSE) between the rendered image and the target, and the target is used strictly through the fitness function (i.e., there is no direct access to target pixels inside mutation). We evaluate the proposed GA on six test images under two conditions: with and without pre-processing of the target (blur, posterization and edge enhancement). For each image we run the GA three times and report the mean and standard deviation of the final fitness, as well as the convergence curves and visual comparisons between the target and the reconstructed results.**

***Index Terms*—Genetic algorithm, evolutionary art, image approximation, computer graphics.**

I. Introduction

Evolutionary algorithms are widely used in optimization problems where gradients are unavailable or the search space is highly non-linear and discrete. A particularly interesting application area is *evolutionary art*, where an algorithm searches for visually pleasing or target-matching images by evolving sets of primitives.

In this project we consider the task of reconstructing a given target image using a limited set of letter-shaped primitives. Each primitive is a small bitmap mask corresponding to one of five letters: `A`, `R`, `T`, `H`, `U`. The goal is to synthesize an image that is visually close to the target while respecting the constraint that the whole image is composed of these letters only.

Unlike pixel-based or triangle-based evolutionary art, the use of discrete glyphs imposes a strong structural prior, which makes the search space highly constrained and non-convex. At the same time, this leads to aesthetically interesting reconstructions that preserve large-scale structure while revealing the underlying letter primitives.

II. Problem Statement

Given a target image $I^\star$ of size $W \times H$, we aim to find a configuration of letters

$$\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_N\}$$

such that the rendered image $R(\mathcal{L})$ is close to $I^\star$ in terms of mean squared error (MSE) over RGB channels.

Each letter ℓ_i is defined by:

- a glyph type $c_i \in \{\mathrm{A}, \mathrm{R}, \mathrm{T}, \mathrm{H}, \mathrm{U}\}$,
- a position (x_i, y_i) on the canvas,
- an integer scale s_i,
- a color (r_i, g_i, b_i) and an alpha value a_i.

Rendering is performed by rasterizing the glyph mask (a fixed 9×9 bitmap) at scale s_i, and alpha-blending its color onto the canvas in the drawing order.

The optimization objective for a whole population of letters (in the first version of the code) or for individual letters (in the final version) is

$$\text{Fitness} = -\text{MSE}(R(\mathcal{L}), I^\star),$$

so that *higher* fitness corresponds to a *smaller* reconstruction error.

III. Methodology

A. Letter Masks and Rendering

Each letter is represented as a binary mask of size 9×9:

$$M_c[y][x] \in \{\texttt{true}, \texttt{false}\}, \quad c \in \{\mathrm{A}, \mathrm{R}, \mathrm{T}, \mathrm{H}, \mathrm{U}\},$$

hard-coded in the Go program.

To render a letter $\ell = (c, x, y, s, r, g, b, a)$ on an RGBA canvas, the algorithm:

1) Iterates over all mask pixels (m_x, m_y) where $M_c[m_y][m_x] = \texttt{true}$.
2) For each mask pixel, covers an $s \times s$ block of canvas pixels starting at
$$(p_x, p_y) = (x + m_x \cdot s,\ y + m_y \cdot s).$$
3) For each covered pixel, applies alpha blending between the existing color and the letter color (r, g, b, a) using a simple "over" operator.

In the final version of the code, when rendering the whole population, letters are drawn in order of increasing fitness, with alpha dynamically adjusted so that better letters are rendered more opaquely. This emphasizes more useful primitives in the final composite image.

B. Genome Representation

In the final implementation each *individual* in the GA encodes a single letter:

- glyph type $rune$,
- position (x, y),
- scale s,
- color (r, g, b, a),

- local fitness value f.

A full rendered image for a given generation is obtained by rendering the whole population of letters.

This design has two important properties:

1) The population size directly controls the effective "number of letters" in the image.
2) Local mutations are easier to evaluate: each letter has its own local fitness measuring how well it matches the target in the region it covers.

C. Fitness Function

For a single letter ℓ, we compute a local fitness as

$$f(\ell) = -\frac{1}{N_\ell} \sum_{(x,y)\in\Omega(\ell)} \left[(r_\ell - r^\star_{x,y})^2 + (g_\ell - g^\star_{x,y})^2 + (b_\ell - b^\star_{x,y})^2\right],$$

where:

- $\Omega(\ell)$ is the set of canvas pixels actually covered by the letter,
- $N_\ell = |\Omega(\ell)|$ is the number of covered pixels,
- $(r^\star_{x,y}, g^\star_{x,y}, b^\star_{x,y})$ is the RGB value of the target image.

Thus, each letter is rewarded for having a color close to the target pixels under its mask. The *global* fitness of the population is defined as the sum of all local fitness values:

$$F(\mathcal{P}) = \sum_{\ell\in\mathcal{P}} f(\ell),$$

where $\mathcal{P}$ is the population of letters. This value is used only for logging and monitoring convergence.

D. Target Usage Policy

The project constraint is that the target image cannot be used directly in mutation or crossover. In the final code:

- The target is used during fitness evaluation to compute the local MSE.
- The mutation operator *does not* read pixels from the target; it only performs random drift in position, scale and color.

Thus the only way information from the target influences the population is through selection based on fitness.

E. Mutation and (Implicit) Selection

The mutation operator takes a copy of an individual letter and perturbs its parameters:

- With some probability (depending on the generation number), the position (x, y) is shifted by a small random offset.
- Scale s is adjusted by a small integer offset and clamped to a valid range $[2, 8]$.
- Color channels (r, g, b) undergo additive integer noise and occasional multiplicative scaling to simulate local color drift.
- Alpha a is perturbed and clamped to $[80, 220]$.
- With small probability (1%), the glyph type c is randomly changed to another letter.

There is no explicit crossover in the final version; instead, the algorithm uses a *hill-climbing*–like scheme inside the GA framework:

1) For each individual in the population, create a mutated candidate.
2) Recompute the local fitness for the candidate.
3) If the candidate's fitness is higher, accept it; otherwise, keep the original.

This acts as a local evolutionary step per letter. Global selection emerges because letters that consistently improve fitness stay in the population and dominate the rendered image, while unhelpful letters remain nearly transparent.

The mutation rate decays linearly from a relatively high value at the beginning of evolution to a lower value at the end, which helps explore the search space early and then refine the configuration.

F. Target Pre-processing

To study the effect of simplifying the optimization landscape, each target image is evaluated under two conditions:

1) **No filter:** the raw JPEG image is used as the fitness target.
2) **With filter:** the image is blurred, posterized to a small number of color levels, and then enhanced with a Sobel-based edge detector.

These operations reduce high-frequency noise and compress the color space, which may make the landscape smoother but does not guarantee a better final approximation in terms of raw pixel MSE.

For each pre-processing setting we run the GA three times with different random seeds.

IV. Experimental Setup

A. Dataset and Implementation Details

We use six test images, denoted as Image 1–6 in the results. All images are loaded as JPEGs and converted to RGBA internally. The algorithm is implemented in Go, using standard libraries for image I/O and the `gonum/plot` package for plotting convergence curves.

For each target image and each pre-processing mode (no filter / with filter), the following procedure is applied:

1) Initialize a population of 9000 letters with random positions, scales and colors.
2) For each generation, mutate each letter once and accept only improvements in local fitness.
3) Log the global fitness and average fitness per generation.
4) Render and save intermediate snapshots every 50 generations.
5) After 500 generations or when the time limit of two minute is reached, save the final composite image.

B. Hyperparameters

The main hyperparameters for the final letter-based GA are:

- Population size: 9000 letters.

TABLE I: Summary of final fitness over 3 runs for each image (values scaled by 10^9).

Image	Preproc.	Gen	μ_{best}	σ_{best}	μ_{avg}	σ_{avg}
1	no filter	299	-1.568	0.207	-1.602	0.199
1	with filter	299	-2.147	0.036	-2.183	0.035
2	no filter	299	-1.225	0.021	-1.247	0.022
2	with filter	299	-1.473	0.026	-1.495	0.025
3	no filter	299	-0.788	0.009	-0.809	0.007
3	with filter	299	-0.962	0.007	-0.984	0.006
4	no filter	299	-3.454	0.044	-3.554	0.044
4	with filter	299	-5.530	0.029	-5.662	0.040
5	no filter	299	-0.392	0.007	-0.404	0.007
5	with filter	299	-0.503	0.005	-0.516	0.006
6	no filter	299	-0.703	0.020	-0.723	0.020
6	with filter	299	-1.195	0.009	-1.217	0.009

- Number of generations: 500 (or until a 2 minute time limit).
- Mutation rate: linearly decays from 0.6 to 0.15.
- Scale range: 2 to 8 pixels per mask cell.
- Alpha range: $[80, 220]$.
- Number of runs per image and mode: 3.

V. Results and Analysis

A. Numerical Results

Table I summarizes the final fitness values over three runs for each image and each pre-processing mode. For better readability, the values are scaled by 10^9; because the fitness is negative MSE, values closer to zero indicate better reconstructions.

Across all images, the GA consistently converges (standard deviations are small), but the effect of pre-processing is not trivial. For all six images, the filtered targets lead to *more negative* fitness values, i.e., higher MSE in raw pixel space, even though the filtered targets are visually simpler.

B. Convergence Behaviour

For each image we plot the evolution of global and average fitness values over generations for both modes. To keep the layout compact, we arrange the curves for all three runs in a single wide figure per image (top row: no filter, bottom row: with filter).

C. Qualitative Visual Analysis

For each test image we also compare:

1) The original input (raw JPEG from the `inputs` folder).
2) The processed target (blurred, posterized and with edges).
3) The final reconstruction without pre-processing.
4) The final reconstruction with pre-processing.

All four images are shown side by side in Figs. 7–12, which allows us to visually assess how well the GA preserves global structure and color under each setting.

VI. Discussion

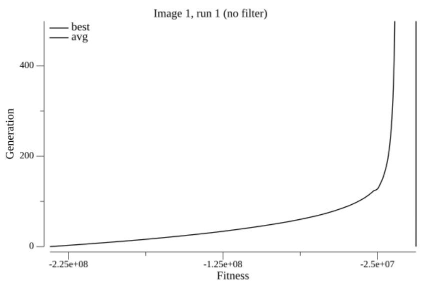
The experimental results highlight several important observations:

- The letter-based representation is expressive enough to capture large-scale structure and approximate the overall color distribution of the target images, while producing a stylized "mosaic of letters" appearance.
- Convergence curves (Figs. 1–6) show stable improvements over generations with diminishing returns near the end of evolution.
- Pre-processing the target with blur, posterization and edge enhancement does not necessarily improve the final pixel-level MSE; in fact, the numerical fitness becomes worse for all images (Table I). However, the filtered targets are visually simpler, which might be beneficial if the optimization goal were perceptual similarity instead of pure MSE.
- Because each letter is evolved independently with local fitness, the algorithm is relatively robust to poor individuals: they tend to remain nearly transparent and thus have little effect on the final composite image.

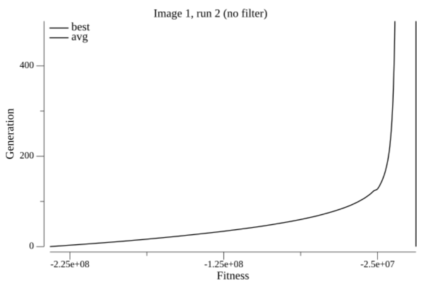
VII. Conclusion

We implemented and evaluated a letter-based genetic algorithm for image reconstruction in Go. The algorithm uses a population of letter glyphs, each defined by position, scale and color, and optimizes their configuration using local mutation and selection based on mean squared error with respect to the target image. The constraint that letters cannot read target pixels during mutation is respected: the target is only used in the fitness function.

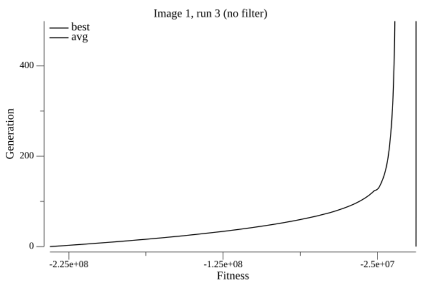
Experiments on six test images demonstrate that the method converges reliably and produces recognizable, stylized reconstructions. Future work includes exploring perceptual loss functions, adaptive population sizes, and more sophisticated crossover operators between letters, as well as extending the alphabet of glyphs beyond the fixed set used in this project.



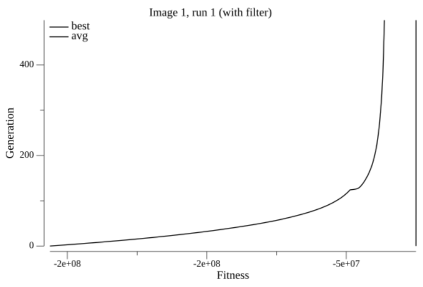
(a) Run 1, no filter



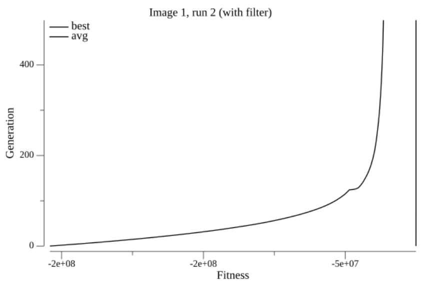
(b) Run 2, no filter



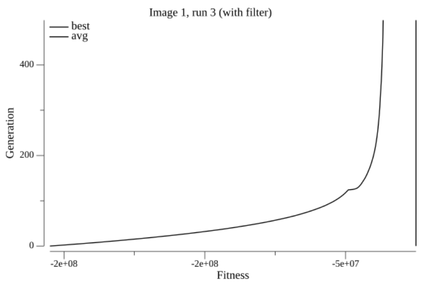
(c) Run 3, no filter



(d) Run 1, with filter

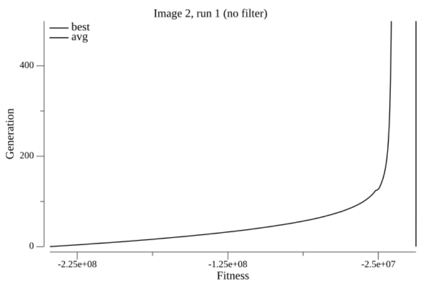


(e) Run 2, with filter

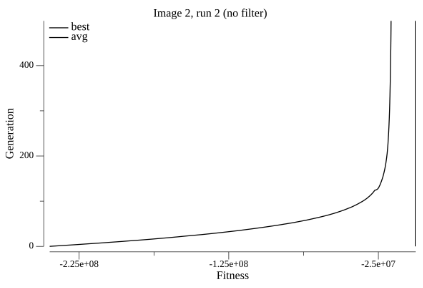


(f) Run 3, with filter

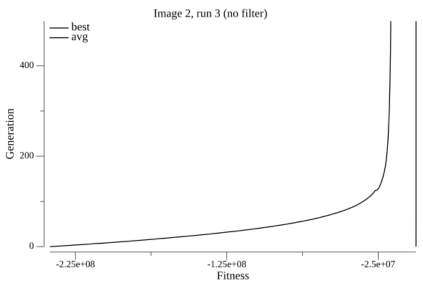
Fig. 1: Fitness curves for Image 1 across three runs.



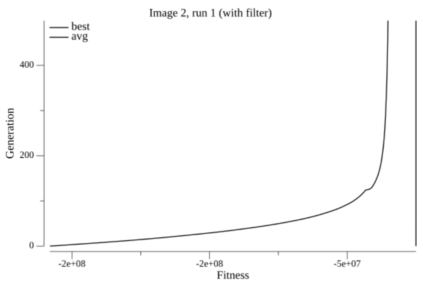
(a) Run 1, no filter



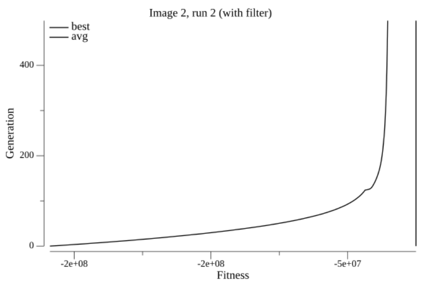
(b) Run 2, no filter



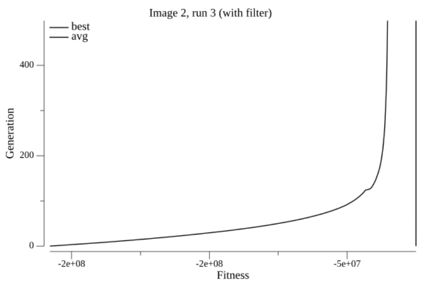
(c) Run 3, no filter



(d) Run 1, with filter

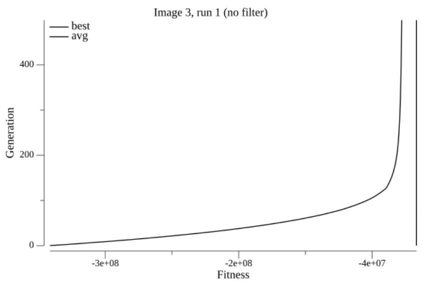


(e) Run 2, with filter

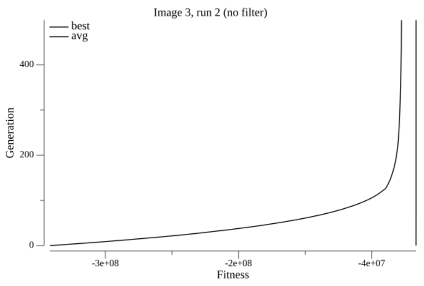


(f) Run 3, with filter

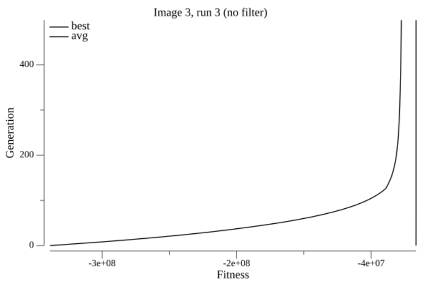
Fig. 2: Fitness curves for Image 2 across three runs.



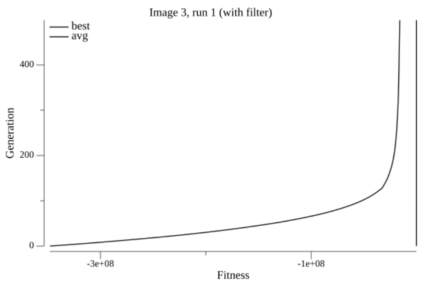
(a) Run 1, no filter



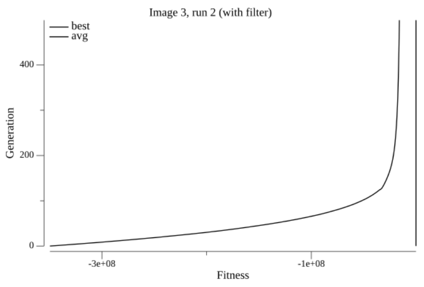
(b) Run 2, no filter



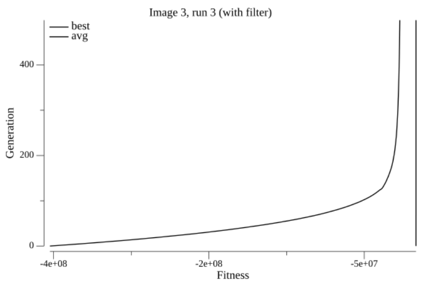
(c) Run 3, no filter



(d) Run 1, with filter

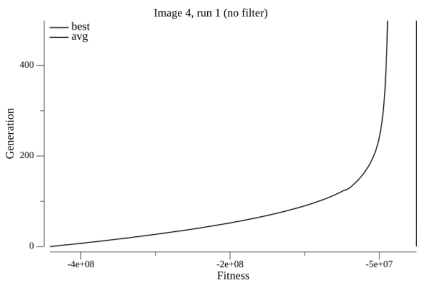


(e) Run 2, with filter

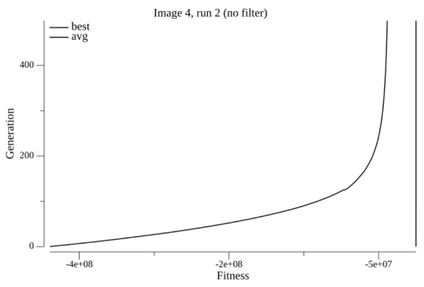


(f) Run 3, with filter

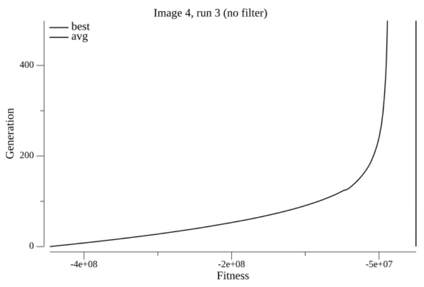
Fig. 3: Fitness curves for Image 3 across three runs.



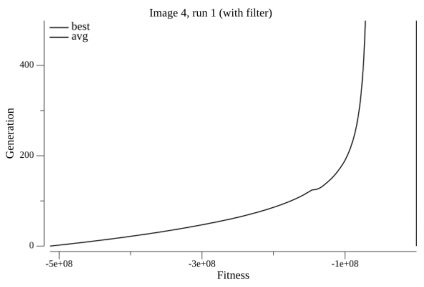
(a) Run 1, no filter



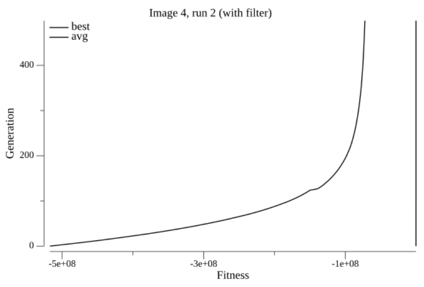
(b) Run 2, no filter



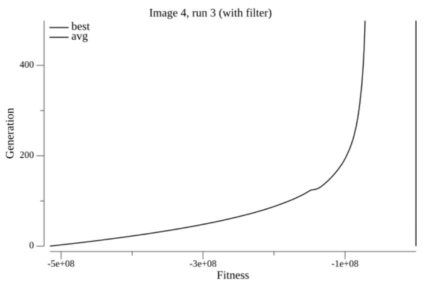
(c) Run 3, no filter



(d) Run 1, with filter

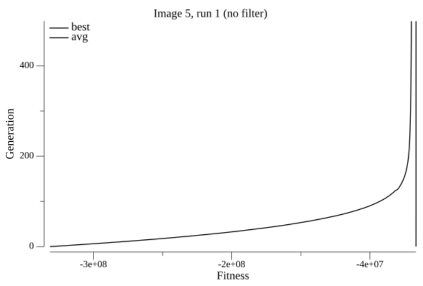


(e) Run 2, with filter

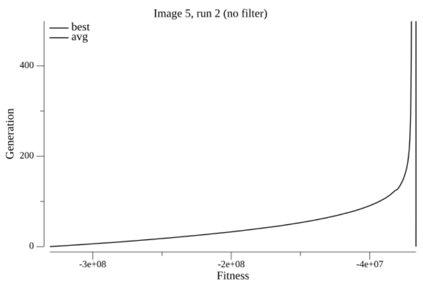


(f) Run 3, with filter

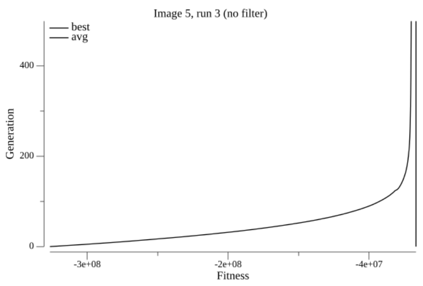
Fig. 4: Fitness curves for Image 4 across three runs.



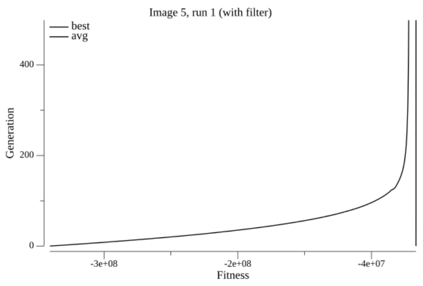
(a) Run 1, no filter



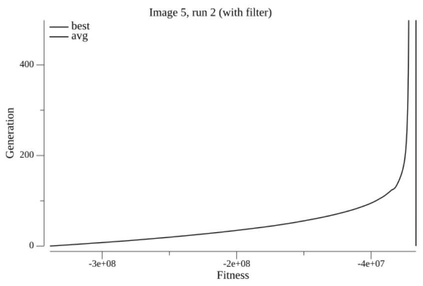
(b) Run 2, no filter



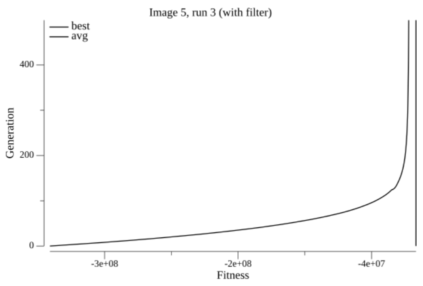
(c) Run 3, no filter



(d) Run 1, with filter

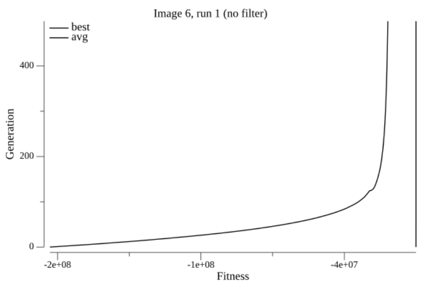


(e) Run 2, with filter

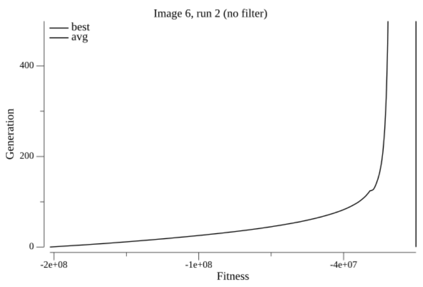


(f) Run 3, with filter

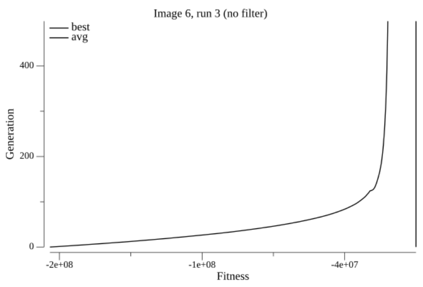
Fig. 5: Fitness curves for Image 5 across three runs.



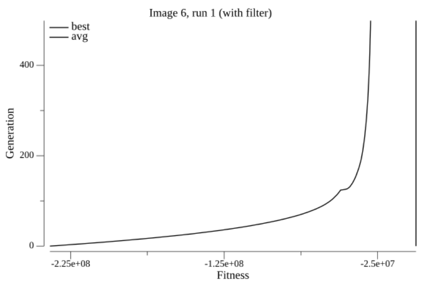
(a) Run 1, no filter



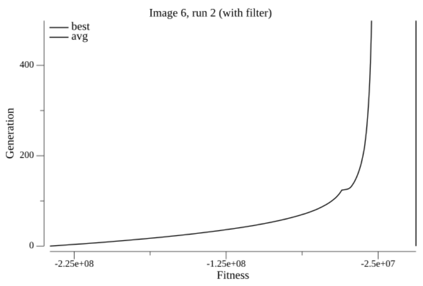
(b) Run 2, no filter



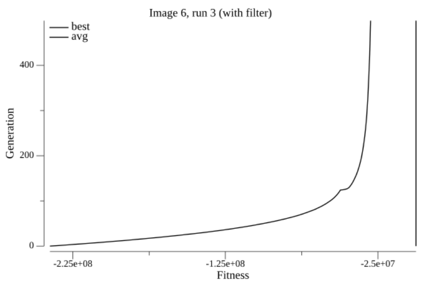
(c) Run 3, no filter



(d) Run 1, with filter

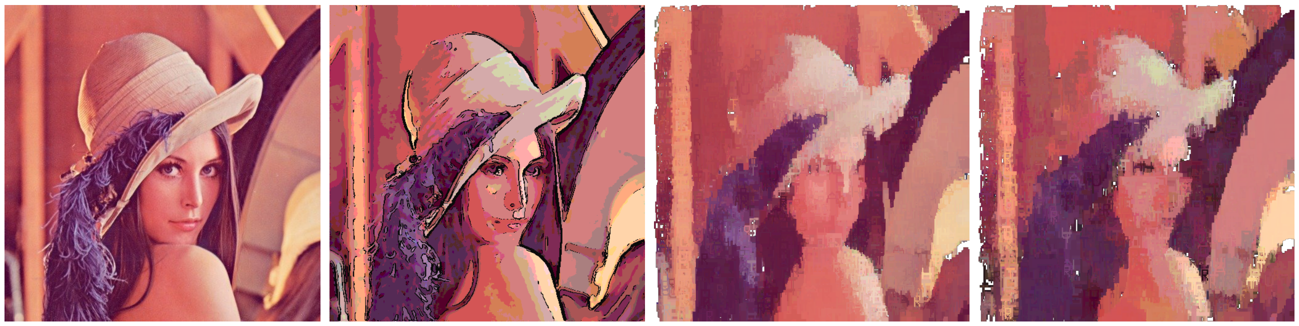


(e) Run 2, with filter



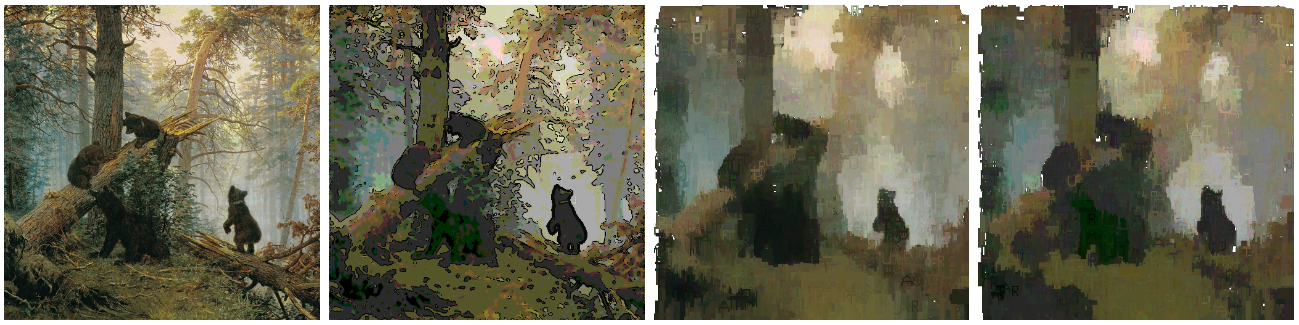
(f) Run 3, with filter

Fig. 6: Fitness curves for Image 6 across three runs.



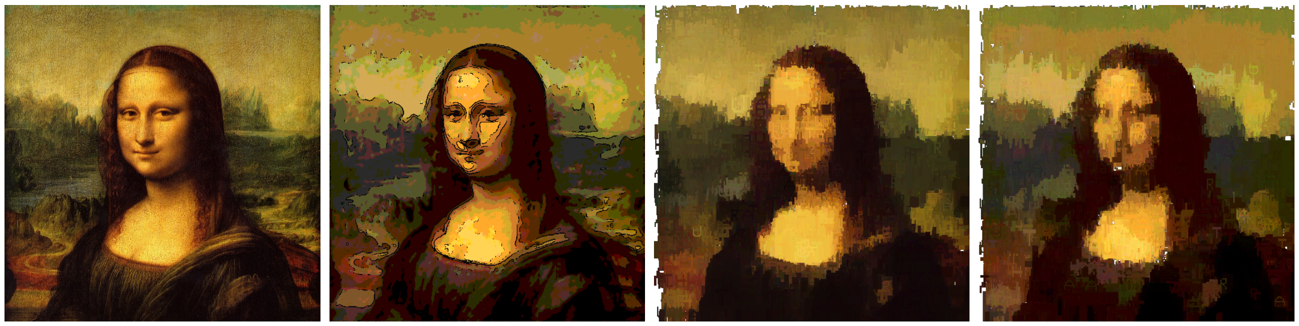
(a) Original input (b) Target (filtered) (c) Result (no filter) (d) Result (with filter)

Fig. 7: Visual comparison for Image 1: original input, pre-processed target and reconstructed images.



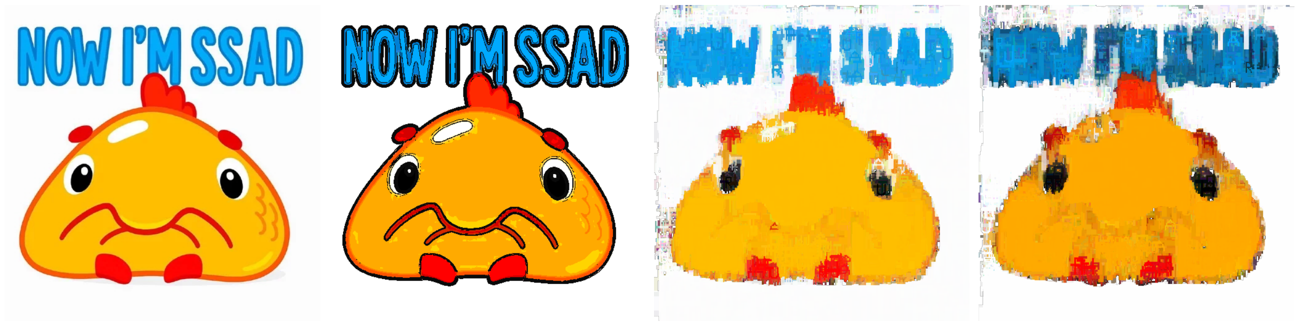
(a) Original input (b) Target (filtered) (c) Result (no filter) (d) Result (with filter)

Fig. 8: Visual comparison for Image 2.



(a) Original input (b) Target (filtered) (c) Result (no filter) (d) Result (with filter)

Fig. 9: Visual comparison for Image 3.



(a) Original input (b) Target (filtered) (c) Result (no filter) (d) Result (with filter)

Fig. 10: Visual comparison for Image 4.

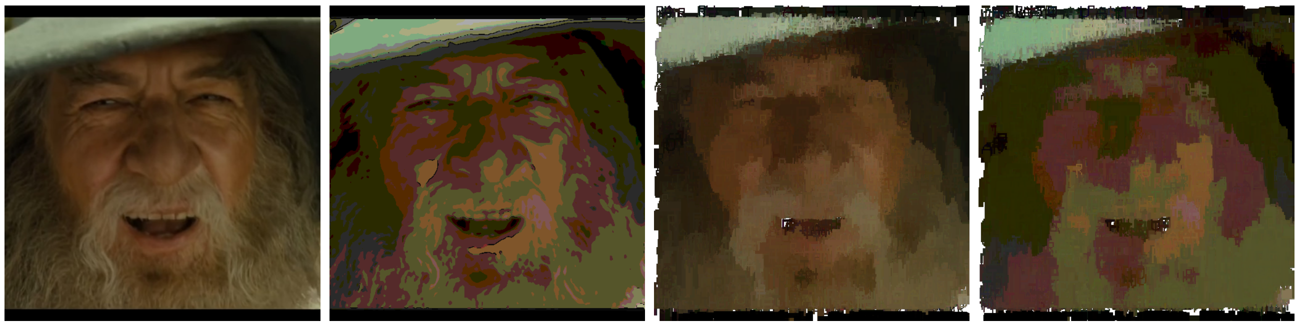

(a) Original input (b) Target (filtered) (c) Result (no filter) (d) Result (with filter)

Fig. 11: Visual comparison for Image 5.

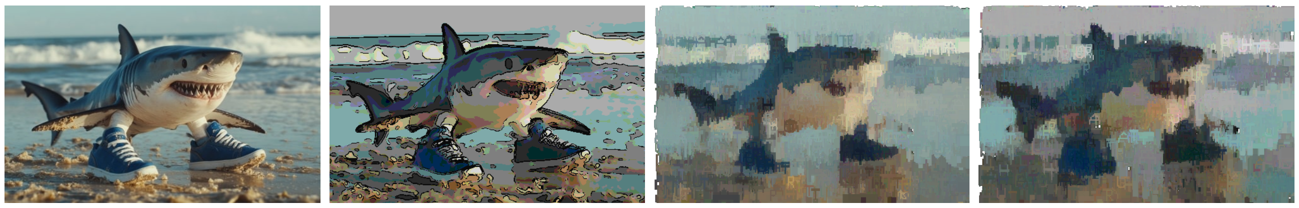

(a) Original input (b) Target (filtered) (c) Result (no filter, run 3) (d) Result (with filter, run 1)

Fig. 12: Visual comparison for Image 6.