Group Project CIFO 2023/2024

Image Recreation Problem

Masters in Data Science and Advanced Analytics

Afonso Gorjão (20230575), Frederico Portela (20181072), Diogo Almeida (20230737), Pedro Carvalho (20230554), João Pedro Mota(20230454)

https://github.com/skynible/CIFO-Project

June 3, 2024

Abstract

This project aims to reconstruct a target image using a Genetic Algorithm. We use a polygon-based approach [2] where each individual comprises a randomly fixed background color and multiple polygons, with random color and vertices. We define numerous genetic operators that can be used for recreating any image followed by experiments that help choose the best model instance for a particular target image. The experimental design follows a top-down approach using statistical tools to compare different model instances.

Statement of Contribution

This project had an equal distribution of tasks, ensuring that everyone's contribution was recognized, and each member's effort was similar. The Genetic Operators (GO) of Fitness Sharing and Mutation were of the responsibility of Pedro Carvalho; the remaining GOs, Selection and Crossover, were of João Pedro Mota. The population aspect was developed by Diogo Almeida and Model Comparison and Experimental Design was lead by Afonso Gorjão. Finally, Results were analysed and documented by Frederico Portela.

Contents

I. Individual and fitness function definition

Every individual is an h x w image, where h is the target height and w is the target width. We decided to represent an individual as a h x w matrix I where I_{ij} is the RGB array of the pixel in row i and column j. So I_{ij0} is the amount of red in pixel I_{ij} etc. This representation allows for atomic control over every aspect of the image. Individuals are initialized with a randomly fixed background color and a random number of polygons chosen within a specified range (poly_range), each with a random color, vertex position, and number of vertices (vertices_range). By using polygons, we can efficiently approximate various shapes found in the target image. An example can be found in Figure 1. A class called *Individual* was created, with the representation and initialization parameters just mentioned.

Since we want to recreate the target image as closely as possible, the use of some type of distance metric seems appropriate and the problem is defined as a minimization. The first impulse might be to use the euclidean distance to measure the distance between two pixels: $d_{ij} = \sqrt{(I_{ij0} - T_{ij0})^2 + (I_{ij1} - T_{ij1})^2 + (I_{ij2} - T_{ij2})^2}$, where T is the matrix representation of the target image. However, this distance can be misleading to our human eyes since we do not perceive color difference as a simple Euclidean distance. It depends on the sensibility our eyes have to red, green, blue, and different color combinations. So we decided to use a distance known as delta-e [3]. This distance tries to take into account the complexity of our eye's sensibility to color in the following way:

$$d_{ij} = \begin{cases} \sqrt{2(I_{ij0} - T_{ij0})^2 + 4(I_{ij1} - T_{ij1})^2 + 3(I_{ij2} - T_{ij2})^2} & \bar{R} < 128 \\ \sqrt{3(I_{ij0} - T_{ij0})^2 + 4(I_{ij1} - T_{ij1})^2 + 2(I_{ij2} - T_{ij2})^2} & \text{otherwise} \end{cases} . \quad (1)$$

where $\bar{R} = \frac{1}{2}(I_{ij0} + T_{ij0})$. The fitness of the individual is then the sum of these distances for all the pixels in the individual: $F = \sum_{ij} d_{ij}$.

II. Genetic Operators

2.1 Selection

Two different Selection methods were implemented: Tournament Selection (tournament_selection) and Fitness Proportionate Selection (fps). Tournament Selection consists of randomly choosing a fixed number (tour_size) of individuals from the population and selecting the best individual in the subset. The draw is made with replacement so the subset can have the same individual more than once. Fitness Proportionate Selection consists of selecting individuals with probability inversely proportionate to the individual's fitness (if it was a maximization problem the probability would be directly proportionate to the fitness). There was an attempt to implement Rank Selection but since it needs to sort individuals based on their fitness it took too long to run and so we decided to drop this method.

2.2 Crossover

Three different Crossover operators were implemented: blend_crossover, cut_crossover, and pixel_crossover. blend_crossover consists in blending the two parents with a randomly chosen opacity $x \in [0, 1]$. So this operator only produces one child: child= x×parent1 + (1-x)×parent2. The cut_crossover consists of randomly selecting either a row or a column of the parent's representation and producing two children by cutting the images in the selected place and switching the slices. This operator also has an argument *mirror_prob* that determines the probability of mirroring each slice. This was implemented to fix a problem with the lack of color in the border of the images (explained later because it needs the full scope of the genetic operators). pixel_crossover consists

of randomly selecting pixels from the parents to fill the child. So for every pixel in the child, there exists a 50% chance for that pixel to be either from parent1 or parent2.

2.3 Mutation

Two different Mutation operators were implemented: polygon_mutation and pixel_mutation_random. polygon_mutation consists of adding to the individual a random number of polygons with a random number of vertices. It has four main arguments (not counting the individual to be mutated which of course has to be an argument). The first is *n_vertices* which determines the number of vertices of the polygons to be added. The second is *n_poly* which is an array with the lower and upper bound of the possible number of polygons (chosen randomly). The third is *mutation_control* which is a bool determining if we want to restrict the size of the polygon or not. The fourth is *mutation_size* which controls the maximum size of the polygons to be added. The size control was done by randomly selecting the coordinates of the center of a square with a side equal to 2×mutation_size. Then we randomly select pixel coordinates inside that square thus controlling the size of the polygon. We also take into account the fact that the square can go out of bounds and slice the square accordingly. pixel_mutation_random consists of randomly selecting a fixed number of pixels and changing their values. This operator has two main arguments, *n_pixels* which determines the number of pixels to be selected, and *same_color* which is a bool deciding if all the pixels are changed to the same randomly selected color or different randomly selected colors.

2.4 Fitness Sharing

This genetic operator was implemented to address genetic diversity in the population [1]. It is one of the standard ways GAs try to tackle this problem. First, we define the sharing function f_{ij} between two individuals in the population:

$$f_{ij} = \begin{cases} 1 - \frac{D_{ij}}{\sigma} & \text{if} \quad D_{ij} < \sigma \\ 0 & \text{otherwise} \end{cases} . \tag{2}$$

where D_{ij} is the distance between the two individuals (same distance used to get the fitness of an individual) and σ can be seen as a radius around the individuals. If individuals are inside each other radii then the fitness function will take non null values. The sharing coefficient of an individual is given by $c_i = \sum_j f_{ij}$. This sharing coefficient is then used to punish individuals who are very close together. Since we are dealing with a minimization problem the new fitness of an individual after fitness sharing is $F_i' = F_i \times c_i$.

III. Population

3.1 Initialization

After the definition of the individuals, fitness, and genetic operators, the python class *Population* was created. This class representation consists of an array of Individuals. It has initialization arguments *size*, which determines population size, and kwargs which is a dictionary with the initialization parameters of the individuals (mentioned in the first section). To evolve the population the method *evolve* was developed. In the typical GA fashion, it iterates the population through cycles of selection, crossover, and mutation over the number of generations desired or until the stopping criteria are met (will be addressed in the arguments of the method). We will only address the main arguments of the evolve method since some of them are not necessary to the discussion.

3.2 Evolve Method

The evolve method has the following arguments: *gens* which determines the number of generations the population will undergo; *selec_alg* chooses the selection operator to be used; *tour_size* determines the size of the tournament in case Tournament Selection is used; *mut_prob* determines the mutation probability; *mutation_alg_prob* determines the probability of using polygon_mutation (so the probability of using pixel_mutation_random is 1-mutation_alg_prob); *pixel_mutation_same_color* is a bool determining if pixel_mutation_random uses the same color or not; *mut_vertices_range* is an array with the lower and upper bounds of the possible number of vertices a polygon can have, when introduced in the image via polygon_mutation; *mut_poly_range* is an array with the lower and upper bounds of the possible number of polygons to be added to the image via polygon_mutation; *mutation_size* controls the maximum size of the polygons to be added via polygon_mutation; *mut_pixel_range* is an array which has the lower and upper bounds of the possible number of pixels to be changed using pixel_mutation_random; *xo_prob* is the probability of two parents undergoing crossover. If crossover does not happen then the parents are replicated to the next generation; *xo_alg_prob* is an array with the probability distribution of applying the different crossover operators. For example [0.5, 0.25, 0.25] says we apply blend_crossover 50% of the times, cut_crossover 25% of the times and pixel_crossover 25% of the times; *mirror_prob* is the probability of mirroring when using cut_crossover. The original idea was to solve the problem of lack of color in the edges of the image since the probability of 2 vertices falling on the same edge is low (either by polygon_mutation or in the individual initialization); *elitism* is a bool determining if elitism is used or not; *fitness_sharing* is a bool determining if fitness sharing is used; *fs_sigma* defines the σ used in fitness sharing; *early_stopping* which determines the number of generations the algorithm will run with the same best individual in the population before stopping. So if early_stopping = 100, the algorithm will stop before it has evolved for *gens* generations, if the same individual in the population is maintained during 100 generations. The evolve method tracks the fitness of the best individual in each generation and the phenotypic variance of the population, given by: $v = \frac{1}{n-1}\sum_i^n (F_i - \bar{F})$, where n =pop_size and $\bar{F}$ is the average fitness in the population.

IV. Model Comparison methods

4.1 Statistical tests

Since GA's are stochastic, comparing two model instances (a model instance is a population that evolves with fixed parameters) is not enough to reach a conclusion regarding their relative and overall performance, it is necessary to run each model instance more than once. So we will run each model instance for T trials. Then each model instance has T data points per generation and the set of data points, of a particular model instance in a particular generation, will be called a *sample*. Sample S_{ij}^k is the jth data point in the ith generation of model instance k. Note that these data points can be two things: fitness of the best individual and variance. To compare two models (model1, model2) we will only look at the last generation samples S_{-1j}^1 and S_{-1j}^2 (where we define the index -1 to be accessing the last generation) and compare the sample means: $\mu_1 = \frac{1}{T}\sum_j^T S_{-1j}^1$, $\mu_2 = ...$ using statistical tests. Since we are dealing with a minimization problem we want to know if we have statistical evidence to say that the sample mean of a particular model is smaller than that of another model.

The first thing is to see if we have statistical evidence to say that the samples are normally distributed. For this, we do a Kolmogorov–Smirnov test with null H0: S_{-1j}^k *is normally distributed*, alternative H1: S_{-1j}^k *is not normally distributed*. In the scenario where we reject the null for some k, we have to perform a non-parametric test to compare the sample means. We chose the Wilcoxon Rank-Sum Test, which compares the samples distributions and not the means, with H0: S_{-1j}^1 *and* S_{-1j}^2 *are drawn from the same distribution*, H1: *The distribution underlying* S_{-1j}^1 *is stochastically smaller than the distribution underlying* S_{-1j}^2. It does not make a statement directly about μ_1 and μ_2. In the case where we cannot reject the null in the Kolmogorov–Smirnov

test, we are going to assume that the samples are normally distributed since we have no statistical evidence contradicting this hypothesis. So we can perform the usual parametric two-sample mean t-student tests. The first thing is to make a Levene test [1] with H0: *The distributions underlying* S^1_{-1j} *and* S^2_{-1j} *have the same variance*, H1: *The distributions underlying* S^1_{-1j} *and* S^2_{-1j} *do not have the same variance.* In the case where we reject the null we perform a Welch's t-test, otherwise, we assume that both samples have the same variance and perform a two-sample mean t-student. Both with H0: *The distribution underlying* S^1_{-1j} *has the same mean as the distribution underlying* S^2_{-1j}, H1: *The distribution underlying* S^1_{-1j} *has a smaller mean then the distribution underlying* S^2_{-1j}.

4.2 Grid method

The class Population has another method called *grid*. It makes a grid search of a particular parameter in the evolve method. It has the following arguments: *tuned_parameter* is a string with the parameter name we want to tune (ex: 'mut_prob'); *param_range* is a list with the values to be tested (ex: [0.05, 0.06, ...]); *num_trials* is the number of trials for each parameter in param_range; *alpha* is the significance level of the statistical tests; *save_dir* is the directory where the files will be saved; *evolve_kwargs* is a dictionary with the base model (it contains the rest of the evolve arguments). This method runs the base model with each parameter in param_range, num_trial times. Then it identifies the parameter k that has the best last-generation sample average μ_k and compares it with all the other last-generation samples using the statistical tests described. It saves a .txt file with all the kwargs used (Individual and evolve) and the results of the statistical tests. It also saves multiple .pkl (pickle) files with the tracked metrics for graphical analysis.

4.3 Experimental Design

There exist obvious correlations between the parameters in the evolve method (ex: mut_prob and mut_alg_prob) however for time-saving purposes, we decided on a top-down approach where we start by tuning the population size and then every other parameter sequentially, with a greedy approach (only accepting a different parameter if it improves on the base model). This way we can arrive at a locally optimized model.

V. Results

5.1 Tuning

The target chosen is in Figure 7, with an image shape of 25x25 pixels. We decided to do 100 trials per model instance and the number of generations will be decided in the first tuning. The range of parameters tuned and the initial base model will not be explicit in the discussion, to see them the reader is referred to Figure 9. The following parameters were tuned, in respective order: Population size. The values tested were 10, 50, and 100 and surprisingly the best population size found was 10, at a 0.05 significance level. In Figure 2 we can see that in earlier generations 10 does not perform as well but in passed generation 950 it gives better results. For this reason from now on we will run 2000 generations for each trial. This result is a very good sight because pop_size of 10 is also better from a computational effort point of view (Figure 3); tour_size, returning a value of 0.2, at a 0.05 significance level. With the ideal tour_size, we then tune the selection algorithm. Tournament Selection gave better results, at a 0.05 significance level; mutation_alg_prob with a result of 1, agreeing with the base model. However, there was no statistical evidence to say that using values from [0.8, 1] was making the model worse. The only statistical evidence backs the fact that 1 is better than 0.7, at a 0.05 significance

[1] This can be used in this circumstance despite being non-parametric. However, this does not make much sense since we just tested for normality. The tests were all run like this and we did not have time to rerun.

level. So we decided to change mutation_alg_prob to 0.9 to add diversity to the population; mut_prob which returned 0.07, better than all other parameters tested at a 0.05 significance level; xo_alg_prob which returned [0.25, 0.7, 0.05], better than all other parameters tested at a 0.05 significance level. This is surprising since blend_crossover resembles a geometric crossover and in principle would be better for problems where a distance to a target is analyzed; mirror_prob returning 0.1 with no statistical evidence to say it is better than the base model value 0.05. Because of this and the fact that mirror_prob helps with the lack of color in the sides of the image, we continued with 0.1; xo_prob returning 0.6, better than the base model at a 0.05 significance level; Next we tested fitness sharing. However, this operator makes the algorithm very slow (computes the pairwise distance between individuals) so we kept the model after xo_prob tuning, turned fitness sharing on, and tuned fs_sigma for 200 generations by trial. fs_sigma tuning returned 0.7, with statistical evidence to say it is better than 0.8 but no statistical evidence for the rest of the values tested (noisy results). So we kept 0.7 and tested the model after xo_prob (again for just 200 generations) tuning against the same model with fitness sharing and fs_sigma=0.7. There was no statistical evidence to say fitness sharing improved the model and since it increased computation effort significantly we decided to drop fitness sharing; With the model after xo_prob tuning, we tuned mutation_size returning 12, with no statistical evidence that it improved the model. So our final model is the one after xo_prob tuning (Figure 9). Examples of individuals before tuning and after tuning are in Figure 7.

5.2 Discussion

Looking at Figure 4 [2] and Figure 5 we see that the tuning process not only improved the model but also increased population variance which indicates that the final model has more diversity in the population [3]. In Figure 6 we see the average run of the different models. The fact that the ideal population size is 10, much lower than the number of pixels in the images, raised the suspicion that this model could be good at generalizing for bigger images. We could not make statistical tests to back this claim due to the computational effort of running a bigger image and time constraints however we compared an instance of the final model, with a target shape 25x25, against an instance of the final model with a target shape 100x100. To do this we need to modify the fitness in the following way: $F' = \frac{1}{N}\sum_{ij} d_{ij}$ where N in the number of pixels in the image. This way F' is the average delta-e distance of the pixels and does not depend on image size. The fitness for 25x25 was $F' = 59.2$ and for 100x100 $F' = 97.4$. Looking at Figure 8

So we cannot say the model generalizes well, although with much uncertainty since there was no statistical study. Either way the final result for a bigger image can be found in Figure 8.

VI. Possible improvements and Conclusion

The local search done was sufficient to improve the model, but given the correlation between multiple parameters in the evolve method, it could be best to search groups of parameters at the same time instead of just one. For example, tuning (mut_prob, mut_alg_prob) together as tuples of values could uncover the correlation between these two parameters. This was not done due to the lack of computational resources that this type of approach would require (more exhaustive search). Another thing that could be improved is trying more methods to deal with genetic diversity. The one implemented (fitness sharing) was not useful in the end but many others could help. More parameters could have been tuned. However, the current approach proved capable of dealing with the image recreation problem.

[2] The fitness sharing tuning is not presented because of report space and lack of relevancy for the final model

[3] There are better metrics for measuring genetic diversity however the saved history at each tuning did not keep the necessary information for calculating just metrics, since most times they are calculated at the population level. Since we did not think of this earlier and rerunning the tuning phase was not possible due to time constraints such metrics are not presented

VII. Annex

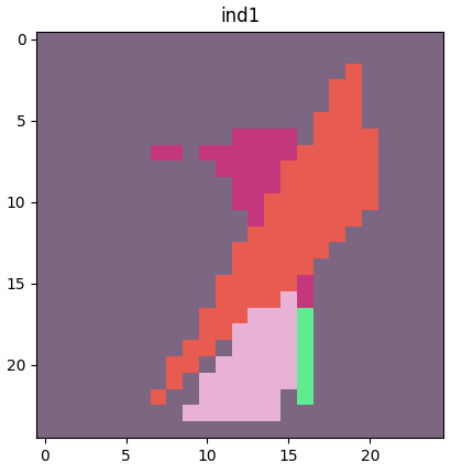

Figure 1: Example of an individual after population initialization

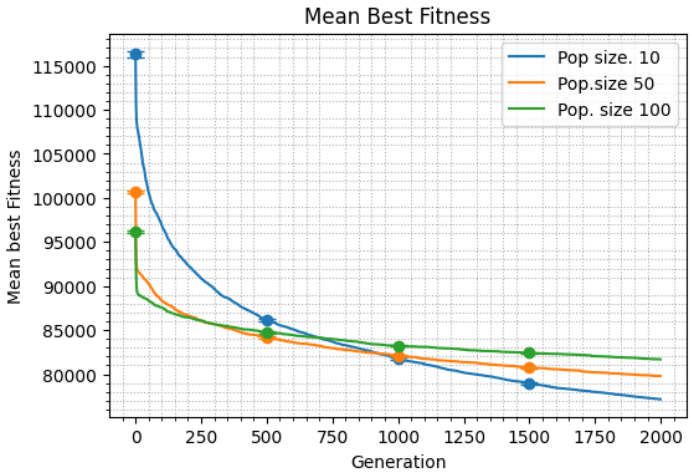

Figure 2: Average runs for every param_range value when tuning pop_size

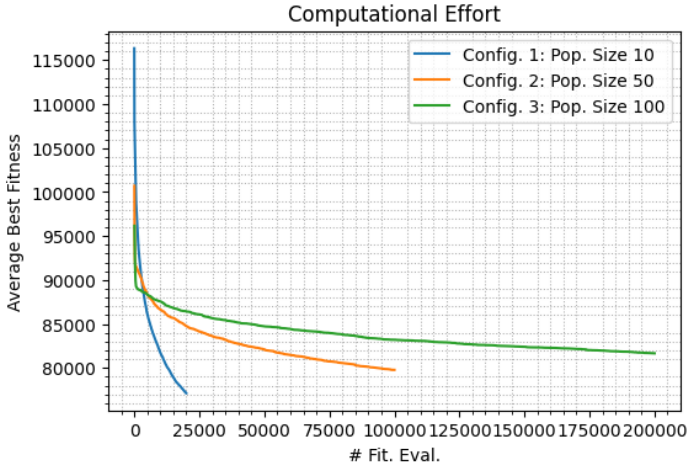

Figure 3: Comparison of computational effort between pop_size parameters

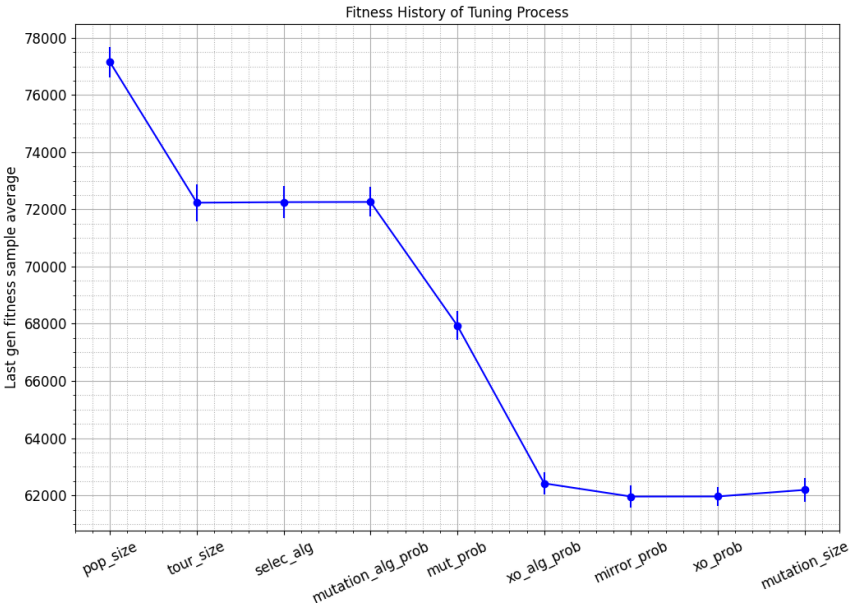

Figure 4: Last generation fitness sample average of best models in each parameter. The parameters are in the same order they were tuned in. The error bars are the Standard error of the mean.

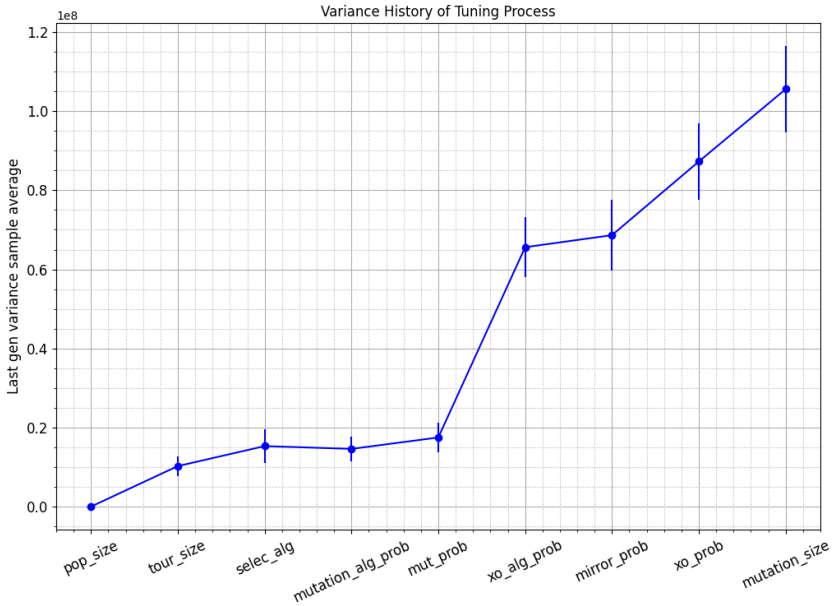

Figure 5: Last generation variance sample average of best models in each parameter. The parameters are in the same order they were tuned in. The error bars are the Standard error of the mean.

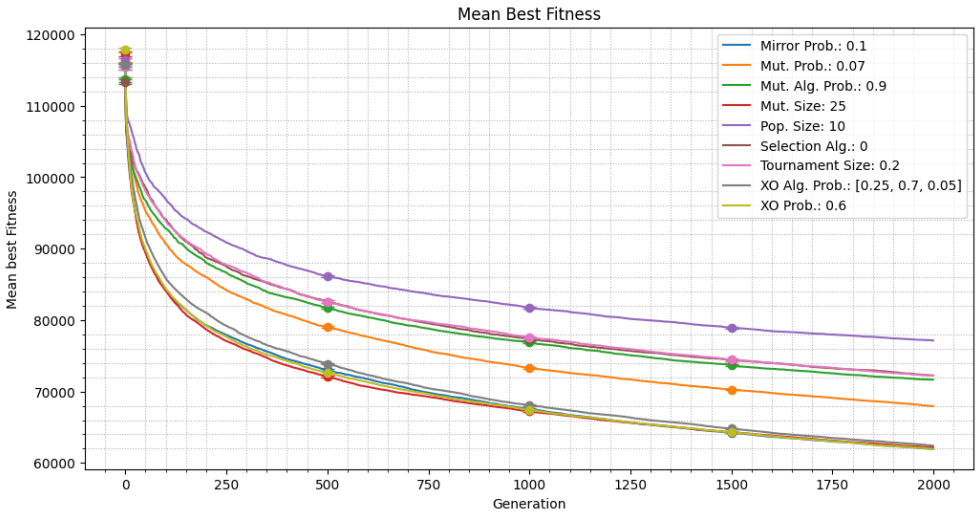

Figure 6: Average runs for best parameter(s) across all configurations

Figure 7: Target (left), Best Individual base model (Center), Best Individual final model (right). 25x25 pixel images. Both ran for 70000 generations.

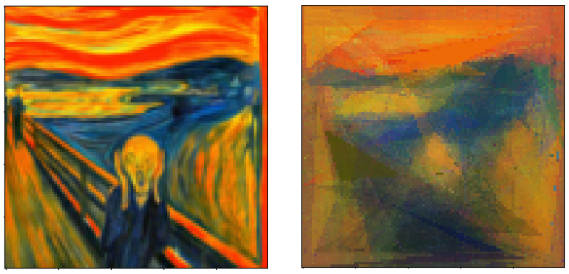

Figure 8: Target (left), Best Individual final model (right). 100x100 pixel images. Ran for 70000 generations

	Evolution function kwargs base model	param_range	Evolution function kwargs final model
gens	2000	----------	2000
selection	[tournament_selection, fps]	----------	[tournament_selection, fps]
selec_alg	1	[0, 1]	0
tour_size	0.1	[0.1, 0.2, 0.3, 0.4]	0.2
mutation	[polygon_mutation,pixel_mutation_random]	----------	[polygon_mutation,pixel_mutation_random]
mut_prob	0.04	[0.02, 0.03, 0.04, 0.05, 0.06, 0.07]	0.07
mutation_alg_prob	1	[0.7, 0.8, 0.9, 1]	0.9
pixel_mutation_same_color	True	----------	True
mut_vertices_range	[3, 5]	----------	[3, 5]
mut_poly_range	[1, 1]	----------	[1, 1]
mutation_size	25	[8, 10, 12, 15, 18, 20, 22, 25]	25
mut_pixel_range	[3.125, 6.25]	----------	[3.125, 6.25]
crossover	[blend_crossover,cut_crossover,pixel_crossover]	----------	[blend_crossover,cut_crossover,pixel_crossover]
xo_prob	0.8	[0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]	0.6
xo_alg_prob	[0.7, 0.25, 0.05]	permutations of [0.7, 0.25, 0.05]	[0.25, 0.7, 0.05]
mirror_prob	0.05	[0.05, 0.1, 0.15, 0.2, 0.25]	0.1
elitism	True	----------	True
fitness_sharing	False	[True, False]	False
fs_sigma	1	[0.5, 0.6, 0.7, 0.8, 0.9, 1]	1
early_stopping	1500	----------	1500

Figure 9: Table with base model and final model parameters. It also has the param_range used in tuning (not ordered in tuning order).

References

[1] Vanneschi, Leonardo, and Sara Silva. Lectures on Intelligent Systems. Springer, 2023.

[2] Charmot, S. (2023, December 31). A true genetic algorithm for image recreation — Painting the Mona Lisa. Medium. https://medium.com/@sebastian.charmot/genetic-algorithm-for-image-recreation-4ca546454aaa

[3] Riemersma, T. (n.d.). Colour metric. https://www.compuphase.com/cmetric.htm