Genetic algorithms applied to image color reduction

1st María Quirós Quiroga
Universidad de Sevilla
Sevilla, Spain
marquiqui@alum.us.es

***Abstract*—This paper aims present a Genetic Algorithm (GA) approach to image color reduction by reducing the color palette of digital images while trying to preserve the original information of the image, so that it is fully interpretable and understandable to the viewer. The algorithm receives as input an image and a target number of colors and evolves a palette of colors, represented as RGB values. Experimentation has been carried to assess the performance of custom crossover and mutation operators, caching and mutation adaptation. A fitness function has been developed to evaluate the individuals based on how closely each pixel of the reduced image matches the original image. The GA was tested on images of varying resolution, complexity and color diversity. The results demonstrate that the proposed method can drastically reduce the number of colors while maintaining perceptual similarities with the original image. A convergence study and a hyperparameter tuning were also conducted. This approach may be helpful for example in the context of retro gaming or image compression.**

***Index Terms*—genetic algorithm, crossover, mutation, palette reduction, image comprehension, color**

I. Introduction

Digital images are a part of our day-to-day lives. They are widely used in modern applications, including entertainment, communication, education, and data analysis. Capturing devices like cameras have evolved to provide higher and higher quality pictures. However, in many contexts like retro video game graphics, printing and image compression the quality of these pictures must be reduced.

One effective technique to deal with these cases is color palette reduction. The objective of this technique is to simplify the image by reducing the number of colors that appear in it. This method not only compresses the image, but can also enhance processing efficiency when applying transformations that rely on color and can be used for stylistic reasons, for example pixel art. The challenge is finding the set of colors that keeps as much information of the original image as possible so that the observer can still fully understand and interpret the result. Traditionally, methods like K-means clustering and the median cut algorithm have been widely used for this purpose and are still researched nowadays [1].

The approach followed in this project is to develop a Genetic Algorithm (GA) in order to reduce the color palette of images while preserving visual quality and maintaining as much information of the original image as possible. The objective is to create compressed versions of the provided images by finding and applying a palette with a limited number of colors that are selected by evolving a population of randomly generated initial color sets (palettes).

Genetic Algorithms are evolutionary algorithms based in real-life natural selection. Given an initial population, a fitness function is defined to determine the best-fitted individuals for survival, which are passed on to the next generation. The rest of individuals of the new generation are generated by mutation or crossover, mimicking evolution and reproduction. The principles of these algorithms were laid out by Holland in 1975 [2] and are still influential today.

Genetic Algorithms (GA) are particularly effective for optimization problems in which the solution space is wide and complex [3]. Modern digital images often contain thousands of colors, subtle variations barely perceptible to the human eye. This results in a very large space when trying to reduce the palette, often to a small number of colors - under a thousand-, while maintaining the same visual quality.

In this project, Genetic Algorithms (GA) are used to evolve a set of candidate color palettes, guided by a fitness function that measures the visual similarity, using the LAB color space, between the original and the reduced image.

II. Structure of the Work

The purposed method applies a Genetic Algorithm (GA) to perform color palette reduction for digital images. As mentioned before, the system receives as input an image and the desired number of colors n. Then, it evolves the population individuals -candidate solutions- through selection, crossover and mutation to produce the next generation, seeking to minimize the difference between the original and the reduced image.

The algorithm will halt either when the predefined maximum number of generations is reached or when the population remains stagnant-the best fitness does not increase-for a number of generations exceeding the configured stagnation threshold.

The implementation has been carried out in Python, with an emphasis on modularity and ease of use. The core components- evolution, mutation, crossover, selection, fitness...-have been developed as different functions to facilitate experimentation. Parameter settings (e.g., population size, mutation rate, crossover strategy) are fully configurable to support extensive testing across different conditions.

A. Individual representation

The first challenge when it comes to defining a Genetic Algorithm (GA) is determining the representation of the individuals of the population. Each individual should be a

possible solution to the problem and to define them, we must specify both a genotype - the encoded representation of a given possible solution - and a phenotype - the meaning behind the representation and how it can be applied in the domain.

In our case, each individual represents a color palette, where each gene is a color of the palette. Two versions of the problem have been developed, one with a restricted and another with an unrestricted palette. In the unrestricted palette version, any color can be a gene in one of the individuals, that is, palettes may have any possible color. In the unrestricted version, palettes may only have colors included in the original image. This second version reduces the search space, as we will see in the experimentation.

Each palette is encoded as list of n RGB triplets. RGB is a color space in which each color is represented by a triple with three values representing the quantity of Red, Blue and Green it contains. This structure allows for easy mapping of each pixel of the original image to its nearest color in the palette during fitness evaluation.

B. Fitness Function

The fitness function evaluates how similar the compressed image is to the original image, by computing the information loss between both. As described in Algorithm 1, for each pixel, the nearest color from the palette is chosen and to be the color of that same pixel in the reduced image. Then, the reduced image is compared to the original one using Euclidean distance over all pixels.

Since the goal is to minimize the distance between the original and the reduced image, the fitness function was designed to increase as distance decreases. So, it is computed as the inverse of the distance plus one, ensuring that the fitness value is contained in the range (0,1], with a perfect match corresponding to a fitness of 1. The addition of 1 in the denominator prevents division by zero in cases where the distance is extremely small or zero.

Additionally, a penalty term was incorporated into the fitness function to consider the fact that colors should not be repeated in the final palette. Since a palette with repeated colors reduces the diversity of the representation (e.g. a palette of n colors with one duplicate behaves as a palette with n-1 colors), it is important to ensure that all colors are unique. To compute the penalty, the ratio of unique colors to the total number of colors on the palette was calculated, and subtracted from 1. This value was then subtracted from the fitness score. As a result, a palette with no repeated colors receives a penalty of 0 while a palette with maximum duplication receives a penalty close to 1. Since the fitness value is between 0 and 1, the penalty is scaled proportionally and does not dominate the fitness value.

As expected, this fitness function has a relatively high computational cost, as multiple iterations must be made along each pixel of the image. An alternative fitness function with a lower computational cost was considered. The fitness was represented by the sum of the number of pixels that had one of the colors of the palette in the original image. However, it was discarded because, even though the algorithm was faster, this fitness function was not really useful as the ideal individual could be easily obtained by creating a palette with the n most common colors on the original image.

Algorithm 1: Fitness Computation with RGB Color Space

Input: An individual (a color palette), the original image
Output: Fitness value

```
Copy originalImage to paletteImage;
foreach Pixel in originalImage do
    foreach Color in the palette do
        Compute the Euclidean distance from the color
          of the pixel to the color in the palette;
        Store the distance in "distances";
    Change the color of the pixel in the same position
      as the given pixel of paletteImage to the
      color with the smaller distance to the
      originalImage pixel in distances.
Compute the euclidean distance between
  originalImage and paletteImage pixel by
  pixel;
Compute the penalty as 1 − unique colors in palette / number colors in palette;
fitness ← 1/(1+distance) − penalty;
return fitness;
```

During preliminary testing, a significant issue with the original fitness function was identified. Specifically, color distances in the RGB color space do not reliably represent the perceptual similarities between colors. For example, two visually similar shades of green may have a higher distance in RGB than a green and yellow pair, due to the non-uniform nature of the RGB model. In some cases, this led to non-intuitive fitness behavior.

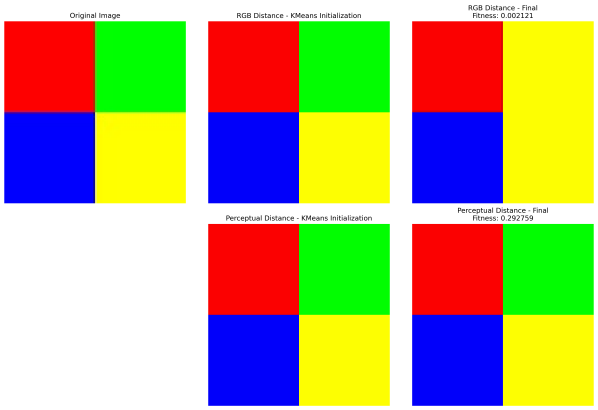


Fig. 1. At the first column, the original image, on the second one the K-means initialized individual and in the third one, from top to bottom, the solution with higher fitness found with RGB and lab color spaces.

This problem was especially noticeable when testing with an image that clearly contained four distinct color regions (first

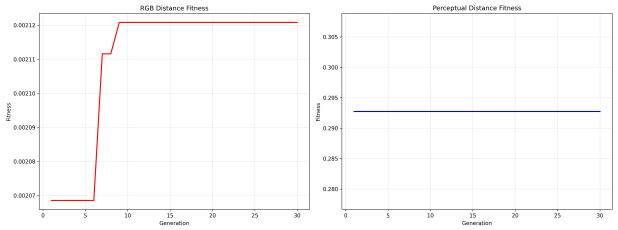


Fig. 2. Graphical representation of the best fitness function by generation. At the left, the RGB color space implementation, at the right the lab color space implementation.

column of Figure 1). When requesting a four-color palette and adding an individual generated by applying K-means with four centers (second column) to the initial population of the algorithm, the algorithm identified a new solution with a higher fitness value. However, as shown in the third column (top row), the resulting image after applying this "best" palette clearly retained less visual information than the K-means initialization. This contradiction revealed a flaw in the perceptual accuracy of the RGB fitness function.

As a result, the fitness function was revised to operate in the LAB color space, where the distances correspond more closely to the perception of human color [4]. After this change, the issue did not persist. As can be seen in the bottom right of Figure 1, the LAB-based fitness function better measured the retention of the original image information. The final version of the fitness function algorithm is provided in Algorithm 2.

Algorithm 2: Fitness Computation with LAB Color Space

Input: An individual (a color palette), the original image
Output: Fitness value

Convert the original image to LAB color space and store as `imageLAB`;
Copy `imageLAB` to `paletteImageLAB`;
foreach *Pixel in imageLAB* **do**
  foreach *Color in the palette* **do**
    Compute the Euclidean distance from the color of the pixel to the color in the palette;
    Store the distance in "distances";
  Change the color of the pixel in the same position as the given pixel of `paletteImageLAB` to the color with the smaller distance to the `imageLAB` pixel in `distances`.
Convert the `paletteImageLAB` to LAB color space;
Compute the euclidean distance between `imageLAB` and `paletteImageLAB` pixel by pixel;
Compute the penalty as $1 - \frac{\texttt{unique colors in palette}}{\texttt{number colors in palette}}$;
$\texttt{fitness} \leftarrow \frac{1}{1+\texttt{distance}} - \texttt{penalty}$;
return `fitness`;

C. Selection

There are three main genetic operators: selection, crossover, and mutation. For the sake of experimentation, multiple operators of each type were implemented.

Regarding selection, three widely used methods were implemented: roulette wheel selection, tournament selection, and rank-based selection. As these are well-known techniques in the GA field, only a conceptual description is provided rather than the full pseudocode.

In roulette wheel selection, each individual's probability of being selected is proportional to its fitness value, resembling the spin of a wheel, where individuals with higher fitness are assigned a larger section on the wheel, increasing their chances of being selected. For this particular implementation, due to the existence of negative fitness values, all values had to be shifted before summing them, setting the lowest negative score at 0.

Tournament selection involves randomly selecting a subset of individuals (a "tournament") from the population and keeping the fittest among them. This process is repeated for each selection, introducing selection pressure and randomness, as the best individual will always be chosen if they are selected but has the same probabilities of being chosen for the tournament as the rest.

Rank-based selection first sorts the population based on fitness values and then assigns selection probabilities according to each individual's rank position in the sorting rather than fitness. This approach mitigates the risk of premature convergence caused by very high-fitness individuals dominating the selection process.

D. Crossover

In this project, crossover operators take two parents as input and return a single child that is obtained by combining the given parents. Three crossover operators were implemented, two well-known ones: uniform crossover and one-point crossover, and a custom one designed specially to fit the color reduction optimization problem, the closest pairs crossover.

The uniform crossover has a probabilistic approach to the parent combination. For each color position in the child palette, the operator generates a random binary decision with equal probability (50%) to determine whether to inherit the color from the first or the second parent. This approach maximizes diversity, allowing the child to potentially combine the best genes from both parents.

The one-point crossover operator implements a traditional strategy adapted for color palette representation. The operator selects a random crossover point between positions 1 and n-1 where n is the total number of colors in the palette. The resulting child inherits all colors from the first parent up to the crossover point and all remaining colors from the second parent. This crossover operator is especially relevant when the other of the genes is important. In our context, this is not the case, but it will be explored nonetheless.

Finally, the custom crossover operator implemented is the closest pairs crossover, a new operator designed for this domain. The goal of this crossover algorithm 3 is to take advantage of perceptual color relationships to introduce more diversity into the palette. For each color in the first parent, the closest perceptual match in the second parent is identified using Euclidean distance. Then, the child randomly inherits one color from each pair, maximizing color diversity.

Algorithm 3: Closest Pairs Crossover Operator

```
Input: First parent and second parent palettes
Output: One child palette
Create the empty child;
foreach color in the first parent do
    Find its closest match in the second parent using
      Euclidean distance to the color of the first parent;
    Select a random value between 0 and 1 (50%
      chance);
    if under 0.5 then
        Add color from first parent to child;
    else
        Add closest color from second parent to child;
if child has duplicated colors then
    Remove duplicate color;
    if there are colors in the parent not in the child
      then
        Add a random color from the parents that is
          not already in child;
    else
        Add a random color (from original image if
          restricted, fully random if not);
return child;
```

To avoid potentially repeated colors in the child, which would be penalized, when duplicated colors are detected, the algorithm will replace one of them with a color from any of the parents that is not already in the child palette. If there are not enough unique colors, a random color is selected. The random color will be randomly generated for the unrestricted palette version of the problem and selected from the original image palette for the restricted version.

This crossover method takes into account the perceptual relation between the colors in the palette to try to keep as much information as possible, while promoting diversity by introducing variations.

E. Mutation

Regarding mutation, two operators were implemented, a simple color-by-color random probability mutation and a custom diverse operator designed specially for this domain.

The standard random mutation operator follows the classical Genetic Algorithm (GA) approach, and for each color position on the palette, the operator performs a mutation with a given probability equal to the given mutation rate. When mutation occurs, the color is generated randomly. For unrestricted palettes, three numbers are randomly generated in the range 0 to 255 and used as RGB values. On the other hand, for the restricted version, a random color is chosen from the color palette of the original image. This approach ensures exploration of the color space by introducing novel color combinations that may no exist in any of the individuals of the current generation.

The diversity mutation operator has the objective of diversifying the palette as much as possible, so palettes do not develop redundant or perceptually similar colors that may reduce the original image information represented. The algorithm 4 begins by converting the palette to the LAB space to be able to compute the distances corresponding to human perception. The algorithm then identifies the most similar color pair by obtaining the distances from every color to every other color of the palette. The closest pair of colors undergoes mutation with a mutation rate probability, potentially being replaced with randomly generated colors. Once again, for the unrestricted version, the colors are fully random and, for the unrestricted version, they are randomly selected from the original image color palette.

Algorithm 4: Diversity Mutation Operator

```
Input: A palette to mutate
Output: A mutated palette
Convert the palette to mutate to LAB color space;
foreach color in the first parent do
    Compute the distance to every other color in the
      palette to mutate;
Get the two closet colors pair ;
foreach color in the pair do
    if a random number between 0 and 1 is lower than
      the mutation rate then
        Mutate the position in the palette by generating
          a random color (fully random -unrestricted- or
          from the original image palette - restricted-);
return mutated palette;
```

F. Algorithm Flow

In this section, the general flow of the genetic algorithm is presented. For simplicity, not all auxiliary operations (e.g., logging, storing, or visualization) are detailed, as they are not essential to the core process.

The algorithm shown in Algorithm 5 begins by generating a random population of color palettes. This random generation depends on the selected mode: restricted or unrestricted palette generation. In restricted mode, each gene (color in the palette) is randomly sampled from the set of colors in the original image. On the other hand, the unrestricted generates colors by independently sampling each RGB component with values between 0 and 255. A random individual is sampled and added to the population until the population size is reached.

An optional feature allows including a K-means-initialized individual into the initial population. This individual is obtained by applying a K-means clustering approach to the original image, with the objective of finding the n most dominant colors, which are then used as the palette.

For each generation, the fitness of each individual in the population is computed using the previously defined fitness function. The algorithm identifies the best individual in the current generation and updates the global best if an improvement is found.

An adaptation mechanism is applied using a stagnation counter, which tracks the number of consecutive generations without improvement in best fitness. When stagnation goes over the adaptation threshold, the mutation rate is increased by multiplying it by the adaptation rate (which should be greater than 1). If improvement is resumed, the mutation rate is reset to its original value.

The algorithm checks for stopping conditions at the end of each generation. If the number of stagnant generations exceeds a predefined threshold, the algorithm terminates early and returns the best solution found. Otherwise, the population is evolved using the configured selection, crossover and mutation operators and the next generation starts.

The procedure used to evolve the population is presented in Algorithm 6. The process begins by initializing an empty list to hold the new population.

As the first step, elitism is applied: the top *e* best individuals of the original population, ranked according to fitness, are copied into the new population. The value of *e* is a configurable parameter and ensures that the best solutions are preserved over generations.

While the number of individuals of the new population remains under the population size, new individuals will be added by selection, crossover, and mutation. For each new individual, a decision is made based on a random probability compared to the crossover rate. If the crossover is to be performed, two parent individuals are selected from the original population using the configured selection method, and the specified crossover operator is applied to generate a new individual. If crossover is skipped (the random probability is above the crossover rate), a single individual is selected as the new individual.

Then, the resulting individual undergoes mutation, which may mutate some of its genes with a probability proportional to the mutation rate configured. This step introduces generic diversity to help the algorithm consider new areas of the search spaces (i.e. new colors).

Finally, the resulting individual is added to the population. Once the new population reaches the desired size, it replaces the original population for the next generation.

III. Performance considerations

Due to the high computational cost of the fitness function - particularly the computations of the distance pixel by pixel between the original image and each color in an individual- several optimizations were implemented to improve efficiency.

Algorithm 5: General flow of the genetic algorithm

Input: Configuration parameters like crossover rate, mutation rate, population size, number of generations...

Output: The best individual found, the fitness corresponding to the best individual, a best fitness and average fitness per generation history of values, the image corresponding to the best palette, the K-means initialization image and palette if applicable and some performance statistics

```
Initialize the population;
foreach generation until the maximum number of generations configured do
    Evaluate the fitness of the individuals of the population;
    Find best individual in generation ;
    Update overall best individual if the best of the generation is fitter than the overall best ;
    if stagnation is detected then
        Increase mutation rate by multiplying it by the adaptation rate to increase diversity ;
    else
        if improvement has been made then
            Reduce mutation rate to original value;
    if it is not the last generation and the stagnation threshold has not been reached then
        Evolve the population ;
    else
        Break loop;
return best individual, fitness of best individual, best fitness for each generation, average fitness of population for each generation, image obtained by applying the best individual, K-means individual image and palette, performance log with performance statistics ;
```

A. Use of NumPy

First, traditional Python loops that involved computations on arrays were modified to make full use of *NumPy* vectorization capabilities. This was especially important in the image transformation step, where each pixel of the image is mapped to the closest color in the given palette. Since this operation is performed each time fitness is evaluated, optimizing it could provide a substantial performance increase.

To evaluate the impact of this optimization, a comparison was made between two runs, one with a basic implementation using Python loops and the other with an optimized version using *NumPy* vectorized operations. Both used the same image (`chihiro.jpg`), color count (6), and configuration.

As shown in Table I, the *NumPy* implementation achieves a significant speed-up with respect to the for loop version,

Algorithm 6: Algorithm to evolve population

Input: Population and computed fitness scores
Output: New population

```
Initialize new population ;
Elitism: Add best e individuals of the given population to the new population;
while the new population does not reach maximum size do
    Generate a random number ;
    if the random number is under the crossover rate then
        Use selection to obtain two individuals from the original population, the parents ;
        Obtain the child by applying the crossover operator over the parents ;
    else
        Use selection to obtain an individual from the original population;
    Apply the mutation operator over the new individual;
    Add new individual to new population;
    return new population
return best individual, fitness of best individual, best fitness for each generation, average fitness of population for each generation, image obtained by applying the best individual, K-means individual image and palette, performance log with performance statistics ;
```

reducing the execution time per generation by an order of magnitude. For example, in Generation 1, the loop-based version took approximately 89 seconds, whereas the vectorized version was completed in only 9 seconds. This improvement is consistent across generations, enabling faster convergence and easing practical experimentation.

TABLE I
PERFORMANCE COMPARISON: LOOP VS. NUMPY IMPLEMENTATION

Generation	Loop-Based Time (s)	NumPy Time (s)
1	89.24	9.19
2	74.71	7.37
3	76.46	7.36
4	62.05	7.22
5	57.06	4.89
6	39.93	4.65
7	49.79	3.74
8	32.24	4.16
9	57.89	2.77
10	22.29	5.16
Avg/Gen	**56.57**	**5.36**

While the optimization reduces code readability, using complex *NumPy* expressions, the performance gains far outweigh trade-offs. This optimization was crucial in enabling tests with bigger images and larger color counts with reasonable execution times.

B. Use of Caching

To further optimize performance, caching mechanisms were implemented to avoid repeated fitness computations. As stated above, for this algorithm, fitness computations are relatively expensive, so reducing the number of unnecessary computations is important to ease experimentation.

In many cases, identical individuals can appear multiple times across generations, especially due to elitism, crossover, or low mutation rates. Without caching, each identical individual would undergo the same expensive evaluation multiple times, but, by storing previously computed fitness values and retrieving them when needed, the algorithm was able to reduce the number of expensive evaluations required per generation.

The fitness values are stored in a caching dictionary using a hash of the individual's genome (i.e., color palette) as a key. When an individual reappears, its fitness is retrieved from the cache instead of recalculated. This optimization is especially beneficial for larger populations or configurations with higher color counts.

Table II summarizes the performance comparison between runs with and without caching using the same image (`houses.jpg`) and configuration. Across all number of colors for the palettes, enabling caching significantly reduces execution time while maintaining equivalent best fitness. For example, in the case with 4 colors, the total execution time dropped from 185.74 seconds to 110.13 seconds, a reduction of over 40% without loss of quality.

TABLE II
EFFECT OF CACHING ON FITNESS EVALUATION RUNTIME

Colors	Configuration	Avg. Fitness	Execution Time (s)
4	caching_enabled	0.054956 ± 0.001070	110.13 ± 3.56
4	caching_disabled	0.054828 ± 0.001894	185.74 ± 4.57
10	caching_enabled	0.079221 ± 0.001477	207.40 ± 7.07
10	caching_disabled	0.078283 ± 0.000840	239.22 ± 2.06
20	caching_enabled	0.093664 ± 0.001180	304.02 ± 10.90
20	caching_disabled	0.091647 ± 0.000428	342.67 ± 4.28

Table III presents the cache hit ratios. Although the hit rates decrease with higher color counts (due to the bigger number of possible palettes and more unique individuals), even a reduced hit rate results in meaningful time savings.

TABLE III
CACHE HIT RATIOS AND COUNTS

Colors	Configuration	Hit Ratio	Hits	Misses
4	caching_enabled	42.20%	633	867
10	caching_enabled	16.53%	248	1252
20	caching_enabled	10.07%	151	1349

IV. EXPERIMENTS AND RESULTS

To evaluate the effectiveness of the proposed Genetic Algorithm (GA) for color palette reduction, a series of controlled experiments were carried out. The objective of these experiments is to assess the impact of the different design configurations, such as selection strategies, crossover and mutation operators, and rate adaptation by measuring execution time,

fitness of the solutions, and performing a qualitative analysis of the final images generated.

The tests were performed on multiple images with different colors and sizes, with special emphasis on two cases: `cat.jpg` (Figure 3), a complex real-world photograph, and `squareSix.jpg`(Figure 4), an image composed of six clearly distinguished color regions designed for testing. Each experiment was run using three different palette sizes: 4, 10, and 20 colors.

To account for the random nature of the GA, each configuration was executed three times. The execution times and fitness values included in this section are averaged over these runs. However, for qualitative evaluation, the best individual across all repetitions was selected and used to generate the final output image.

Fig. 3. Real-world test image (`cat.jpg`).

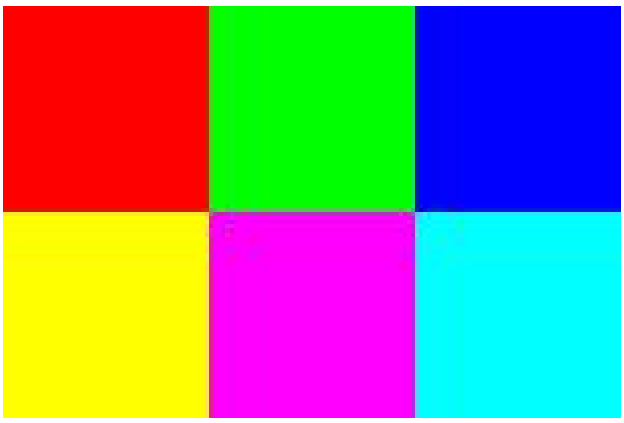


Fig. 4. Test image (`squareSix.jpg`) with six visually distinct regions.

In the following subsections, the results of each set of experiments will be presented along with an analysis of their implications. Unless otherwise specified, all experiments were conducted using the following base configuration. The population size was set to 10, and the algorithm was run for 30 generations. A tournament selection method with a tournament size of 3 was used. The crossover and mutation rates were set to 0.8 and 0.2, respectively, with an elite individual preserved for each generation. The default configuration used a restricted palette mode (that is, colors sampled from the original image) and employed the *closest pairs* crossover method. No adaptation or halting based on stagnation was enabled unless explicitly tested. Caching was enabled by default to improve performance. Unless stated otherwise, K-means initialization and the diversity-based mutation operator were disabled.

TABLE IV
DEFAULT GENETIC ALGORITHM PARAMETERS

Parameter	Value
Palette mode	Restricted
Population size	10
Generations	30
Crossover rate	0.8
Mutation rate	0.2
Selection method	Tournament (size = 3)
Elitism	1
Adaptation rate	1 (disabled)
Adaptation threshold	None
Halting stagnation threshold	None
Crossover method	Closest pairs
Diverse mutation	Disabled
K-means initialization	Disabled
Caching	Enabled

For each experiment, a set of images showing charts and comparisons was automatically generated. These include comparative plots of nest fitness, execution time, mutation rate evolution, stagnation frequency, and cache hit ratios across the multiple configurations tested. In addition, line plots of the evolution of fitness per generation were produced for each tested color count by averaging the runs. A statistical summary report in Markdown format was also generated, detailing performance metrics and variance across runs. Selected visualizations are included in this report for illustration purposes.

A. Effects of Palette Restriction and K-means Initialization

In this experiment, the impact of two key design choices in the algorithm configuration are evaluated:

- **Palette Restriction**: Whether the candidate palette colors are limited to those found in the original image (*restricted*) or can include any RGB values (*unrestricted*).
- **K-means Initialization**: Whether one individual in the initial population is generated using K-means clustering to capture the dominant colors in the image.

Four configurations were tested:

1) `unrestricted`: Full RGB palette, random initialization.
2) `unrestricted_kmeans`: Full RGB palette with initial K-means individual.
3) `restricted`: palette limited to image colors, random initialization.
4) `restricted_kmeans`: palette limited to colors in the original image with initial K-means individual.

Final Fitness and Runtime: Tables V and VI summarize the average best fitness and execution times for each configuration.

TABLE V
AVERAGE BEST FITNESS AND EXECUTION TIME ON `cat.jpg`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	unrestricted	0.0595	31.67
	unrestricted_kmeans	**0.1044**	27.51
	restricted	0.0970	29.47
	restricted_kmeans	**0.1044**	31.00
10	unrestricted	0.0838	62.20
	unrestricted_kmeans	0.1455	61.54
	restricted	0.1368	84.80
	restricted_kmeans	**0.1464**	57.60
20	unrestricted	0.0928	113.11
	unrestricted_kmeans	**0.1953**	160.36
	restricted	0.1655	169.09
	restricted_kmeans	**0.1953**	167.78

TABLE VI
AVERAGE BEST FITNESS AND EXECUTION TIME ON `squareSix.jpg`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	unrestricted	0.0294	4.26
	unrestricted_kmeans	0.0265	3.77
	restricted	**0.0341**	3.68
	restricted_kmeans	0.0323	4.35
10	unrestricted	0.0533	5.98
	unrestricted_kmeans	0.3688	4.03
	restricted	0.2126	6.86
	restricted_kmeans	**0.3723**	4.41
20	unrestricted	0.0888	8.86
	unrestricted_kmeans	**0.5651**	5.66
	restricted	0.4625	8.30
	restricted_kmeans	**0.5651**	5.98

When comparing the unrestricted and restricted palette modes, it can be observed that the best fitness of the restricted configurations consistently exceeds the fitness of unrestricted versions of the algorithm. This aligns with expectations: By limiting the search space to colors already present in the original image, the algorithm more efficiently explores relevant solutions.

As expected, configurations initialized with a K-means individual have also achieved better fitness overall. The K-means palette acts as a heuristic, effectively raising performance for early generations.

Interestingly, due to the use of the caching mechanism, the additional computation costs associated with the generation of the K-means individual is almost not perceived, especially for higher color counts, where the cost is amortized, but there are many cache hits for said individual.

For simplicity, regarding convergence, we will focus on the 10-color case on `cat.jpg`. However, the observed trends are representative of all the color counts and images tested, which can be found in the attached project. Figure 5 plots the best fitness (solid lines) and average population fitness (dashed lines) over generations for the four configurations.

As can be observed, the restricted configuration consistently starts with a higher baseline fitness. This is the expected behavior, as the search space is reduced and the colors generated will exactly match more of the pixels in the image. In addition, both the best and average fitness curves improve steadily. In contrast, the unrestricted configuration begins at

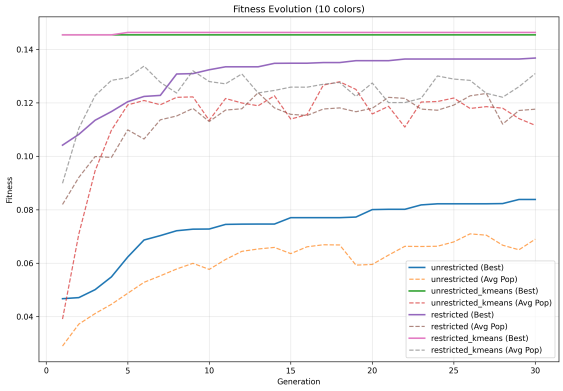


Fig. 5. Fitness evolution over generations for 10-color palettes on `cat.jpg`. Solid lines: best fitness. Dashed lines: average fitness.

a disadvantage due to the lack of information regarding the original image, which broadens the search space. However, the growth pattern is similar, indicating that despite the broad search space, the algorithm can still make progress with sufficient generations and guidance.

This guidance can be obtained by K-means initialization. In both cases, K-means initialization leads to a significantly higher starting fitness, which in the unrestricted variant of the configurations helped direct early generations towards promising regions of the search while still maintaining more freedom to explore the broader search space and possibly identify new promising colors. This effect is especially visible in the average fitness curves, where the unrestricted configuration with K-means quickly reaches the fitness levels of the restricted configuration, demonstrating its guiding capabilities.

Although K-means initialization improves early performance, it also appears to be more prone to stagnation in both variants. The best fitness is barely improved over generations, suggesting that while K-means provide a strong start, it can also reduce diversity, limiting exploration. This highlights a potential trade-off between fast convergence and long-term exploration.

Visual Quality: Figure 6 shows a comparison of the output of each of the configurations for the `cat.jpg` image with 10 colors.

The configurations initialized with K-means produce results that closely resemble the original image. They preserve fine detail and handle gradients correctly, which can be specially seen in the fur and the face of the cat.

Interestingly, even without K-means initialization, the restricted version of the algorithm is capable of generating a faithful image. Although some detail is lost compared to the K-means versions, the result is perceptually coherent.

On the other hand, the unrestricted configuration without K-means, despite having no guidance regarding palette, manages to produce an image that resembles the colors in the original one, even though no colors are exactly the same. However, a closer inspection reveals a key weakness: the final palette

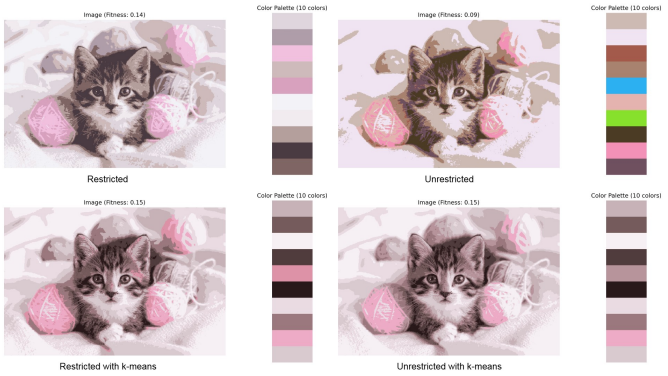


Fig. 6. 10-color palettes on `cat.jpg`: top left restricted, bottom left restricted_kmeans, top right unrestricted, bottom right unrestricted_kmeans.

often includes colors that are not present in the output image at all (in this case, green and blue). These unused colors affect the quality of the final image, as, practically, the palette behaves as if it had fewer colors. These unused colors reflect the challenge of searching in the broader search space, where many generated colors are not useful for representing the original image.

Despite this, this same characteristic could represent a long-term advantage. Given sufficient generations and diversity, the unrestricted version might discover new colors that do not exist in the original image but effectively summarize visual content, enabling perceptual compression beyond simple color matching.

Discussion: From this comparison,we can conclude the following.

- **K-means initialization** provides a strong starting point for the algorithm, improving convergence speed and final fitness, especially for the unrestricted case, where the initial search space is large.
- **Palette restriction** improves performance when K-means is not used, as it restricts the solution space to colors present in the original image, allowing faster convergence.
- **Unrestricted palettes** can theoretically outperform restricted ones by introducing new color combinations that better approximate the image as a whole. However, this advantage comes at the cost of more generations needed to reach high-quality solutions.

B. Effects of the Selection Method

In this experiment, the impact of the selection method used for the algorithm is evaluated. As stated above, three different selection methods were implemented: roulette wheel selection, rank selection, and tournament selection. A total of five configurations were tested:

1) `roulette_selection`: Individuals are selected with probability proportional to their fitness, simulating a weighted ”wheel spin”.
2) `rank_selection`: Individuals are sorted by fitness and assigned selection probabilities based on their ranking rather than fitness.
3) `tournament_size_3`: Tournament selection with groups of 3 individuals, where the best is selected in each group.
4) `tournament_size_5`: Tournament selection with groups of 5 individuals.
5) `tournament_size_10`: Tournament selection with groups of 10 individuals.

Final Fitness and Runtime: Tables VII and VIII summarize the average best fitness and execution times for each configuration.

TABLE VII
AVERAGE BEST FITNESS AND EXECUTION TIME FOR SELECTION METHODS ON `CAT.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	roulette_selection	0.0958	35.82
	rank_selection	0.0963	32.40
	tournament_size_3	0.0967	39.70
	tournament_size_5	**0.0998**	56.15
	tournament_size_10	0.0964	52.78
10	roulette_selection	0.1281	103.14
	rank_selection	0.1306	113.31
	tournament_size_3	0.1347	96.25
	tournament_size_5	0.1376	112.48
	tournament_size_10	**0.1403**	107.15
20	roulette_selection	0.1549	155.00
	rank_selection	0.1625	169.43
	tournament_size_3	0.1633	172.51
	tournament_size_5	0.1666	179.37
	tournament_size_10	**0.1715**	151.04

TABLE VIII
AVERAGE BEST FITNESS AND EXECUTION TIME FOR SELECTION METHODS ON `SQUARESIX.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	roulette_selection	0.0342	4.48
	rank_selection	0.0351	6.46
	tournament_size_3	**0.0382**	9.14
	tournament_size_5	0.0299	8.34
	tournament_size_10	0.0338	6.94
10	roulette_selection	0.3162	11.88
	rank_selection	0.2885	10.74
	tournament_size_3	0.3538	14.39
	tournament_size_5	0.3308	15.06
	tournament_size_10	**0.3550**	15.29
20	roulette_selection	0.4151	18.26
	rank_selection	0.4593	16.55
	tournament_size_3	0.4278	16.15
	tournament_size_5	**0.4736**	18.83
	tournament_size_10	0.4704	9.34

When comparing the three main selection strategies-roulette wheel, rank-based, and tournament selection-we observe that the tournament selection consistently returns higher fitness across nearly all executions. In general, size 5 or 10 selection usually provides the best results. This is particularly interesting, as size 10 is exactly the size of the population. However, the difference is not very significant, so it may have been a coincidence.

However, the increased performance comes at a computational cost. As expected, larger tournament sizes require more comparisons, increasing execution time. This happens particularly in larger images with more colors (like `cat.jpg`). For example, in this image with 20 colors, the execution time for tournament size 5 is about 179 s versus 155 s for roulette. However, the difference is modest and justifiable given the trade-off in performance.

In smaller images like `squareSix.jpg` the execution time differences are smaller. And interestingly, the tournament selection with size 10 is faster than the rest of configurations while maintaining the second-best fitness. This could be caused by the selection pressure, because as the tournament size is exactly the size of the population, the best individual is always chosen, so there are more probabilities of obtaining the same individual multiple times, and so the cache hits increase.

Regarding tournament size, the results show that the fitness differences between the three are small.

For simplicity, regarding convergence, we will focus on the 20-color case in `cat.jpg`. However, the observed trends are representative of all the color counts and images tested, which can be found in the attached project. Figure 7 plots the best fitness (solid lines) and average population fitness (dashed lines) over generations for the five configurations.

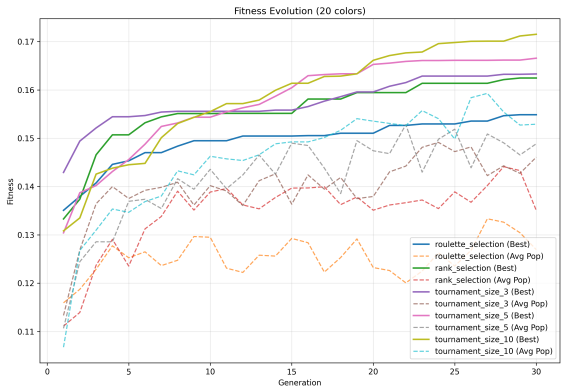


Fig. 7. Fitness evolution over generations for 20-color palettes on `cat.jpg`. Solid lines: best fitness. Dashed lines: average fitness.

As can be observed, the roulette selection method is the worse performing one. While most of the configurations best fitness rapidly increase, the roulette selection method leads to a less steep curve, which mostly plateaus after the tenth generation. This is not the case for the rest of the selection methods, which continue to produce better individuals until the last generations. Furthermore, the average fitness of the population for the roulette wheel selection method is also consistently the lowest.

On the other hand, the rank selection method provides results very similar to the 5 and 3 tournament selection, providing a consistent increase in the fitness. It is also important to highlight that the average fitness for these three changes considerably while maintaining a general tendency to grow. This is also important because it indicates that the population is constantly changing, leaving space to find new better individuals.

The same cannot be said for the tournament of size ten; we can theorize that the increased average population fitness and the small variations from generation to generation are mainly due to the fact that the same individuals are constantly being chosen. This operator does not bring randomness to the algorithm. However, it still seems to be performing the best thanks to the randomness provided by the mutation operator and the crossover operator.

Visual Quality: Figure 8 shows a comparison of the output of each of the configurations for the `cat.jpg` image with 20 colors.

Fig. 8. 20-color palettes on `cat.jpg`

Regarding visual quality, the best results seem to correspond to the tournament selection configurations, especially for the tournament of size five. These images preserve the shading and texture of the fabrics in the scene more effectively. In contrast, roulette and rank-based selection occasionally produce palettes with some strange colors that do not match the original image as well.

Discussion: From this comparison,we can conclude the following.

- **Roulette-wheel selection** consistently produces the lowest fitness values in most configurations. This method is more susceptible to noise, especially when fitness values include negatives, as may happen in our case.
- **Rank selection** offers better performance than roulette, but was generally outperformed by tournament selection, especially when it comes to fitness.
- **Tournament selection** achieves the best results consistently across color counts and test images. It provides a stronger selection pressure, which accelerates convergence towards high-quality solutions. Among the tested sizes, tournament sizes 5 and 10 offer the best solution quality.

C. Effects of the Mutation Operator

In this experiment, the impact of different mutation operators and mutation rates on the performance of the genetic algorithm will be evaluated. The two mutation strategies explained above were considered: a standard mutation, where each gene has a fixed possibility of being mutated, and a diverse mutation, which aims to replace the most perceptually similar colors to promote color diversity. Each was tested with two mutation rates: a lower rate (0.2) and a higher rate (0.6). A total of four configurations were evaluated:

1) `standard_low`: Standard mutation with a low mutation rate. Each gene has a small chance of being replaced independently.
2) `standard_high`: Standard mutation with a high mutation rate of 0.6. Mutations occur more frequently, increasing exploration, but risking disruption of fit individuals.
3) `diverse_low`: Diverse mutation with low mutation rate. The algorithm targets the most similar color pairs in the palette and mutates them selectively.
4) `diverse_high`: Diverse mutation with high mutation rate. This promotes aggressive diversification of similar colors within the palette.

Final Fitness and Runtime: Tables IX and X summarize the average best fitness and execution times for each configuration.

TABLE IX
AVERAGE BEST FITNESS AND EXECUTION TIME FOR MUTATION STRATEGIES ON `CAT.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	standard_low	0.1001	25.93
	standard_high	0.1004	40.11
	diverse_low	0.0910	16.66
	diverse_high	0.0983	34.68
10	standard_low	**0.1364**	49.94
	standard_high	0.1330	53.31
	diverse_low	0.1304	20.59
	diverse_high	0.1312	46.54
20	standard_low	0.1649	126.33
	standard_high	0.1592	168.08
	diverse_low	0.1599	73.20
	diverse_high	**0.1664**	106.01

TABLE X
AVERAGE BEST FITNESS AND EXECUTION TIME FOR MUTATION METHODS ON `SQUARESIX.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	standard_low	0.0318	3.47
	standard_high	0.0353	3.81
	diverse_low	0.0336	2.91
	diverse_high	**0.0357**	4.18
10	standard_low	**0.3056**	7.18
	standard_high	0.3027	5.87
	diverse_low	0.1432	3.47
	diverse_high	0.1980	4.67
20	standard_low	**0.4848**	8.67
	standard_high	0.3935	6.60
	diverse_low	0.4127	5.48
	diverse_high	0.4502	7.49

In this case, the results do not seem consistent across images and the number of colors. while the diverse operator performs better for higher color counts, while the standard operator seems to perform better for lower color counts. The underperformance of the diverse operator in lower color counts is probably due to the fact that the concept of the two most perceptually similar colors on a palette with very few colors is not so accurate. For example, in a color palette with a green, a yellow, a red, and a blue value, the green and blue may be marked as most similar but may be already sufficiently diverse.

However, we can draw a clear conclusion about mutation rates. When performing a standard mutation, the average best fitness seems to improve when the mutation rate is low. On the other hand, for the diverse operator, the opposite is true. This discrepancy can be caused by the way these operators make use of the mutation rate.

In the standard operator, the mutation rate is considered for each gene, making a high mutation rate dangerous as it increases the probability of random mutation of multiple genes. This may create an individual who is very different from the original, which is not the recommended outcome.

On the other hand, in the diverse operator, this mutation rate is considered only for the closest pair of values, so at maximum, two of the genes of the original individual can change, the objective is to make the palette more diverse instead of randomly changing genes. So, a higher mutation rate increases the probabilities of diversifying the palette through mutation. However, a low value produces very few mutations in comparison to the standard operator.

Additionally, diverse mutation seems to have lower execution times overall, especially when the number of colors in the palette increases.

Figure **??** shows the evolution of the best and average fitness for the 10-color experiments in the `cat.jpg` image. As we saw before, with a lower number of colors, the standard configuration achieves the highest results with a low mutation rate, maintaining a consistent and increasing fitness. However, it is interesting that the diverse mutation with a high mutation rate shows faster early convergence and performs similarly in average fitness. This indicates that introducing diversity can help accelerate the optimization process in the early stages.

As commented before, the standard strategy with a high mutation rate underperforms, probably due to too many randomized values.

This plot illustrates that standard mutation is more stable, while diverse mutation may offer faster but riskier improvements with a small number of colors.

In comparison, Figure **??** presents the evolution of fitness between generations for the 20-color case on the `cat.jpg` image. In this scenario, we can see how the diverse mutation operation gains traction with a higher number of colors, as diversity becomes more and more important. We can observe that this configuration consistently outperforms the others in both average population fitness and best fitness values. In addition, the difference between the average and the best fitness is slim, which is a sign of better convergence.

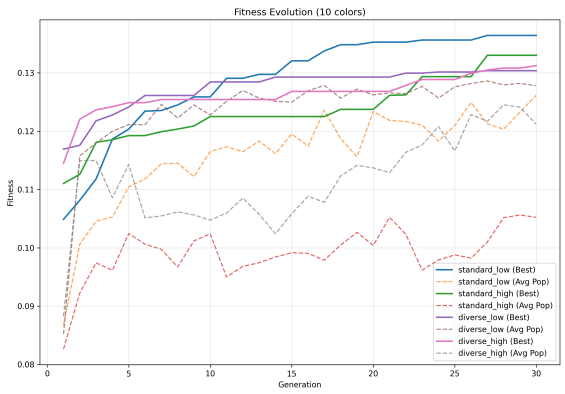


Fig. 9. Fitness evolution over generations for 10-color palettes on `cat.jpg`. Solid lines: best fitness. Dashed lines: average fitness.

The standard low configuration performs well early on but converges to a lower fitness than the diverse high configuration. Once again, the other configurations perform worse than their counterparts with different mutation rates.

Overall, this figure highlights how diverse mutation with a higher rate is the most effective strategy when dealing with complex color distributions.

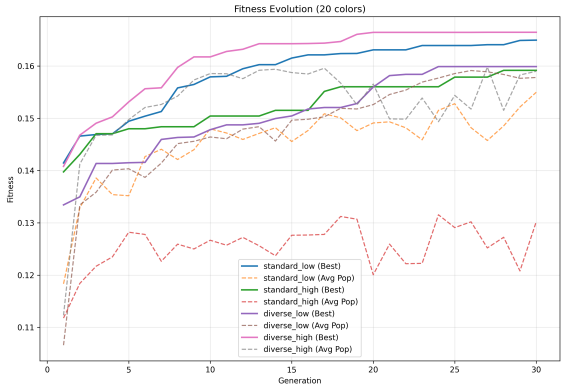


Fig. 10. Fitness evolution over generations for 20-color palettes on `cat.jpg`. Solid lines: best fitness. Dashed lines: average fitness.

Visual Quality: Figure 11 shows a comparison of the output of each of the configurations for the `cat.jpg` image with 20 colors.

With regard to visuals, the diverse operator performs as expected and introduces diversity to the palette. In the rest of the palettes, we can observe some very similar colors, but in this image we can see a palette with more unique colors. This is also reflected in the quality of the final image, as the textures are better preserved. There are no more easily perceived differences.

Discussion: From this comparison,we can conclude the following.

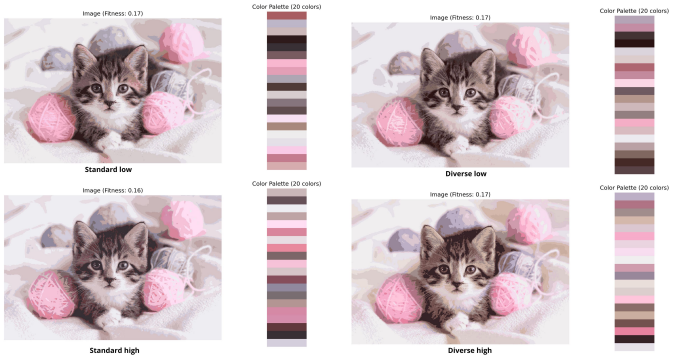


Fig. 11. 20-color palettes on `cat.jpg`

- Regarding **Mutation rates**, diverse mutation performs better with higher mutation rates, while standard mutation performs better with lower rates.
- **Standard mutation** is generally slower than diverse mutation, especially for larger color counts. Additionally, it returns fitter individuals for lower color counts.
- **Diverse mutation** outperforms the standard mutation for higher color counts while maintaining a lower execution time.

D. Effects of the Crossover Operator

In this section, the impact of the use of different crossover operators in the algorithm will be measured. The three crossover operators explained above were considered:

1) `one_point_crossover`: Single-point crossover where a random cut point is selected and the colors at each side of the point are combined to create the child. Maintains larger blocks of genetic material from each parent.
2) `uniform_crossover`: Uniform crossover where each color gene is independently chosen from either parent with equal probability.
3) `closest_pairs_crossover`: Custom crossover method that pairs similar colors from both parents before exchanging genetic material. The objective is to maintain palette diversity and coherence.

Final Fitness and Runtime: Tables XI and XII summarize the average best fitness and execution times for each configuration.

TABLE XI
AVERAGE BEST FITNESS AND EXECUTION TIME FOR CROSSOVER METHODS ON `CAT.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	one_point_crossover	**0.1009**	30.73
	uniform_crossover	0.0971	36.32
	closest_pairs_crossover	0.0976	36.27
10	one_point_crossover	0.1342	70.00
	uniform_crossover	0.1392	84.10
	closest_pairs_crossover	**0.1421**	97.21
20	one_point_crossover	0.1687	110.42
	uniform_crossover	**0.1713**	111.30
	closest_pairs_crossover	0.1703	105.90

TABLE XII
AVERAGE BEST FITNESS AND EXECUTION TIME FOR CROSSOVER METHODS ON `SQUARESIX.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	one_point_crossover	0.0331	4.00
	uniform_crossover	**0.0360**	4.97
	closest_pairs_crossover	0.0338	5.37
10	one_point_crossover	**0.3652**	9.04
	uniform_crossover	0.2774	9.59
	closest_pairs_crossover	0.3521	9.79
20	one_point_crossover	**0.4989**	10.48
	uniform_crossover	0.4709	8.57
	closest_pairs_crossover	0.4770	9.23

Firstly, we can observe that for a very small number of colors like 4, all crossover operators provide practically the same results. This is caused by the fact that the palettes are too small to combine in smart ways. In this color count, we can also observe that closest pairs crossover is the most demanding resource-wise as the distance between each pair of colors must be computed.

In general, the performance of the operators is very varied, with the uniform crossover performing better in the `cat.jpg` image while the one-point crossover performs better in the `squareSix.jpg` image. However, in most test configurations, even though it is not the best, closest pairs crossover is consistently close to the best fitness and does not falter.

To better analyze the difference between the operators, we can observe Figure 12, which shows the evolution of the best and average fitness for the 20-color experiments on the `cat.jpg` image.

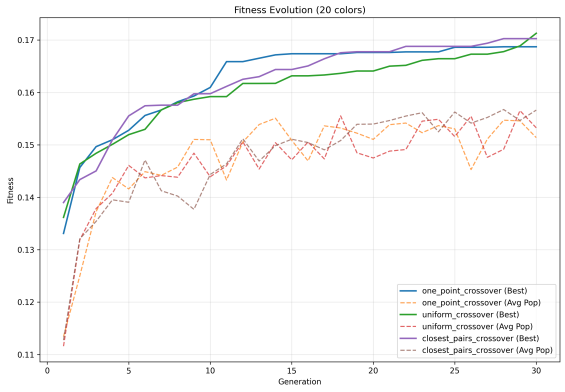


Fig. 12. Fitness evolution over generations for 20-color palettes on `cat.jpg`. Solid lines: best fitness. Dashed lines: average fitness.

As in the table results, all three operators show steady convergence with very similar results.

The one-point crossover exhibits a more stagnant approach, increasing rapidly at the start and then maintaining the same value of fitness while for some time before it slightly increases.

Uniform crossover results in slightly lower fitness through the execution, with a convergence exhibiting more ups and downs. This is likely caused by the disruption of useful color combinations, as the algorithm does not select the colors from each parent with any intent.

Finally, the closest pairs crossover follows closely behind the one-point crossover when it comes to convergence. Additionally, it maintains an increasing best fitness through the generations, as it can be appreciated by the smooth curve and slightly better average population fitness.

Despite these differences, all methods have similar results, with one-point crossover performing marginally better. However, the custom closest-pairs operator offers a good trade-off between diversity and convergence, making it a robust alternative for preserving perceptual color relationships.

Visual Quality: Figure 13 shows a comparison of the output of each of the configurations for the `cat.jpg` image with 20 colors.

Fig. 13. 20-color palettes on `cat.jpg`

Regarding visuals, it is interesting to observe how the closest pairs crossover operator is introducing diversity into the palette. We can see that it includes a wider range of colors (light blue of wool in the background), while in the rest of palettes some very similar colors can be observed (many shades of pink which do not make a difference).

E. Adaptation of mutation rate

In this experiment, we evaluated the impact of adapting the mutation rate during the evolution process. The objective of mutation rate adaptation is to increase the mutation rate to promote diversity when no improvements have been made to the best fitness for a given number of generations. Three configurations were compared:

- `no_adaptation`: Fixed mutation rate across all generations.
- `low_adaptation`: The mutation rate increases slightly when short stagnation is detected.
- `high_adaptation`: The mutation rate increases more aggressively under longer stagnation.

Fitness and Runtime Impact: As shown in Table XIII, the three strategies perform similarly in terms of final fitness.

In most of the cases, the highly adaptable configuration performs better than the configuration without adaptation and executes in less time.

Additionally, in the cases with a higher number of colors, the low-adaptation configuration achieves a similar fitness to the other two while significantly reducing execution time, especially when compared with the version with no adaptation.

The results seem to suggest that modest adaptability can help decrease execution time while maintaining a good fitness value.

TABLE XIII
SUMMARY OF AVERAGE BEST FITNESS AND RUNTIME FOR MUTATION ADAPTATION STRATEGIES IN `CAT.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	no_adaptation	0.1004	91.11
	low_adaptation	**0.1010**	**82.98**
	high_adaptation	0.1006	89.81
10	no_adaptation	**0.1433**	116.98
	low_adaptation	0.1379	**87.25**
	high_adaptation	0.1429	100.56
20	no_adaptation	0.1744	232.13
	low_adaptation	**0.1755**	**118.76**
	high_adaptation	0.1748	119.38

Mutation Rate Adaptation: Figure 14 shows the average value of the mutation rate for each configuration across generations.

The original mutation rate for all three configurations was 0.2, so the configuration without adaptation shows a steady 0.2 value.

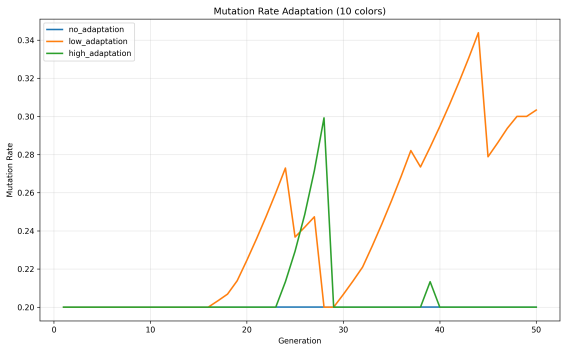


Fig. 14. Mutation rate value over generations for `cat.jpg` with 10 colors.

It is interesting to observe the contrasting behaviors of the two adaptation configurations. Low adaptation exhibits smooth, gradually increasing, and decreasing curves in mutation rate over generations. This shows the incremental nature of the adaptation, where the mutation rate changes slowly in response to stagnation, resulting in more stable fluctuations.

On the other hand, the configuration with a high adaptation shows sharp spikes in the mutation rate, representing the aggressive response to stagnation. While this can help the algorithm escape local optima, it may also lead to excessive randomness if overused, explaining the variability in its convergence behavior.

Discussion: While high adaptation introduces more variance, low adaptation offers the best trade-off. It provides consistent results across different number of colors and improves execution time, introducing enough variability to reduce stagnation without disrupting convergence.

F. Early stopping

In this experiment we evaluated the impact of early stopping due to the stagnation detection mechanism. The objective is to stop the algorithm when the best fitness has not improved in a given number of generations. The objective is to potentially reduce unnecessary computations. Three configurations were considered:

1) `no_early`: Runs for the maximum number of generations (50), regardless of progress.
2) `early_10`: Stops early if no improvement is observed for 10 consecutive generations.
3) `early_20`: Stops early if stagnation persists for 20 generations.

For this test, the number of generations was set to 50.

Results: Table XIV shows a summary of the results of early stopping. For both configurations in which early stopping was implemented, it significantly reduced execution time when triggered without major fitness differentiation. For example, in the 4-color case, early stopping with threshold 10 reduced execution time by more than 50%. A similar case occurs for 10 colors.

In general, a lower threshold is more aggressive, which may lead to more frequent stopping, but also higher risk of stopping before actual convergence. A higher threshold, while triggered less often, offers a safer compromise between speed and quality.

TABLE XIV
SUMMARY OF EARLY STOPPING EFFECTS ON PERFORMANCE AND RUNTIME

Cols	Config	Avg.Fitness	Exec.Time(s)	Early(%)	Avg.Gens
4	no_early	0.1005	43.11	0.0%	50.0
	early_10	0.1000	**19.12**	66.7%	21.7
	early_20	**0.1012**	35.50	66.7%	40.3
10	no_early	**0.1407**	79.65	0.0%	50.0
	early_10	0.1402	**41.66**	66.7%	24.7
	early_20	0.1395	80.93	0.0%	50.0
20	no_early	0.1705	188.17	0.0%	50.0
	early_10	0.1634	**130.49**	33.3%	26.0
	early_20	**0.1741**	250.33	0.0%	50.0

V. HYPERPARAMETER TUNING

A. Objective

In this section, the effect of key hyperparameters that have not been already studied on the performance of the GA will be presented. The objective is to identify the setting that maximizes solution quality (fitness) while minimizing the execution time. The tested hyperparameters include population size, number of generations, crossover rate and elitism.

The following values were independently evaluated:

- **Population Size:** 5, 10, 20
- **Number of Generations:** 15, 30, 50
- **Crossover Rate:** 0.70, 0.85
- **Elitism:** 1, 4

Tests used the `cat.jpg` image and were repeated 3 times for statistical significance. The results below are presented as the average best fitness and execution time of all three runs.

B. Results and Observations

After running the test, the results in Table XV were obtained.

TABLE XV
AVERAGE BEST FITNESS AND EXECUTION TIME FOR HYPERPARAMETER TUNING ON `CAT.JPG`

Colors	Configuration	Avg. Fitness	Exec. Time (s)
4	population_size_5	0.0966	**24.36**
	population_size_10	0.0996	44.65
	population_size_20	**0.1030**	73.54
	gen_15	0.0977	**20.75**
	gen_30	0.0991	33.01
	gen_50	**0.0997**	47.57
	crossover_rate_70	**0.0971**	29.18
	crossover_rate_85	0.0957	29.77
	elite_1	**0.1009**	37.18
	elite_4	0.0980	**24.40**
10	population_size_5	0.1292	**40.27**
	population_size_10	0.1317	97.88
	population_size_20	**0.1375**	218.91
	gen_15	0.1245	**52.33**
	gen_30	0.1368	97.61
	gen_50	**0.1395**	140.15
	crossover_rate_70	**0.1338**	**52.81**
	crossover_rate_85	0.1309	56.23
	elite_1	0.1283	**57.19**
	elite_4	**0.1339**	71.04
20	population_size_5	0.1583	**65.42**
	population_size_10	**0.1671**	108.53
	population_size_20	0.1555	240.93
	gen_15	0.1541	**45.69**
	gen_30	0.1587	143.53
	gen_50	**0.1638**	236.08
	crossover_rate_70	0.1599	**147.48**
	crossover_rate_85	**0.1631**	145.41
	elite_1	0.1623	**170.95**
	elite_4	0.1604	77.01

Population Size: With regards to population size, as expected, increasing it also increases fitness but increases the computation cost. For example, with 10 colors, fitness improves from **0.129** (size 5) to **0.138** (size 20), but execution time more than quintuples.

Although in general the best population size from the ones tested would be 20, due to its increased (more than double in most cases) execution time, the best balance would be maintaining 10.

Number of Generations: Similar to population size, increasing the number of generations typically leads to improved convergence and final fitness, especially when the number of colors increases. For example, with 10 colors, fitness increases from **0.1245** (15 generations) to **0.1395** (50 generations). However, this improvement comes at the cost of significantly longer execution times, which tend to grow exponentially with the number of colors.

Therefore, the optimal number of generations should be carefully selected based on the specific goals of the task and the computational capacity of the device in which the algorithm is run. For better quality use higher generation counts, while for improved performance use lower counts.

Crossover Rate: The results show minimal impact when adjusting the crossover rate. For 20 colors, performance at 0.85 (**0.1631**) was slightly better than at 0.70 (**0.1599**), but the difference was small. This suggests that the crossover rate is not very sensitive in this problem.

Elitism: Elitism plays a stabilizing role in convergence. At lower color counts, elitism of 1 yielded slightly better performance (e.g. 0.1009 for 4 colors), while for 20 colors, elitism of 1 (**0.1623**) narrowly outperformed elitism of 4 (**0.1604**). However, elitism of 4 consistently reduced runtime.

VI. CONLUSION

This paper has presented a Genetic Algorithm-based approach for image color palette reduction, focusing on preserving perceptual similarity between the output and the original image. Through the implementation of domain-specific operators that promoted a diverse palette of colors, such as closest pair crossover and diverse mutation, the algorithm has demonstrated strong performance across a wide range of configurations.

A series of experiments evaluated the impact of several hyperparameters and design choices, including mutation methods, selection methods, crossover operators, population sizes, and early stopping conditions. Most importantly, employing the LAB color space for fitness computation dramatically improved the perceptual quality of the results, and the introduction of caching mechanisms and *NumPy*-based optimizations improved performance and scalability.

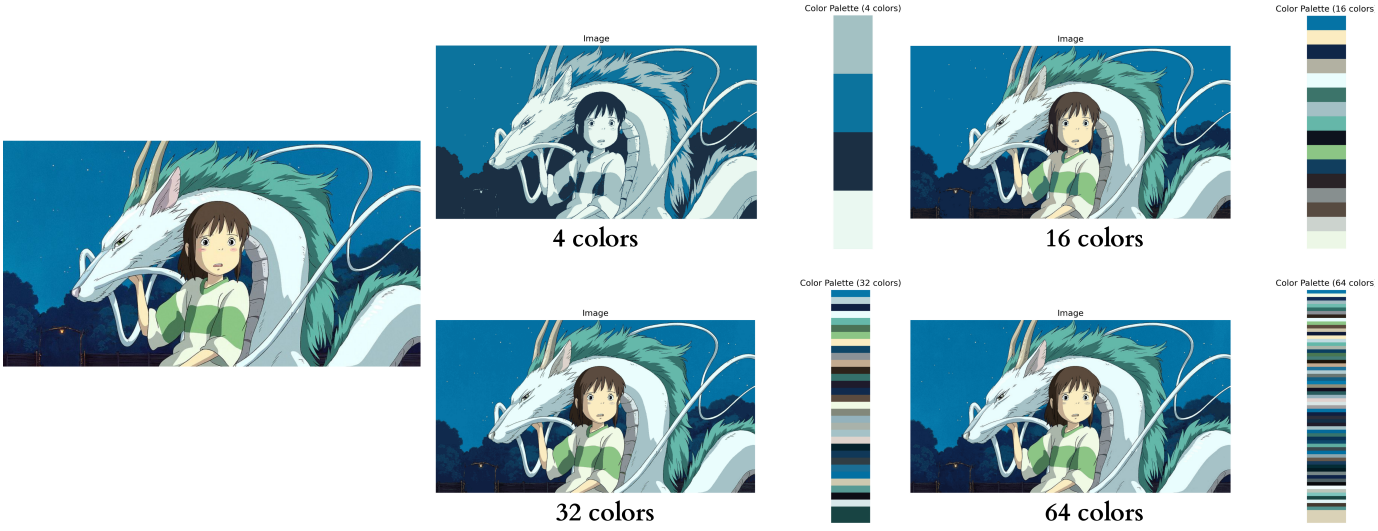
Finally, after taking into account the results of all experiments, a balanced configuration was identified, offering both computational efficiency and solution quality. This configuration is summarized in Table XVI.

TABLE XVI
FINAL GENETIC ALGORITHM CONFIGURATION

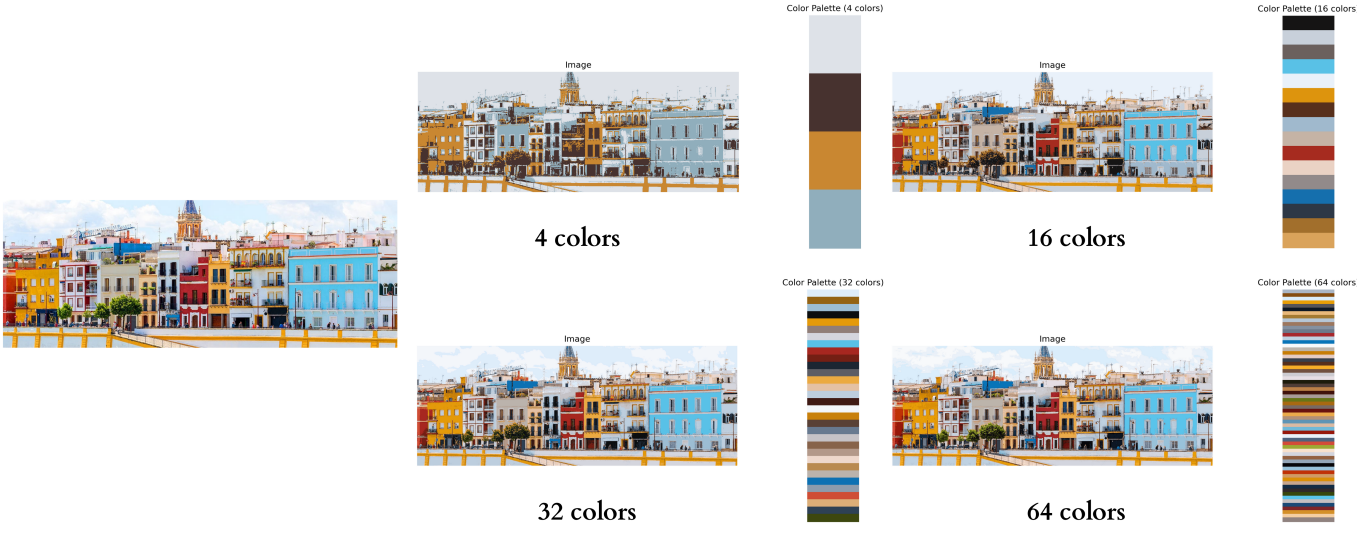
Parameter	Value
`use_caching`	`True`
`restricted`	`True`
`kMeans`	`True`
`population_size`	10
`generations`	50
`mutation_rate`	0.6
`crossover_rate`	0.8
`elitism`	1
`halting_stagnation_threshold`	20
`adaptation_rate`	1.1
`adaptation_threshold`	10
`selection_method`	`tournament`
`tournament_size`	3
`mutate_diverse`	`True`
`crossover_method`	`closest_pairs`

Some example output of the algorithm with multiple images with different characteristics, sizes, and number of colors can be found in the Examples VI. They were not included in the experimentation section of the report due to space constraints.

Examples

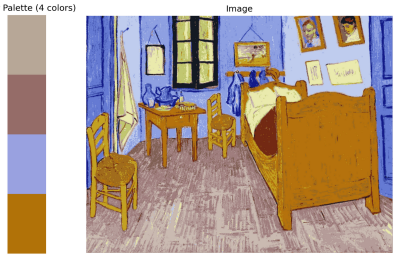


Comparisson for chihiro.jpg



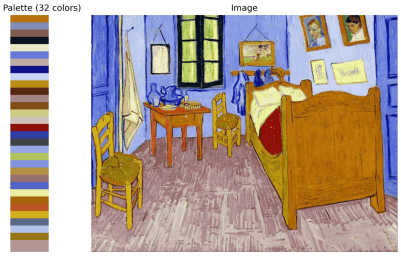
Comparisson for houses.jpg

4 colors



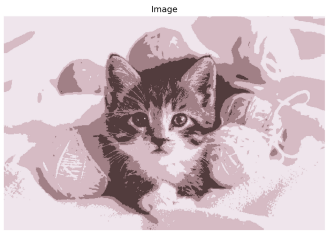
16 colors

32 colors



64 colors

Comparisson for cuadro.jpg



4 colors

16 colors

32 colors

64 colors

Comparisson for cat.jpg

REFERENCES

[1] Celebi, M. Emre. (2011). Improving the Performance of K-means for Color Quantization. Image and Vision Computing - IVC. 29. 10.1016/j.imavis.2010.10.002.

[2] Sivanandam, S. N. & Deepa, S. N. (2008). Introduction to Genetic Algorithms. Springer.

[3] Sangwan, Shabnam. (2018). Literature Review on Genetic Algorithm. International Journal of Research. 5. 1142.

[4] CIE, *Colorimetry – Part 4: CIE 1976 L*a*b* Colour Space*, CIE Publication 15.3, Commission Internationale de l'Eclairage, 2004.