Image Compression

*

Jakob Moritz
Universidad de Sevilla
Ruprecht-Karls-Universität Heidelberg
Heidelberg, Germany
moritz.jakob02@stud.uni-heidelberg.de

***Abstract*—This project explores the use of genetic algorithms for optimizing color palettes used in image compression. By evolving palettes through crossover, mutation, and selection strategies, the system aims to minimize visual differences as measured by metrics such as PSNR. The modular design enables easy experimentation with different genetic operators, while results demonstrate effective reduction in color complexity with preserved image quality.**

I. Introduction

Digital images are found in many domains such as entertainment, medical imaging, surveillance, and computer vision. Each image typically contains millions of colors, with each pixel represented in the RGB (Red, Green, Blue) space using three values ranging from 0 to 255. This creates millions of possible color combinations. Such a high amount of color data presents challenges for tasks like compression, classification and transmission. More compact and simplified representations are therefore advantageous. Image compression aims to minimize the number of distinct colors in an image while retaining as much visual quality as possible. This is especially relevant where storage constraints or rendering demand a limited color set. Reducing the color space also simplifies analysis and classification tasks lowering the data dimensionality. Among different algorithms which are enable for this task, Genetic Algorithms (GA) have emerged as a powerful tool due to their adaptive, population-based search. They work by evolving a population of candidate solutions over generations, using genetic operations such as selection, crossover, and mutation. This population-based approach allows GAs to explore a wide search space and escape local optima which is making them particularly suitable for optimization problems.

Recent development in this area have significantly impacted the field of image compression. Genetic Algorithms have proven to be powerful for solving complex search problems. A broad review by Katoch et al. [1] outlines the strengths of GAs in diverse domains, emphasizing their adaptability and capability to escape local minima, making them ideal for image-related tasks that involve nonlinear search spaces.

In the context of assessing the visual impact of image transformations, quality metrics play a crucial role. Among these, Peak Signal-to-Noise Ratio (PSNR) and Mean Squared Error (MSE) are mostly used. They offer simple evaluations of distortion on a pixel level. Although not always aligned with human perception, PSNR remains a standard due to its interpretability and computational efficiency. Structural Similarity Index (SSIM) has also emerged as a more perceptually aligned metric [2].

Built on these foundations, multiple studies have demonstrated the use of GAs for image compression. Harman and Koçyiğit [3] investigated different selection strategies within GAs. They used roulette wheel, elitist and a novel pool-based method for optimizing codebooks in vector quantization. Their experiments showed that pool-based selection consistently achieved superior PSNR and MSE results, highlighting the importance of preserving diversity and elite solutions in the evolutionary process.

Sun et al. [4] proposed a hybrid approach combining Principal Component Analysis (PCA) with GA to improve the efficiency of codebook generation in VQ. By first reducing the dimensionality of training vectors via PCA and then applying the GA to optimize partitioning along the principal axis, their method outperformed the classical LBG algorithm in both reconstruction quality and computational time, especially for smaller codebooks.

Another approach, Mitra et al. [5] applied GA to fractal image compression, leveraging the algorithm's global search capability to identify optimal transformations for encoding image blocks. By integrating a block classification step and using an elitist GA model, their method reduced the search space while maintaining competitive PSNR values. This approach showcased GAs potential in alternative compression schemes.

Together, these studies demonstrate the viability of Genetic Algorithms as an effective tool for optimizing color palettes and compression in image processing.

II. System Architecture and Design Methodology

A. Overview

This section provides an overview of the architectural structure, the design decisions undertaken during implementation and the methodology followed during the development of the project.

B. Methodology

The implementation was followed by an incremental approach, beginning with basic image manipulation, which was also followed by the creation of individuals and color palettes to test an initial version of the genetic algorithm. The modules were subsequently refined by incorporating a new fitness function and various crossovers and mutation operators. Finally, the algorithm was fine-tuned and visualization tools were added to enable a more effective evaluation of the results.

The project was developed using NumPy, PIL (Python Imaging Library), and scikit-learn (sklearn) as the core libraries. The project has been designed with a modular architecture to ensure flexibility, ease of testing and future extensibility. Each component is responsible for a distinct functionality within the overall image compression pipeline.

C. Image I/O Module

The image input/output module handles the loading and preprocessing of input images. It converts images into RGB format, saves the final compressed image and creates a visual representation of the final color palette. It supports the reading of image files by converting them into RGB arrays using the PIL (Pillow) library. Saving image arrays back to disk in standard formats to produce the compressed output. Another design decision was the inclusion of visual outputs to display each color of the final palette as a grid of color squares, each labeled with its RGB values. An optional header, that includes metadata, can also be added. This decision enhanced the interpretability of results and allowed for an intuitive assessment of the final color palette.

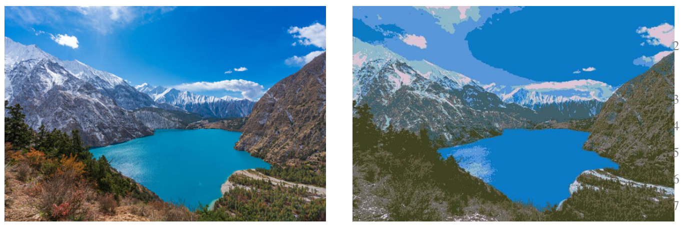

(a) Original (311,120 colors) (b) 8 Colors

Fig. 1: Visual comparison of image compression results with varying numbers of colors.

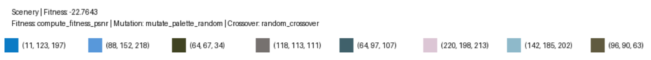


Fig. 2: Color Palette Example

D. Chromosome Representation

Central to each genetic algorithm is the representation of its chromosomes. Therefore, the individual class was defined, it encapsulates a color palette and the corresponding fitness value. Hereby the decision was made to represent each chromosome as a color palette, where each gene corresponds to one RGB value. This structure allows a straightforward application of genetic operations, such as mutation and crossover, directly on the color values and aligns with the goal of reducing the color palette.

E. Fitness Evaluation

The decision on how to represent individuals within the genetic algorithm is closely associated with the choice of an appropriate fitness function. Since the goal is to assess how well a given color palette can approximate the original image, selecting a meaningful image similarity metric is essential. Hence, the fitness evaluation component utilizes image quality metrics to quantify the similarity between the original image and its reconstruction, which is generated by applying a color palette to the image. This is done by replacing each pixel in the original image with the closest color from the palette. The closest color is determined using the Euclidean distance in RGB space. This ensures that each pixel is mapped to the most visually similar color of the palette. The result is a reconstructed image composed exclusively of colors from the current palette. Initially, the Mean Squared Error (MSE) was implemented as the fitness measure due to its straightforward computation and direct interpretation. However, MSE values lack an intuitive scale and have no clear upper bound, making it difficult to establish a stopping criterion, besides defining a number of max generations. To address this, the Peak Signal-to-Noise Ratio (PSNR) was added. PSNR offers a logarithmic scale, providing a more interpretable metric for fitness evaluation. PSNR values above approximately 50 dB roughly, generally indicate almost perfect reconstruction, because of this the this threshold is used as a second stopping criterion.

```
function computePSNR(originalImage,
    reconstructedImage):
    mse = meanSquaredError(originalImage,
        reconstructedImage)
    if mse == 0:
        return infinity  // Identical image
    maxPixelValue = 255.0
    psnr = 20*log10(maxPixelValue/sqrt(mse))
    return psnr
```

Although the Structural Similarity Index Measure (SSIM) was considered, PSNR was chosen due to its simpler computation and suitability for setting a clear stopping criterion.

F. Reproducibility

Since testing and evaluating algorithmic performance is a critical aspect of this project, it was necessary to establish a mechanism for ensuring reproducibility. Therefore ,consistent experimental conditions are achieved by the color palette module. It supports the generation of random color palettes but also allows the use of fixed seeds, which enables the exact same initial conditions to be recreated across multiple runs.

G. Genetic Operators

To drive the search and exploration of the solution space Genetic algorithms heavily rely on the design of crossover and mutation operators. Therefore multiple crossover and mutation operators were implemented and tested.

The project is driven by the two operator modules: crossover and mutation. These modules combine and alter color palettes to produce better generations. This promotes exploration of the solution space and convergence toward optimal palettes.

H. Crossover Operators

The crossover module supports distinct strategies for recombining two parent palettes into a new palette. Since each palette is modeled as a fixed-length array of RGB color vectors, the crossover is applied at the level of the individual color or their components.

The Uniform Crossover strategy generates a random boolean mask to determine, for each color in the palette, whether the corresponding color in the offspring will be inherited from the first or second parent. This method promotes genetic diversity by randomly mixing parent genes on a per-color basis.

The Segmented Crossover selects a random point along the palette length and constructs the new chromosome by concatenating the prefix from one parent with the suffix from the other at the selected point.

The BLX-alpha (Blend Crossover) introduces controlled variation by choosing colors from an extended range surrounding the parent values. For each color component RGB it looks at the values from both parents and then chooses a new value from a slightly wider range around them. Therefore this method supports both exploitation and extrapolation.

To encourage variation in recombination behavior, a Random Crossover Selector is implemented. It randomly selects one of the before-mentioned crossover operators at each recombination event, thereby introducing an additional level of variation to the evolutionary process.

Additionally, an Average Crossover, which produces offspring by averaging the corresponding RGB values of the parents was also implemented. However, this method was not integrated into the main algorithm due to its limited capacity to introduce variation.

I. Mutation Operators

The mutation module introduces random changes to the individual color palettes, which enhances the algorithm to explore new possibilities and avoids getting stuck in suboptimal solutions. As a result, diversity in the population will also be maintained over time.

The Gaussian Mutation operator slightly changes the RGB values of one or more randomly selected colors by adding noise. This mutation allows for fine modifications while preserving the overall color structure of the palette. The number of colors to mutate is determined randomly for each mutation event, constrained by a predefined maximum. This approach is beneficial because it introduces small changes without drastically changing the overall look of the palette.

The Component-Wise Mutation strategy modifies a single color channel of one randomly selected color within the palette. These small, local changes help to fine-tune individual color properties.

In contrast, the Random Resetting Mutation replaces an entire color in the palette with a new color sampled uniformly at random from the RGB space. This operation introduces high variation and is particularly beneficial for adding new, diverse colors to the population. This prevents to get stuck in local optima.

To encourage variation in mutation behavior, a Random Mutation Selector is implemented. It randomly chooses from the available mutation operators for each mutation event, enabling a balance between small refinements and larger changes across generations.

As with the crossover module, some mutation functions were tested but not included in the final setup. For example, a Color Swap Mutation that exchanges the positions of two randomly selected colors was implemented. Since this preserves the set of colors in the palette it was not very beneficial and therefore is excluded from the final implementation.

The combination of diverse crossover and mutation strategies ensures a robust and flexible framework for exploring the search space.

J. Genetic Algorithm Control

This flexibility is embodied in the genetic algorithm module, which serves as the core component of implementing the genetic algorithm. It manages the entire evolutionary process, from population initialization to selection, crossover, mutation, fitness evaluation, and stopping criteria. It allows for extensive customization of parameters and operators to suit different needs. Key parameters include the number of colors in the palette, population size, number of generations, and probabilities for crossover and mutation. Additionally, users can specify the fitness function, crossover and mutation operators. As mentioned an optional fixed palette can also be provided to initialize the population with predefined colors. Selection is performed using tournament selection: a small random subset of the population is chosen, and the fittest individual from this group is selected as a parent. This approach maintains a balance between exploring new solutions and exploiting the best ones found so far. In addition, also elitism is added by carrying over the best individual found in the current generation to the next generation. This guarantees that the best solution is never lost as the algorithm progresses. The fitness values evolve throughout the evolutionary run and to evaluate the progress the best fitness values for each generation are printed. After completing the iterations or reaching the predefined target fitness value, which can be provided as a user-defined parameter, the function returns the compressed image generated using the best palette and also produces an image visualizing the final palette, which includes the RGB color values along with metadata describing the final fitness score and the names of the selected crossover and mutation operators, as well as the used fitness function. It is important to note that the early stopping criterion based on a fitness threshold is only applied when PSNR is used as the fitness function. This choice was made to ensure consistent behavior, as PSNR provides a standardized scale suitable for setting meaningful threshold values.

K. Conclusion

Overall, this implementation offers a clear and customizable framework for evolving color palettes with genetic algorithms, allowing detailed control over each stage of the process and providing informative outputs for analysis and visualization.

III. Experiments

To evaluate the effectiveness of the genetic algorithm in compressing images through color palette reduction, a series of systematic tests were conducted. The experiments were designed to analyze how various parameters, such as population size, maximum number of generations, number of colors, mutation and crossover probabilities and how different genetic operators affect both convergence behavior and output quality. Images of different types and complexities were used to assess the algorithm's generalization ability. Key performance metrics include best and average fitness scores, runtime, and visual quality of the resulting compressed images. To account for randomness in the algorithm, each configuration was run multiple times, and aggregated results were used for comparison.

To provide a consistent foundation for comparative evaluation, a baseline configuration was established using average or commonly used values across all genetic algorithm parameters. This baseline setup included the following settings: num_colors = 16, max_generations = 50, population_size = 20, crossover_prob = 0.85, mutation_prob = 0.2, the input image set to bird, and the compute_fitness_psnr function as the fitness evaluator. The operators used were random_crossover and mutate_palette_random, with a target_fitness threshold of 50 and a fixed random seed of 45 to ensure reproducibility. All baseline results were averaged over five runs. After exploring different crossover and mutation operators, it was observed that increasing the population size significantly improved convergence and fitness scores. Consequently, all further testing was conducted with population size = 50

Crossover	Best Fitness	Avg Fitness (mean)	Runtime (mean in s)
Random	-25.4811	**-24.8801*	**79.87**
Segmented	-25.2294	-23.9844	81.13
Uniform	-25.0446	-23.9047	80.48
Blend	**-25.6651**	-24.3639	79.97

TABLE I: Crossover Operators in Comparison

To assess the impact of different crossover strategies on convergence and performance, four crossover methods were evaluated. The Blend crossover achieved the best overall fitness (-25.6651), suggesting it was most effective at refining palettes. However, the Random crossover led to the highest average fitness across generations (-24.8801) and the lowest runtime (79.87s), indicating more consistent performance with faster convergence. While Segmented and Uniform performed slightly worse in both best and average fitness, their runtime remained comparable.

Mutation	Best Fitness	Avg Fitness (mean)	Runtime (mean in s)
Random	**-25.4811**	**-24.8801*	79.87
Component	-24.5287	-23.9681	80.33
Gaussian	-25.3553	-24.8094	**79.41**
Reset	-25.3256	-24.3674	79.86

TABLE II: Mutation Operators in Comparison

Different mutation strategies were tested to examine their influence on fitness progression and execution time. The Random mutation method achieved both the best overall fitness (-25.4811) and the highest average fitness (-24.8801), indicating strong performance in promoting diversity while preserving quality solutions. The Gaussian mutation performed nearly as well in terms of fitness and had the fastest runtime (79.41s), making it a good balance between quality and efficiency. Component and Reset mutations lagged slightly in fitness outcomes, suggesting they may introduce less effective variability in the population. Overall, Random mutation proved to be the most robust option in this comparison.

As observed in both crossover and mutation experiments, the Random selection method consistently achieved the best or near-best results in terms of both best and average fitness. This highlights the importance of variability and operator diversity in evolutionary processes. By randomly selecting among multiple strategies, the algorithm maintains a better balance between exploration and exploitation, leading to more robust convergence and higher-quality solutions across runs.

Population Size	Best Fitness	Avg Fitness	Runtime (mean in s)
20	-25.4811	-24.8801	79.87
50	-26.6189	-26.1858	197.87
150	-27.5941	-27.1532	631.01

TABLE III: Population Sizes in Comparison

Increasing the population size significantly improved both best and average fitness values. While a population of 150 achieved the best overall fitness (-27.5941) and highest average fitness (-27.1532), the improvement over a population size of 50 (-26.6189 best fitness) was relatively small. However, the runtime increased dramatically—from around 198 seconds at 50 individuals to over 630 seconds at 150. Given this small gain in fitness compared to the large increase in computation time, the population size was set to 50 for the following experiments to balance performance and efficiency.

Crossover Prob	Best Fitness	Avg Fitness	Avg Runtime (in s)
0.5	-26.0929	-25.8333	197.87
0.7	-26.5246	-25.9768	200.03
0.85	**-26.6189**	**-26.1858**	197.87
0.95	-26.3988	-26.0048	198.42

TABLE IV: Crossover Probability in Comparison

The impact of varying crossover probabilities on algorithm performance was evaluated across four levels. The highest best and average fitness values were achieved at a crossover probability of 0.85 (-26.6189 best fitness and -26.1858* average), indicating this value offers the most effective balance between preserving existing solutions and generating new ones. While lower probabilities like 0.5 resulted in slightly lower fitness, the differences were relatively small. Additionally, the runtime remained nearly constant across all tested probabilities, suggesting that tuning crossover probability primarily affects solution quality without significant impact on computational cost. Based on these results, a crossover probability of 0.85 is recommended for optimal performance.

Mutation Prob	Best Fitness	Avg Fitness	Avg Runtime (in s)
0.1	-25.7895	-25.4035	203.86
0.2	**-26.6189**	**-26.1859**	**197.87**
0.3	-26.4517	-26,1610	198.31

TABLE V: Mutation Probability in Comparison

Testing different mutation probabilities showed that a value of 0.2 yielded the best results, achieving the highest best fitness (-26.6189) and average fitness (-26.1859), while also maintaining the lowest runtime (197.87 seconds). Mutation probabilities lower (0.1) or higher (0.3) than this value resulted in slightly worse fitness scores. These findings suggest that a mutation rate of 0.2 balances exploration and exploitation most effectively.

Max Generations	Best Fitness	Avg Fitness	Avg Runtime (in s)
50	-26.6189	-26.1858	197.87
150	-27.2819	-26.9040	240.21
300	-27.6062	-27.4928	518.10

TABLE VI: Max Generations in Comparison

Increasing the number of generations consistently improved both best and average fitness values. The highest fitness was achieved at 300 generations (-27.6062 best fitness), showing that more iterations help refine solutions. However, the improvement from 50 to 150 to 300 generations is relatively small, especially when weighed against the substantial increase in runtime—from about 198 seconds at 50 generations to over 518 seconds at 300. For a good balance between solution quality and efficiency, 150 generations is recommended, offering most of the fitness benefits without excessive runtime.

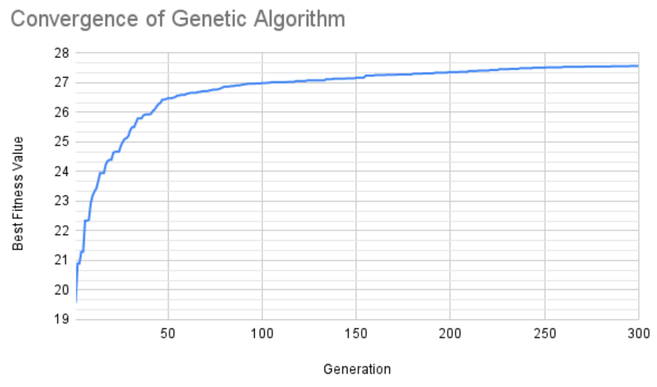


Fig. 3: Average Convergence of the Genetic Algorithm over 300 generations

This images demonstrates how the fitness value evolves over 300 generations. Initially, from Generation 1 to about Generation 50 the best fitness improves rapidly. This demonstrates that the algorithm quickly finds significantly better solutions early on.

After this fast initial progress, the improvement rate slows down. Between Generations 50 and 300, the fitness values continue to decrease but at a much slower pace, finally reaching around -27.5. This suggests that the algorithm is refining the solution with smaller incremental gains because it approaches an optimal or near optimal solution.

This pattern shows how on the fast initial improvement a fine-tuning phase with slow improvements follows. This is typical for genetic algorithms, indicating effective exploration early on and exploitation of promising solutions in later stages.

Number of Colors	Best Fitness	Avg Runtime (in s)
8	-23.9178	150,39
16	-27,3152	200.06
64	-29.5308	629.92

TABLE VII: Number of Colors in Comparison (Bird Picture)

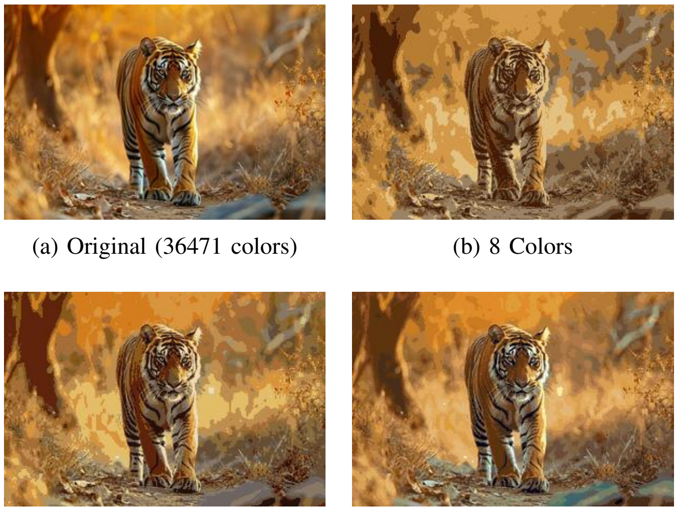
The table shows the impact of the number of colors on the best fitness achieved by the genetic algorithm and the average runtime. As the number of colors increases from 8 to 64, the best fitness value improves significantly, indicating better compression quality with more colors available. However, this improvement comes at the cost of increased computational time, with runtime rising from approximately 150 seconds for 8 colors to nearly 630 seconds for 64 colors. This trade-off highlights that higher color resolution leads to better results but requires substantially more processing time.

Number of Colors of Original Image	Best Fitness	Avg Runtime (in s)
36471	-26.1723	10.12
122386	-25,3152	200.06
311120	-23.2304	390.59

TABLE VIII: Number of Colors of Orignal Image in Comparison (Bird Picture)

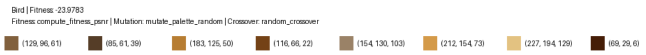
The table shows the effect of the original images color complexity on the compression performance when using 16 colors for compression. As the number of original colors increases from 36,471 to 311,120 the best fitness value worsens. This is expected since compressing images with more original colors into only 16 colors results in a greater loss of detail. Additionally, the average runtime increases with the number of original colors, indicating that more complex images require longer processing times.

Lastly, we aim to visually compare the results of the algorithm to better understand the impact of compression on image quality.



(a) Original (36471 colors) (b) 8 Colors (c) 16 Colors (d) 64 Colors

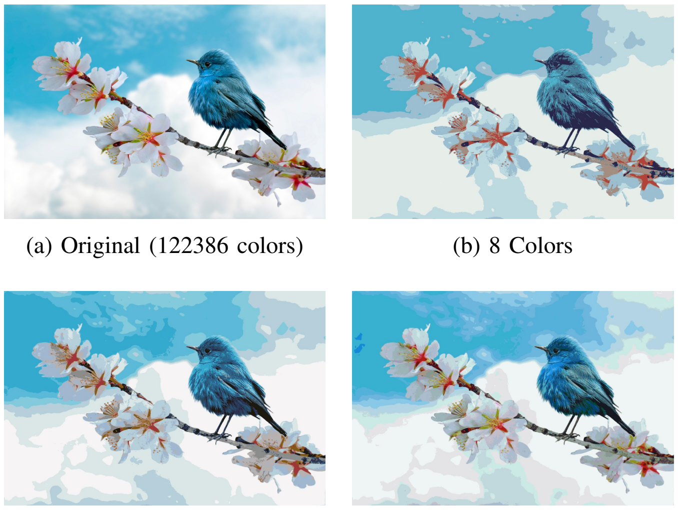
Fig. 4: Visual comparison of image compression results with varying numbers of colors.



Bird | Fitness: -23.9783
Fitness: compute_fitness_psnr | Mutation: mutate_palette_random | Crossover: random_crossover

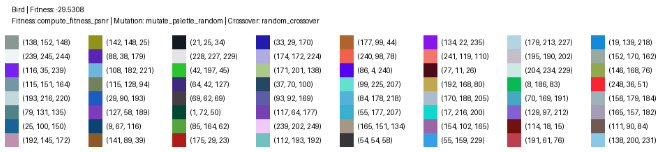
■ (129, 96, 61)	■ (85, 61, 39)	■ (183, 125, 50)	■ (116, 66, 22)	■ (154, 130, 103)	■ (212, 154, 73)	■ (227, 194, 129)	■ (69, 29, 6)

Fig. 5: Color Palette for Tiger Image (8 Colors)



(a) Original (122386 colors) (b) 8 Colors (c) 16 Colors (d) 64 Colors

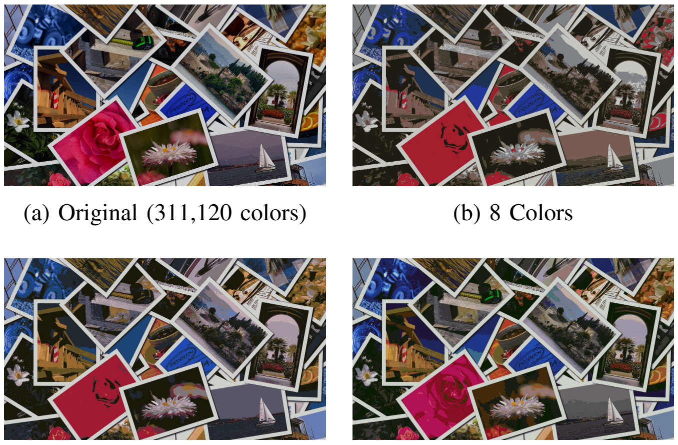
Fig. 6: Visual comparison of image compression results with varying numbers of colors.



Bird | Fitness: -29.5308
Fitness: compute_fitness_psnr | Mutation: mutate_palette_random | Crossover: random_crossover

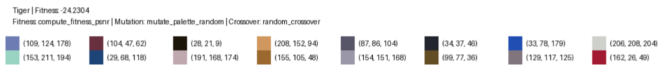
■ (138, 152, 148)	■ (142, 148, 25)	■ (21, 25, 34)	■ (33, 29, 170)	■ (177, 99, 44)	■ (134, 22, 235)	■ (179, 213, 227)	■ (19, 139, 218)
■ (239, 245, 244)	■ (88, 38, 179)	■ (228, 227, 229)	■ (174, 172, 224)	■ (240, 98, 78)	■ (241, 119, 110)	■ (195, 190, 202)	■ (152, 170, 162)
■ (116, 35, 239)	■ (108, 182, 221)	■ (42, 197, 45)	■ (171, 201, 138)	■ (86, 4, 240)	■ (77, 11, 26)	■ (204, 234, 229)	■ (146, 168, 76)
■ (115, 151, 164)	■ (115, 128, 94)	■ (64, 42, 127)	■ (37, 70, 100)	■ (99, 225, 207)	■ (192, 168, 80)	■ (8, 186, 83)	■ (248, 36, 51)
■ (193, 216, 220)	■ (29, 90, 193)	■ (69, 62, 69)	■ (93, 92, 169)	■ (84, 178, 218)	■ (170, 188, 205)	■ (70, 169, 191)	■ (156, 179, 184)
■ (79, 131, 135)	■ (127, 58, 189)	■ (1, 72, 50)	■ (117, 64, 177)	■ (55, 177, 207)	■ (17, 216, 200)	■ (129, 97, 212)	■ (165, 157, 182)
■ (25, 100, 150)	■ (9, 67, 116)	■ (85, 164, 62)	■ (239, 202, 249)	■ (165, 151, 134)	■ (154, 102, 165)	■ (114, 18, 15)	■ (111, 90, 84)
■ (192, 145, 172)	■ (141, 89, 39)	■ (175, 29, 23)	■ (112, 193, 192)	■ (54, 54, 58)	■ (55, 159, 229)	■ (191, 61, 76)	■ (138, 200, 231)

Fig. 7: Color Palette for Bird Image (64 Colors)



(a) Original (311,120 colors) (b) 8 Colors (c) 16 Colors (d) 64 Colors

Fig. 8: Visual comparison of image compression results with varying numbers of colors.



Tiger | Fitness: -24.2304
Fitness: compute_fitness_psnr | Mutation: mutate_palette_random | Crossover: random_crossover

■ (109, 124, 178)	■ (104, 47, 62)	■ (28, 21, 9)	■ (208, 152, 94)	■ (87, 86, 104)	■ (34, 37, 46)	■ (33, 78, 179)	■ (206, 208, 204)
■ (153, 211, 194)	■ (29, 68, 118)	■ (191, 168, 174)	■ (155, 105, 48)	■ (154, 151, 168)	■ (99, 77, 36)	■ (129, 117, 125)	■ (162, 26, 49)

Fig. 9: Color Palette for Pictures Image (16 Colors)

Conclusion and Further Development

The experiments demonstrate that the genetic algorithm performs effectively for the given task, showing consistent convergence and improvement over generations. The results indicate that various combinations of genetic operators (such as crossover and mutation methods) can achieve strong performance, likely due to the diversity they introduce into the population. This is essential for exploring the solution space and avoiding premature convergence. Future improvements could include adaptive operator selection, where the algorithm dynamically adjusts the use of different operators based on their performance during runtime. Additionally, testing more selection strategies or hybrid approaches with local search methods could enhance convergence speed and final solution quality. Another approach to consider would be to implement Structural Similarity Index (SSIM) as a fitness function. Unlike pixel-based differences, SSIM considers perceived changes in structural information. This could potentially lead to results that appear more similar to the original image. Also the use of the RGB color space could limit perceptual quality, as it does not reflect how humans perceive color differences. Switching to perceptually uniform color spaces such as LAB or HSV could result in better results that are visually more similar to the original image. This represents different ideas further investigations and development.

References

[1] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, pp. 8091-8126, 2021.

[2] U. Sara, M. Akter, and M. S. Uddin, "Image quality assessment through FSIM, SSIM, MSE and PSNR—A comparative study," *Journal of Computer and Communications*, vol. 7, pp. 8-18, 2019.

[3] F. Harman and Y. Koçyiğit, “A new approach to genetic algorithm in image compression,” in *Proc. 10th International Conference on Electrical and Electronics Engineering (ELECO)*, Bursa, Türkiye, Nov. 29–Dec. 2, 2017, pp. 894–898, Jan. 2018.

[4] H. Sun, K.-Y. Lam, S.-L. Chung, W. Dong, M. Gu, and J. Sun, “Efficient vector quantization using genetic algorithm,” *Neural Computing Applications*, vol. 14, pp. 203-211, 2005.

[5] S. K. Mitra, C. A. Murthy, and M. K. Kundu, “Technique for fractal image compression using genetic algorithm,” *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 586-593, Apr. 1998.

[6] Pillow Contributors, “Pillow (PIL Fork) Documentation,” [Online]. Available: https://pillow.readthedocs.io/en/stable/. [Accessed: May 14, 2025].

[7] scikit-learn Developers, “sklearn.modules.generated,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/. [Accessed: May 16, 2025].

[8] Stack Overflow contributors, “Mean squared error in numpy,” [Online]. Available: https://stackoverflow.com/questions/16774849/mean-squared-error-in-numpy. [Accessed: May 16, 2025].

[9] Tutorialspoint, “Finding the Peak Signal to Noise Ratio (PSNR) using Python,” [Online]. Available: https://www.tutorialspoint.com/finding-the-peak-signal-to-noise-ratio-psnr-using-python. [Accessed: May 19, 2025].

[10] PyImageSearch, “Python: Compare two images,” [Online]. Available: https://pyimagesearch.com/2014/09/15/python-compare-two-images/. [Accessed: May 19, 2025].

[11] Naukri Code360, “Mutation in Genetic Algorithm,” [Online]. Available: https://www.naukri.com/code360/library/mutation-in-genetic-algorithm. [Accessed: May 19, 2025].

[12] Stack Overflow contributors, “BLX-alpha crossover - what approach is the right one?,” [Online]. Available: https://stackoverflow.com/questions/38952879/blx-alpha-crossover-what-approach-is-the-right-one. [Accessed: May 20, 2025].

[13] J. Sudhoff, “Lecture 04: Genetic Algorithms,” Purdue University, [Online]. Available: https://engineering.purdue.edu/ sudhoff/ee630/Lecture04.pdf. [Accessed: May 19, 2025].

[14] NumPy Developers, “NumPy Documentation,” [Online]. Available: https://numpy.org/devdocs/index.html. [Accessed: May 14, 2025].