DRAWING PICTURES WITH PATTERNS USING GENETIC ALGORITHM

ÖMER ERCE BERBER

INTRODUCTION

In this study, it is aimed to best represent five different binary images of 24x24 size using 7 different binary patterns of 3x3 size. Each 24x24 image will be divided into 3x3 blocks and the most suitable pattern will be assigned to each block. Thus, a loss value will be created by calculating the difference between the new image obtained and the original image. The total loss value calculated for the five images will determine the success of the given solution. This optimization problem will be solved using genetic algorithm and different hyperparameters (population size, mutation rate, production strategy of the new generation) will be examined and the parameter combinations that give the best result will be analyzed. In the study, how the selection, crossover and mutation stages, which are the evolutionary processes of the genetic algorithm, optimize the loss values will be discussed in detail.

EXPLANATION OF THE PROGRAM

Generating Binary Images and the load_images function:

Binary images are in the form of circles, squares, crosses, triangles and hearts. These images are pre-generated using mathematical functions of those shapes in generate_binary_images.py. Each is 24x24 in size, and is written to files with .npy and .txt extensions for later use. The load_images function in the main program reads these binary images from the files and saves them in a list.

create_individual function:

In this function, 7 3x3 patterns are randomly generated and returned.

initialize_population function:

This function generates individuals (7 3x3 patterns) as many as the pop_size parameter, that is, the specified population size, using the create_individual function, stores it in a list and returns it.

find_best_match function:

This function takes two parameters, block and pattern. Block: any of the 3x3 blocks of a binary image, patterns: any element of the population, i.e. 7 3x3 patterns. This function returns the most suitable pattern among the 7 patterns in patterns for the block (the one with the least difference between the block and the pattern).

calculate_fitness function:

This function takes two parameters, individual and images. individual: an individual in the population, i.e. 7 3x3 patterns. images: a list of 5 binary images. The most suitable pattern is placed in each 3x3 block of each binary image using the find_best_match function, the difference between the new image and the original image is calculated and returned.

selection function:

This function takes two parameters, population and images. population: A list of 7 3x3 patterns in pop_size. images: A list of 5 binary images. Two sample individuals are taken from population, the fitness value of these individuals is calculated with the calculate_fitness function, and the individual with the lower fitness value, i.e. the more fit individual, is returned.

crossover function:

This function takes two parameters, parent1 and parent2. Both of these parameters are individuals from the population (i.e. 7 3x3 patterns) and a new child is created and returned by crossing the two. A random number is selected between 1-6 and the part to the left of this number is taken from parent1, the part to the right is taken from parent2 and these parts are combined to produce the child. (For example, let the random number be 3, the child's 0, 1 and 2nd indices are taken from parent1, and the 3, 4, 5 and 6th indices are taken from parent2 and combined.)

mutate function:

This function takes individual and mutation_rate as parameters. Similar to other functions, individual is a 7x3x3 individual from the population. Mutation_rate is the probability of mutation. When obtaining outputs, mutation_rate will vary as 0.01 (low), 0.05 (medium) and 0.1 (high). In the main loop of the function, 7 patterns in individual are returned, if a random number between 0-1 comes smaller than mutation_rate, the mutation process starts. Two numbers x and y are randomly selected as 0, 1, 2. Because the blocks are 3x3. The x value represents the row, the y value represents the column. Then the x and y indexed points of the relevant pattern are inverted. (If 0, it is converted to 1, if 1, it is converted to 0.) The individual that has undergone mutation or not is returned.

genetic_algorithm function:

The main function where everything in the program is used and implemented.

Parameters:

- **generations**: This parameter determines how many generations the population will evolve over in the genetic algorithm. In each generation, individuals are subjected to crossover and mutation processes, the best solutions are selected, and the population is updated by producing new individuals. A higher generations value increases the probability that the algorithm will work for a longer period of time and find a better solution. However, a very high value can increase the computational cost and lead to problems such as overfitting. Therefore, determining an appropriate generations value is important for the efficiency of the optimization process.

- **pop_size**: This parameter determines the number of individuals (solutions) found in each generation in the genetic algorithm. A larger population allows exploring a larger part of the solution space, increasing the probability of finding better solutions. However, a very large population size can increase computational time and lead to inefficiency. Small populations can reduce diversity and increase the risk of getting stuck in local minima. Therefore, the pop_size value should be carefully chosen to ensure a balance between exploration and exploitation.

- **mutation_rate:** This parameter determines the probability that each individual in the genetic algorithm will mutate at a certain rate. Mutation increases diversity by adding random changes to the population and prevents the algorithm from getting stuck in local minima. A low mutation_rate can make solutions more stable, but may cause diversity to decrease. A high mutation_rate can increase discovery, but can lead to excessive randomness, causing the algorithm to not converge consistently to good solutions. Therefore, the mutation_rate should be carefully selected and optimized according to the problem type.

Function body:

First, the binary images are written to the images list with the load_images function. Then, a population of size pop_size is initialized with the initialize_population function and stored in population. An empty array named fitness_history is initialized to hold the fitness (or loss) values.

Main loop loops for generations number of iterations. For each generation, the population is first sorted according to its fitness values. An empty new_generation list is started for the next generation. In a loop that will loop half the population size (pop_size), two parents are selected with the selection function. These parents are crossed with the crossover function to create two children. These children are inserted into the empty new_generation started above by mutating (or not) with the mutate function. The population list is updated as new_population. The best fitness value in the population is calculated with the calculate_fitness function (here, this function is given the value of population[0] because the population is already sorted according to its fitness values and the value of population[0] has the lowest fitness (loss) value.) The best fitness value is then placed in the fitness_history list to plot the fitness values at the end. As a result, the population[0] and fitness_history values with the lowest fitness (loss) value are returned.

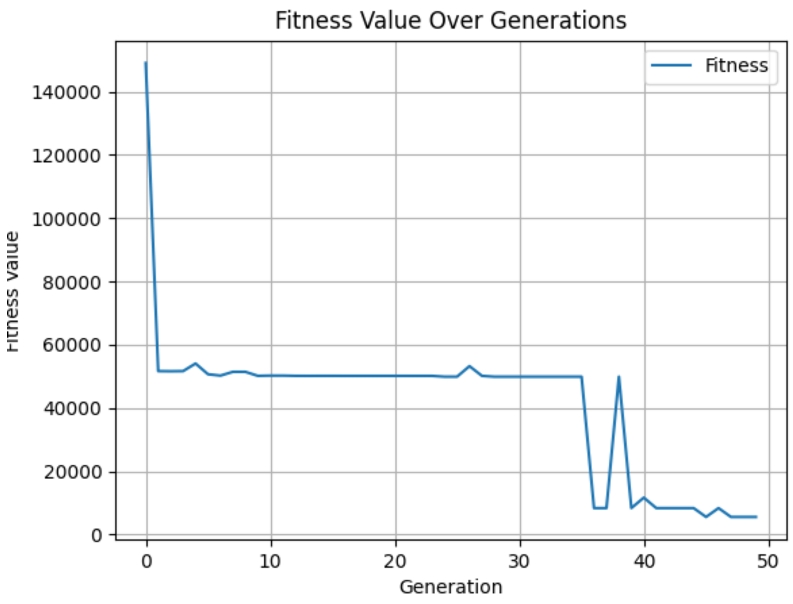
The genetic_algorithm function is called and run last, thus completing the program. The last pattern generated is saved in optimized_patterns.npy, and the best fitness value in each generation is saved in fitness_history.npy files. The plot_optimized_patterns.py program plots the last pattern generated, the plot_fitness.py program plots the fitness values recorded over generations, and the plot_reconstructed_images.py program plots new binary images created using optimal patterns. These outputs will be given in the report.

General Information:

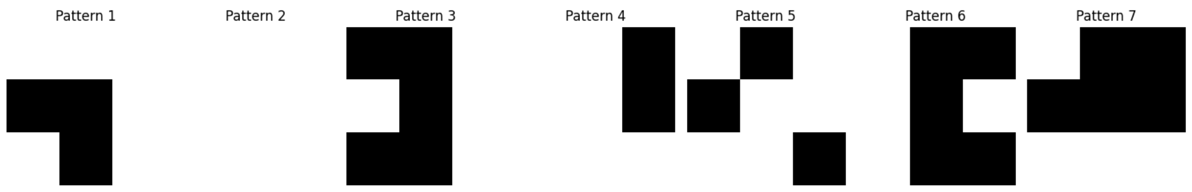
The genetic algorithm was run for this project for generation numbers of 50, 100 and 500; population sizes of 20, 50, 100, 200, 500 and 1000; mutation probability of 0.01, 0.05 and 0.1 and the results were observed. The outputs are given below according to these values:

Generations: 50, pop_size: 20, mutation_rate:0.01

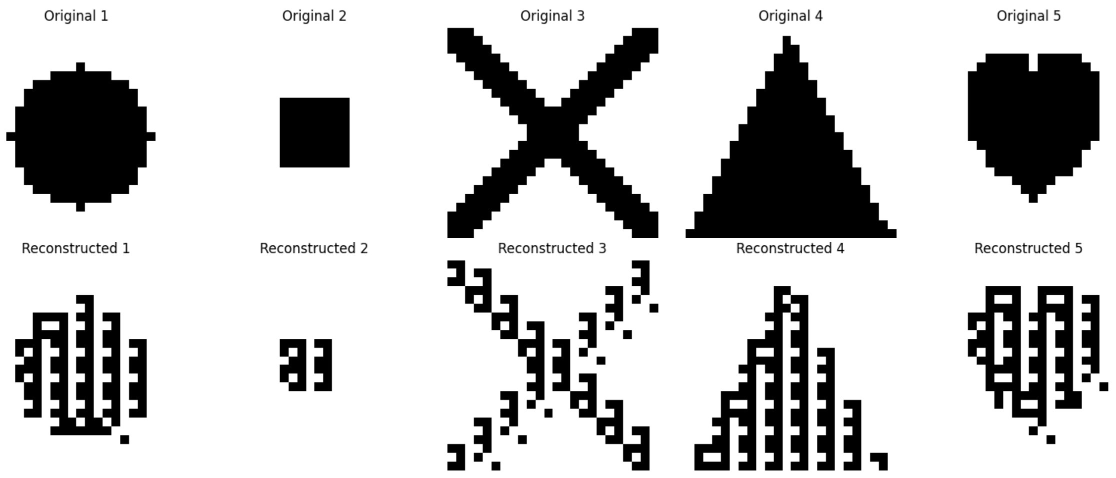
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

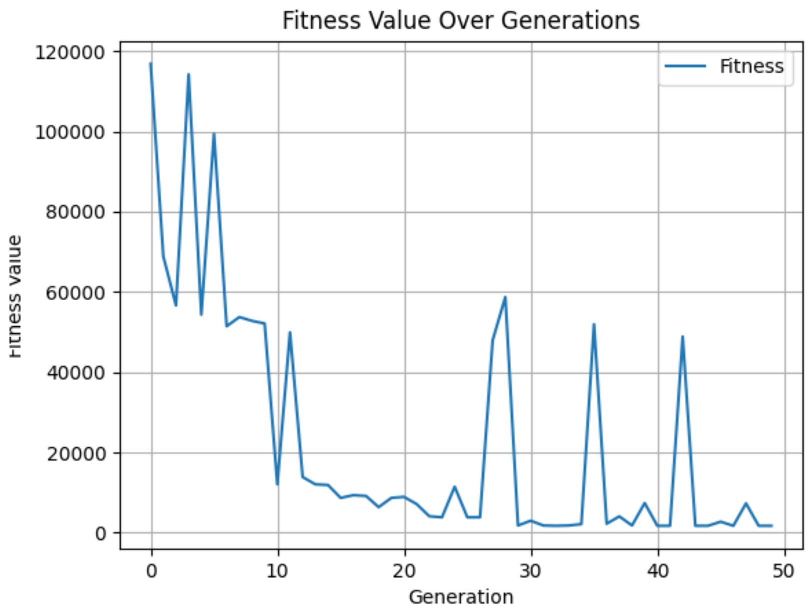


The resulting new images are:

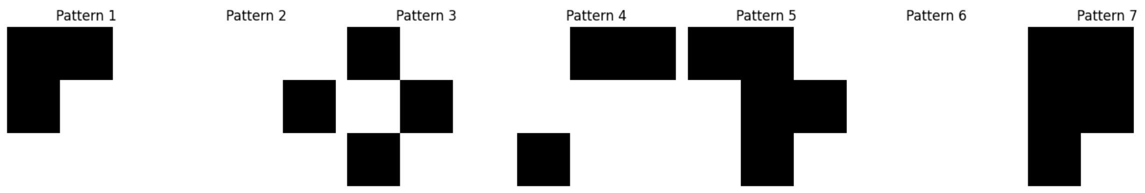


Generations: 50, pop_size: 20, mutation_rate:0.05

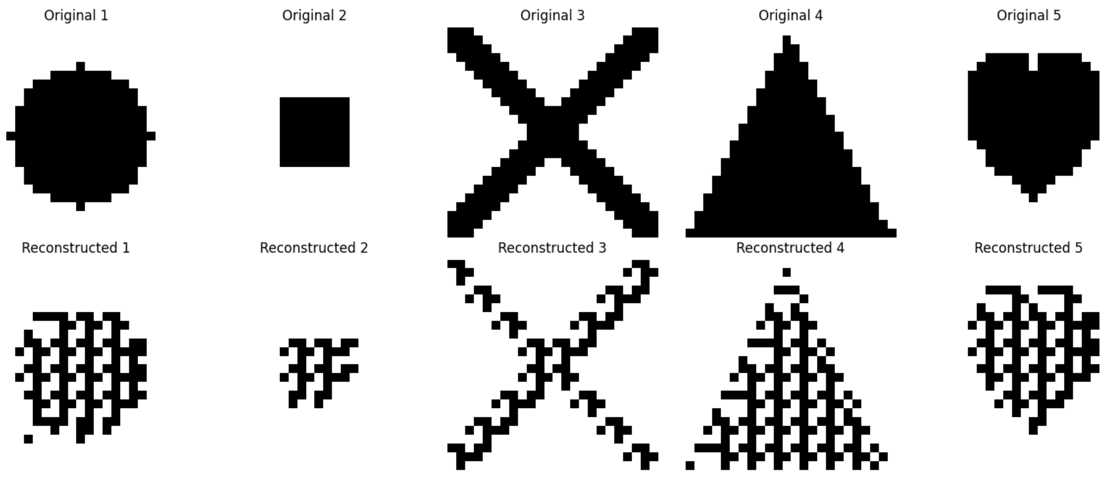
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

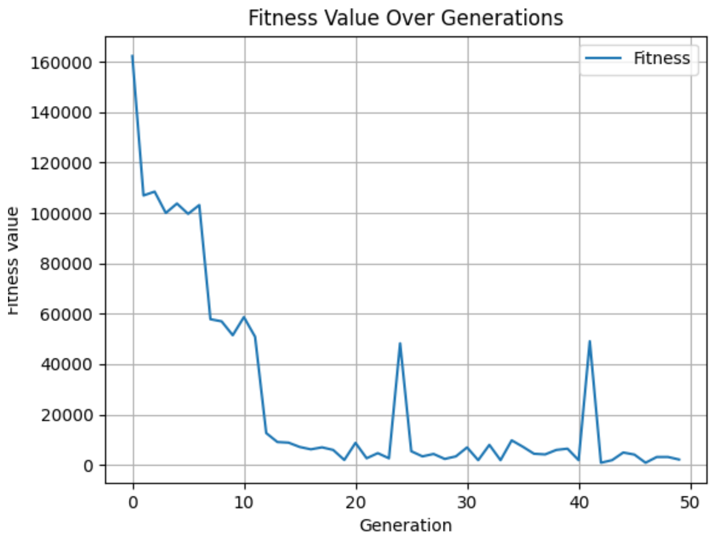


The resulting new images are:

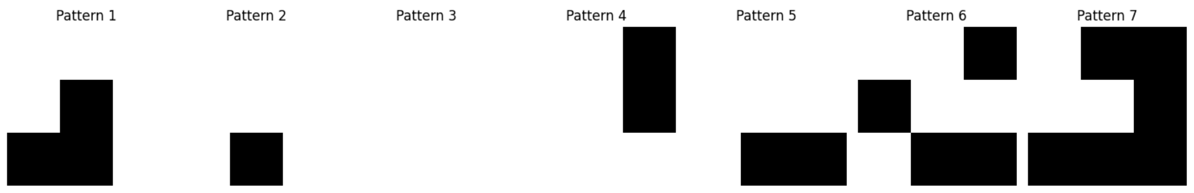


Generations: 50, pop_size: 20, mutation_rate:0.1

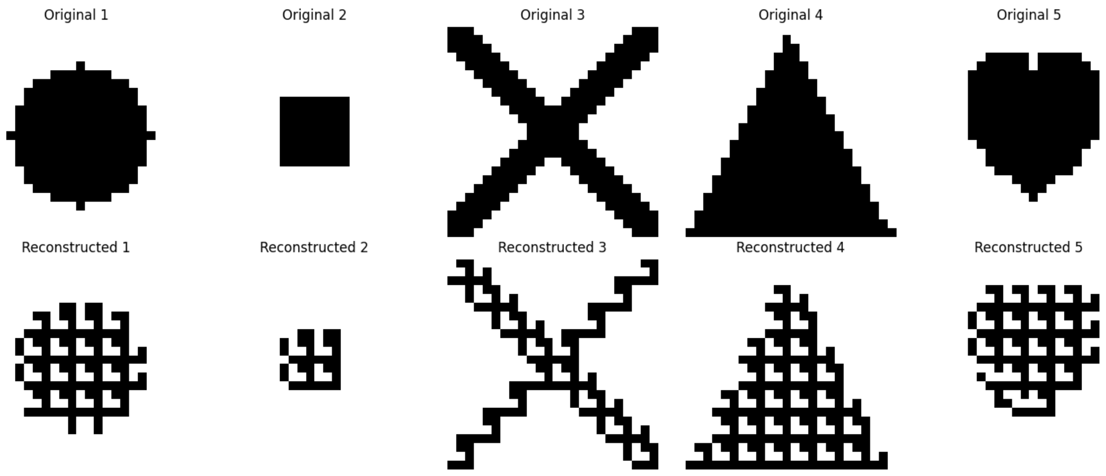
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

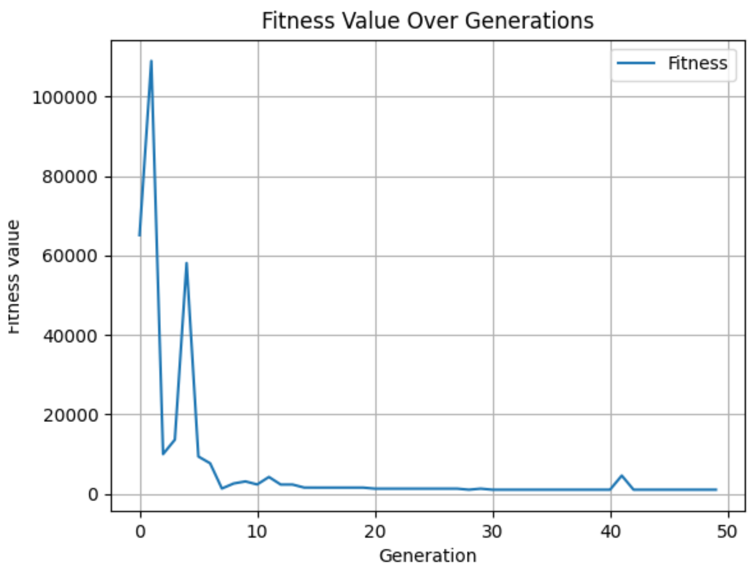


The resulting new images are:

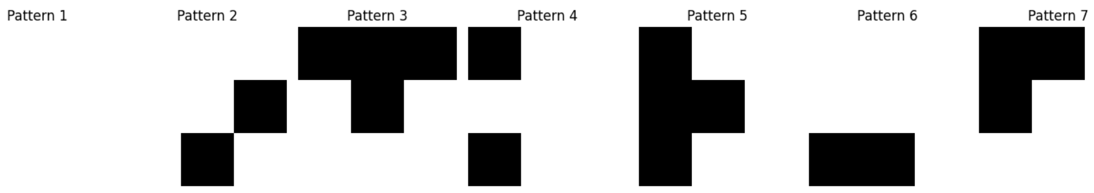


Generations: 50, pop_size: 50, mutation_rate:0.01

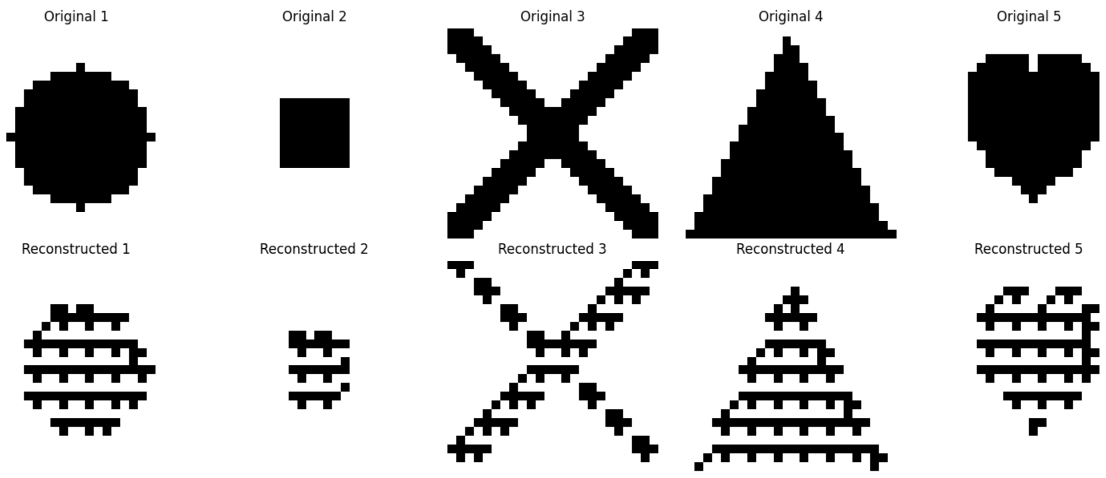
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

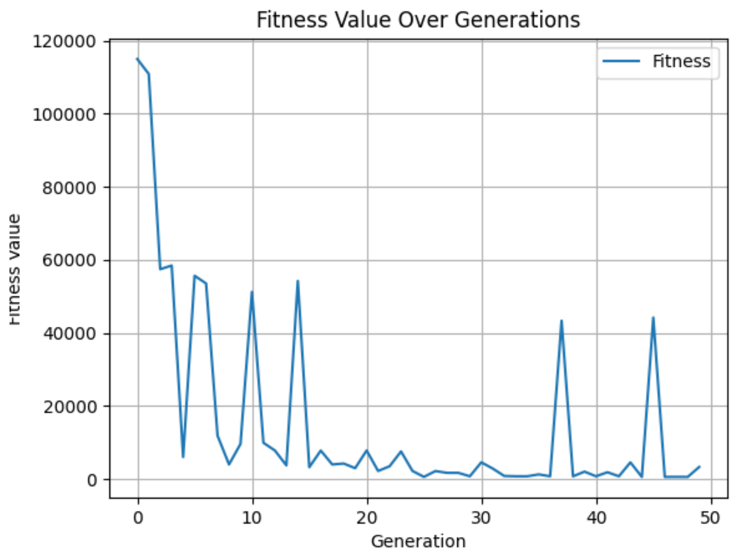


The resulting new images are:

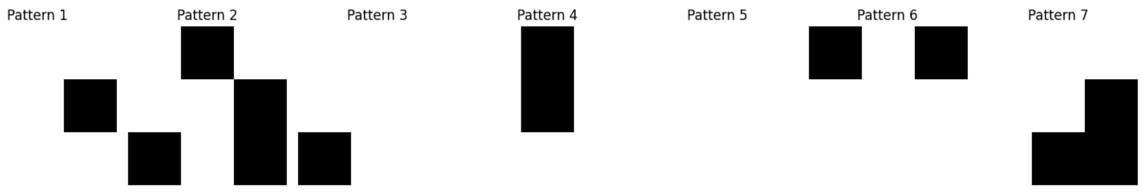


Generations: 50, pop_size: 50, mutation_rate:0.05

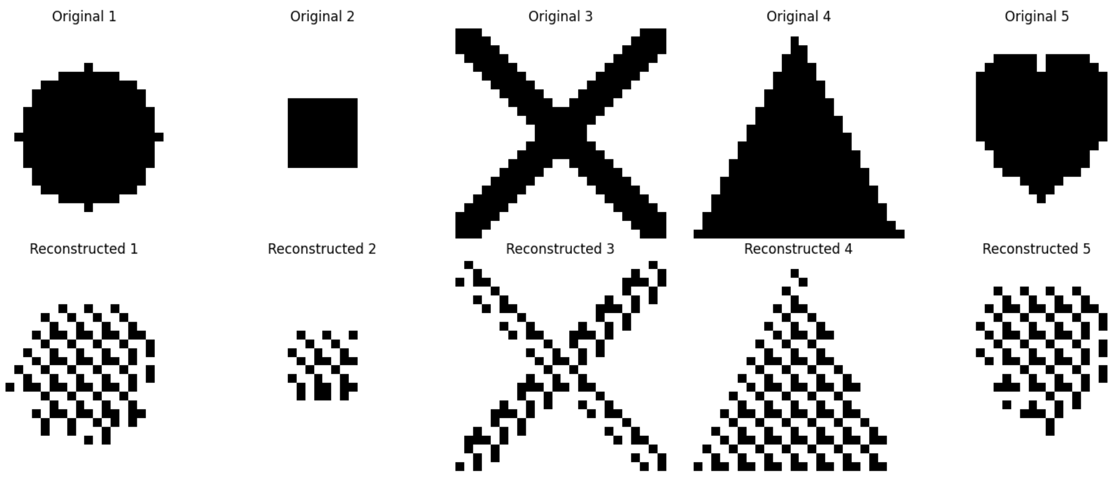
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

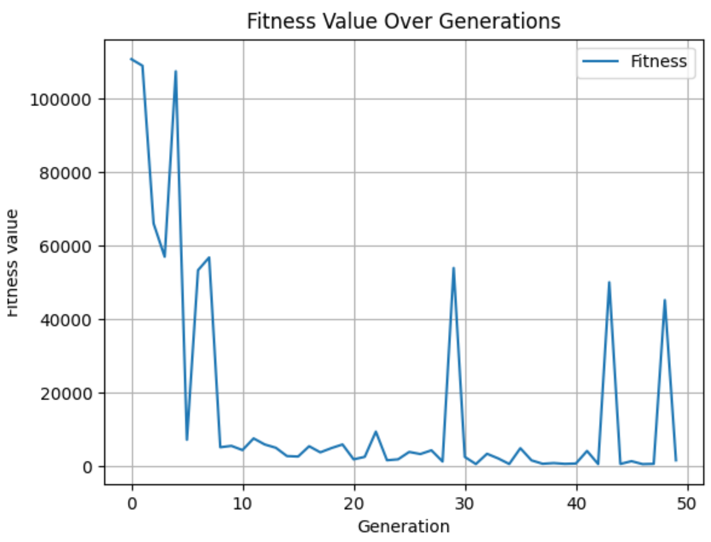


The resulting new images are:

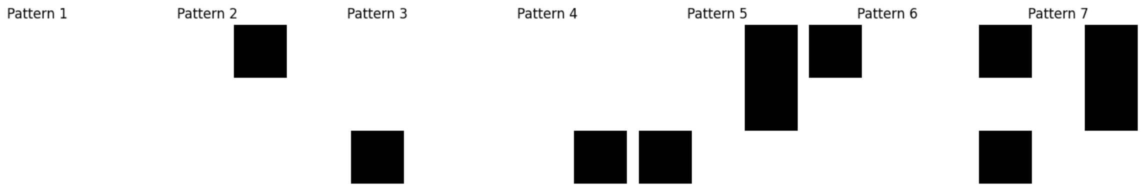


Generations: 50, pop_size: 50, mutation_rate:0.1

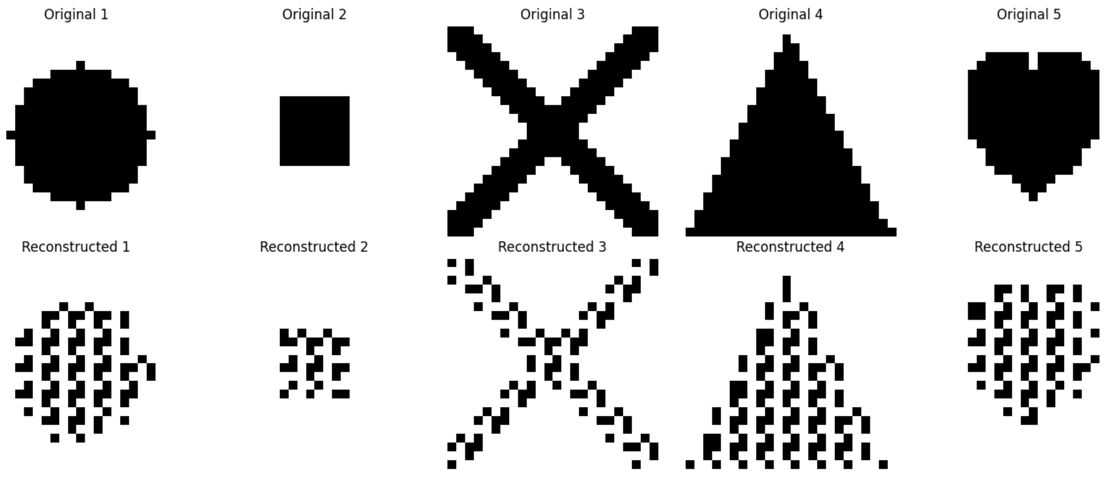
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

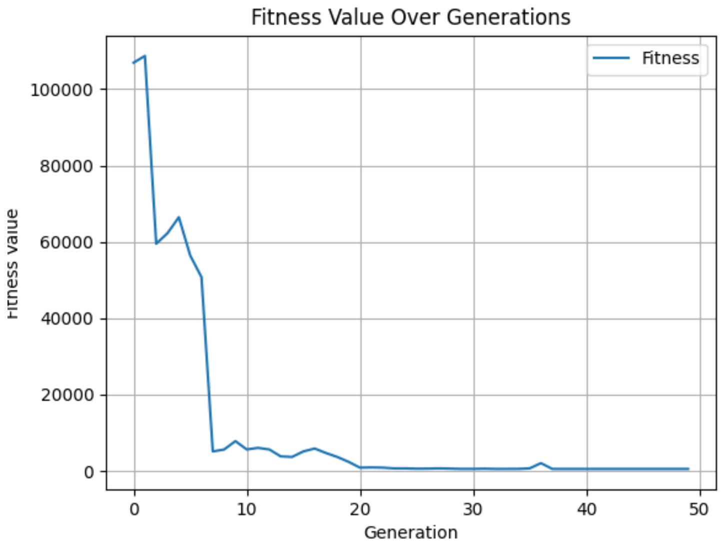


The resulting new images are:

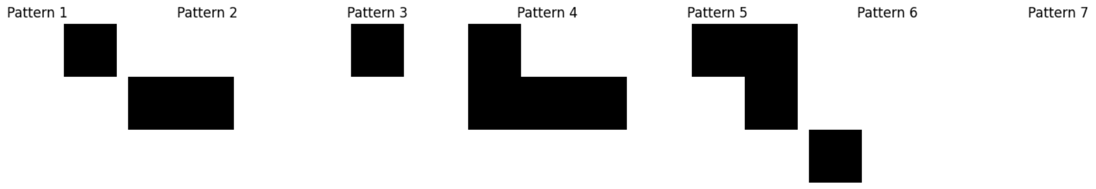


Generations: 50, pop_size: 100, mutation_rate:0.01

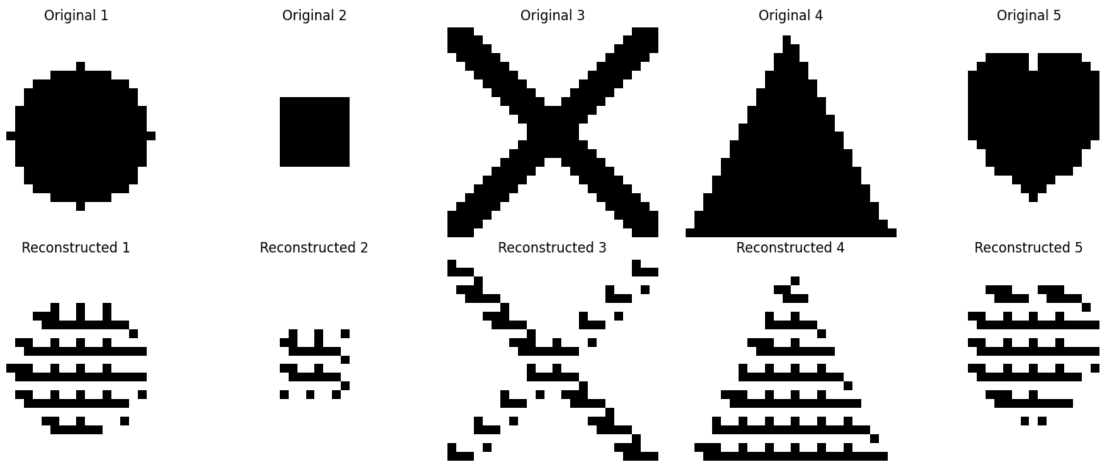
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

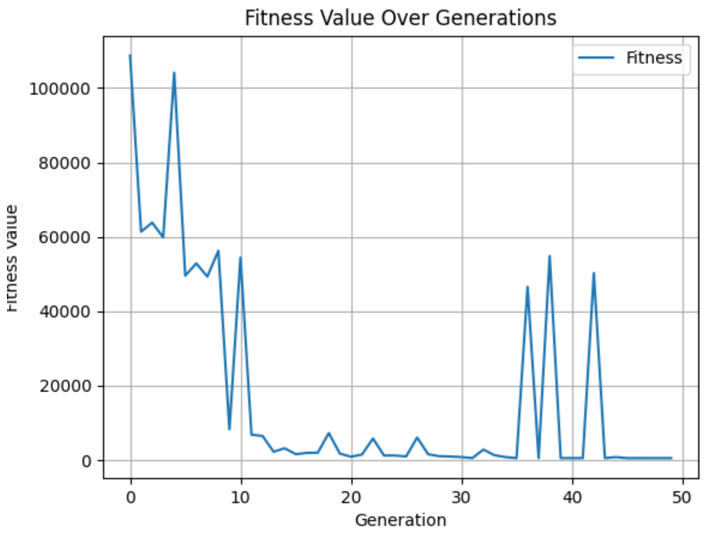


The resulting new images are:

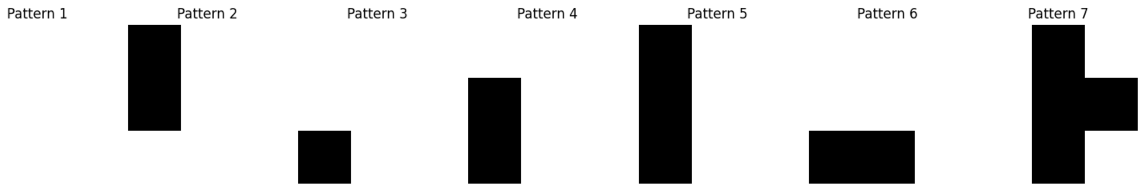


Generations: 50, pop_size: 100, mutation_rate:0.05

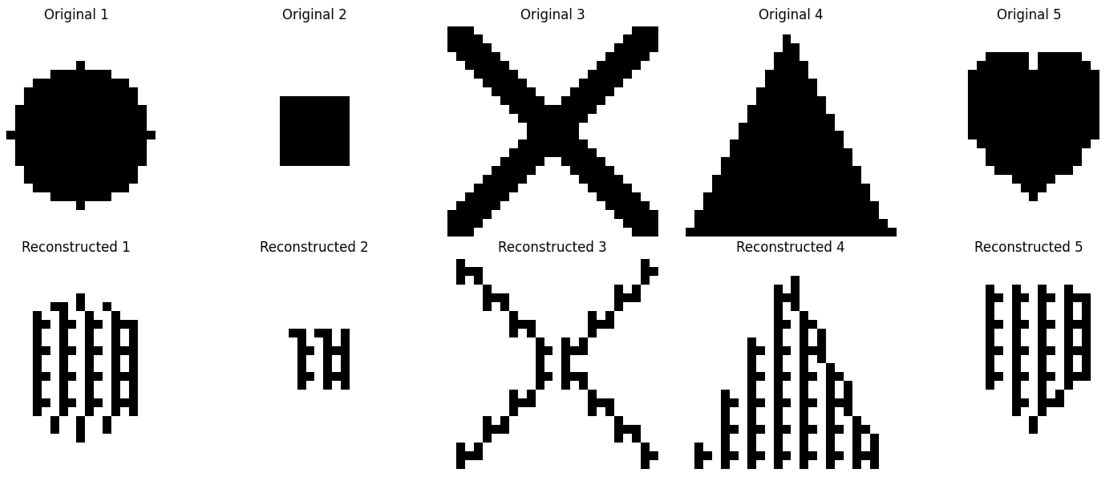
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

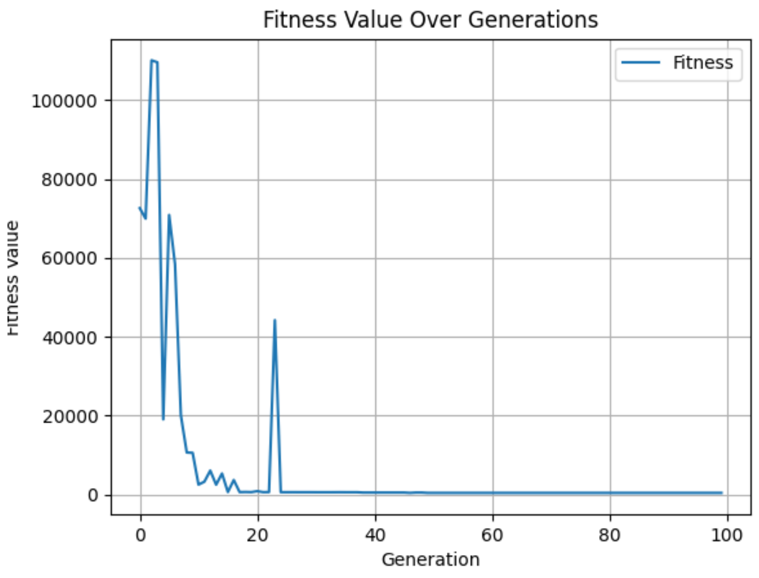


The resulting new images are:

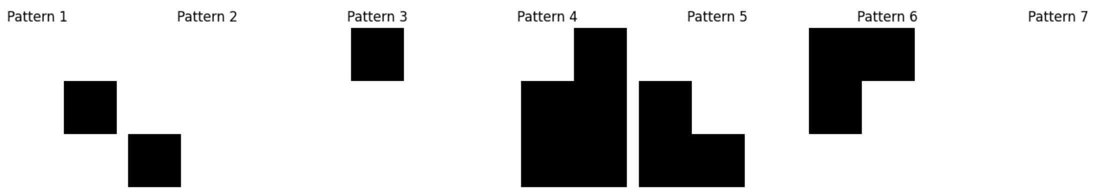


Generations: 100, pop_size: 200, mutation_rate:0.01

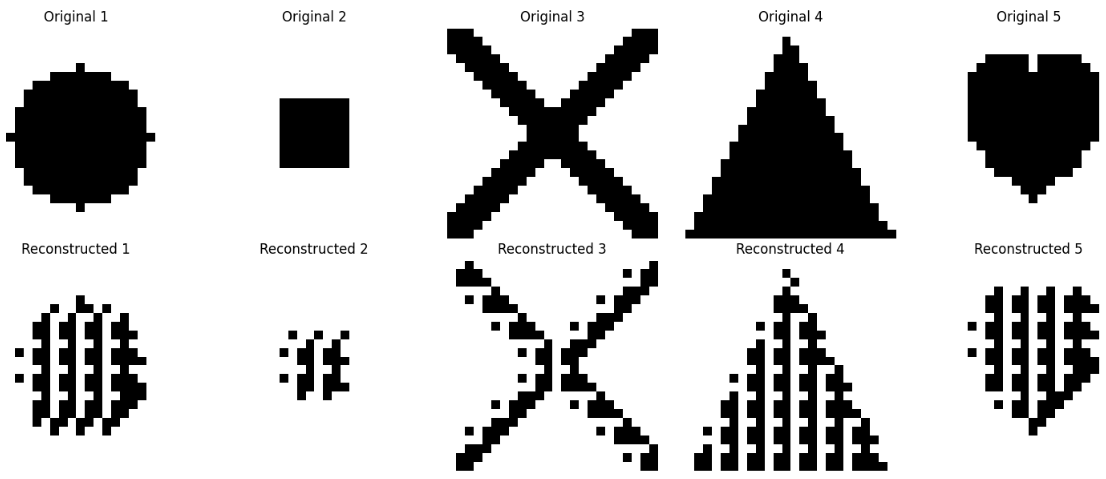
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

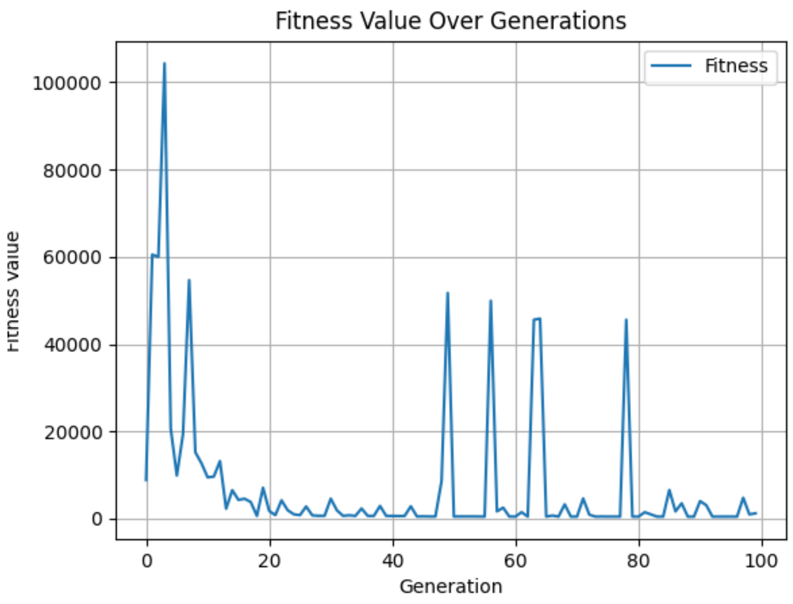


The resulting new images are:

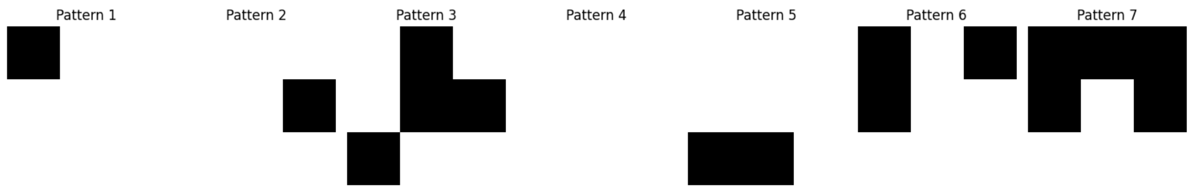


Generations: 100, pop_size: 200, mutation_rate:0.05

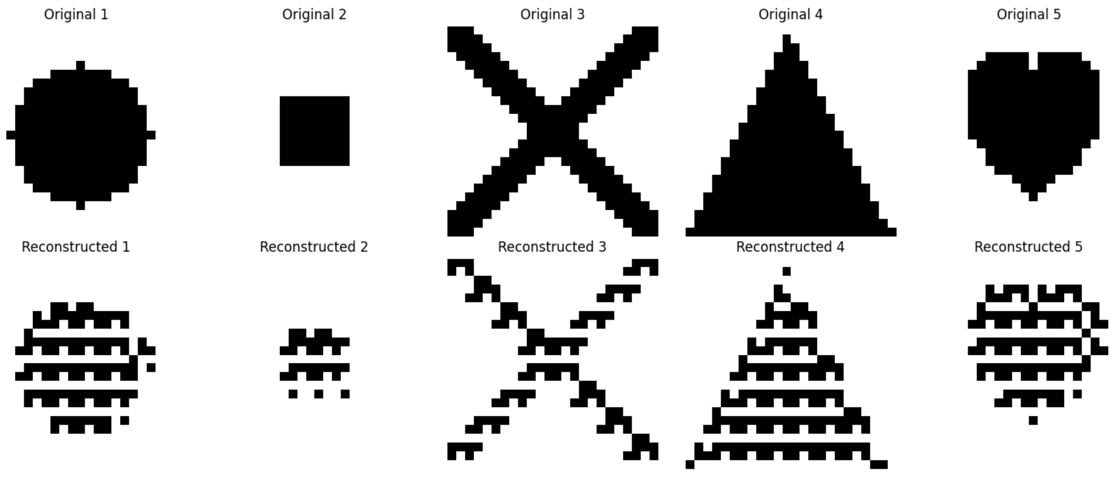
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

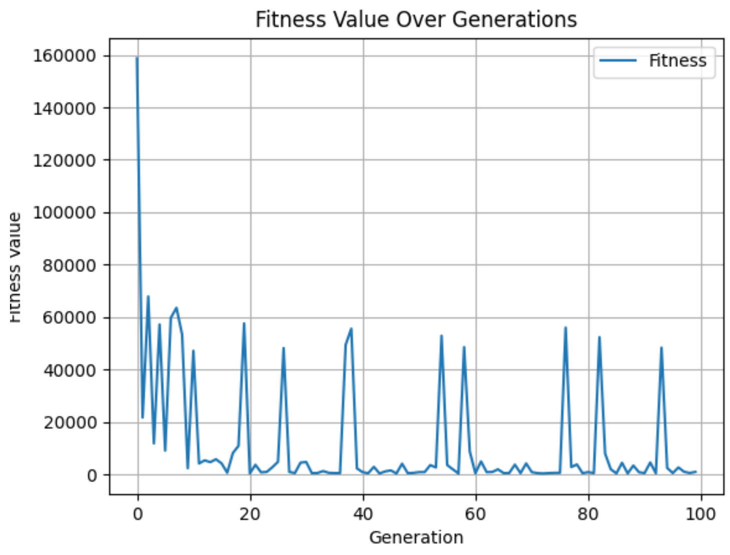


The resulting new images are:

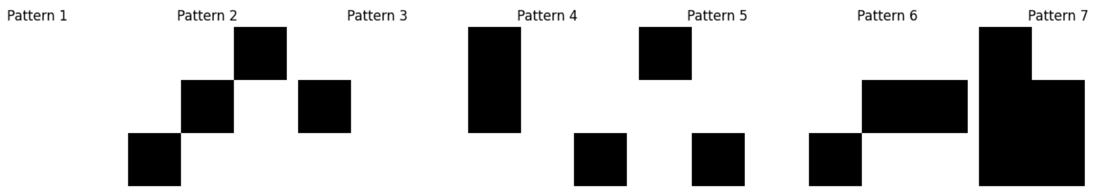


Generations: 100, pop_size: 200, mutation_rate:0.1

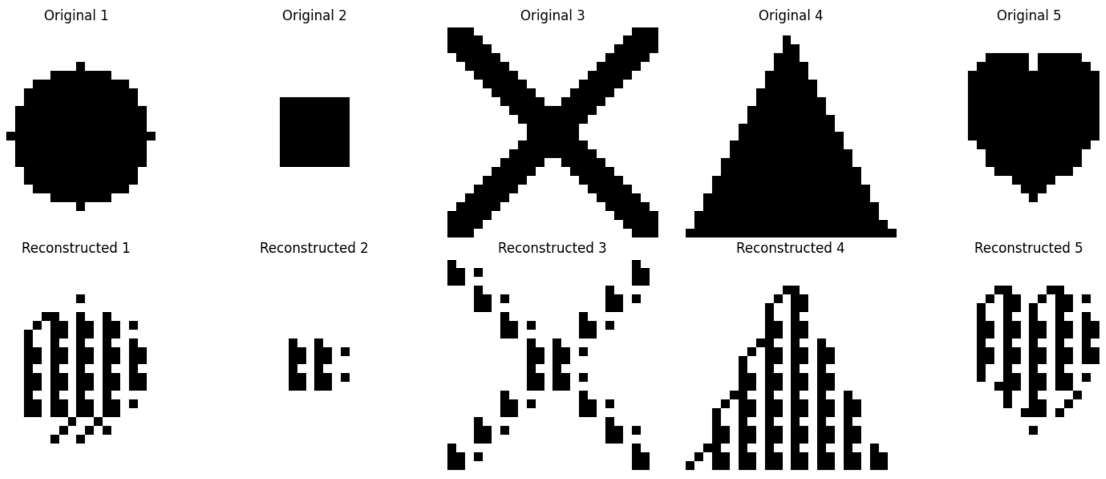
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

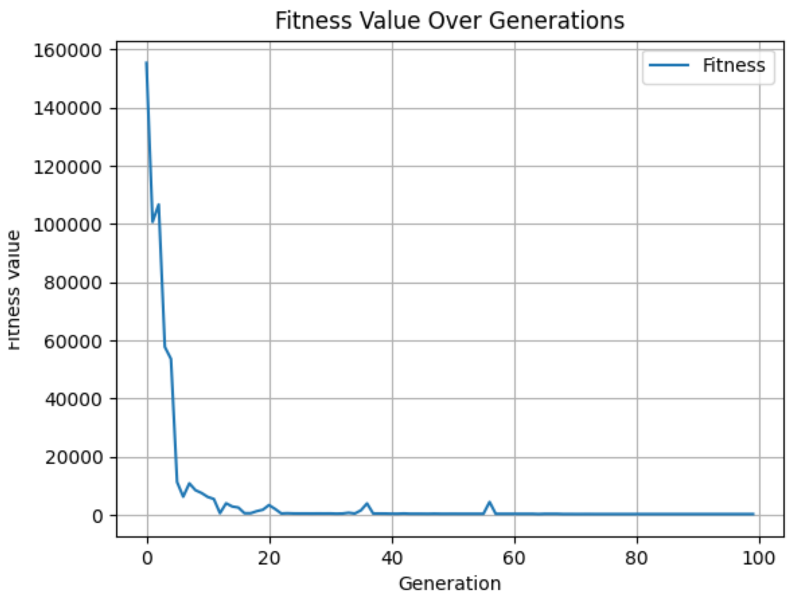


The resulting new images are:

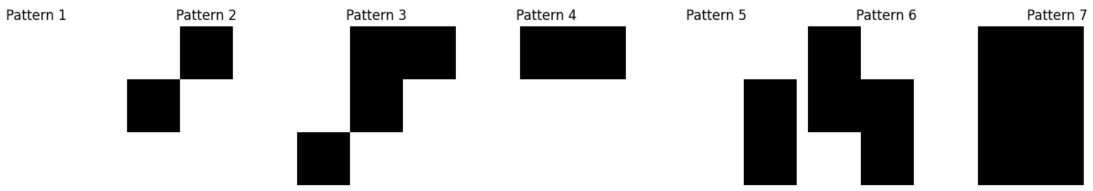


Generations: 100, pop_size: 500, mutation_rate:0.01

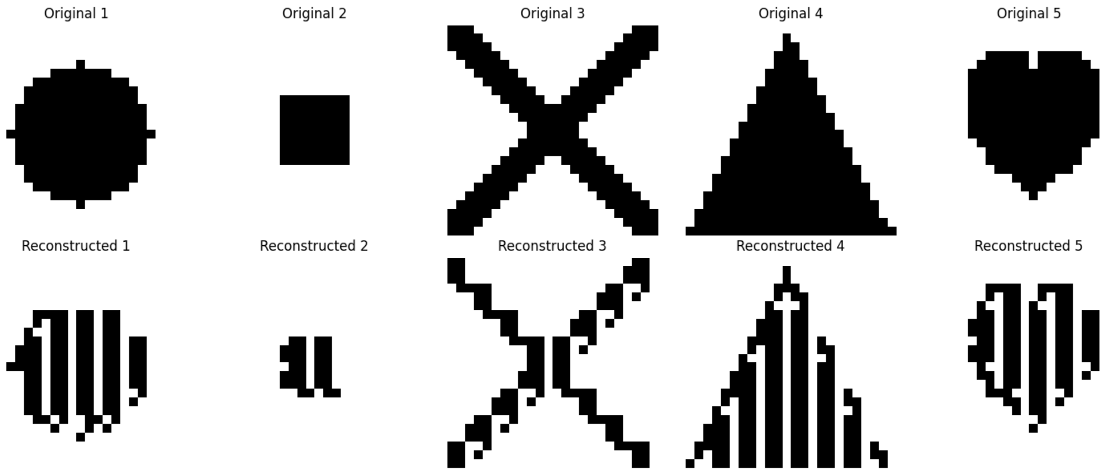
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

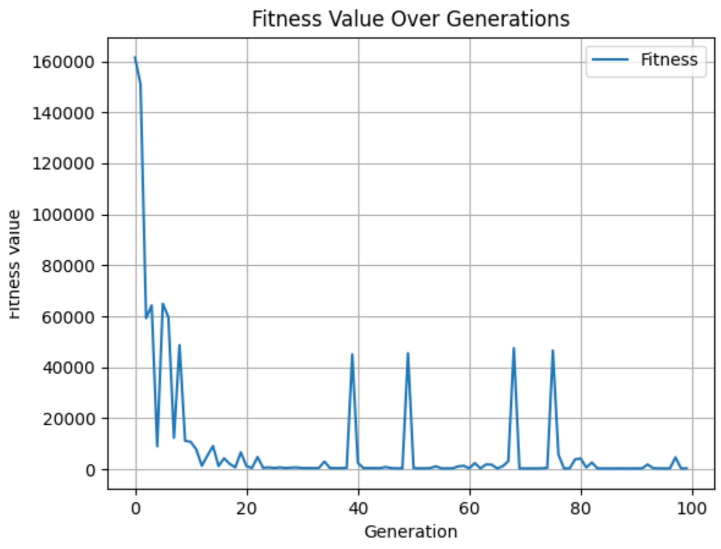


The resulting new images are:

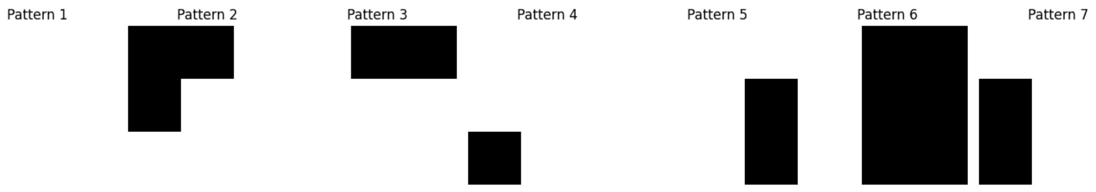


Generations: 100, pop_size: 500, mutation_rate:0.05

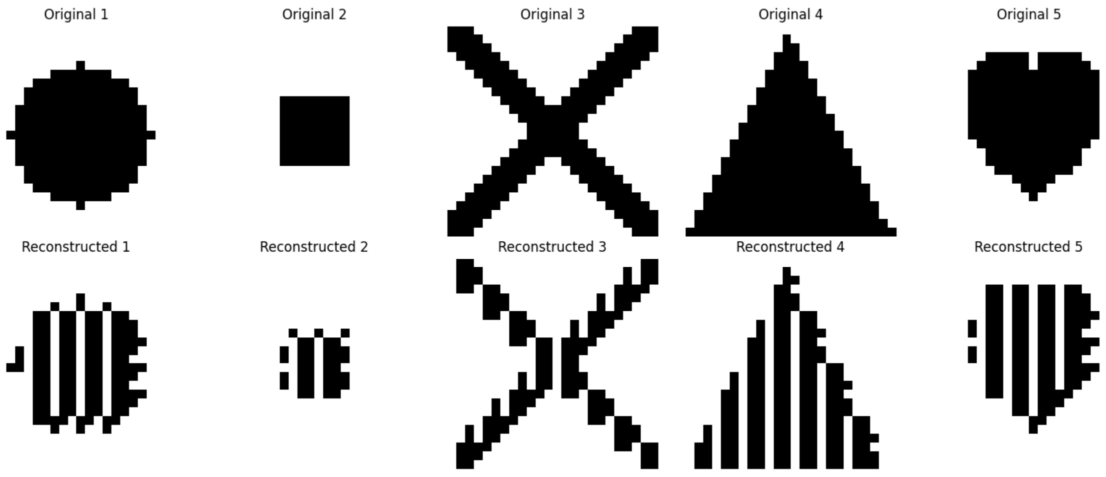
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

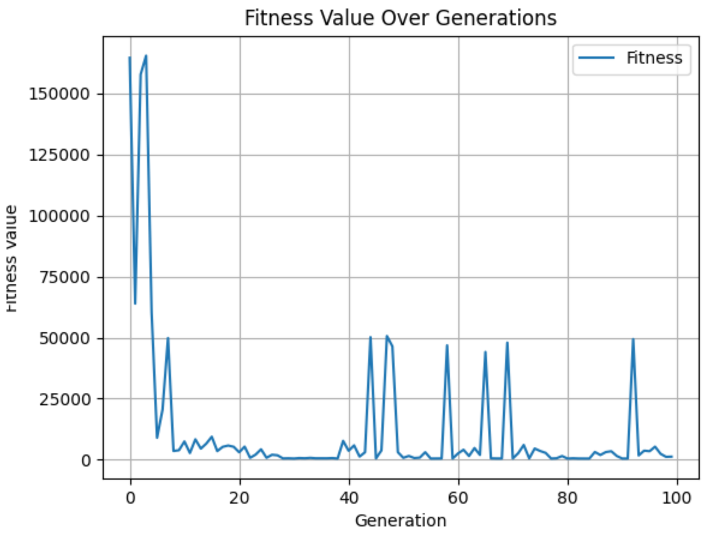


The resulting new images are:

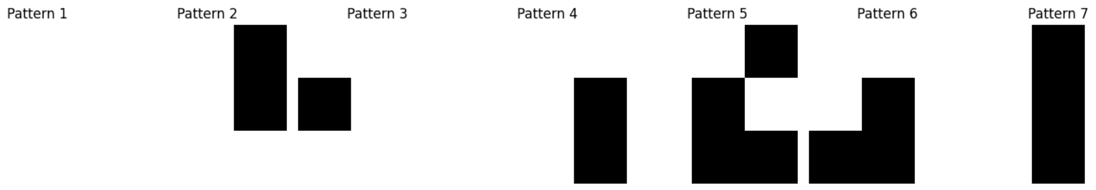


Generations: 100, pop_size: 500, mutation_rate:0.1

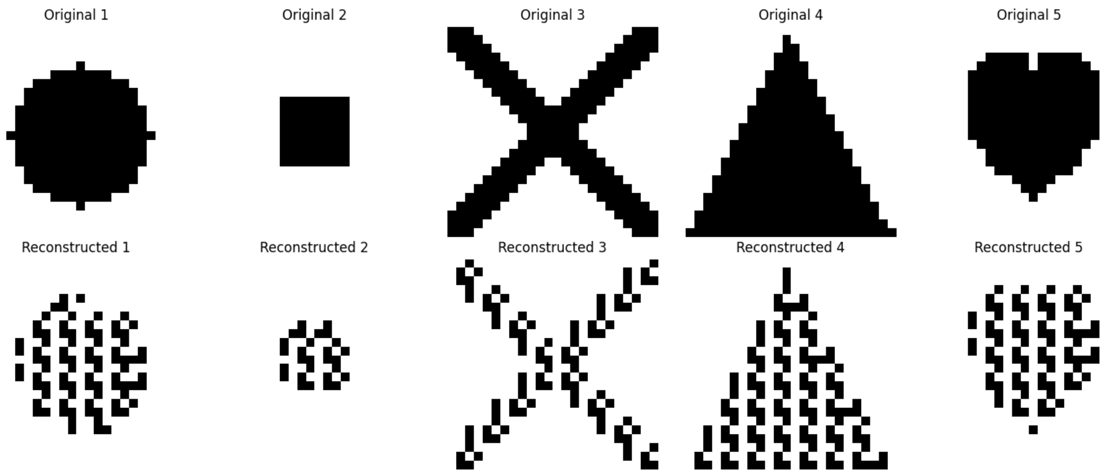
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

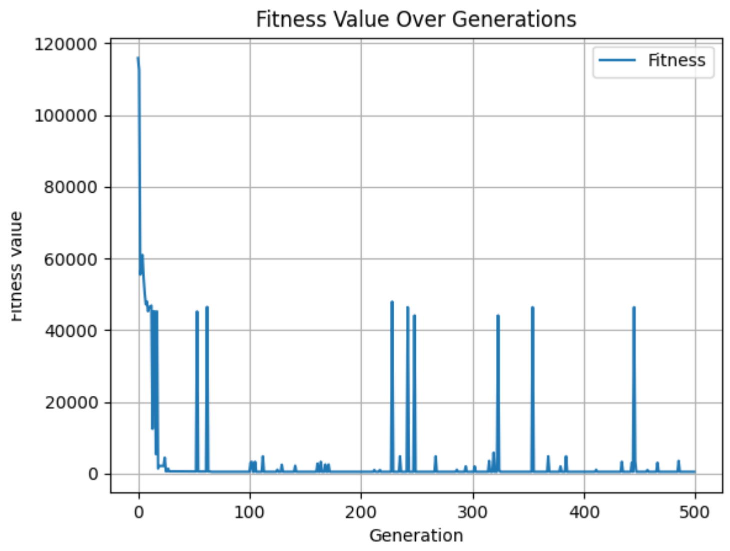


The resulting new images are:

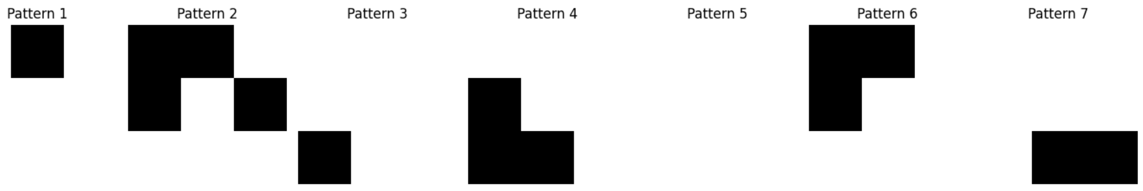


Generations: 500, pop_size: 50, mutation_rate:0.01

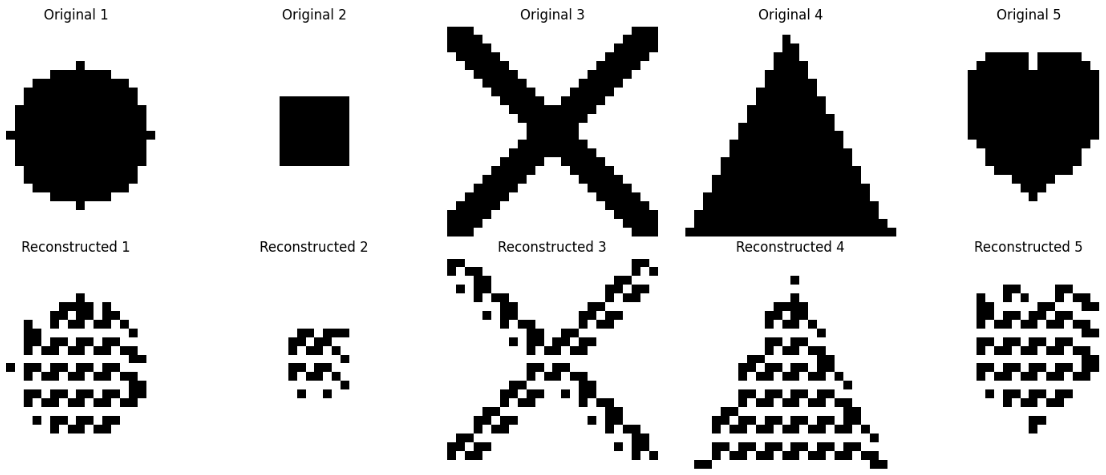
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

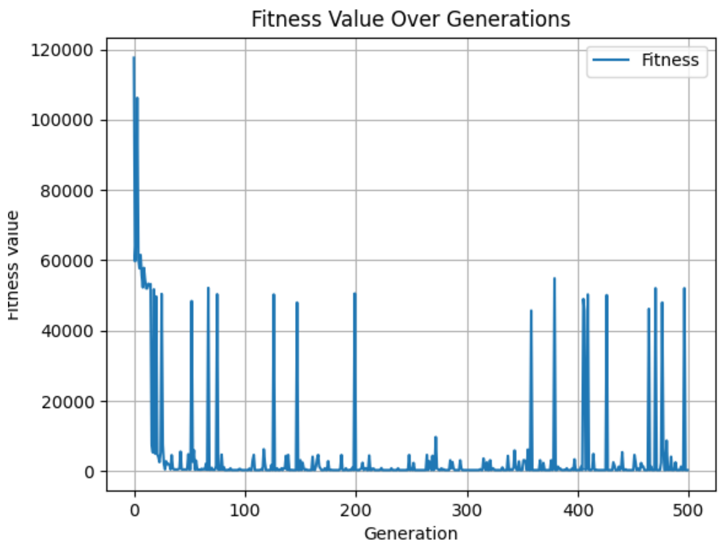


The resulting new images are:

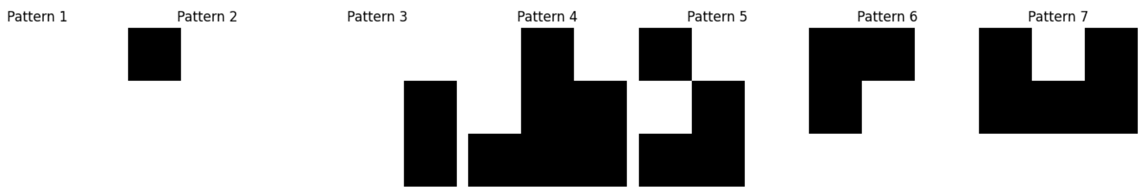


Generations: 500, pop_size: 50, mutation_rate:0.05

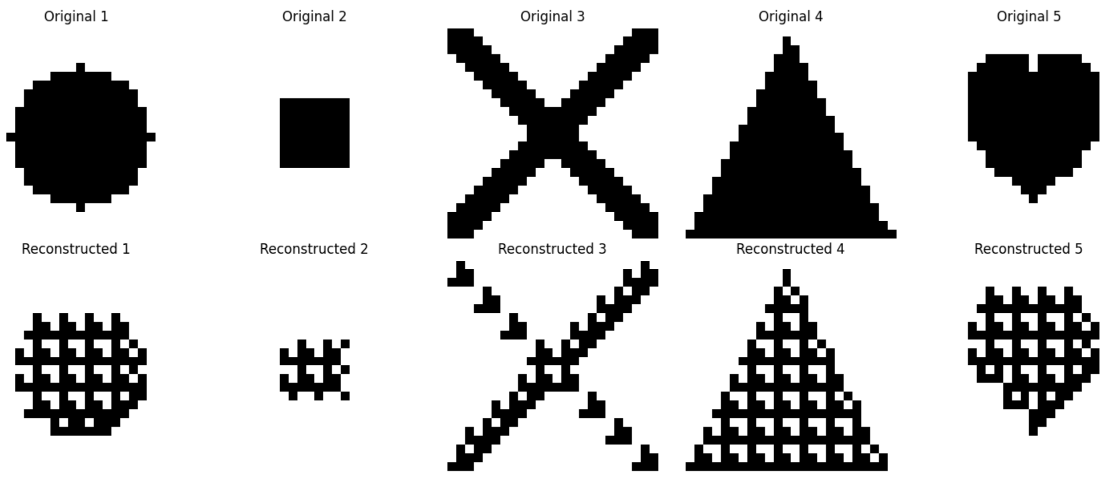
Changes in optimal fitness values across generations:



The latest optimized patterns produced:

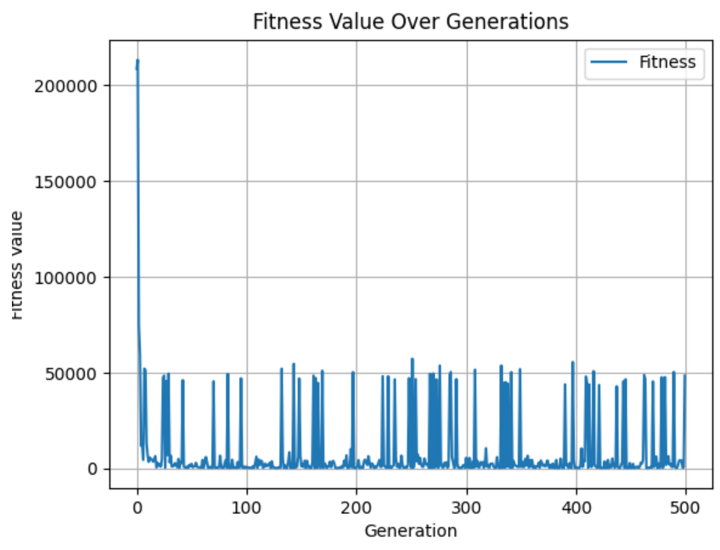


The resulting new images are:

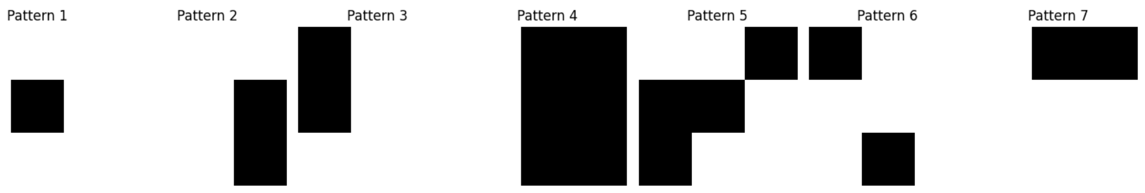


Generations: 500, pop_size: 50, mutation_rate:0.1

Changes in optimal fitness values across generations:



The latest optimized patterns produced:

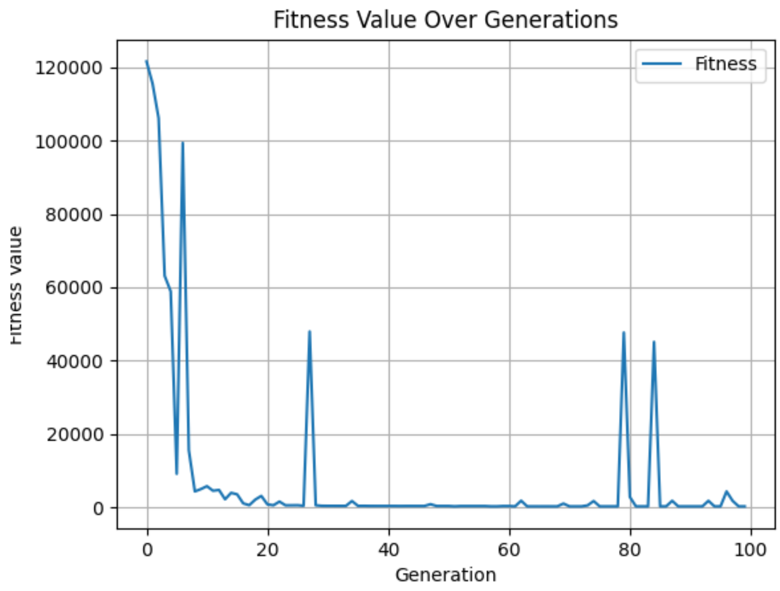


The resulting new images are:

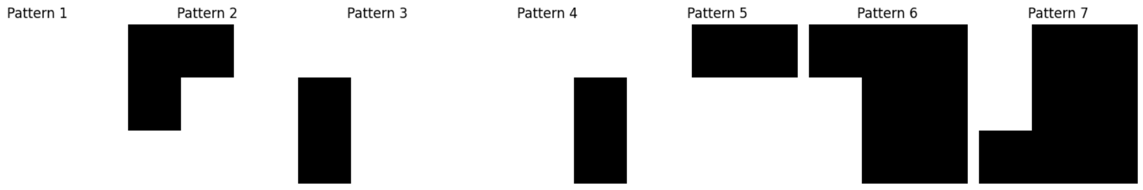
(In this example, there was a misfortune; the place where the jump occurred due to mutation coincided with the last generation and the final pattern obtained had a high fitness value, whereas a more suitable result would have been obtained since it was ran for 500 generations.)

Generations: 100, pop_size: 1000, mutation_rate:0.02

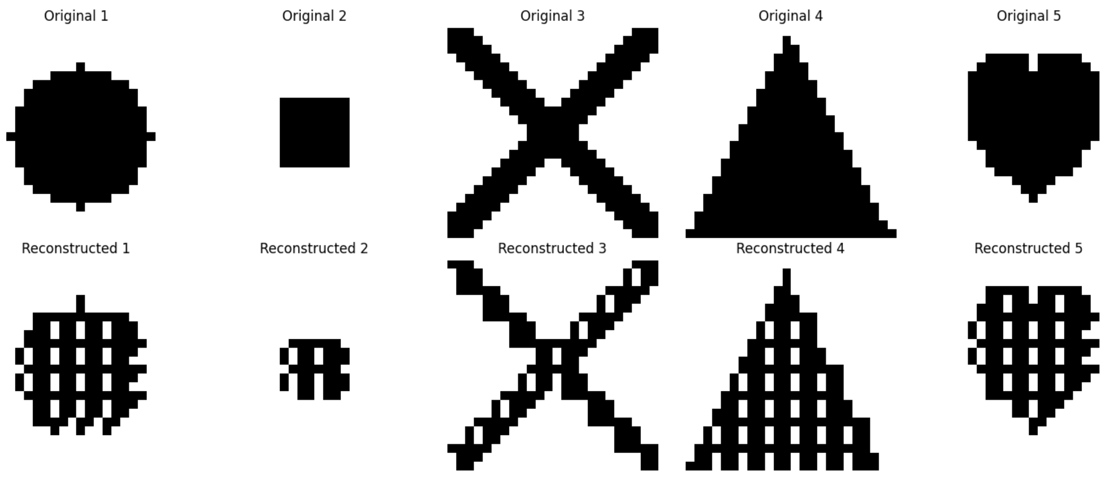
Changes in optimal fitness values across generations:



The latest optimized patterns produced:



The resulting new images are:



Comments:

Experiments on the number of generations, population size and mutation rate have revealed the effects of the genetic algorithm on the optimization process. While increasing the number of generations helps to find better solutions by providing more evolutionary steps, it has been observed that improvement slows down after a certain point. Increasing the population size allowed the discovery of a larger solution space, but it was observed that the computational cost increased at high population values. The mutation rate was a critical factor in terms of providing diversity, and while there was a risk of convergence at low rates, it was observed that solutions became unstable and sudden jumps in fitness values occurred at high mutation rates. In particular, tests conducted with mutation rates of 1%, 5% and 10% showed that as the mutation rate increased, more variability occurred and sometimes the best solutions suddenly deteriorated. Choosing these parameters in a balanced way plays a critical role in increasing the effectiveness of the genetic algorithm.