

**SCHOOL OF COMPUTER ENGINEERING**

**KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY**

**BHUBANESWAR, ODISHA - 751024**

**A REPORT**

**on**

**“Genetic Algorithm: Implementation of a program that takes an image as its input and generates the same image using N number of squares.”**

**Submitted to**

**Sohail Khan**

**Submitted by**

**Shreya Snehal-21051338**

**UNDER THE GUIDANCE OF**

**Mr. Sohail Khan**

**Acknowledgements**

I am profoundly grateful to Mr. Sohail Khan for his expert guidance and continuous encouragement throughout to see that this project meets its target since its commencement to its completion.

**Introduction**

A genetic algorithm, or GA, is a potent optimisation and search method that draws inspiration from evolution and natural selection. It is frequently used to solve complicated issues where the search space is large and challenging to effectively explore. The larger topic of evolutionary algorithms includes GAs.

A population of possible solutions, frequently depicted as people or chromosomes, is the fundamental component of a genetic algorithm. Evolutionary operators are applied to these individuals that resemble crossover, mutation, and natural selection.

1. **Initialization**: A population of arbitrary people is formed, each of whom stands for a possible fix for the issue. These people usually have a phenotype (the real solution) that corresponds to their genotype (a collection of characteristics or genes).

2. **Evaluation:** Each individual's fitness is assessed using an objective function. The fitness score quantifies how well an individual solves the problem, and it guides the search towards better solutions.

3. **Selection:** Based on fitness, individuals are selected for reproduction. Individuals with greater fitness levels are more likely to be chosen, thus replicating the "survival of the fittest" theory.

4. **Crossover:** To produce one or more children, a pair of chosen individuals exchange genetic material, often their genes. This is akin to natural genetic recombination.

5. **Mutation:** The offspring's DNA undergo haphazard alterations that provide genetic variety to the community. New areas of the search space are explored with the aid of mutation.

6. **Replacement:** The children replace the least suitable members of the population, maybe in conjunction with some of the parents. This ensures that the population size remains the same.

7. **Termination:** The algorithm iteratively performs these steps over multiple generations. Termination can be based on a set number of generations, reaching a satisfactory solution, or other criteria.

GAs are versatile and have been applied to numerous real-world problems, from optimizing complex engineering designs to training machine learning models and scheduling tasks. They excel in scenarios where traditional search methods are less effective due to the high dimensionality of the solution space or the presence of multiple optima.

By mimicking the evolutionary process, Genetic Algorithms efficiently explore the search space and gradually converge towards solutions that often outperform those found by other optimization techniques. They are a valuable tool for tackling complex optimization problems across various domains.

CROSSOVER

The crossover operation in a genetic algorithm is a critical step where new solutions, often referred to as offspring, are created by combining elements from two or more parent solutions. In the context of generating an image using squares, the crossover operation should combine squares from parent solutions to create new sets of squares.

MUTATION OPERATION

The mutation operation in a genetic algorithm introduces small random changes to individuals in the population. Its purpose is to maintain genetic diversity and explore new regions of the solution space. In the context of generating an image using squares, the mutation operation pertains to modifying the properties of individual squares (e.g., positions, sizes, or colors) within a solution.

Types of Mutations:

1.Position Mutation:This mutation involves changing the position (coordinates) of one or more squares within a solution. For example, a square's position can be shifted by a small random amount in the horizontal and vertical directions.

2.Size Mutation: Size mutation changes the dimensions (width and height) of squares. You can increase or decrease the size of a square by a small random factor.

3.Color Mutation: Color mutation involves modifying the color properties of squares. You can change the fill color, stroke color, or transparency of a square, again by a small random amount.

4.Rotation Mutation: In some cases, you might introduce rotation mutations to change the orientation of squares. This could be relevant if squares are allowed to be rotated in the generated image.

5.Other Custom Mutations: Depending on the specific requirements of your problem, you can introduce custom mutations. For example, if the image is composed of more complex shapes besides squares, you might define mutations specific to those shapes.

In summary, mutation is a crucial operator in genetic algorithms that introduces random variations to individuals, enabling the algorithm to explore and adapt in the search for an optimal solution. The mutation rate should be carefully chosen to strike the right balance between exploration and exploitation.

SELECTION METHOD

In a genetic algorithm, the selection method is a crucial process for choosing which individuals (solutions) from the current population will be used to create the next generation. Different selection methods can be employed, but two common ones are "Roulette Wheel Selection" and "Tournament Selection."

In summary, the selection method in a genetic algorithm is responsible for deciding which individuals become parents for the next generation. It is a balance between favoring high-fitness solutions for convergence and allowing lower-fitness solutions to maintain diversity and adaptability. The choice of selection method depends on the specific problem and the desired trade-off between exploration and exploitation.

**Implementation Details:**

**Canvas Generation**: I have defined a function generate\_canvas(h, w) that generates an empty canvas as a NumPy array to match the input image's dimensions. This canvas will be used to compose the final generated image.

**Square Generation**: The generate\_squares(n, h, w) function generates n random squares. Each square is represented by its position (x, y), color rgb, and opacity. This is the initial population of squares.

**Evaluation Function**: The evaluate\_square(square, input\_image) function calculates the quality of a square in relation to the input image. It currently uses a simple squared difference method for evaluation, which can be customized for more advanced fitness evaluations.

**Genetic Algorithm Loop:** Lastly, remember to visualize or save intermediate results during the loop to monitor the progress of the genetic algorithm and understand how the image evolves over generations. I used libraries like Matplotlib to display intermediate results or save them to analyze the algorithm's performance.

Crossover

Crossover is a key genetic operation where two or more parent squares are combined to create new offspring squares. In this code, the crossover operation is implemented as follows:

crossover\_squares(squares) function selects two random parent squares from the current population.

A crossover point is randomly chosen in the RGB values of the squares.

Two offspring squares are created by swapping the RGB values between the parents at the crossover point.

The offspring squares are added to the population.

def crossover\_squares(squares):

"""Performs crossover on two squares."""

square1, square2 = random.sample(squares, 2)

x1, y1, rgb1, opacity1 = square1

x2, y2, rgb2, opacity2 = square2

# Randomly select a crossover point.

crossover\_point = random.randint(0, len(rgb1))

# Create two offspring squares by combining RGB values.

new\_rgb1 = rgb1[:crossover\_point] + rgb2[crossover\_point:]

new\_rgb2 = rgb2[:crossover\_point] + rgb1[crossover\_point:]

# Add offspring squares to the new population.

squares.extend([(x1, y1, new\_rgb1, opacity1), (x2, y2, new\_rgb2, opacity2)])

Mutation (Step 4):

Mutation introduces small random changes in the properties of squares to add diversity to the population. The code implements mutation as follows:

For each square in the population, a mutation operation is applied based on a mutation rate.

Random properties like position (x, y), RGB values, and opacity are selected for mutation.

Random changes within certain bounds are applied to the chosen properties.

The original square is replaced with the mutated square in the population.

def mutate\_square(square, mutation\_rate):

"""Mutates a square with a given mutation rate."""

x, y, rgb, opacity = square

if random.uniform(0, 1) < mutation\_rate:

# Randomly mutate the position.

x += random.randint(-1, 1)

y += random.randint(-1, 1)

if random.uniform(0, 1) < mutation\_rate:

# Randomly mutate the RGB values.

rgb += np.random.randint(-5, 6, size=3)

rgb = np.clip(rgb, 0, 255) # Ensure RGB values are within bounds

if random.uniform(0, 1) < mutation\_rate:

# Randomly mutate the opacity.

opacity += random.uniform(-0.1, 0.1)

opacity = max(0, min(1, opacity)) # Ensure opacity is within bounds

return (x, y, rgb, opacity)

Selection (Step 5):

Selection involves choosing the best squares from the current population to form the next generation. The code implements selection as follows:

Evaluate the quality of each square in the current population using an objective function.

Sort the squares based on their objective function values, with higher values indicating better squares.

Select the top N squares to form the next generation.

def select\_squares(squares, N):

"""Selects N best squares from a population based on their objective function values."""

objective\_function\_values = [evaluate\_square(square) for square in squares]

sorted\_indices = np.argsort(objective\_function\_values)[::-1]

selected\_squares = [squares[i] for i in sorted\_indices[:N]]

return selected\_squares

These genetic operations are repeated for multiple generations to evolve the population and create an image using the Genetic Algorithm. The algorithm aims to create a set of squares that, when combined, approximate the target image.

**Githublink:<https://github.com/ShreyaSnehal/21051338_AI>**