NAME: UDITAA GARG ROLL NO: 21052632 SEC: CS_18

Report on Implementation of Crossover, Mutation, and Selection

1. Crossover:

Crossover is a genetic operator that combines two or more parent solutions to create new offspring. In the context of generating an image using squares, crossover can be used to combine the attributes (position, color, opacity) of two squares to create new squares.

Implementation:
One-Point Crossover: One approach is to perform one-point crossover, where you randomly select a point in the DNA (square representation) and exchange the information before and after that point between two parent squares. This can result in two new child squares.

```
def one_point_crossover(parent1, parent2):
    # Randomly choose a crossover point
    crossover_point = random.randint(0, len(parent1))

    # Create two children by swapping DNA segments
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]

    return child1, child2
```

2. Mutation:

Mutation is a genetic operator that introduces random changes into individual solutions. It is essential for maintaining genetic diversity in the population. In this context, mutation can be used to make small random adjustments to the attributes of a square.

Implementation:
- Random Mutation: You can apply a mutation with a certain probability to each attribute (position, color, opacity) of a square. For example, you can randomly adjust the position coordinates, change the RGB values, or modify the opacity within predefined ranges.

```
def mutate_square(square):
```

```
    if random.random() < mutation_rate:
        # Mutate the position
        square.x = random.randint(0, w)
        square.y = random.randint(0, h)

    if random.random() < mutation_rate:
        # Mutate the color
        square.color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))

    if random.random() < mutation_rate:
        # Mutate the opacity
        square.opacity = random.uniform(0.1, 1.0)
```

3. Selection:

Selection is the process of choosing individuals from the current population to form the next generation. There are various selection strategies you can employ, depending on your objectives.

Implementation:
- Elitism Selection: You can select the top N squares from the current generation based on their fitness scores. This ensures that the best-performing squares survive to the next generation. This strategy promotes the convergence of the genetic algorithm.

```
def select_best_squares(population, N):
    population.sort(key=lambda square: objective_function(square, target_image))
    return population[:N]
```

Conclusion:

In the implementation of the genetic algorithm for generating an image using squares, crossover combines attributes from parent squares to create new squares, mutation introduces random changes to squares, and selection chooses the best squares to form the next generation. The effectiveness of these operators and strategies will influence the algorithm's performance and the quality of the generated image. Experimentation and fine-tuning of these components are essential for achieving the desired results.

Github link: https://github.com/uditaa-garg/21052632_AI.git