Evolutionary Algorithm

Defne Odabaşı

April 28, 2024

Contents

1 The Evalutionary Algorithm

The aim of this task is to create an image made by filled circles, visually similar to a given RGB source image. In this task the 'Girl with a Pearl Earring' by Johannes Vermeer's painting is used as a reference source image as below.

Figure 1: Source Image for the task

2 Experimental Work

For each hyperparameter selection, there are three types of results that are presented: the final best individual's image, fitness plots showing the progress from the first to the 10000th generation and 1000th to 10000th generation, and images of the top individual at every thousandth generation.

2.1 Default Setting

The default hperparameters for this task are as follows:

Table 1: Default Parameters of the Evolutionary Algorithm

Parameter	**Default Value**
Number of Individuals (`num_inds`)	20
Number of Genes (`num_genes`)	50
Tournament Size (`tm_size`)	5
Fraction of Elites (`frac_elites`)	0.2
Fraction of Parents (`frac_parents`)	0.6
Mutation Probability (`mutation_prob`)	0.2
Mutation Type (`mutation_type`)	Guided

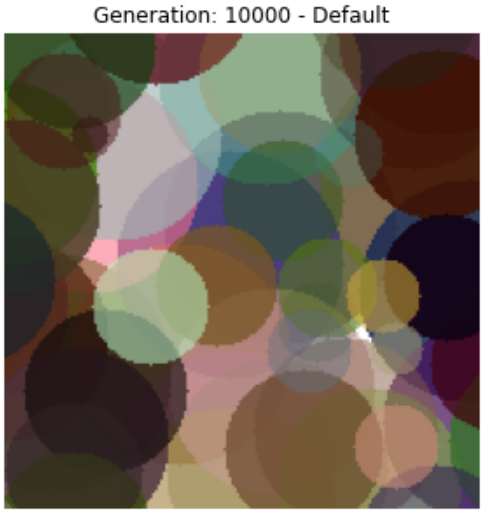

Figure 2: Best Individual for default hyperparameters.

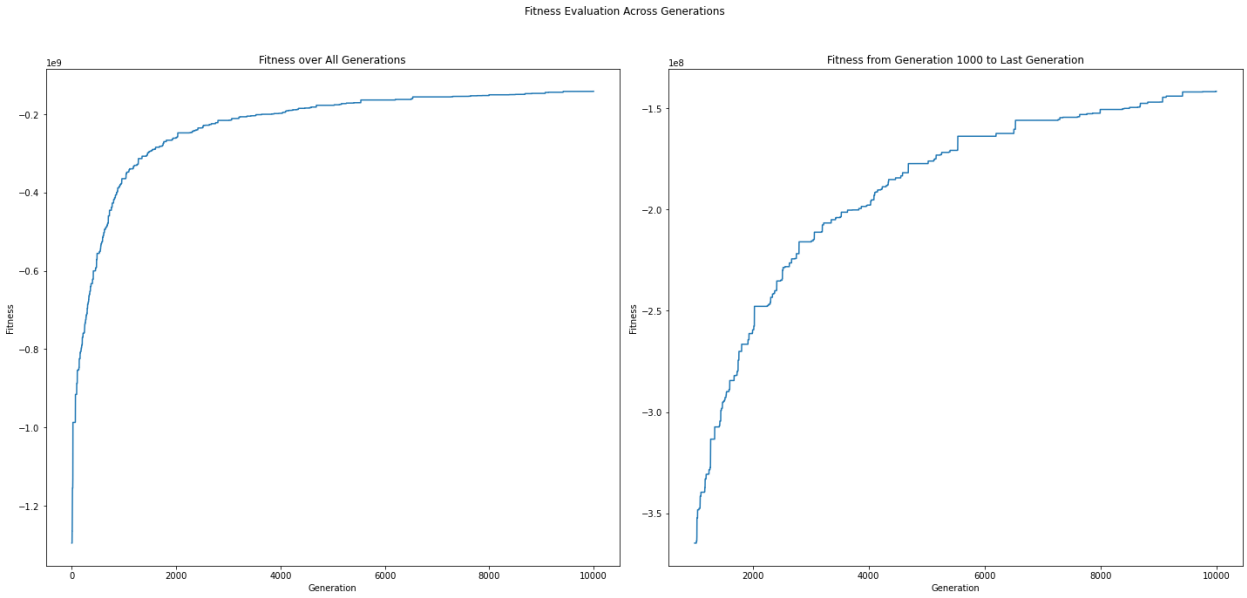

Figure 3: Fitness Plots for default hyperparameters.

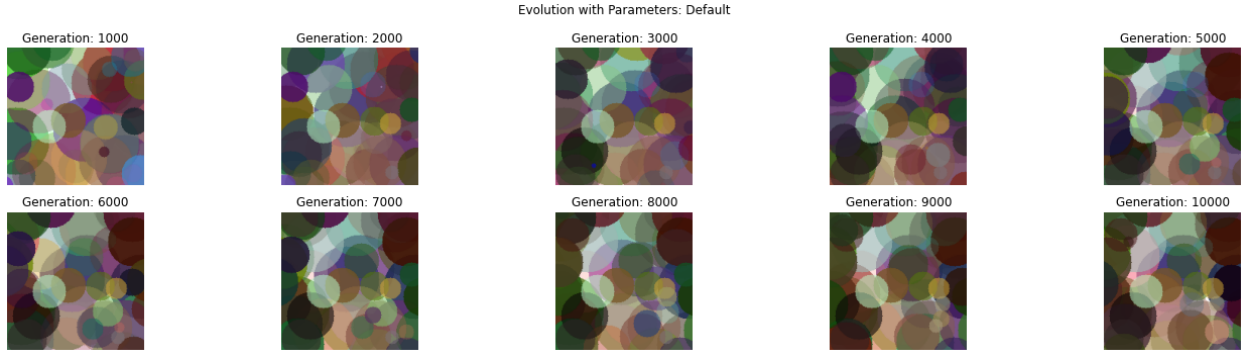

Figure 4: Best Individual in the population at every 1000th generation for default hyperparameters.

Below are the fitness values of the best individual at varios generations under default parameters:

Table 2: Fitness of the Best Individual at Each Generation

Generation	Fitness Value
1000	-364722874
2000	-259409383
3000	-215981769
4000	-197843804
5000	-177350437
6000	-163843827
7000	-155980501
8000	-150616119
9000	-147003470
10000	-141654825

2.2 Number of Individuals (`num_inds`)

The algorithm was evaluated with a discrete set of population sizes, specifically 5, 10, 20 (the designated default), 40, and 60 individuals. The remaining hyperparameters are selected as default.

2.2.1 Number of individuals = 5

The following plots showcase the outcomes for population 5, with all other hyperparameters set to their default values

Figure 5: Best Individual <num_inds> = 5.

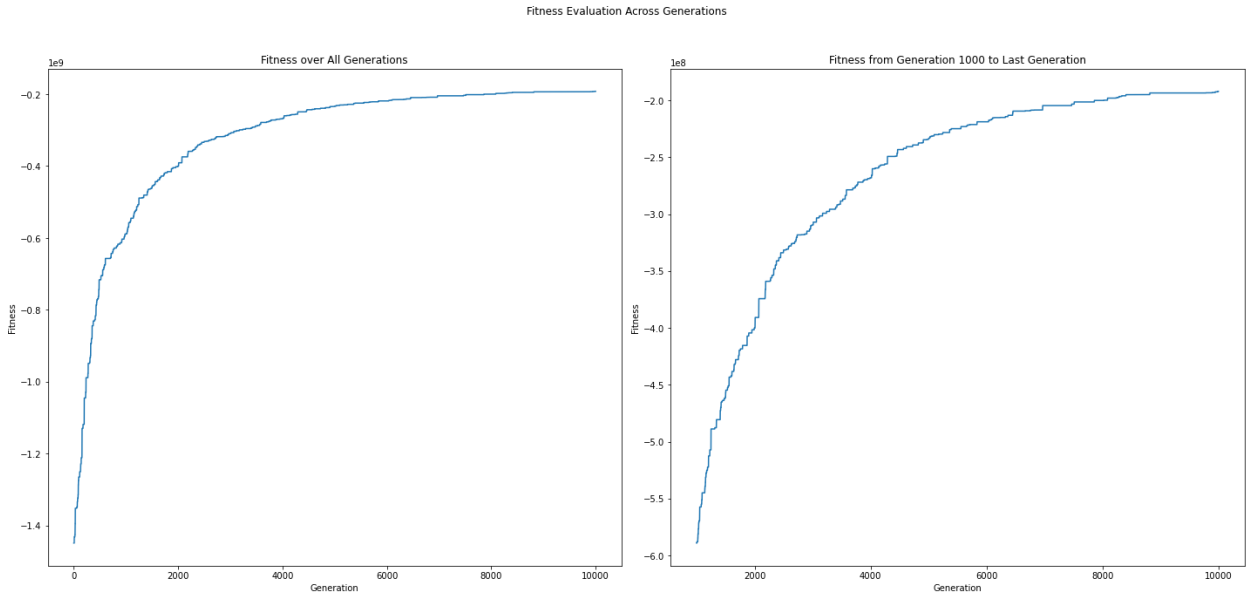

Figure 6: Fitness Plots <num_inds> = 5.

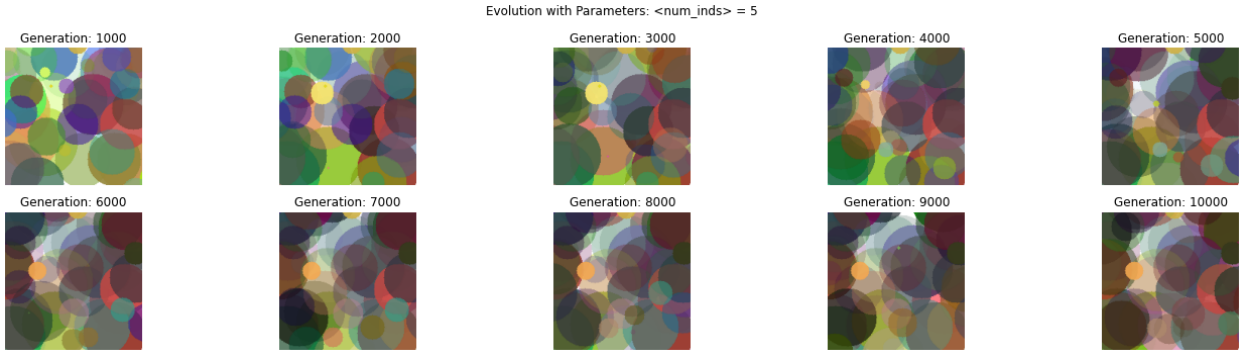

Figure 7: Best Individual in the population at every 1000th generation <num_inds> = 5.

2.2.2 Number of individuals = 10

The following plots showcase the outcomes for population 10, with all other hyperparameters set to their default values.

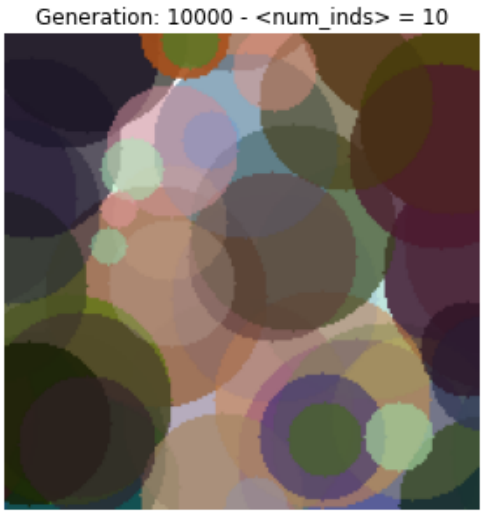

Figure 8: Best Individual <num_inds> = 10.

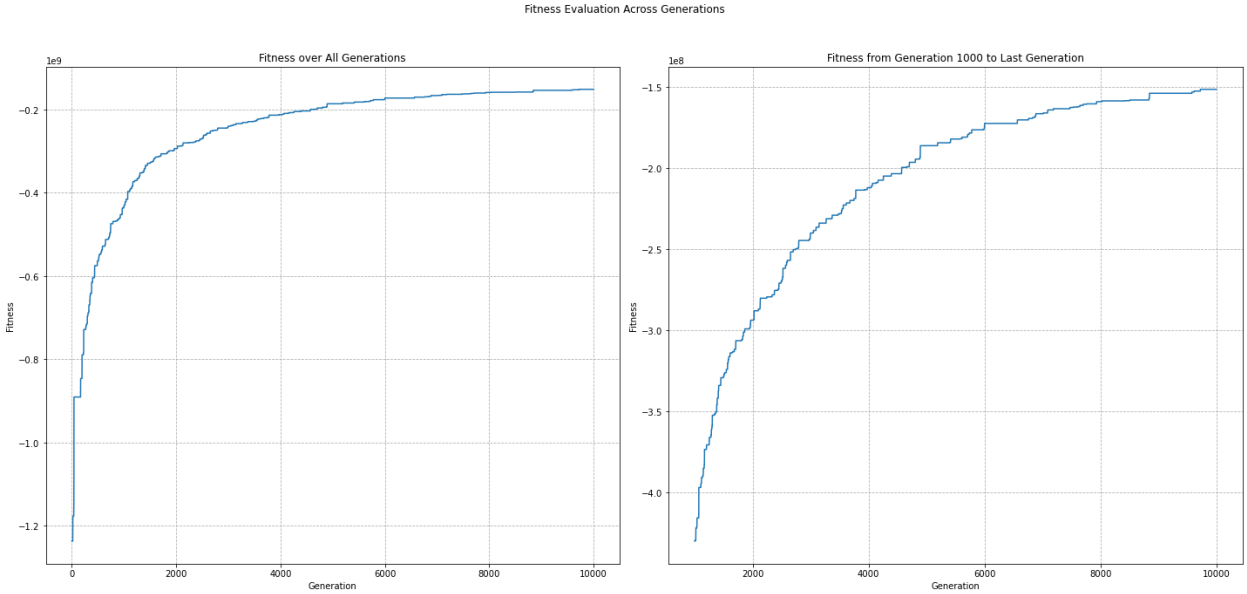

Figure 9: Fitness Plots<num_inds> = 10.

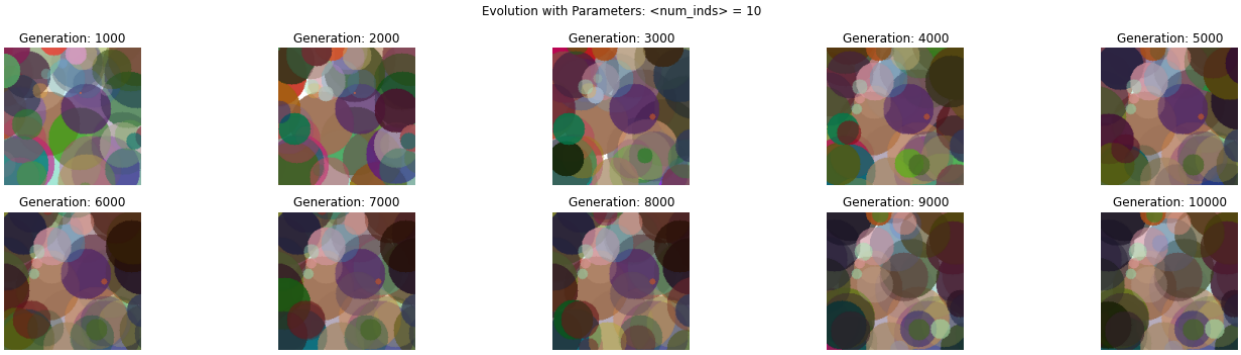

Figure 10: Best Individual in the population at every 1000th generation <num_inds> = 10.

2.2.3 Number of individuals = 40

The following plots showcase the outcomes for population 40, with all other hyperparameters set to their default values.

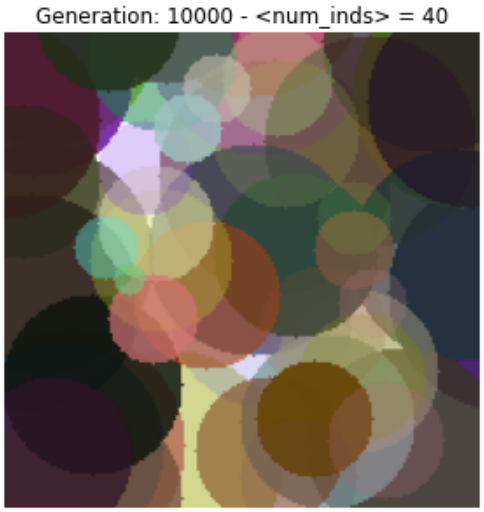

Figure 11: Best Individual <num_inds> = 40.

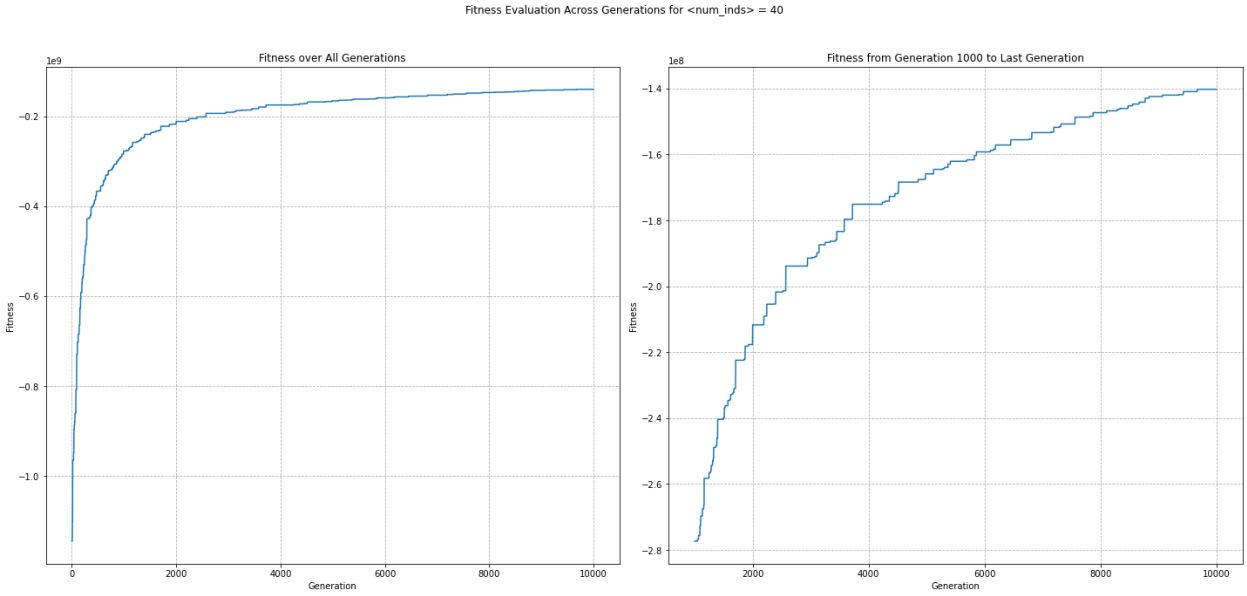

Figure 12: Fitness Plots<num_inds> = 40.

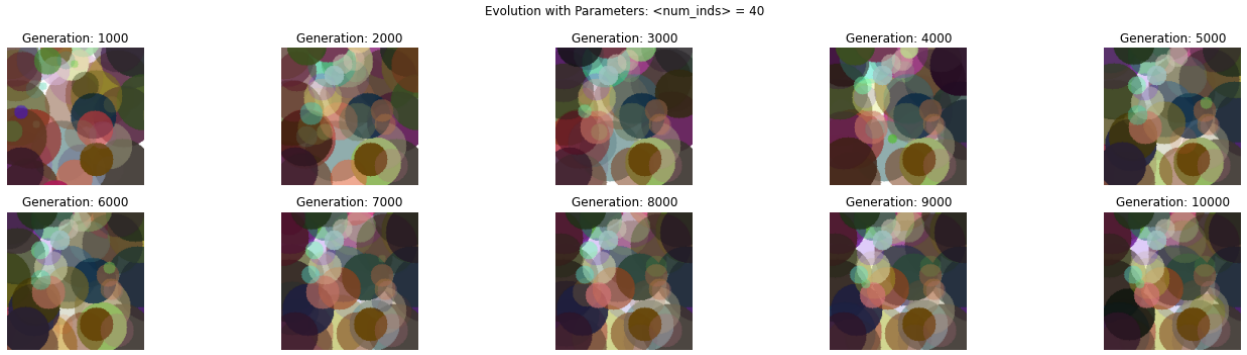

Figure 13: Best Individual in the population at every 1000th generation <num_inds> = 40.

2.2.4 Number of individuals = 60

The following plots showcase the outcomes for population 60, with all other hyperparameters set to their default values.

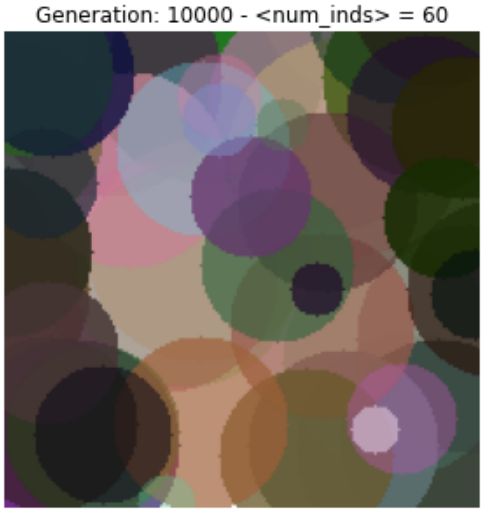

Figure 14: Best Individual <num_inds> = 60.

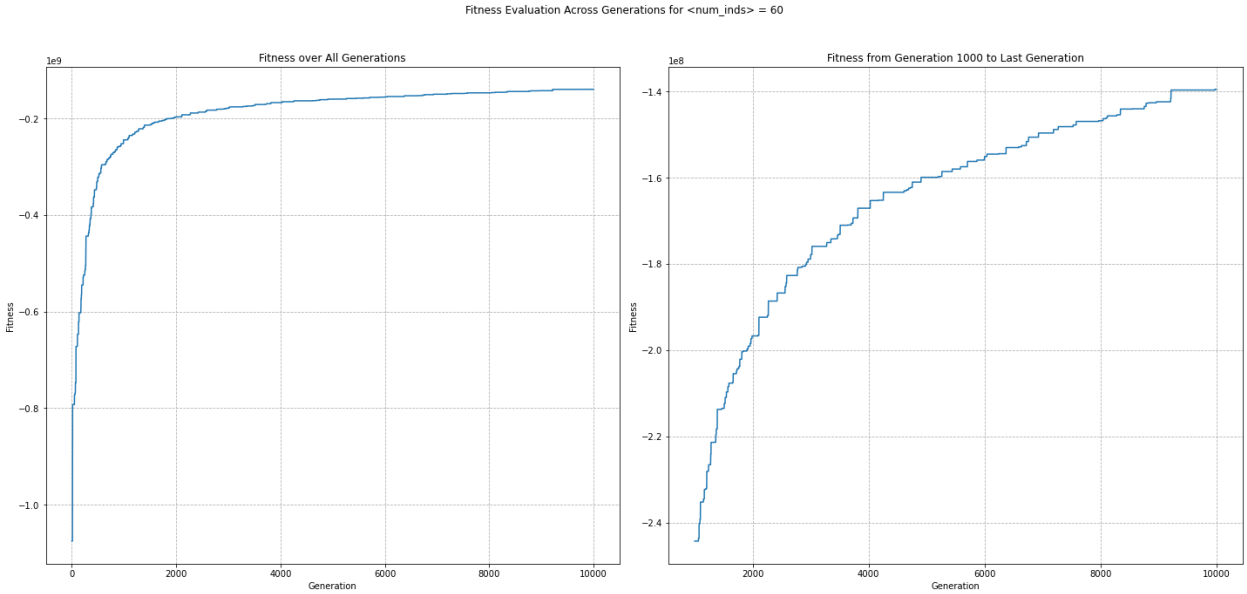

Figure 15: Fitness Plots<num_inds> = 60.

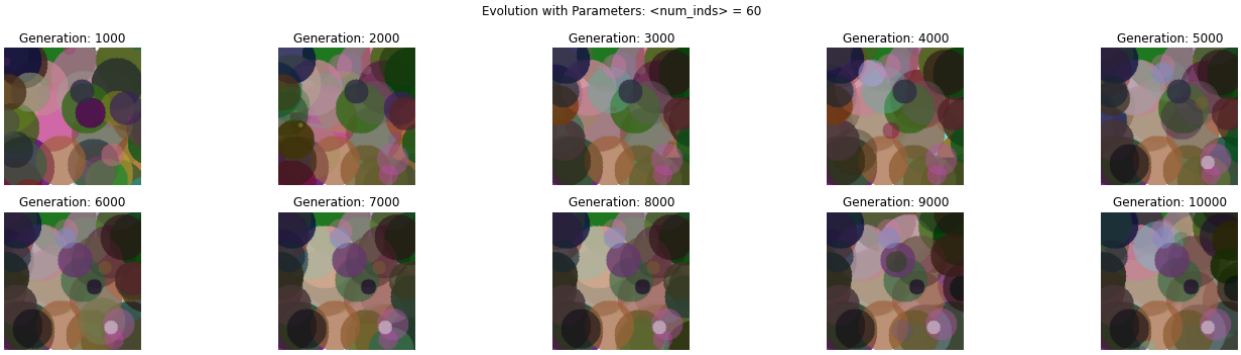

Figure 16: Best Individual in the population at every 1000th generation <num_inds> = 60.

2.2.5 Analysis

The hyperparameter 'number of individuals' is related to the diversity of the population in an evolutionary algorithm. Therefore, as the diversity increases, the algorithm can explore the solution space in a more effective manner.

Table 3: Fitness of the Best Individual at the 10,000th Generation for Different Number of Individuals

Number of Individuals	Fitness at 10,000th Generation
5	-191990701
10	-158380114
20	-141654825
40	-140225647
60	-139517008

As shown in Table 3, increasing the number of individuals results in improved fitness scores at the final generation. This improvement can be attributed to the genetic diversity in bigger populations, which enhances the exploration ability of the algorithm. The risk of convergence to the local maximum is minimized by this diversity. In the Best individual figures provided, we can see that as the number of individuals in the population increased, the best individual image became more and more similar to the source image. Also from the fitness plots, we can observe that algorithm converges faster as the number of individuals increases in the population.

The best performance was observed with 60 individuals in the population, where the fitness at the 10,000th generation reached -139,517,008. Also we can see from the Best individual plot in 14 that the final image is very similar to the source image.

2.3 Number of Genes (num_genes)

The algorithm was evaluated with a discrete set of number of genes, specifically 15, 30, 50 (the designated default), 80, and 120 individuals. The remaining hyperparameters are selected as default.

2.3.1 Number of Genes = 15

The following plots showcase the outcomes for number of genes = 15, with all other hyper-parameters set to their default values

Figure 17: Best Individual <num_genes> = 15.

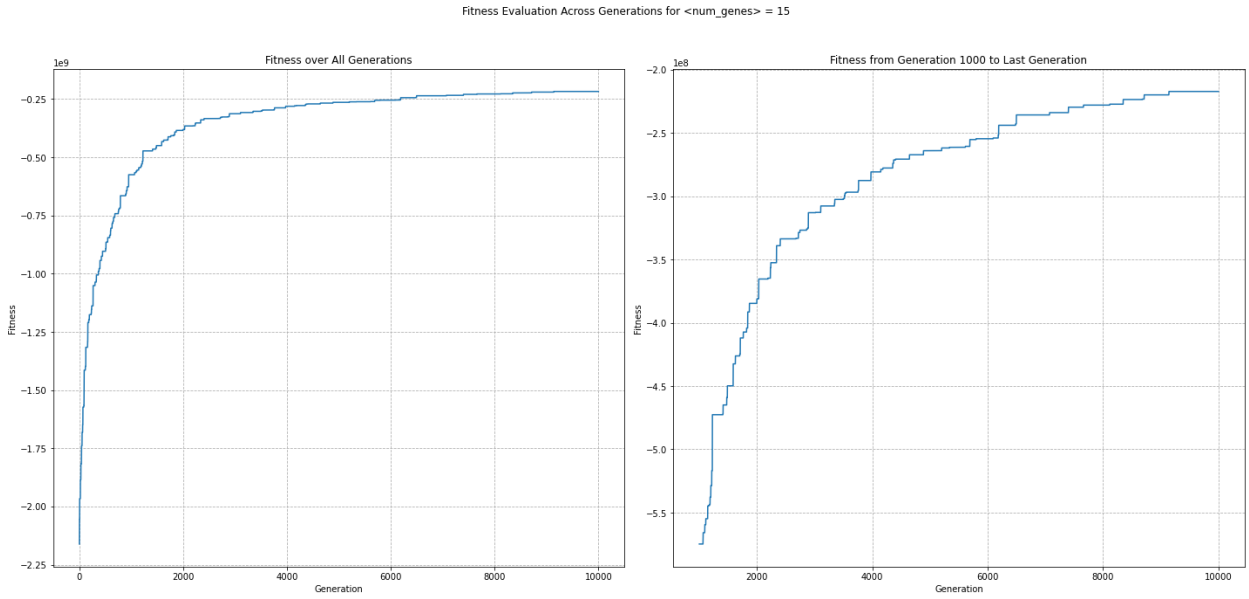

Figure 18: Fitness Plots <num_genes> = 15.

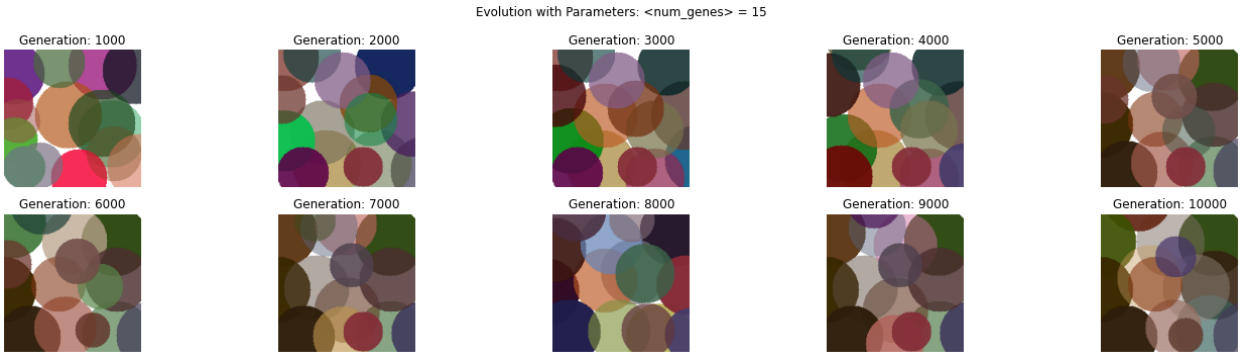

Figure 19: Best Individual in the population at every 1000th generation <num_genes> = 15.

2.3.2 Number of Genes = 30

The following plots showcase the outcomes for number of genes = 30, with all other hyper-parameters set to their default values

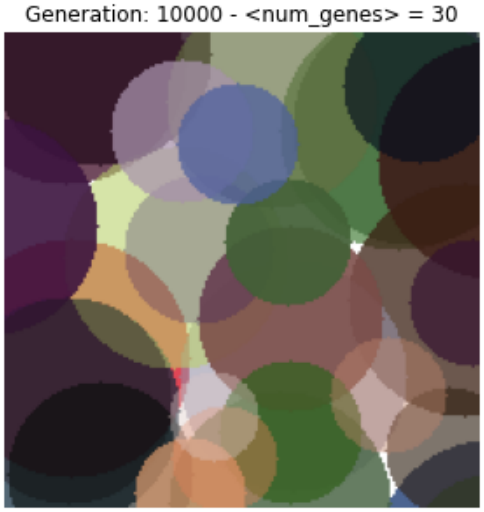

Figure 20: Best Individual <num_genes> = 30.

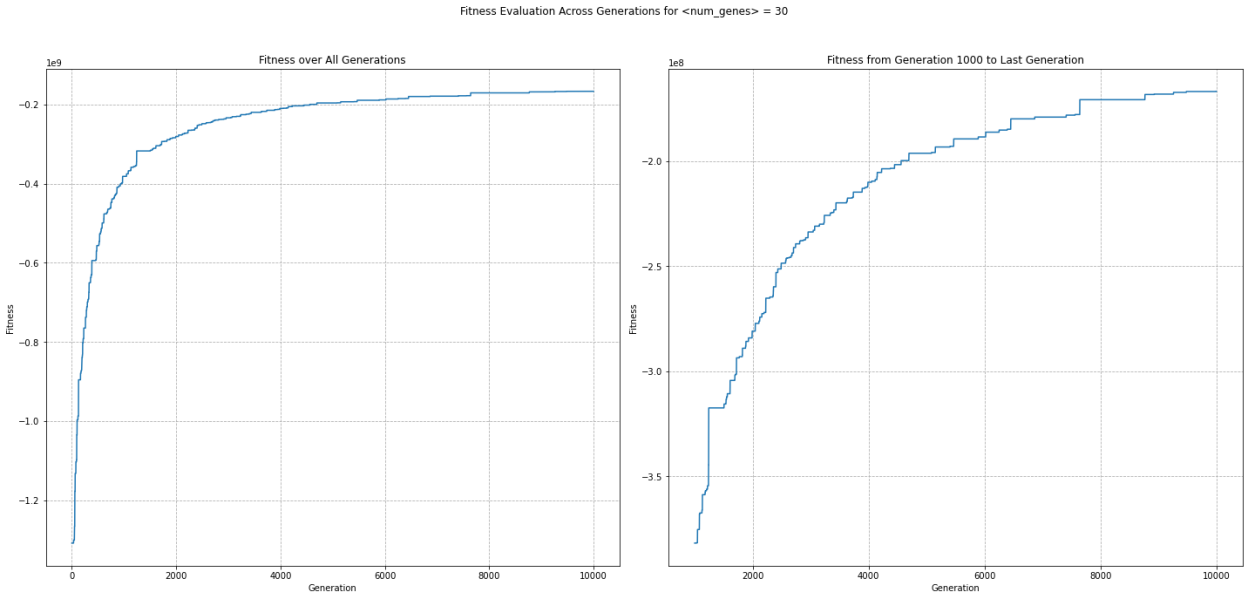

Figure 21: Fitness Plots <num_genes> = 30.

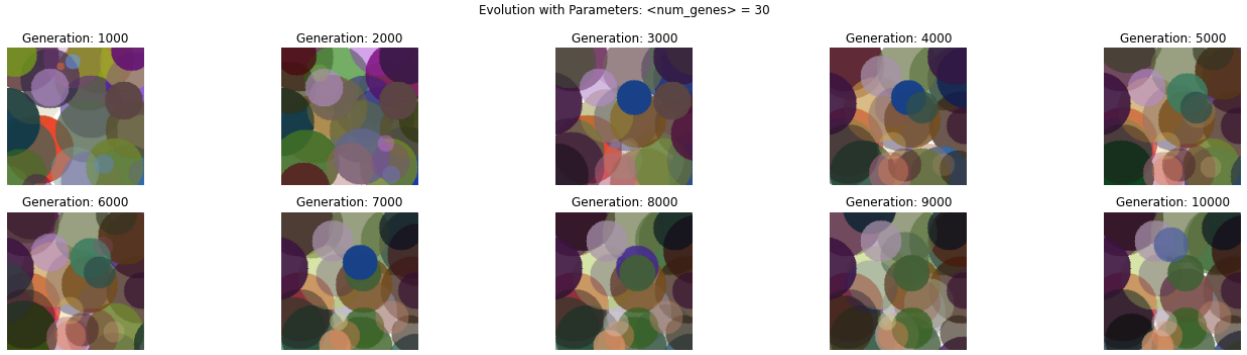

Figure 22: Best Individual in the population at every 1000th generation <num_genes> = 30.

2.3.3 Number of Genes = 80

The following plots showcase the outcomes for number of genes = 80, with all other hyper-parameters set to their default values.

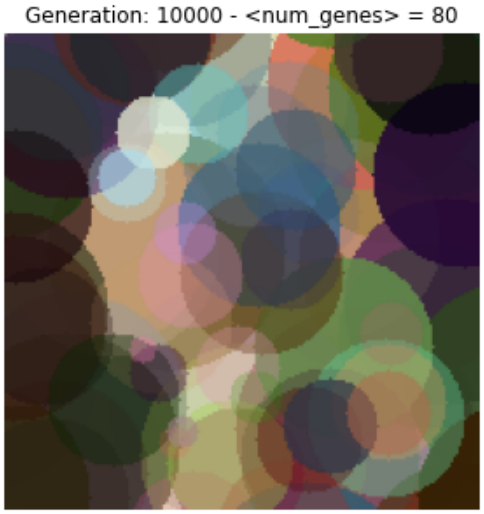

Figure 23: Best Individual <num_genes> = 80.

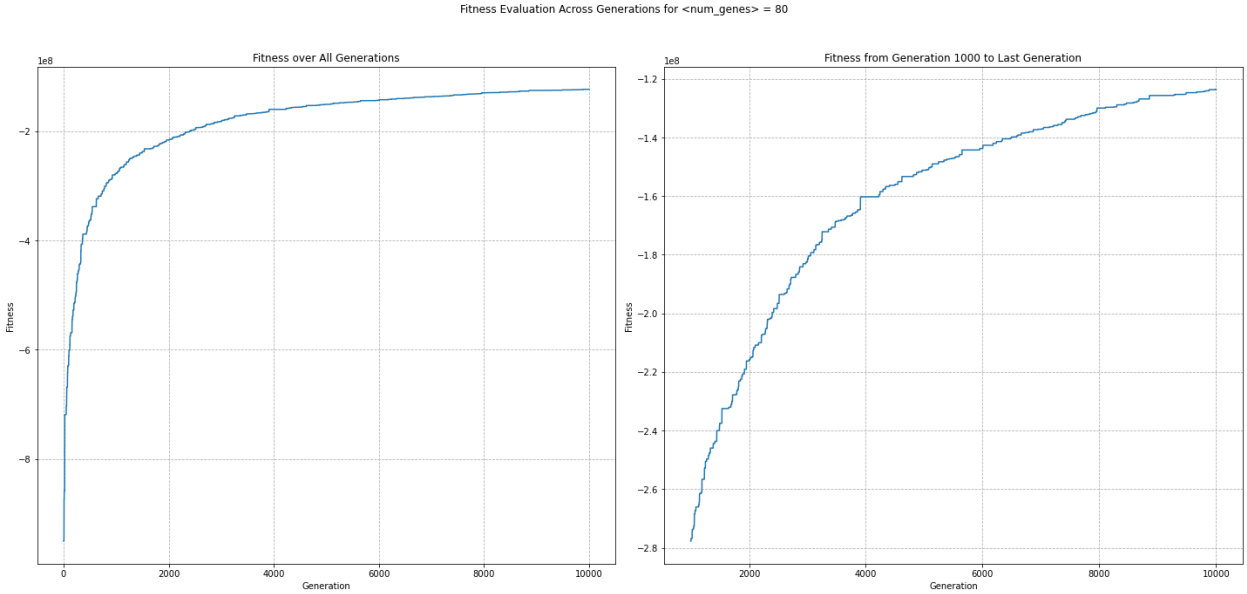

Figure 24: Fitness Plots <num_genes> = 80.

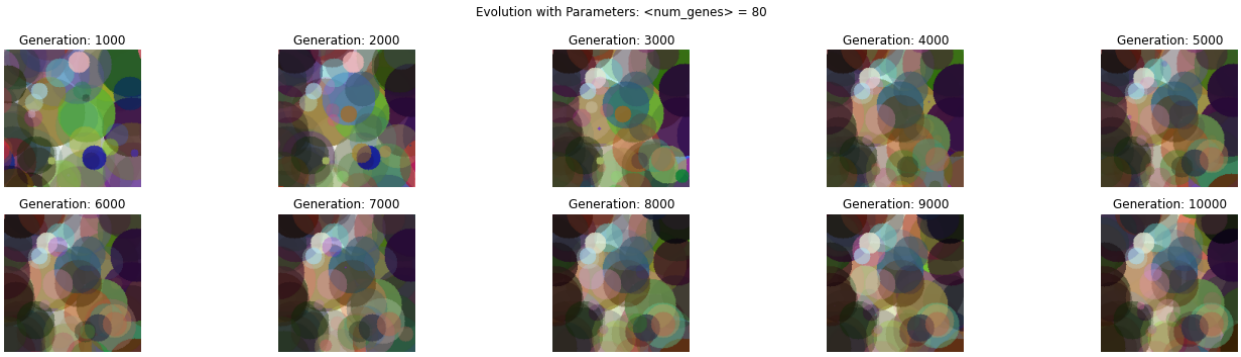

Figure 25: Best Individual in the population at every 1000th generation <num_genes> = 80.

2.3.4 Number of Genes = 120

The following plots showcase the outcomes for number of genes = 120, with all other hyper-parameters set to their default values.

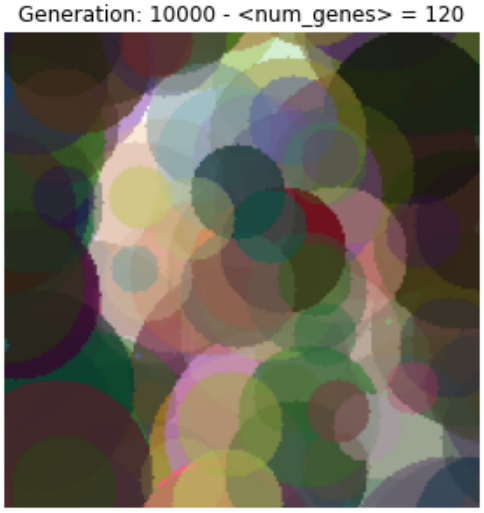

Figure 26: Best Individual <num_genes> = 120.

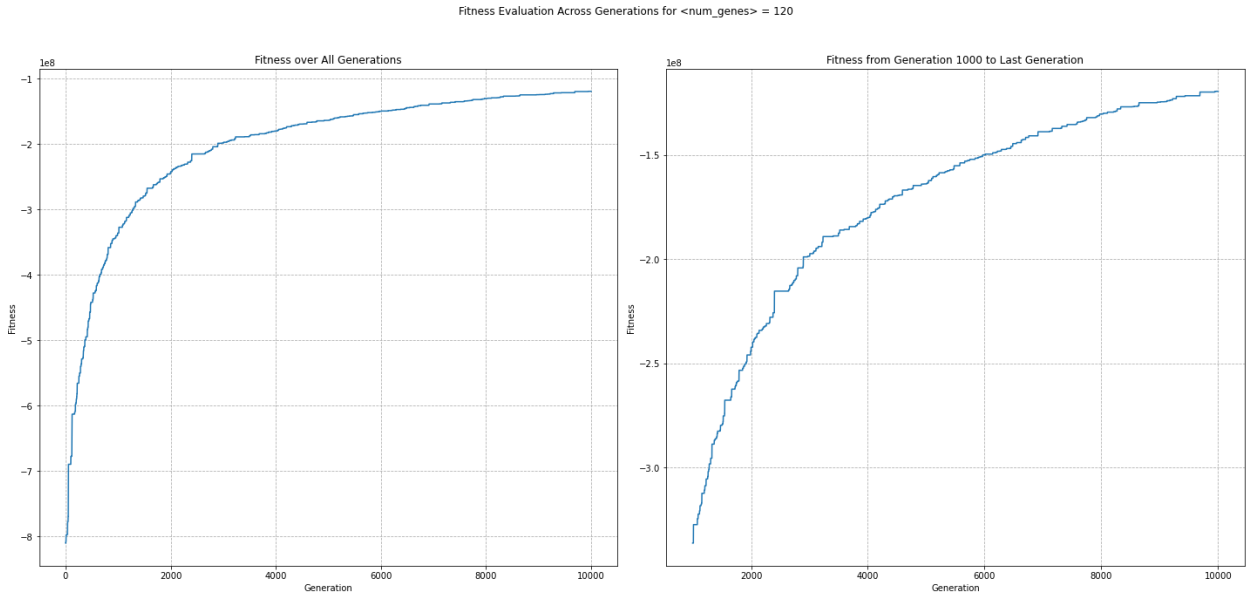

Figure 27: Fitness Plots <num_genes> = 120.

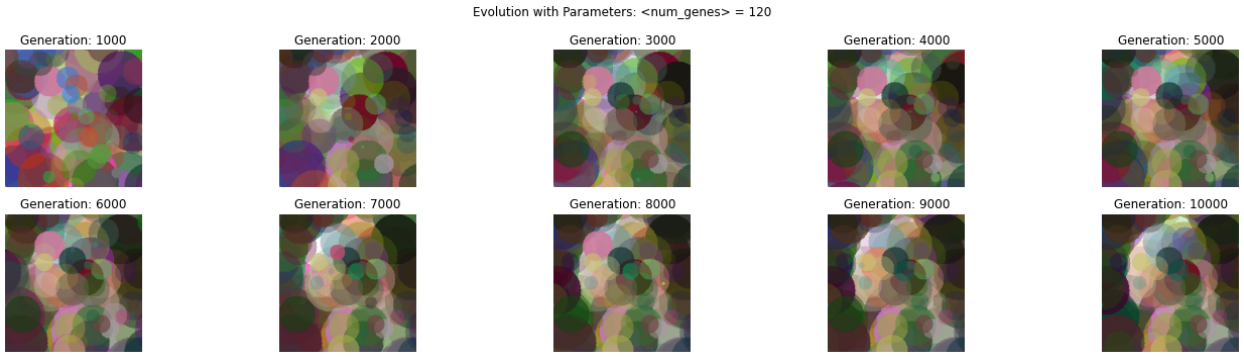

Figure 28: Best Individual in the population at every 1000th generation <num_genes> = 120.

2.3.5 Analysis

The number of genes corresponds to the number of circles that will be used to produce the image. We experimented with different gene counts to assess their impact on the fitness of the best individual by the end of the evolutionary process. The experiments were conducted with gene counts of 15, 30, 80, 50 and 120. The fitness values observed at the 10,000th generation for each setting are as follows:

Table 4: Fitness of the Best Individual at the 10,000th Generation for Different Number of Genes in an Individual

Number of Genes	Fitness at 10,000th Generation
15	-217240750
30	-166797025
50	-141654825
80	-123570798
120	-119496512

As observed in Table 4, increasing the number of genes leads to improved fitness scores. This expected as the number of genes increased, we can represent the image with higher number of circles which enables fine-tuning in the image. In the fitness graphs we can observe that higher number of genes has a faster convergence to at the first 1000th generations. From the 26 of the best individual image at the final generation with 120 genes we can see that the image provides more details.

The best performance was achieved with 120 genes, yielding a fitness score of -119496512 at the 10,000th generation. This indicates that for our problem, a higher number of genes effectively captures the necessary details to optimize the solution further. The only downside of increasing the for this task is that computational time increases.

2.4 Tournament Size (`tm_size`)

The algorithm was evaluated with a discrete set of tournament size, specifically 2, 5(the designated default), 8 and 16 individuals. The remaining hyperparameters are selected as default.

2.4.1 Tournament Size = 2

The following plots showcase the outcomes for tournament size = 2, with all other hyperparameters set to their default values

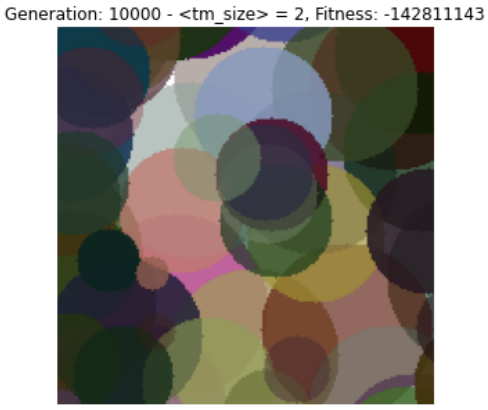

Figure 29: Best Individual `<tm_size>` = 2.

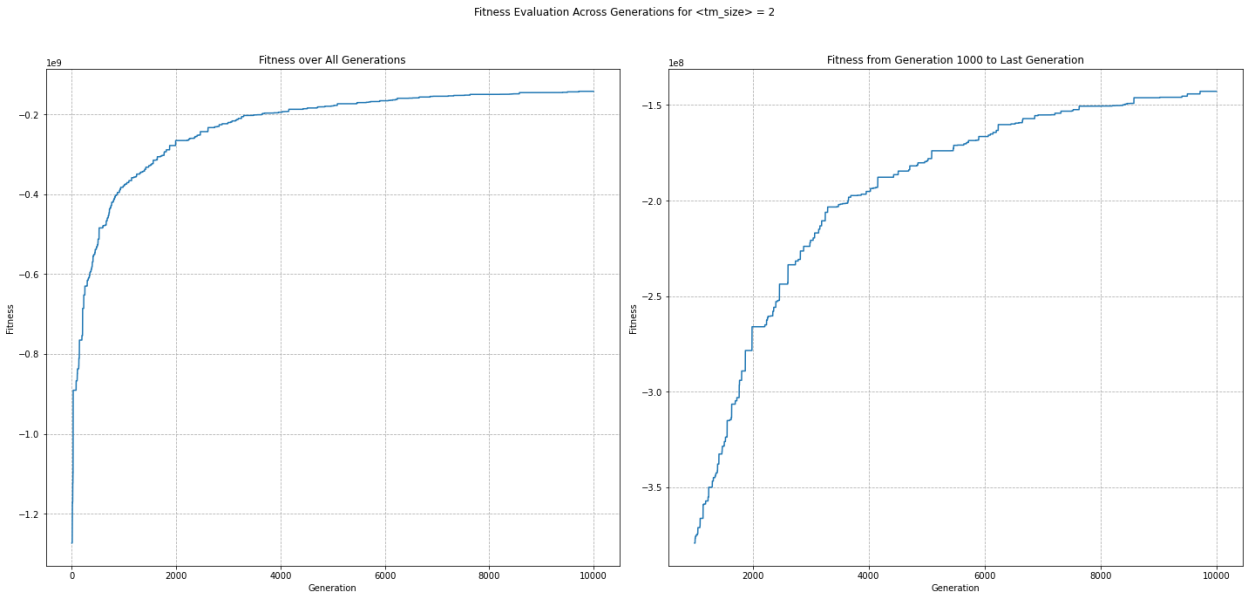

Figure 30: Fitness Plots <tm_size> = 2.

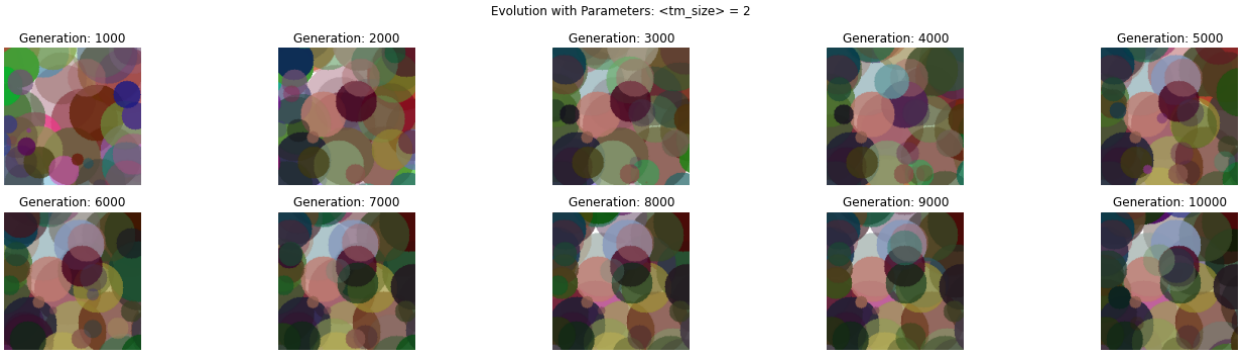

Figure 31: Best Individual in the population at every 1000th generation <tm_size> = 2.

2.4.2 Tournament Size = 8

The following plots showcase the outcomes for tournament size = 8, with all other hyperparameters set to their default values

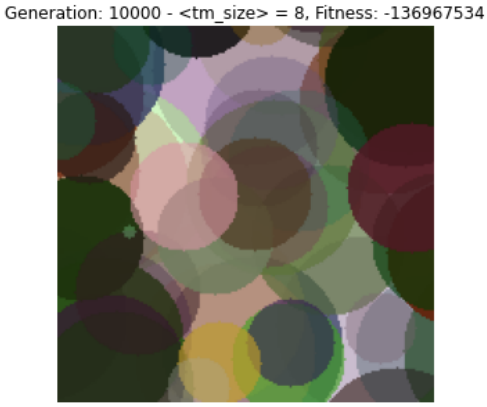

Figure 32: Best Individual <tm_size> = 8.

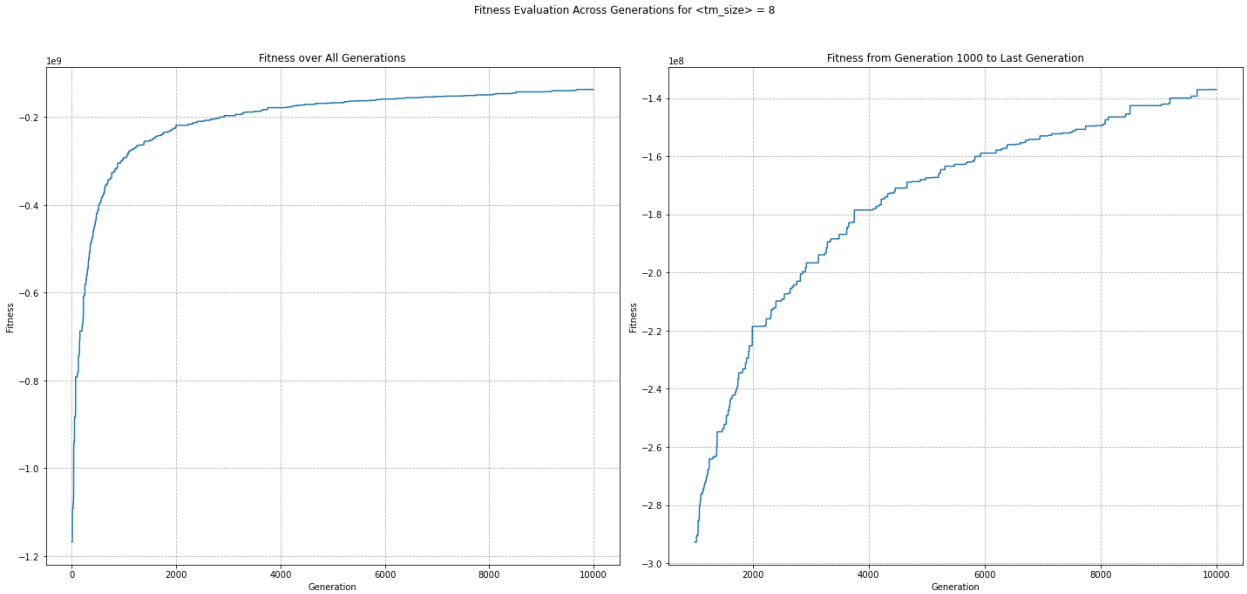

Figure 33: Fitness Plots <tm_size> = 8.

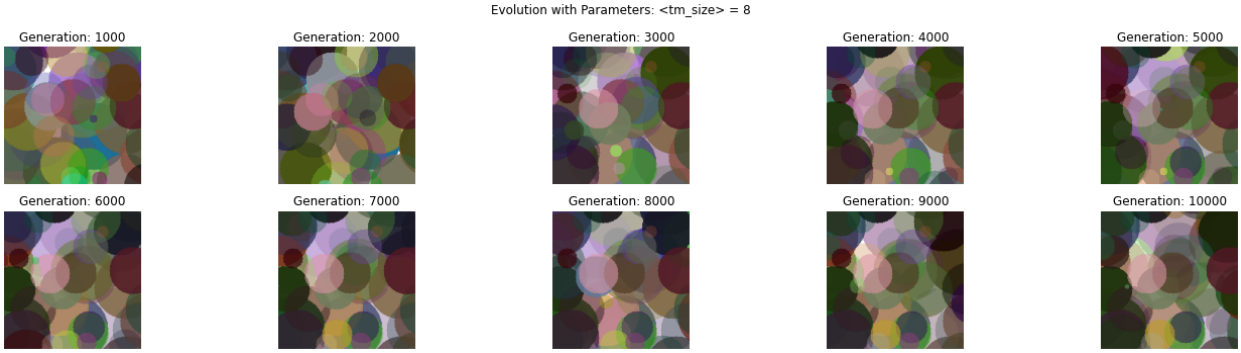

Figure 34: Best Individual in the population at every 1000th generation <tm_size> = 8.

2.4.3 Tournament Size = 16

The following plots showcase the outcomes for tournament size = 16, with all other hyper-parameters set to their default values

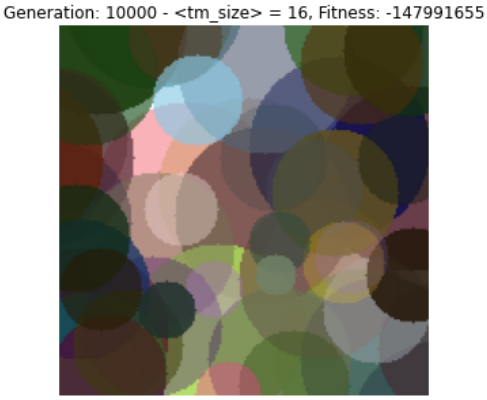

Figure 35: Best Individual <tm_size> = 16.

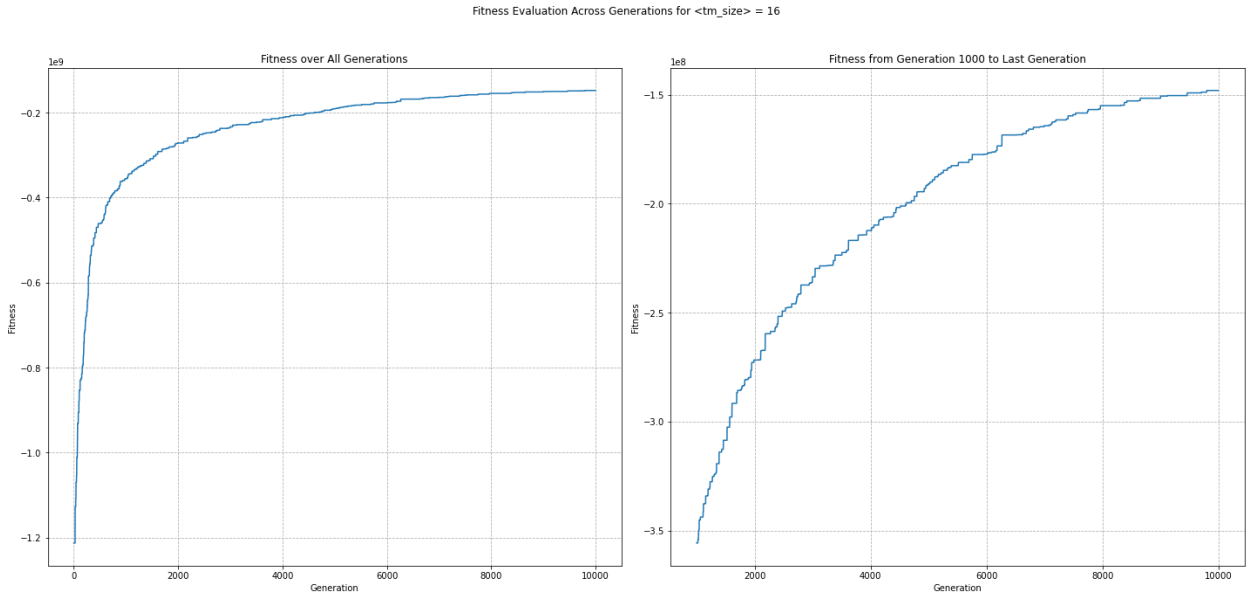

Figure 36: Fitness Plots <tm_size> = 16.

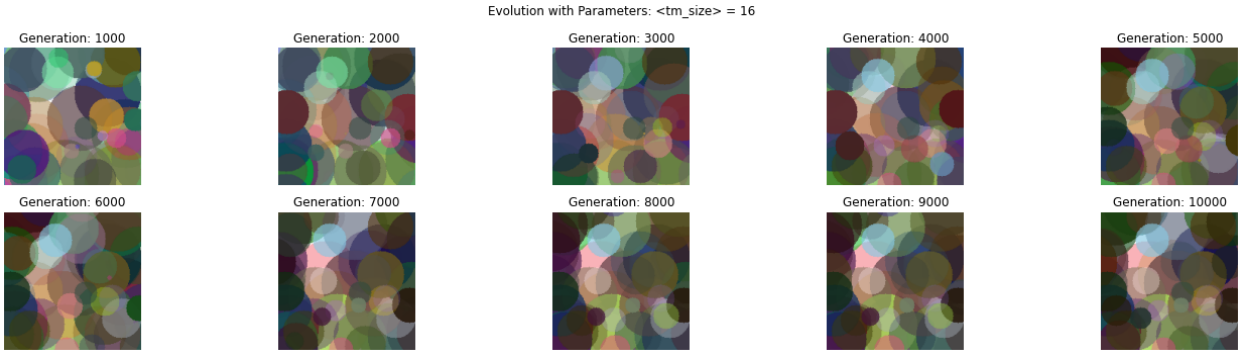

Figure 37: Best Individual in the population at every 1000th generation <tm_size> = 16.

2.4.4 Analysis

In the tournament selection ,which is performed on the individuals other than elites, sample of individuals are drawn and they carry out a tournament with each other. The winner of the tournament is the one with the highest fitness value. This way the next generation is obtained. In my adaptation for the evaluation algorithm the winner of the tournament can still participate in the remaining tournaments. Therefore the same individual may win multiple tournaments.

Table 5: Fitness of the Best Individual at the 10,000th Generation for Different Tournament Sizes

Tournament Sizes	**Fitness at 10,000th Generation**
2	-142811143
5	-141654825
8	-136967534
16	-147991655

As shown in Table 5, the least favorable fitness outcome occurs with a tournament size of 16. This outcome aligns with expectations, given the population size of 20 individuals, a default elite fraction of 0.2, and thus four elites per generation (20×0.2). The remaining 16 individuals undergo tournament selection, which reduces diversity within the population and leads to convergence towards a local optimum. Similarly, a tournament size of 2 is not optimal as it increases the likelihood of less fit individuals progressing to the next generation. Analysis of my algorithm indicates that a tournament size of 8 yields the best results for this algorithm.

2.5 Fraction of Elites (`frac_elites`)

The algorithm was evaluated with various fraction of elites, specifically 0.04, 0.2 (the designated default) and 0.35. The remaining hyperparameters are selected as default.

2.5.1 Fraction of Elites = 0.04

The following plots showcase the outcomes for fraction of elites = 0.04, with all other hyperparameters set to their default values

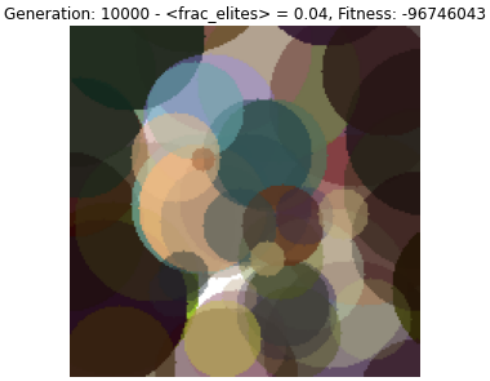

Figure 38: Best Individual `<frac_elites>` = 0.04.

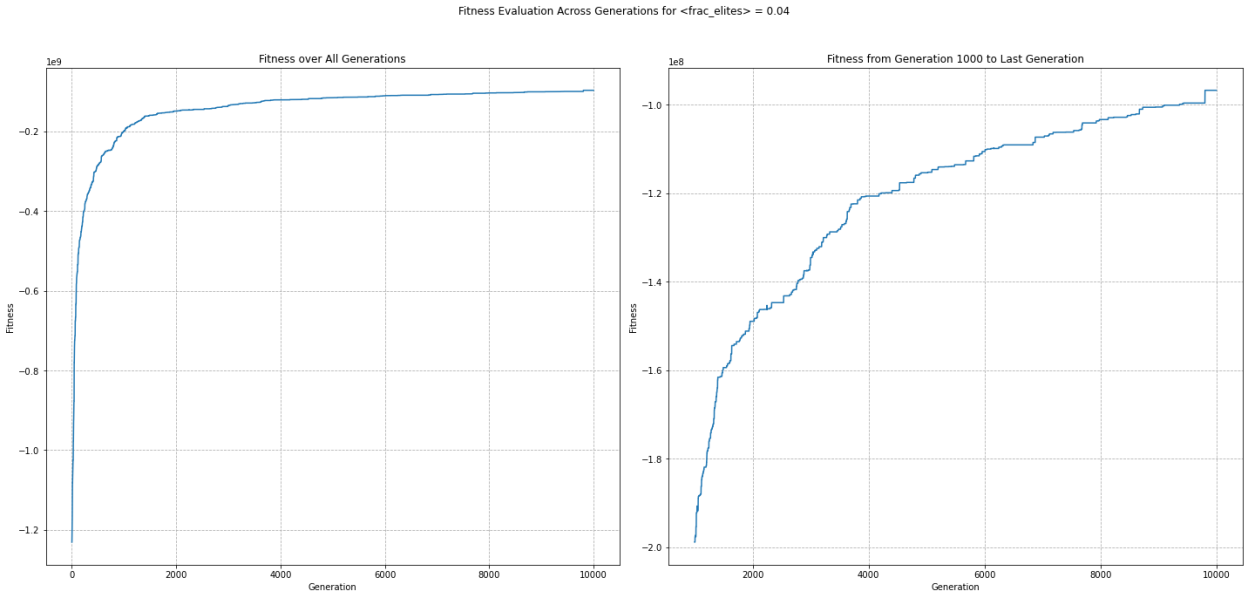

Figure 39: Fitness Plots `<frac_elites>` = 0.04.

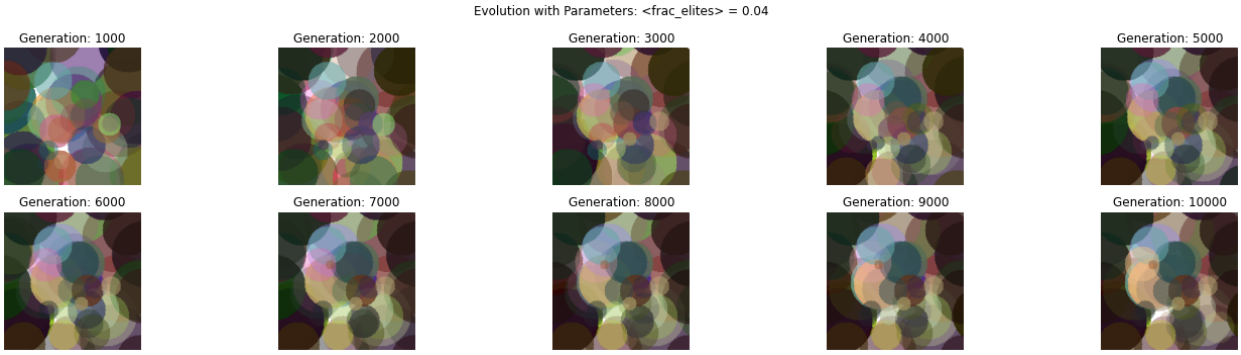

Figure 40: Best Individual in the population at every 1000th generation <frac_elites> = 0.04.

2.5.2 Fraction of Elites = 0.35

The following plots showcase the outcomes for fraction of elites = 0.35, with all other hyperparameters set to their default values

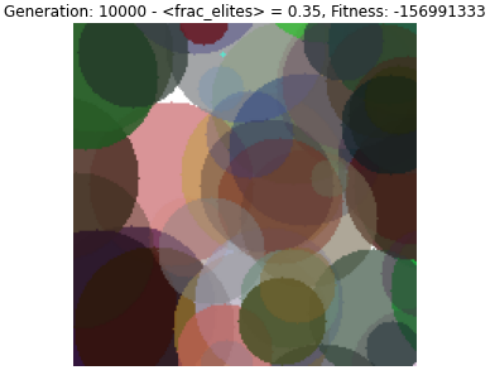

Figure 41: Best Individual <frac_elites> = 0.35.

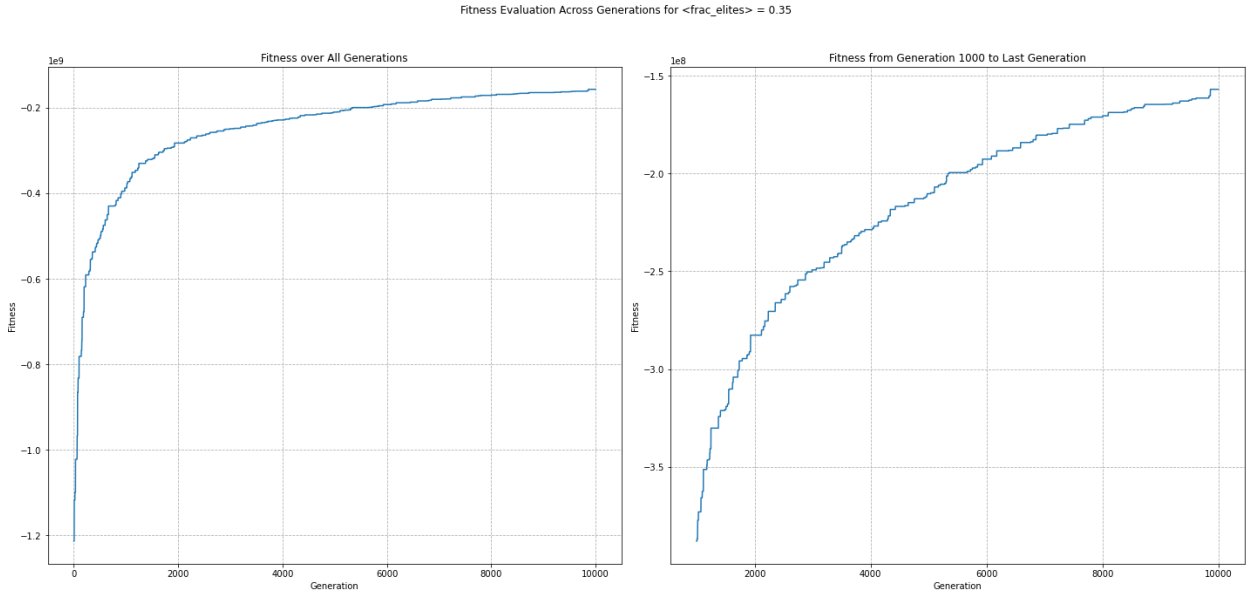

Figure 42: Fitness Plots <frac_elites> = 0.35.

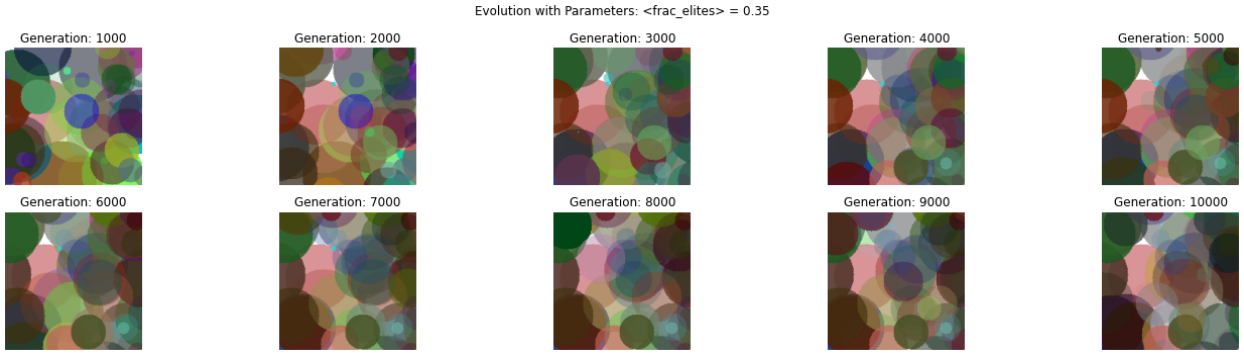

Figure 43: Best Individual in the population at every 1000th generation <frac_elites> = 0.35.

2.5.3 Analysis

The ‘number of elites‘ hyperparameter determines the fraction of the best fitness valued individuals that are preserved unchanged from one generation to the next. This ensures that the best solutions are carried over. The fraction of elites is important because it may risk diversity if it is set too high.

Table 6: Fitness of the Best Individual at the 10,000th Generation for Different Fraction of Elites in the Population

Fraction of Elites	Fitness at 10,000th Generation
0.04	-96746043
0.2	-141654825
0.35	-156991333

Table 6 shows the fitness values at the 10,000th generation for different elite fractions. A lower fraction of elites which is 0.04 resulted in best performance for the fitness not only among the other fraction of elites values but also among all the hyperparameters. It can be seen in Figure 38 that the final output image is very detailed including the white neck and eyes. The reason for such a high fitness score for this task can be explained by diversity. When the number of elites is 0, all the population is subjected to tournament selection. After the best individuals become parent and the diversity increases among the population. In this case instead of higher selective pressure, wider deviation of the individuals over the search space is preferred. Additionally, from the Fitness plots specifically for 39, we can see that convergence occurred very fast at the first 1000th generation.

2.6 Fraction of Parents (`frac_parents`)

The algorithm was evaluated with variuos fraction of parents, specifically 0.15, 0.3, 0.6 (the designated default) and 0.75. The remaining hyperparameters are selected as default.

2.6.1 Fraction of Parents = 0.15

The following plots showcase the outcomes for fraction of parents = 0.15, with all other hyperparameters set to their default values

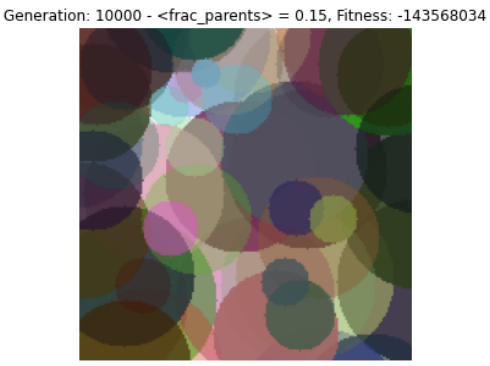

Figure 44: Best Individual <frac_parents> = 0.15.

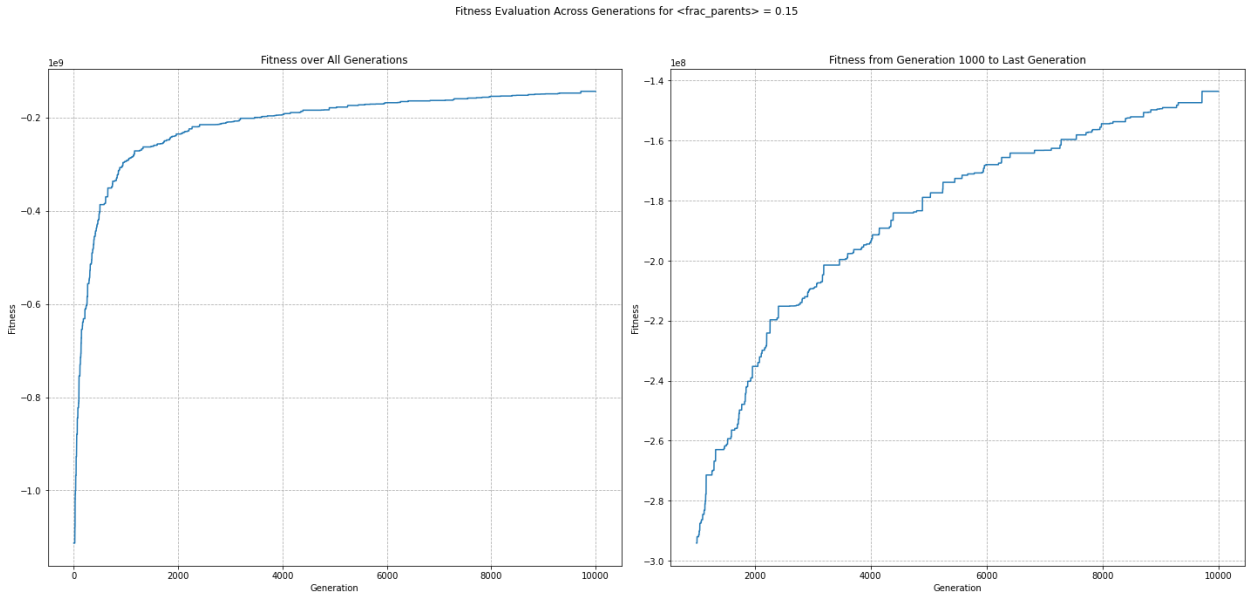

Figure 45: Fitness Plots <frac_parents> = 0.15.

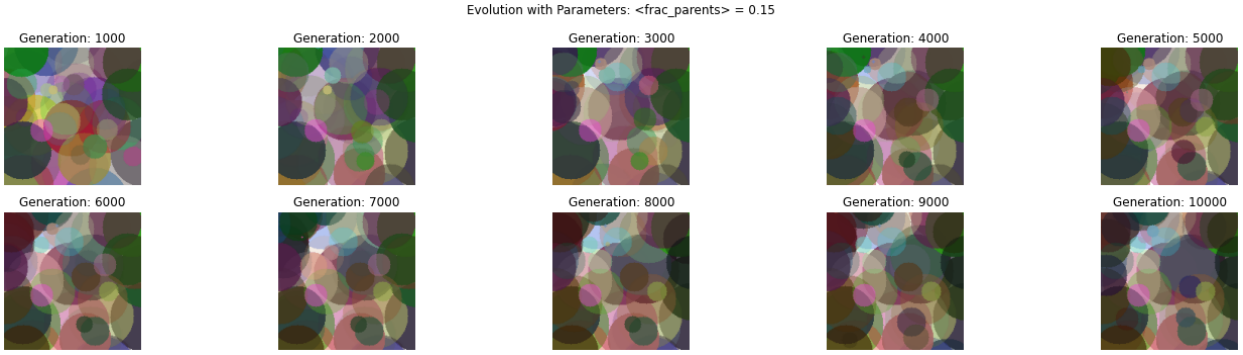

Figure 46: Best Individual in the population at every 1000th generation <frac_parents> = 0.15.

2.6.2 Fraction of Parents = 0.3

The following plots showcase the outcomes for fraction of parents = 0.3, with all other hyperparameters set to their default values

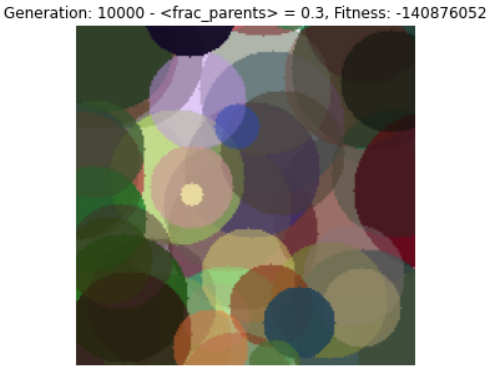

Figure 47: Best Individual <frac_parents> = 0.3.

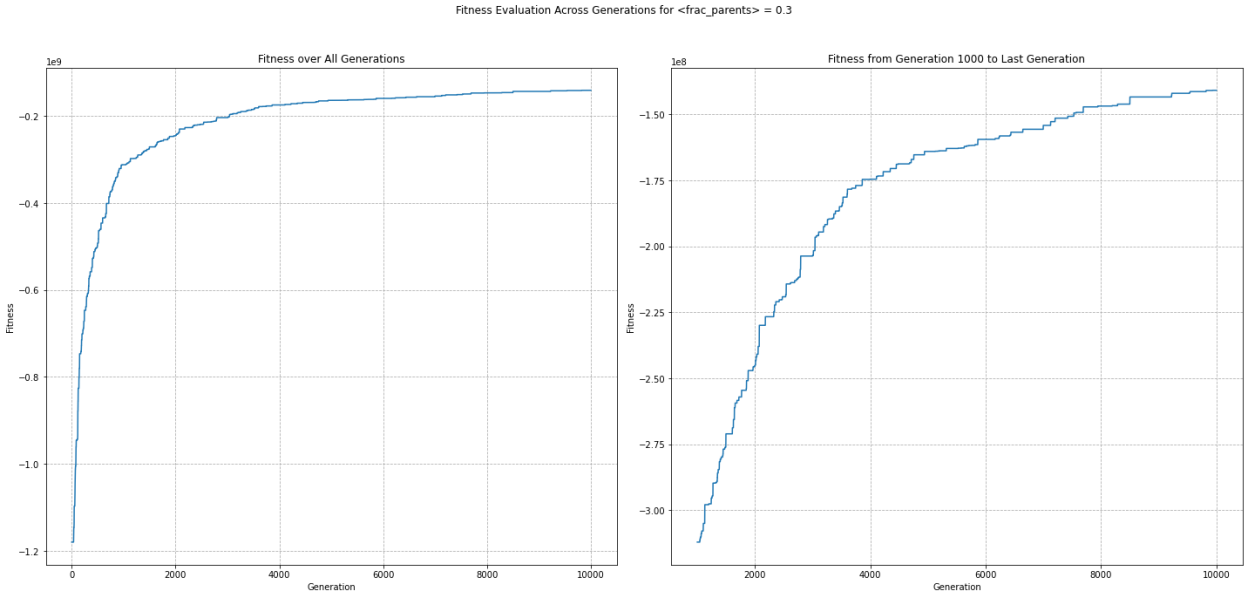

Figure 48: Fitness Plots <frac_parents> = 0.3.

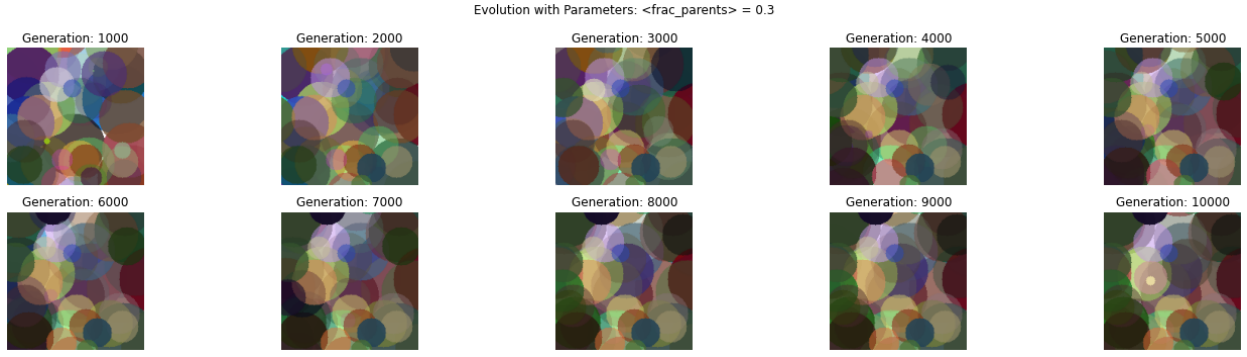

Figure 49: Best Individual in the population at every 1000th generation <frac_parents> = 0.3.

2.6.3 Fraction of Parents = 0.75

The following plots showcase the outcomes for fraction of parents = 0.75, with all other hyperparameters set to their default values

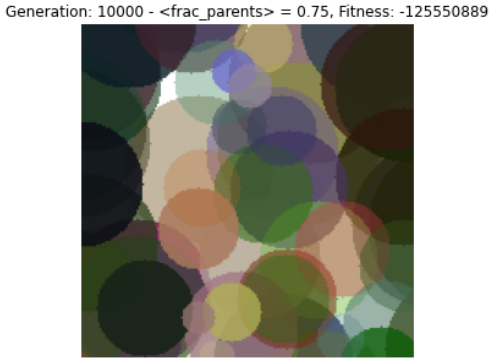

Figure 50: Best Individual <frac_parents> = 0.75.

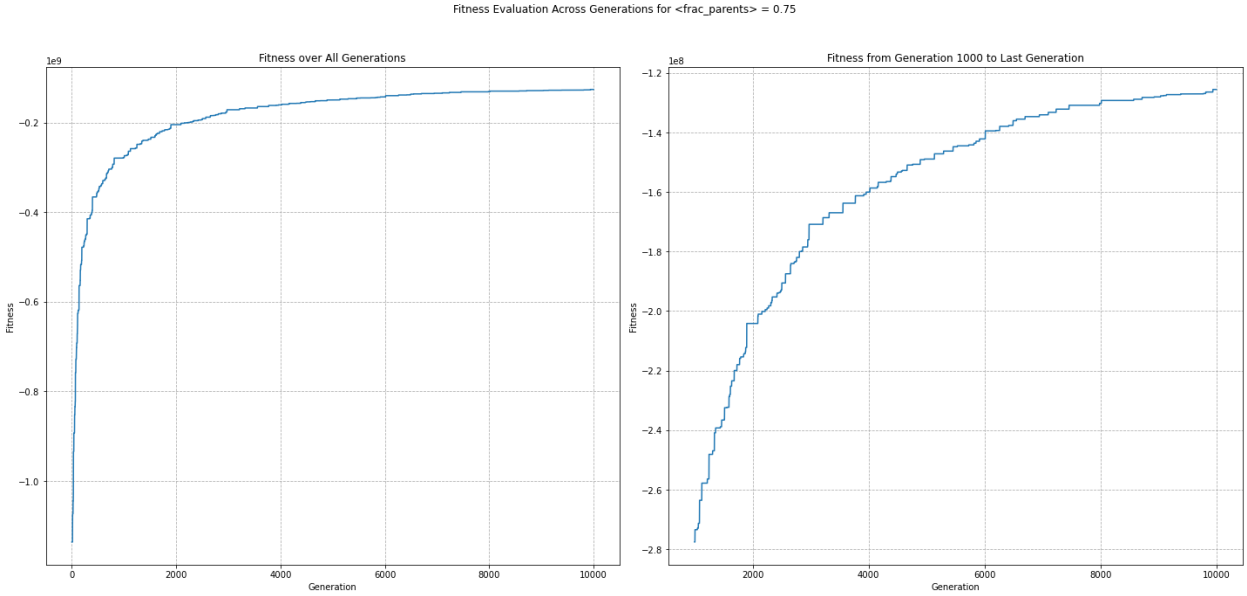

Figure 51: Fitness Plots <frac_parents> = 0.75.

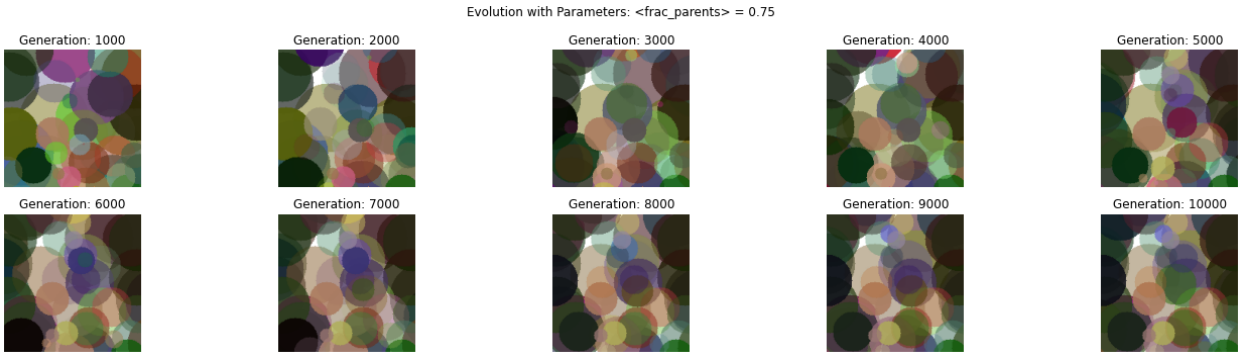

Figure 52: Best Individual in the population at every 1000th generation <frac_parents> = 0.75.

2.6.4 Analysis

The 'fraction of parents' hyperparameter specifies the proportion of the population selected to participate in generating offspring. This selection is critical as it influences the genetic diversity of the next generation and the algorithm's ability to explore new areas of the solution space effectively.

Table 7: Fitness of the Best Individual at the 10,000th Generation for Different Fraction of Parents in the Population

Fraction of Parents	**Fitness at 10,000th Generation**
0.15	-143568034
0.3	-140876052
0.6	-141654825
0.75	-125550889

As depicted in Table 7, different settings for the fraction of parents show varying impacts on fitness. We can see that low fraction of parents provide lower fitness scores due to the lack of diversity. On the plots the similarity is low for lower parent fractions. For the high fraction parents (0.75), the obtained fitness score is high but high fraction of parents can be risky for later generations. If the algorithm has been runed for high fraction of parents multiple times the obtained fitness could be low. This is because in the algorithm parents are removed from the next generation and offsprings take their place. Therefore, individuals with high fitness may replaced by individuals with low fitness score in later generations.

2.7 Mutation Probability (`mutation_prob`)

The algorithm was evaluated with various mutation probabilities, specifically 0.1, 0.2 (the designated default), 0.4 and 0.75. The remaining hyperparameters are selected as default.

2.7.1 Mutation Probability = 0.1

The following plots showcase the outcomes for mutation probability = 0.1, with all other hyperparameters set to their default values

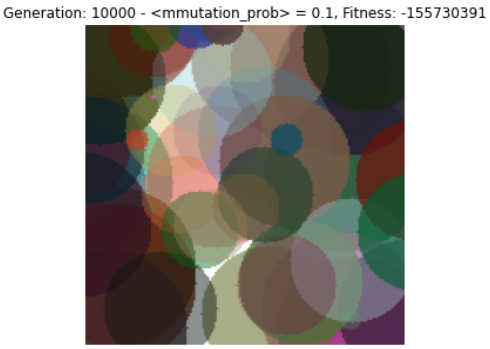

Figure 53: Best Individual <mutation_prob> = 0.1.

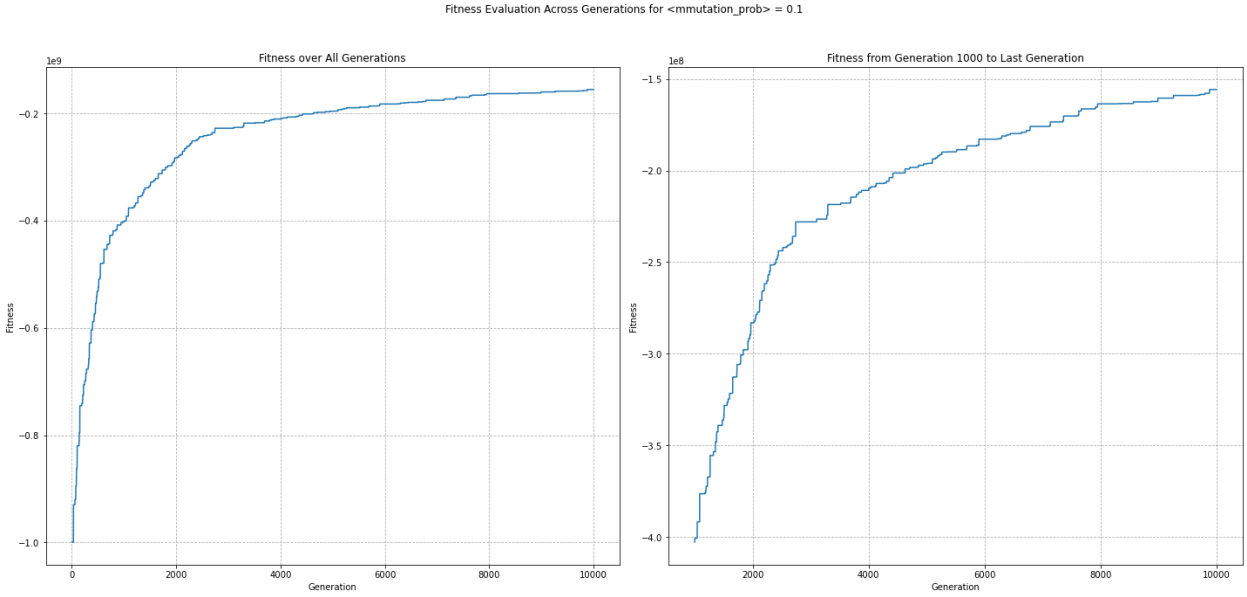

Figure 54: Fitness Plots <mutation_prob> = 0.1.

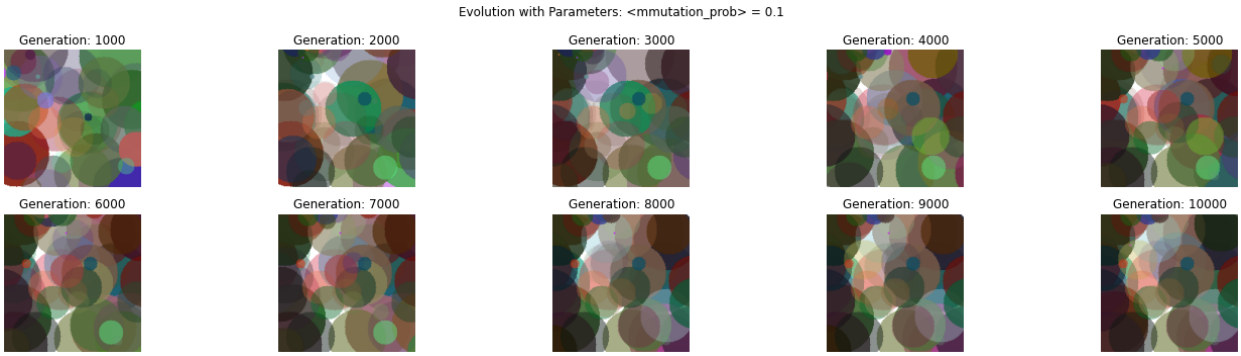

Figure 55: Best Individual in the population at every 1000th generation <mutation_prob> = 0.1.

2.7.2 Mutation Probability = 0.4

The following plots showcase the outcomes for mutation probability = 0.4, with all other hyperparameters set to their default values

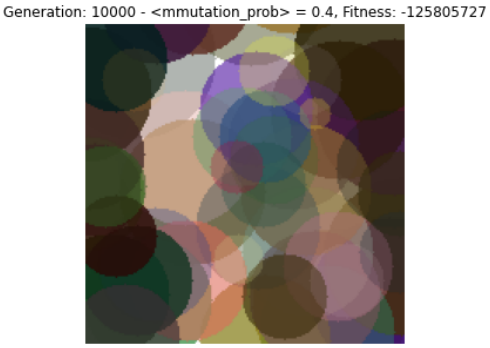

Figure 56: Best Individual <mutation_prob> = 0.4.

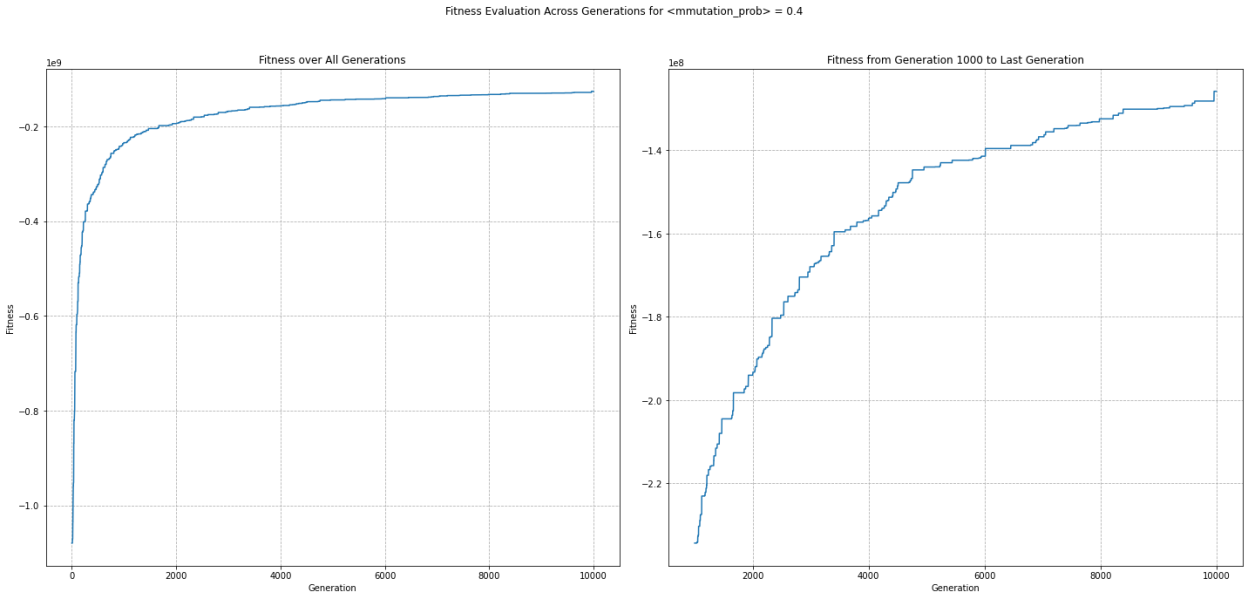

Figure 57: Fitness Plots <mutation_prob> = 0.4.

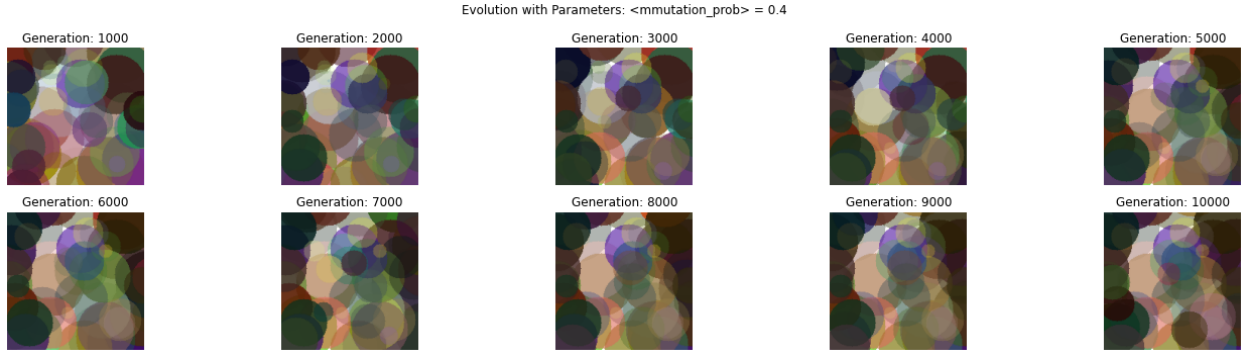

Figure 58: Best Individual in the population at every 1000th generation <mutation_prob> = 0.4.

2.7.3 Mutation Probability = 0.75

The following plots showcase the outcomes for mutation probability = 0.75, with all other hyperparameters set to their default values

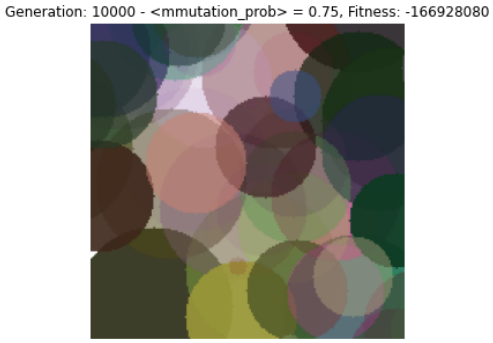

Figure 59: Best Individual <mutation_prob> = 0.75.

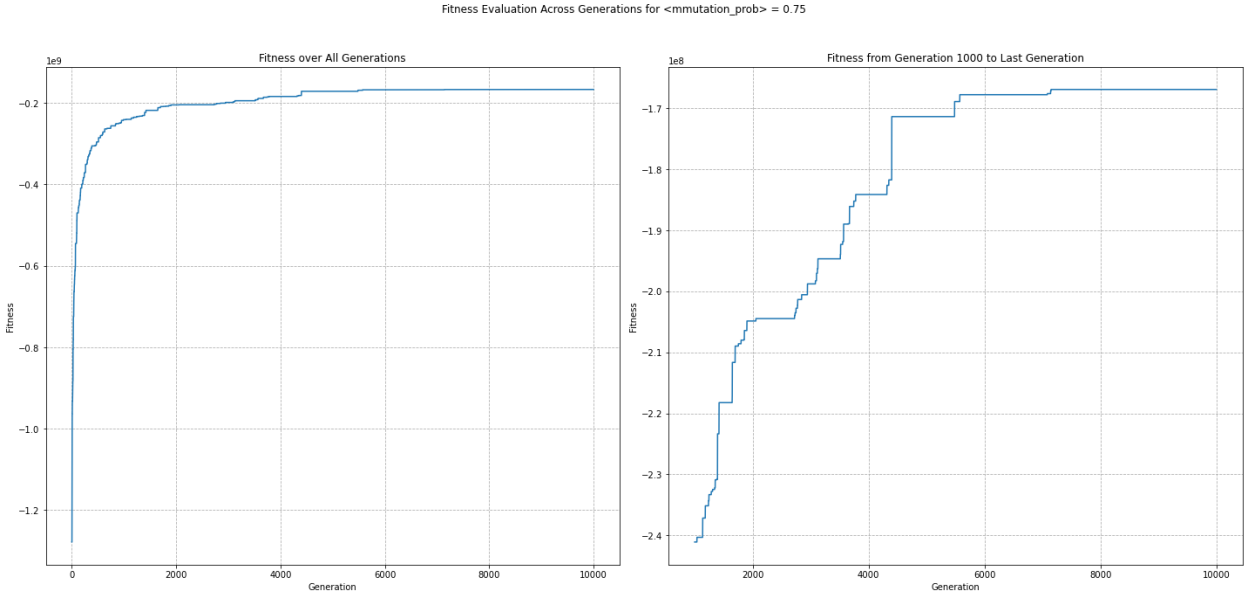

Figure 60: Fitness Plots <mutation_prob> = 0.75.

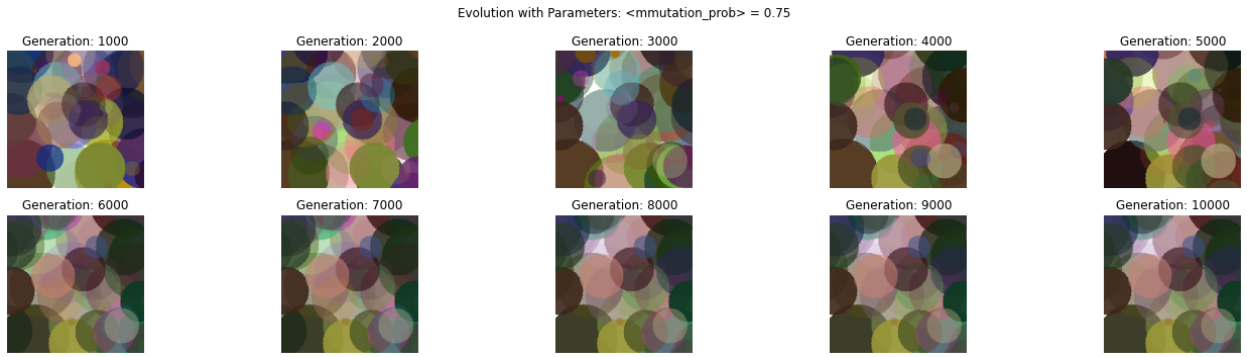

Figure 61: Best Individual in the population at every 1000th generation <mutation_prob> = 0.4.

2.7.4 Analysis

Mutation of the individuals provide diversity among the population. In the algorithm a random number is generated and if this number is smaller than the mutation probability, a random gene is selected to be mutated.

Table 8: Fitness of the Best Individual at the 10,000th Generation for Different Mutation Probabilities

Mutation Prob	Fitness at 10,000th Generation
0.1	-155730391
0.2	-141654825
0.4	-125805727
0.75	-166928080

We can see that the best results are obtained for mutation probability 0.4. Too low and too high mutation probabilities has lower fitness scores. The best score is obtained for mutation probability 0.4. Too low mutation probability (0.1) diminished the diversity since it becomes very unlikely that mutation is performed. In the fitness plots for various mutation probabilities we can see that the convergence of the mutation probability of 0.1 is very later as well. On the other hand, when the mutation probability is too high, it is highly likely that a random gene will be selected. This results may suggest more exploration and less exploitation. Therefore due to the less selective pressure for the older generations the fitness become very low.

2.8 Mutation Type (`mutation_type`)

The algorithm was evaluated with Unguided and Guided (default) mutation type The remaining hyperparameters are selected as default.

2.8.1 Mutation type = Unguided

The following plots showcase the outcomes for mutation type = Unguided, with all other hyperparameters set to their default values.

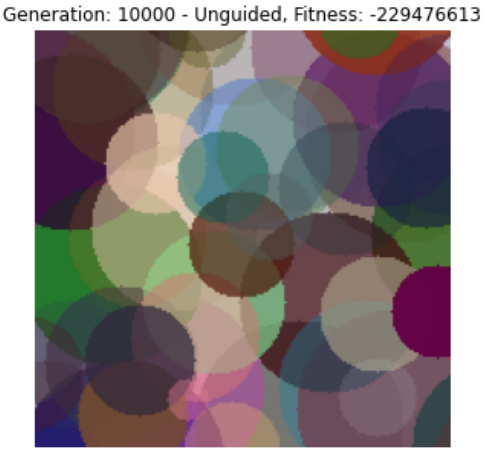

Figure 62: Best Individual <mutation_type> = Unguided.

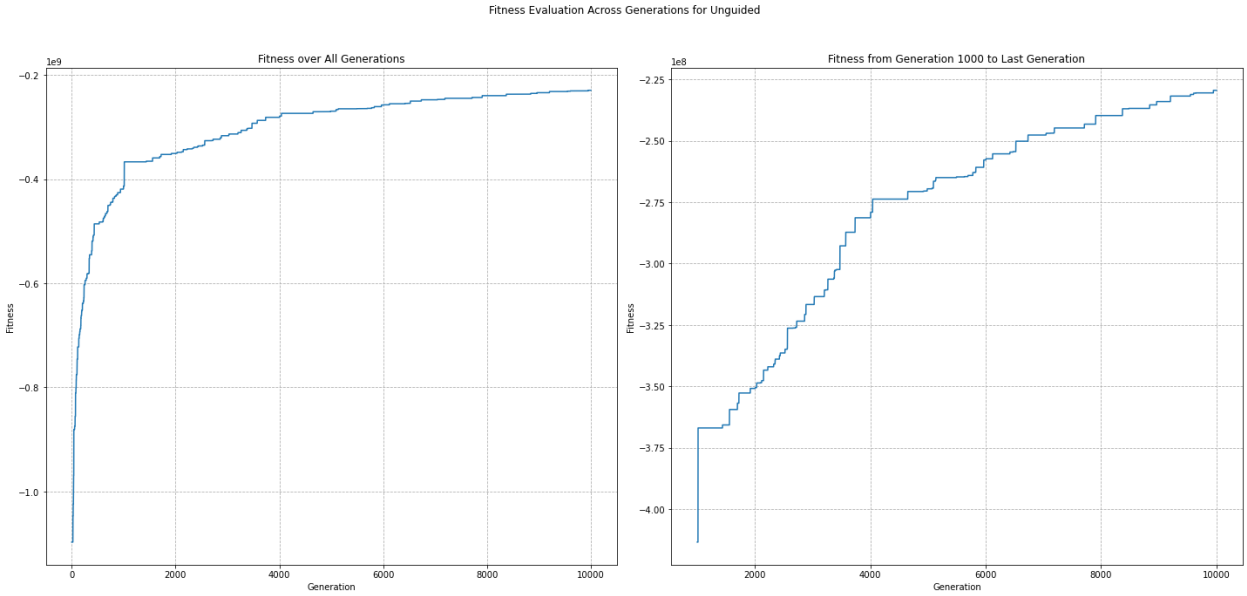

Figure 63: Fitness Plots <mutation_type> = Unguided.

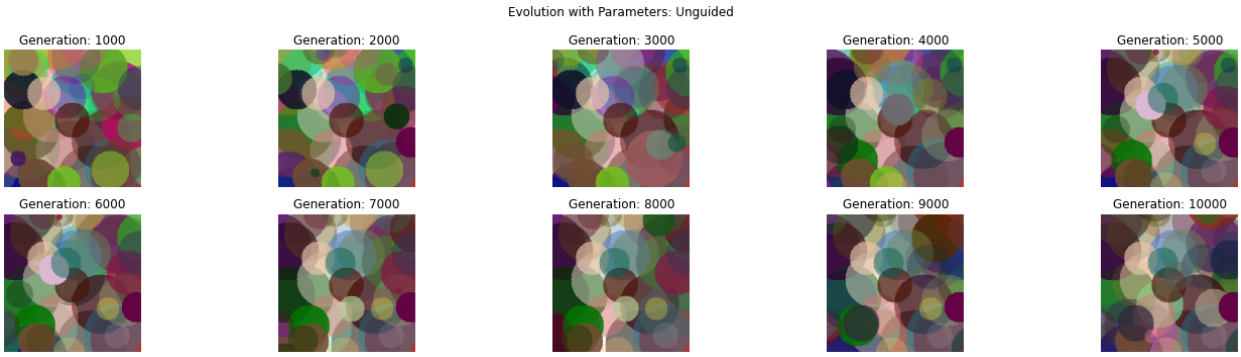

Figure 64: Best Individual in the population at every 1000th generation <mutation_type> = Unguided.

2.8.2 Analysis

In the algorithm, Comparison is made for two primary types of mutation: unguided and guided. Unguided mutation randomly alters genes while guided mutation makes changes around the previous values of the genes.

Table 9: Fitness of the Best Individual at the 10,000th Generation for Mutation types

Mutation Type	Fitness at 10,000th Generation
Guided	-141654825
Unguided	-234053297

The Unguided mutation performed the worst among all hyperparameters. This is because in the unguided mutation genes are altered randomly. For the first generations random changes in the genes may increase the exploration of the space but later due to random initialization of these genes we may loose the good performing individuals in the population. Therefore from the Figure 62 we can see that the details are very few compared to other hyperparameter initializations. Fitness plot is also converging later compared to others.

3 Discussion

Proposition of the following enhancements to improve the convergence of the evolutionary algorithm:

- **Adaptive Mutation Rates**
- **Adaptive Deviation of Radius**
- **Adaptive Deviation for RGB and alpha in guided mutation**

To obtain a faster and possibly better convergence the best strategy to use can be using adaptive parameters which can be generations dependent. This way we can use low selective pressure in the prior generations to obtain a wider search space for finding the global optimum. Then, higher selective pressure in later generations to keep the best individuals unchanged. Therefore, I firstly used an adaptive mutation function which changes the mutation probability from high mutation probability to low probability within generations. This adaptive mutation function can further be improved by determining the rate of change based on the prior and current fitness values. However due to computational reasons I have kept the change of the mutation rate relatively constant. Second suggestion is to deviate the value of radius in the guided mutation around smaller values as the number of generations increase. This is particularly done on the radius deviation. This is also conducted with an adaptive manner at first deviation is conducted around a higher value then after the convergence observed, the deviation is conducted around smaller values for the radius. For the final change RGB and alpha deviation is optimized in an adaptive manner to first increase the exploration then as the convergence observed the values are deviated within small ranges.

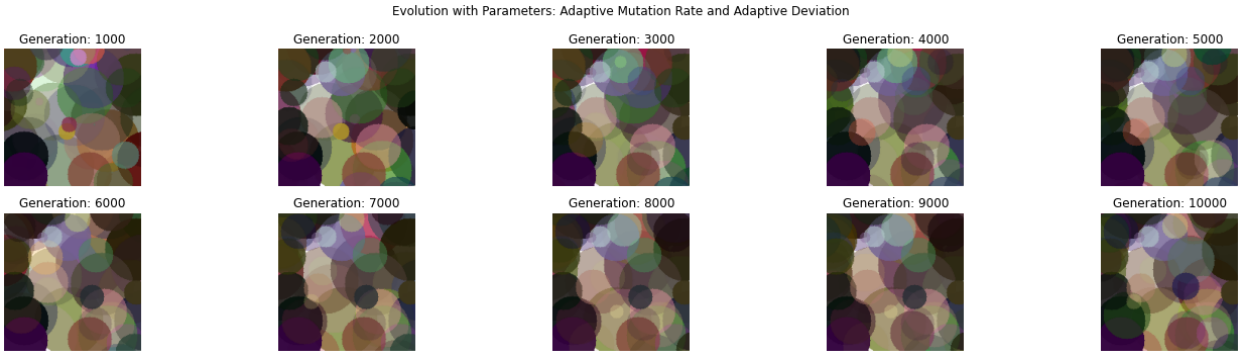

Figure 65: Generations for every 1000th generation for suggested changes

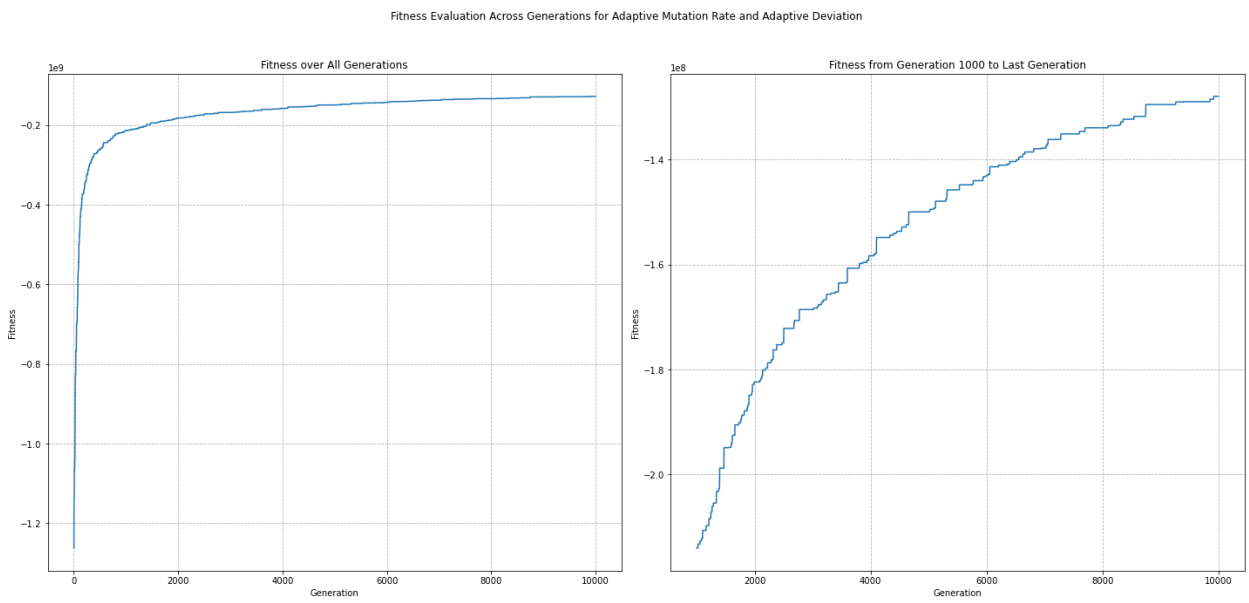

Figure 66: Fitness plots for suggested changes

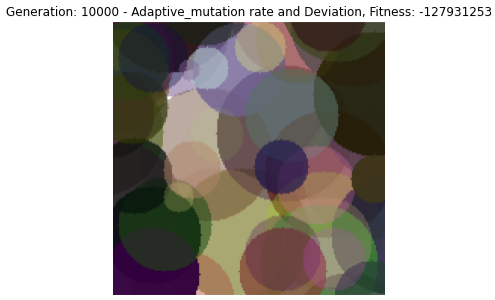

Figure 67: Best individual image with final fitness value

As can be seen from the Figure 66 convergence not only happens very fast it also has a better final fitness value which is presented in Figure 67.

4 Python Code

The code provided includes the suggested changes for faster convergence. The previous parts are commented.

```
1
2 # -*- coding: utf-8 -*-
3 """
4 Created on Thu Apr 18 16:08:16 2024
5
6 @author: defne
7 """
8
9 """
```

```
Initialize population with <num_inds> individuals each having <num_genes>
    genes
While not all generations (<num_generations>) are computed:
    Evaluate all individuals
    Select individuals
    Do crossover on some individuals
    Mutate some individuals
"""

import numpy as np
import random
import cv2
import copy
import matplotlib.pyplot as plt

#Invidiual has one chromosome. Each gene in a chromosome represents one circle
    to be drawn.
#Each gene has at least 7 values: (x,y), r (radius), colors (green, blue,alpha
    ) and A

#Define global variables to be used
source_path = 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW2
    \\painting.png'

source_image = cv2.imread(source_path)
image_shape = np.shape(source_image)

IMG_WIDTH = image_shape[0]
IMG_HEIGHT = image_shape[1]

upper_limit = min(IMG_HEIGHT, IMG_WIDTH) // 4

class Gene:
    def __init__(self):
        while True:
            self.x = random.randint(-2*IMG_WIDTH, 2*IMG_WIDTH) #center x-
    coordinate
            self.y = random.randint(-2*IMG_HEIGHT, 2*IMG_HEIGHT) #center y-
    coordinate
            self.radius = random.randint(1, min(IMG_HEIGHT, IMG_WIDTH)//4) #
    radius
            self.R = random.randint(0, 255) # R
            self.G = random.randint(0, 255) # G
            self.B = random.randint(0, 255) # B
            self.alpha = random.uniform(0, 1) # Alpha

            if not self.valid_circle():
                #self.print_gene()
                break #Exit the loop

    def valid_circle(self):
        if (self.x + self.radius <= 0) or (self.x - self.radius >= IMG_WIDTH)
    or (self.y + self.radius <= 0) or (self.y - self.radius >= IMG_HEIGHT):
            #Outside the image
```

```
            #Should be reinitialized
            return True
        else:
            return False

    def correct_gene(self):

        # Correct gene attributes to valid values if necessary
        self.x = max(-2 * IMG_WIDTH, min(2 * IMG_WIDTH, self.x))
        self.y = max(-2 * IMG_HEIGHT, min(2 * IMG_HEIGHT, self.y))
        self.radius = max(1, min(min(IMG_HEIGHT, IMG_WIDTH) // 4, self.radius)
    )
        self.R = max(0, min(255, self.R))
        self.G = max(0, min(255, self.G))
        self.B = max(0, min(255, self.B))
        self.alpha = max(0, min(1, self.alpha))

        print('corrected somehow')

    def print_gene(self):
        print(f"  x: {self.x}")
        print(f"  y: {self.y}")
        print(f"  radius: {self.radius}")
        print(f"  R: {self.R}")
        print(f"  G: {self.G}")
        print(f"  B: {self.B}")
        print(f"  alpha: {self.alpha}")
        print(f"  Color: ({self.B}, {self.G}, {self.R})")


class Individual:
    def __init__(self, num_genes, chromosome = []):
        self.num_genes = num_genes
        self.chromosome = chromosome if chromosome else [Gene() for i in range
    (num_genes)] #Individual has one chromosome, each gene in a chromosome
    represents one circle to be drawn.
        self.fitness = 0 #fitness value

        # if len(self.chromosome) == 0:
        #     for _ in range(num_genes):
        #         self.chromosome.append(Gene()) #a gene object is created and
     added to the chromosome

    def sort_genes(self):
        self.chromosome = sorted(self.chromosome, key=lambda x: x.radius ,
    reverse = True)
    # def initialize_random(self):
    #     #Initialize genes randomly within image boundaries
    #     for _ in range(self.num_genes):

    #         self.chromosome.append(Gene()) #a gene object is created and
    added to the chromosome

    # Draw the individual circle
```

```
    def draw(self):
        #Draw circles on the image
        image_new = 255 * np.ones(image_shape, np.uint8) #Initializing overlay as white image

        self.sort_genes() #genes are sorted according to their radius

        for gene in self.chromosome:
            #A deepcopy of the whiteFrame is created
            overlay = copy.deepcopy(image_new)

            color = (gene.B, gene.G, gene.R)  # OpenCV uses BRG format
            # a circle is created on to the deepcopy of the overlay
            #print(f"Color: {color}")
            cv2.circle(overlay, (gene.x,gene.y), gene.radius, color, -1) #A filled circle
            # there is a problem here fix please
            cv2.addWeighted(overlay, gene.alpha, image_new, 1 - gene.alpha, 0, image_new)

        return image_new

    # Evaluate the fitness of the individual
    def evaluate_fitness(self):
        # Draw circles and calculate fitness
        # Implementation of drawing circles and calculating fitness goes here
        image = self.draw()
        image = np.array(image, np.int64)
        source_np = np.array(source_image, np.int64)

        squared_diff = np.square(source_np - image)
        self.fitness = -np.sum(squared_diff)

    def print_genes(self):
        for idx, gene in enumerate(self.chromosome):
            print(f"Gene {idx + 1}:")
            print(f"  x: {gene.x}")
            print(f"  y: {gene.y}")
            print(f"  radius: {gene.radius}")
            print(f"  R: {gene.R}")
            print(f"  G: {gene.G}")
            print(f"  B: {gene.B}")
            print(f"  alpha: {gene.alpha}")
    # Mutate the individuals
    def mutate_gene(self, mutation_prob, mutation_type, radius_deviation, RGB_deviation, alpha_deviation):
        # if random.random() >= mutation_prob:
        #     return

        #Randomly change random genes
        while random.random() < mutation_prob:

            #random_gene = random.choice(self.chromosome) #A random gene is selected
```

```
153             # Randomly select a gene to be mutated
154             index_to_mutate = random.randint(0, self.num_genes - 1)
155             random_gene = self.chromosome[index_to_mutate]
156 
157             if mutation_type == 'Unguided':
158                 #print('there is mutation unguided')
159 
160                 while True:
161                     random_gene.x = random.randint(-2*IMG_WIDTH, 2*IMG_WIDTH)
    #center x-coordinate
162                     random_gene.y = random.randint(-2*IMG_HEIGHT, 2*IMG_HEIGHT
    ) #center y-coordinate
163                     random_gene.radius = random.randint(1, min(IMG_HEIGHT,
    IMG_WIDTH)//4) #radius
164                     random_gene.R = random.randint(0, 255) # R
165                     random_gene.G = random.randint(0, 255) # G
166                     random_gene.B = random.randint(0, 255) # B
167                     random_gene.alpha = random.uniform(0, 1) # Alpha
168 
169                     if not random_gene.valid_circle():
170                         #Replace the mutated gene to its place
171                         #self.chromosome[index_to_mutate] = copy.deepcopy(
    random_gene)
172                         break
173 
174             elif mutation_type == 'Guided':
175 
176                 x_backup = random_gene.x
177                 y_backup = random_gene.y
178                 radius_backup = random_gene.radius
179 
180                 R_backup = random_gene.R
181                 G_backup = random_gene.G
182                 B_backup = random_gene.B
183                 alpha_backup = random_gene.alpha
184 
185                 lower_x_limit = x_backup - IMG_WIDTH//4 + 1
186                 upper_x_limit = x_backup + IMG_WIDTH//4 - 1
187                 lower_y_limit = y_backup - IMG_HEIGHT//4 + 1
188                 upper_y_limit = y_backup + IMG_HEIGHT//4 - 1
189 
190 
191                 while True:
192                     #Deviate the x,y,radius,R,G,B,A
193 
194                     random_gene.x = random.randint(lower_x_limit,
    upper_x_limit)
195                     random_gene.y = random.randint(lower_y_limit,
    upper_y_limit)
196 
197 
198 
199 
```

```
                    if (radius_backup - radius_deviation < 1 and radius_backup + radius_deviation <= upper_limit):
                        random_gene.radius = random.randint(1, radius_backup + radius_deviation)
                    elif (radius_backup - radius_deviation >= 1 and radius_backup + radius_deviation > upper_limit):
                        random_gene.radius = random.randint(radius_backup - radius_deviation, upper_limit)
                    elif (radius_backup - radius_deviation < 1 and radius_backup + radius_deviation > upper_limit):
                        random_gene.radius = random.randint(1, upper_limit)
                    else:
                        random_gene.radius = random.randint(radius_backup - radius_deviation, radius_backup + radius_deviation)


                    if not random_gene.valid_circle():
                        #Replace the mutated gene to its place
                        #random_gene.print_gene()

                        upper_limit_rgb = 255
                        if (R_backup - RGB_deviation < 0 and R_backup + RGB_deviation <= upper_limit_rgb):
                            random_gene.R = random.randint(0, R_backup + RGB_deviation)
                        elif (R_backup - RGB_deviation >= 0 and R_backup + RGB_deviation > upper_limit_rgb):
                            random_gene.R = random.randint(R_backup - RGB_deviation, upper_limit_rgb)
                        else:
                            random_gene.R = random.randint(R_backup - RGB_deviation, R_backup + RGB_deviation)

                        if (G_backup - RGB_deviation < 0 and G_backup + RGB_deviation <= upper_limit_rgb):
                            random_gene.G = random.randint(0, G_backup + 64)
                        elif (G_backup - RGB_deviation >= 0 and G_backup + RGB_deviation > upper_limit_rgb):
                            random_gene.G = random.randint(G_backup - RGB_deviation, upper_limit_rgb)
                        else:
                            random_gene.G = random.randint(G_backup - RGB_deviation, G_backup + RGB_deviation)

                        if (B_backup - RGB_deviation < 0 and B_backup + RGB_deviation <= upper_limit_rgb):
                            random_gene.B = random.randint(0, B_backup + RGB_deviation)
                        elif (B_backup - RGB_deviation >= 0 and B_backup + RGB_deviation > upper_limit_rgb):
                            random_gene.B = random.randint(B_backup - RGB_deviation, upper_limit_rgb)
                        else:
```

```
                            random_gene.B = random.randint(B_backup -
RGB_deviation, B_backup + RGB_deviation)


                    if (alpha_backup - alpha_deviation < 0 and
alpha_backup + alpha_deviation <= 1):
                            random_gene.alpha = random.uniform(0, alpha_backup
 + alpha_deviation)
                    elif (alpha_backup - alpha_deviation >= 0 and
alpha_backup + alpha_deviation > 1):
                            random_gene.alpha = random.uniform(alpha_backup -
alpha_deviation, 1)
                    else:
                            random_gene.alpha = random.uniform(alpha_backup -
alpha_deviation, alpha_backup + alpha_deviation)

                    # upper_limit_rgb = 255
                    # if (R_backup - 64 < 0 and R_backup + 64 <=
upper_limit_rgb):
                    #     random_gene.R = random.randint(0, R_backup + 64)
                    # elif (R_backup - 64 >= 0 and R_backup + 64 >
upper_limit_rgb):
                    #     random_gene.R = random.randint(R_backup - 64,
upper_limit_rgb)
                    # else:
                    #     random_gene.R = random.randint(R_backup - 64,
R_backup + 64)

                    # if (G_backup - 64 < 0 and G_backup + 64 <=
upper_limit_rgb):
                    #     random_gene.G = random.randint(0, G_backup + 64)
                    # elif (G_backup - 64 >= 0 and G_backup + 64 >
upper_limit_rgb):
                    #     random_gene.G = random.randint(G_backup - 64,
upper_limit_rgb)
                    # else:
                    #     random_gene.G = random.randint(G_backup - 64,
G_backup + 64)

                    # if (B_backup - 64 < 0 and B_backup + 64 <=
upper_limit_rgb):
                    #     random_gene.B = random.randint(0, B_backup + 64)
                    # elif (B_backup - 64 >= 0 and B_backup + 64 >
upper_limit_rgb):
                    #     random_gene.B = random.randint(B_backup - 64,
upper_limit_rgb)
                    # else:
                    #     random_gene.B = random.randint(B_backup - 64,
B_backup + 64)


                    # if (alpha_backup - 0.25 < 0 and alpha_backup + 0.25
<= 1):
```

```
                        #     random_gene.alpha = random.uniform(0,
    alpha_backup + 0.25)
                        # elif (alpha_backup - 0.25 >= 0 and alpha_backup +
    0.25 > 1):
                        #     random_gene.alpha = random.uniform(alpha_backup
    - 0.25, 1)
                        # else:
                        #     random_gene.alpha = random.uniform(alpha_backup
    - 0.25, alpha_backup + 0.25)


                        #self.chromosome[index_to_mutate] = copy.deepcopy(
    random_gene)
                        break

class Population:

    def __init__(self, num_inds, num_genes):
        self.num_inds = num_inds
        self.num_genes = num_genes
        self.population = [Individual(num_genes) for _ in range(num_inds)] #
    start the population

        #self.initialize_population()
        # for _ in range(self.num_inds):
        #     self.population.append(Individual(self.num_genes))

    # def initialize_population(self):
    #     for _ in range(self.num_inds):
    #         #individual = Individual(self.num_genes)
    #         #individual.initialize_random()
    #         self.population.append(Individual(self.num_genes))

    def selection(self, frac_elites, frac_parents, tm_size):

        num_elites = int(self.num_inds * frac_elites) # num elites are chosen
        num_parents = int(self.num_inds * frac_parents) # num of parents are
    chosen

        #number of parents should be even
        if num_parents % 2 == 1:
            num_parents += 1 # even

        #Sort the indviduals based on fitness
        self.population.sort(key=lambda x: x.fitness, reverse = True)

        #Select elites
        elites = self.population[:num_elites]

        other_ind = self.population[num_elites:]

        non_elites = []

```

```
        # the selection of other individuals are done with tournament selection
        for _ in range(len(other_ind)):
            tournament = random.sample(other_ind , min(tm_size, len(other_ind)))
            winner = max(tournament, key=lambda x: x.fitness)
            non_elites.append(winner)

        #another tournament selection for the parents now
        # The selection of parent individuals
        # Same individual can still win the tournament as a parent.
        parents = []

        non_elites.sort(key = lambda x: x.fitness, reverse = True)
        parents = non_elites[:num_parents] #best of the non_elites are taken as the parents

        # Remove the selected parents from non_elites using list comprehension
        non_elites = non_elites[num_parents:]

        return elites, non_elites, parents

    def crossover(self, parents):

        childeren = []

        # Pair up parents
        for i in range(0, len(parents), 2):
            parent1 = parents[i]
            parent2 = parents[i+1]

            child1_chromosome = []
            child2_chromosome = []

            for j in range(self.num_genes):
                # Exchange of gene is calculated individually with equal probability
                if random.random() < 0.5:
                    child1_chromosome += [copy.deepcopy(parent1.chromosome[j])] #child1 gets the gene from parent1
                    child2_chromosome += [copy.deepcopy(parent2.chromosome[j])] #child2 gets the gene from parent2

                else:
                    child1_chromosome += [copy.deepcopy(parent2.chromosome[j])] #child1 gets the gene from parent2
                    child2_chromosome += [copy.deepcopy(parent1.chromosome[j])] #child2 gets the gene from parent1

            #After the exchange the childs become individuals with the settled genes
            child1 = Individual(self.num_genes, child1_chromosome) #Individual object of child1
```

```
            child2 = Individual(self.num_genes, child2_chromosome) #Individual
     object of child2
            child1.evaluate_fitness()
            child2.evaluate_fitness()

            childeren.append(child1)
            childeren.append(child2)

        #Parents are lost
        return childeren
    def mutate_pop(self, population, mutation_prob, mutation_type, generation)
    :
        #Mutate some individuals
        for individual in population:
            if generation <= 3000:
                individual.mutate_gene(mutation_prob, mutation_type, 15, 100,
    0.4) #deviation of the radius is raduced
            elif generation > 3000 and generation <= 7000:
                individual.mutate_gene(mutation_prob, mutation_type, 10, 64,
    0.25) #deviation of the radius is reduced
            elif generation > 7000:
                individual.mutate_gene(mutation_prob, mutation_type, 5, 32,
    0.1) #deviation of the radius is reduced

    def evolutionary_algorithm(self, num_generations, frac_elites,
    frac_parents, tm_size, mutation_prob, mutation_type):

        fitness_history = []
        image_filenames = [] # Add this line to keep track of the saved image
    filenames
        mutation_prob = 0.6

        for generation in range(num_generations):
            #print(f"Generation {generation + 1}")

            if(generation <= 1000):
                if (generation + 1) % 100 == 0:
                    #high probability for higher diversity
                    mutation_prob = mutation_prob - (0.2/10)
            elif (generation > 1000) and (generation < 5000):
                if (generation + 1) % 1000 == 0:
                    #reduces the mutation probability with small and reaches
    to 0.1 after generation 5000
                    mutation_prob = mutation_prob - (0.1/4)

            #Evaluate all individuals
            for individual in self.population:
                individual.evaluate_fitness()

            #select individuals
            elites, non_elites, parents = self.selection(frac_elites,
    frac_parents, tm_size)

            #Do crossover on some individuals
```

```
            childeren = self.crossover(parents)

            non_elites += childeren


            #Mutate some individuals
            self.mutate_pop(non_elites, mutation_prob, mutation_type,
    generation)

            for i in non_elites:
                i.evaluate_fitness()

            #reassigning the population
            self.population = elites + non_elites

            best_individual = max(self.population, key=lambda x : x.fitness)

            # Store fitness of the best individual
            fitness_history.append(best_individual.fitness)

            if (generation + 1) % 1000 == 0:
                filename = f'best_individual_generation_{generation + 1}.png'
                print(f"Fitness of best individual at generation {generation +
     1}: {best_individual.fitness}")
                best_individual_image = best_individual.draw()
                cv2.imwrite(filename, best_individual_image)
                image_filenames.append(filename) # Save the filename

        # # Plot fitness history
        # plt.plot(range(1, num_generations + 1), fitness_history)
        # plt.xlabel('Generation')
        # plt.ylabel('Fitness')
        # plt.title('Fitness Plot from Generation 1 to Generation 10000')
        # plt.show()

        # plt.plot(range(1000, num_generations + 1), fitness_history[999:])
        # plt.xlabel('Generation')
        # plt.ylabel('Fitness')
        # plt.title('Fitness Plot from Generation 1000 to Generation 10000')
        # plt.show()

        best = max(self.population, key=lambda x: x.fitness)
        return best, image_filenames, fitness_history # Return the collected
    image filenames along with the best individual

num_inds = 20
num_genes = 50
tm_size = 5
frac_elites = 0.2
frac_parents = 0.6
mutation_prob = 0.2
mutation_type = 'Guided'
```

```
pop = Population(num_inds, num_genes)
#pop.initialize_population()
best, image_filenames, fitness_history = pop.evolutionary_algorithm(10000,
    frac_elites, frac_parents, tm_size, mutation_prob, mutation_type)

def plot_evolution(images, parameters):
    num_rows = 2
    num_cols = 5
    #plt.figure(fig_size=(24,12))
    fig, axs = plt.subplots(num_rows, num_cols, figsize=(20,5)) # Adjust the
    figsize based on your requirement

    for i, img_path in enumerate(images):
        img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
        row = i // num_cols  # Calculate row index
        col = i % num_cols   # Calculate column index
        axs[row, col].imshow(img)
        axs[row, col].set_title(f'Generation: {(i + 1) * 1000}') # Title with
    generation
        axs[row, col].axis('off') # Hide axis

    plt.suptitle(f"Evolution with Parameters: {parameters}") # Main title with
     parameter
    plt.tight_layout()
    plt.show()

def plot_fitness(fitness_history, num_generations, parameter):
    #plt.figure(fig_size=(24,12))
    fig, axs = plt.subplots(1, 2, figsize=(20, 10))  # Adjust the figsize to
    make the figure larger

    # Plot for the entire range of generations
    axs[0].plot(range(1, num_generations + 1), fitness_history)
    axs[0].set_xlabel('Generation')
    axs[0].set_ylabel('Fitness')
    axs[0].set_title('Fitness over All Generations')
    axs[0].grid(linestyle = "dashed")

    # Plot for the range from 1000th generation to the last
    axs[1].plot(range(1000, num_generations + 1), fitness_history[999:])
    axs[1].set_xlabel('Generation')
    axs[1].set_ylabel('Fitness')
    axs[1].set_title('Fitness from Generation 1000 to Last Generation')
    axs[1].grid(linestyle = "dashed")


    # Add a main title for the whole figure
    plt.suptitle(f'Fitness Evaluation Across Generations for {parameter}')

    # Show plot with appropriate layout
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust the rect if the title
    overlaps with the subplots
    plt.show()

```

```
# Call the plotting function with the collected filenames and parameters
plot_evolution(image_filenames, 'Adaptive Mutation Rate and Adaptive Deviation')
plot_fitness(fitness_history, 10000, 'Adaptive Mutation Rate and Adaptive Deviation')

best_image = best.draw()
best.evaluate_fitness()
filename = 'best_individual_generation_10000_adaptive_mutation_rate_and_deviation.png'
image_path = 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW2\\' + filename
cv2.imwrite(filename, best_image)


img = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)  # Convert from BGR to RGB
plt.figure(figsize=(10, 5))  # Set the figure size as needed
plt.imshow(img)
plt.title(f'Generation: 10000 - Adaptive_mutation rate and Deviation, Fitness: {best.fitness}')
plt.axis('off')  # Hide axis

plt.show()
print("Fitness:", best.fitness)

# cv2.namedWindow('image_new', cv2.WINDOW_NORMAL)
# cv2.resizeWindow('image_new', 200, 200)
# cv2.imshow('image_new', best_image)

# cv2.waitKey(0) #Waiting for a key press to close the window
# cv2.destroyAllWindows() #Close the window
```