Deniz Karakay - 2443307 Date: 18/05/2023

EE449- Homework 2

Evolutionary Art

1. The Evolutionary Algorithm

The codes that I have written for the purpose of individual representation, fitness function evaluation, tournament selection, crossover, and mutation are available for examination in *Appendix I.* The visualization helper functions can be found in *Appendix II*. Additionally, a comprehensive analysis of other significant classes, methods, and code expressions can also be found therein.

2. Experiment Results

In this experimental work, I run the algorithm for several varied hyperparameters (total 21 times). When I am testing a parameter, I keep remaining parameters as default as it is asked in the experiment part.

2.1. Default parameters

Default parameters are as follows:
<num_inds> = **20**
<num_genes> = **50**
<tm_size> = **5**
<frac_elites> = **0.2**
<frac_parents> = **0.6**
<mutation_prob> = **0.2**
<mutation_type> = **guided**

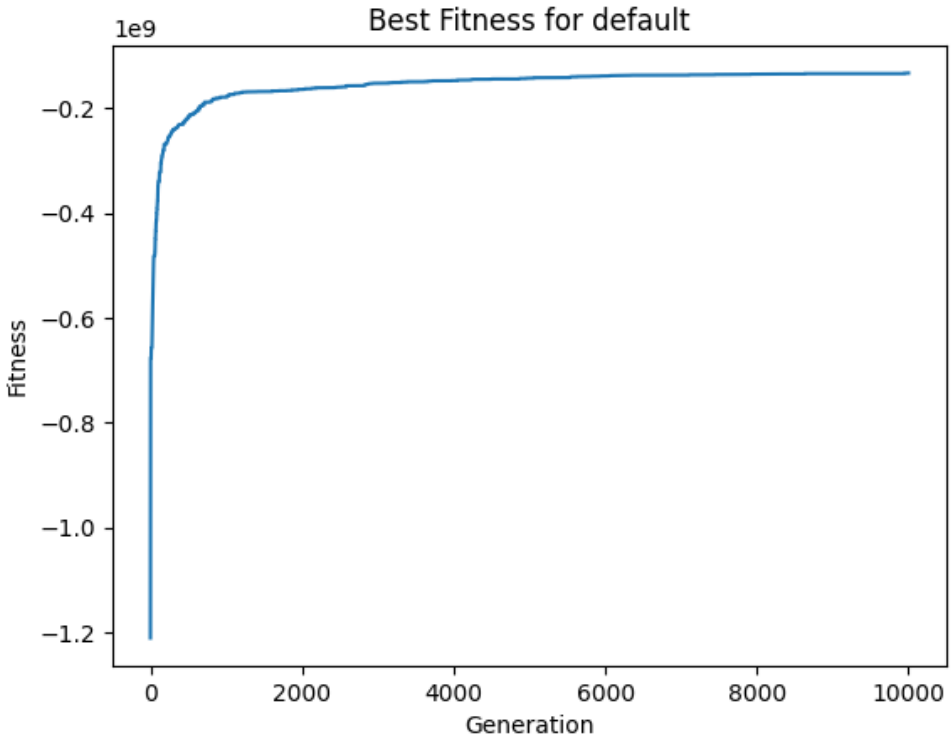


Figure 1. The Fitness Plot Generation (1-10000) for default

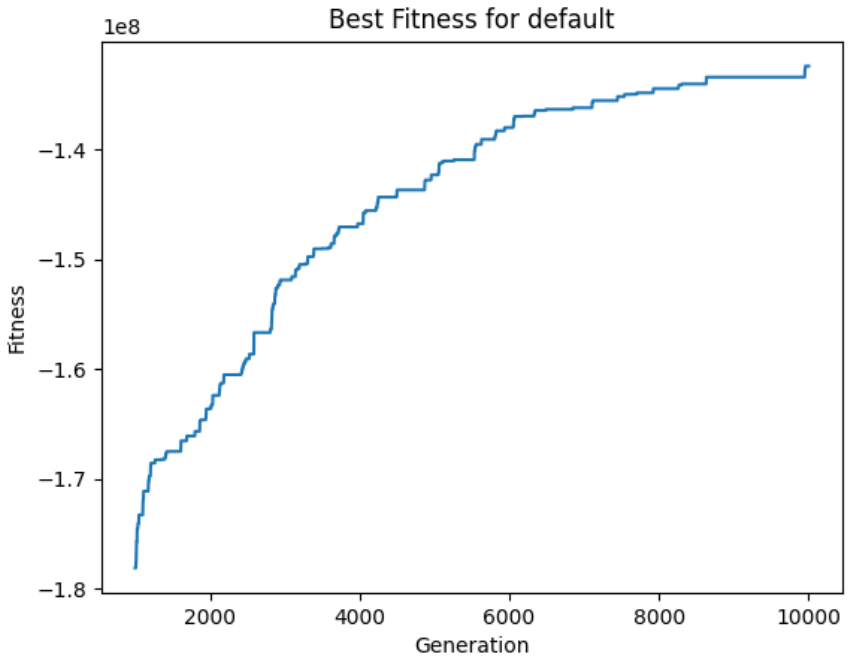


Figure 2. The Fitness Plot Generation (1000-10000) for default

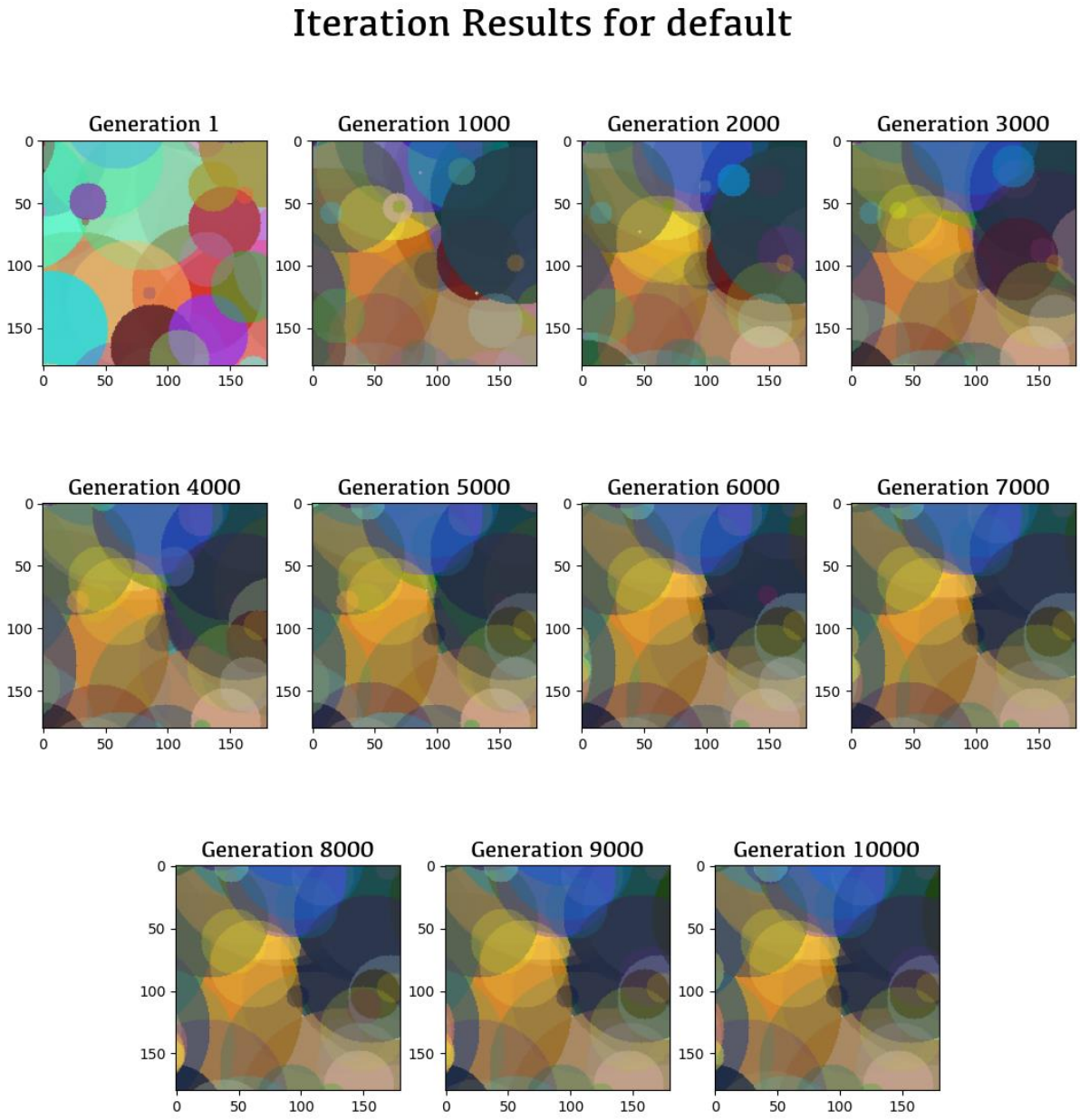


Figure 3. Corresponding Images of Best Individuals for default

As it can be seen, from Figure 1, after around 1500 generation, the fitness started to saturate. The best fitness value is *-132389734.*

2.2. Number of Individuals

2.2.1. Num_inds = 5

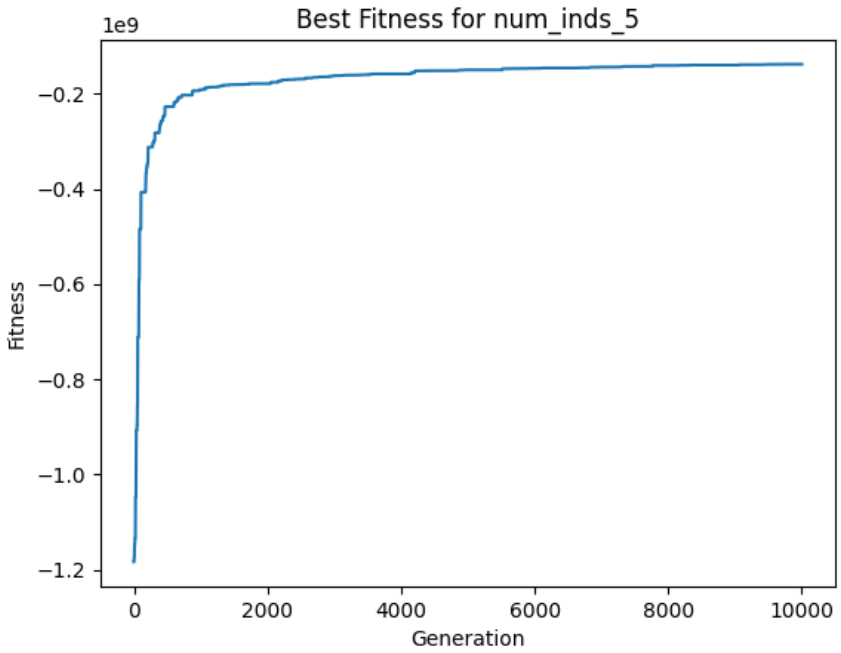


Figure 4. The Fitness Plot Generation (1-10000) for num_inds=5

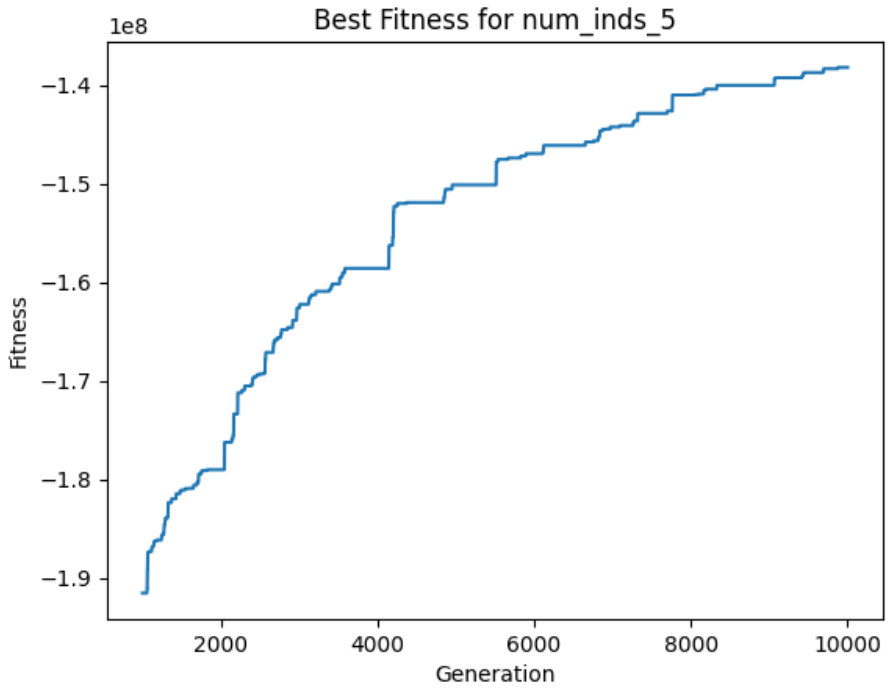


Figure 5. The Fitness Plot Generation (1000-10000) for num_inds=5

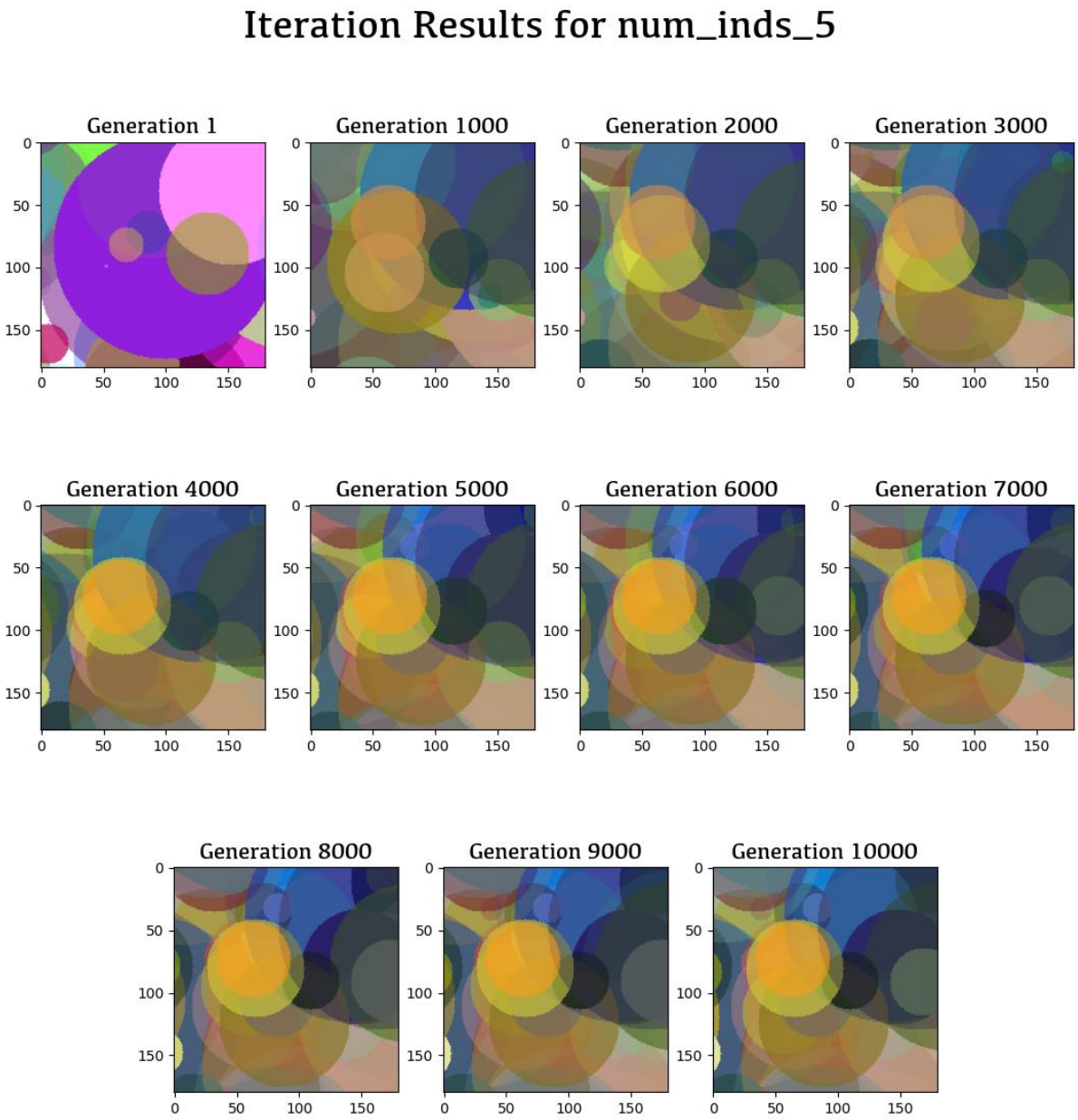


Figure 6. Corresponding Images of Best Individuals for num_inds=5

The best fitness value is *-138160430.*

2.2.2. Num_inds = 10

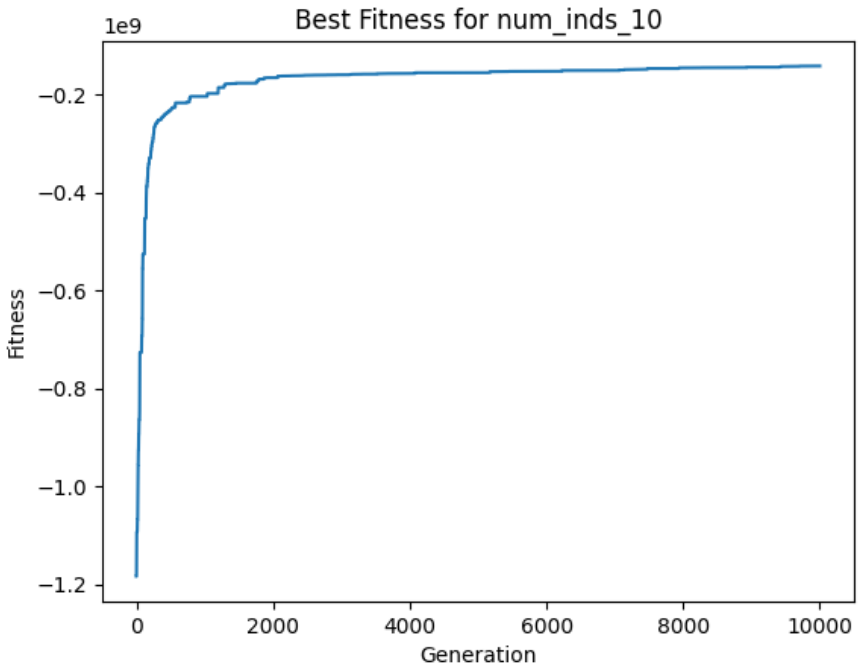


Figure 7. The Fitness Plot Generation (1-10000) for num_inds=10

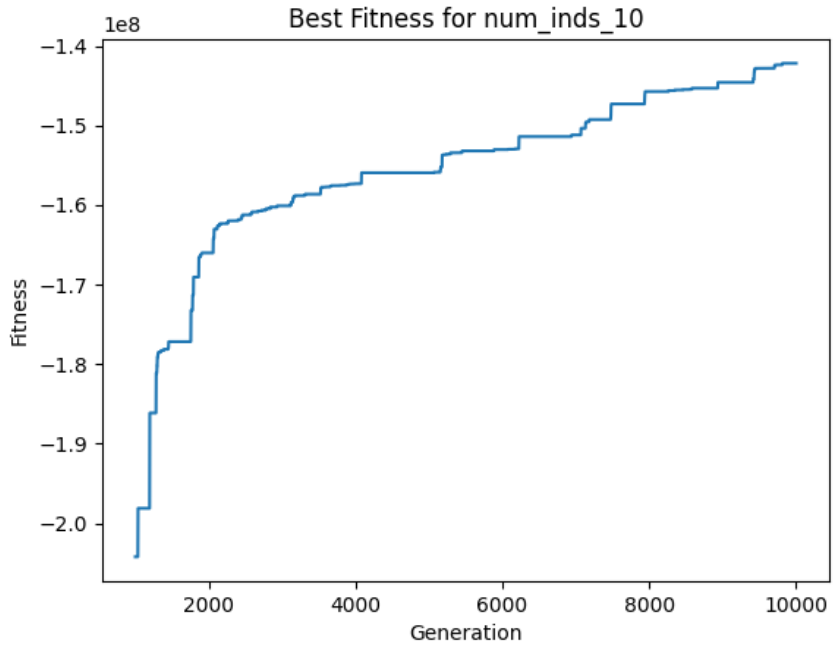


Figure 8. The Fitness Plot Generation (1-10000) for num_inds=10

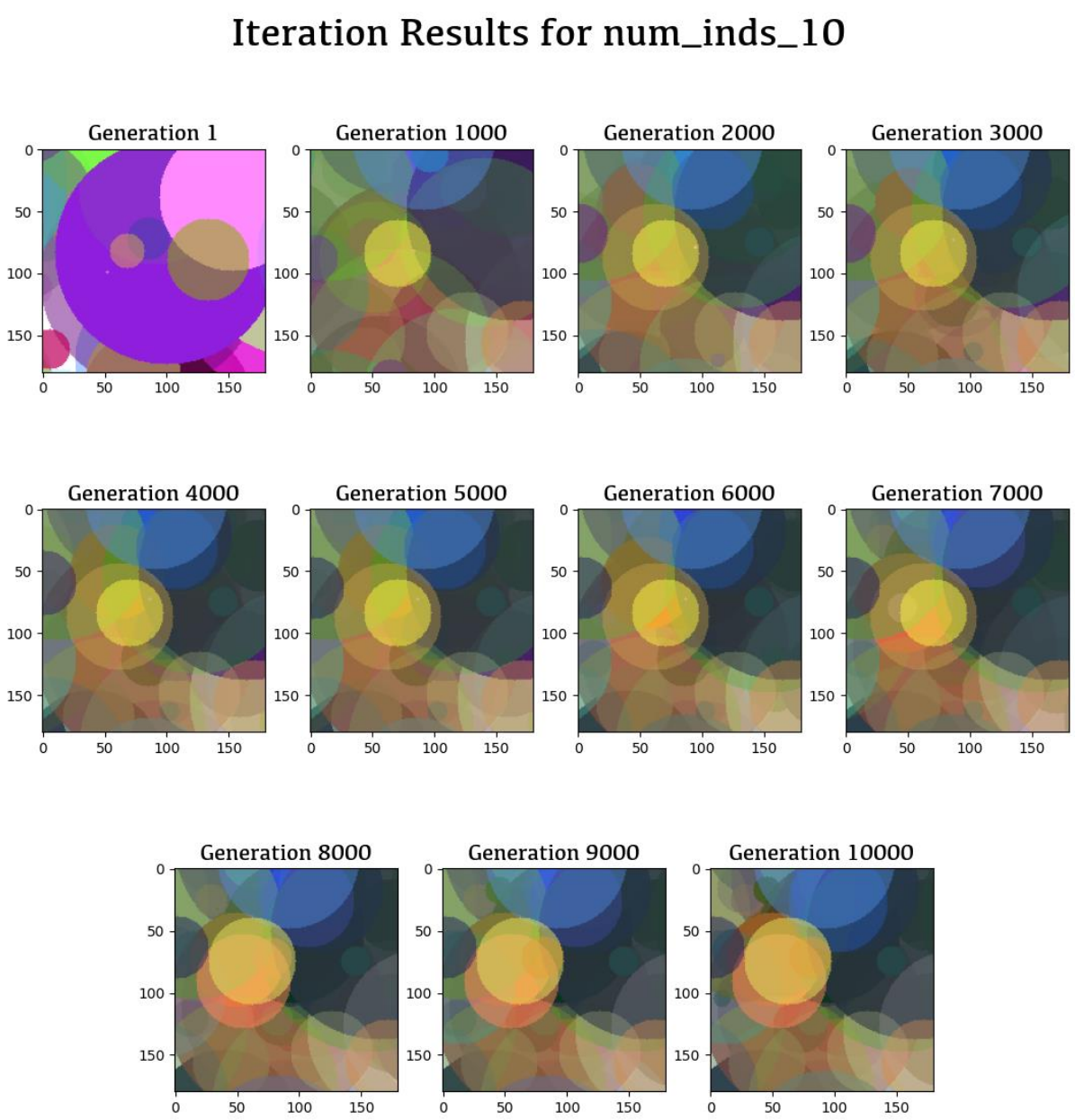


Figure 9. Corresponding Images of Best Individuals for num_inds=10

The best fitness value is *-142162726.*

2.2.3. Num_inds = 40

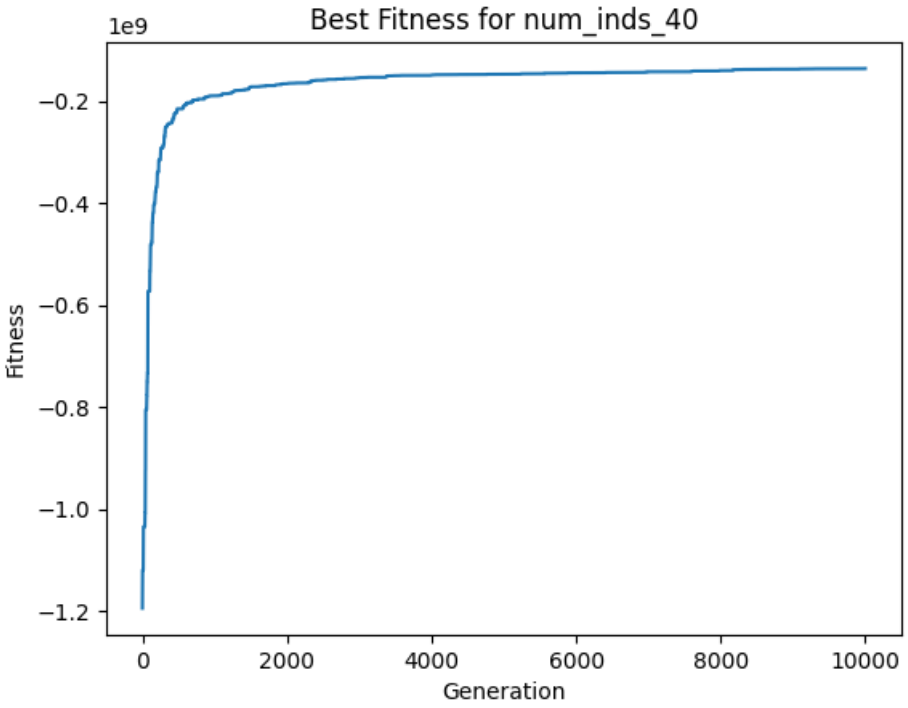


Figure 10. The Fitness Plot Generation (1-10000) for num_inds=40

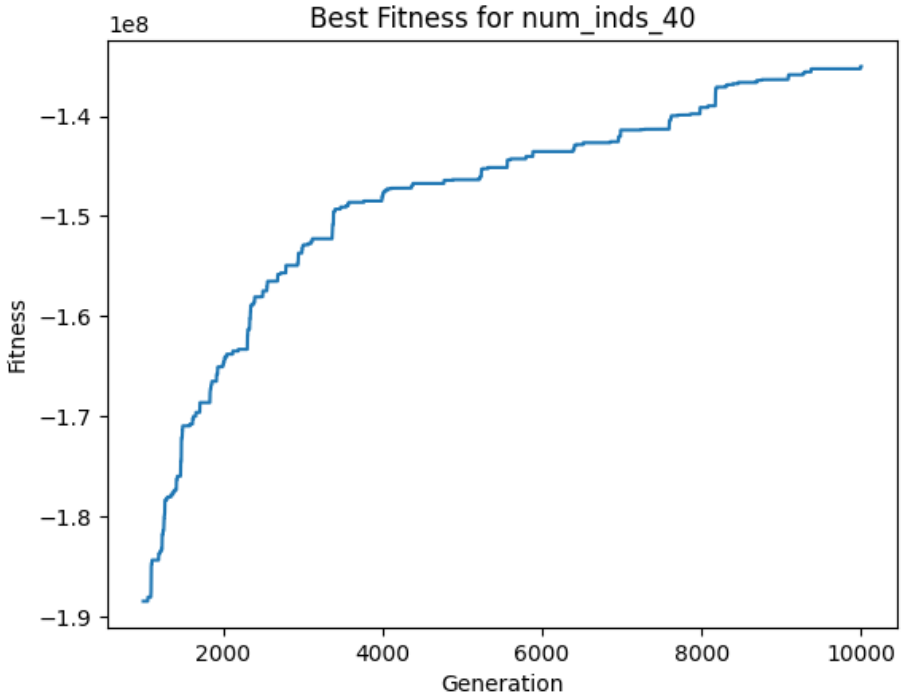


Figure 11. The Fitness Plot Generation (1000-10000) for num_inds=40

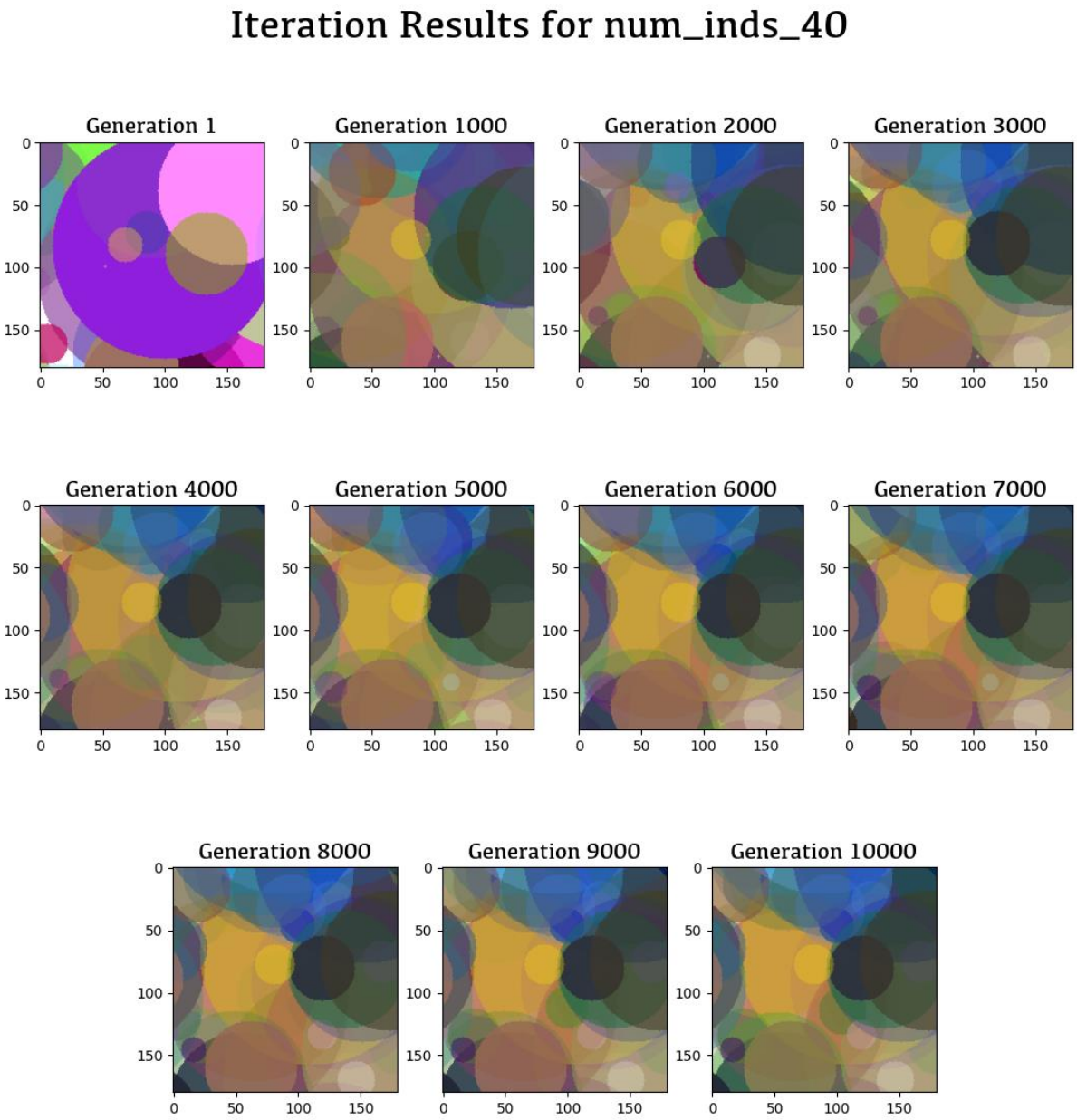


Figure 12. Corresponding Images of Best Individuals for num_inds=40

The best fitness value is *-135048313.*

2.2.4. Num_inds = 60

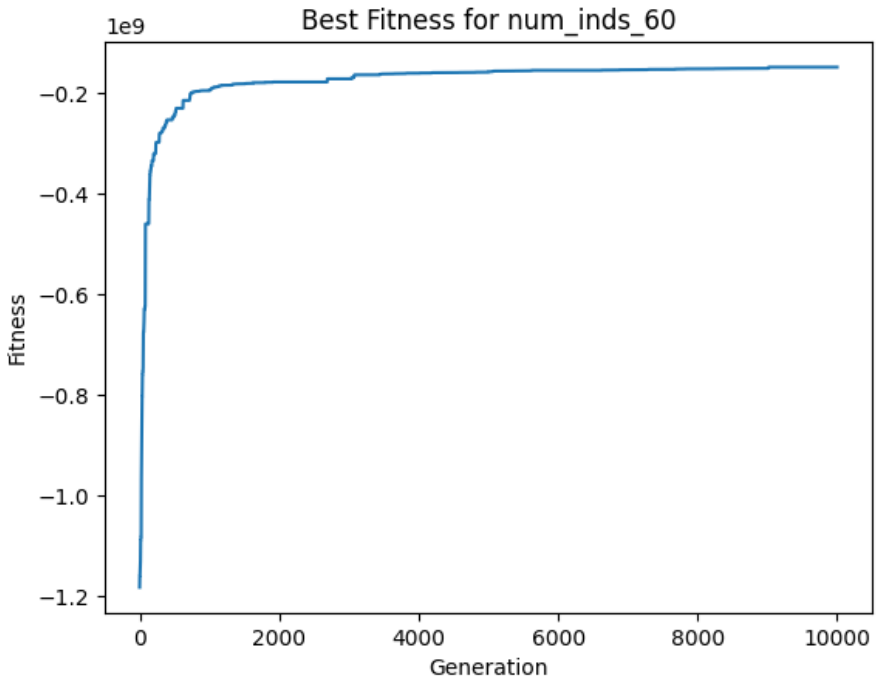


Figure 13. The Fitness Plot Generation (1-10000) for num_inds=60

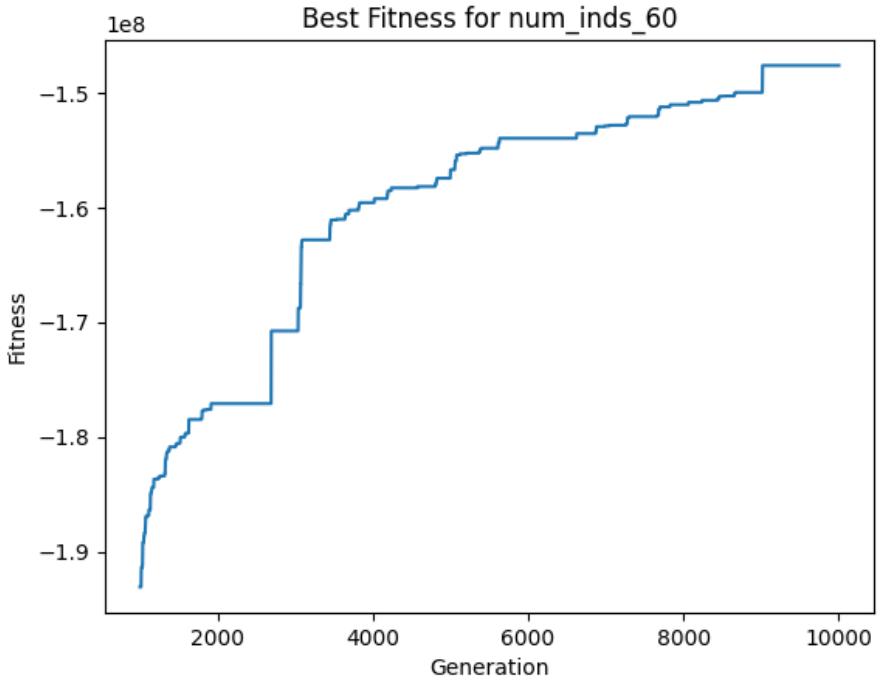


Figure 14. The Fitness Plot Generation (1000-10000) for num_inds=60

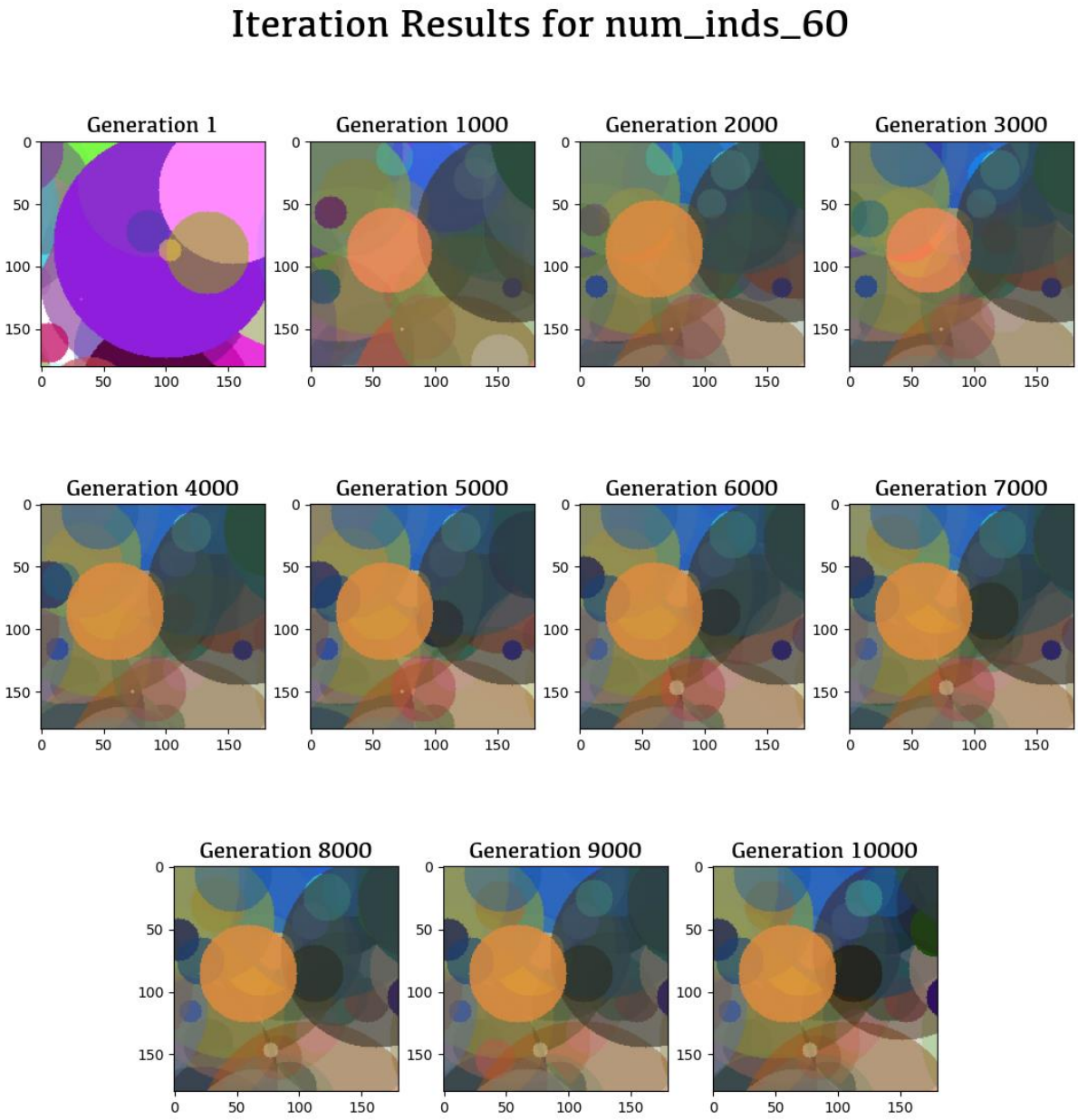


Figure 15. Corresponding Images of Best Individuals for num_inds=60

The best fitness value is *-147595683*

Upon examination of *Figures (1-15)*, it is evident that the optimal fitness value is achieved when the default parameter **num_inds = 20** is utilized. This is since a smaller population size will result in a lack of diversity in the results, while a larger population size will require a longer time to converge to a minimum, ultimately leading to a worse fitness value.

2.3. Number of Genes

2.3.1. Num_genes = 15

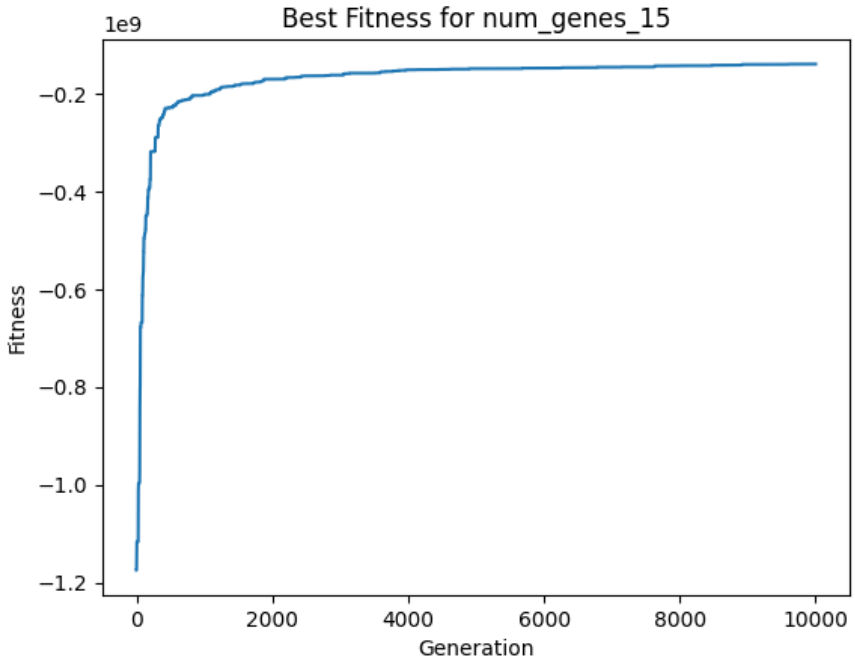


Figure 16. The Fitness Plot Generation (1-10000) for num_genes=15

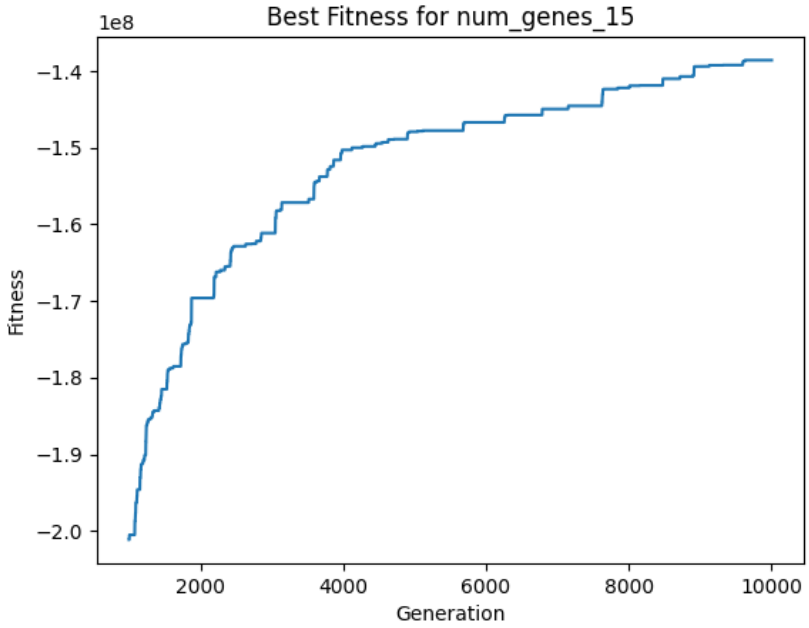


Figure 17. The Fitness Plot Generation (1000-10000) for num_genes=15

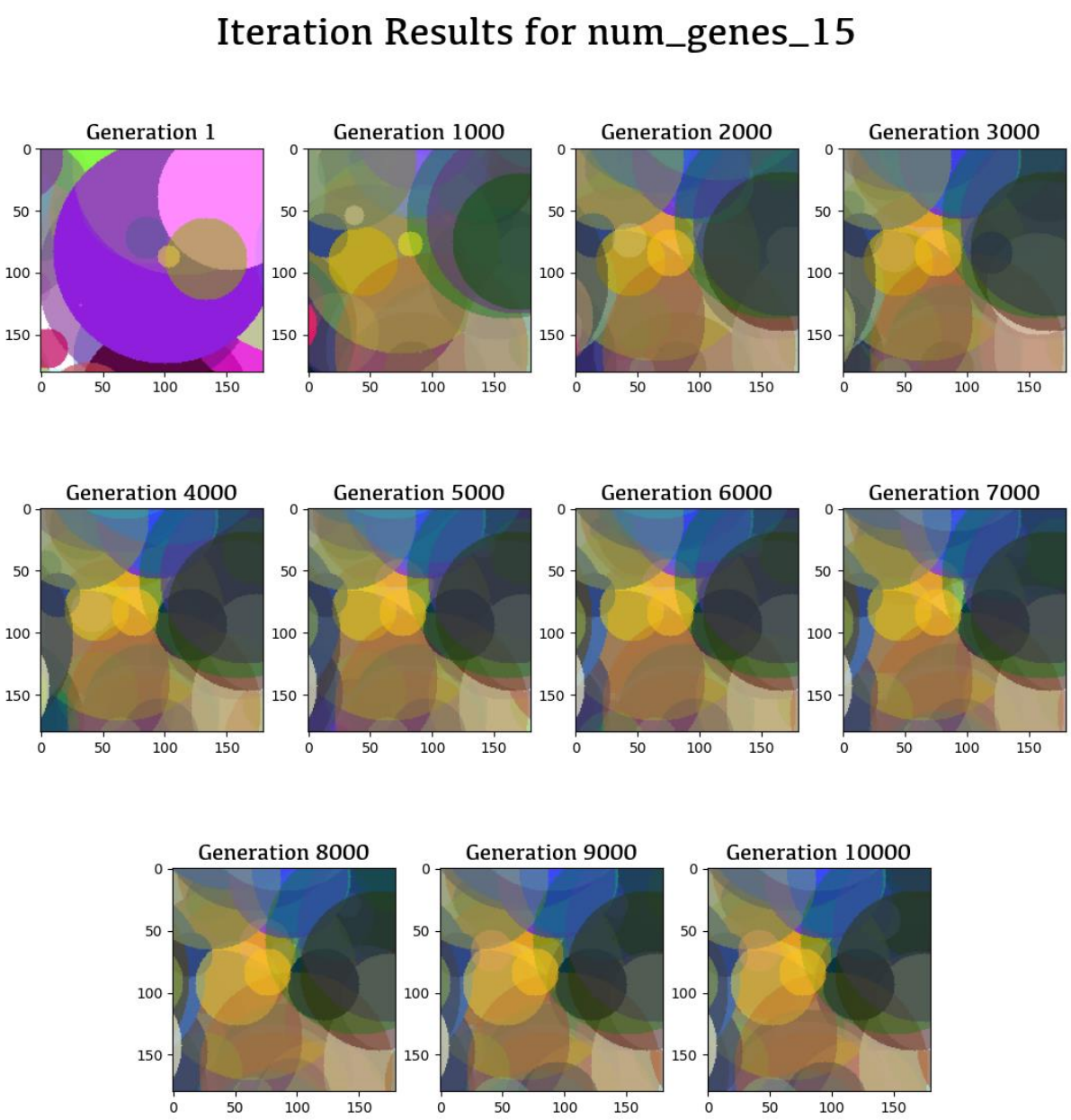


Figure 18. Corresponding Images of Best Individuals for num_genes=15

The best fitness value is *-138564758.*

2.3.2. Num_genes = 30

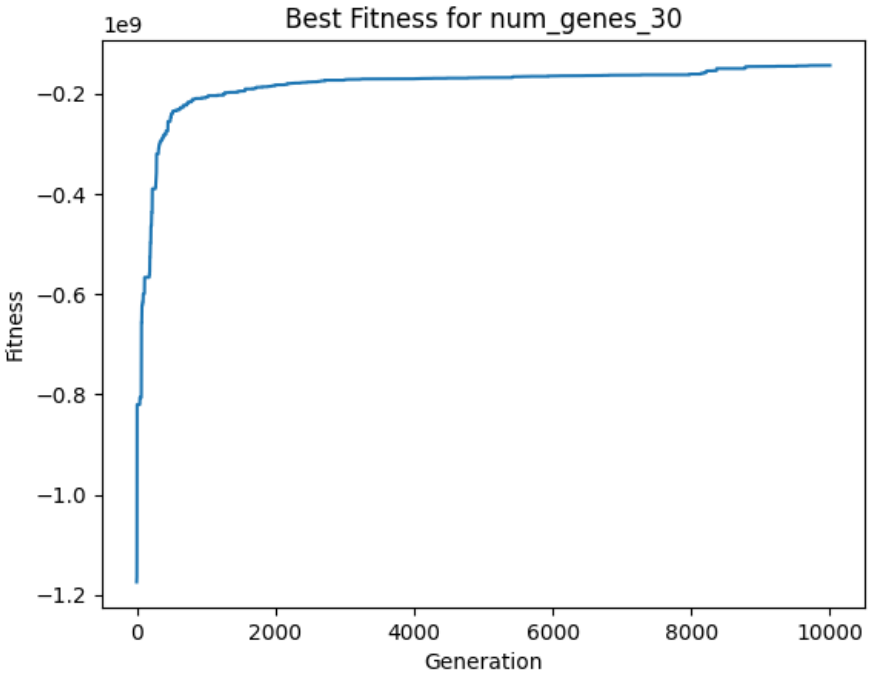


Figure 19. The Fitness Plot Generation (1-10000) for num_genes=30

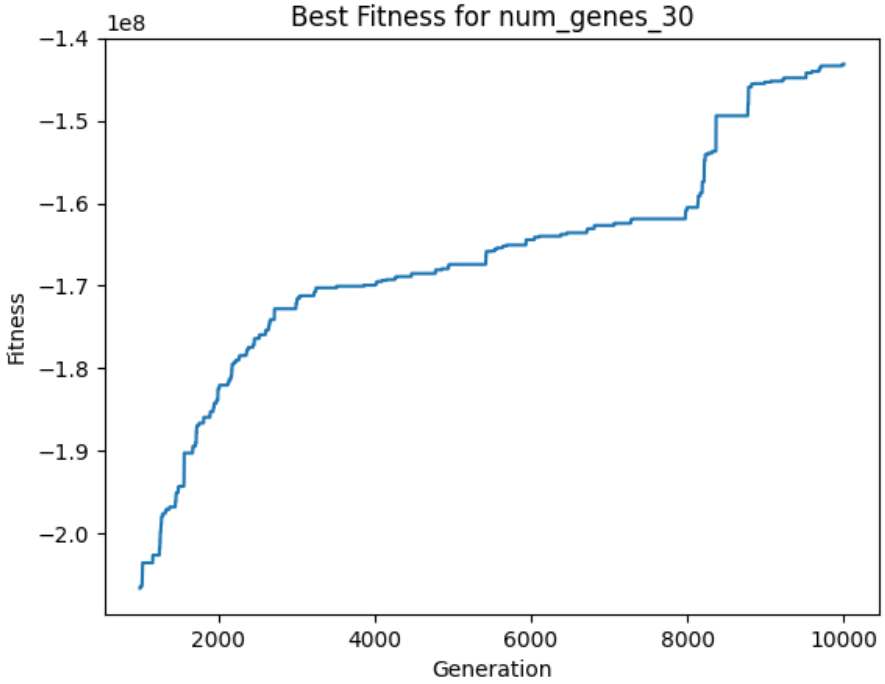


Figure 20. The Fitness Plot Generation (1000-10000) for num_genes=30

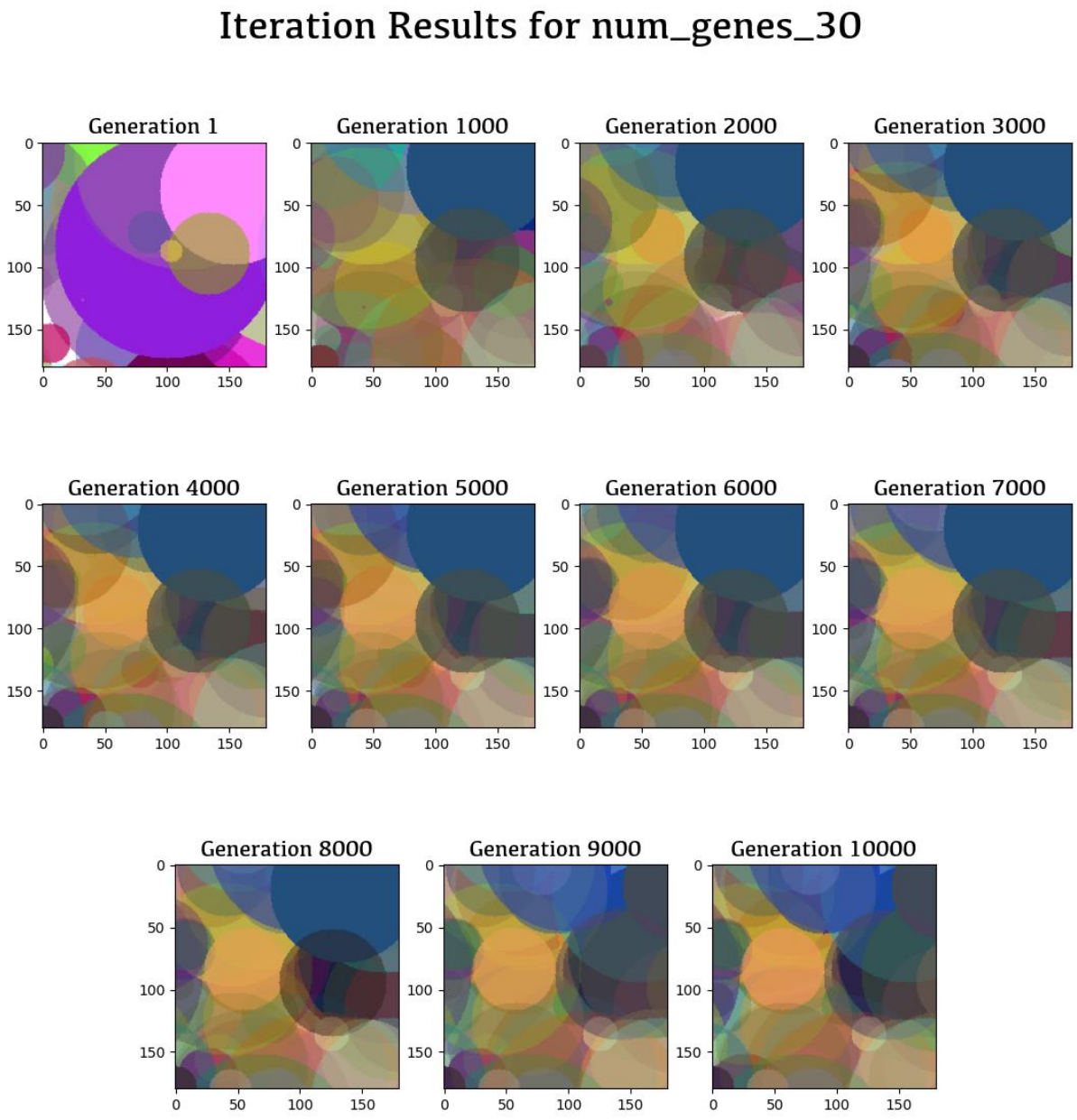


Figure 21. Corresponding Images of Best Individuals for num_genes=30

The best fitness value is -*143149421.*

2.3.3. Num_genes = 80

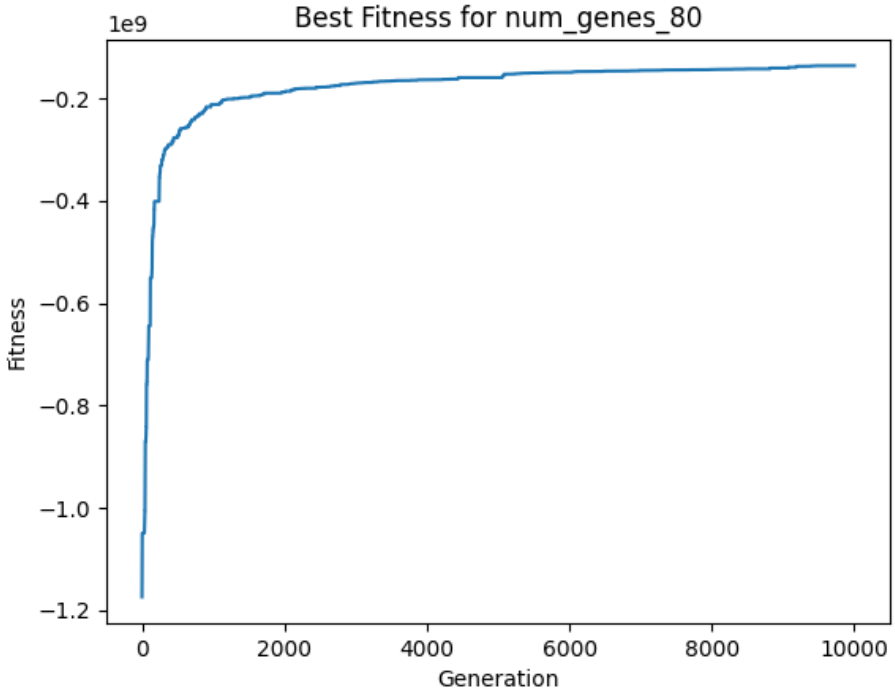


Figure 22. The Fitness Plot Generation (1-10000) for num_genes=80

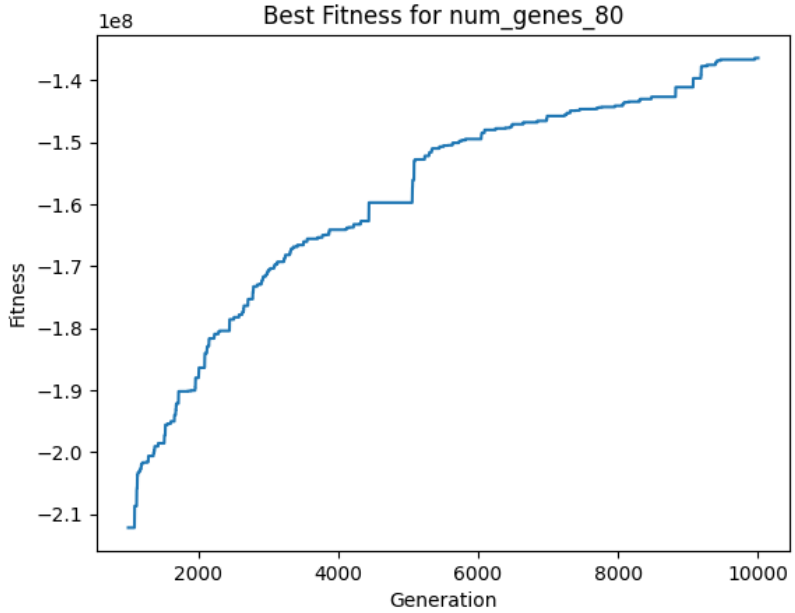


Figure 23. The Fitness Plot Generation (1000-10000) for num_genes=80

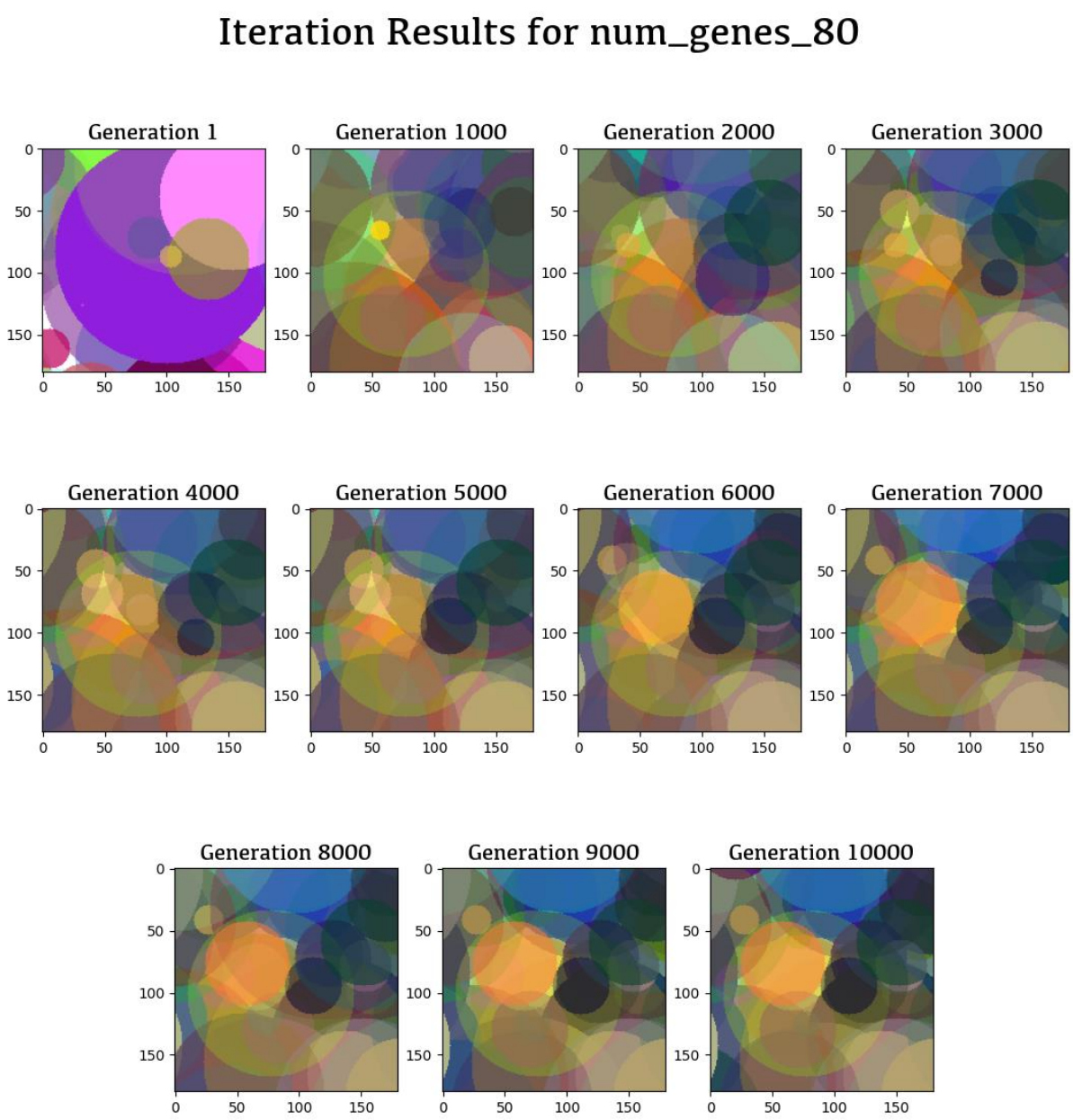


Figure 24. Corresponding Images of Best Individuals for num_genes=80

The best fitness value is -*136389845*.

2.3.4. Num_genes = 120

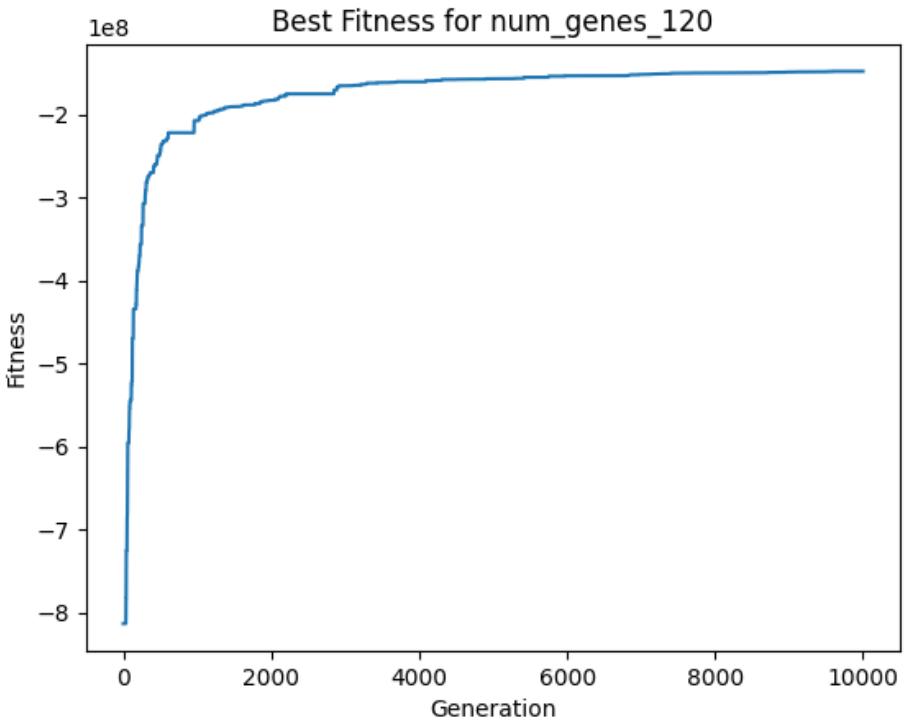


Figure 25. The Fitness Plot Generation (1-10000) for num_genes=120

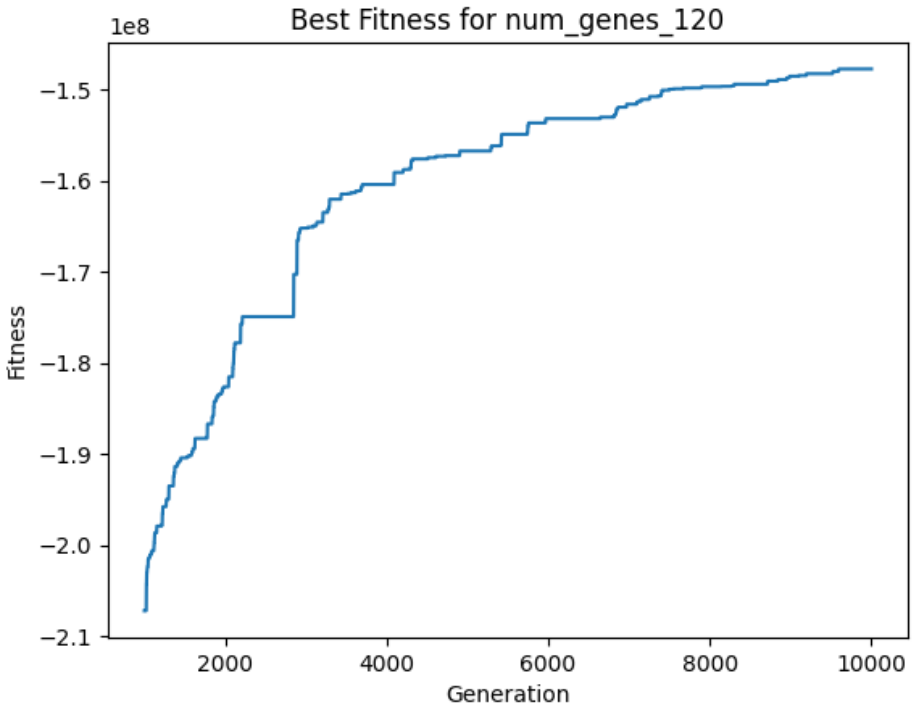


Figure 26. The Fitness Plot Generation (1000-10000) for num_genes=120

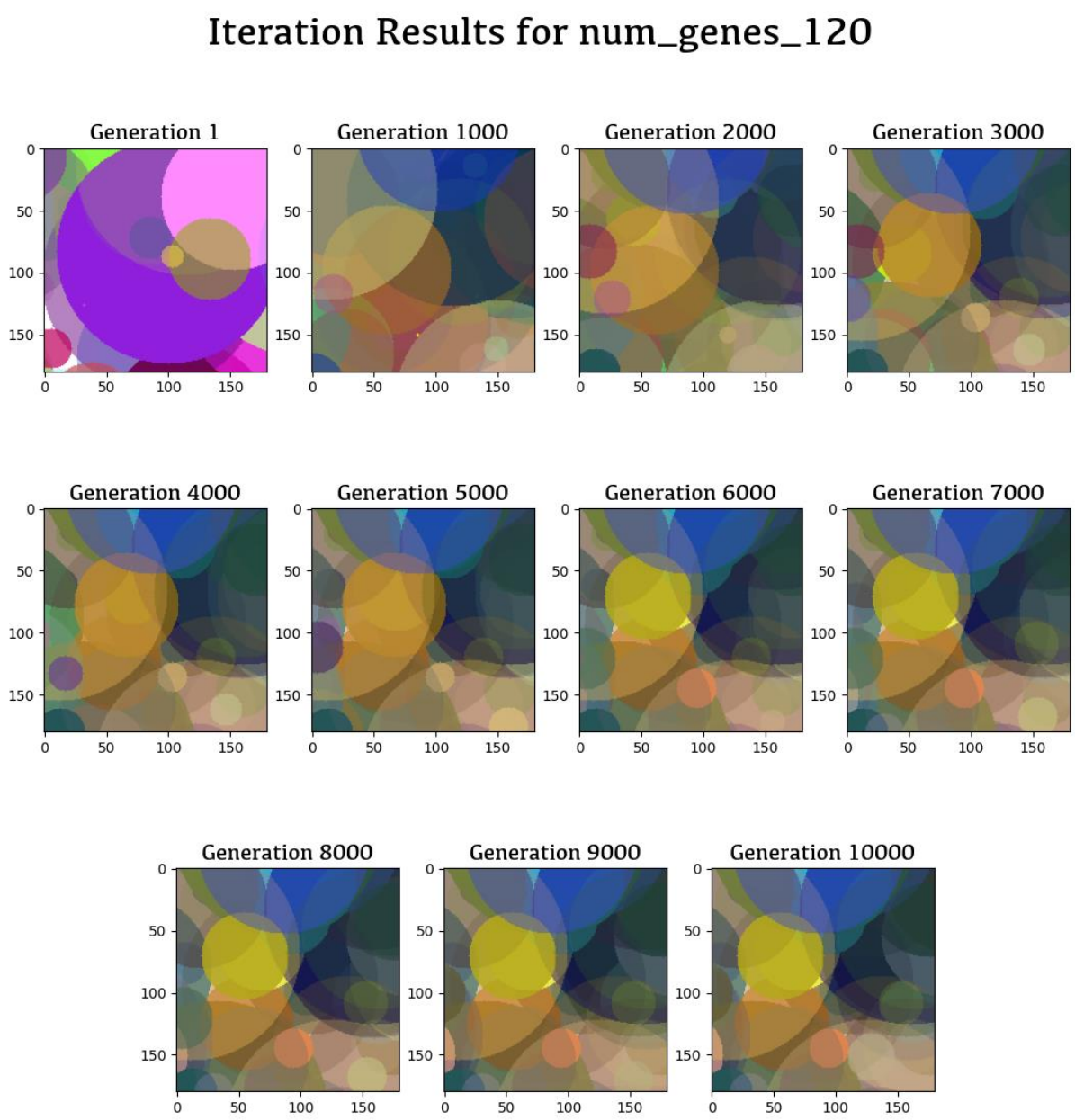


Figure 27. Corresponding Images of Best Individuals for num_genes=120

The best fitness value is -147764787.

Upon examination of *Figures (1-3)* and *Figures (16-27),* it is evident that the optimal fitness value is achieved when the default parameter **num_genes = 50 i**s utilized. This is since a small number of genes will result in a lack of diversity in the results, while a larger number of genes will require a longer time to converge to a minimum, ultimately leading to a worse fitness value.

2.4. Tournament Size

2.4.1. Tm_size = 2

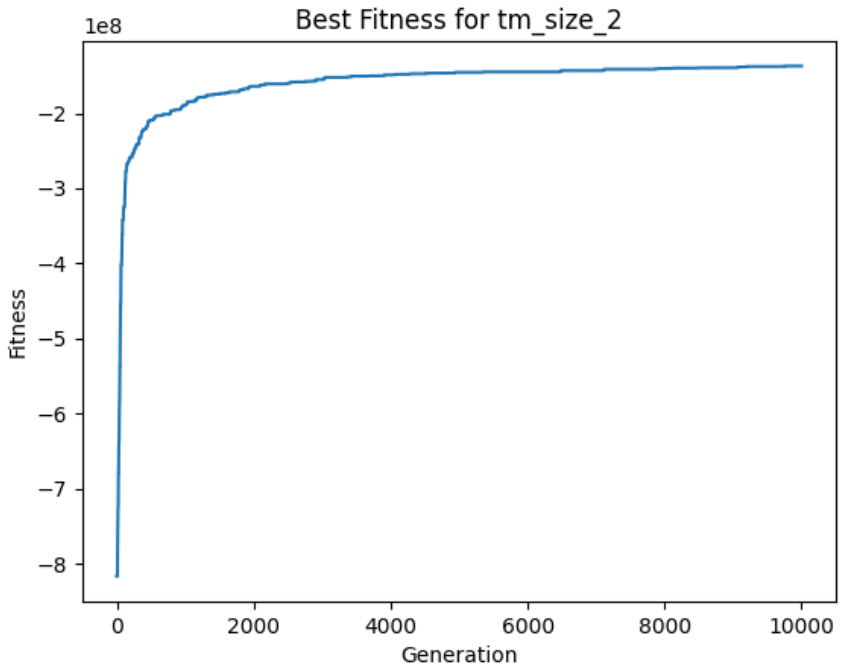


Figure 28. The Fitness Plot Generation (1-10000) for tm_size = 2

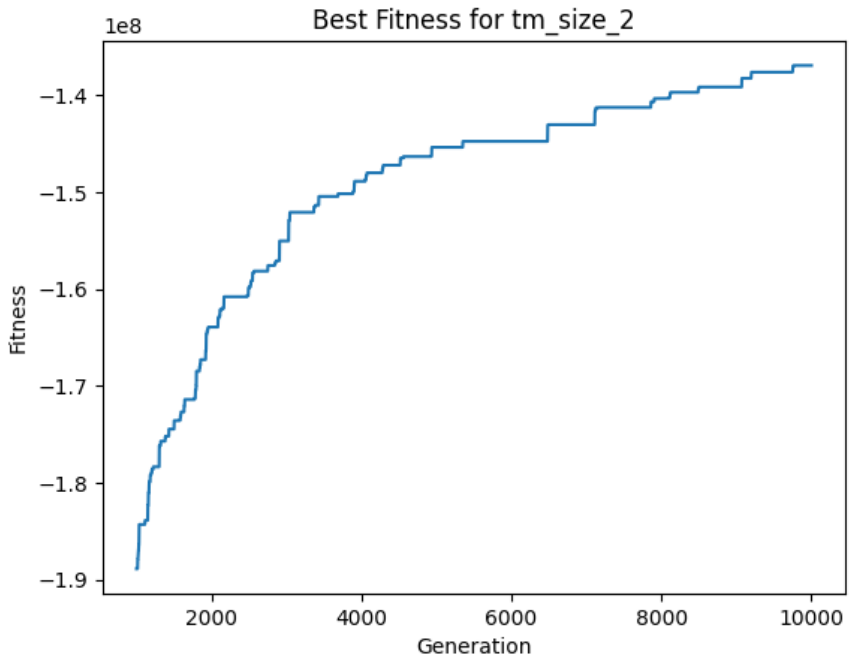


Figure 29. The Fitness Plot Generation (1000-10000) for tm_size = 2

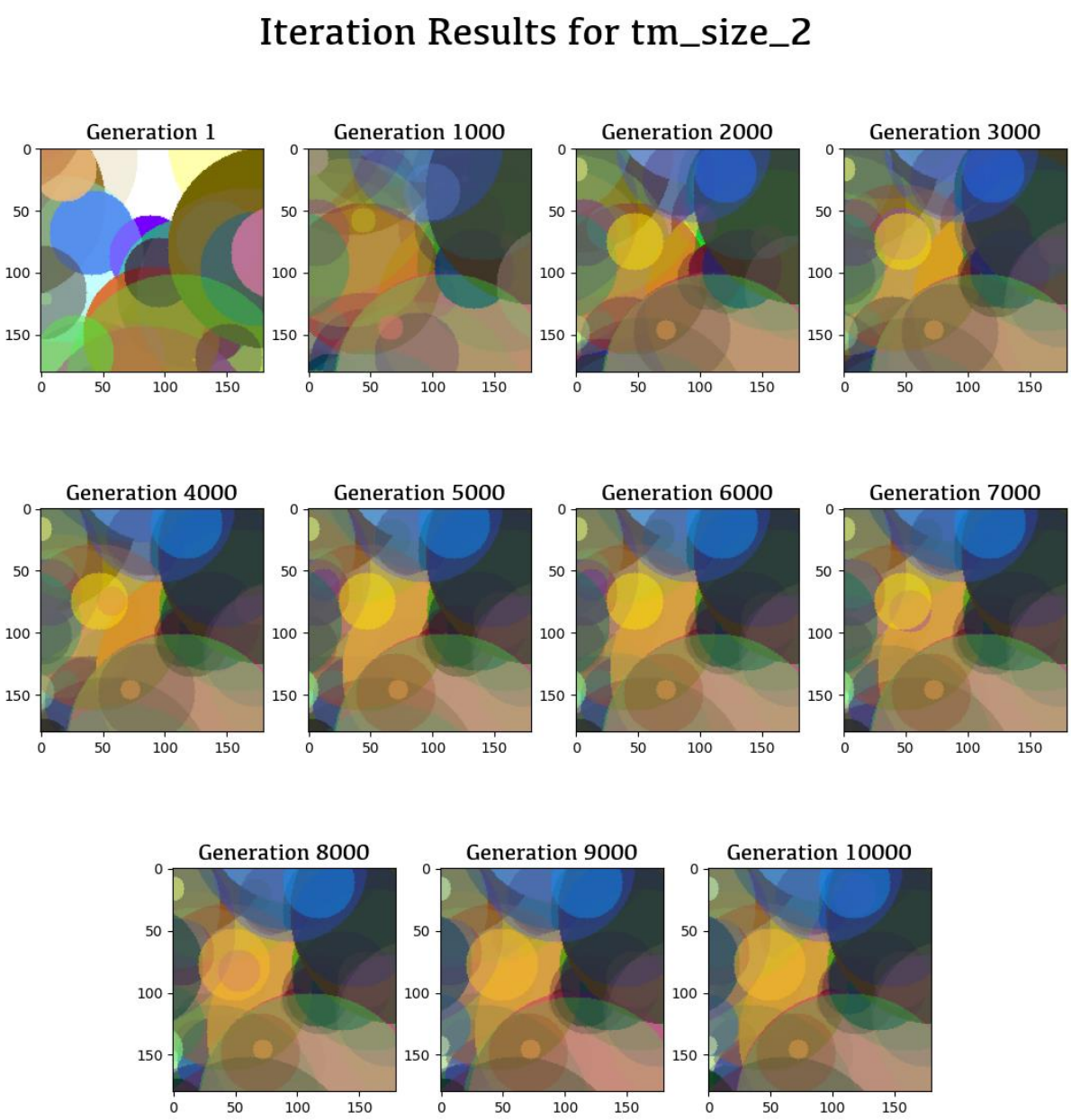


Figure 30. Corresponding Images of Best Individuals for tm_size=2

The best fitness value is *-136895442.*

2.4.2. Tm_size = 8

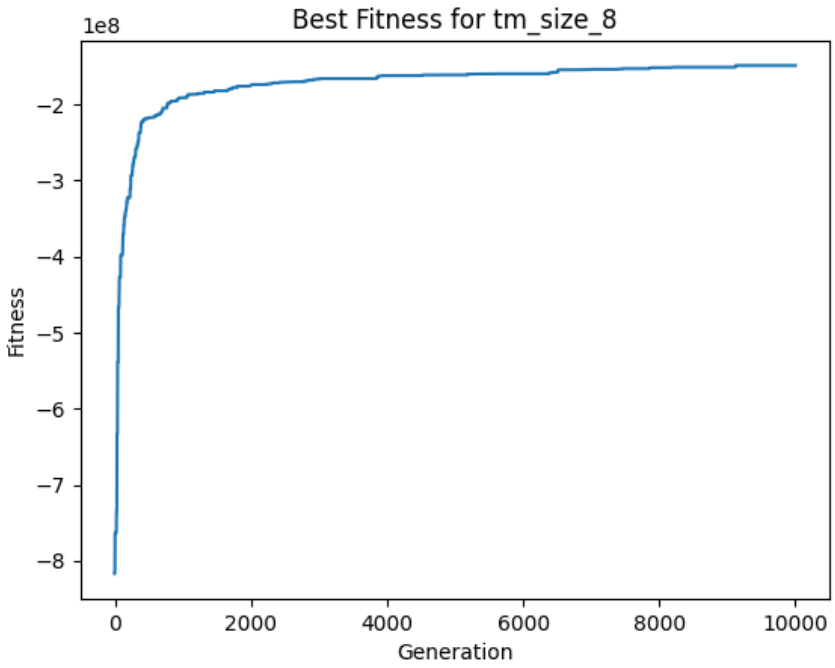


Figure 31. The Fitness Plot Generation (1-10000) for tm_size = 8

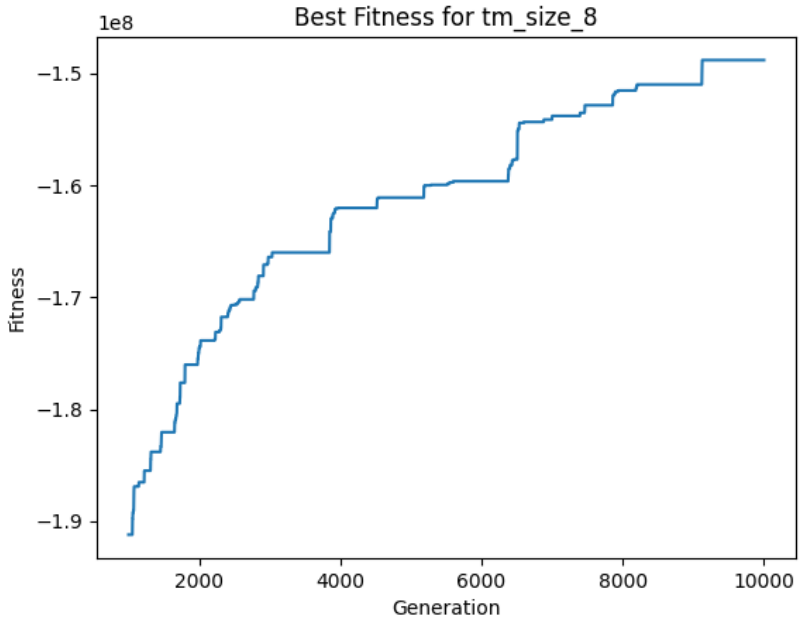


Figure 32. The Fitness Plot Generation (1000-10000) for tm_size = 8

The best fitness value is *-148817617.*

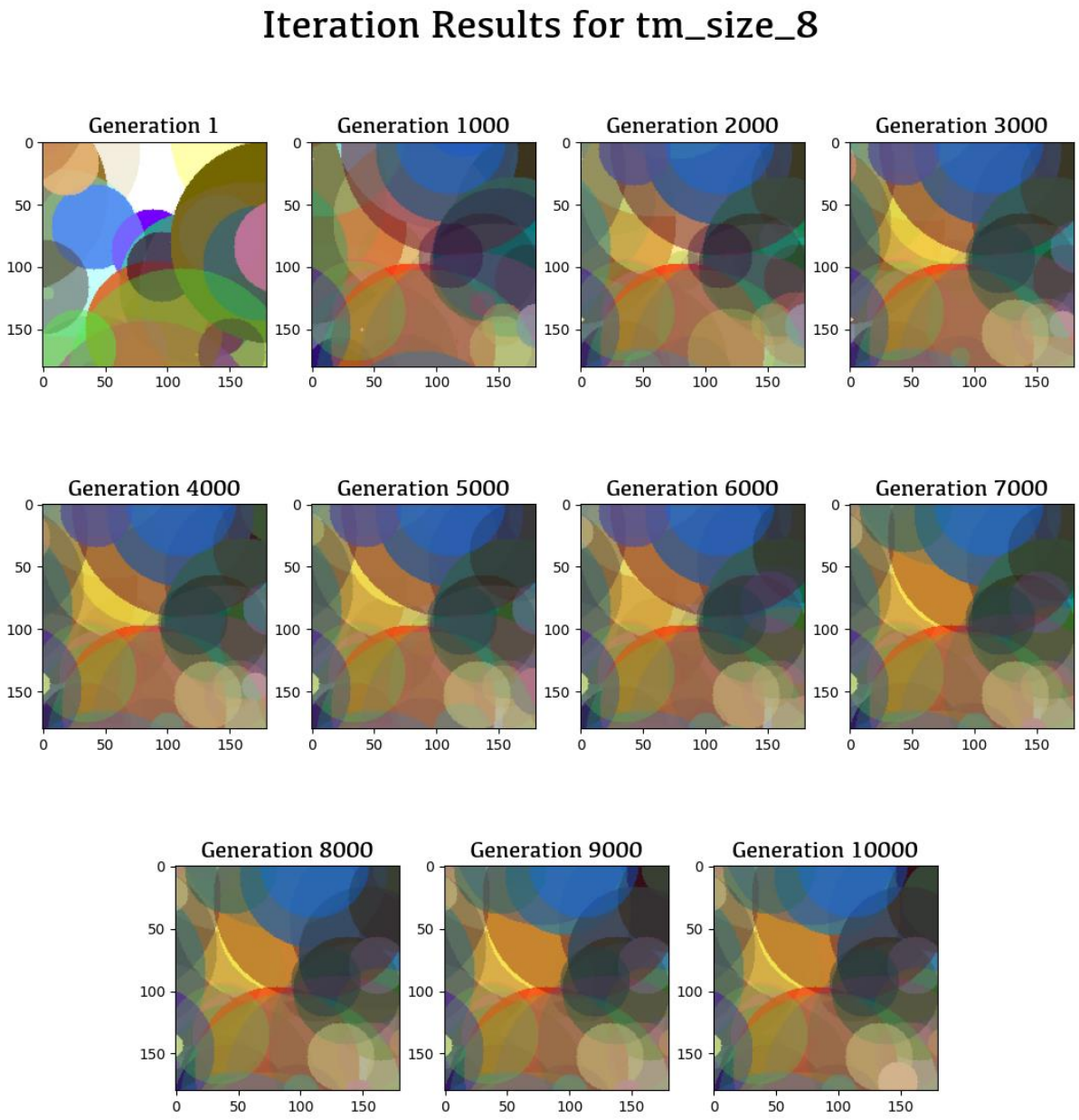


Figure 33. Corresponding Images of Best Individuals for tm_size=8

2.4.3. Tm_size = 16

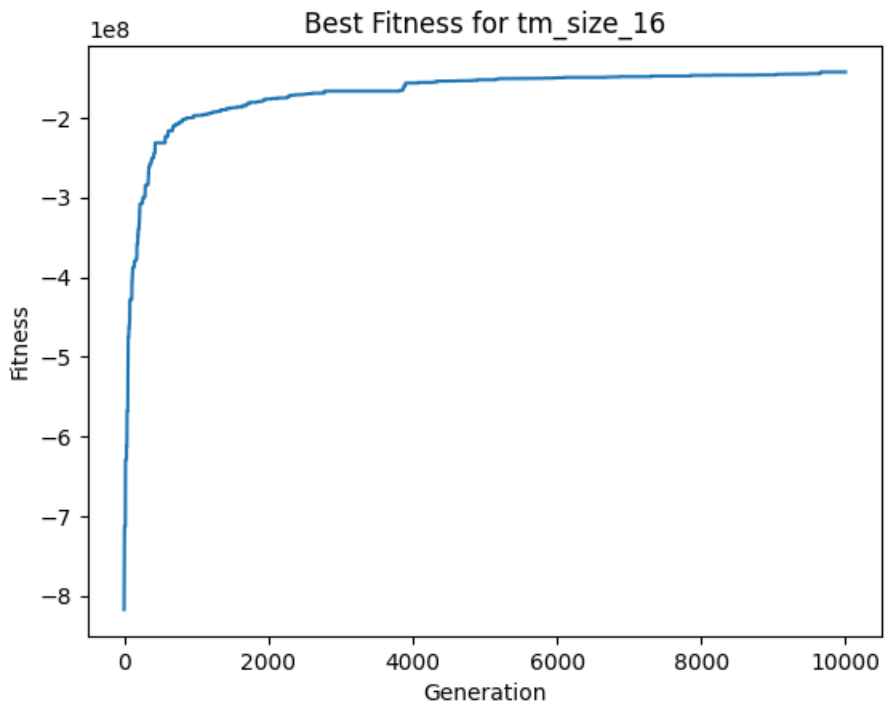


Figure 34. The Fitness Plot Generation (1-10000) for tm_size = 16

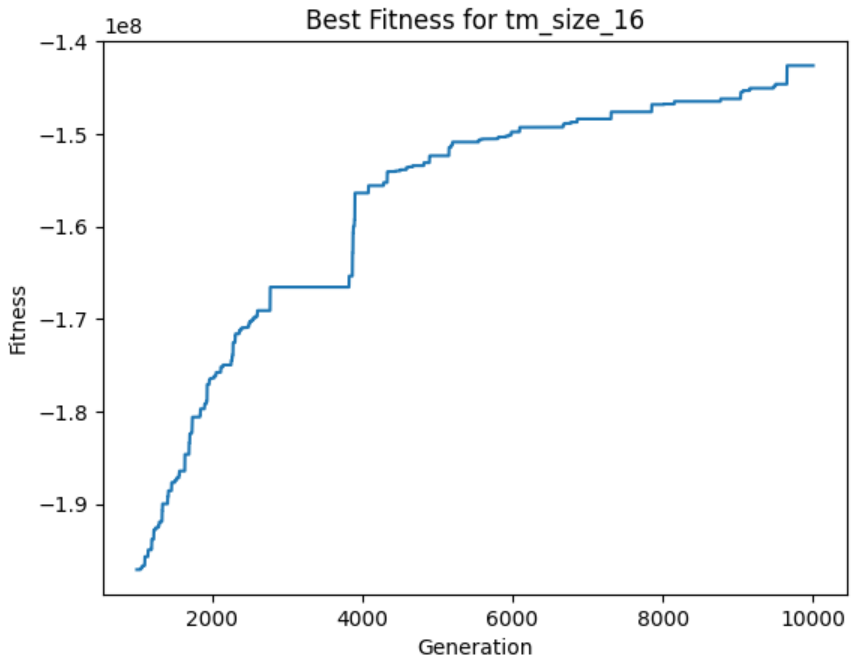


Figure 35. The Fitness Plot Generation (1000-10000) for tm_size = 16

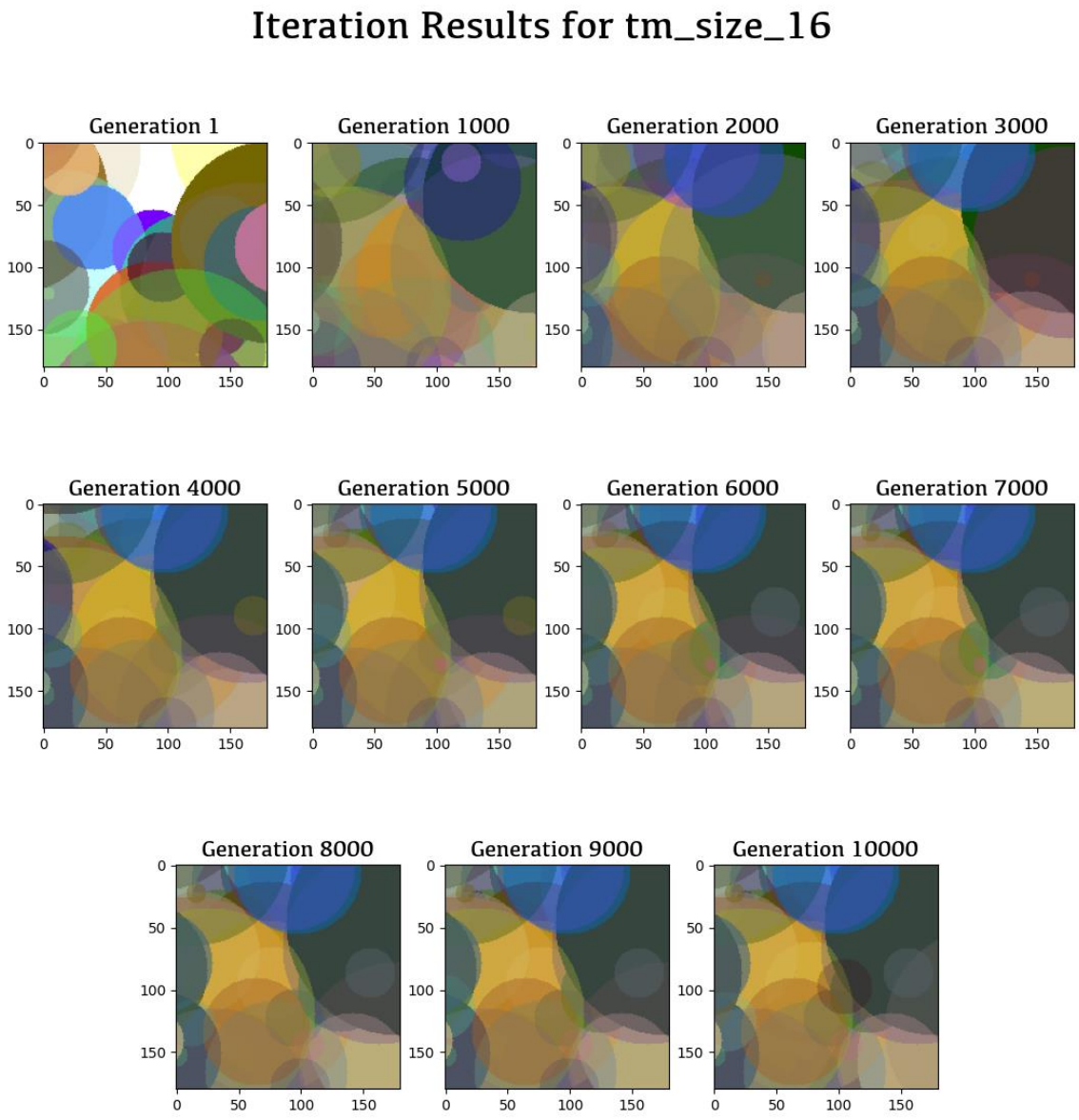


Figure 36. Corresponding Images of Best Individuals for tm_size=16

The best fitness value is *-142582584*.

Upon examination of *Figures (1-3)* and *Figures (28-36)*, it is evident that the optimal fitness value is achieved when the default parameter **tm_size = 5** is utilized. This is since a small tournament size will result in a lack of diversity in the results, while a large tournament size will require a longer time to converge to a minimum, ultimately leading to a worse fitness value.

2.5. Fraction of Elites (Number of Elites)

2.5.1. Fraction_elites = 0.04

Number_of_elites = 0 since I used integer for number of elites. int(20*0.04) = 0

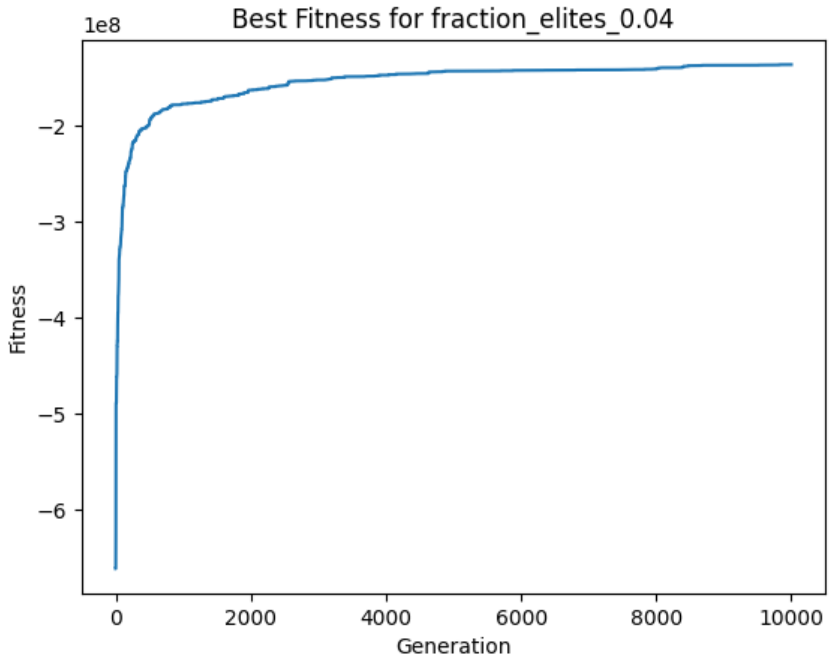


Figure 37. The Fitness Plot Generation (1-10000) for fraction_elites=0.04

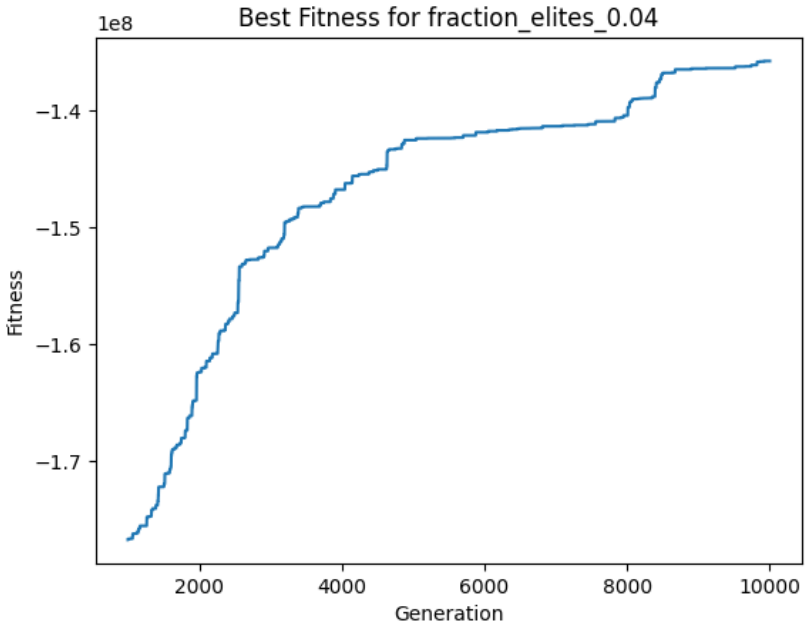


Figure 38. The Fitness Plot Generation (1000-10000) for fraction_elites=0.04

The best fitness value is -

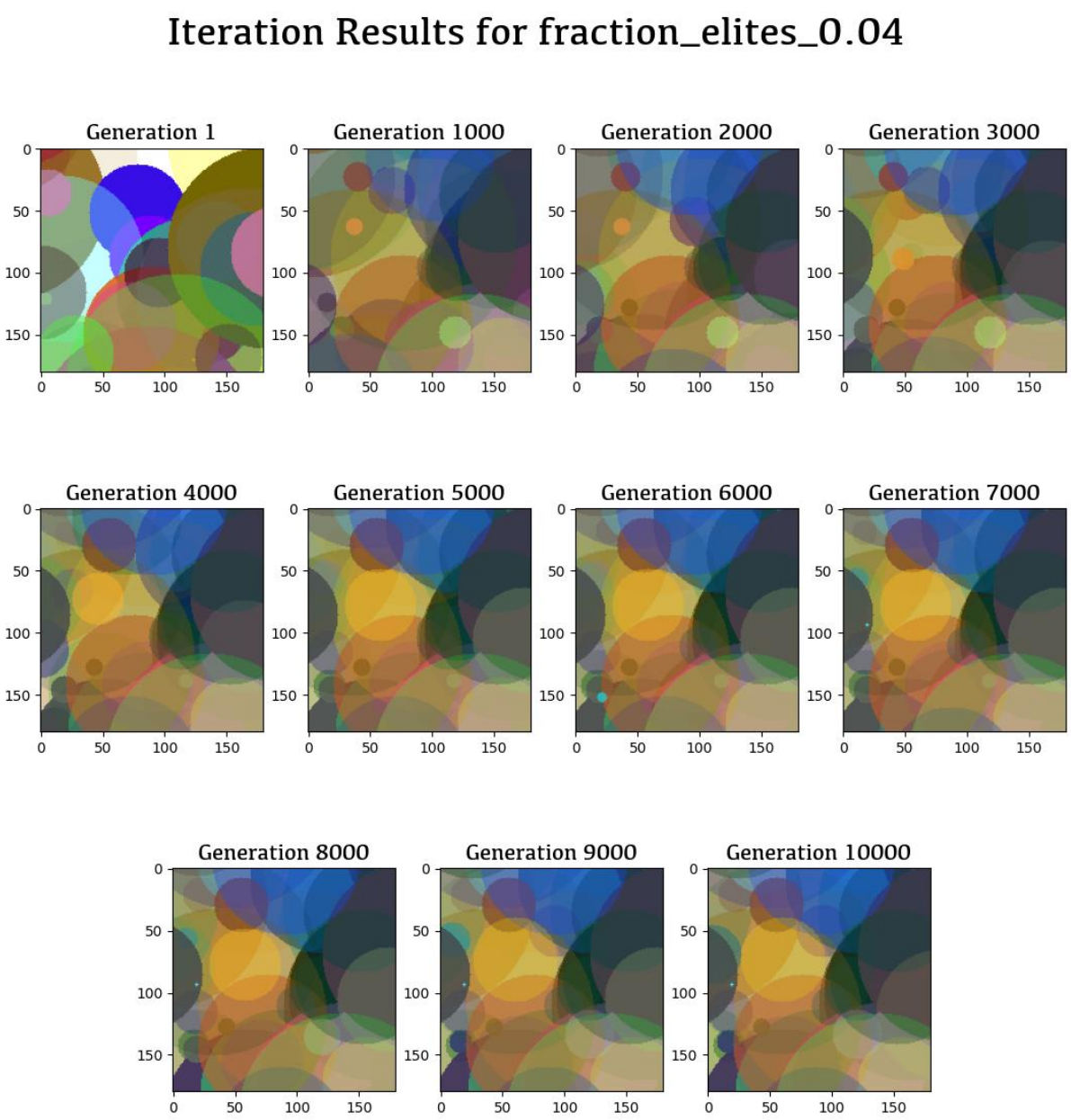


Figure 39. Corresponding Images of Best Individuals for fraction_elites=0.04

135769786

2.5.2. Fraction_elites = 0.35

Number_of_elites = 7 since I used integer for number of elites. int(20*0.35) = 7

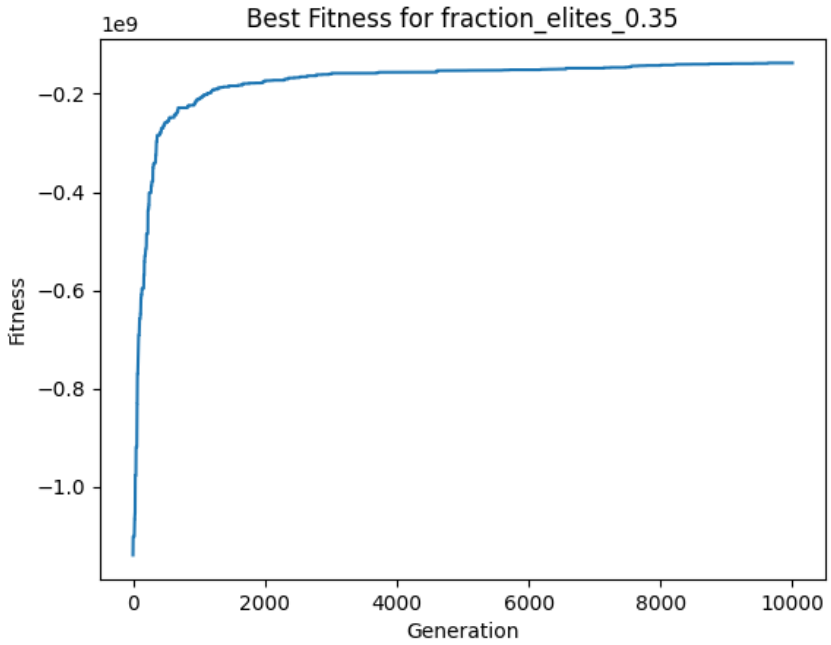


Figure 40. The Fitness Plot Generation (1-10000) for fraction_elites=0.35

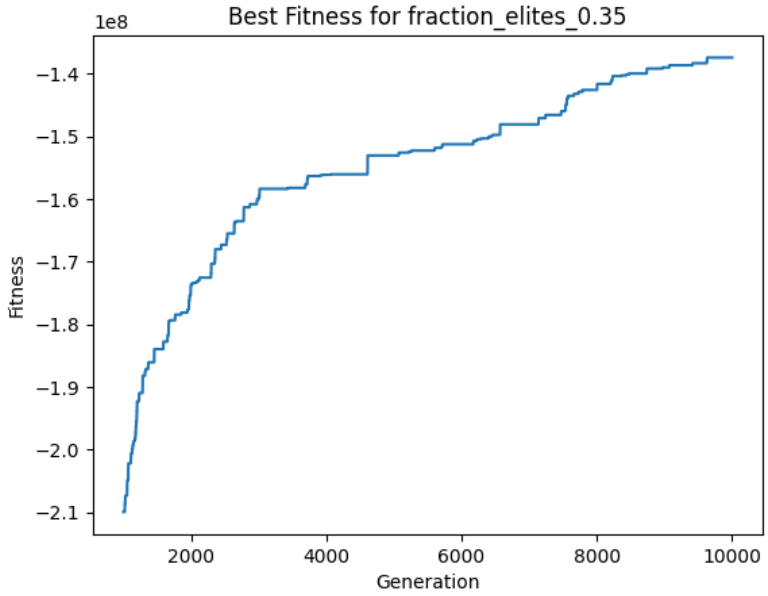


Figure 41. The Fitness Plot Generation (1000-10000) for fraction_elites=0.35

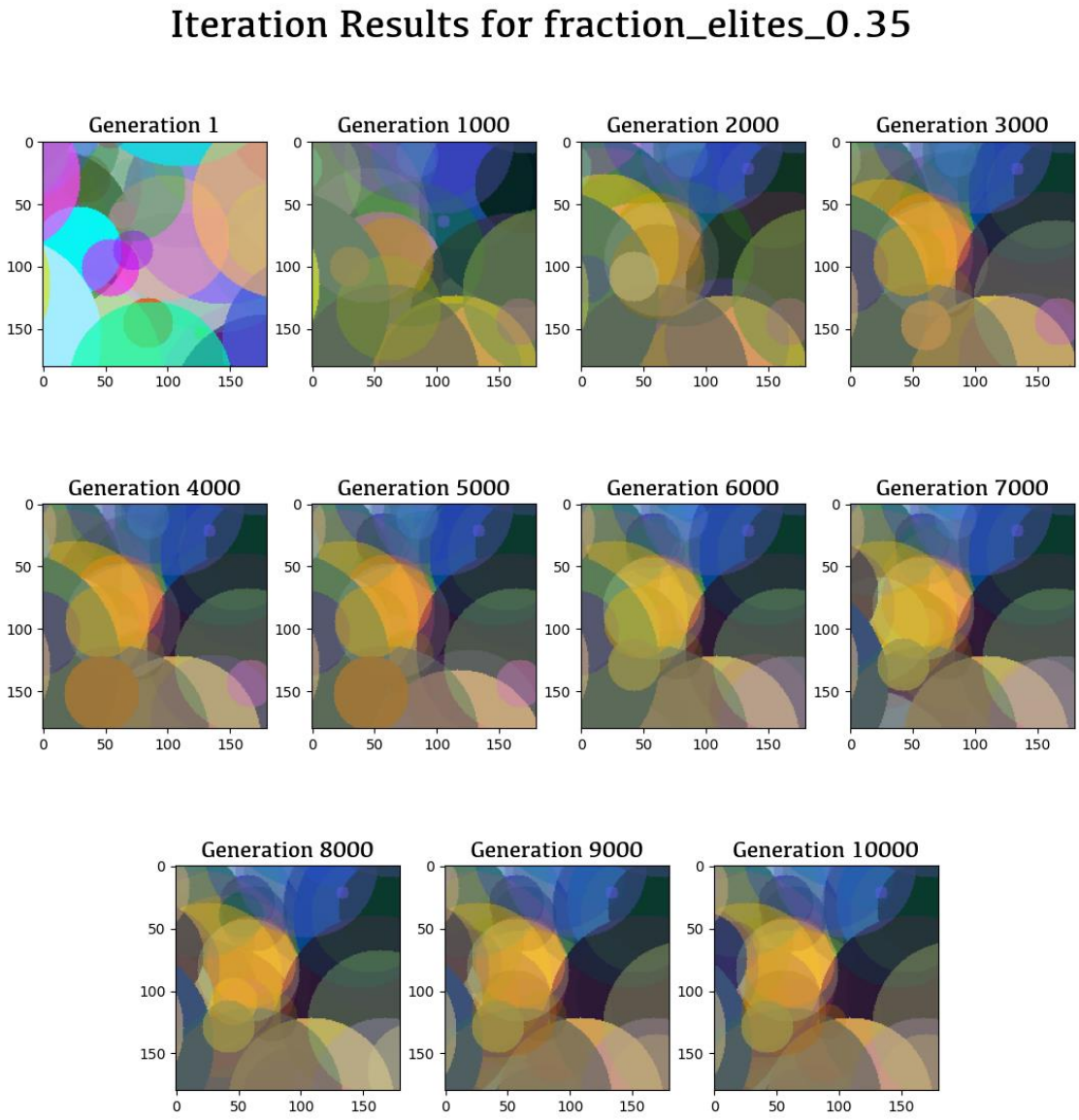


Figure 42. Corresponding Images of Best Individuals for fraction_elites=0.35

The best fitness value is *-142582584.*
Upon examination of *Figures (1-3)* and *Figures (37-42)*, it is evident that the optimal fitness value is achieved when the default parameter **fraction_elites = 0.2** is utilized. This is since a larger number of elites will result in lack of diversity in the results (it is due to the fact that elites will directly pass to the next stage), while a smaller number of elites will require a longer time to converge to a minimum, ultimately leading to a worse fitness value.

2.6. Fraction of Parents (Number of Parents)

2.6.1. Fraction_parents = 0.15

Number_of_parents = 3 since I used integer for number of elites. int(20*0.15) = 3

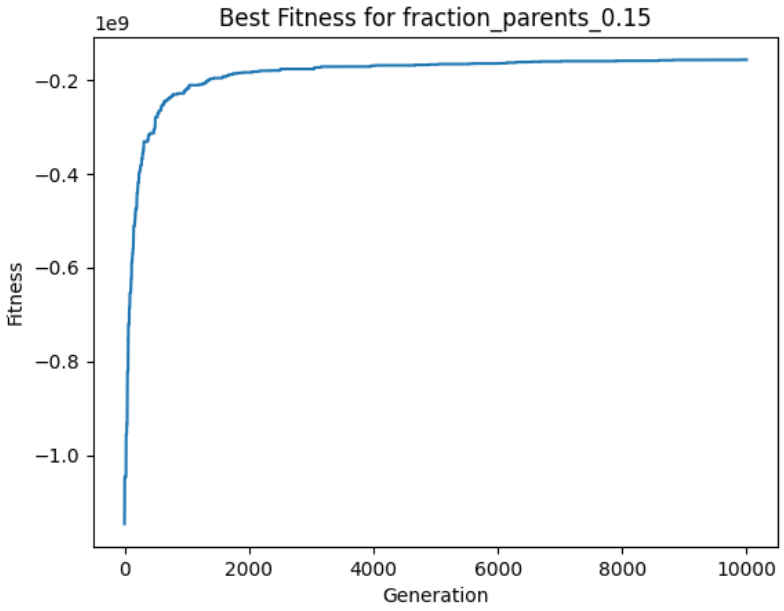


Figure 43. The Fitness Plot Generation (1-10000) for fraction_parents=0.15

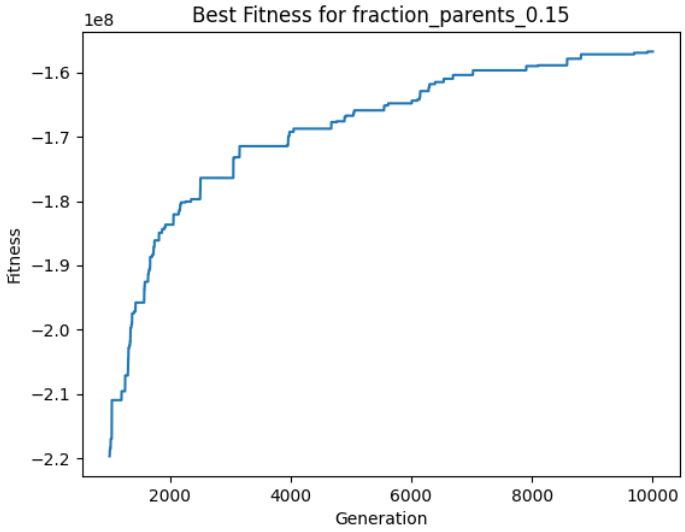


Figure 44. The Fitness Plot Generation (1000-10000) for fraction_parents=0.15

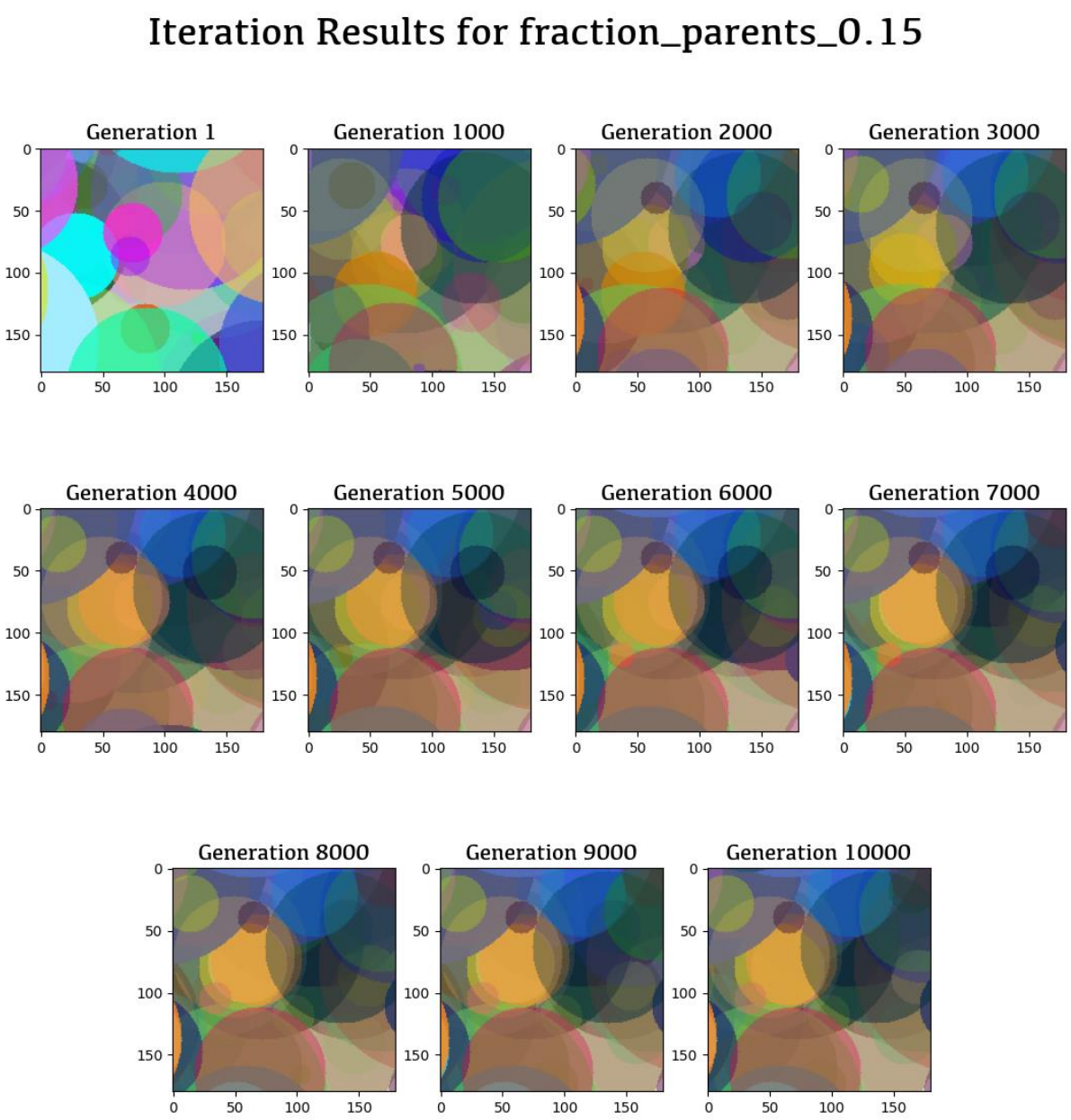


Figure 45. Corresponding Images of Best Individuals for fraction_parents=0.15

The best fitness value is *-156761065*

2.6.2. Fraction_parents = 0.3

Number_of_parents = 6 since I used integer for number of elites. int(20*0.3) = 6

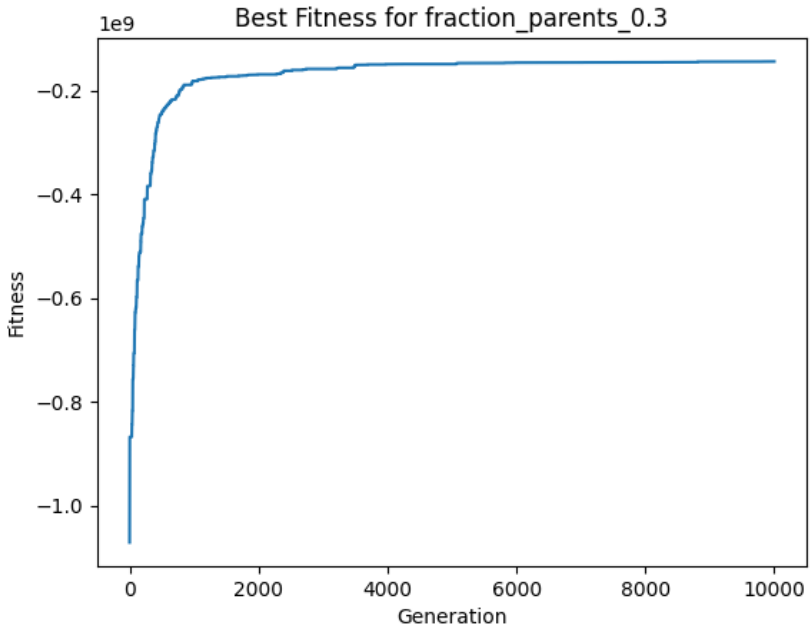


Figure 46. The Fitness Plot Generation (1-10000) for fraction_parents=0.3

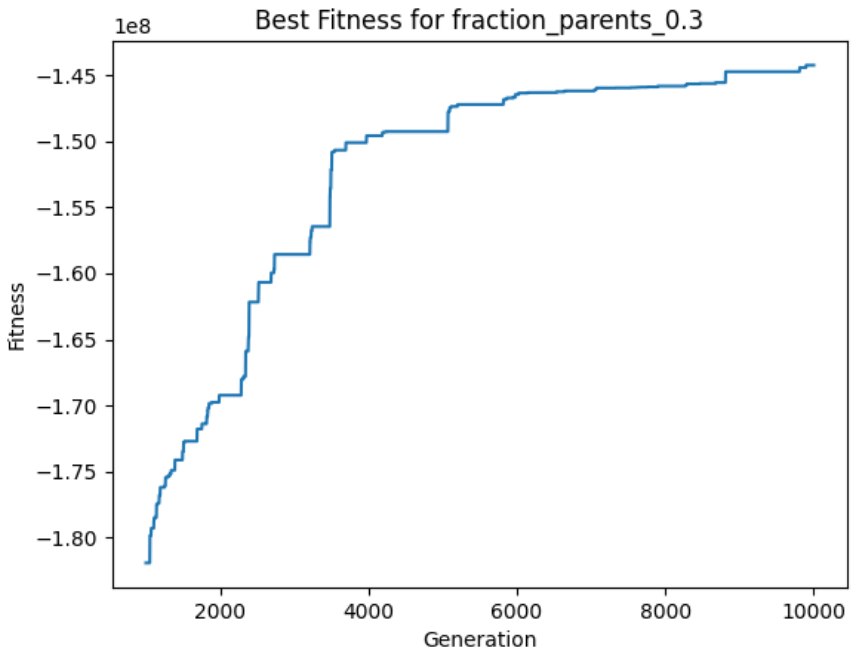


Figure 47. The Fitness Plot Generation (1000-10000) for fraction_parents=0.3

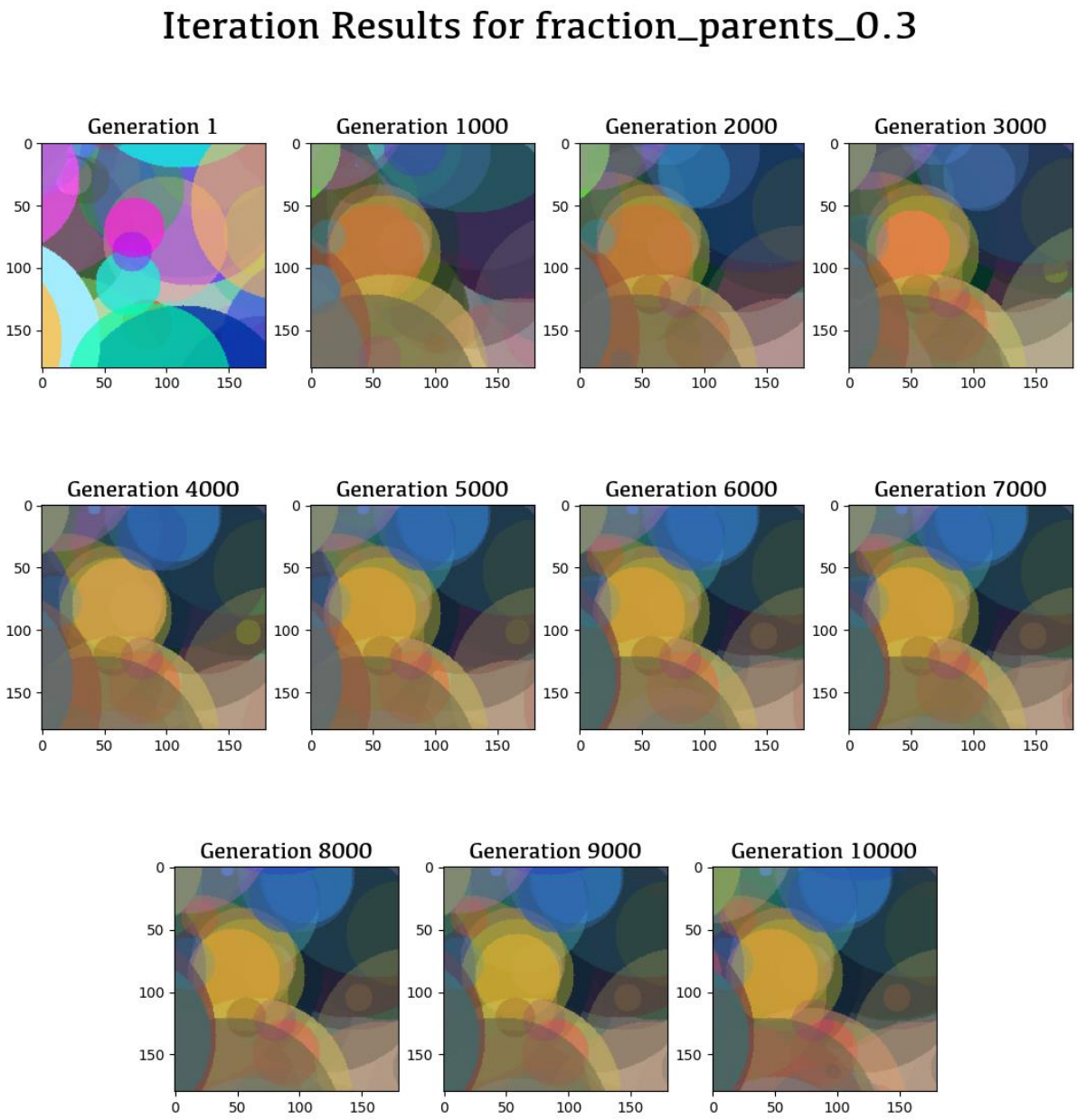


Figure 48. Corresponding Images of Best Individuals for fraction_parents=0.3

The best fitness value is *-144259625*

2.6.3. Fraction_parents = 0.75

Number_of_parents = 15 since I used integer for number of elites.
int(20*0.75) = 15

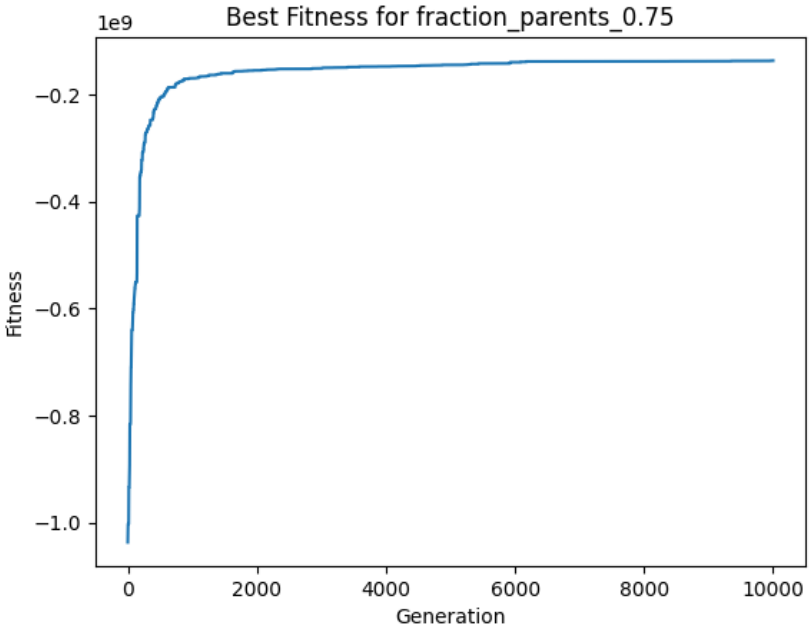


Figure 49. The Fitness Plot Generation (1-10000) for fraction_parents=0.75

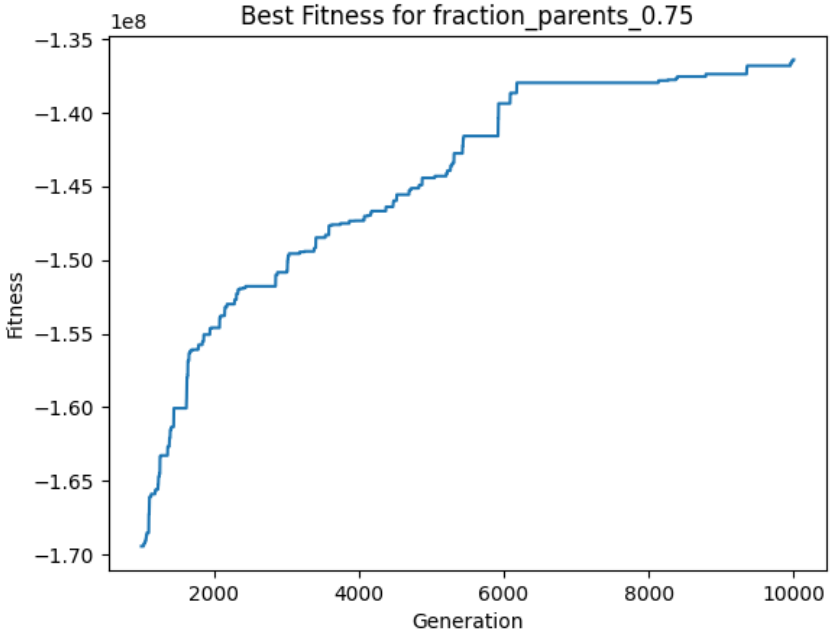


Figure 50. The Fitness Plot Generation (1000-10000) for fraction_parents=0.75

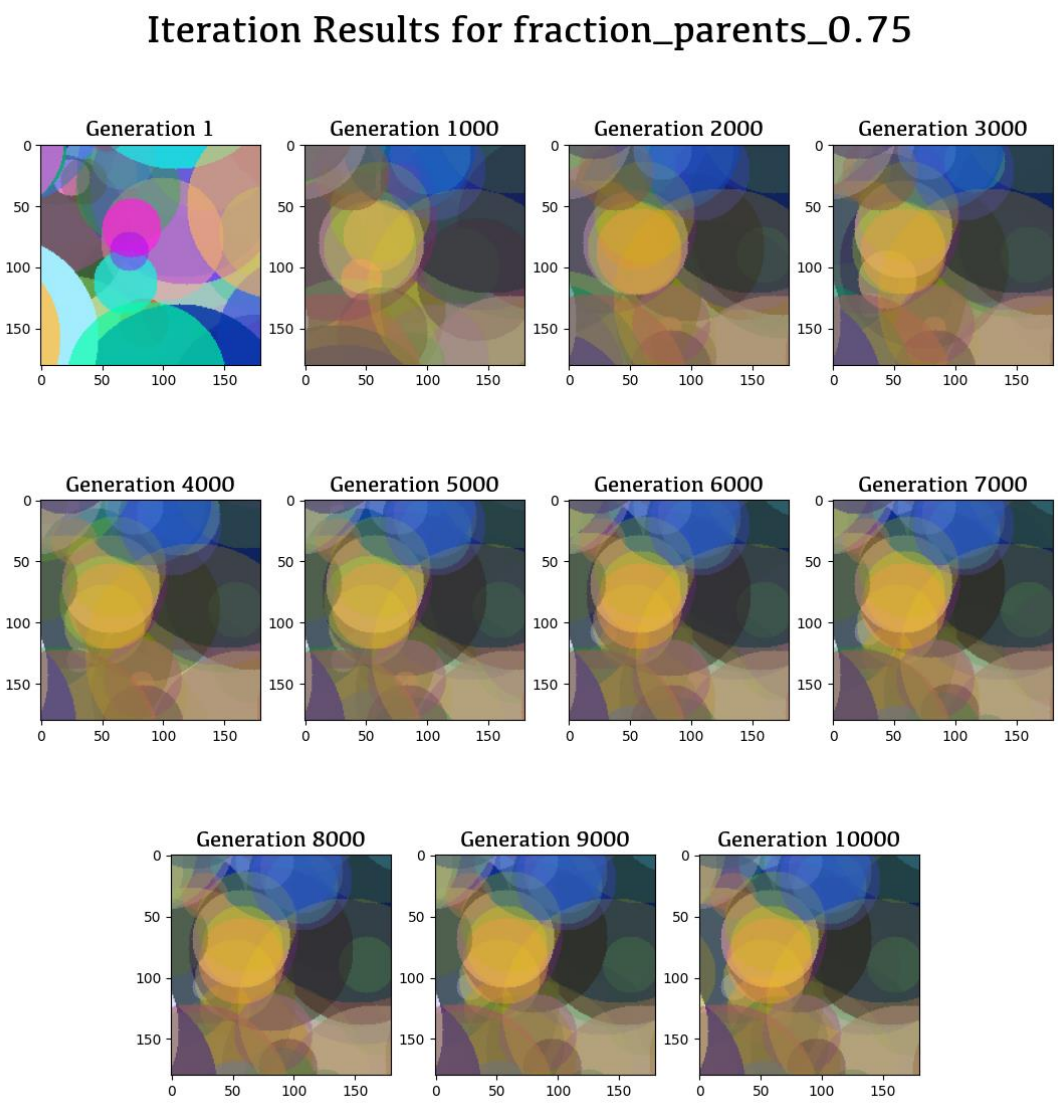


Figure 51. Corresponding Images of Best Individuals for fraction_parents=0.75

The best fitness value is – *136396908.*
Upon examination of *Figures (1-3)* and *Figures (44-51)*, it is evident that the optimal fitness value is achieved when the default parameter **fraction_parents=0.6** is utilized. This is since a smaller number of parents will result in lack of diversity in the results, while a larger number of parents will require a longer time to converge to a minimum, ultimately leading to a worse fitness value most of the time.

2.7. Mutation Probability

2.7.1. Mutation_prob = 0.1

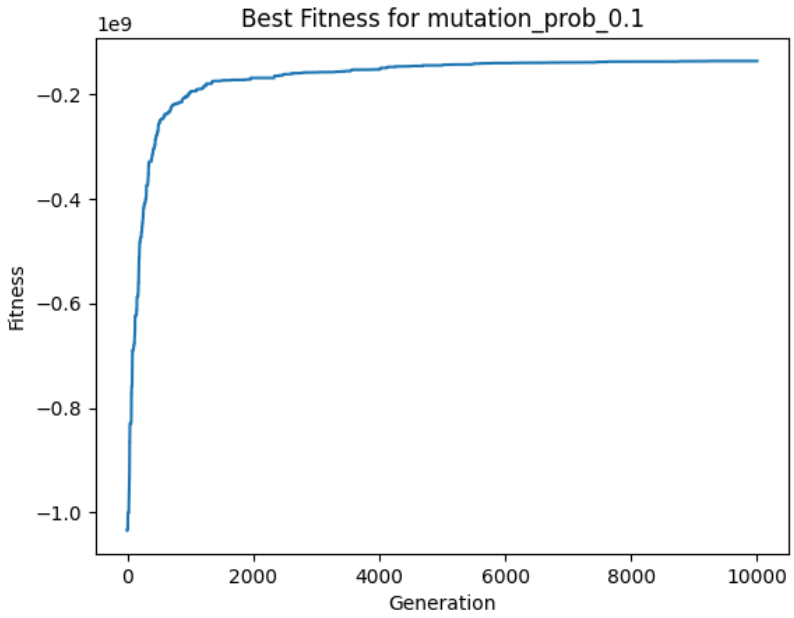


Figure 52. The Fitness Plot Generation (1-10000) for mutation_prob=0.1

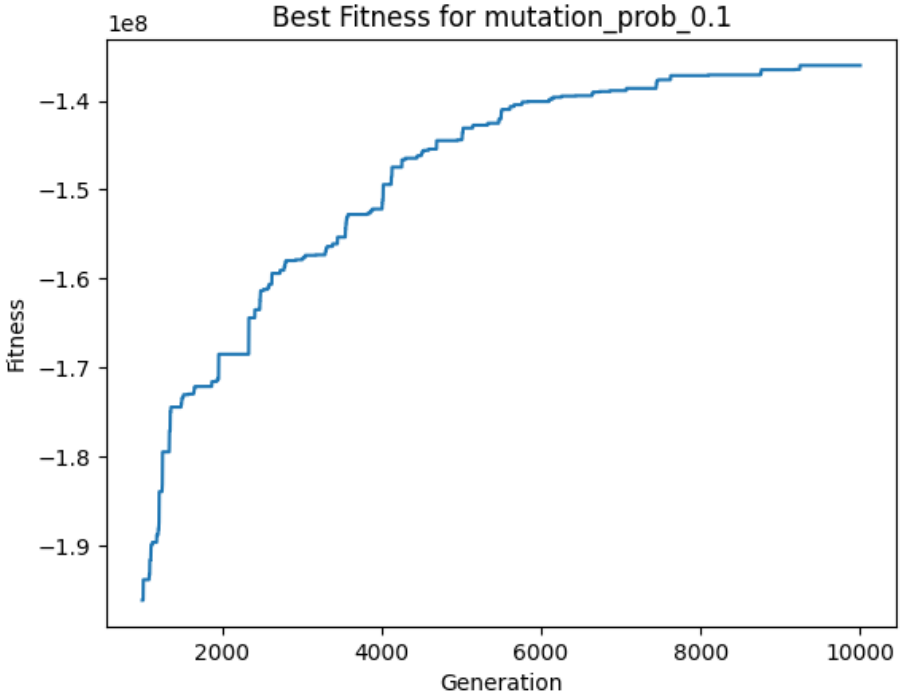


Figure 53. The Fitness Plot Generation (1000-10000) for mutation_prob=0.1

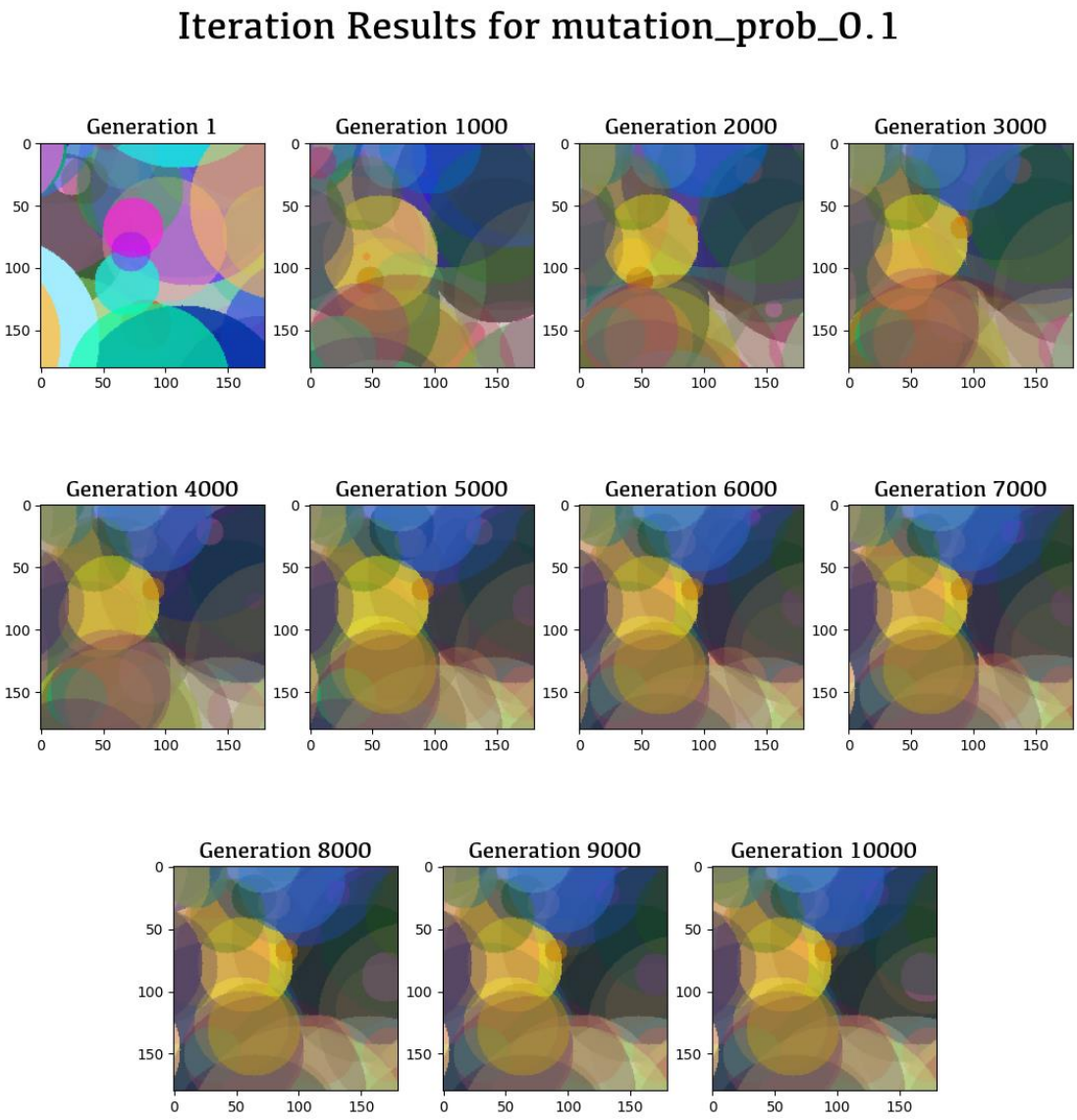


Figure 54. Corresponding Images of Best Individuals for mutation_prob=0.1

The best fitness value is *-136036560.*

2.7.2. Mutation_prob = 0.4

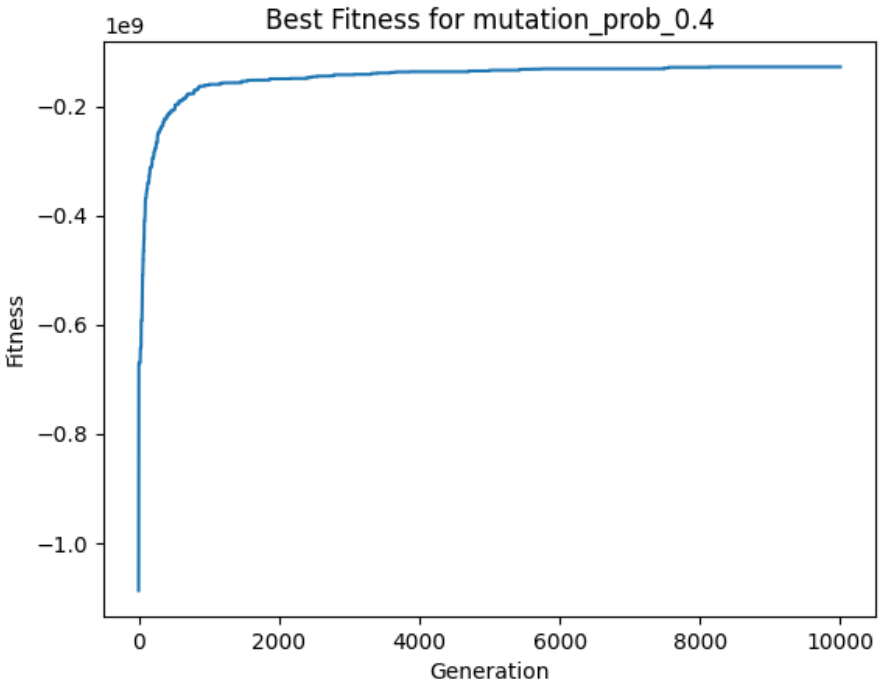


Figure 55. The Fitness Plot Generation (1-10000) for mutation_prob=0.4

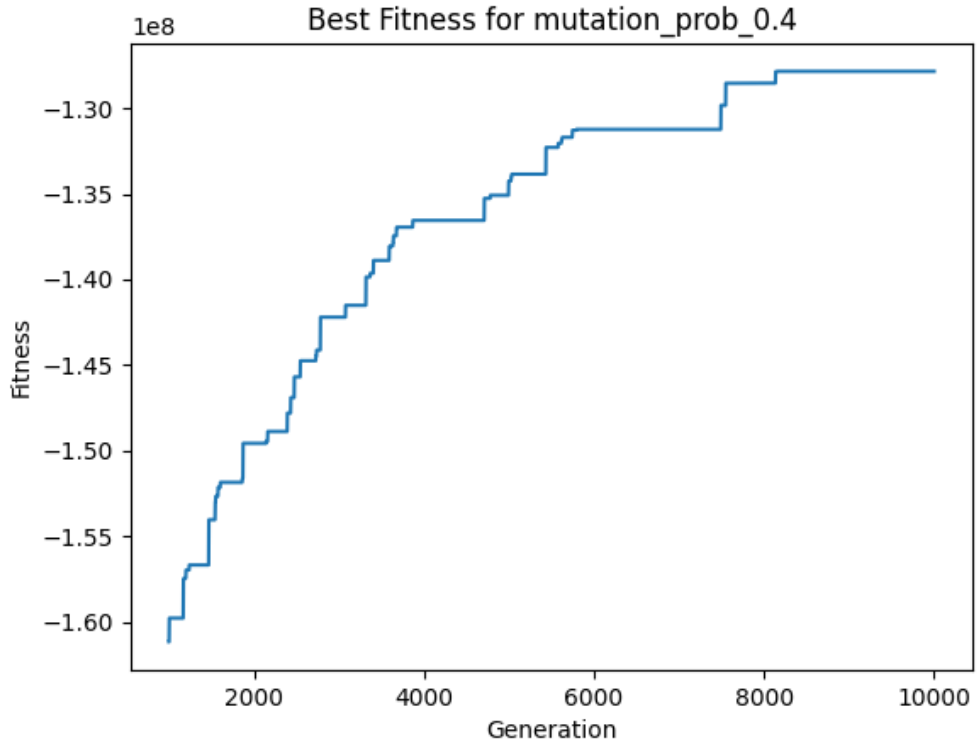


Figure 56. The Fitness Plot Generation (1000-10000) for mutation_prob=0.4

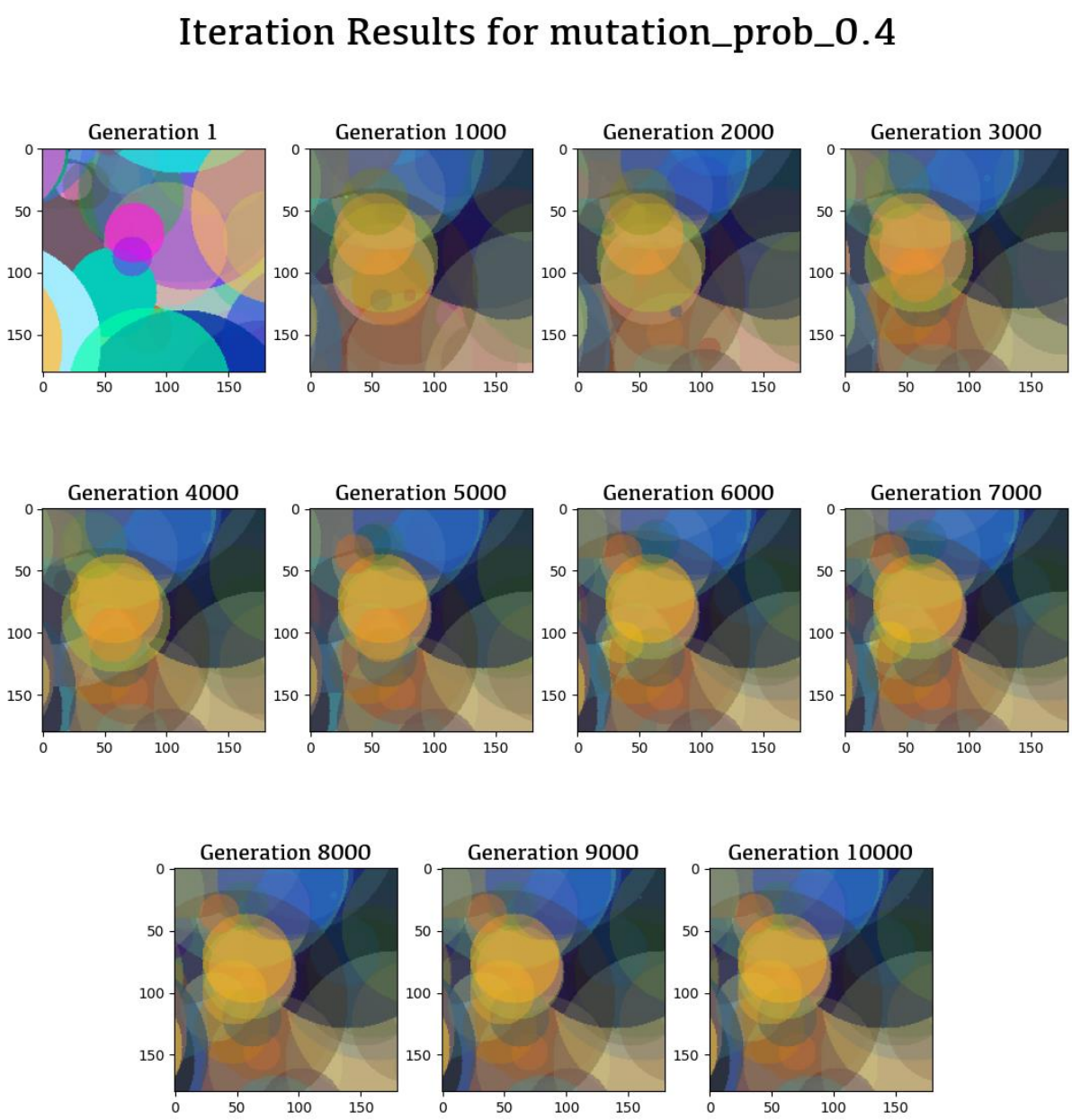


Figure 57. Corresponding Images of Best Individuals for mutation_prob=0.4

The best fitness value is -*127844707.*

2.7.3. Mutation_prob = 0.75

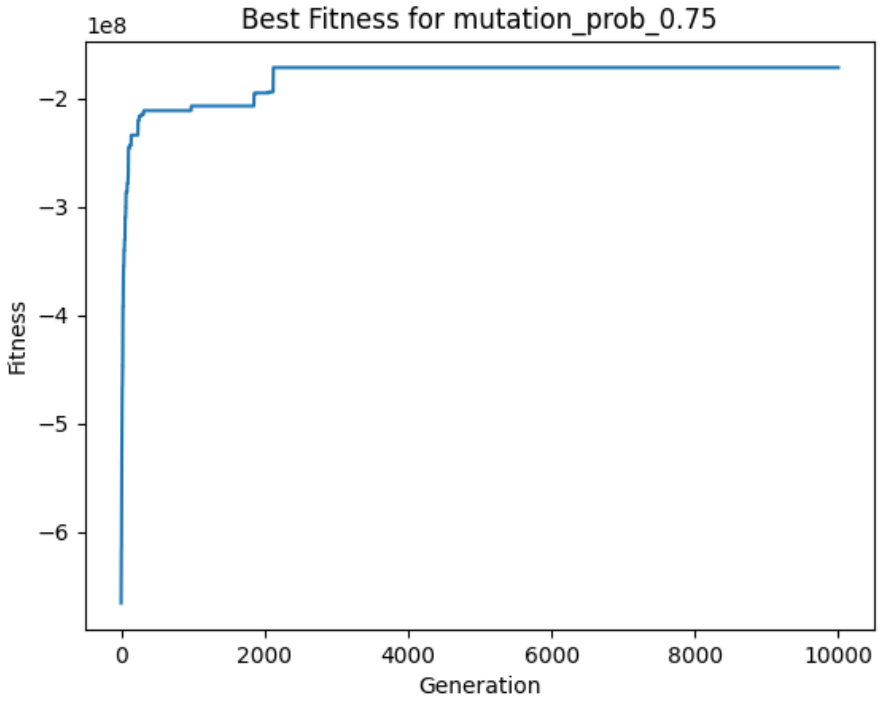


Figure 58. The Fitness Plot Generation (1-10000) for mutation_prob=0.75

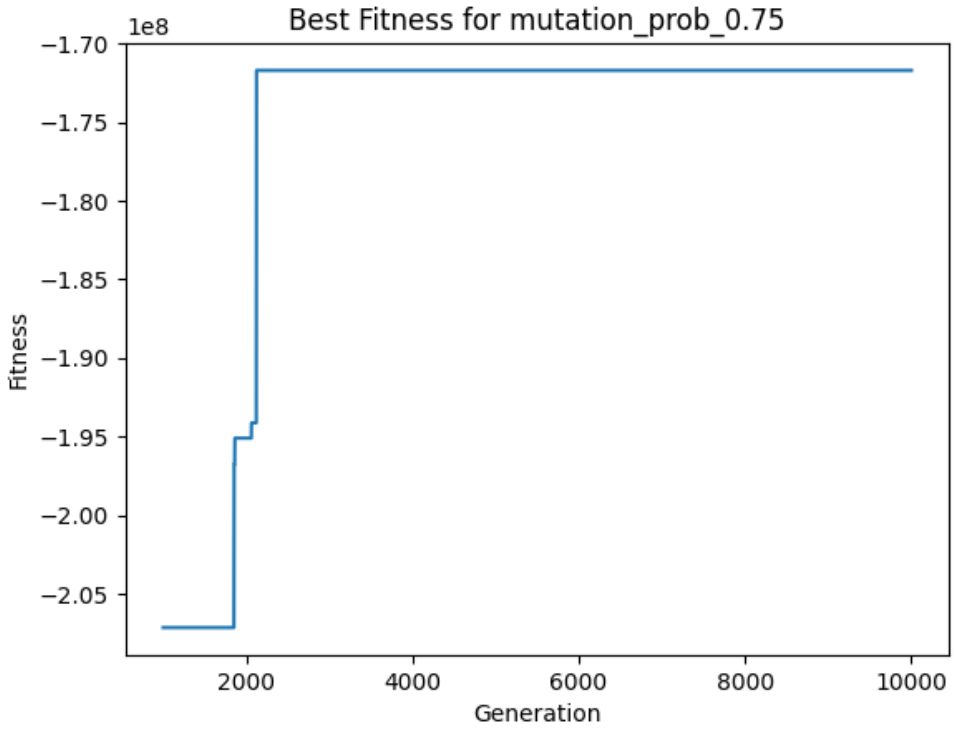


Figure 59. The Fitness Plot Generation (1000-10000) for mutation_prob=0.75

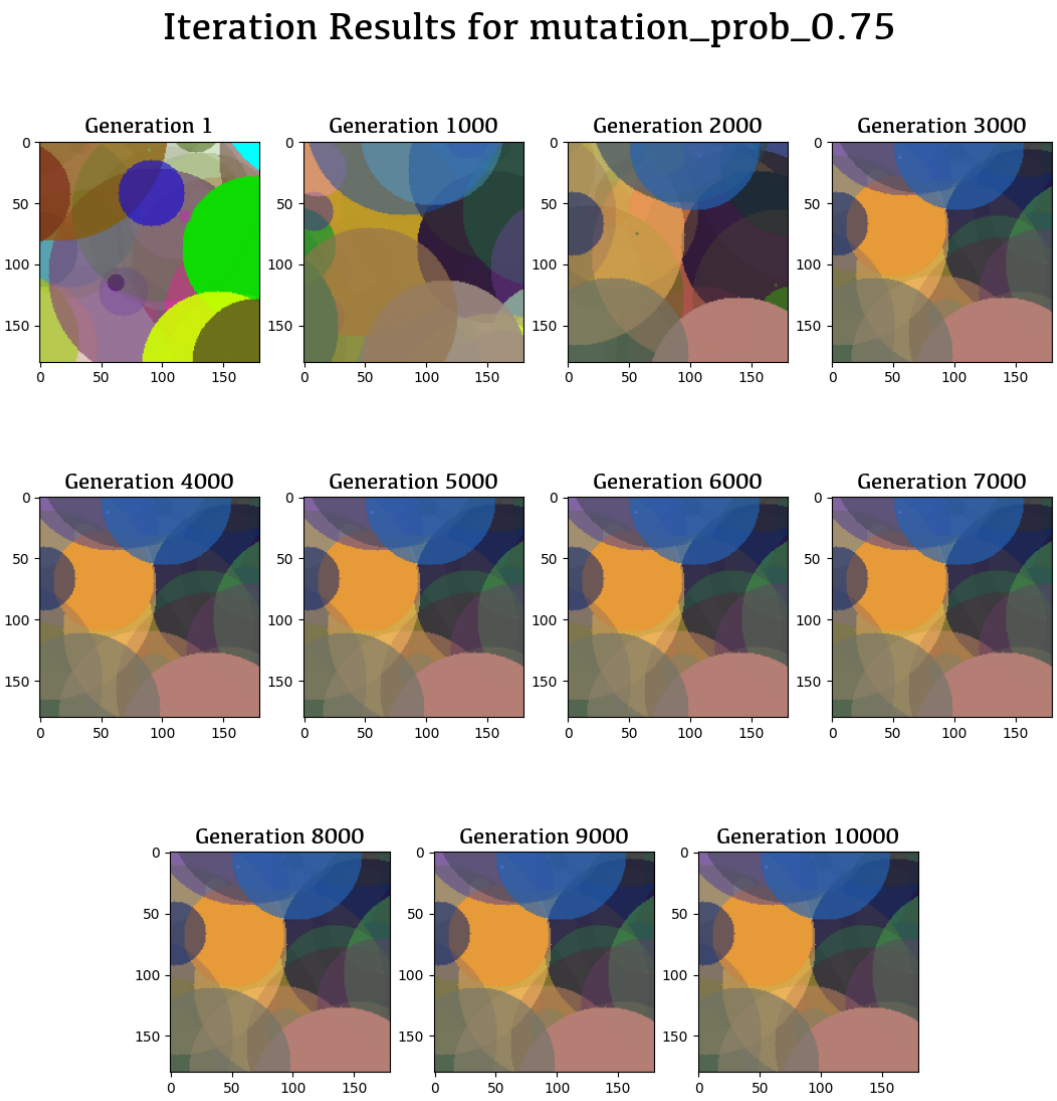


Figure 60. Corresponding Images of Best Individuals for mutation_prob=0.75

The best fitness value is *-171708021.*

Upon examination of *Figures (1-3)* and *Figures (52-60)*, it is evident that the optimal fitness value is achieved when the **mutation_prob=0.4** is utilized, different from default value. This is since a smaller mutation probability will result in lack of diversity in the results, while a larger mutation probability will require a longer time to converge to a minimum or cannot find the global minimum, ultimately leading to a worse fitness value most of the time.

2.8. Mutation Type

2.8.1. Unguided Mutation

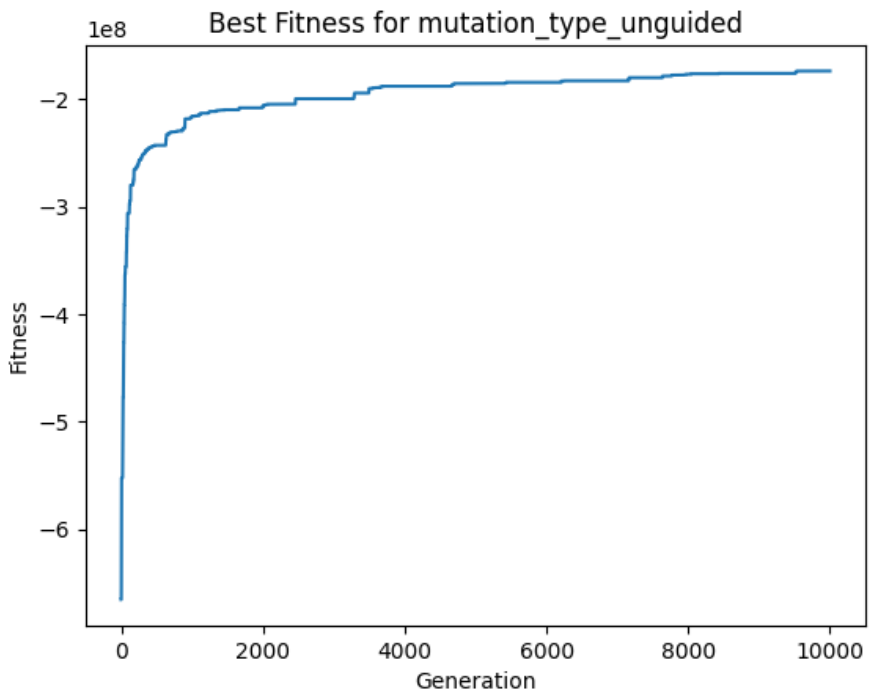


Figure 61. The Fitness Plot Generation (1-10000) for unguided mutation

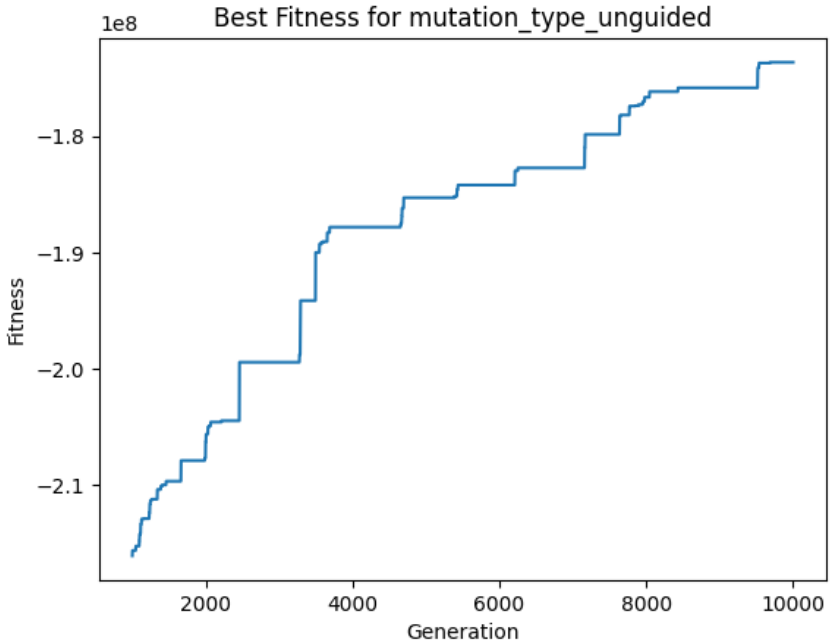


Figure 62. The Fitness Plot Generation (1000-10000) for unguided mutation

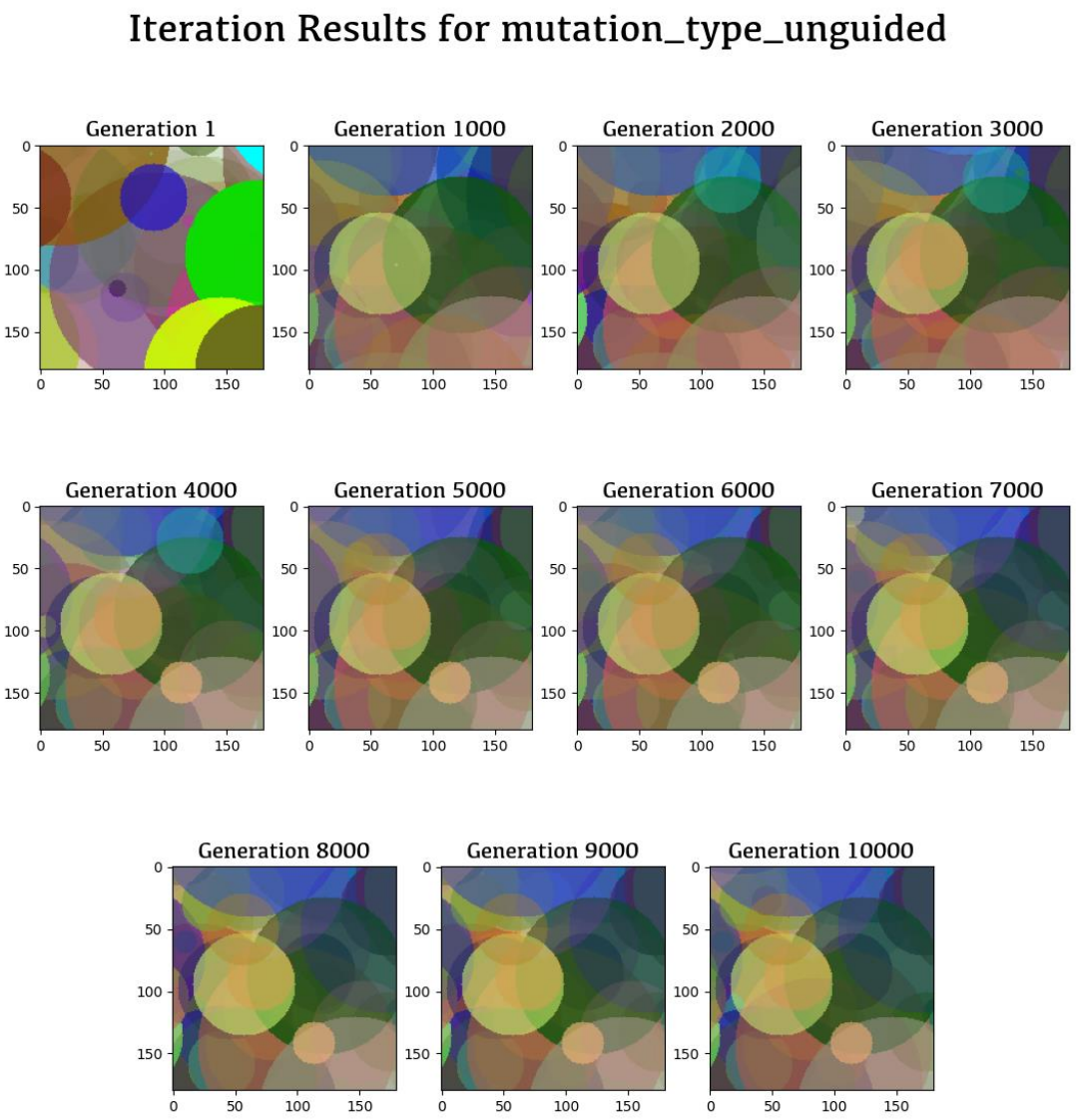


Figure 63. Corresponding Images of Best Individuals for unguided mutation

The best fitness value is *-173611503*.

Upon examination of *Figures (1-3)* and *Figures (61-63)*, it is evident that the optimal fitness value is achieved when the **guided mutation** is utilized. This happened since unguided mutation provides fully random results which causes to find a global minimum.

3. Discussion

3.1. Dynamic Mutation Probability

In our previous homework, I explored the idea of implementing a dynamic learning rate to enhance the performance of our algorithm. To achieve this, I carefully analyzed the mutation results by examining their corresponding graphs. Based on my observations, I proposed a dynamic method to optimize the mutation rate, which I believed would lead to improved results.

After conducting several tests and scrutinizing the behavior of the fitness function, I decided to adopt the following strategy for adjusting the mutation probability based on the generation number:

Table 1. Mutation Probability Values for Generations

Generation Range	Mutation Probability
0 - 750	0.7
751 - 2000	0.6
2001 - 4000	0.5
4001 - 8000	0.4
8001 - 9000	0.3
9001 - 10000	0.2

This approach entails gradually decreasing the mutation probability as the generation number increases. By doing so, we can strike a balance between exploration and exploitation, ultimately leading to a more effective optimization process.

The best fitness value for my suggestion is: **-127631536**. On the other hand, original best fitness value was: **-132389734.** It is clear that not only we converged faster, but also a better result. Details can be examined in Figure *64-66*.

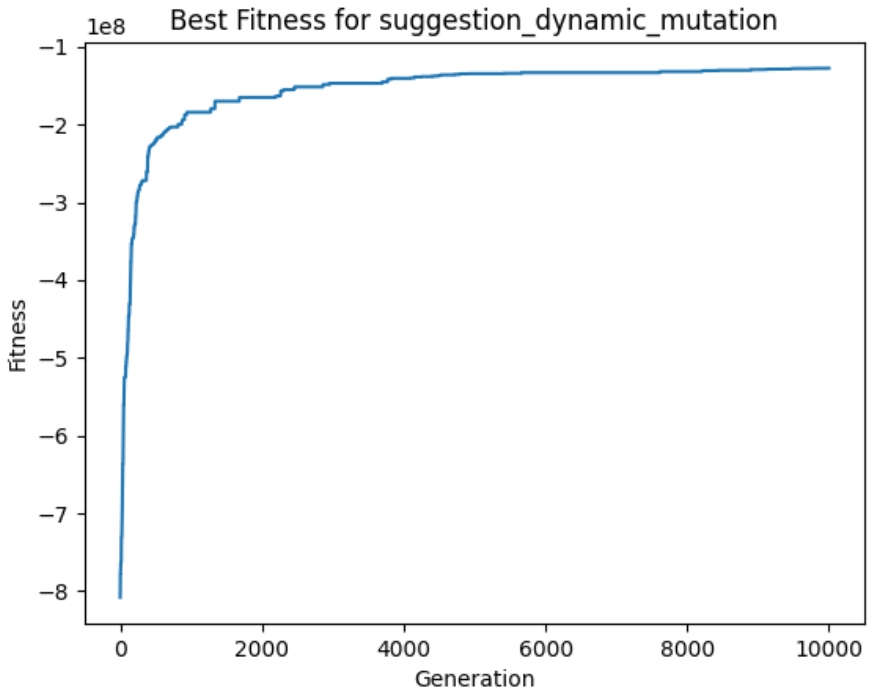


Figure 64. The Fitness Plot Generation (1-10000) for dynamic mutation probability

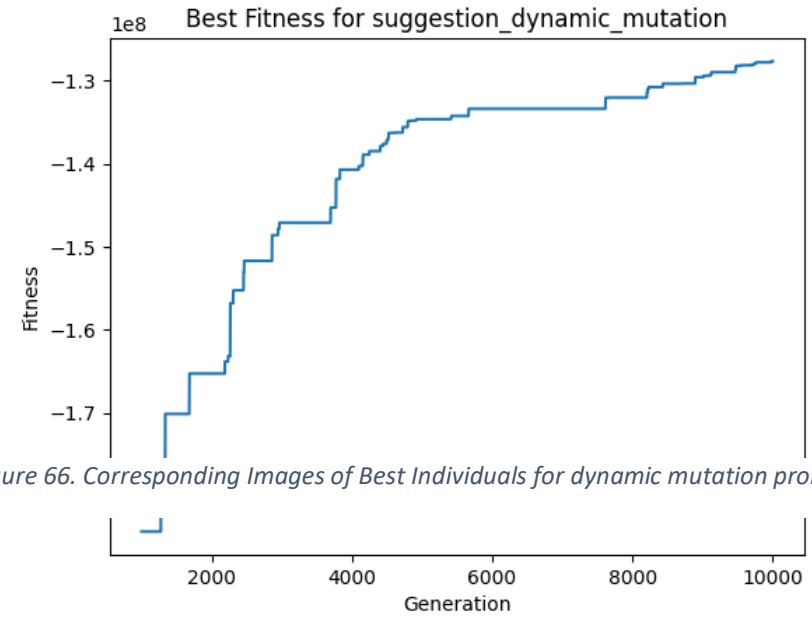
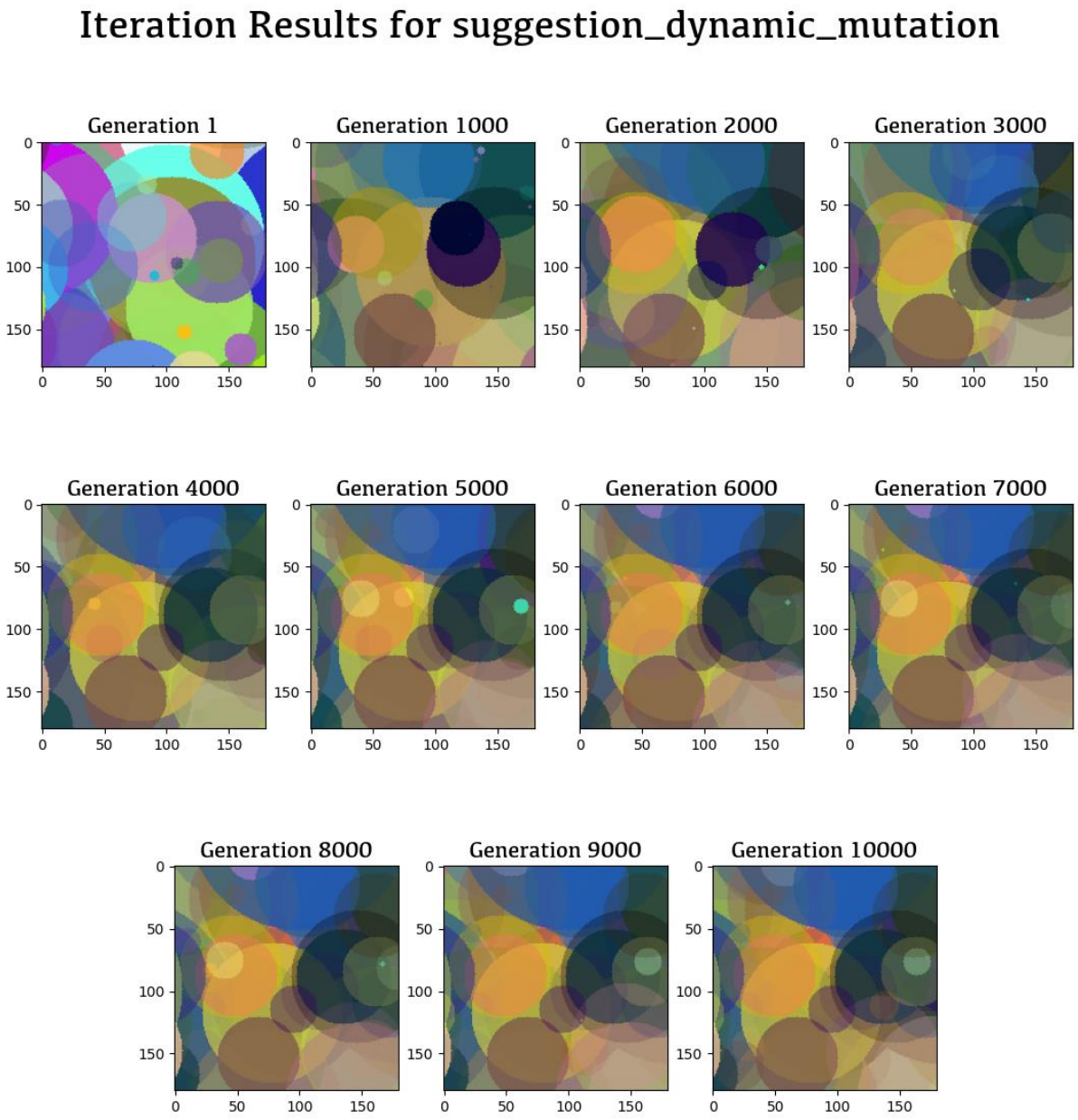


Figure 66. Corresponding Images of Best Individuals for dynamic mutation probability

Figure 65. The Fitness Plot Generation (1000-10000) for dynamic mutation probability



3.2. Dynamic Mutation Type

For my second suggestion, I employed a similar approach by analyzing the mutation

type results through their respective graphs. Based on my observations, I decided to experiment with two additional mutation types: **one less guided** and **one more guided**. To implement these new mutation types, I modified the base code for the guided mutation by adjusting the parameters' offset for each type. For the less guided mutation, I **multiplied the offset by 2**, while for the more guided mutation, I **divided the offset by 2.**

I then applied these mutation types at specific generation ranges, as outlined in the table below:

Table 2. Mutation Types for Generations

Generation Range	Mutation Type
0 - 750	Unguided
751 - 3000	Less Guided
3001 - 6000	Guided
6001 - 10000	More Guided

This strategy involves changing the mutation type at different stages of the optimization process, with a maximum generation number of 10,000. By incorporating these additional mutation types, we aim to further enhance the algorithm's performance and achieve better results.

The best fitness value for my suggestion is: **- 132400808**. On the other hand, original best fitness value was: **-132389734.** It is clear that the best fitness value is almost same as original. However, we observed that the convergence to best value is faster than original, details can be examined in *Figures 67-69*.

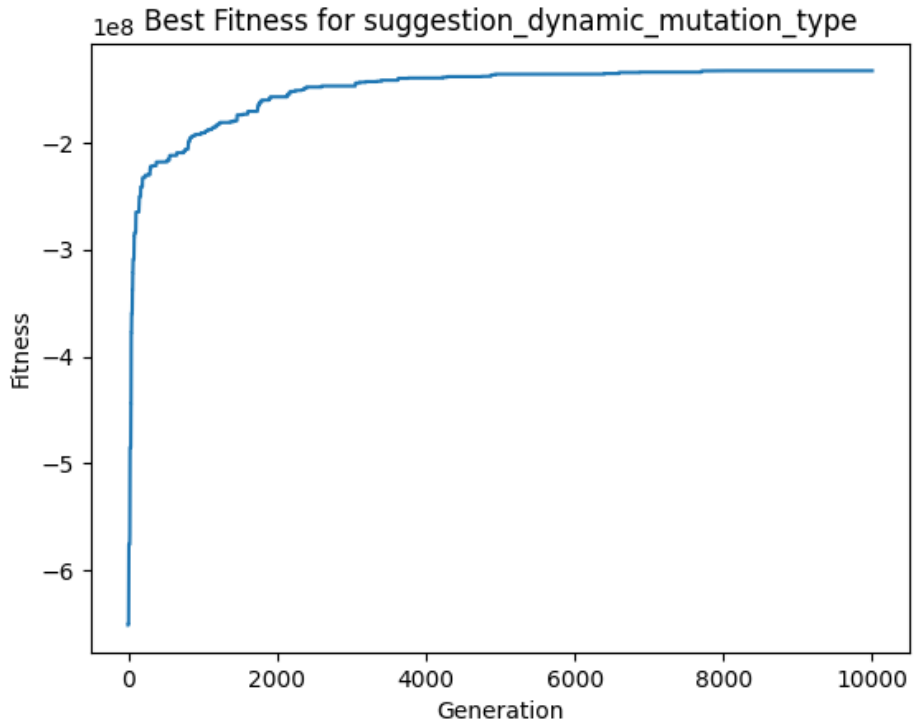


Figure 67. The Fitness Plot Generation (1-10000) for dynamic mutation type

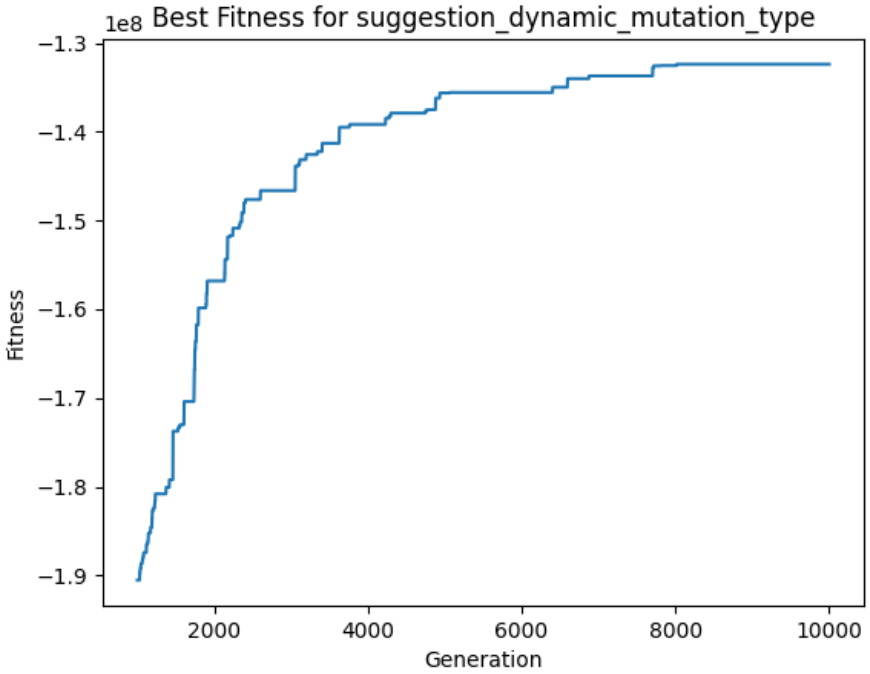


Figure 68. The Fitness Plot Generation (1000-10000) for dynamic mutation type

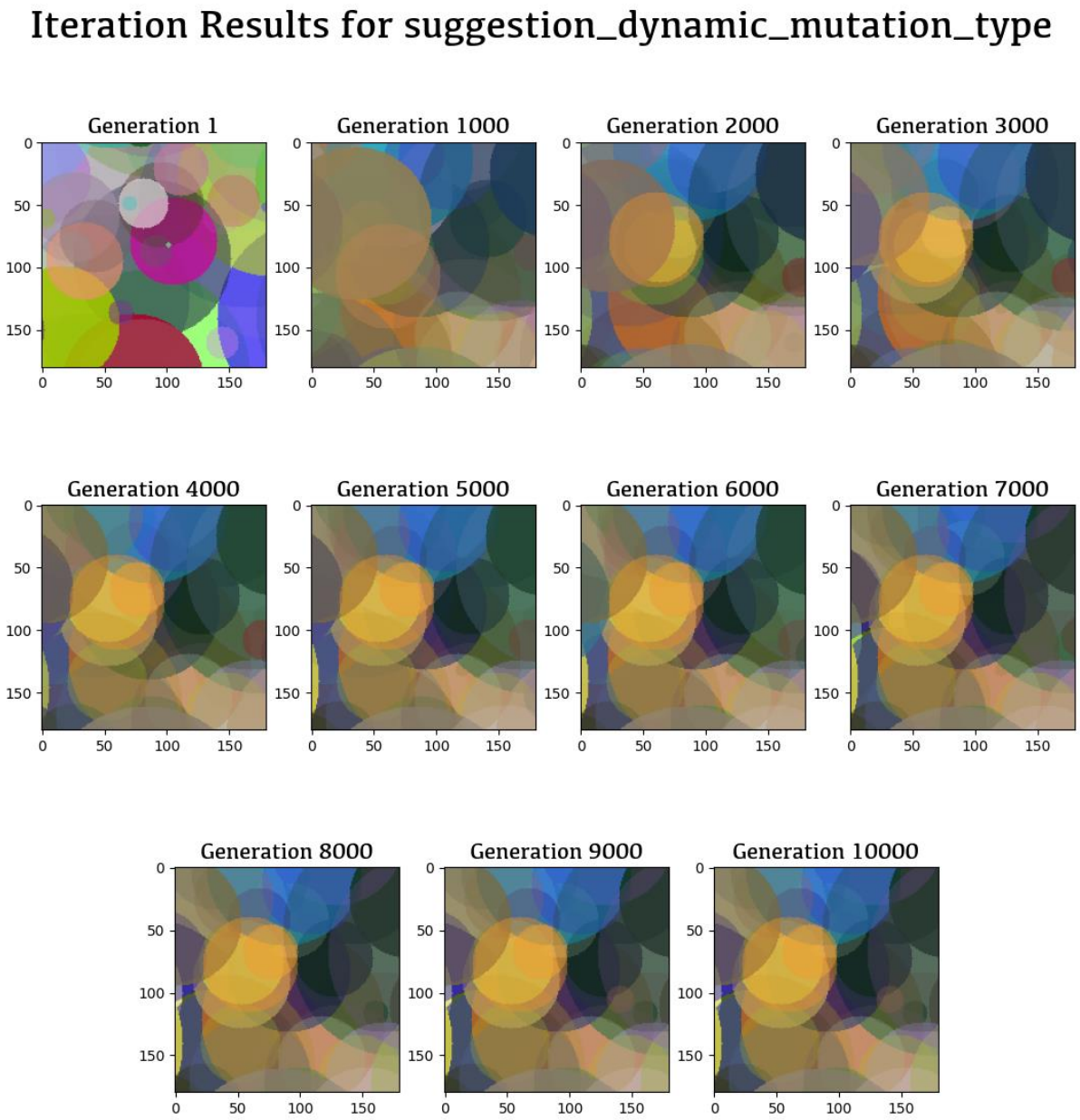


Figure 69. Corresponding Images of Best Individuals for dynamic mutation type

3.3. Dynamic Parent Elite Selection

For my third suggestion, I decided to simultaneously modify the fraction of parents and elites throughout the optimization process. I carefully observed the characteristics of fitness values for both operations and conducted several tests to determine the optimal balance. I discovered that starting with a larger number of parents allowed for more mutations and greater diversity. As the generations progressed, I gradually decreased the number of parents while increasing the number of elites.

The following table outlines the adjustments made to the fraction of parents and elites at specific generation ranges, with a maximum generation number of 10,000:

Table 3. Fraction Parents & Elites for Generations

Generation Range	Fraction of Parents	Fraction of Elites
0 - 750	0.75	0.03
751 - 2000	0.7	0.05
2001 – 4000	0.65	0.08
4001 – 8000	0.6	0.1
8001 – 9000	0.5	0.2
9001 – 10000	0.4	0.25

By adjusting the fractions of parents and elites throughout the optimization process, we aim to strike a balance between exploration and exploitation, ultimately leading to improved algorithm performance and better results.

The best fitness value for my suggestion is: **-130055801**. On the other hand, original best fitness value was: **-132389734**. It is clear that not only we converged faster, but also a better result. Details can be examined in Figure *70-72*.

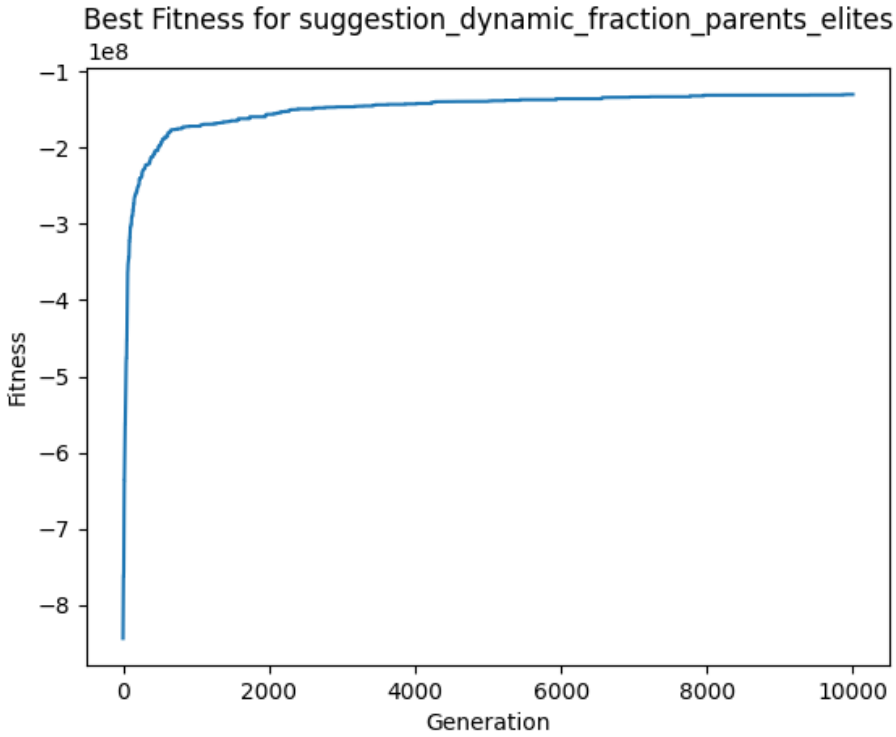


Figure 70. The Fitness Plot Generation (1-10000) for dynamic fraction for parents and elites

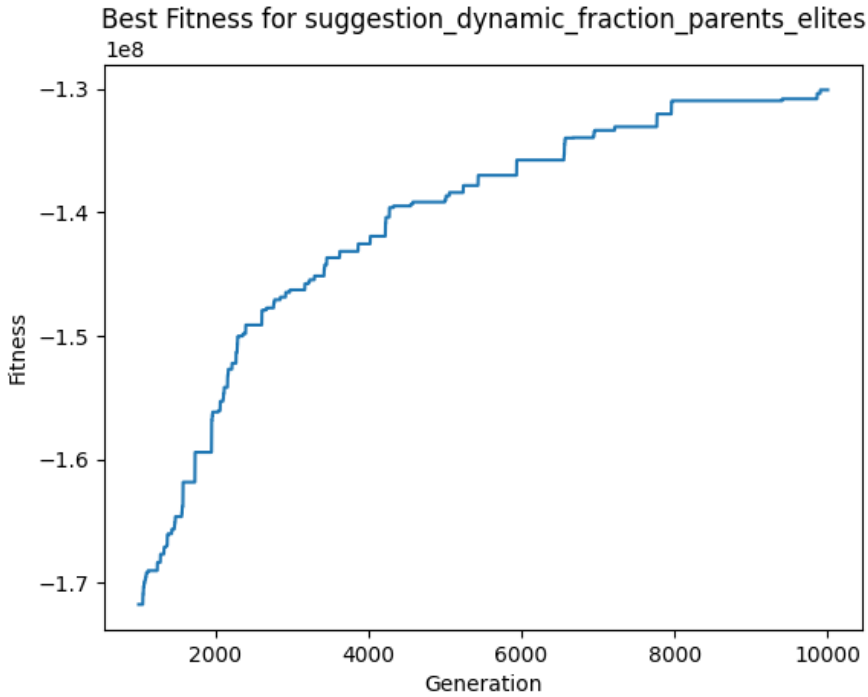


Figure 71. The Fitness Plot Generation (1000-10000) for dynamic fraction for parents and elites

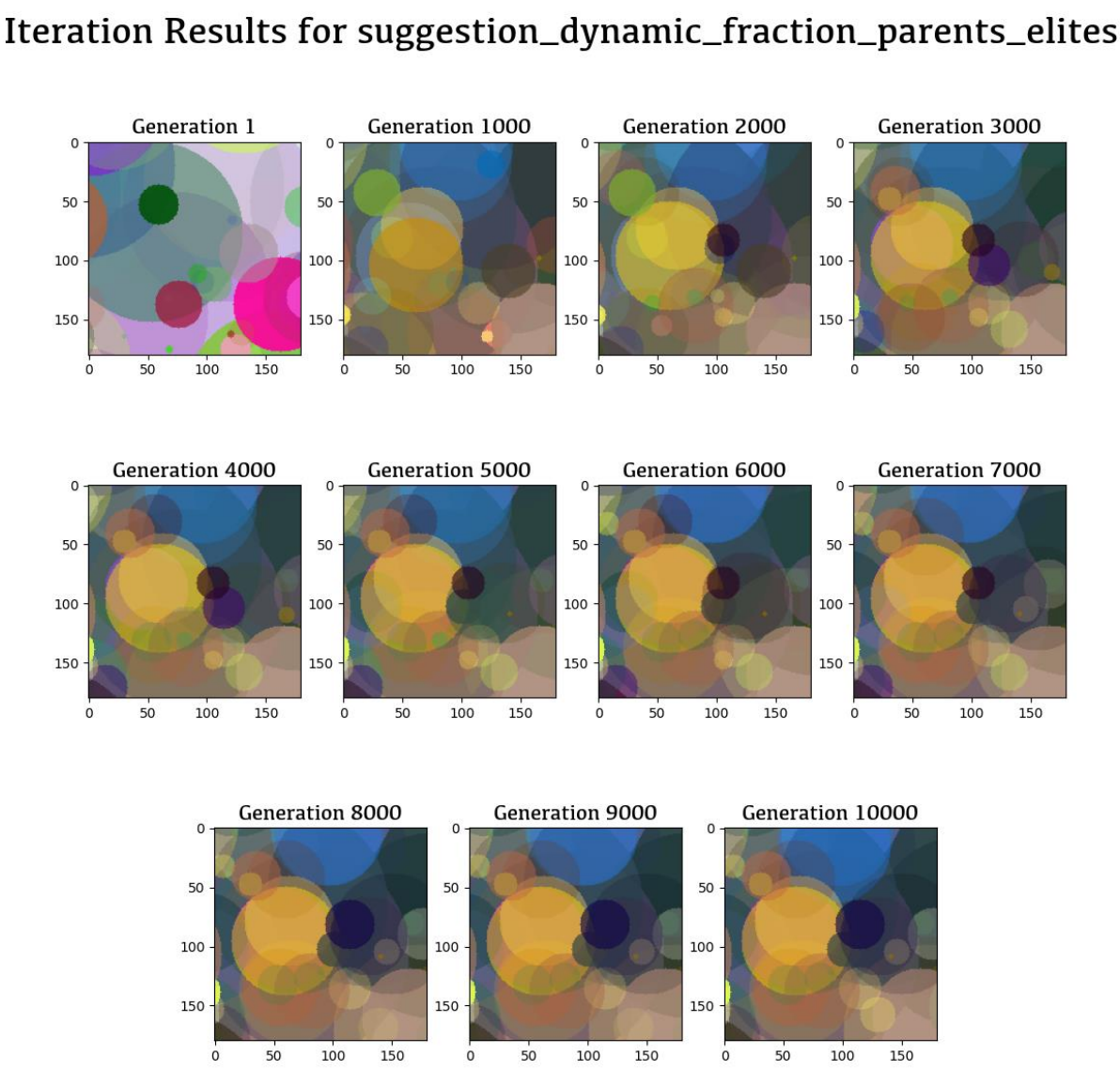


Figure 72. Corresponding Images of Best Individuals for dynamic fraction for parents and elites

Appendix I

hw2.py – The Main Code

```
# Created by Deniz Karakay (2443307) on 30.04.2023
# Description: This file contains the main program for the second homework of EE449 course.
#              The program uses evolutionary algorithm to create an image that looks like the given image.
#              The given image is painting.png that it Café Terrace at Night by Vincent van Gogh.
#              The program also contains 3 suggestions for the evolutionary algorithm.
#              The program can be run with the following command:
#              python hw2.py

import copy
import datetime
import os
import random
import shutil
import time
import cv2
import numpy as np
import pickle

# Global variables for the image
IMG_PATH = "painting.png"
IMG_WIDTH = 0
IMG_HEIGHT = 0


class Gene:
    def __init__(self, id=-2):
        self.id = id
        self.radius = random.randint(1, max(IMG_WIDTH, IMG_HEIGHT) // 2)
        self.x, self.y = self.determine_center_coordinates()
        self.red = random.randint(0, 255)
        self.green = random.randint(0, 255)
        self.blue = random.randint(0, 255)
        self.alpha = random.random()

    def is_valid_circle(self, x, y):
```

```
    # Check if the circle is completely outside the image
    if x - self.radius >= IMG_WIDTH or x + self.radius < 0:
        return False
    if y - self.radius >= IMG_HEIGHT or y + self.radius < 0:
        return False

    # Check if the circle intersects with the image
    if x - self.radius < 0 or x + self.radius >= IMG_WIDTH:
        return True
    if y - self.radius < 0 or y + self.radius >= IMG_HEIGHT:
        return True

    # Check if the circle is completely inside the image
    if x - self.radius >= 0 and x + self.radius < IMG_WIDTH:
        return True
    if y - self.radius >= 0 and y + self.radius < IMG_HEIGHT:
        return True

    return False

# Determine the center coordinates of the gene
def determine_center_coordinates(self, guided=False):
    while True:
        if guided:
            x = self.x + random.randint(-IMG_WIDTH // 4, IMG_WIDTH // 4)
            y = self.y + random.randint(-IMG_HEIGHT // 4, IMG_HEIGHT // 4)
        else:
            x = random.randint(IMG_WIDTH * -1.6, IMG_WIDTH * 1.6)
            y = random.randint(IMG_HEIGHT * -1.6, IMG_HEIGHT * 1.6)

        if self.is_valid_circle(x, y):
            return x, y

# Mutate the gene with a guided mutation
def guided_mutation(self):
    self.radius = np.clip(
        self.radius + random.randint(-10, 10), 1, min(IMG_WIDTH, IMG_HEIGHT) // 2
```

```
        )

        # Determine the center coordinates of the gene
        self.x, self.y = self.determine_center_coordinates(guided=True)

        # Mutate the color and alpha of the gene
        self.red = int(np.clip(self.red + random.randint(-64, 64), 0, 255))
        self.green = int(np.clip(self.green + random.randint(-64, 64), 0, 255))
        self.blue = int(np.clip(self.blue + random.randint(-64, 64), 0, 255))
        self.alpha = np.clip(self.alpha + random.uniform(-0.25, 0.25), 0, 1)

    # Mutate the gene with more guided mutation (for suggestions)
    def more_guided_mutation(self):
        self.radius = np.clip(
            self.radius + random.randint(-20, 20), 1, min(IMG_WIDTH, IMG_HEIGHT) // 2
        )

        # Determine the center coordinates of the gene
        self.x, self.y = self.determine_center_coordinates(guided=True)

        # Mutate the color and alpha of the gene
        self.red = int(np.clip(self.red + random.randint(-128, 128), 0, 255))
        self.green = int(np.clip(self.green + random.randint(-128, 128), 0, 255))
        self.blue = int(np.clip(self.blue + random.randint(-128, 128), 0, 255))
        self.alpha = np.clip(self.alpha + random.uniform(-0.5, 0.5), 0, 1)

    # Mutate the gene with less guided mutation (for suggestions)
    def less_guided_mutation(self):
        self.radius = np.clip(
            self.radius + random.randint(-5, 5), 1, min(IMG_WIDTH, IMG_HEIGHT) // 2
        )

        # Determine the center coordinates of the gene
        self.x, self.y = self.determine_center_coordinates(guided=True)

        # Mutate the color and alpha of the gene
        self.red = int(np.clip(self.red + random.randint(-32, 32), 0, 255))
```

```
        self.green = int(np.clip(self.green + random.randint(-32, 32), 0, 255))
        self.blue = int(np.clip(self.blue + random.randint(-32, 32), 0, 255))
        self.alpha = np.clip(self.alpha + random.uniform(-0.12, 0.12), 0, 1)

    # Print every property of the gene in one line
    def print(self):
        print(
            "Gene ID:",
            self.id,
            "R:",
            self.radius,
            "X:",
            self.x,
            "Y:",
            self.y,
            "RGBA:",
            self.red,
            self.green,
            self.blue,
            self.alpha.__round__(2),
        )


class Individual:
    def __init__(self, num_genes, id=-2, fitness=0, chromosome=[]):
        self.id = id
        self.chromosome = chromosome
        self.fitness = fitness

        if len(chromosome) == 0:
            # Create a chromosome with num_genes
            for i in range(num_genes):
                self.chromosome.append(Gene(i))

            # Sort chromosomes by radius by descending order
            self.chromosome.sort(key=lambda x: x.radius, reverse=True)
```

```python
# Draw the individual to the screen
def draw(self):
    # Create a white image
    img = np.full((IMG_WIDTH, IMG_HEIGHT, 3), 255, np.uint8)

    for gene in self.chromosome:
        # Create an overlay image
        overlay = img.copy()

        color = (gene.red, gene.green, gene.blue)

        # Draw the gene to the image
        cv2.circle(overlay, (gene.x, gene.y), gene.radius, color, -1)

        img = cv2.addWeighted(overlay, gene.alpha, img, 1 - gene.alpha, 0, img)

    return img

# Evaluate the fitness of the individual
def eval(self):
    # Convert source and individual to numpy arrays
    source = cv2.imread(IMG_PATH)
    source_np = np.array(source, dtype=np.int64)

    individual = self.draw()
    individual_np = np.array(individual, dtype=np.int64)

    # Calculate the difference between source and individual
    diff = np.subtract(source_np, individual_np)

    # Square the difference
    squared = np.square(diff)

    # Sum the squared difference
    self.fitness = -np.sum(squared)

    return self.fitness
```

```
# Mutate the individual with mutation_type
def mutate(self, mutation_type, mutation_prob):
    mutations = []
    while random.random() < mutation_prob:
        random_gene_id = random.randint(0, len(self.chromosome) - 1)

        if len(mutations) >= len(self.chromosome):
            return

        while random_gene_id in mutations:
            random_gene_id = random.randint(0, len(self.chromosome) - 1)

        mutations.append(random_gene_id)
        if mutation_type == "unguided":
            self.chromosome[random.randint(0, len(self.chromosome) - 1)] = Gene(
                id=random_gene_id
            )
        elif mutation_type == "guided":
            self.chromosome[random_gene_id].guided_mutation()
        # For suggestions (dynamic mutation type)
        elif mutation_type == "more_guided":
            self.chromosome[random_gene_id].more_guided_mutation()
        # For suggestions (dynamic mutation type)
        elif mutation_type == "less_guided":
            self.chromosome[random_gene_id].less_guided_mutation()

# Print every property of the individual in one line
def print(self):
    print(
        "Individual ID:",
        self.id,
        "Fitness:",
        self.fitness,
        "Genes:",
        len(self.chromosome),
    )
```

```
        for gene in self.chromosome:
            gene.print()


class Population:
    def __init__(self, num_inds, num_genes, population=[]):
        self.population = population
        self.best_population = []

        if len(population) == 0:
            # Create a population with num_inds individuals
            for i in range(num_inds):
                self.population.append(Individual(id=i, num_genes=num_genes))

    # Evaluate every individual in the population
    def evaluate(self):
        for individual in self.population:
            individual.eval()

    # Sort the population by fitness in descending order
    def sort_population(self):
        t = sorted(self.population, key=lambda x: x.fitness, reverse=True)
        return t

    # Sort the population by fitness in descending order with respect to the given population
    def sort_inds(self, pop):
        return sorted(pop, key=lambda x: x.fitness, reverse=True)

    # Select the best num_elites individuals in the population
    # Return the best num_elites individuals and the rest of the population
    def selection(self, fraction_elites, fraction_parents, tm_size):
        self.population = self.sort_population()

        # Select the best num_elites individuals in the population
        num_elites = int(len(self.population) * fraction_elites)

        # Select the best num_parents individuals in the population
```

```
    num_parents = int(len(self.population) * fraction_parents)

    # If the number of parents is odd, make it even
    if num_parents % 2 == 1:
        num_parents += 1

    # Divide the population into elites and non_elites
    elites = self.population[:num_elites]
    non_elites = self.population[num_elites:]

    # Select num_parents individuals from the non_elites using tournament selection
    parents = []
    for i in range(num_parents):
        tournament = random.sample(non_elites, min(tm_size, len(non_elites)))
        best = self.sort_inds(tournament)[0]
        parents.append(best)
        non_elites.remove(best)

    return elites, non_elites, parents

# Select two parents from the population using tournament selection
def crossover(self, parents, num_genes):
    children = []
    for i in range(0, len(parents), 2):
        parent1 = parents[i]
        parent2 = parents[i + 1]

        chromosome_child1 = []
        chromosome_child2 = []

        for j in range(len(parent1.chromosome)):
            # 50% chance to get the gene from parent1 or parent2
            if random.random() >= 0.5:
                chromosome_child1.append(copy.deepcopy(parent1.chromosome[j]))
                chromosome_child2.append(copy.deepcopy(parent2.chromosome[j]))
            # 50% chance to get the gene from parent2 or parent1
            else:
```

```
                chromosome_child1.append(copy.deepcopy(parent2.chromosome[j]))
                chromosome_child2.append(copy.deepcopy(parent1.chromosome[j]))

            #  Create two children with the same length as parents
            child1 = Individual(num_genes=num_genes, chromosome=chromosome_child1)
            child2 = Individual(num_genes=num_genes, chromosome=chromosome_child2)
            child1.eval()
            child2.eval()

            pop = Population.sort_inds(self, [child1, child2, parent1, parent2])
            children.append(pop[0])
            children.append(pop[1])

        return children

    # Print every property of the population in one line
    def print(self):
        for individual in self.population:
            individual.print()


# Save the population to a file using pickle
def save_population(population, path):
    with open(path + ".pkl", "wb") as f:
        pickle.dump(population, f, pickle.HIGHEST_PROTOCOL)


# Print the elapsed time in a readable format
def print_elapsed_time(elapsed_time):
    seconds = elapsed_time.seconds
    minutes = seconds // 60 % 60
    hours = seconds // 3600 % 3600

    if elapsed_time < datetime.timedelta(minutes=1):
        return f"{seconds} secs"
    elif elapsed_time < datetime.timedelta(hours=1):
        return f"{minutes} mins {seconds % 60} secs"
```

```
    else:
        return f"{hours} hours {minutes % 60} mins {seconds % 60} secs"


# Save the population and the best individual
def save_all(pop, name, generation, path, best_population, image_only=True):
    print(best_population.fitness)

    current_name = f"{name}_{generation}"
    file_path = f"{path}{current_name}"
    if not image_only:
        # Save the population
        save_population(pop, file_path)

    # Save the best individual
    cv2.imwrite(f"{path}{current_name}.png", best_population.draw())


# Run the evolutionary algorithm for the given parameters
def evolutionary_algorithm(
    name,
    num_generations=10000,
    num_inds=20,
    num_genes=50,
    tm_size=5,
    mutation_type="guided",
    fraction_elites=0.2,
    fraction_parents=0.6,
    mutation_prob=0.2,
):
    # Create a folder for the results
    path = f"results/{name}/"

    # Delete the folder if it exists
    if os.path.exists(path):
        shutil.rmtree(path)
```

```
# Create the folder if it does not exist
if not os.path.exists(path):
    os.mkdir(path)

# Create a population with num_inds individuals
pop = Population(num_inds, num_genes)

for generation in range(num_generations):
    # Evaluate the population
    pop.evaluate()

    # Select the best num_elites individuals in the population and the rest of the population
    elites, non_elites, parents = pop.selection(
        fraction_elites, fraction_parents, tm_size
    )

    # Create children from the parents
    children = pop.crossover(parents, num_genes)

    # Mutate the children and non_elites
    for individual in non_elites + children:
        individual.mutate(mutation_type, mutation_prob)

    # Add the elites, children and non_elites to the population
    pop.population = elites + children + non_elites

    sorted_population = pop.sort_population()
    pop.best_population.append(sorted_population[0])

    if generation % 1000 == 0 or generation == 0:
        save_all(pop, name, generation, path, sorted_population[0])

    if generation % 100 == 0:
        elapsed_time = datetime.datetime.now() - start_time
        print(
            "Generation:",
            generation,
```

```
            "Time:",
            print_elapsed_time(elapsed_time),
            "Fitness:",
            sorted_population[0].fitness,
        )

    pop.evaluate()
    sorted_population = pop.sort_population()
    pop.best_population.append(sorted_population[0])
    save_all(pop, name, 10000, path, sorted_population[0], image_only=False)


# Run the evolutionary algorithm for suggestions
def evolutionary_algorithm_for_suggestions(
    name,
    num_generations=10000,
    num_inds=20,
    num_genes=50,
    tm_size=5,
    mutation_type="guided",
    fraction_elites=0.2,
    fraction_parents=0.6,
    mutation_prob=0.2,
    suggestion_type="dynamic_mutation",
):
    path = f"results/{name}/"

    if os.path.exists(path):
        shutil.rmtree(path)

    if not os.path.exists(path):
        os.mkdir(path)

    pop = Population(num_inds, num_genes)

    for generation in range(num_generations):
        # Dynamic mutation probability
```

```
    if suggestion_type == "dynamic_mutation":
        if generation == 1:
            mutation_prob = 0.7
        elif generation == 750:
            mutation_prob = 0.6
        elif generation == 2000:
            mutation_prob = 0.5
        elif generation == 4000:
            mutation_prob = 0.4
        elif generation == 8000:
            mutation_prob = 0.3
        elif generation == 9000:
            mutation_prob = 0.2

    # Dynamic fraction of parents and elites
    elif suggestion_type == "dynamic_fraction_parents_elites":
        if generation == 1:
            fraction_parents = 0.75
            fraction_elites = 0.03
        elif generation == 750:
            fraction_parents = 0.7
            fraction_elites = 0.05
        elif generation == 2000:
            fraction_parents = 0.65
            fraction_elites = 0.08
        elif generation == 4000:
            fraction_parents = 0.6
            fraction_elites = 0.1
        elif generation == 8000:
            fraction_parents = 0.5
            fraction_elites = 0.2
        elif generation == 9000:
            fraction_parents = 0.4
            fraction_elites = 0.25

    # Dynamic mutation type
    elif suggestion_type == "dynamic_mutation_type":
```

```
        if generation == 1:
            mutation_type = "unguided"
        elif generation == 750:
            mutation_type = "less_guided"
        elif generation == 3000:
            mutation_type = "guided"
        elif generation == 6000:
            mutation_type = "more_guided"

    pop.evaluate()

    # Select the best num_elites individuals in the population and the rest of the population
    elites, non_elites, parents = pop.selection(
        fraction_elites, fraction_parents, tm_size
    )

    # Create children from the parents
    children = pop.crossover(parents, num_genes)

    # Mutate the children and non_elites
    for individual in non_elites + children:
        individual.mutate(mutation_type, mutation_prob)

    # Add the elites, children and non_elites to the population
    pop.population = elites + children + non_elites

    # Sort the population by fitness
    sorted_population = pop.sort_population()
    pop.best_population.append(sorted_population[0])

    # Save the best individual and the population in every 1000 generations
    if generation % 1000 == 0 or generation == 0:
        save_all(pop, name, generation, path, sorted_population[0])

    # Print the generation, elapsed time and fitness in every 100 generations
    if generation % 100 == 0:
        elapsed_time = datetime.datetime.now() - start_time
```

```
            print(
                "Generation:",
                generation,
                "Time:",
                print_elapsed_time(elapsed_time),
                "Fitness:",
                sorted_population[0].fitness,
            )

    # Evaluate the population one last time
    pop.evaluate()
    sorted_population = pop.sort_population()
    pop.best_population.append(sorted_population[0])
    save_all(pop, name, 10000, path, sorted_population[0], image_only=False)


if __name__ == "__main__":
    start_time = datetime.datetime.now()

    img = cv2.imread(IMG_PATH)
    IMG_WIDTH = img.shape[0]
    IMG_HEIGHT = img.shape[1]

    IMG_RADIUS = (IMG_WIDTH + IMG_HEIGHT) / 2

    # Default parameters
    # Num_inds = 20,
    # Num_genes = 50,
    # Tm_size = 5,
    # Fraction_elites = 0.2,
    # Fraction_parents = 0.6,
    # Mutation_prob = 0.2,
    # Mutation_type = "guided"
    evolutionary_algorithm(name="default")

    # # NUM_INDS = 5, 10, 40 and 60
    evolutionary_algorithm(name="num_inds_5", num_inds=5)
```

```
evolutionary_algorithm(name="num_inds_10", num_inds=10)
evolutionary_algorithm(name="num_inds_40", num_inds=40)
evolutionary_algorithm(name="num_inds_60", num_inds=60)

## NUM_GENES = 15, 30, 80 and 120
evolutionary_algorithm(name="num_genes_15", num_genes=15)
evolutionary_algorithm(name="num_genes_30", num_genes=30)
evolutionary_algorithm(name="num_genes_80", num_genes=80)
evolutionary_algorithm(name="num_genes_120", num_genes=120)

# TM_SIZE = 2, 8 and 16
evolutionary_algorithm(name="tm_size_2", tm_size=2)
evolutionary_algorithm(name="tm_size_8", tm_size=8)
evolutionary_algorithm(name="tm_size_16", tm_size=16)

# FRACTION_ELITES = 0.04 and 0.35
evolutionary_algorithm(name="fraction_elites_0.04", fraction_elites=0.04)
evolutionary_algorithm(name="fraction_elites_0.35", fraction_elites=0.35)

# FRACTION_PARENTS = 0.15, 0.3 and 0.75
evolutionary_algorithm(name="fraction_parents_0.15", fraction_parents=0.15)
evolutionary_algorithm(name="fraction_parents_0.3", fraction_parents=0.3)
evolutionary_algorithm(name="fraction_parents_0.75", fraction_parents=0.75)

# MUTATION_PROB = 0.1, 0.4 and 0.75
evolutionary_algorithm(name="mutation_prob_0.1", mutation_prob=0.1)
evolutionary_algorithm(name="mutation_prob_0.4", mutation_prob=0.4)
evolutionary_algorithm(name="mutation_prob_0.75", mutation_prob=0.75)

# MUTATION_TYPE = "unguided"
evolutionary_algorithm(name="mutation_type_unguided", mutation_type="unguided")

# SUGGESTIONS

# Suggestion #1 - Dynamic mutation probability
evolutionary_algorithm_for_suggestions(
    name="suggestion_dynamic_mutation",
```

```
    suggestion_type="dynamic_mutation",
)

# Suggestion #2 - Dynamic mutation type
evolutionary_algorithm_for_suggestions(
    name="suggestion_dynamic_mutation_type",
    suggestion_type="dynamic_mutation_type",
)

# Suggestion #3 - Dynamic Parent and Elite Selection
evolutionary_algorithm_for_suggestions(
    name="suggestion_dynamic_fraction_parents_elites",
    suggestion_type="dynamic_fraction_parents_elites",
)

# Print the elapsed time
elapsed_time = datetime.datetime.now() - start_time
print("All time:", print_elapsed_time(elapsed_time))
```

Appendix II

visualizations.py – For visualization of the plots, combining images etc.

```
# Created by Deniz Karakay (2443307) on 02.05.2023
# Description: Visualize the results of the genetic algorithm
#  - Plot the best fitness from 1 to 10000
#  - Plot the best fitness from 1000 to 10000
#  - Combine 11 images into a single image


import pickle
import matplotlib.pyplot as plt
from matplotlib.transforms import Bbox
import numpy as np
```

```python
import os
from hw2 import Population, Individual, Gene
from PIL import Image, ImageDraw, ImageFont


def ensure_non_decreasing(values):
    # Ensure that a list of values is non-decreasing.
    for i in range(1, len(values)):
        values[i] = max(values[i], values[i - 1])
    return values


def combine_images():
    # Combine 11 images into a single image
    image_paths = [f"results/{folder}/{folder}_{i*1000}.png" for i in range(11)]

    # Load the images
    images = [Image.open(path) for path in image_paths]

    # Titles for each image
    titles = [f"Generation {i*1000}" for i in range(11)]
    titles[0] = "Generation 1"

    # Create a grid of subplots
    num_cols = 4
    num_rows = (len(images) - 1) // num_cols + 1
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(13, 13))

    # Plot each image on a different subplot
    for i, ax in enumerate(axes.flat):
        if i < len(images):
            ax.imshow(images[i])
            ax.set_title(f"{titles[i]}", fontname="Kefa", fontsize=16)
        else:
            # Remove the last subplot if it is empty
            fig.delaxes(ax)
```

```
    # Center the last row of subplots if there are an odd number of images
    if len(images) % 2 == 1:
        last_row_axes = axes[-1]
        for ax in last_row_axes:
            bbox = ax.get_position()
            bbox = bbox.translated(0.1, 0.0)
            ax.set_position(bbox)

    # Add a large title to the final plot
    fig.suptitle(
        f"Iteration Results for {folder}",
        fontsize=28,
        fontweight="heavy",
        fontname="Kefa",
        y=0.95,
    )

    final_path = f"results/{folder}/combined_{folder}.png"
    plt.savefig(final_path, bbox_inches="tight", pad_inches=0.3)
    plt.clf()
    plt.close()


subfolder = "results"

# Get all folders names
folders = [
    f for f in os.listdir(subfolder) if os.path.isdir(os.path.join(subfolder, f))
]

folders.sort()
for folder in folders:
    all = []
    p = pickle.load(
        open(f"results/{folder}/{folder}_10000.pkl", "rb")
    )  # type: Population
```

```
fitnesses = [i.fitness for i in p.population]
x = np.sort(fitnesses)
y = sorted(p.population, key=lambda x: x.fitness, reverse=True)
print(folder, ":", len(p.best_population))

for i in range(len(p.best_population)):
    all.append(p.best_population[i].fitness)

all = ensure_non_decreasing(all)
# Print the best fitness
print(all[-1])

# Plot the best fitness from 1 to 10000
plt.plot(all)
plt.ylabel("Fitness")
plt.xlabel("Generation")
plt.title(f"Best Fitness for {folder}")
plt.savefig(f"results/{folder}/plot_{folder}_best_fitness.png")
plt.clf()
plt.close()

all = all[1000:]
# start from 1000 to the end of length of all
x = list(range(1000, len(all) + 1000))

# Plot the best fitness from 1000 to 10000
plt.plot(x, all)
plt.ylabel("Fitness")
plt.xlabel("Generation")
plt.title(f"Best Fitness for {folder}")
plt.savefig(f"results/{folder}/plot_{folder}_best_fitness_after_1000.png")
plt.clf()
plt.close()

# Combine images
combine_images()
```