Mutation type is unguided

Other parameters are set to their default values

Iteration 1000

Iteration 2000

Iteration 3000

Iteration 4000

Iteration 5000

Iteration 6000

Iteration 7000

Iteration 8000

Iteration 9000

Iteration 10000

Source Image

Best fitness after 10000 generations:
-166702060.0

FIGURE 63. RESULTS WHEN MUTATION TYPE IS UNGUIDED.

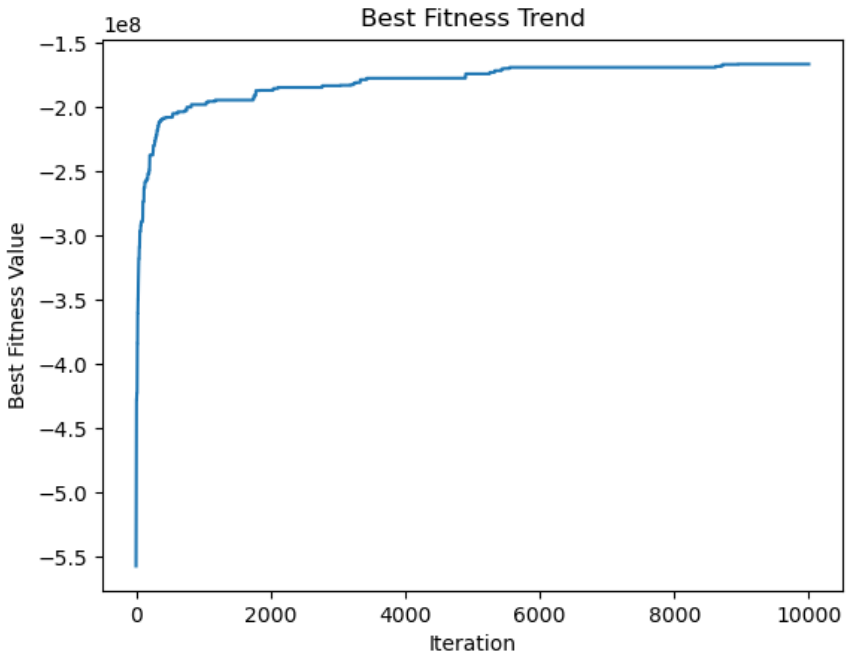


FIGURE 64. FITNESS PLOT FROM GENERATION 1 TO 10000, WHEN MUTATION TYPE IS UNGUIDED.

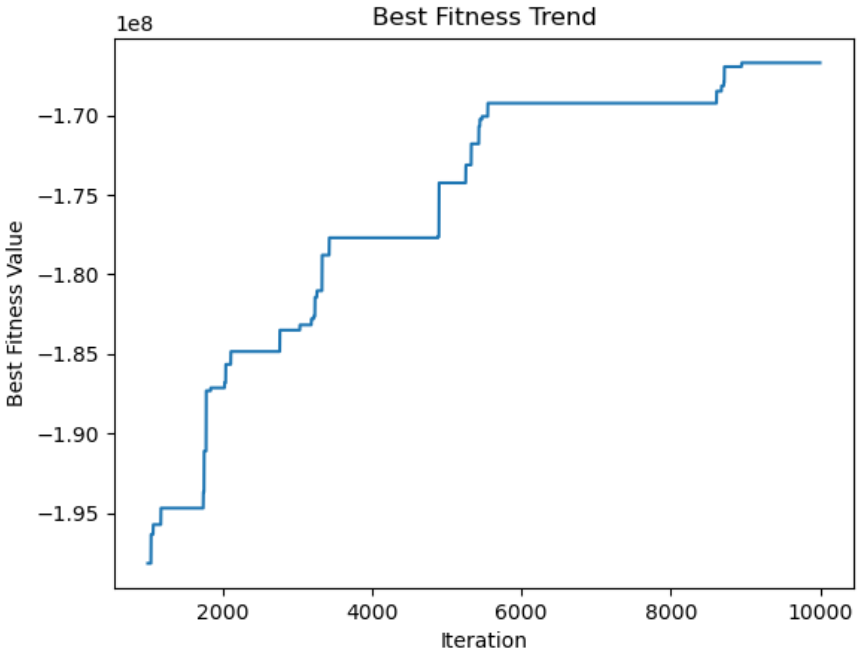


FIGURE 65. FITNESS PLOT FROM GENERATION 1000 TO 10000, WHEN MUTATION TYPE IS UNGUIDED.

Changing the number of genes parameter, the images in *Figures 18, 21, 24 and 27* are obtained. When the number of genes is 80, the best fitness value obtained after 10000 iterations is higher. Moreover, the image generated becomes more detailed as the number of genes increases, which is expected. Inspecting the Best Fitness plots in *Figures 19, 20, 22, 23, 25, 26, 28, and 29*, it can be seen that the trend is similar to the default case.

Evaluating the population for different values of tournament size parameter, the images in *Figures 30, 33, and 36* are obtained. When the tournament size is equal to 2, the resulting image resembles more to the source image than the image generated in the default case. It can be deduced that as the tournament size increases, the performance of the individuals decreases.

Changing the fraction of elites parameter, images in *Figures 39 and 42* are obtained. A fraction of 0.04 for the elite group, yields better results comparing to the default case where the fraction of elites is 0.2 and the case when fraction of elites is 0.35. More elites do not mean a better population.

Figures 45, 48, and 51, contain images generated when fraction of parents is set to 0.15, 0.30 and 0.75. When the fraction of parents parameter is 0.75, the best fitness value after 10000 generations is higher than the other cases including the default case where the fraction of elites is 0.6. This may lead to the conclusion that population develops more when the more individuals enter crossover and mutation.

Testing out different mutation probabilities for 10000 generations, the images in *Figures 54, 57 and 60* are generated. As the mutation probability increases, the performance of the population decreases, since the best fitness value provided by the case when the mutation probability is set to 0.04 is greater than the other cases.

Moreover, the impact of the mutation type is investigated. Figure 63 contains the images generated when the mutation type parameter is set to “unguided”. Clearly, guided mutation yields a better population when the best fitness values are compared to the default case.

To see the results more clear, best fitness values for all cases are printed in *Figure 66.* The case in which the number of individuals is set to 40, provided the highest “best fitness value” after 10000 generations. To test all these cases, code ran for 12 hours and 33 minutes. Time may be improved, by changing the order of the functions and by removing the dispensable lines in my algorithm.

```
Fitness Values after 10000 Generations
Default Case:       -148634320.0
num_inds_5:         -157048960.0
num_inds_10:        -154604620.0
num_inds_40:        -135191100.0  <--- best case
num_inds_60:        -141917400.0
num_genes_15:       -171409020.0  <--- worst case
num_genes_30:       -155974160.0
num_genes_80:       -140602960.0
num_genes_120:      -145649520.0
tm_size_2:          -135579460.0  <--- second best case
tm_size_8:          -144468450.0
tm_size_16:         -145802820.0
frac_elites_04:     -145365260.0
frac_elites_35:     -146890690.0
frac_parents_15:    -150951780.0
frac_parents_30:    -147087700.0
frac_parents_75:    -139664000.0  <--- third best case
mutation_prob_10:   -147493520.0
mutation_prob_40:   -148621800.0
mutation_prob_75:   -155151250.0
mutation_unguided:  -166702060.0

Runtime: 12 hours 33 minutes
```

FIGURE 66. COMPARISON OF THE BEST FITNESS VALUES AFTER 10000 GENERATIONS.

To improve the performance of the evolutionary algorithm, changes within the algorithm are introduced as “suggestions.” From now on, the case when the number of individuals is set to 40 is utilized, due to the fact that it provided the best performance. Suggestions are as follows:

Suggestion 1: *Qualified Refugees*

Every 1000, a new individual is introduced to the population, increasing the population size. This individual is created from a randomly selected elite. Since "twins" (individuals who share the identical chromosome) are undesired, two randomly selected genes of the elite is (guided) mutated before joining the population.

Images generated of the case which suggestion "Qualified Refugees" is utilized, can be observed in *Figure 67*.

Suggestion 2: *Leakage at the Nuclear Plant*

Starting from the 1000^{th} generation, population is exposed to radiation which increases the mutation probability gradually (every 50 generations) until the 2000^{th} generation. Moreover, three randomly selected genes are mutated with the mutation probability rather than one. Starting from the 3000^{th} generation, mutation is decreased gradually, until the evaluation reaches 4000^{th} generations. At the 4000^{th} generation, authorities get the leakage fully under control and evaluation continues with the default parameters.

Images generated of the case which suggestion "Leakage at the Nuclear Plant" is utilized, can be observed in *Figure 70*.

Suggestion 3: *Evil Dynasty*

Up to the 1500^{th} generation, a randomly selected elite replaces the weakest individual among the non-elites and enters the tournament followed by crossover and mutation. Fraction of parents are increased at every 1000 generations until the 4000^{th}, which leads to the growth of the royal families. After the 7000^{th} generation, dynasty chooses the weakest individual among the non-elites, use this individual as an experimental guinea pig and mutate them at each generation, to improve the performance of the population with the hopes of finding a better solution candidate that has a higher fitness value.

Images generated of the case which suggestion "Evil Dynasty is utilized, can be observed in *Figure 73*.

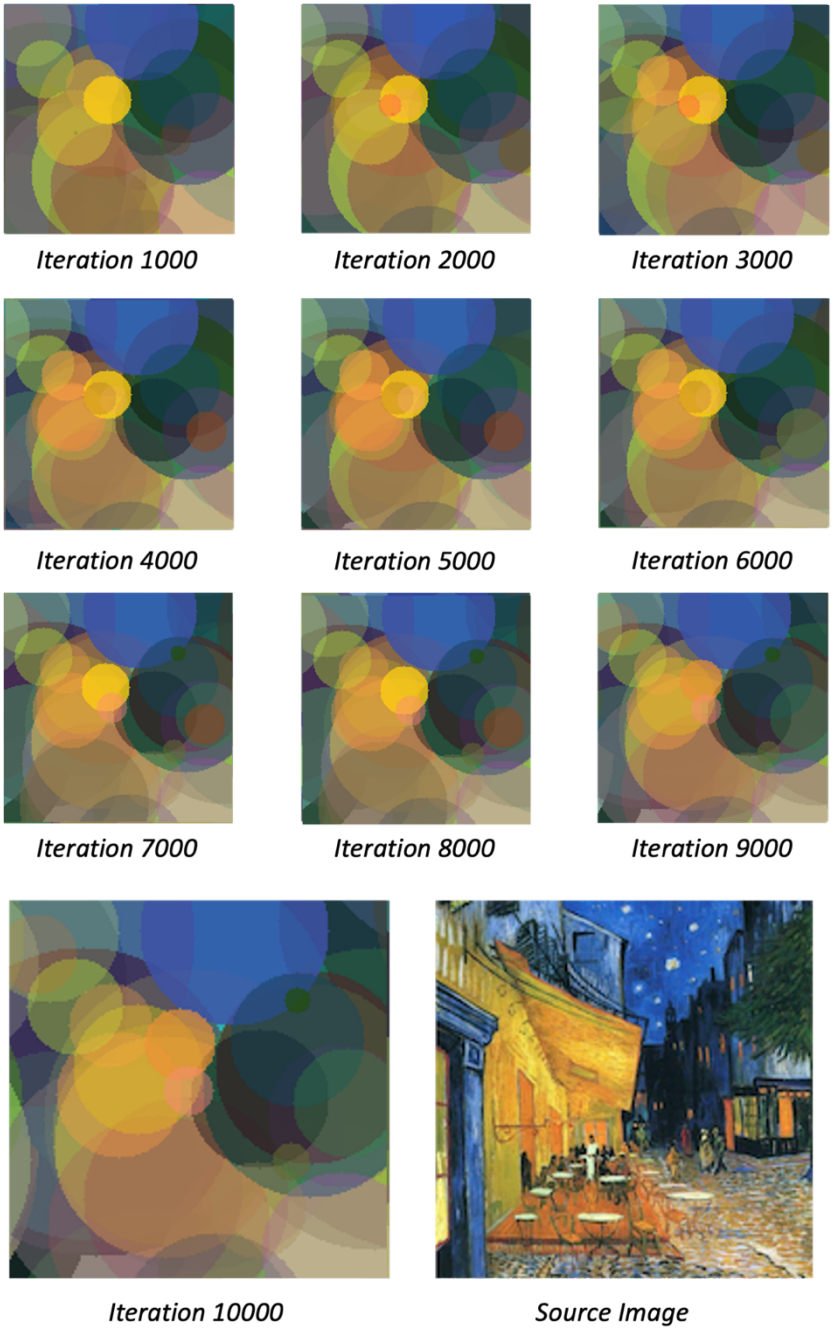
Inspecting the "Best Fitness Trend" plots in *Figures 68, 69, 71, 72, 74, and 75*, a similar trend is seen. The best fitness value never gets stuck at distinct value which prevents the population to find a better solution candidate.

Suggestion 1:

Qualified Refugees

Number of individuals is set to 40

Other parameters are set to their default values



Best fitness after 10000 generations:

-138065740.0

FIGURE 67. RESULTS OF SUGGESTION 1.

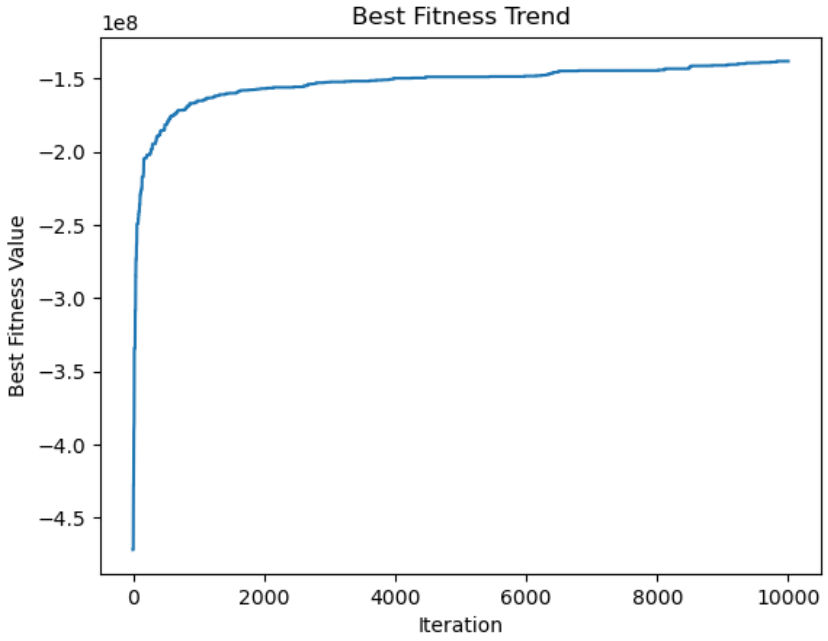


FIGURE 68. FITNESS PLOT FROM GENERATION 1 TO 10000, FOR SUGGESTION 1.

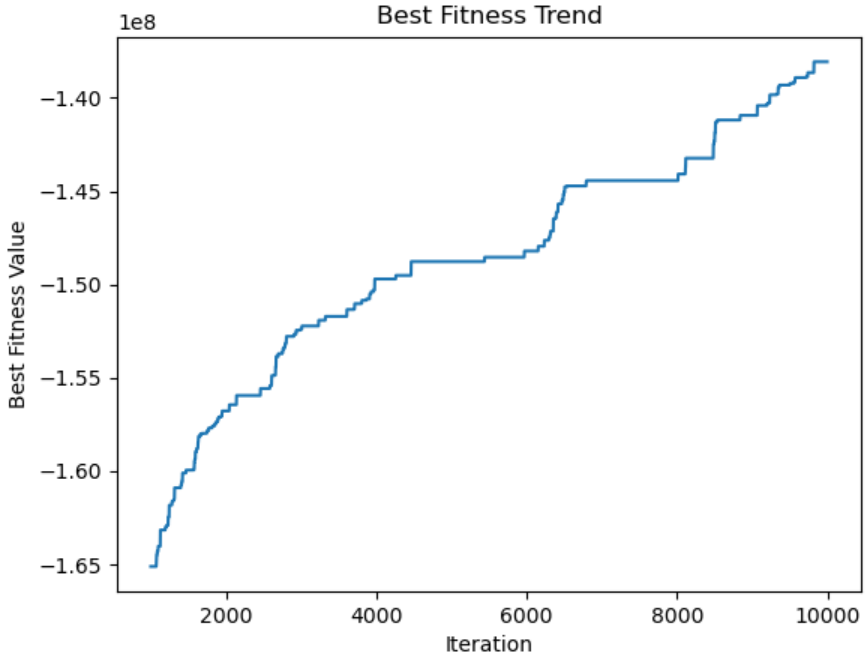
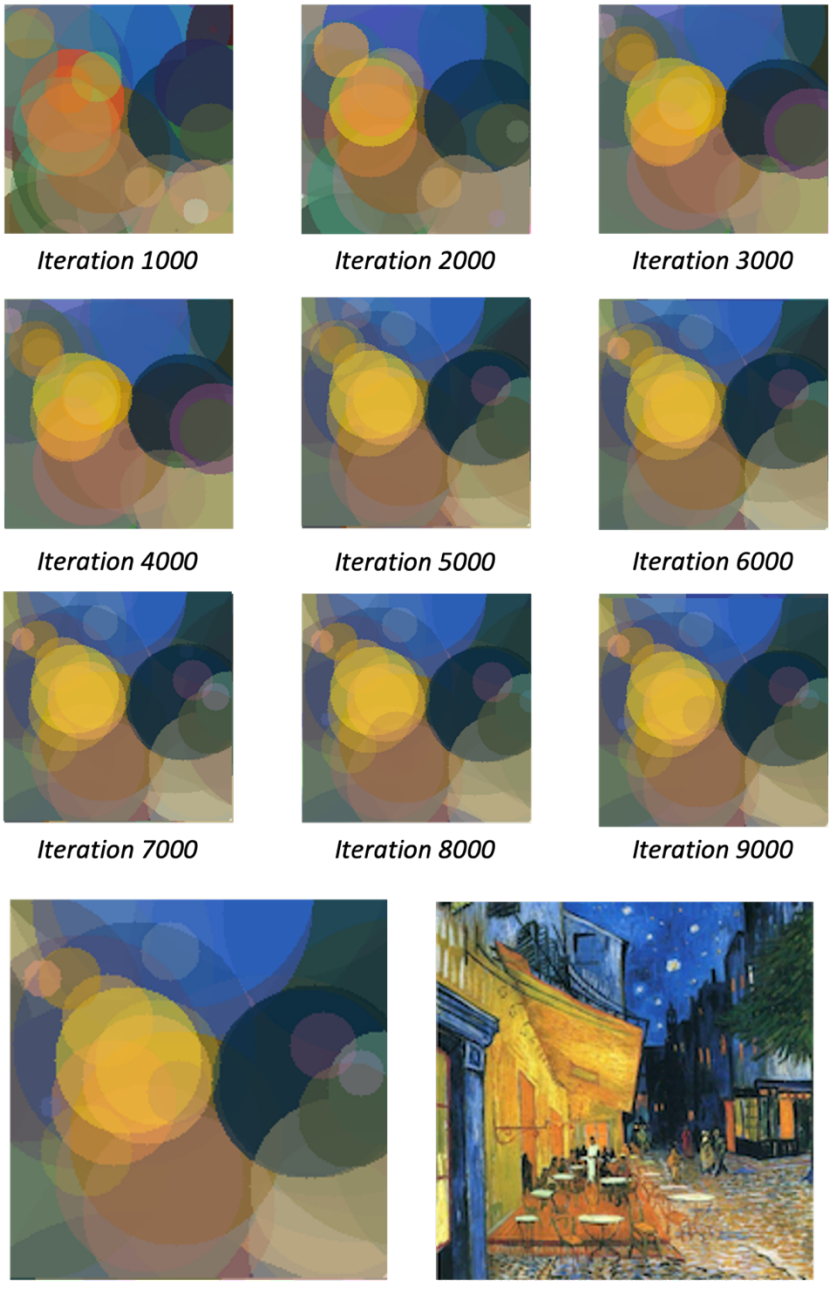


FIGURE 69. FITNESS PLOT FROM GENERATION 1000 TO 10000, FOR SUGGESTION 1.

Suggestion 2:

Leakage at the Nuclear Plant

Number of individuals is set to 40
Other parameters are set to their default values



Best fitness after 10000 generations:
-136622850.0

FIGURE 70. RESULTS OF SUGGESTION 2.

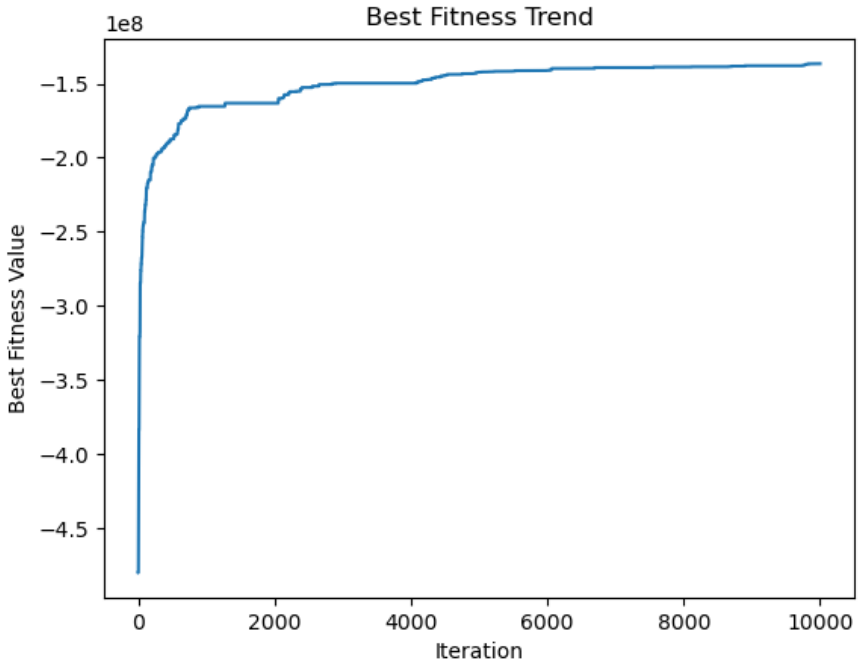


FIGURE 71. FITNESS PLOT FROM GENERATION 1 TO 10000, FOR SUGGESTION 2.

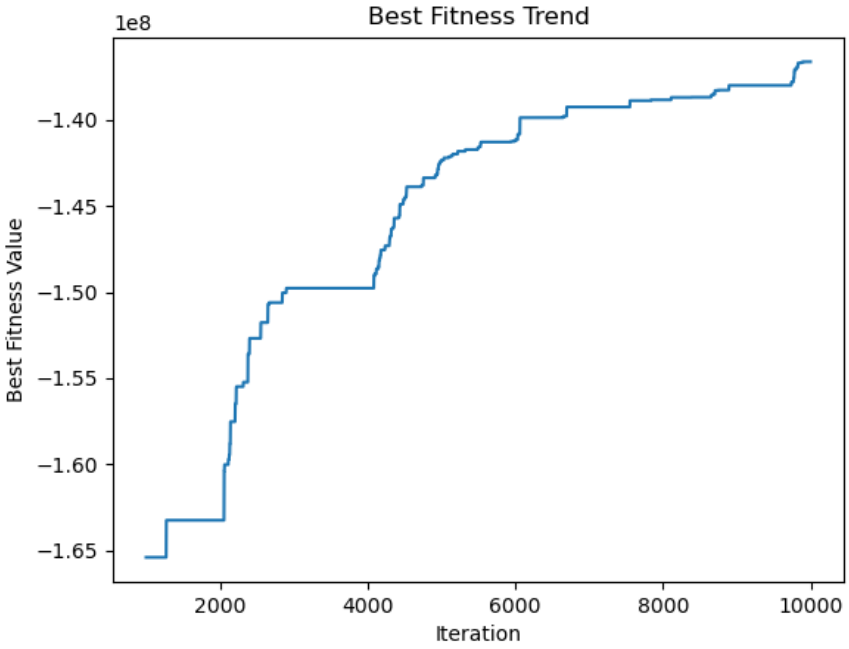


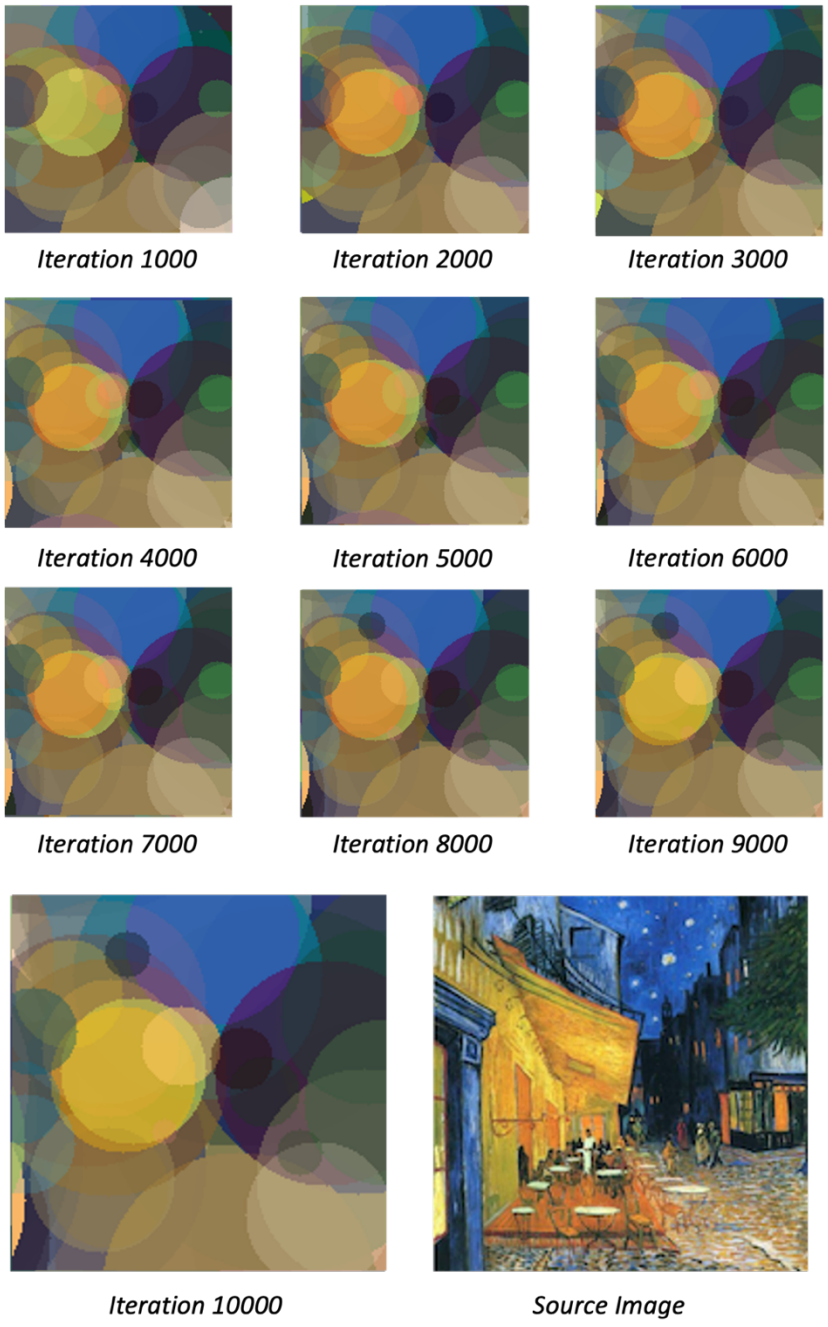
FIGURE 72. FITNESS PLOT FROM GENERATION 1000 TO 10000, FOR SUGGESTION 2.

Suggestion 3:

Evil Dynasty

Number of individuals is set to 40

Other parameters are set to their default values



Best fitness after 10000 generations:

-134715140.0

FIGURE 73. RESULTS OF SUGGESTION 3.

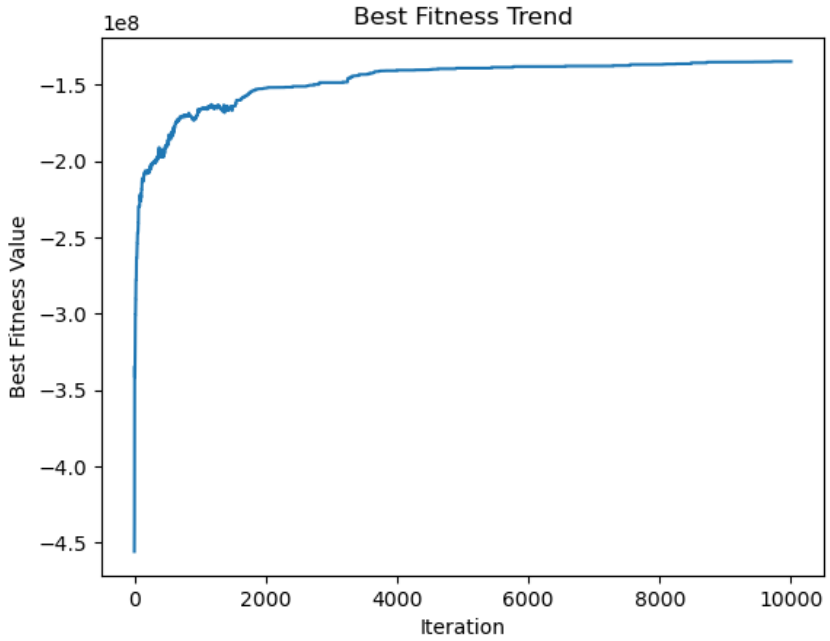


FIGURE 74. FITNESS PLOT FROM GENERATION 1 TO 10000, FOR SUGGESTION 3.

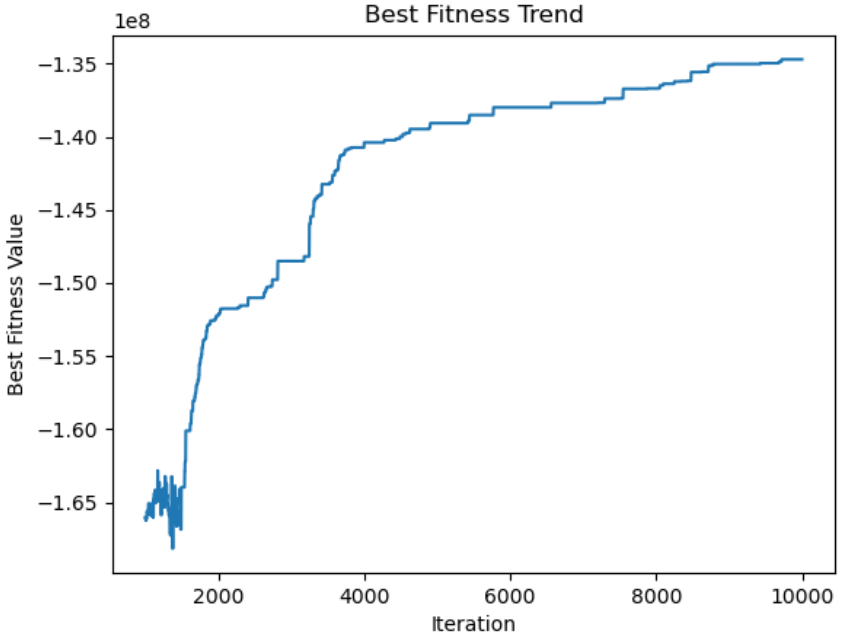


FIGURE 75. FITNESS PLOT FROM GENERATION 1000 TO 10000, FOR SUGGESTION 3.

The comparison between the suggested cases and the case when the number of individuals is set to 40 can be done easier, inspecting the data in *Figure 76*, where the fitness values of the suggested cases after 10000 generations are written.

```
num_inds_40:                              -135191100.0  <-- previous best case

Fitness Values of the 'Suggested Cases' after 10000 Generations

Qualified Refugees:                       -138065740.0 (no improvement)
Leakage at the Nuclear Plant:             -136622850.0 (no improvement)
Evil Dynasty:                             -134715140.0  <-- new best case !!!
```

FIGURE 76. COMPARISON OF THE PERFORMANCES OF THE SUGGESTIONS.

Although the results are close to each other, "**Qualified Refugees"** and "**Leakage at the Nuclear Plant**" Suggestions did improve the performance of the evolutionary algorithm no further. However, a higher fitness value after 10000 generations can be observed for the "**Evil Dynasty"** Suggestion.

In conclusion, this experiment explored the application of evolutionary algorithms for generating an image resembling Van Gogh's "Terrasse du Café le Soir" painting. By manipulating various hyperparameters, we investigated their impact on the algorithm's performance over 10,000 generations.

Through systematic experimentation, we observed that adjusting specific hyperparameters significantly influenced the quality and convergence speed of the generated image. Notably, one of the suggested improvements (Suggestion 3) yielded superior results, highlighting the potential for optimizing the evolutionary algorithm.

These findings emphasize the importance of carefully selecting and fine-tuning hyperparameters for image generation tasks. By exploring different parameter configurations, we can further enhance the algorithm's performance and generate visually appealing images.

In summary, this study underscores the significance of hyperparameter optimization in evolutionary algorithms and provides valuable insights for future research in this domain.

APPENDIX

```
################################################################################################
##############
# EE449 Homework 2 - Evolutionary Algorithms for Image Generation
# Arda Ünver 2444081
# This file contains the main function of the program.
################################################################################################
##############

# Import libraries
import time
import cv2
from random import random
import random
import numpy as np
import os
import matplotlib.pyplot as plt
import copy
import pickle

# Gene Class
class Gene:
    def __init__(self, image_width, image_height):
        self.image_width = image_width
        self.image_height = image_height

        # Initialize the center coordinates of the gene
        temp_center = self.random_center()
        # Initialize the radius of the gene
        temp_radius = random.randint(1, max(image_width, image_height)/2)

        # Ensure that the circle has an area within the image boundaries
        while not self.within_the_image(temp_center, temp_radius):
            temp_center = self.random_center()
            temp_radius = random.randint(1, max(image_width, image_height)/2)

        self.center = temp_center
        self.radius = temp_radius
        self.color = self.random_color()

    def random_center(self):
        # Generate random center coordinates within image boundaries
        return random.randint(0, 1.65*self.image_width), random.randint(0,
1.65*self.image_height)

    def random_color(self):
        return (
            # Generate a random value for the red channel
            random.randint(0, 255),
```

```
            # Generate a random value for the green channel
            random.randint(0, 255),
            # Generate a random value for the blue channel
            random.randint(0, 255),
            # Generate a random value for the alpha channel (transparency)
            random.uniform(0, 1),
        )

    def within_the_image(self, center, radius):
        x_location = abs(center[0] - self.image_width / 2)
        y_location = abs(center[1] - self.image_height / 2)

        if x_location > (self.image_width / 2 + radius):
            return False
        if y_location > (self.image_height / 2 + radius):
            return False

        if x_location <= (self.image_width / 2):
            return True
        if y_location <= (self.image_height / 2):
            return True

        corner = (x_location - self.image_width / 2) ** 2 + (y_location -
self.image_height / 2) ** 2

        return (corner <= (radius ** 2))

    def guided_mutation(self):
        # Deviate the center coordinates by adding random values within a quarter of
image width and height
        # center[0] and center[1] denote x and y coordinates of the center,
respectively
        center_temp = (
            self.correct_value(
                self.center[0]
                + random.randint(-self.image_width // 4, self.image_width // 4)
            ),
            self.correct_value(
                self.center[1]
                + random.randint(-self.image_height // 4, self.image_height // 4)
            ),
        )

        # Deviate the radius by adding a random value within the range of -10 to 10
        # correct_value() function is used to ensure that the radius does not exceed
        # the maximum image width or height or fall below 1
        radius_temp = self.correct_value(
            self.radius + random.randint(-10, 10),
            1,
            max(self.image_width, self.image_height),
```

```
        )

        while not self.within_the_image(center_temp, radius_temp):
            center_temp = (
                self.correct_value(
                    self.center[0]
                    + random.randint(-self.image_width // 4, self.image_width // 4)
                ),
                self.correct_value(
                    self.center[1]
                    + random.randint(-self.image_height // 4, self.image_height // 4)
                ),
            )

            radius_temp = self.correct_value(
                self.radius + random.randint(-10, 10),
                1,
                max(self.image_width, self.image_height),
            )

        self.center = center_temp
        self.radius = radius_temp

        # Deviate the color components by adding random values within the range of -64
to 64
        # correct_value() function is used to ensure that the color components do not
exceed
        # 255 or fall below 0
        self.color = (
            self.correct_value(self.color[0] + random.randint(-64, 64), 0, 255),
            self.correct_value(self.color[1] + random.randint(-64, 64), 0, 255),
            self.correct_value(self.color[2] + random.randint(-64, 64), 0, 255),
            self.correct_value(self.color[3] + random.uniform(-0.25, 0.25), 0, 1),
        )

    def unguided_mutation(self):
        # Randomly reinitialize the center coordinates
        self.center = self.random_center()
        # Randomly assign a new radius within the range of 1 to the maximum image
width or height
        self.radius = random.randint(1, max(self.image_width, self.image_height))
        # Randomly assign a new color
        self.color = self.random_color()


    def correct_value(self, value, min_value=None, max_value=None):
        # Correct the value if it exceeds the specified minimum or maximum values
        if min_value is not None:
            value = max(value, min_value)
        if max_value is not None:
```

```
            value = min(value, max_value)
        return value

# Individual Class
class Individual:
    def __init__(self, num_genes, image_width, image_height):
        # Initialize an individual with a given number of genes in the chromosome
        self.num_genes = num_genes
        self.image_width = image_width
        self.image_height = image_height
        self.chromosome = []
        self.fitness = 0
        # Generate a random ID for the individual
        self.ID = random.randint(0, 1000000)

        # Create genes for the chromosome
        for _ in range(num_genes):
            # Create a gene object
            gene = Gene(image_width, image_height)

            # Add the gene to the chromosome by appending to the chromosome list
            self.chromosome.append(gene)

    def evaluate(self, source_image):

        # Sort the genes in chromosome by descending radius order
        self.chromosome = sorted(self.chromosome, key=lambda x: x.radius,
reverse=True)

        # Convert the source image to a numpy array
        source_image = np.array(source_image, dtype=np.float32)

        # Initialize a white image with the same shape as the source image
        image = np.ones_like(source_image) * 255

        # Iterate over each gene in the chromosome
        for gene in self.chromosome:
            # Create a copy of the image to overlay the circle
            overlay = np.copy(image)

            # Draw the circle on the overlay
            overlay = self.draw_circle(overlay, gene)

            # Retrieve the alpha value (transparency) from the gene's color
            alpha = gene.color[3]

            # addWeighted() function performs alpha blending
            image = cv2.addWeighted(overlay, alpha, image, 1 - alpha, 0, image)
```

```
        # Calculate the fitness value using the squared difference between the source
image and the individual's image
        self.fitness = -np.sum(np.square((source_image - image)))

        return self.fitness

    def mutation(self, guided=True):
        # Perform guided or unguided mutation based on the 'guided' parameter
        # Performs guided mutation by default
        for gene in self.chromosome:
            if guided:
                gene.guided_mutation()
            else:
                gene.unguided_mutation()

        self.evaluate(source_image)

    def draw_circle(self, image, gene):
        # print("Drawing Circle of Individual")
        # Retrieve the center coordinates and radius of the gene
        center_x, center_y = gene.center
        radius = gene.radius

        # Retrieve the color components of the gene
        color_r, color_g, color_b, _ = gene.color

        # Draw the circle on the image using the bounding box and color
        cv2.circle(
            image,
            center=(int(center_x), int(center_y)),
            radius=int(radius),
            color=(int(color_r), int(color_g), int(color_b)),
            thickness=-1,
        )
        return image


class Population:
    def __init__(
        self,
        num_inds = 20,
        num_genes = 50,
        tm_size = 5,
        frac_elites = 0.2,
        frac_parents = 0.6,
        mutation_prob = 0.2,
        mutation_type = "guided",
        im_width = 180,
        im_height = 180,
        num_iterations = 10000,
```

```
        suggestion = 'None',
        beginning_time = time.time(),
    ):
        self.num_inds = num_inds
        self.num_genes = num_genes
        self.tm_size = tm_size
        self.frac_elites = frac_elites
        self.frac_parents = frac_parents
        self.mutation_prob = mutation_prob
        self.mutation_type = mutation_type
        self.population = []
        self.current_generation = 0
        self.num_iterations = num_iterations
        self.suggestion = suggestion

        # Defined globally
        self.image_width = im_width
        self.image_height = im_height

        # Time at which the algorithm started
        self.beginning_time = beginning_time

        # List to store the best fitness values of each generation
        self.best_fitness_values = []

        # Create the initial population
        for _ in range(num_inds):
            individual = Individual(num_genes, self.image_width, self.image_height)
            self.population.append(individual)

    def tournament(self, remaining):
        # Randomly choose one individual as the temporary winner
        best_individual = random.randint(0, len(remaining) - 1)
        best_fitness = remaining[best_individual].fitness

        # Iterate tm_size - 1 times
        for _ in range(self.tm_size - 1):
            # Randomly choose another individual as a warrior
            current_individual = random.randint(0, len(remaining) - 1)
            current_fitness = remaining[current_individual].fitness

            # Compare the fitness of the warrior with the best fitness
            if current_fitness > best_fitness:
                # If the warrior has a higher fitness, update the best fitness and
index
                best_fitness = current_fitness
                best_individual = current_individual

        # Remove the winner from the current generation and return it
        winner = remaining.pop(best_individual)
```

```
        return winner, remaining

    def crossover(self, parents, num_parents):

        children = []

        for i in range(0, num_parents, 2):
            chromosome1 = []
            chromosome2 = []
            # Create a random binary string
            binary = np.random.randint(2, size=self.num_genes)

            for gene in range(self.num_genes):
                if binary[gene] == 0:
                    chromosome1.append(copy.deepcopy(parents[i].chromosome[gene]))
                    chromosome2.append(copy.deepcopy(parents[i + 1].chromosome[gene]))
                else:
                    chromosome1.append(copy.deepcopy(parents[i + 1].chromosome[gene]))
                    chromosome2.append(copy.deepcopy(parents[i].chromosome[gene]))

            # Create two children
            child1 = Individual(self.num_genes, im_width, im_height)
            child2 = Individual(self.num_genes, im_width, im_height)

            # Set the chromosomes of the children via crossover
            child1.chromosome = chromosome1
            child2.chromosome = chromosome2

            # Evaluate the children
            child1.evaluate(source_image)
            child2.evaluate(source_image)

            # Select the best two individuals from the family
            family = self.sortIndividuals([parents[i], parents[i + 1], child1,
child2])

            # Add the best two individuals to the children list
            children.append(family[0])
            children.append(family[1])

        return children

    # Sorts the individuals in a group in descending order of fitness
    def sortIndividuals(self, group):
        return sorted(group, key=lambda item: item.fitness, reverse=True)

    def mutation(self, non_elites):
        for individual in non_elites:
            # Randomly select a gene from the chromosome
            index = random.randrange(len(individual.chromosome))
```

```
            if random.random() < self.mutation_prob:
                if self.mutation_type == "guided":
                    individual.chromosome[index].guided_mutation()
                else:
                    individual.chromosome[index].unguided_mutation()
            # Evaluate the fitness of the individual
            individual.evaluate(source_image)

    def leakage_at_the_nuclear_plant(self, non_elites, iteration):
        # Leakage at the nuclear plant starts at 1000th iteration
        if (iteration >= 1000) and (iteration < 2000):

            # Increment the mutation probability by 0.01 at every 50 iterations
            self.radiation(trend='up', iteration=iteration)

            for individual in non_elites:
                # Randomly select a gene from the chromosome
                idx1 = random.randrange(len(individual.chromosome))
                idx2 = random.randrange(len(individual.chromosome))
                idx3 = random.randrange(len(individual.chromosome))

                # Mutate the 3 genes
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx1].guided_mutation()
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx2].guided_mutation()
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx3].guided_mutation()

                # Evaluate the fitness of the individual
                individual.evaluate(source_image)

        # Authorities got the leak under control at 3000th iteration
        elif (iteration >= 3000) and (iteration < 4000):

            # Decrement the mutation probability by 0.01 at every 50th iteration
            self.radiation(trend='down', iteration=iteration)

            for individual in non_elites:
                # Randomly select a gene from the chromosome
                idx4 = random.randrange(len(individual.chromosome))
                idx5 = random.randrange(len(individual.chromosome))
                idx6 = random.randrange(len(individual.chromosome))

                # Mutate the 3 genes
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx4].guided_mutation()
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx5].guided_mutation()
```

```
                if random.random() < self.mutation_prob:
                    individual.chromosome[idx6].guided_mutation()

                # Evaluate the fitness of the individual
                individual.evaluate(source_image)
        else:
            if iteration == 4000:
                print("Leakage at the nuclear plant has been fixed!")

            for individual in non_elites:
                # Randomly select a gene from the chromosome
                idx = random.randrange(len(individual.chromosome))

                if random.random() < self.mutation_prob:
                    if self.mutation_type == "guided":
                        individual.chromosome[idx].guided_mutation()
                    else:
                        individual.chromosome[idx].unguided_mutation()
                # Evaluate the fitness of the individual
                individual.evaluate(source_image)

    def radiation(self, trend, iteration):
        if trend == "up":
            # Increment the mutation probability by 0.01 at every 50 iterations
            if iteration % 50 == 0:
                self.mutation_prob += 0.01
                print("Radiation leak at the nuclear plant! Mutation probability
increased by 0.01")
                print("Mutation probability is now: ",
"{:.2f}".format(self.mutation_prob))
        elif trend == "down":
            # Decrement the mutation probability by 0.01 at every 50th iteration
            if iteration % 50 == 0:
                self.mutation_prob -= 0.01
                print("Radiation leakage stopped! Mutation probability decreased by
0.01")
                print("Mutation probability is now: ",
"{:.2f}".format(self.mutation_prob))

    def evaluate(self, source_image, operation_name):

        for i in range(1, self.num_iterations + 1):

            # Evaluate the fitness of each individual in the population
            fitness_values = [individual.evaluate(source_image) for individual in
self.population]

            # Record the best fitness value of the current generation
            best_fitness = max(fitness_values)
            self.best_fitness_values.append(best_fitness)
```

```
            # Sort the population in descending order of fitness
            self.population = sorted(self.population, key=lambda x: x.fitness,
reverse=True)

            #  Determine the number of elites, and parents
            num_elites = int(self.frac_elites * self.num_inds)
            num_parents = int(self.frac_parents * self.num_inds)

            # Suggestion: Qualified Refugees
            if (self.suggestion == "refugee") and (i % 1000 == 0):
                self.qualified_refugees()

            # If the number of parents is odd, make it even
            if num_parents % 2 != 0:
                num_parents += 1

            # Select the elites (best individuals)
            # to advance directly to the next generation
            elites = self.population[:num_elites]
            remaining = self.population[num_elites:]

            # Suggestion: Dynasty
            if self.suggestion == "dynasty":
                remaining = self.evil_dynasty_rules(num_elites=num_elites,
elites=elites, remaining=remaining, iteration=i)

            parents = []

            # Select the parents for crossover from the remaining individuals
            for z in range(num_parents):
                winner, remaining = self.tournament(remaining)
                parents.append(winner)

            others = remaining

            offspring = self.crossover(parents, num_parents)

            non_elites = offspring + others

            # Suggestion: Radiation
            if self.suggestion == "radiation":
                self.leakage_at_the_nuclear_plant(non_elites=non_elites, iteration=i)
            else:
                self.mutation(non_elites=non_elites)

            next_generation = elites + non_elites

            self.population = next_generation
```

```
            self.population = sorted(self.population, key=lambda x: x.fitness,
reverse=True)

            if i % 50 == 0:
                # Print the best and worst fitness in the population
                print(f"Iteration {i}: ID: {self.population[0].ID}, "
                      f"Best Fitness: {self.population[0].fitness},"
                      f"Time: {time.time() - self.beginning_time}")

            save_path = f"results/{operation_name}"

            # Display the images every 100 iterations
            if i % 1000 == 0:
                self.display_images(i, save_path)

            if i == 10000:
                # Save 2 plots
                # 1. Fitness vs Iteration 1-10000
                # 2. Fitness vs Iteration 1000-10000
                self.plot_fitness(save_path)

                pickle_path = f"results/{operation_name}best_fitness_values.pickle"
                # Save the data as a pickle file
                with open(pickle_path, 'wb') as file:
                    pickle.dump(self.best_fitness_values, file)

    def qualified_refugees(self):
        # Qualified refugees suggestion

        # Refugee immigration
        refugee = Individual(self.num_genes, self.image_width, self.image_height)

        #  Determine the number of elites, and parents
        num_elites = int(self.frac_elites * self.num_inds)

        # Choose a random elite
        ref_idx = random.randrange(num_elites)

        # Set the refugee's chromosome to the elite's chromosome
        refugee.chromosome = self.population[ref_idx].chromosome

        # Mutate the refugee's chromosome by a random gene twice
        gene_idx1 = random.randrange(len(refugee.chromosome))
        gene_idx2 = random.randrange(len(refugee.chromosome))
        refugee.chromosome[gene_idx1].guided_mutation()
        refugee.chromosome[gene_idx2].guided_mutation()

        # Evaluate the refugee
        refugee.evaluate(source_image)
```

```
        # Add the refugee to the population
        self.population.append(refugee)
        # Increase the population size
        previous_num_inds = self.num_inds
        self.num_inds += 1

        print("Population size: ", previous_num_inds, " --> ", self.num_inds)

    def evil_dynasty_rules(self, num_elites, elites, remaining, iteration):

        if iteration <= 1500:
            # Choose a random elite which will enter the tournament at the earlier
stages
            royal_idx = random.randrange(num_elites)

            # Replace the weakest individual with the elite
            remaining[-1].chromosome = elites[royal_idx].chromosome

            # Evaluate the fitness of the individual
            remaining[-1].evaluate(source_image)

        # Some elites enter the tournament (selection) at the earlier stages to
        # grow their royal families. As the dynasty grows, fraction of parents
increases.
        if (iteration <= 4000) and (iteration % 1000 == 0):
            previous_frac_parents = self.frac_parents
            self.frac_parents += 0.025
            print("Fraction of parents: ", previous_frac_parents, " --> ",
"{:.2f}".format(self.frac_parents))

        # At later stages, the dynasty chooses the weakest, use these individuals
        # as an experimental guniea pig and mutate them to find better solutions
        if iteration <= 7000:
            # Choose random genes from the chromosome and mutate them
            gene_idx1 = random.randrange(len(remaining[-1].chromosome))
            gene_idx2 = random.randrange(len(remaining[-1].chromosome))
            remaining[-1].chromosome[gene_idx1].guided_mutation()
            remaining[-1].chromosome[gene_idx2].guided_mutation()
            remaining[-1].evaluate(source_image)

        return sorted(remaining, key=lambda x: x.fitness, reverse=True)

    def display_images(self, iteration, output_dir):

        # Create a directory to save the generated images
        os.makedirs(output_dir, exist_ok=True)

        best = self.population[0]

        # Initialize a white image with the same shape as the source image
```

```
        image = np.ones_like(source_image) * 255

        # Iterate over each gene in the chromosome
        for gene in best.chromosome:
            # Create a copy of the image to overlay the circle
            overlay = np.copy(image)

            # Draw the circle on the overlay
            overlay = best.draw_circle(overlay, gene)

            # Blend the overlay with the image using alpha (transparency) blending
            alpha = gene.color[3]

            # addWeighted calculates the weighted sum of two arrays
            image = cv2.addWeighted(overlay, alpha, image, 1 - alpha, 0, image)

        print("Displaying Images...")
        # Save the image created by the individual
        image_path = f"{output_dir}iteration{iteration}.png"
        cv2.imwrite(image_path, image)


    def plot_fitness(self, save_path):
        plt.plot(range(10000), self.best_fitness_values)
        plt.xlabel('Iteration')
        plt.ylabel('Best Fitness Value')
        plt.title('Best Fitness Trend')
        plt.savefig(save_path + "plot_1to10000_.png")
        plt.clf()
        plt.close()

        plt.plot(range(1000, 10000), self.best_fitness_values[1000:10000])
        plt.xlabel('Iteration')
        plt.ylabel('Best Fitness Value')
        plt.title('Best Fitness Trend')
        plt.savefig(save_path + "plot_1000to10000_.png")
        plt.clf()
        plt.close()

source_image = cv2.imread("image/painting.png")

im_width, im_height, _ = source_image.shape

beginning_time = time.time()

# Default Case

default_population = Population()

default_population.evaluate(source_image, "default/")
```

```
# Testing num_inds parameter
population_num_inds_5 = Population(num_inds=5)
population_num_inds_5.evaluate(source_image, "num_inds_5/")
population_num_inds_10 = Population(num_inds=10)
population_num_inds_10.evaluate(source_image, "num_inds_10/")
population_num_inds_40 = Population(num_inds=40)
population_num_inds_40.evaluate(source_image, "num_inds_40/")
population_num_inds_60 = Population(num_inds=60)
population_num_inds_60.evaluate(source_image, "num_inds_60/")
# Testing num_genes parameter
population_num_genes_15 = Population(num_genes=15)
population_num_genes_15.evaluate(source_image, "num_genes_15/")
population_num_genes_30 = Population(num_genes=30)
population_num_genes_30.evaluate(source_image, "num_genes_30/")
population_num_genes_80 = Population(num_genes=80)
population_num_genes_80.evaluate(source_image, "num_genes_80/")
population_num_genes_120 = Population(num_genes=120)
population_num_genes_120.evaluate(source_image, "num_genes_120/")
# Testing tm_size parameter
population_tm_size_2 = Population(tm_size=2)
population_tm_size_2.evaluate(source_image, "tm_size_2/")
population_tm_size_8 = Population(tm_size=8)
population_tm_size_8.evaluate(source_image, "tm_size_8/")
population_tm_size_16 = Population(tm_size=16)
population_tm_size_16.evaluate(source_image, "tm_size_16/")
```

```
# Testing frac_elites parameter
population_frac_elites_04 = Population(frac_elites=0.04)
population_frac_elites_04.evaluate(source_image, "frac_elites_04/")
population_frac_elites_35 = Population(frac_elites=0.35)
population_frac_elites_35.evaluate(source_image, "frac_elites_35/")
# Testing frac_parents parameter
population_frac_parents_15 = Population(frac_parents=0.15)
population_frac_parents_15.evaluate(source_image, "frac_parents_15/")
population_frac_parents_30 = Population(frac_parents=0.30)
population_frac_parents_30.evaluate(source_image, "frac_parents_30/")
population_frac_parents_75 = Population(frac_parents=0.75)
population_frac_parents_75.evaluate(source_image, "frac_parents_75/")
# Testing mutation_prob parameter
population_mutation_prob_10 = Population(mutation_prob=0.1)
population_mutation_prob_10.evaluate(source_image, "mutation_prob_10/")
population_mutation_prob_40 = Population(mutation_prob=0.4)
population_mutation_prob_40.evaluate(source_image, "mutation_prob_40/")
population_mutation_prob_75 = Population(mutation_prob=0.75)
population_mutation_prob_75.evaluate(source_image, "mutation_prob_75/")

# Testing mutation_type parameter
population_mutation_unguided = Population(mutation_type="unguided")
population_mutation_unguided.evaluate(source_image, "mutation_unguided/")
# Testing Suggestions
dynasty_population = Population(num_inds=40, suggestion="dynasty")
```

```
dynasty_population.evaluate(source_image, "evil_dynasty/")

radiated_population = Population(num_inds=40, suggestion="radiation")

radiated_population.evaluate(source_image, "leakage_at_nuclear_plant/")

refuguee_population = Population(num_inds=40, suggestion="refugee")

refuguee_population.evaluate(source_image, "qualified_refugees/")

with open('results/default/best_fitness_values.pickle', 'rb') as file:
    default_bf = pickle.load(file)

with open('results/num_inds_5/best_fitness_values.pickle', 'rb') as file:
    num_inds_5_bf = pickle.load(file)

with open('results/num_inds_10/best_fitness_values.pickle', 'rb') as file:
    num_inds_10_bf = pickle.load(file)

with open('results/num_inds_40/best_fitness_values.pickle', 'rb') as file:
    num_inds_40_bf = pickle.load(file)

with open('results/num_inds_60/best_fitness_values.pickle', 'rb') as file:
    num_inds_60_bf = pickle.load(file)

with open('results/num_genes_15/best_fitness_values.pickle', 'rb') as file:
    num_genes_15_bf = pickle.load(file)

with open('results/num_genes_30/best_fitness_values.pickle', 'rb') as file:
    num_genes_30_bf = pickle.load(file)

with open('results/num_genes_80/best_fitness_values.pickle', 'rb') as file:
    num_genes_80_bf = pickle.load(file)

with open('results/num_genes_120/best_fitness_values.pickle', 'rb') as file:
    num_genes_120_bf = pickle.load(file)

with open('results/tm_size_2/best_fitness_values.pickle', 'rb') as file:
    tm_size_2_bf = pickle.load(file)

with open('results/tm_size_8/best_fitness_values.pickle', 'rb') as file:
    tm_size_8_bf = pickle.load(file)

with open('results/tm_size_16/best_fitness_values.pickle', 'rb') as file:
    tm_size_16_bf = pickle.load(file)

with open('results/frac_elites_04/best_fitness_values.pickle', 'rb') as file:
    frac_elites_04_bf = pickle.load(file)

with open('results/frac_elites_35/best_fitness_values.pickle', 'rb') as file:
```

```
    frac_elites_35_bf = pickle.load(file)

with open('results/frac_parents_15/best_fitness_values.pickle', 'rb') as file:
    frac_parents_15_bf = pickle.load(file)

with open('results/frac_parents_30/best_fitness_values.pickle', 'rb') as file:
    frac_parents_30_bf = pickle.load(file)

with open('results/frac_parents_75/best_fitness_values.pickle', 'rb') as file:
    frac_parents_75_bf = pickle.load(file)

with open('results/mutation_prob_10/best_fitness_values.pickle', 'rb') as file:
    mutation_prob_10_bf = pickle.load(file)

with open('results/mutation_prob_40/best_fitness_values.pickle', 'rb') as file:
    mutation_prob_40_bf = pickle.load(file)

with open('results/mutation_prob_75/best_fitness_values.pickle', 'rb') as file:
    mutation_prob_75_bf = pickle.load(file)

with open('results/mutation_unguided/best_fitness_values.pickle', 'rb') as file:
    mutation_unguided_bf = pickle.load(file)

with open('results/dynasty/best_fitness_values.pickle', 'rb') as file:
    dynasty_bf = pickle.load(file)

with open('results/evil_dynasty/best_fitness_values.pickle', 'rb') as file:
    dynasty_bf = pickle.load(file)

with open('results/leakage_at_nuclear_plant/best_fitness_values.pickle', 'rb') as
file:
    radiation_bf = pickle.load(file)

with open('results/qualified_refugees/best_fitness_values.pickle', 'rb') as file:
    refugees_bf = pickle.load(file)

print("\n")
print("     Fitness Values after 10000 Generations")
print("     Default Case:      ", default_bf[-1])
print("     num_inds_5:        ", num_inds_5_bf[-1])
print("     num_inds_10:       ", num_inds_10_bf[-1])
print("     num_inds_40:       ", num_inds_40_bf[-1], " <--- best case")
print("     num_inds_60:       ", num_inds_60_bf[-1])
print("     num_genes_15:      ", num_genes_15_bf[-1], " <--- worst case")
print("     num_genes_30:      ", num_genes_30_bf[-1])
print("     num_genes_80:      ", num_genes_80_bf[-1])
print("     num_genes_120:     ", num_genes_120_bf[-1])
print("     tm_size_2:         ", tm_size_2_bf[-1], " <--- second best case")
print("     tm_size_8:         ", tm_size_8_bf[-1])
print("     tm_size_16:        ", tm_size_16_bf[-1])
```

```
print("     frac_elites_04:    ", frac_elites_04_bf[-1])
print("     frac_elites_35:    ", frac_elites_35_bf[-1])
print("     frac_parents_15:   ", frac_parents_15_bf[-1])
print("     frac_parents_30:   ", frac_parents_30_bf[-1])
print("     frac_parents_75:   ", frac_parents_75_bf[-1], " <--- third best case")
print("     mutation_prob_10:  ", mutation_prob_10_bf[-1])
print("     mutation_prob_40:  ", mutation_prob_40_bf[-1])
print("     mutation_prob_75:  ", mutation_prob_75_bf[-1])
print("     mutation_unguided: ", mutation_unguided_bf[-1], "\n")

print("     Runtime: 12 hours 33 minutes")

print("\n")
print("     num_inds_40:                           ", num_inds_40_bf[-1], " <--
previous best case\n")
print("     Fitness Values of the 'Suggested Cases' after 10000 Generations")
print("     Qualified Refugees:                    ", refugees_bf[-1], "(no
improvement)")
print("     Leakage at the Nuclear Plant:          ", radiation_bf[-1], "(no
improvement)")
print("     Evil Dynasty:                         ", dynasty_bf[-1], " <-- new best
case !!!")
```