Ahmet Akman 2442366
Middle East Technical University
Electrical and Electronics Engineering

April 28, 2024

HOMEWORK 2 — Report

1 Experimental Work

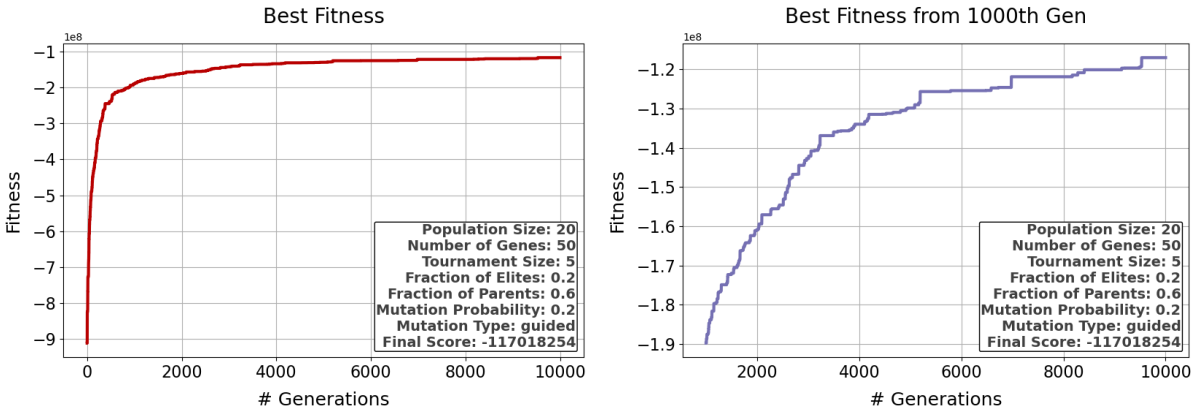
The parameter set provided in Table 1 is used to perform qualitative analysis of different evoulationary algorithm parameters.

Table 1: Parameter set used in this experiment.

Parameter	**Values**				
<num_inds>	5	10	**20**	40	60
<num_genes>	15	30	**50**	80	120
<tm_size>	2	**5**	8	16	
<frac_elites>	0.04	**0.2**	0.35		
<frac_parents>	0.15	0.3	**0.6**	0.75	
<mutation_prob>	0.1	**0.2**	0.4	0.75	
<mutation_type>	unguided	**guided**			

1.1 Default Parameter Set

Let us first start with the results of the default parameter set provided in the homework description to be used as a baseline throughout the experiment. Figure 1 shows the plots associated with the fitness value of the best individual. The default parameter set is indicated as bold in Table 1.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 1: Fitness curves.

The figures tell us that in the first 1000 generations, the fitness value of the best individual increases rapidly. However, after the 1000th generation, the increase in fitness value slows down. However, the fitness value of the best individual is still increasing, as can be seen in the figure on the right. That means the algorithm continues to improve even after a rapid jump.

Figure 2 shows the evolution of the best individual quantitatively. The first image is the image in the 1000th generation, and the last image is the final image. The images in between are the images of the best individual in every 1000 generations. The images are generated by overlaying the circle represented by each gene one by one on the plain white image as instructed.

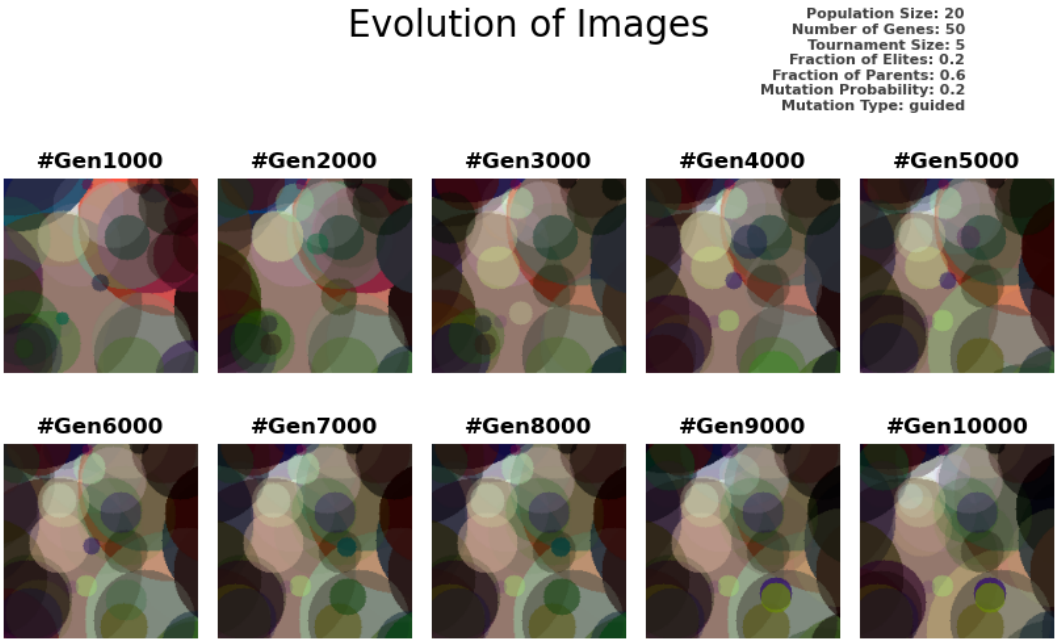


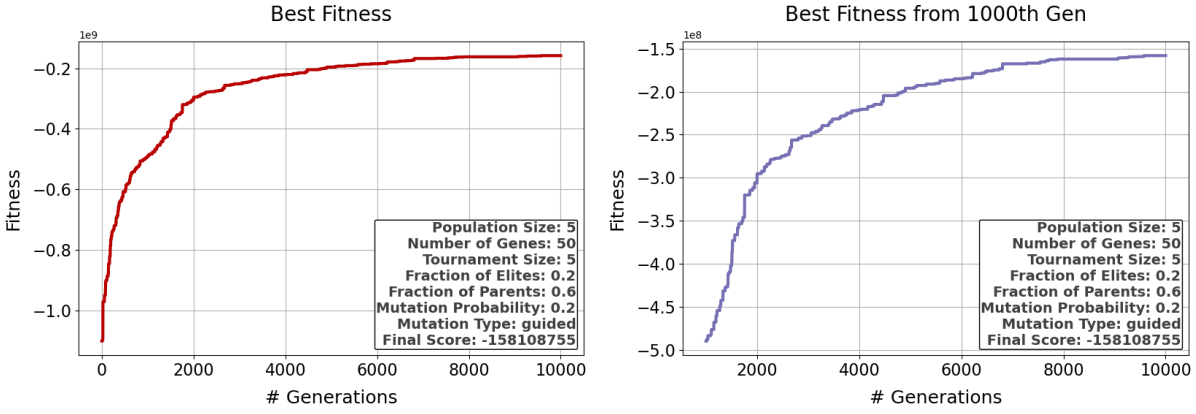
Figure 2: Quantitative evolution of the best individual in the population.

1.2 Number of Individuals

Let us first provide necessary plots for the parameter <num_inds>.

1.2.1 5 Individuals

Figure 3 shows the plots associated with the fitness value of the best individual for 5 individuals.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 3: Fitness curves for 5 individuals.

The Figure 4 shows the evolution of the best individual quantitatively for 5 individuals.

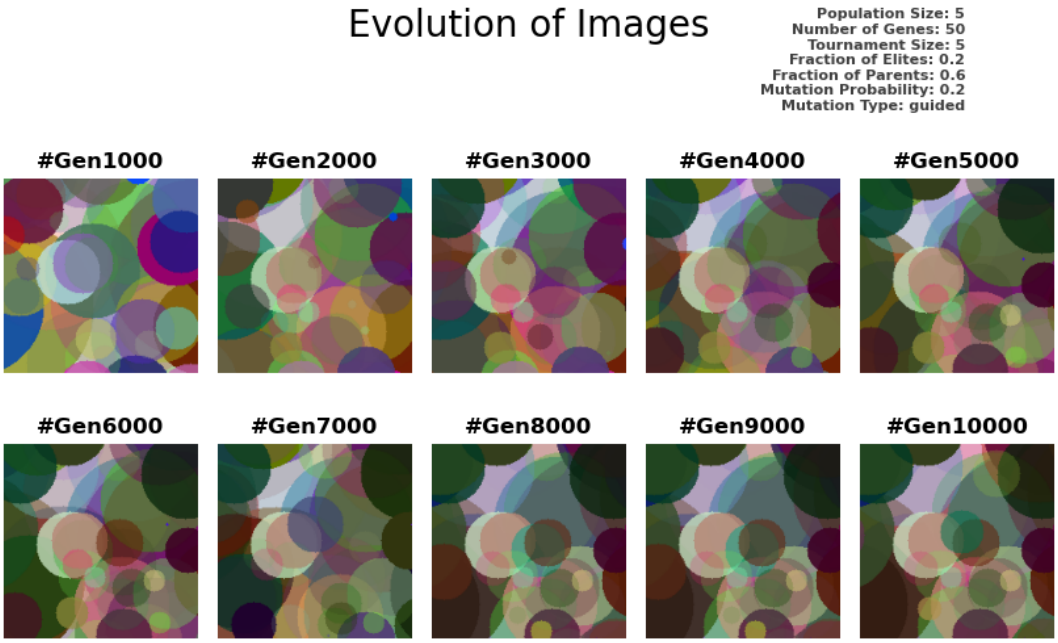
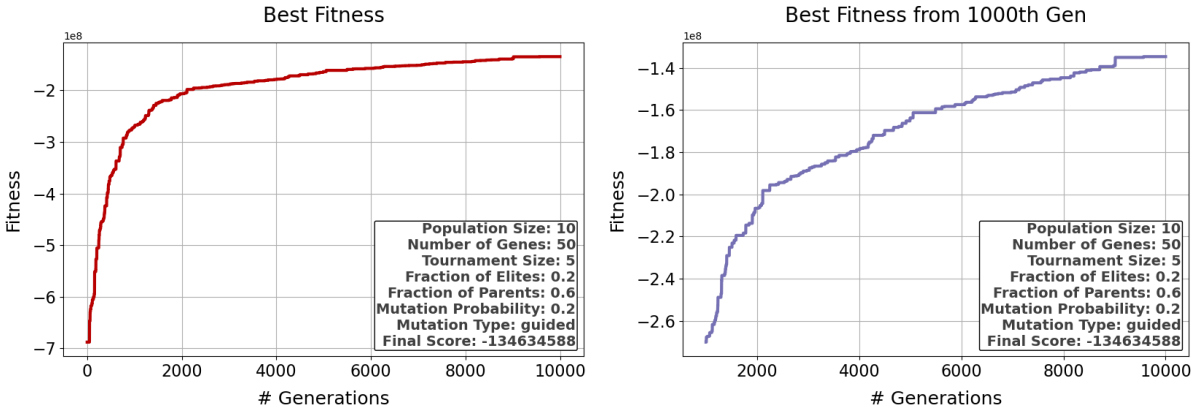


Figure 4: Quantitative evolution of the best individual in the population for 5 individuals.

1.2.2 10 Individuals

Figure 5 illustrates the plots associated with the fitness value of the best individual for 10 individuals.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 5: Fitness curves for 10 individuals.

The Figure 6 shows the evolution of the best individual quantitatively for 10 individuals.

1.2.3 40 Individuals

Figure 7 shows the plots associated with the fitness value of the best individual for 40 individuals.

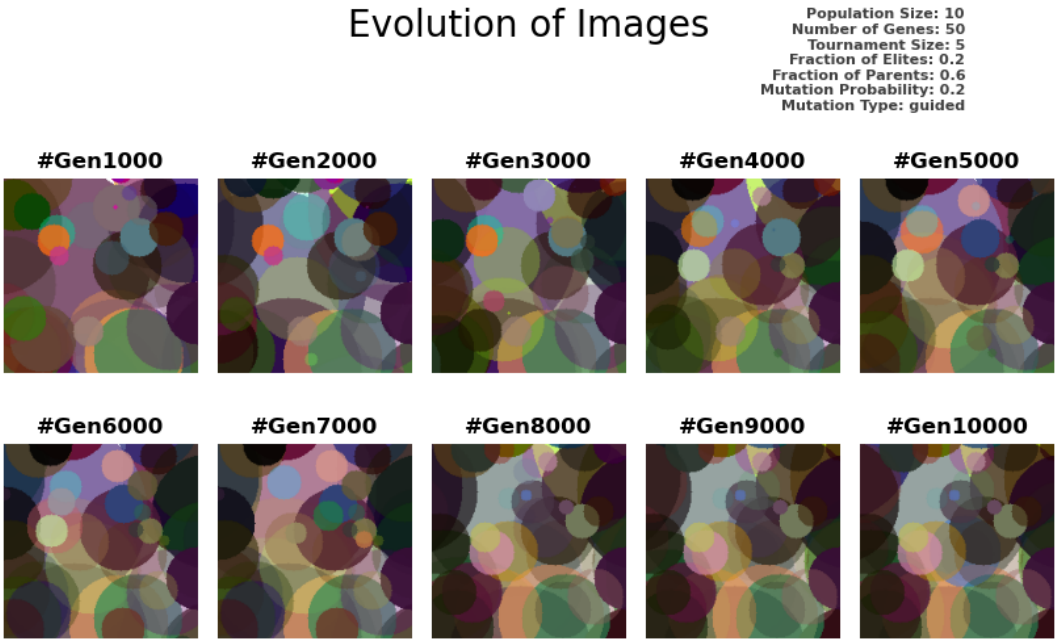
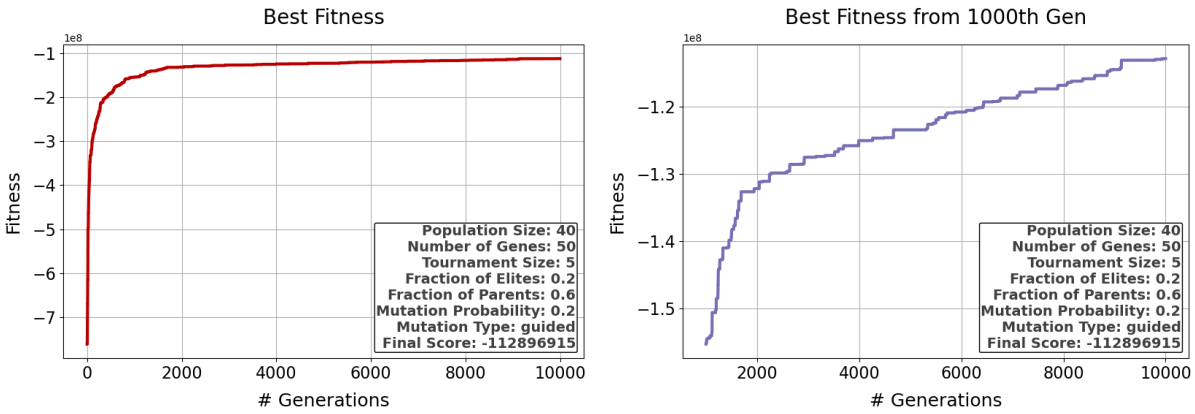


Figure 6: Quantitative evolution of the best individual in the population for 10 individuals.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 7: Fitness curves for 40 individuals.

The Figure 8 indicates the evolution of the best individual quantitatively for 40 individuals.

1.2.4 60 Individuals

Figure 9 shows the plots associated with the fitness value of the best individual for 60 individuals.

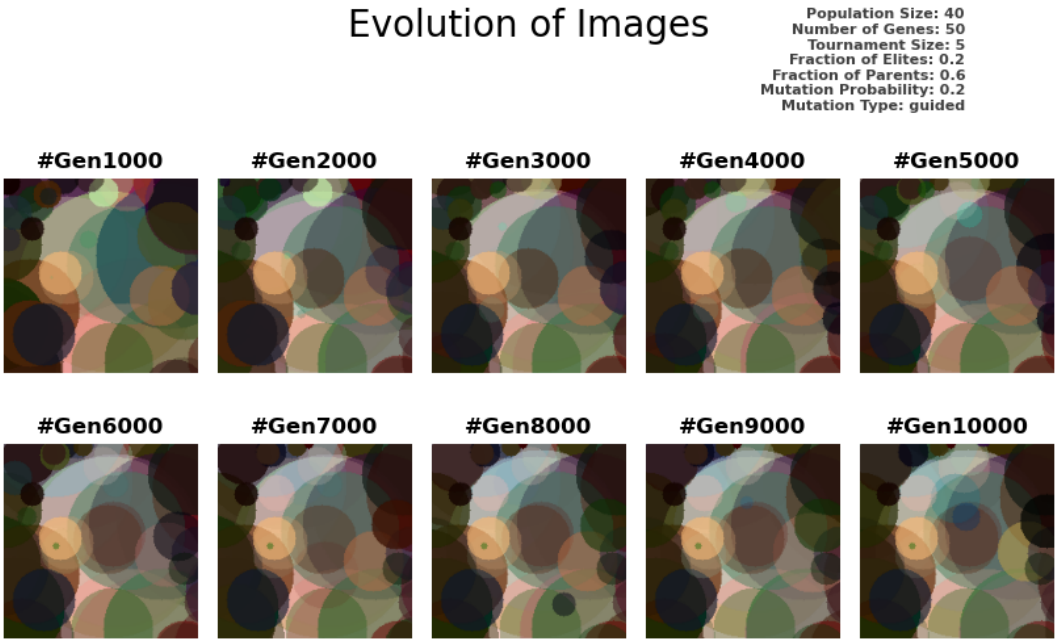
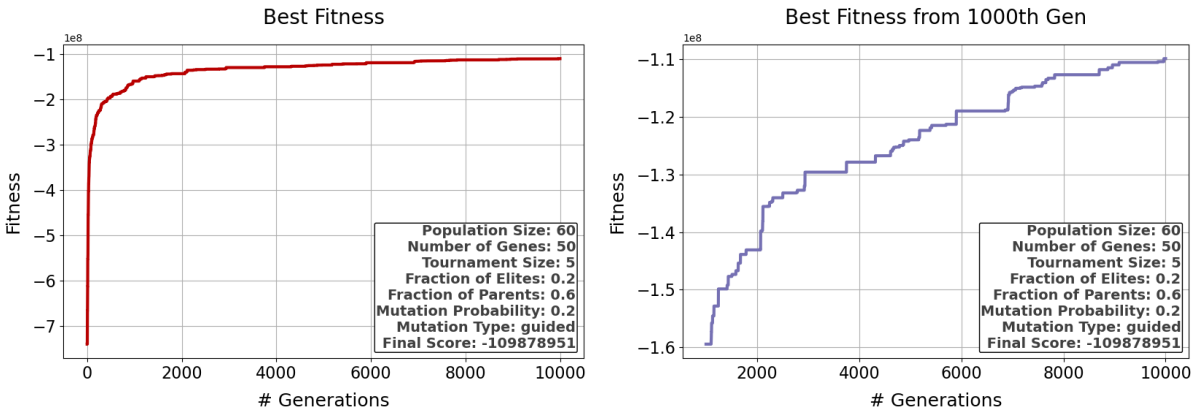


Figure 8: Quantitative evolution of the best individual in the population for 40 individuals.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 9: Fitness curves for 60 individuals.

The Figure 10 shows the evolution of the best individual quantitatively for 60 individuals.

Discussion: The results show that the number of individuals in the population has a significant effect on the performance of the algorithm. It can be deduced that as the number of individuals in the population increases, the algorithm converges to a better solution. This is because the diversity in the population increases as the number of individuals increases.

1.3 Number of Genes

Let us first provide necessary plots for the parameter <num_genes>.

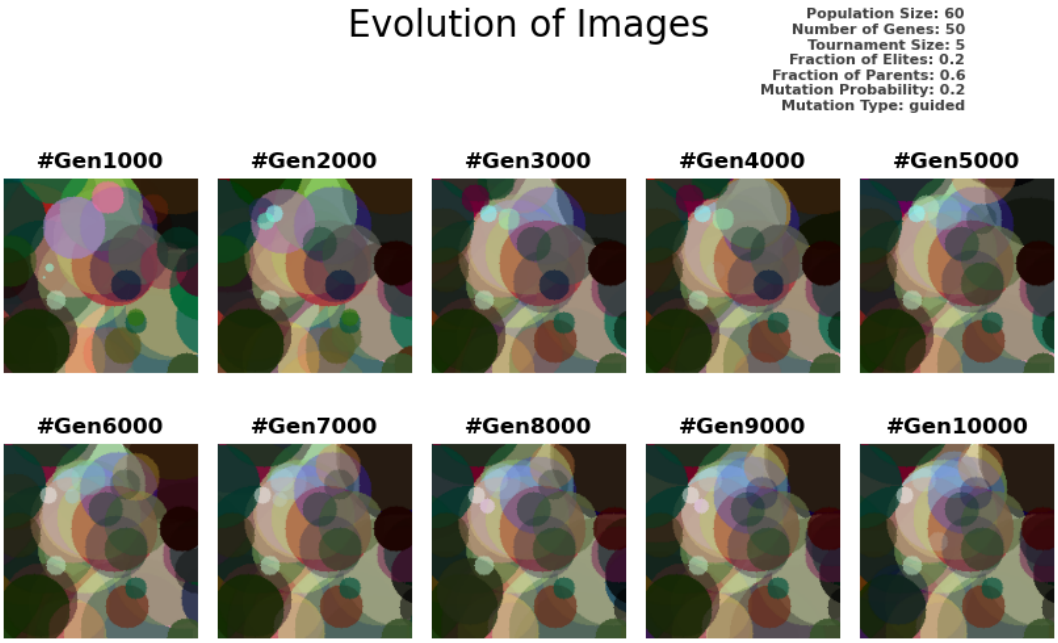
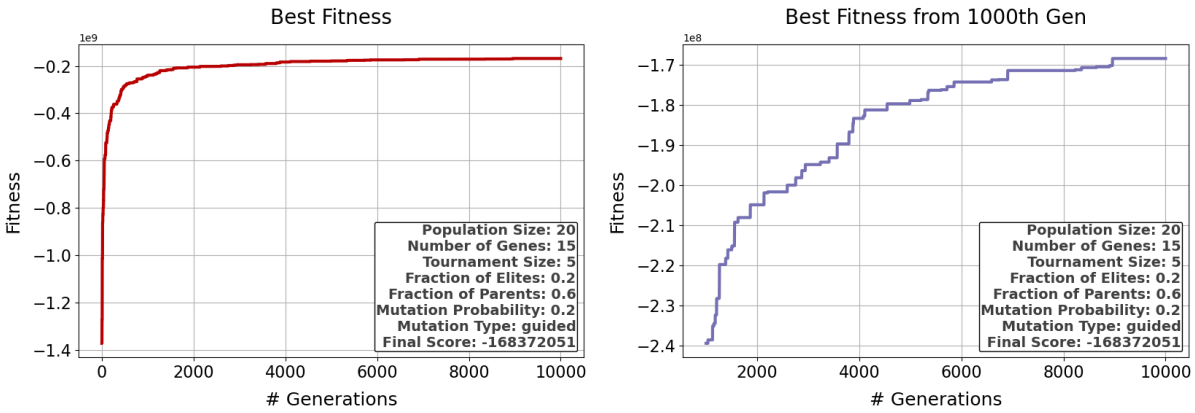


Figure 10: Quantitative evolution of the best individual in the population for 60 individuals.

1.3.1 15 Genes

Figure 11 shows the plots related to the fitness value of the best individual for 15 genes.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 11: Fitness curves for 15 genes.

The Figure 12 illustrates the evolution of the best individual figuratively for 15 genes.

1.3.2 30 Genes

Figure 13 shows the plots related to the fitness value of the best individual for 30 genes.

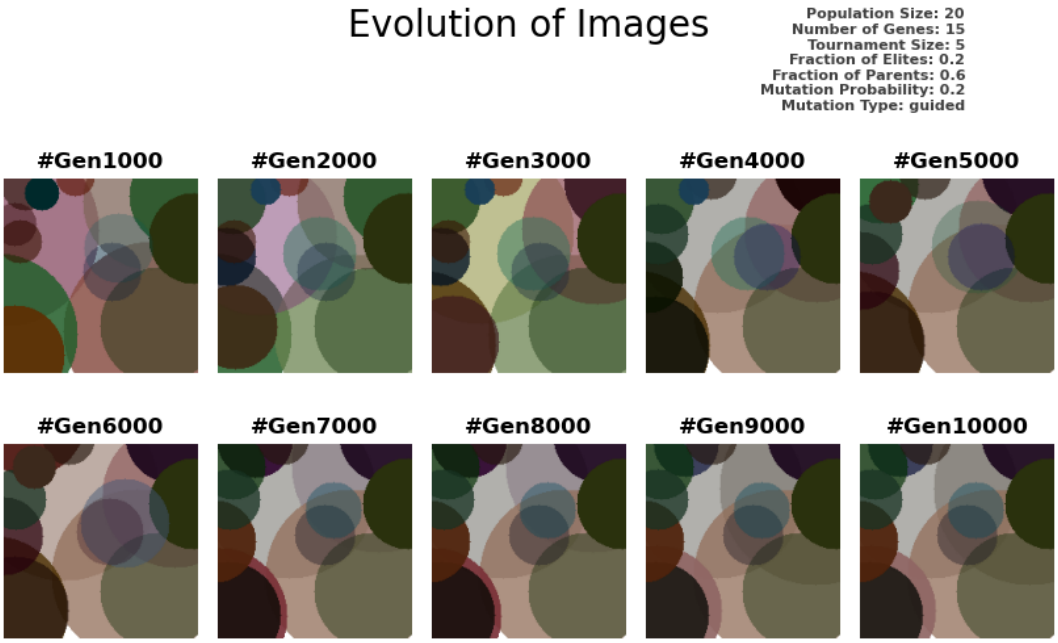
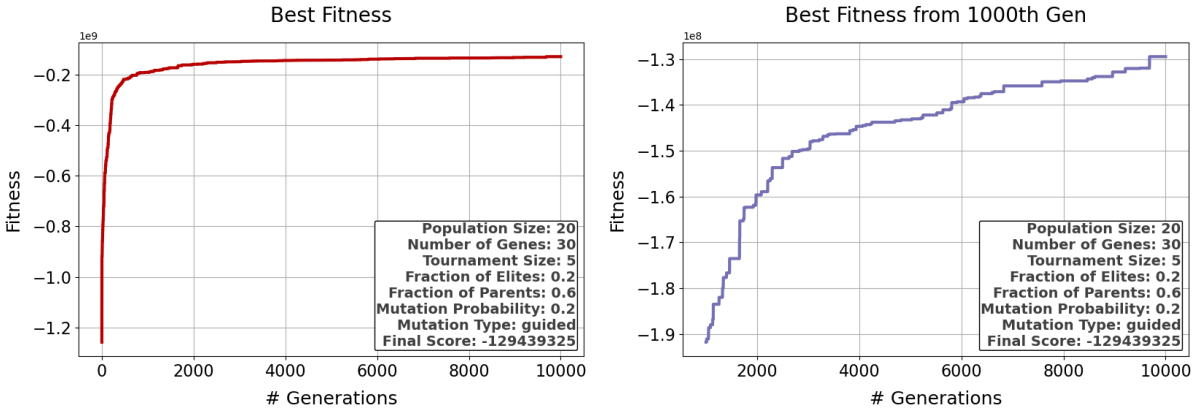


Figure 12: Quantitative evolution of the best individual in the population for 15 genes.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 13: Fitness curves for 30 genes.

The Figure 14 shows the evolution of the best individual figuratively for 30 genes.

1.3.3 80 Genes

Figure 15 shows the plots related to the fitness value of the best individual for 80 genes.

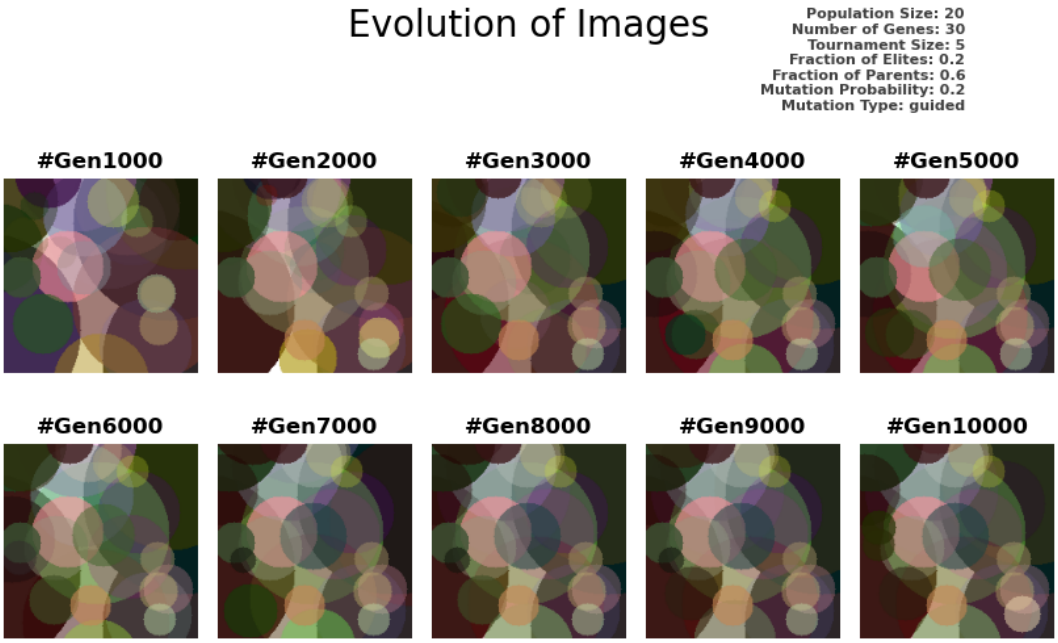
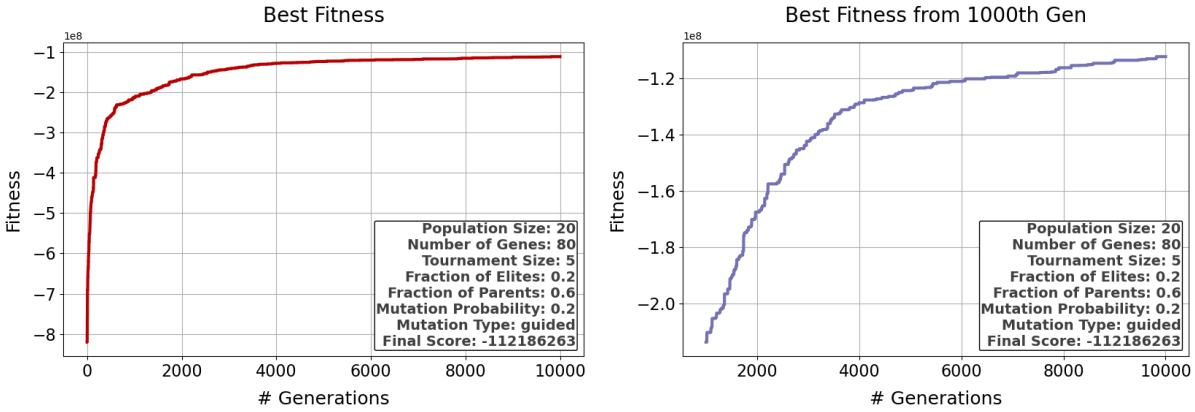


Figure 14: Quantitative evolution of the best individual in the population for 30 genes.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 15: Fitness curves for 80 genes.

The Figure 16 shows the evolution of the best individual figuratively for 80 genes.

1.3.4 120 Genes

Figure 17 shows the plots related to the fitness value of the best individual for 120 genes.

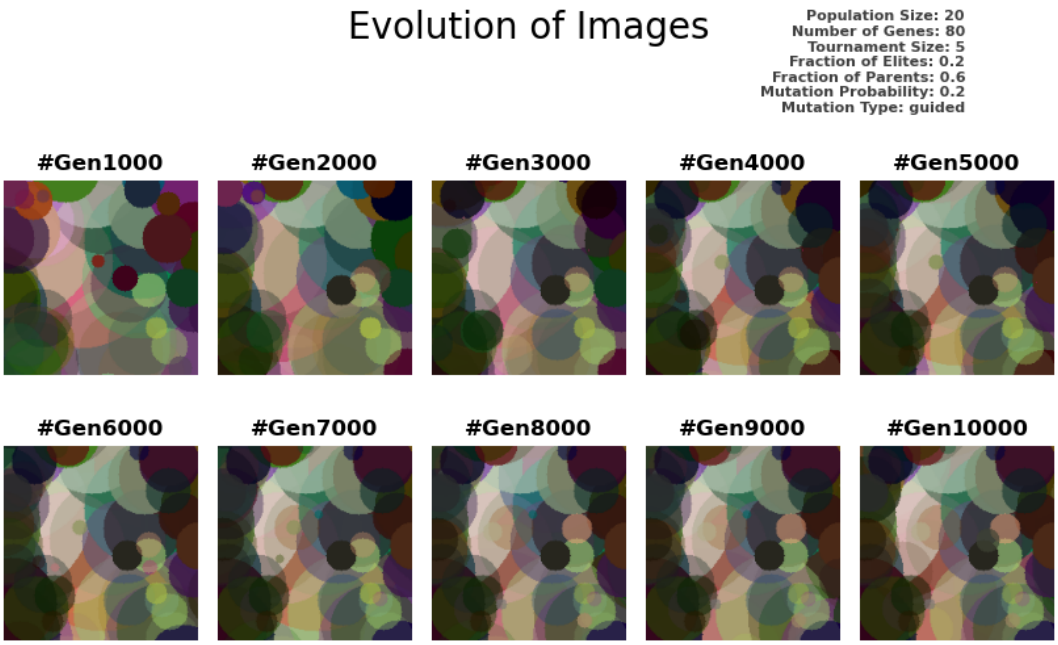
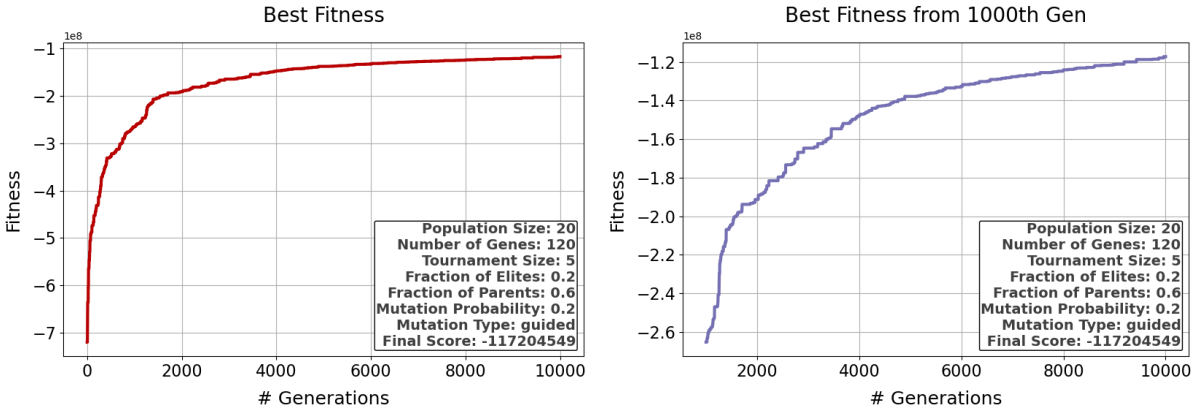


Figure 16: Quantitative evolution of the best individual in the population for 80 genes.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 17: Fitness curves for 120 genes.

The Figure 18 is given for the evolution of the best individual for 120 genes.

Discussion: We see that, as the number of genes in the individual increases, the algorithm converges to a better solution. This is because the number of genes in the individual increases, the search space increases, and the algorithm can explore more possibilities. This was somewhat expected since the number of genes in the individual is directly related to the "paint brushes" we have in this problem. However, from 80 genes to 120 genes, the improvement is not observable 80 genes converged to a better solution. We can interpret that for 120 genes, there need to be more generations to converge to a better solution. So, as the number of genes decreases, the number of generations needed to converge is also decreased. Given we fixed the number of generations, the results are natural in this sense.

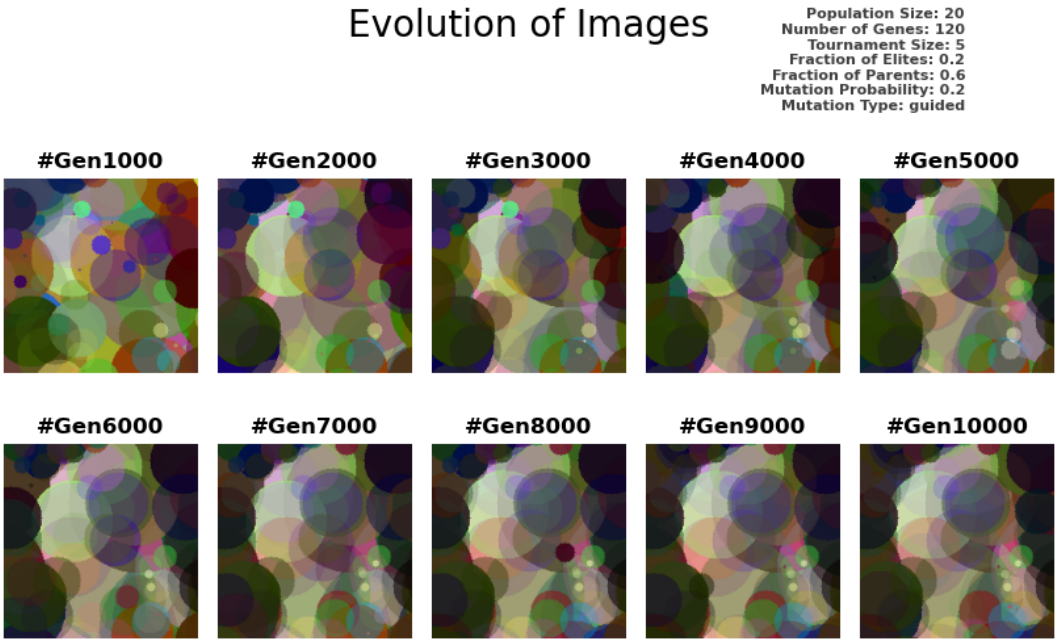


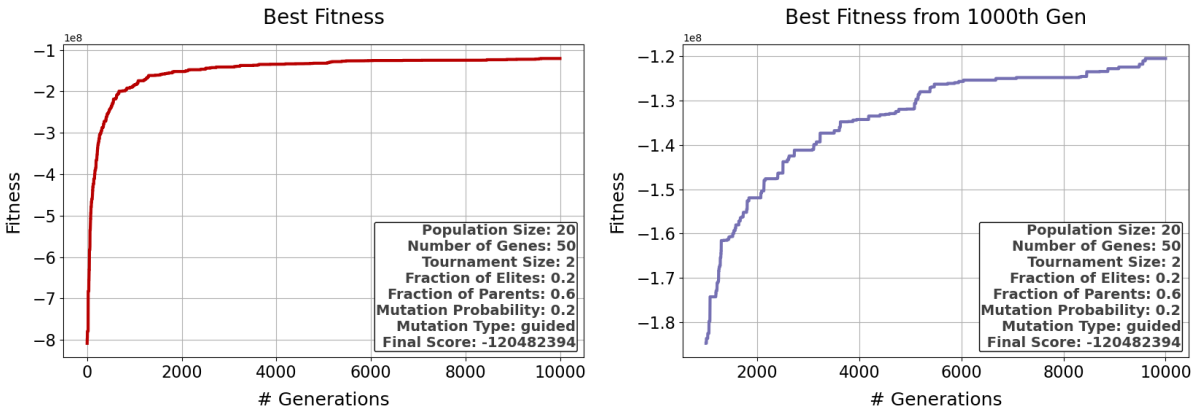
Figure 18: Quantitative evolution of the best individual in the population for 120 genes.

1.4 Tournament Size

Let us first provide necessary plots for the parameter <tm_size>.

1.4.1 2 Tournament Size

Figure 19 shows the plots related to the fitness value of the best individual for 2 tournament size.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

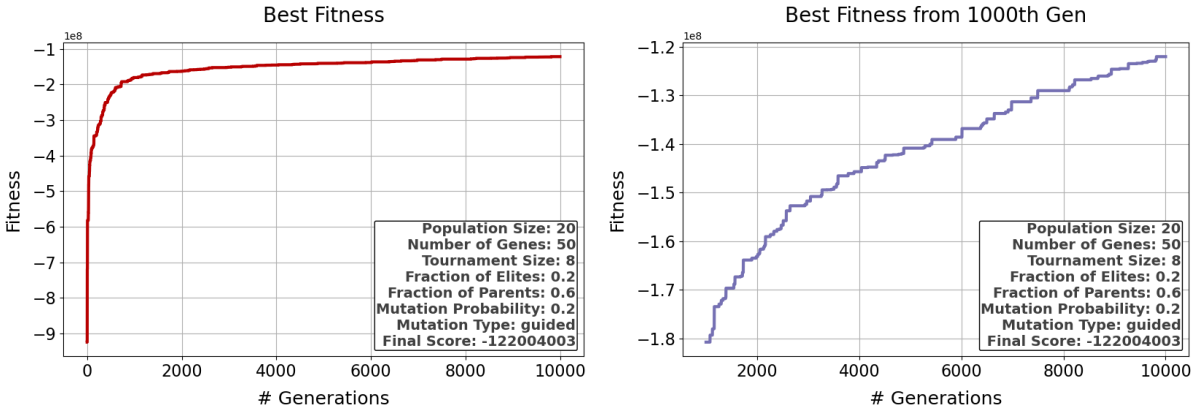
Figure 19: Fitness curves for 2 tournament size.

The Figure 20 shows the evolution of the best individual figuratively for 2 tournament size.

1.4.2 8 Tournament Size

Figure 21 shows the plots related to the fitness value of the best individual for 8 tournament size.

Figure 20: Quantitative evolution of the best individual in the population for 2 tournament size.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 21: Fitness curves for 8 tournament size.

The Figure 22 shows the evolution of the best individual figuratively for 8 tournament size.

1.4.3 16 Tournament Size

Figure 23 shows the plots related to the fitness value of the best individual for 16 tournament size.

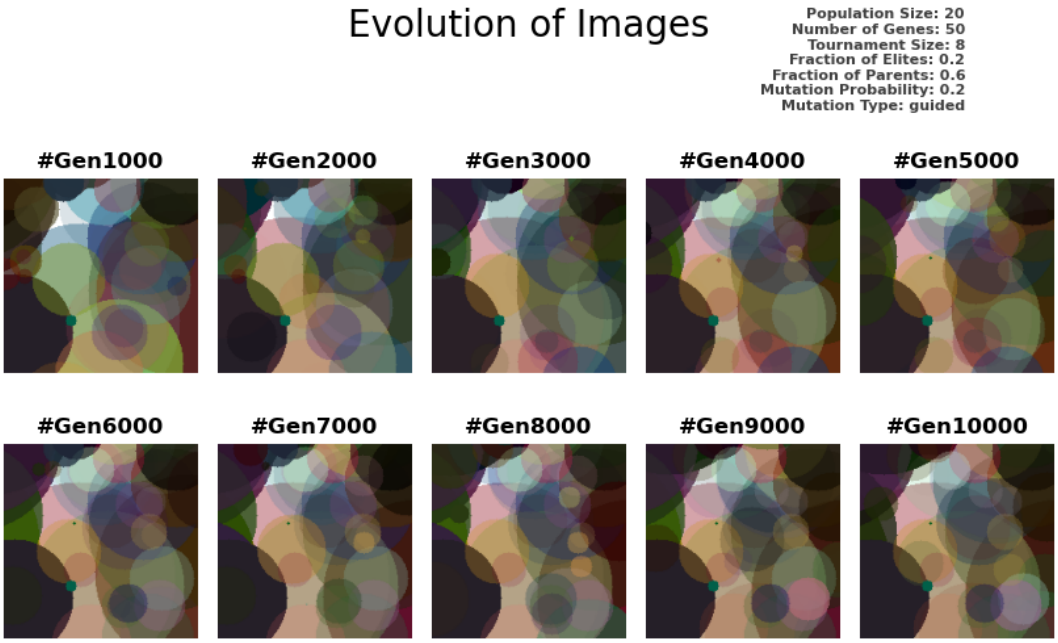
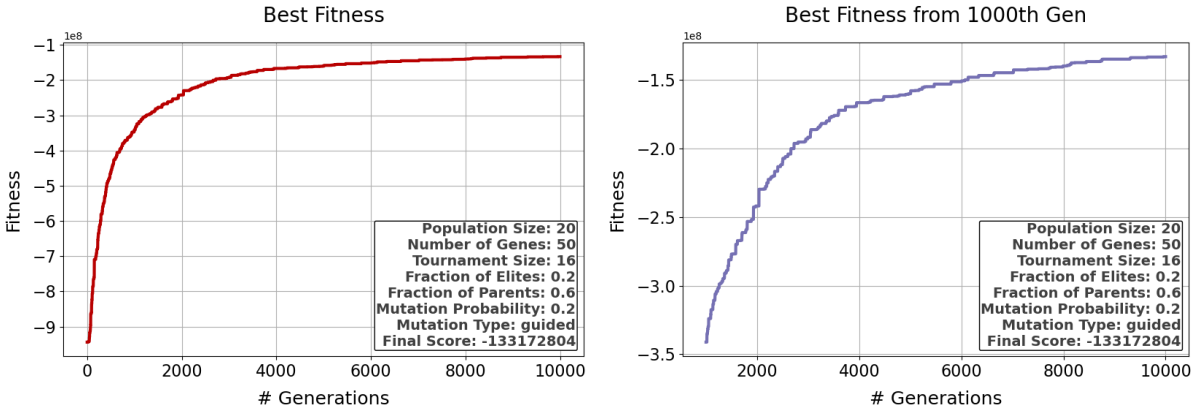


Figure 22: Quantitative evolution of the best individual in the population for 8 tournament size.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 23: Fitness curves for 16 tournament size.

The Figure 24 shows the evolution of the best individual figuratively for 16 tournament size.

Discussion: From the results, we see that the best tournament size selection for this parameter space was 5. This means there is no linear relationship between the tournament size and the performance of the algorithm. As the tournament size increases too much, the diversity decreases. It can be said that tournament size is a hyperparameter that needs to be tuned for the specific problem. If we change the other parameters, such as the number of individuals, number of genes, etc., the best tournament size would change.

1.5 Number of Individuals Advancing to Next Generation

Let us first provide necessary plots for the parameter <frac_elites>.

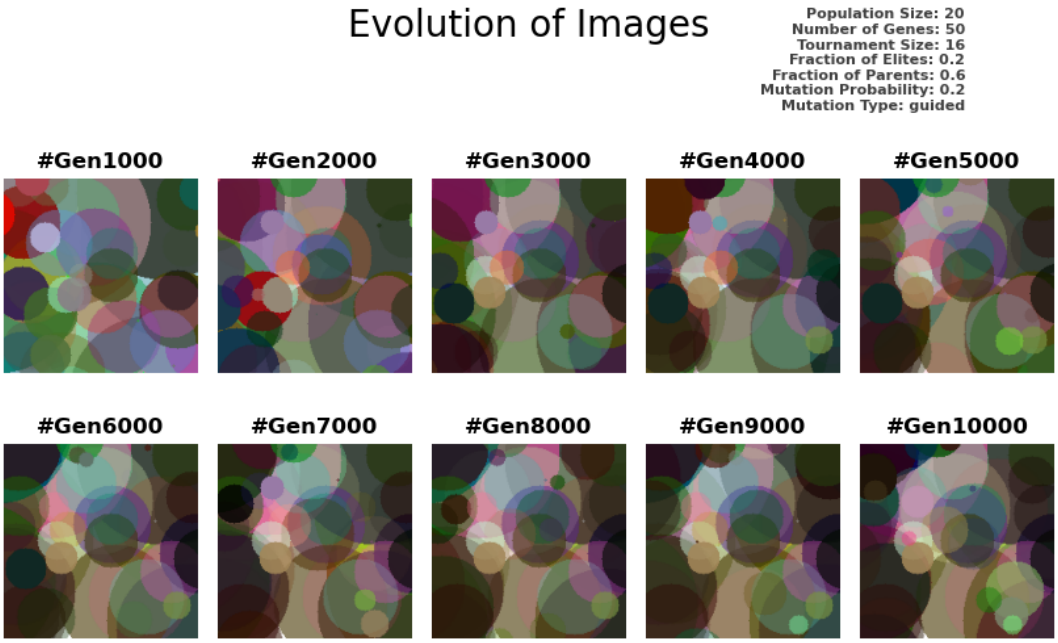
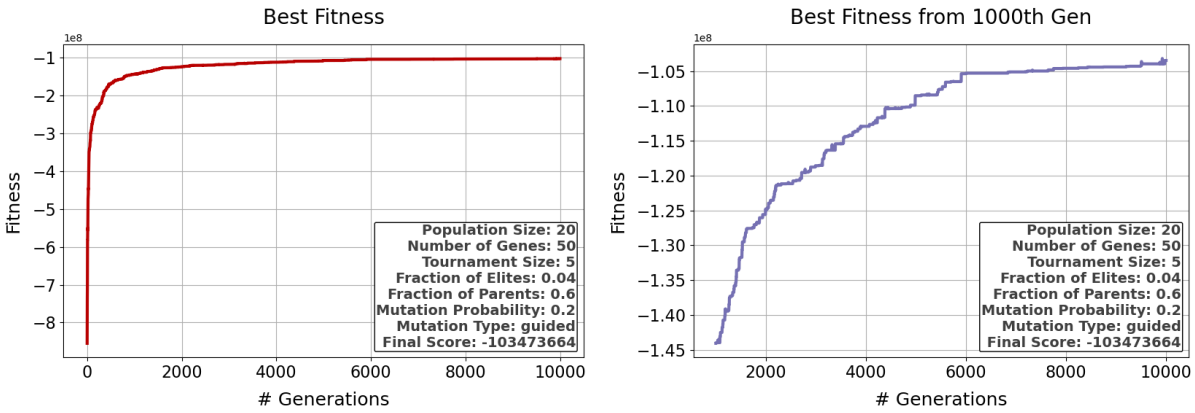


Figure 24: Quantitative evolution of the best individual in the population for 16 tournament size.

1.5.1 0.04 Fraction of Elites

Figure 25 shows the plots related to the fitness value of the best individual for 0.04 fraction of elites.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 25: Fitness curves for 0.04 fraction of elites.

The Figure 26 shows the evolution of the best individual figuratively for 0.04 fraction of elites.

1.5.2 0.35 Fraction of Elites

Figure 27 shows the plots related to the fitness value of the best individual for 0.35 fraction of elites.

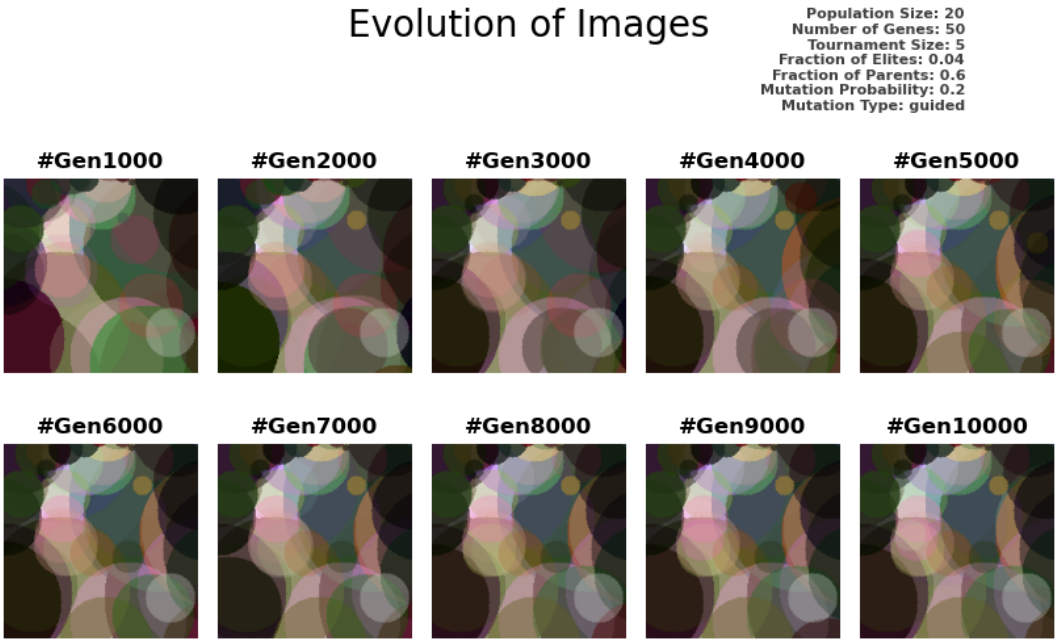
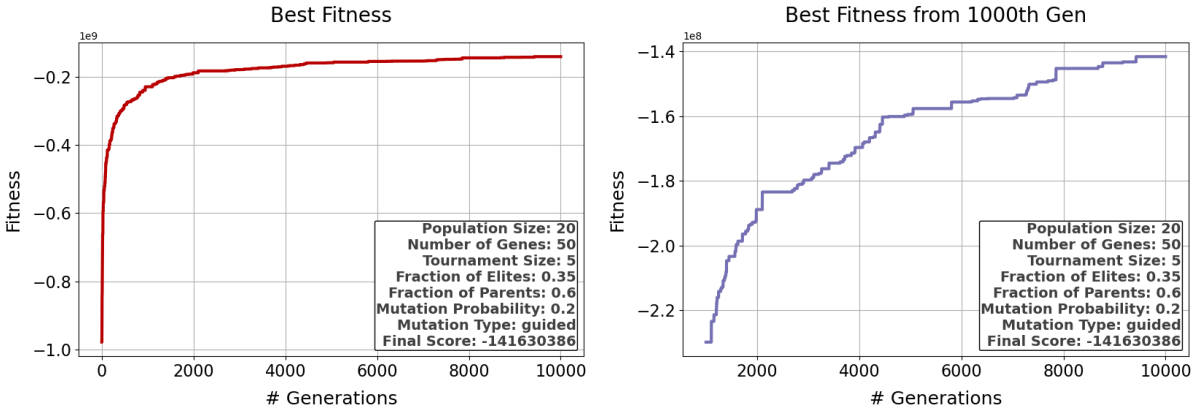


Figure 26: Quantitative evolution of the best individual in the population for 0.04 fraction of elites.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 27: Fitness curves for 0.35 fraction of elites.

The Figure 28 shows the evolution of the best individual figuratively for 0.35 fraction of elites.

Discussion: The results indicate that the best selection for the fraction of elites is 0.04. We may interpret this as the algorithm needs more diversity in the population to converge to a better solution. On the other hand, this implies that as the number of elites decreases, the parents selected are shifted to a better group of individuals. So, it is hard to tell if the number of elites is decoupled from the other parameters. If we select parents first from the best individuals and then from the rest of the population, the best fraction of elites would change.

1.6 Number of Parents Used In Crossover

Let us first provide necessary plots for the parameter <frac_parents>.

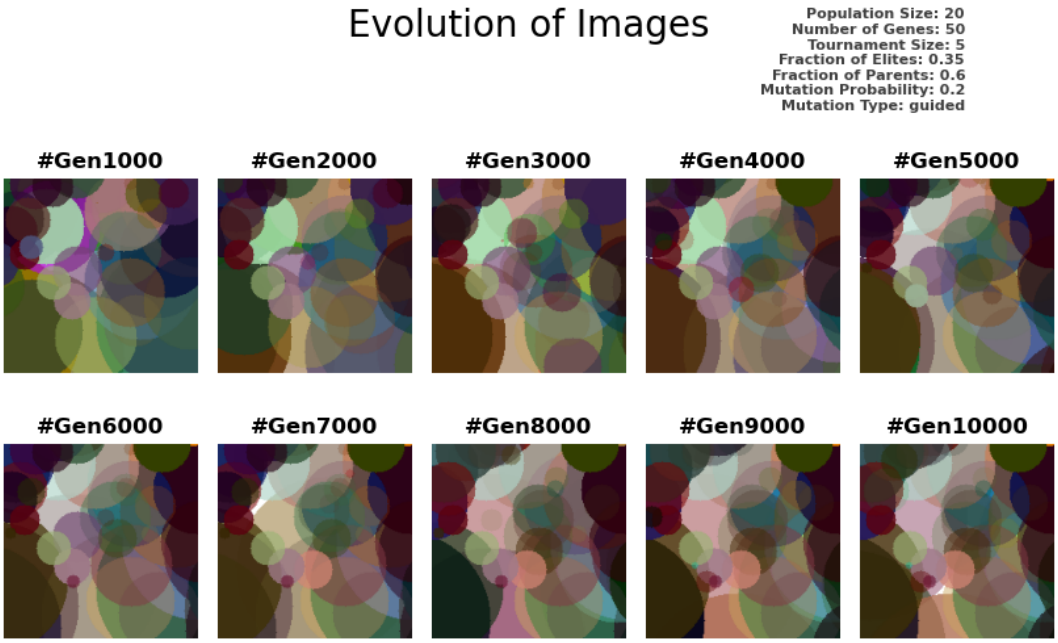
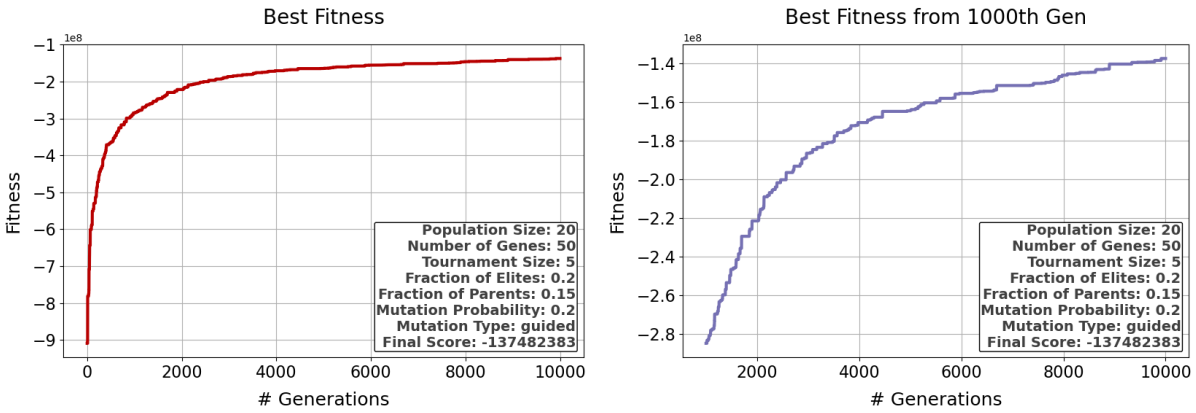


Figure 28: Quantitative evolution of the best individual in the population for 0.35 fraction of elites.

1.6.1 0.15 Fraction of Parents

The plots about to the fitness metric of the best individual for 0.15 fraction of parents in Figure 29.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 29: Fitness curves for 0.15 fraction of parents.

The evolution of the best individual for 0.15 fraction of parents is given in Figure 30.

1.6.2 0.3 Fraction of Parents

The plots about to the fitness metric of the best individual for 0.3 fraction of parents in Figure 31.

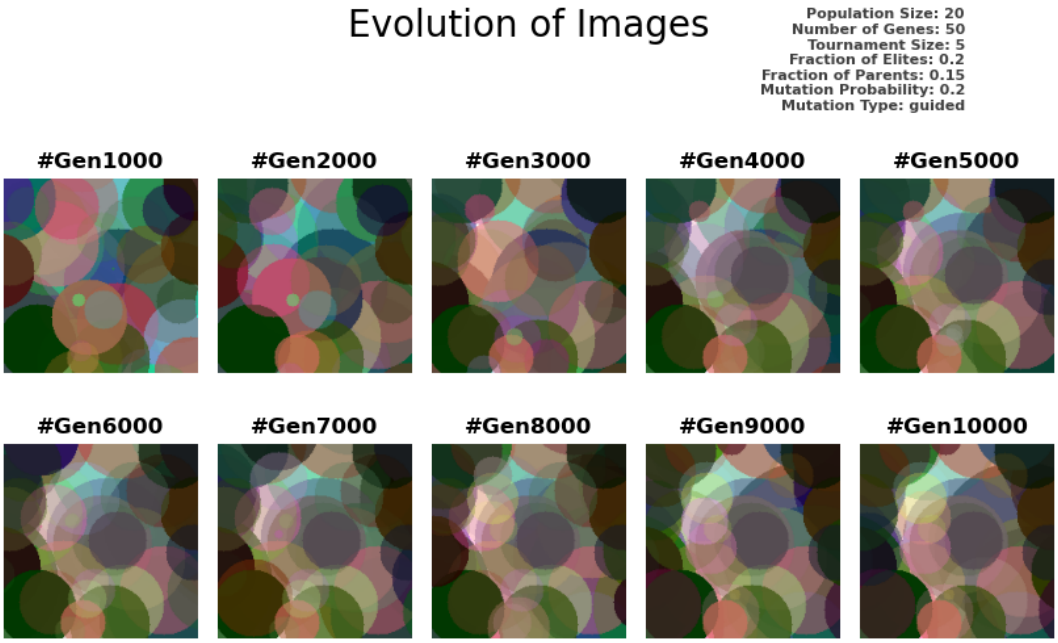
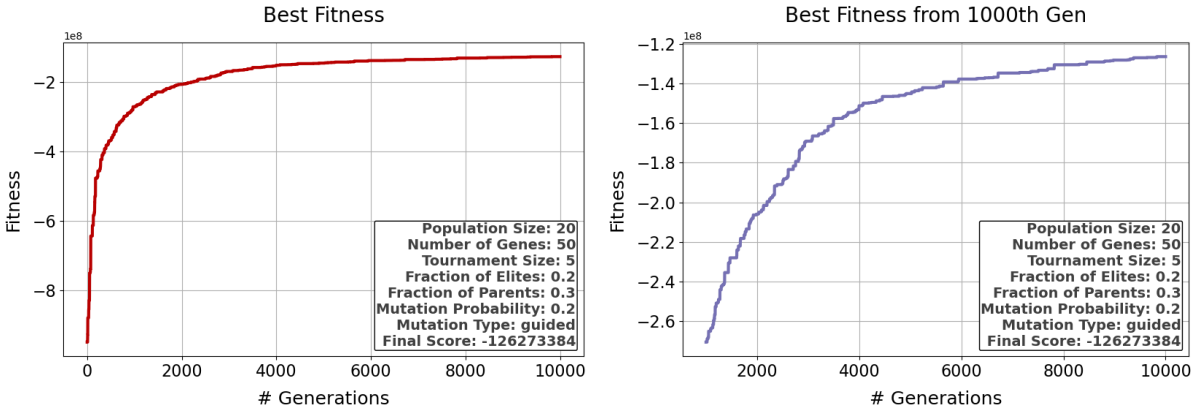


Figure 30: Quantitative evolution of the best individual in the population for 0.15 fraction of parents.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 31: Fitness curves for 0.3 fraction of parents.

The evolution of the best individual for 0.3 fraction of parents is given in Figure 32.

1.6.3 0.75 Fraction of Parents

The plots about to the fitness metric of the best individual for 0.75 fraction of parents in Figure 33.

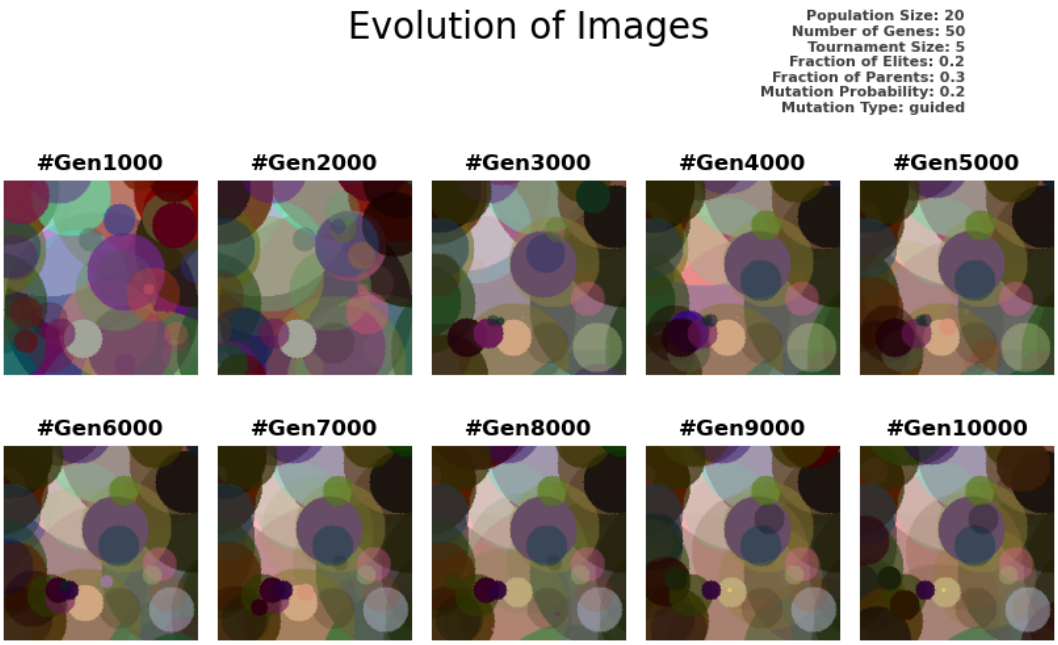
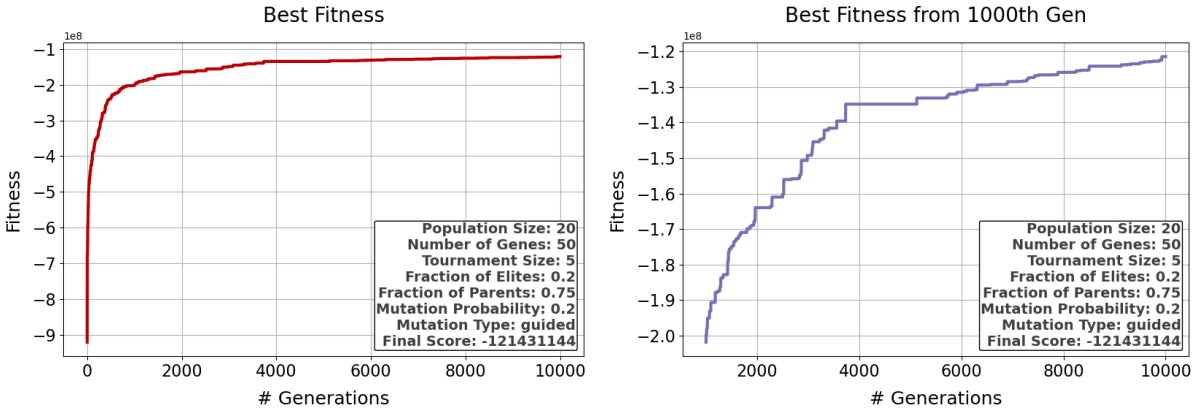


Figure 32: Quantitative evolution of the best individual in the population for 0.3 fraction of parents.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 33: Fitness curves for 0.75 fraction of parents.

The evolution of the best individual for 0.75 fraction of parents is given in Figure 34.

Discussion: The results show that the best selection for the fraction of parents is 0.6. This means that the algorithm needs to select parents from a wide range of individuals to converge on a better solution. This is because the diversity in the population is important for the algorithm to explore more possibilities. If the algorithm selects parents from a small group of individuals, the algorithm may converge to a local minimum. On the other hand, in the case of 0.75 percent of parents, we see that increasing the percentage of parents did not help. We may interpret this result as the algorithm needs to select parents from a wide range of individuals but not from the whole population.

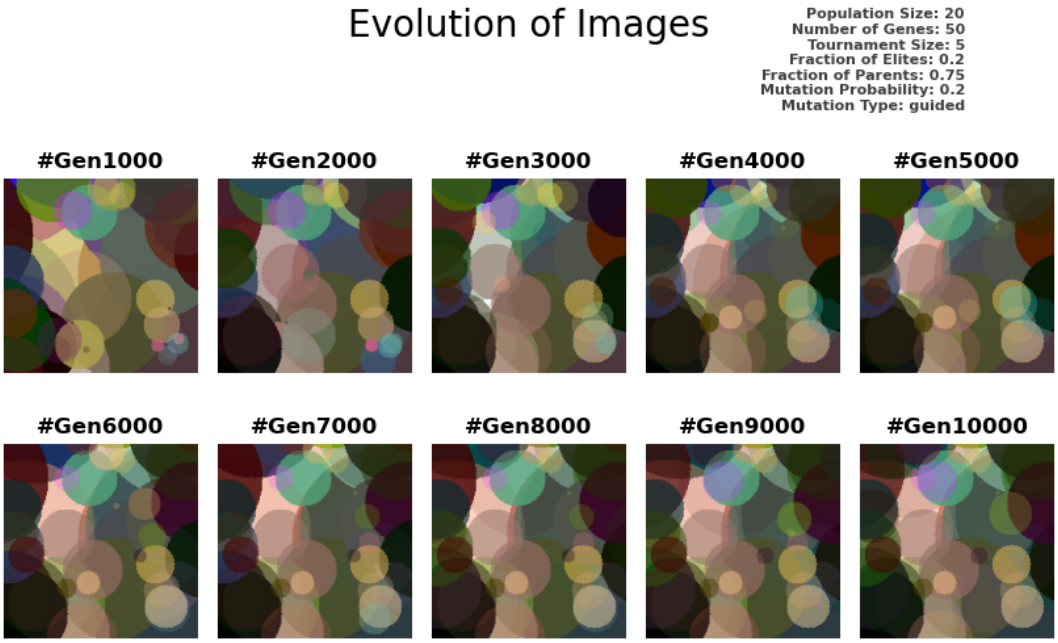


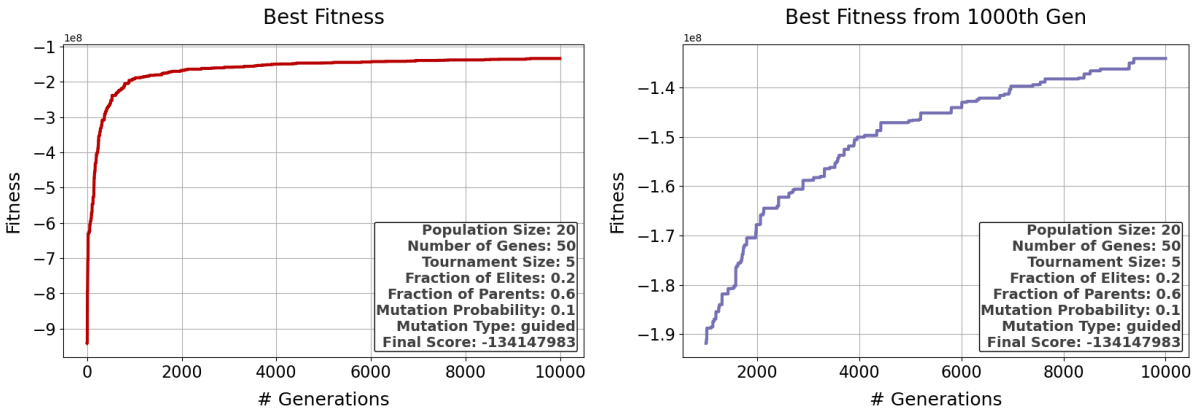
Figure 34: Quantitative evolution of the best individual in the population for 0.75 fraction of parents.

1.7 Mutation Probability

Let us first provide necessary plots for the parameter <mutation_prob>.

1.7.1 0.1 Mutation Probability

The plots about to the fitness metric of the best individual for 0.1 mutation probability in Figure 35.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 35: Fitness curves for 0.1 mutation probability.

The evolution of the best individual for 0.1 mutation probability is given in Figure 36.

1.7.2 0.4 Mutation Probability

The plots about to the fitness metric of the best individual for 0.4 mutation probability in Figure 37.

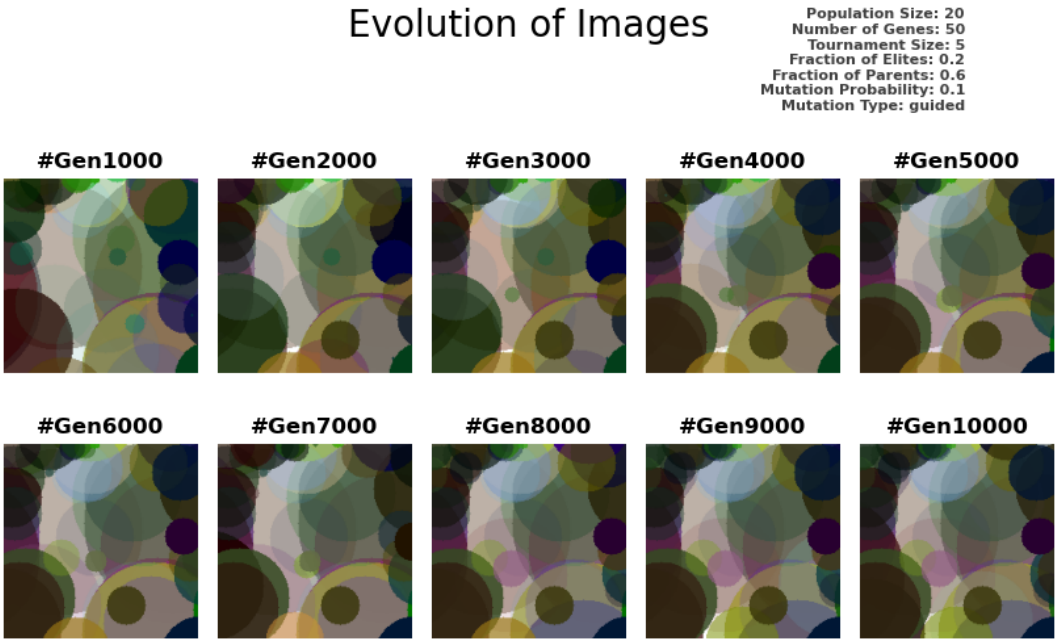
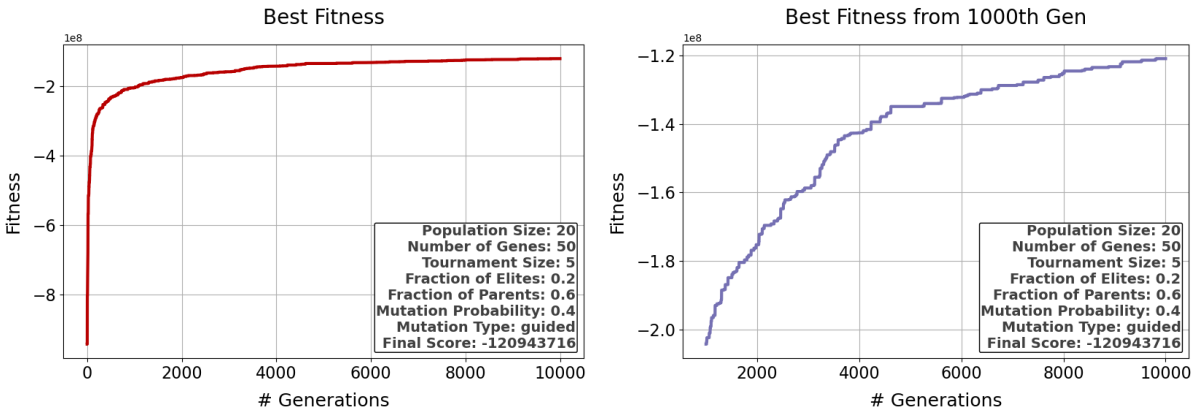


Figure 36: Quantitative evolution of the best individual in the population for 0.1 mutation probability.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 37: Fitness curves for 0.4 mutation probability.

The evolution of the best individual for 0.4 mutation probability is given in Figure 38.

1.7.3 0.75 Mutation Probability

The plots about to the fitness metric of the best individual for 0.75 mutation probability in Figure 39.

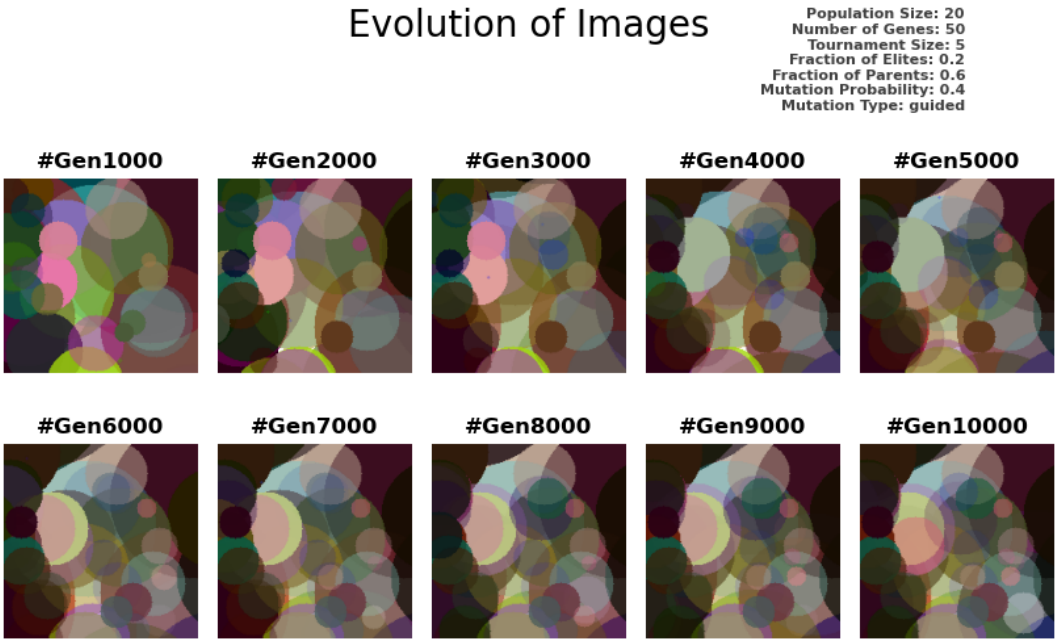
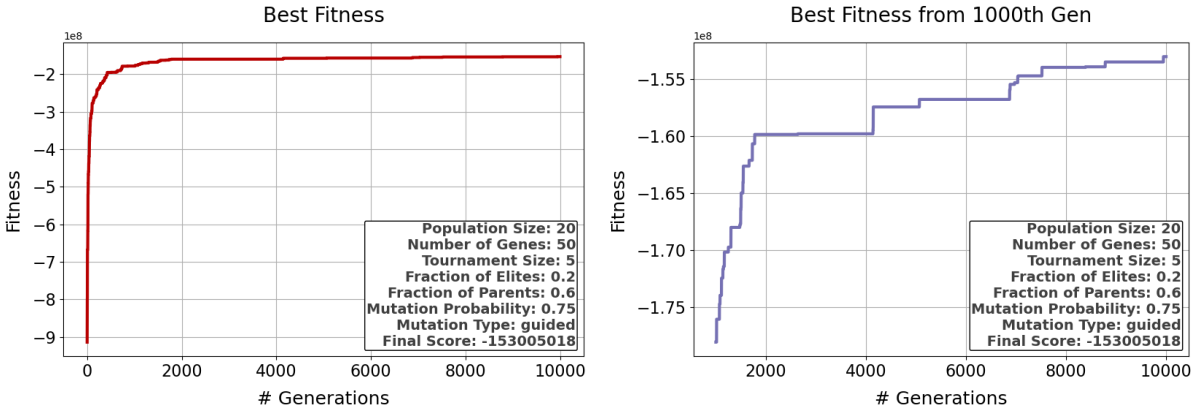


Figure 38: Quantitative evolution of the best individual in the population for 0.4 mutation probability.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 39: Fitness curves for 0.75 mutation probability.

The evolution of the best individual for 0.75 mutation probability is given in Figure 40.

Discussion: The results show that the best selection for the mutation probability is 0.2. This tells us two things. First, if the mutation probability is too low, the algorithm may not be able to explore the search space effectively. Second, if the mutation probability is too high, the algorithm may not be able to converge to a better solution. This is because if the mutation probability is too high, the algorithm may not be able to preserve the good genes in the population. So, the mutation probability is a hyperparameter that needs to be tuned for the specific problem.

1.8 Mutation Type

Let us first provide necessary plots for the parameter <mutation_type>.

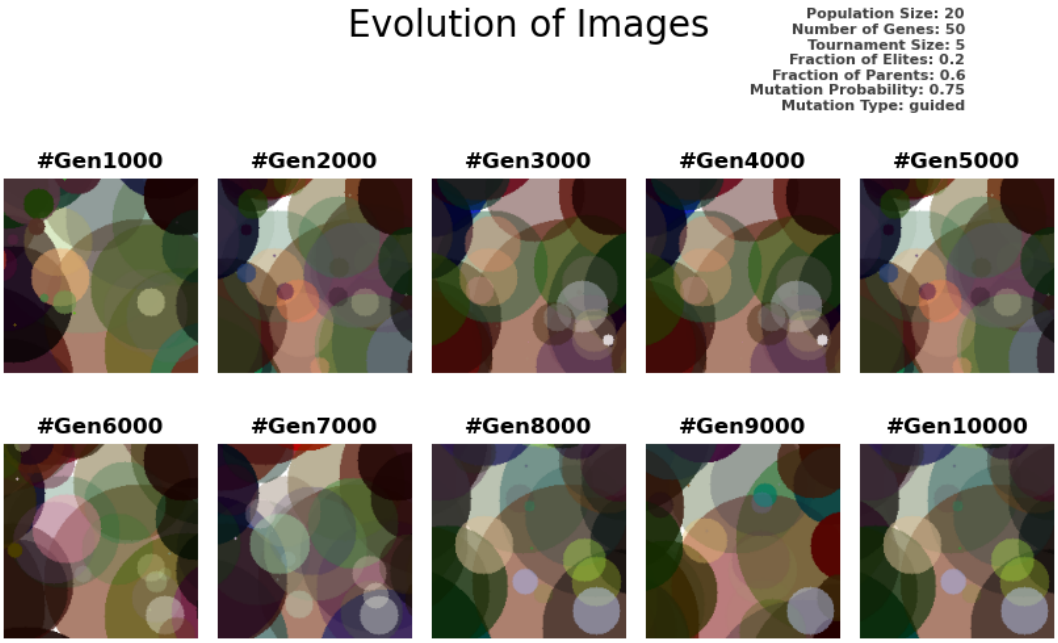
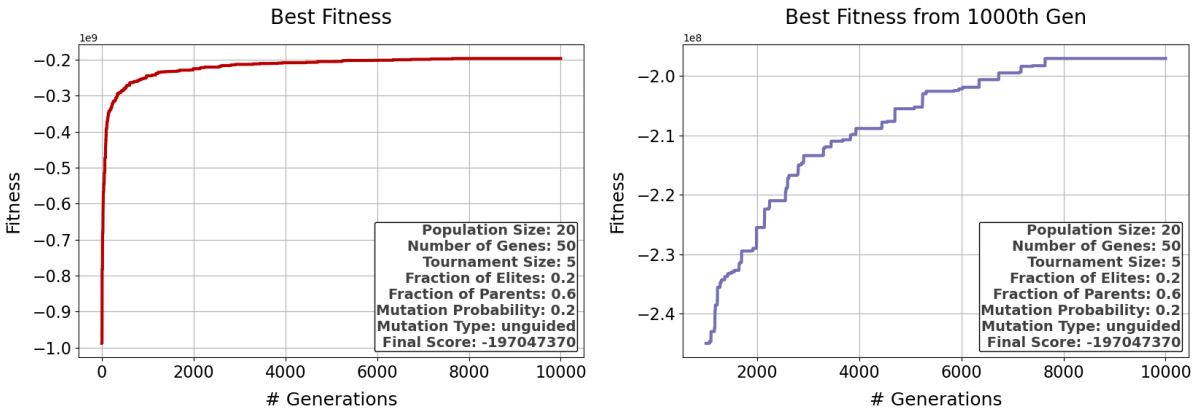


Figure 40: Quantitative evolution of the best individual in the population for 0.75 mutation probability.

1.8.1 Unguided Mutation

The plots about to the fitness metric of the best individual for unguided mutation in Figure 41.



(a) From first generation to 10000th generation. (b) From 1000th generation to 10000th generation.

Figure 41: Fitness curves for unguided mutation.

The evolution of the best individual for unguided mutation is given in Figure 42.

Discussion: It is quite obvious that if the mutation is unguided, the algorithm cannot converge to a better solution. This is because the mutation become reinitialization of the gene. So, the mutation loses its evolutionary meaning.

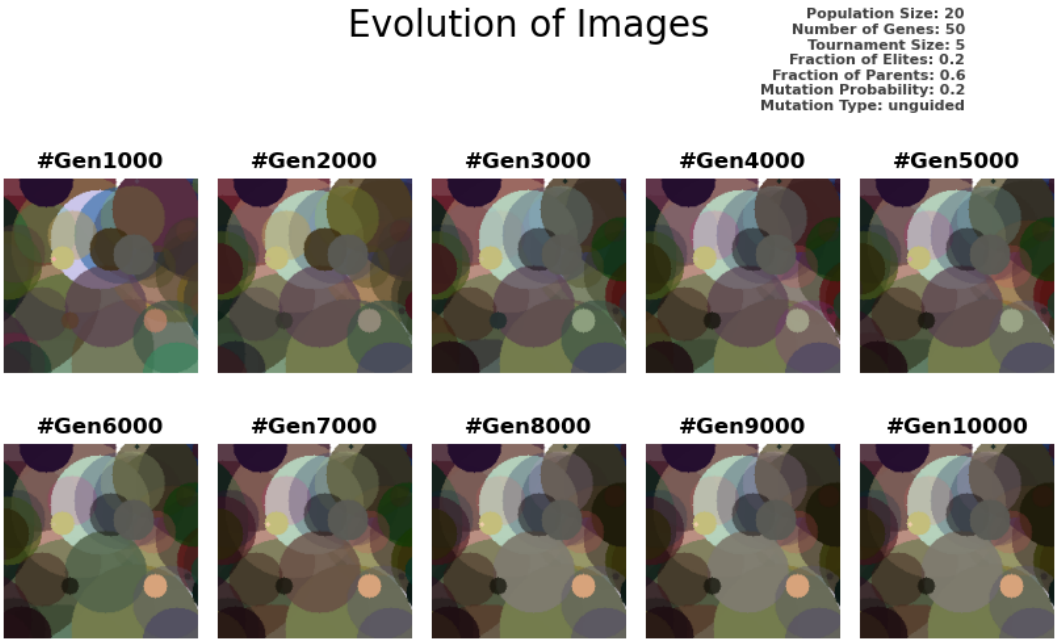


Figure 42: Quantitative evolution of the best individual in the population for unguided mutation.

2 Discussion - Suggestions

There are three suggestions, based on our observations and results, that can be made to improve the performance of the algorithm. Even though having only one trial for each suggestion is not enough to make a definitive conclusion, the results are promising. The suggestions are as follows:

2.1 Probabilistic Mutation Probability

The first suggestion is that the mutation probability can be made stochastic. This means that the mutation probability can be changed in each generation. This can be done by using a probability distribution. For example, the mutation probability can be sampled from a beta distribution in each generation. This can help the algorithm to explore the search space more effectively. Figure 44 shows the fitness curves for the best individual when the mutation probability is stochastic. An example pdf for beta distribution is shown in Figure 43.

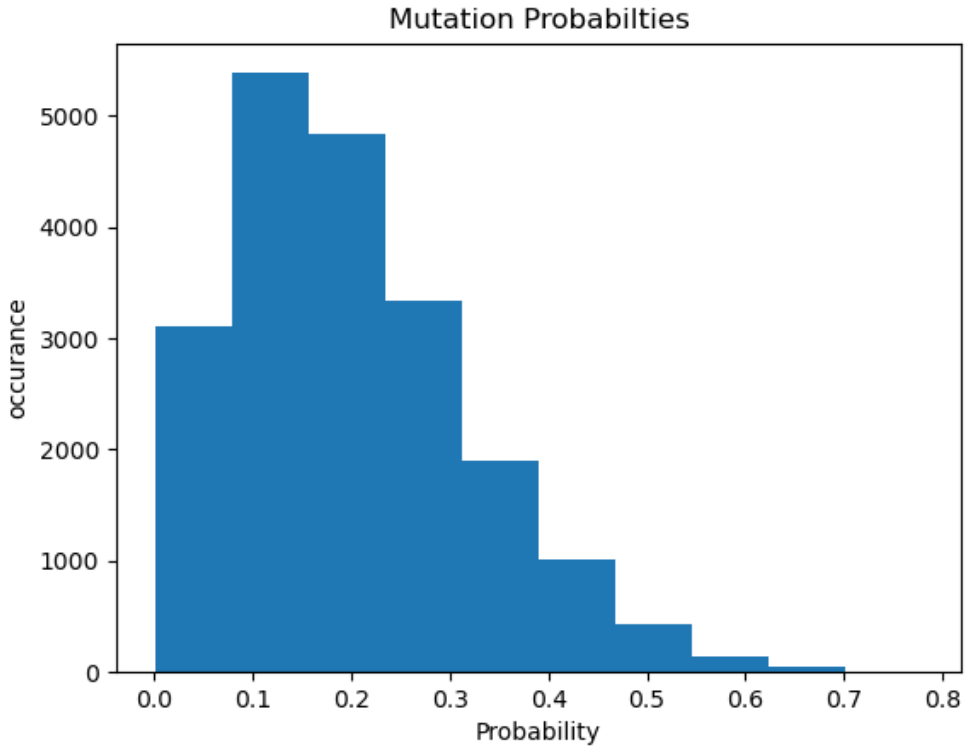
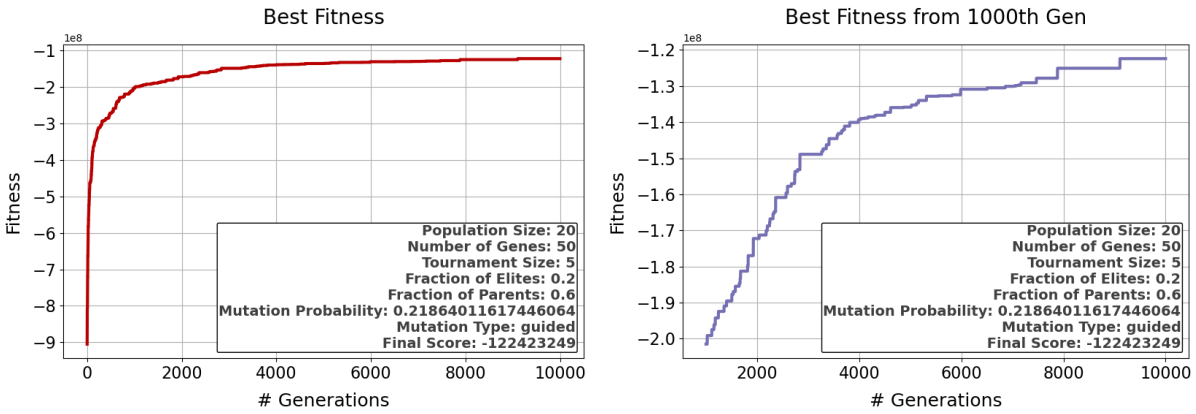


Figure 43: A sample histogram for beta distributed samples when value is around 0.2.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 44: Fitness curves for stochastic mutation probability.

The evolution of the best individual for stochastic mutation probability is given in Figure 45.

Even though the results are not conclusive, the stochastic mutation probability can be a good idea to explore the search space more effectively by deviating around the mutation probability in each generation. This would increase the randomness in the system, but almost everything is random in the evolution algorithm.

2.2 Mutation Probability Scheduling

Similarly, the mutation probability can be made dynamic. This means that the mutation probability can be changed based on the number of generations. This can help the algorithm to escape from local

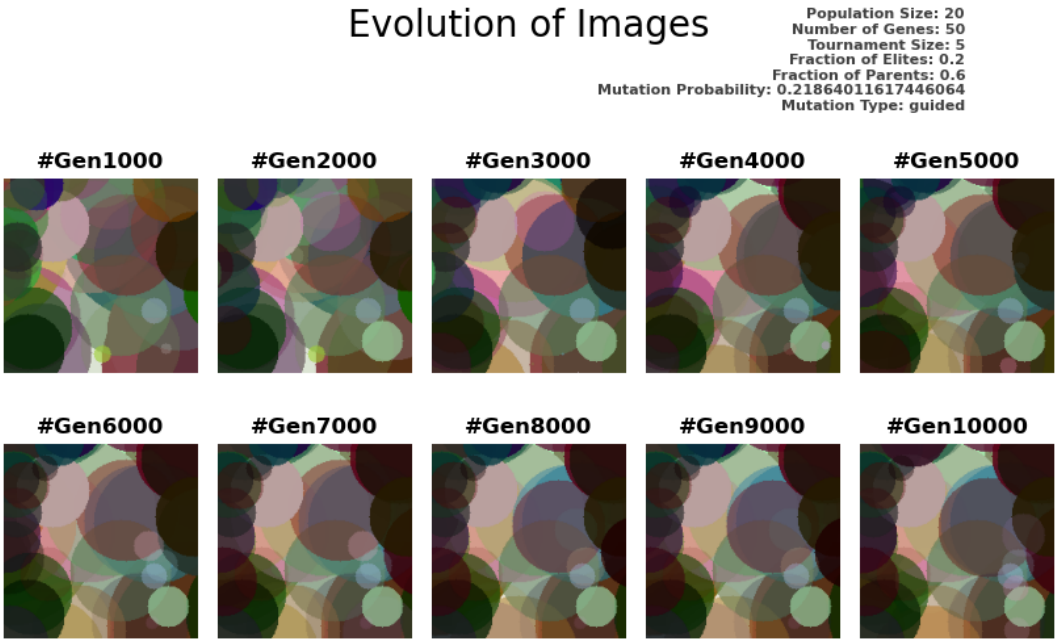
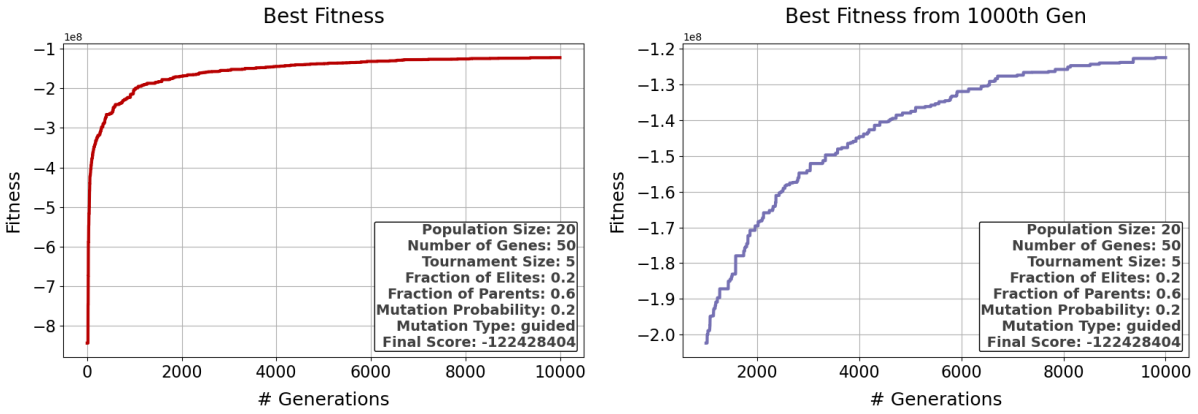


Figure 45: Quantitative evolution of the best individual in the population for stochastic mutation probability.

minima. Figure 46 shows the fitness curves for the best individual when the mutation probability is dynamic. The mutation probability decreases in each thousand generations, starting from 0.8.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 46: Fitness curves for mutation probability scheduling.

The evolution of the best individual for mutation probability scheduling is given in Figure 47.

We can see that the sche helped the algorithm to converge to a better solution. This is because the mutation probability is decreased in each thousand generations, and the algorithm can escape from local minima.

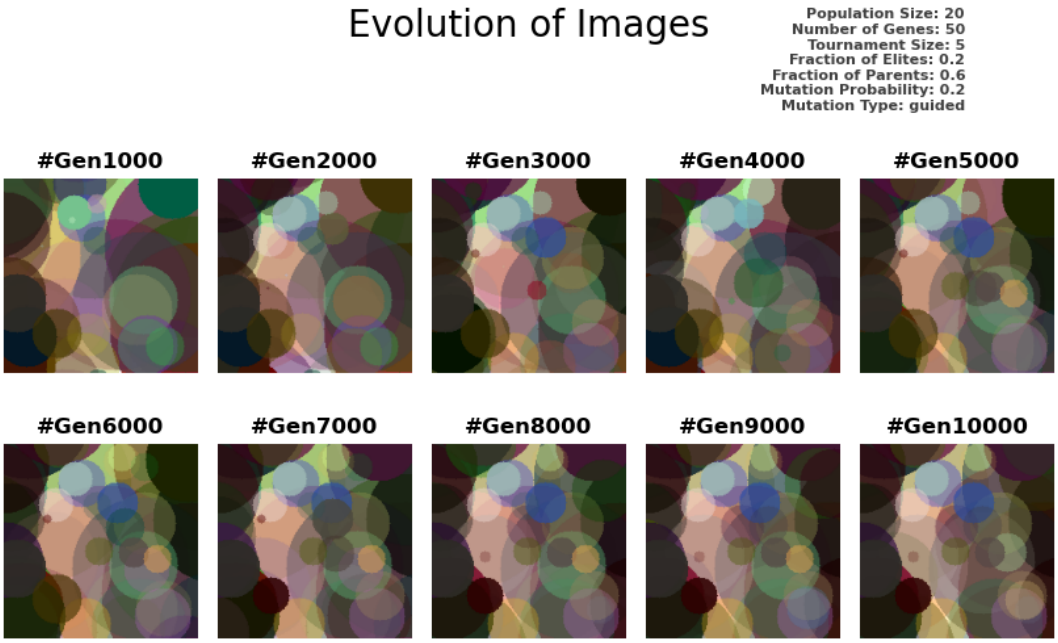
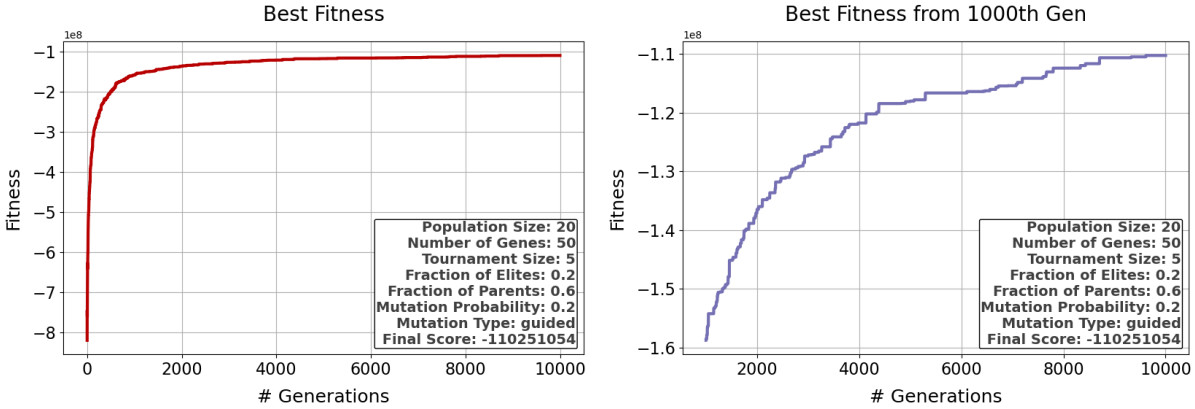


Figure 47: Quantitative evolution of the best individual in the population for dynamic mutation probability.

2.3 Elite/Parents Size Scheduling

The parent group size and range can be scheduled. This means that the elite and parents' sizes can be changed based on the number of generations. This would help the algorithm to favor elites later in the generations. Figure 48 shows the fitness curves for the best individual when the number of elites and a number of parents are tuned. The number of parents is decreased in each thousand generation while the number of elites is increased. That is, the fraction of parents decreased from 0.8 to 0.35 gradually, whereas the fraction of elites increased from 0.03 to 0.25.



(a) From first generation to 10000th generation.

(b) From 1000th generation to 10000th generation.

Figure 48: Fitness curves for elite/parents size scheduling.

The evolution of the best individual for dynamic family size is given in Figure 49.

This scheduling also helped the algorithm to converge to a better solution. This is because the

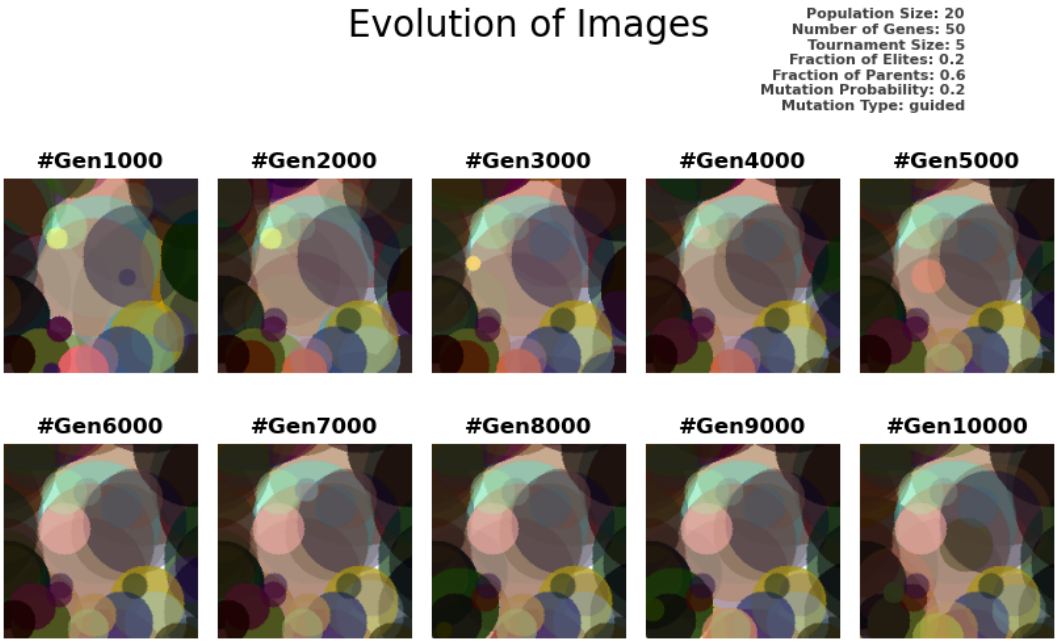


Figure 49: Quantitative evolution of the best individual in the population for elite/parents size scheduling.

algorithm can explore more possibilities in the early generations and then converge to a better solution in the later generations.

Appendix

The code set used throughout this homework is provided as follows.

```
# # Homework 2 Evolutionary Algorithms.
# This file will include all the necessary code for evolutionary algorithm homework. The task is described as "In this homework, you will perform experiments on evolutionary algorithm and draw conclusions from the experimental results. The task is to create an image made of filled circles, visually similar to a given RGB source image (painting.png)"
#
# ## Pseudo code
# Initialize population with <num_inds> individuals each having <num_genes> genes
# While not all generations (<num_generations>) are computed:
# Evaluate all the individuals
# Select individuals
# Do crossover on some individuals
# Mutate some individuals

# "Individual" class definition for evolutionary algoritm. There will be on chromosome and N number of genes. Each gene will have center coordinates (x,y), Radius, and RGB color.

import random
import numpy as np
import cv2
import copy
```

```
import h5py
from tqdm import tqdm


# Individual class definition
class Individual:
    def __init__(self, num_genes, image_size):
        self.num_genes = num_genes
        self.image_size = image_size
        self.chromosome = []
        self.fitness = np.float128(0.0)
        self.elite = False
        self.radius_max = image_size[0]//4

        for i in range(num_genes):
            x = random.randint(-1*self.radius_max, self.image_size[0]+1*self.radius_max)
            y = random.randint(-1*self.radius_max, self.image_size[1]+1*self.radius_max)
            r = random.randint(1, self.radius_max)
            color = [random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)]
            alpha = random.random()

            self.chromosome.append([x, y, r, color, alpha])


    def is_visible(self, gene=None):
        x = gene[0]
        y = gene[1]
        r = gene[2]
        if ((x + r) < 0) or (x  > self.image_size[0]+r) or ((y + r) < 0) or (y > self.image_size[1]+r):
            return False
        else:
            return True
    def mutate(self, mutation_probability, guidance = None):
        if not self.elite:
            if random.random() < mutation_probability:
                while True:
                    i = random.randint(0, self.num_genes-1)

                    if guidance is "unguided":
                        while True:
                            if not self.is_visible(self.chromosome[i]):
                                # randomly initialize the gene and check again
                                self.chromosome[i][0] = random.randint(-2*self.radius_max, self.image_size[0]+2*self.radius_max)
                                self.chromosome[i][1] = random.randint(-2*self.radius_max, self.image_size[1]+2*self.radius_max)
                                self.chromosome[i][2] = random.randint(0, self.radius_max*2)
                            else:
                                break

                        self.chromosome[i][3][0] = random.randint(0, 255)
                        self.chromosome[i][3][1] = random.randint(0, 255)
                        self.chromosome[i][3][2] = random.randint(0, 255)

                        self.chromosome[i][4] = random.random()
                    else:
```

```
                        #Guided mutation, deviate x,y, radius, color and alpha around
    their previous values

                        temp_x = copy.deepcopy(self.chromosome[i][0])
                        temp_y = copy.deepcopy(self.chromosome[i][1])
                        temp_r = copy.deepcopy(self.chromosome[i][2])

                        # mutate under the condition of visibility again and again until
    the gene is visible
                        while True:
                            self.chromosome[i][0] = int(temp_x + (self.image_size[0]/4)*
    random.uniform(-1,1))
                            self.chromosome[i][1] = int(temp_y + (self.image_size[1]/4)*
    random.uniform(-1,1))

                            if self.chromosome[i][0] < -2*self.radius_max:
                                self.chromosome[i][0] = -2*self.radius_max
                            elif self.chromosome[i][0] > self.image_size[0]+2*self.
    radius_max:
                                self.chromosome[i][0] = self.image_size[0]+2*self.
    radius_max

                            if self.chromosome[i][1] < -2*self.radius_max:
                                self.chromosome[i][1] = -2*self.radius_max
                            elif self.chromosome[i][1] > self.image_size[1]+2*self.
    radius_max:
                                self.chromosome[i][1] = self.image_size[1]+2*self.
    radius_max

                            self.chromosome[i][2] = int(temp_r + 10*random.uniform(-1,1))
                            if self.chromosome[i][2] < 0:
                                self.chromosome[i][2] = 1
                            elif self.chromosome[i][2] > self.radius_max*2:
                                self.chromosome[i][2] = self.radius_max*2

                            if self.is_visible(self.chromosome[i]):
                                break

                        self.chromosome[i][3][0] = int(self.chromosome[i][3][0] + 64*
    random.uniform(-1, 1))
                        if self.chromosome[i][3][0] < 0:
                            self.chromosome[i][3][0] = 0
                        elif self.chromosome[i][3][0] > 255:
                            self.chromosome[i][3][0] = 255

                        self.chromosome[i][3][1] = int(self.chromosome[i][3][1] + 64*
    random.uniform(-1, 1))
                        if self.chromosome[i][3][1] < 0:
                            self.chromosome[i][3][1] = 0
                        elif self.chromosome[i][3][1] > 255:
                            self.chromosome[i][3][1] = 255

                        self.chromosome[i][3][2] = int(self.chromosome[i][3][2] + 64*
    random.uniform(-1, 1))
                        if self.chromosome[i][3][2] < 0:
                            self.chromosome[i][3][2] = 0
                        elif self.chromosome[i][3][2] > 255:
                            self.chromosome[i][3][2] = 255

```

```
                        self.chromosome[i][3] = [int(x) for x in self.chromosome[i][3]]

                        self.chromosome[i][4] = self.chromosome[i][4] + 0.25*random.uniform(-1, 1)
                        if self.chromosome[i][4] < 0:
                            self.chromosome[i][4] = 0.001
                        elif self.chromosome[i][4] > 1:
                            self.chromosome[i][4] = 1

                    if random.random() > mutation_probability:
                        break
            else:
                pass
        else:
            # print("Cannot mutate elite individual")
            pass
    def draw(self):
        # First sort the genes by radius
        self.chromosome.sort(key=lambda x: x[2], reverse=True)

        # Create a blank image white background
        image = np.ones((self.image_size[1], self.image_size[0], 3), np.uint8)*255

        for i, gene in enumerate(self.chromosome):
            #check if the circle is visible in the image, center does not have to be in the image but the circle should be visible
            while True:
                if not self.is_visible(gene):
                    # randomly initialize the gene and check again
                    gene[0] = random.randint(-2*self.radius_max, self.image_size[0]+2*self.radius_max)
                    gene[1] = random.randint(-2*self.radius_max, self.image_size[1]+2*self.radius_max)
                    gene[2] = random.randint(0, self.radius_max*2)
                else:
                    break
            self.chromosome[i] = gene
            overlay = image.copy()
            cv2.circle(overlay, (gene[0], gene[1]), gene[2], gene[3], -1)
            image = cv2.addWeighted(overlay, gene[4], image, 1 - gene[4], 0)
        return image

    def calculate_fitness(self, target):
        image = self.draw()
        # Calculate the difference between the target image and the generated image
        target_np = np.array(target, dtype=np.int64)
        image_np = np.array(image, dtype=np.int64)
        diff = np.subtract(target_np, image_np)
        # take the square of the difference
        diff = np.square(diff)
        # sum of the squared differences
        # print(np.sum(diff))
        self.fitness = -np.sum(diff)

    def crossover(self, partner):
        child1 = Individual(self.num_genes, self.image_size)
        child2 = Individual(self.num_genes, self.image_size)

        for i in range(self.num_genes):
```

```
            if random.random() < 0.5:
                child1.chromosome[i] = copy.deepcopy(self.chromosome[i])
                child2.chromosome[i] = copy.deepcopy(partner.chromosome[i])
            else:
                child1.chromosome[i] = copy.deepcopy(partner.chromosome[i])
                child2.chromosome[i] = copy.deepcopy(self.chromosome[i])
        return child1, child2

# Popoulation class definition

class Population:
    def __init__(self, num_individuals, num_genes, image_size, frac_elites, frac_parents, tm_size, target_image, guidance):
        self.num_individuals = num_individuals
        self.num_genes = num_genes
        self.image_size = image_size
        self.num_elites = int(frac_elites*self.num_individuals)
        self.num_parents = int(frac_parents*self.num_individuals)
        if self.num_parents % 2 != 0:
            self.num_parents += 1
        self.tm_size = tm_size
        self.guidance = guidance

        self.target = target_image
        self.individuals = []
        self.parents = []

        for i in range(self.num_individuals):
            self.individuals.append(Individual(self.num_genes, self.image_size))

    def selection(self):

        self.individuals.sort(key=lambda x: x.fitness, reverse=True)

        new_individuals = []
        # Mark the best individuals as elite
        for i in range(self.num_elites):
            self.individuals[i].elite = True
            new_individuals.append(self.individuals[i])
        # Select the rest of the individuals using tournament selection.
        non_elite_individuals = self.individuals[self.num_elites:]

        parentable_individuals = []

        # update the non elite group by adding the best individual from each group to parentable individuals

        for i in range(len(non_elite_individuals)):
            group = random.sample(non_elite_individuals, min(self.tm_size, len(non_elite_individuals)))

            group.sort(key=lambda x: x.fitness, reverse=True)

            parentable_individuals.append(copy.deepcopy(group[0]))

        # select the best parents from the parentable individuals
        parentable_individuals.sort(key=lambda x: x.fitness, reverse=True)
        self.parents = parentable_individuals[:self.num_parents]
        # non elite non parent elements will be added to the new individuals
```

```
231         new_individuals.extend(parentable_individuals[self.num_parents:])
232         # The new generation except the children will be the new individuals
233         self.individuals = new_individuals
234 
235     def crossover(self):
236         # parents will create new individuals by crossover. Two parents will create two
    children
237         children = []
238         random.shuffle(self.parents)
239 
240         for i in range(0, self.num_parents, 2):
241             parent1 = self.parents[i]
242             parent2 = self.parents[i+1]
243 
244             child1, child2 = parent1.crossover(parent2)
245 
246             child1.calculate_fitness(self.target)
247             child2.calculate_fitness(self.target)
248 
249             children.append(child1)
250             children.append(child2)
251 
252         self.individuals.extend(children)
253 
254     def mutation(self, mutation_probability):
255         #check if the individual is an elite, if so do not mutate
256         for individual in self.individuals:
257             individual.mutate(mutation_probability, guidance = self.guidance)
258 
259     def evaluate(self):
260         for individual in self.individuals:
261             individual.calculate_fitness(self.target)
262 
263     def get_best(self):
264         self.individuals.sort(key=lambda x: x.fitness, reverse=True)
265         return self.individuals[0]
266 
267     def get_average_fitness(self):
268         return sum([x.fitness for x in self.individuals]) / self.num_individuals
269 
270 
271 
272 
273 # default parameters in dictionary
274 parameters_list = [{
275     "num_individuals": 20,
276     "num_genes": 50,
277     "tournament_size": 5,
278     "frac_elites": 0.2,
279     "frac_parents": 0.6,
280     "mutation_probability": 0.2,
281     "mutataion_type": "guided"
282 }, {
283     "num_individuals": 5,
284     "num_genes": 50,
285     "tournament_size": 5,
286     "frac_elites": 0.2,
287     "frac_parents": 0.6,
288     "mutation_probability": 0.2,
```

```
    "mutataion_type": "guided"
}, {
    "num_individuals": 10,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 40,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 60,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 15,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 30,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 80,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 120,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
```

```
    "num_genes": 50,
    "tournament_size": 2,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 8,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 16,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.04,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.35,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.15,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.3,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
```

```
    "frac_parents": 0.75,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.1,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.4,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.75,
    "mutataion_type": "guided"
},{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "unguided"
}
]

for i, parameters in enumerate(parameters_list):
    num_individuals = parameters["num_individuals"]
    num_genes = parameters["num_genes"]
    tournament_size = parameters["tournament_size"]
    frac_elites = parameters["frac_elites"]
    frac_parents = parameters["frac_parents"]
    mutation_probability = parameters["mutation_probability"]
    mutataion_type = parameters["mutataion_type"]

    print("Parameters for run ", i+1)

    print("Summary of parameters")
    print("Number of individuals: ", num_individuals)
    print("Number of genes per individual: ", num_genes)
    print("Tournament size: ", tournament_size)
    print("Fraction of elites: ", frac_elites)
    print("Fraction of parents: ", frac_parents)
    print("Mutation probability: ", mutation_probability)
    print("Mutation type: ", mutataion_type)

    # load input image
```

```
    target = cv2.imread("hw2/painting.png")

    image_size = (target.shape[1], target.shape[0])


    pop = Population(num_individuals, num_genes, image_size, frac_elites, frac_parents,
    tournament_size, target,guidance=mutataion_type)


    # iterate over generations
    average_fitness = []
    best_fitness = []
    image_of_best = []

    num_generations = 10000

    for i in tqdm(range(num_generations)):
        pop.evaluate()
        pop.selection()
        pop.crossover()
        pop.mutation(mutation_probability)
        pop.evaluate()
        #reset elite status
        for individual in pop.individuals:
            individual.elite = False
        # print("num individuals left",len(pop.individuals))
        average_fitness.append(pop.get_average_fitness())
        best_fitness.append(pop.get_best().fitness)
        if (i+1) % 1000 == 0:
            image_of_best.append(pop.get_best().draw())
    print("Generation: ", i, "Average fitness: ", pop.get_average_fitness(), "Best
    fitness: ", pop.get_best().fitness)

    # set filename using the parameters

    filename = "hw2/out_sweep/output_" + str(num_individuals) + "_" + str(num_genes) + "_
    " + str(tournament_size) + "_" + str(frac_elites) + "_" + str(frac_parents) + "_" +
    str(mutation_probability) + "_" + mutataion_type + ".h5"

    # save images and average best fitness values to a file
    with h5py.File(filename, "w") as f:
        f.create_dataset("average_fitness", data=average_fitness)
        f.create_dataset("best_fitness", data=best_fitness)
        for i, image in enumerate(image_of_best):
            f.create_dataset("image_"+str(i-1000), data=image)
```

```
# this file will be used to analyze the data from the h5py data files located in the data
    folder

import h5py
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cm
import matplotlib.patches as patches
import matplotlib.gridspec as gridspec
import matplotlib.ticker as ticker
import matplotlib.colors as colors
```

```

import os
import cv2

# function to read the data from the h5py file
def read_data(file_name):
    # open the file
    file = h5py.File(file_name, 'r')
    # get the data called "average_fitness"
    average_fitness = file['average_fitness'][:]
    # get the data called "best_fitness"
    best_fitness = file['best_fitness'][:]
    # get the data called "image_-1000"
    image_0 = file['image_-1000'][:]
    # get the data called "image_-999"
    image_1 = file['image_-999'][:]
    # get the data called "image_-998"
    image_2 = file['image_-998'][:]
    # get the data called "image_-997"
    image_3 = file['image_-997'][:]
    # get the data called "image_-996"
    image_4 = file['image_-996'][:]
    # get the data called "image_-995"
    image_5 = file['image_-995'][:]
    # get the data called "image_-994"
    image_6 = file['image_-994'][:]
    # get the data called "image_-993"
    image_7 = file['image_-993'][:]
    # get the data called "image_-992"
    image_8 = file['image_-992'][:]
    # get the data called "image_-991"
    image_9 = file['image_-991'][:]
    # put them into the data array
    data = [image_0, image_1, image_2, image_3, image_4, image_5, image_6, image_7,
    image_8, image_9]

    # return the data and labels
    return data, average_fitness, best_fitness

# get list of files with h5 extension in the data folder
data_folder = "hw2/out_sweep/" #
files = os.listdir(data_folder)
files = [file for file in files if file.endswith(".h5")]


# loop through the files and read the data
for file in files:
    file = "outputFRAC_20_50_5_0.2_0.6_0.2_guided.h5"
    data, average_fitness, best_fitness = read_data(data_folder + "/" + file)
    # get the paremeter values from the file name
    parts = file.split("_")

    # get the population size
    num_individuals = parts[1]
    # get the number og genes
    num_genes = parts[2]
    # get the tournament size
    tournament_size = parts[3]
    # get the fraction of elites
```

```
    fraction_elites = parts[4]
    # get the fraction of parents
    fraction_parents = parts[5]
    # get the mutation probability
    mutation_probability = parts[6]
    # get the mutation type
    mutation_type = parts[7].split(".")[0]

    # create the text info
    text_info = "Population Size: " + num_individuals + "\nNumber of Genes: " + num_genes
     + "\nTournament Size: " + tournament_size + "\nFraction of Elites: " +
    fraction_elites + "\nFraction of Parents: " + fraction_parents + "\nMutation
    Probability: " + mutation_probability + "\nMutation Type: " + mutation_type
    # plot the average fitness
    # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
    fig = plt.figure(figsize=(8.3, 5.8))
    gs = gridspec.GridSpec(1, 1)
    plt.rcParams["font.family"] = "sans-serif"
    plt.rcParams["font.sans-serif"] = ["DejaVu Sans"]
    ax0 = plt.subplot(gs[0, 0])
    ax0.plot(average_fitness, linestyle='-', color='#5F8670', linewidth=3.0, alpha=1)
    ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)
    ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
    ax0.grid()
    ax0.set_title('Average Fitness', fontsize=20, pad=20)
    ax0.tick_params(axis='both', which='major', labelsize=16)
    ax0.text(0.985, 0.225, text_info + "\n Final Score: " + str(average_fitness[-1]),
    horizontalalignment='right', verticalalignment='center', transform=ax0.transAxes,
    fontsize=14, fontweight='bold', color='black', alpha=0.75, bbox=dict(facecolor='white
    ', edgecolor='black', boxstyle='round,pad=0.1'))
    plt.tight_layout()
    #plt.show()
    # save the figure
    fig.savefig("hw2/out_sweep/average_fitness_" + file.split(".h")[0] + ".png")

    # plot the best fitness
    # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
    fig = plt.figure(figsize=(8.3, 5.8))
    gs = gridspec.GridSpec(1, 1)
    plt.rcParams["font.family"] = "sans-serif"
    plt.rcParams["font.sans-serif"] = ["DejaVu Sans"]
    ax0 = plt.subplot(gs[0, 0])
    ax0.plot(best_fitness, linestyle='-', color='#B80000', linewidth=3.0, alpha=1)
    ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)
    ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
    ax0.grid()
    ax0.set_title('Best Fitness', fontsize=20, pad=20)
    ax0.tick_params(axis='both', which='major', labelsize=16)
    ax0.text(0.985, 0.225, text_info + "\n Final Score: " + str(best_fitness[-1]),
    horizontalalignment='right', verticalalignment='center', transform=ax0.transAxes,
    fontsize=14, fontweight='bold', color='black', alpha=0.75, bbox=dict(facecolor='white
    ', edgecolor='black', boxstyle='round,pad=0.1'))
    plt.tight_layout()
    # plt.show()
    fig.savefig("hw2/out_sweep/best_fitness_" + file.split(".h")[0] + ".png")


    # plot the best fitness
    # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
```

```
    fig = plt.figure(figsize=(8.3, 5.8))
    gs = gridspec.GridSpec(1, 1)
    plt.rcParams["font.family"] = "sans-serif"
    plt.rcParams["font.sans-serif"] = ["DejaVu Sans"]
    ax0 = plt.subplot(gs[0, 0])
    ax0.plot(range(1000, 10000), best_fitness[1000:], linestyle='-', color='#7570b3', linewidth=3.0, alpha=1)
    ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)
    ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
    ax0.grid()
    ax0.set_title('Best Fitness from 1000th Gen', fontsize=20, pad=20)
    ax0.tick_params(axis='both', which='major', labelsize=16)
    ax0.text(0.985, 0.225, text_info + "\n Final Score: " + str(best_fitness[-1]), horizontalalignment='right', verticalalignment='center', transform=ax0.transAxes, fontsize=14, fontweight='bold', color='black', alpha=0.75, bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.1'))
    plt.tight_layout()
    # plt.show()
    fig.savefig("hw2/out_sweep/best_fitness_1000_" + file.split(".h")[0] + ".png")


    # plot the images for the 10 generations
    fig, axs = plt.subplots(2, 5, figsize=(8.3, 5.8))
    fig.suptitle("Evolution of Images", fontsize=20)
    for i in range(10):
        ax = axs[i//5, i%5]
        # convert the image BGR to RGB
        data[i] = cv2.cvtColor(data[i], cv2.COLOR_BGR2RGB)

        ax.imshow(data[i])
        ax.set_title("#Gen" + str((i+1)*1000), fontsize=12, fontweight='bold')
        ax.axis('off')
    # add text to the figure
    fig.text(0.9, 0.845, text_info, ha='right', fontsize=8, fontweight='bold', color='black', alpha=0.75)
    fig.tight_layout(rect=[0, 0.03, 1, 0.95])
    # plt.show()
    fig.savefig("hw2/out_sweep/images_" + file.split(".h")[0] + ".png")

    print("Done")
```