EE449 HW2 EVOLUTIONARY ALGORITHMS

Kutay Uğurlu 2232841

June 2, 2021

Introduction and Simulation Procedure

In this section, first the task is described and then some details regarding the implementation of the algorithm is presented.

The population is initialized with the specified number of individuals and the fitness evaluation of the individuals is performed by calculating the Frobenius norm of the difference of the source image and the formed image, $\|IMG_{source} - IMG\|_F$. Using this definition for fitness causes different phenotypes to yield same fitness value, since a fixed fitness value describes "a sphere" having a radius of $\mathcal{F}(image) = \|image\|_F$ in the 180 x 180 x 3 dimensional space, describing the number of pixels. Hence the performance comparison of the different parameter configurations will be mainly based on the fitness values that the algorithm optimizes thorough generations, although some amount of covergence is observed in every experiment at least as in the form of color theme matching. In addition, the tournament selection is performed on randomly and simultaneously selected individuals by shuffling and slicing the population array. The results obtained using the default parameter configuration is given below.

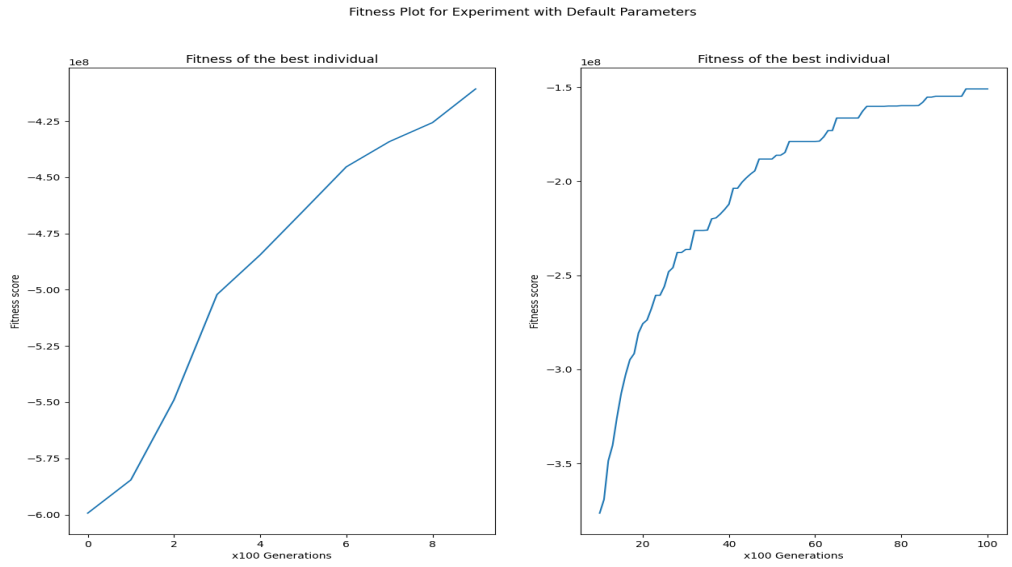

Figure 1: Fitness of the best individual in default parameter experiments

As can be observed in *Figure.2*, the best individual seems to approach the source image, with a couple of matching details including the body and head at the bottom of the figure, matching color themes of the orange-ish sky and matching colors at the bottom right of the figure. Similar convergence is also observed in the other experiments, although not significantly due to the different initializations and randomized operation like selection and mutation. However, to compare the hyperparameters' effects, the discussion is going to be mainly based on the fitness plots, excluding the already conducted default parameter experiment.

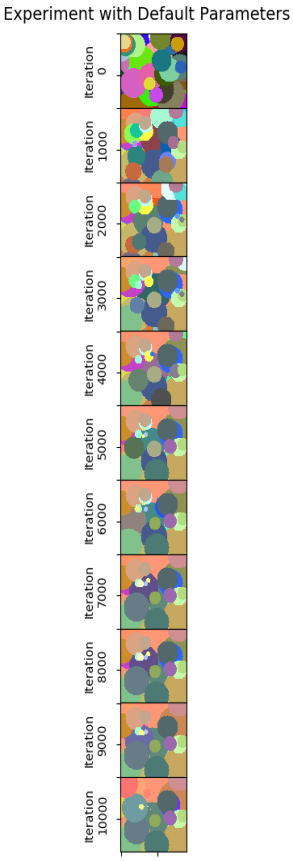

Figure 2: Phenotype of the best individual in default parameters experiments

Results

In the following section, first the results regarding the different experiments will be presented, then related discussion takes place.

Number of Genes

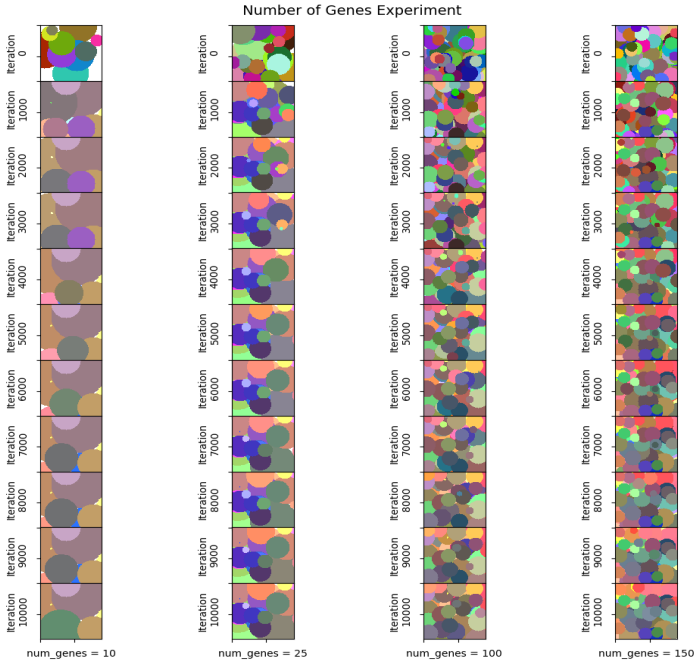

Figure 3: Phenotype of the best individual in number of genes experiment

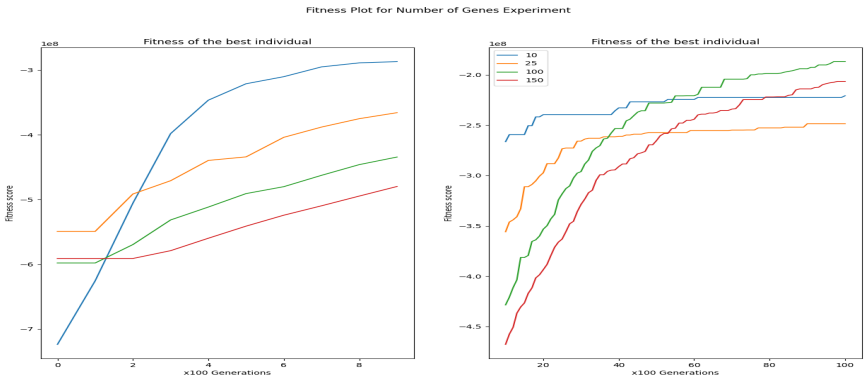

Figure 4: Fitness of the best individual in number of genes experiments

Number of Individuals

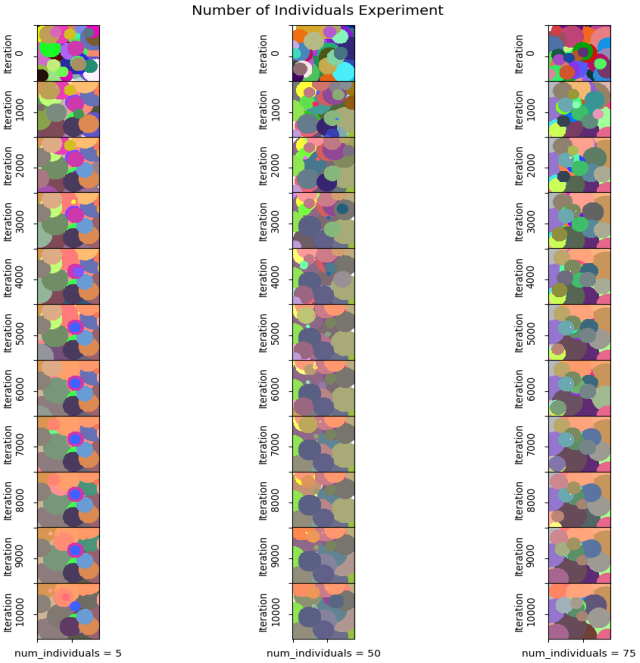

Figure 5: Phenotype of the best individual in number of individuals experiment

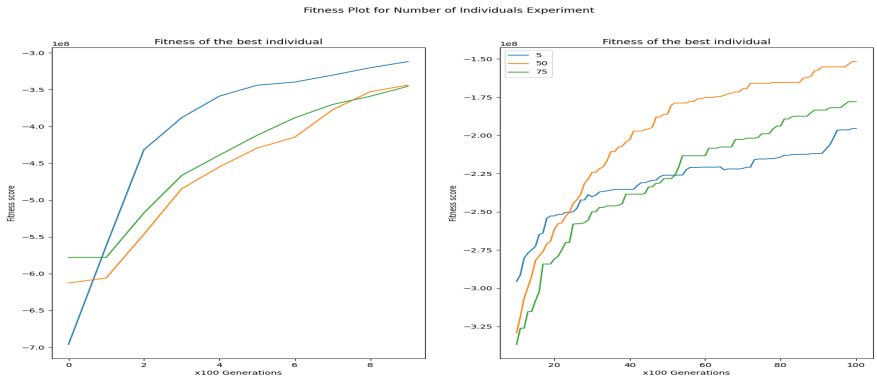

Figure 6: Fitness of the best individual in number of individuals experiments

Mutation Probability

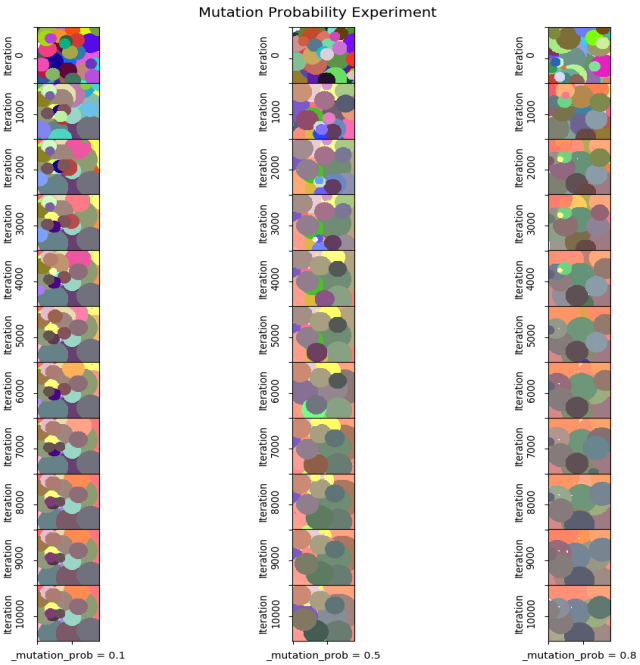

Figure 7: Phenotype of the best individual in mutation probability experiment

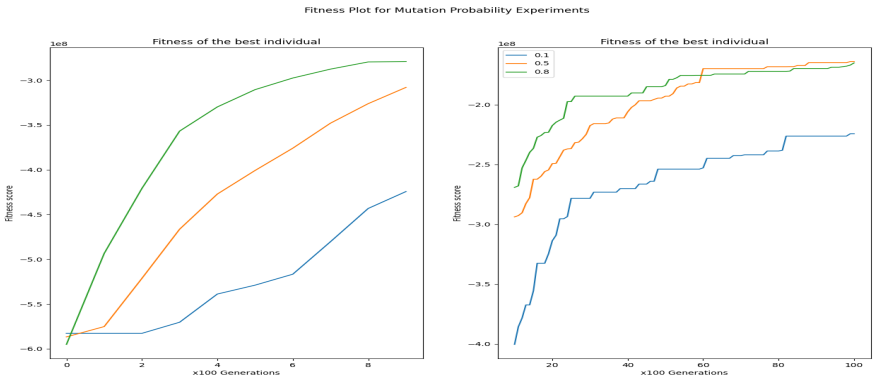

Figure 8: Fitness of the best individual in mutation probability experiment

Mutation Type

Figure 9: Phenotype of the best individual in mutation type experiment

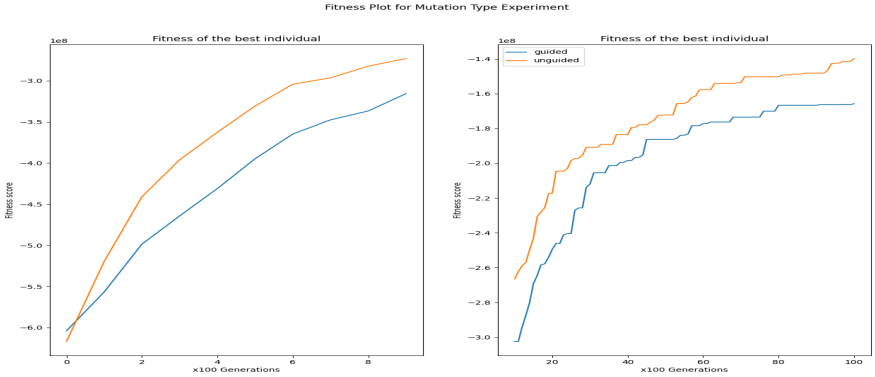

Figure 10: Fitness of the best individual in mutation type experiment

Fraction of Parents

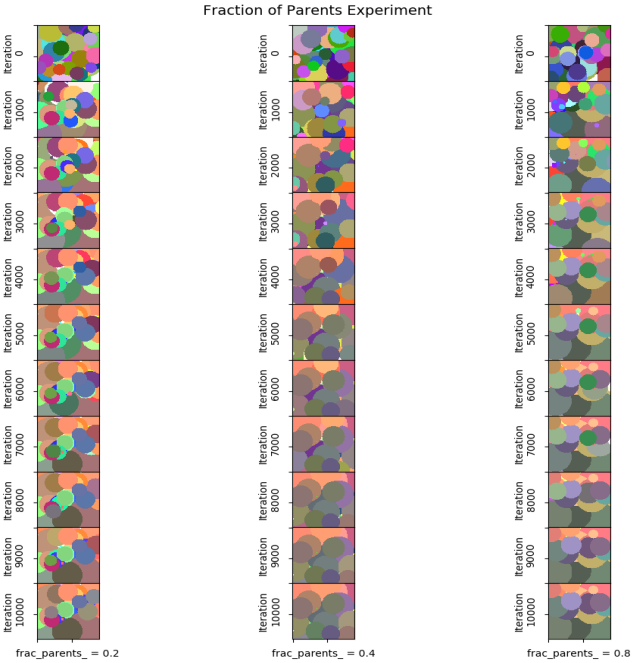

Figure 11: Phenotype of the best individual in fraction of parents experiment

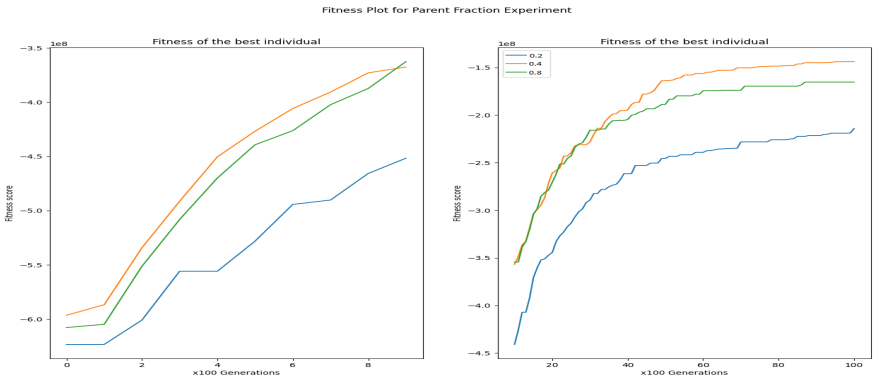

Figure 12: Fitness of the best individual in fraction of parents experiment

Fraction of Elites

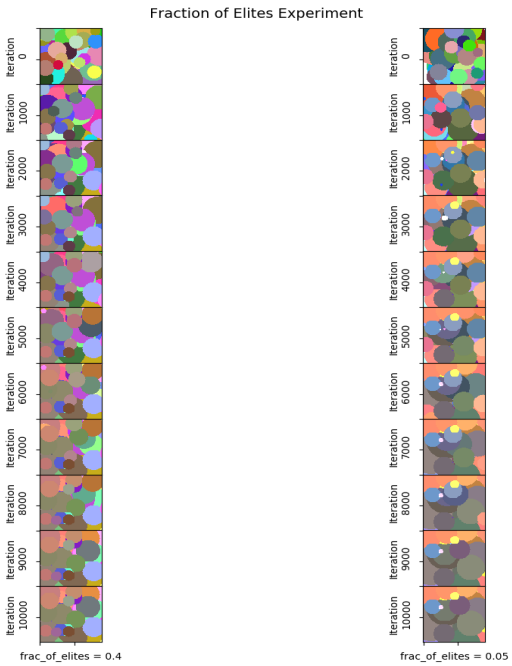

Figure 13: Phenotype of the best individual in fraction of elites experiment

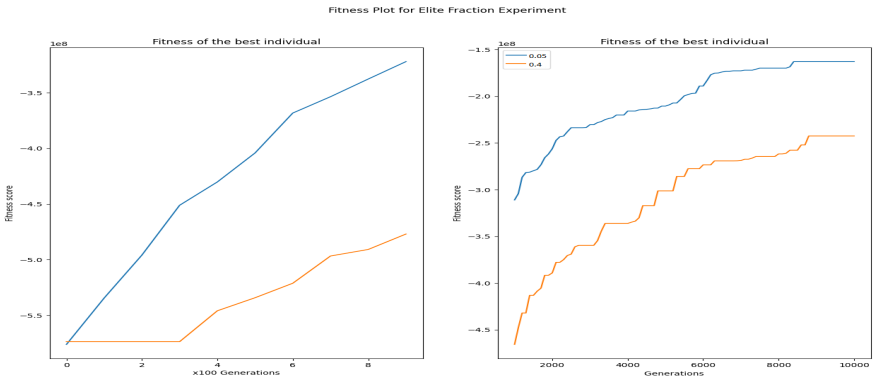

Figure 14: Fitness of the best individual in fraction of elites experiment

Tournament Size

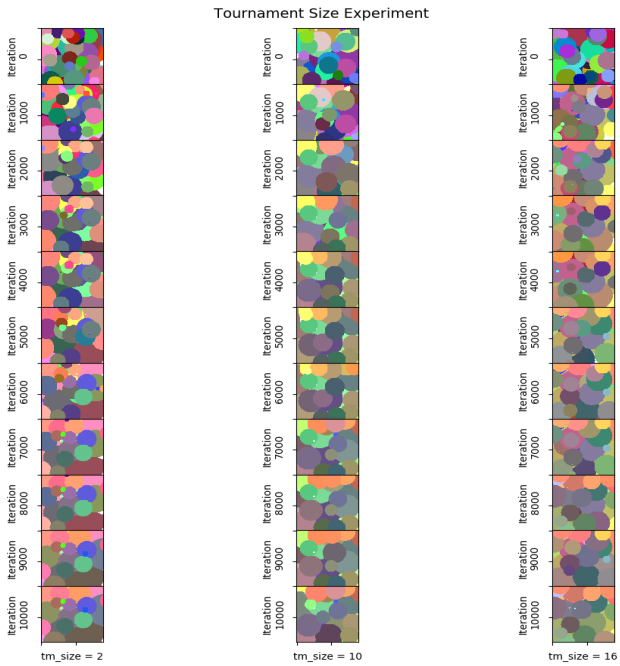

Figure 15: Phenotype of the best individual in tournament size experiment

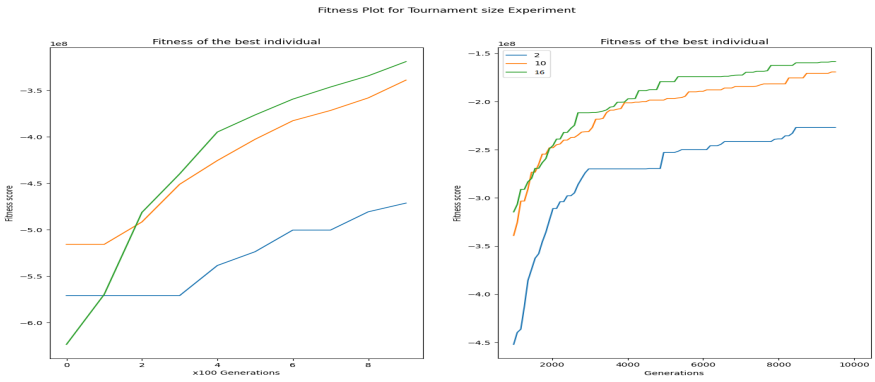

Figure 16: Fitness of the best individual in tournament size experiment

Discussion

The brief conclusion of the experiments are as follows:

1. **Number of Genes**: Although convergence with lower number of genes was faster, higher number of genes provide a chance to achieve higher variation and find a better solution. However, 100 genes revealed better fitness than 150 genes, probably due to the decreased chance of choosing better individuals with tournament. In addition, 150 genes case may require more time to converge a better fitness value. Therefore, using fixed number of generation may have resulted in this relation between the fitness curves.

2. **Number of Individuals**: Again, lower number of individuals resulted in faster convergence, although 75 individual has not achieved a fitness value that 50 genes achieved. This situation can be explained as in the number of genes case, and 50 genes can be used in the tuned-hyperparameter experiment at the end.

3. **Tournament Size**: The higher tournament size always has the higher probability to select the better individuals. The results are as expected.

4. **Mutation Type**: Guided mutations changes a value of the gene around the current value of the gene, hence restricting the change in the fitness around the fitness value of the individual. On the other hand, unguided mutations change the value of the randomly selected genes completely randomized, providing a better reach for the better individuals in the search space.

5. **Mutation Probability**: Higher mutation probability caused a higher genetic variation in the population, hence better exploration of the search space.

6. **Fraction of Elites**: Choosing a lower elite fraction resulted in better performance, providing chance for more individuals to vary their genetic material, hence fitness.

7. **Fraction of Parents**: Choosing 40% of the Non-Elite individuals revealed better results.

Changes to Improve the Performance of the Evolutionary Algorithm

Optimized/Tuned Hyperparameters

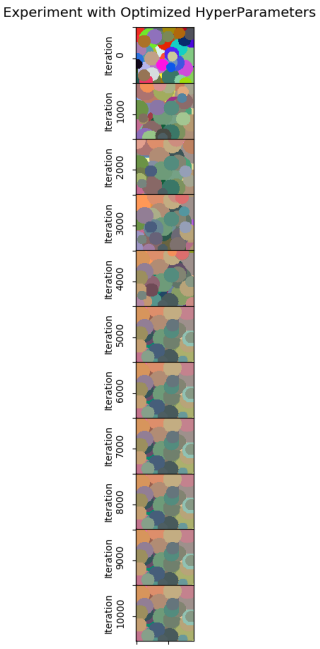

Figure 17: Phenotype of the best individual in the Optimized Hyperparameters experiment

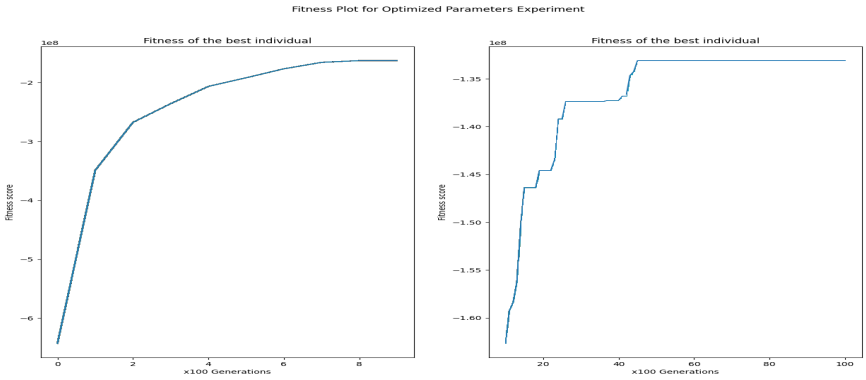

Figure 18: Fitness of the best individual in the Optimized Hyperparameters Experiment

Scheduled Fraction of Elites

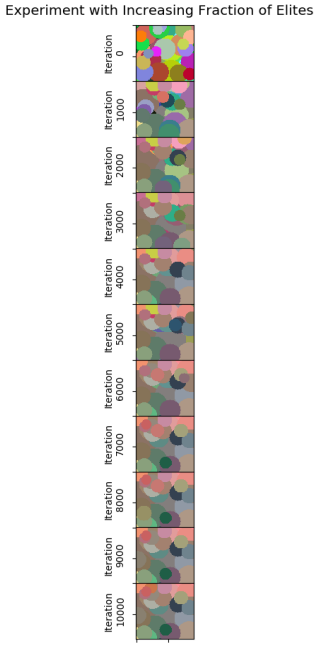

Figure 19: Phenotype of the best individual in the Scheduled Fraction of Elites experiment

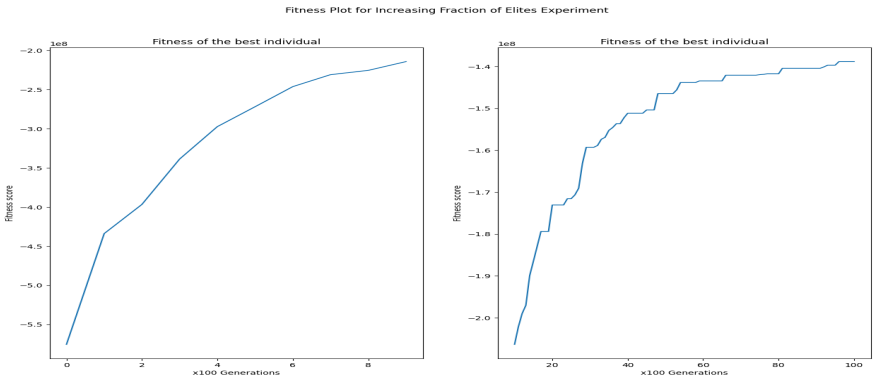

Figure 20: Fitness of the best individual in the Scheduled Fraction of Elites Experiment

Scheduled Mutation Probability

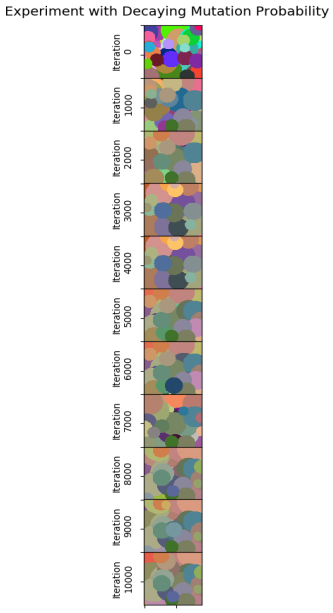

Figure 21: Phenotype of the best individual in the Scheduled Mutation Probability experiment

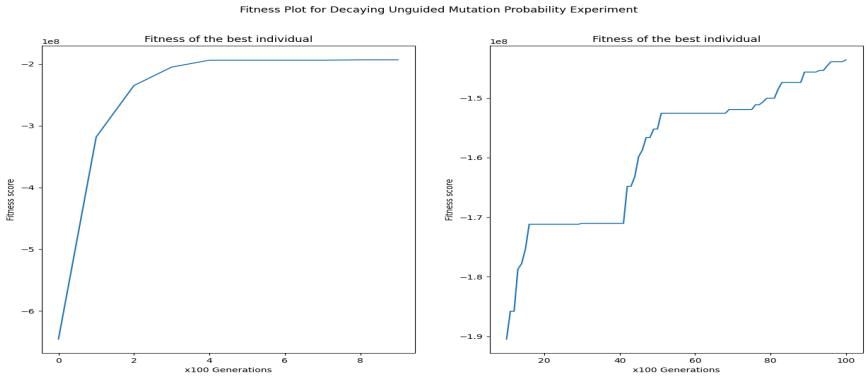

Figure 22: Fitness of the best individual in the Scheduled Mutation Probability Experiment

Conclusions

The results of the three suggestions to improve the convergence of the algorithm was presented in the above figures. These suggestions are as follows:

- **Hyperparameter Optimization/Tuning** : By looking at the previous results, the algorith was run with the parameters that are found as most successful :
  1. Number of Genes: 100
  2. Number of Individuals : 50
  3. Fraction of Elites : 0.05
  4. Fraction of Parents : 0.4
  5. Mutation Prob: 0.8
  6. Mutation Type : Unguided
- **Scheduled Mutation Probability** : To explore the search space first, and to exploit them in the latter generations, a decresing mutation probability thoroughout the generations were applied in the following manner:
  1. Generations 0 - 4000 : 0.8
  2. Generations 4000 - 7000 : 0.5
  3. Generations 7000 - 10000 : 0.2
- **Increasing Fraction of Elites** : The idea iin this experiment gets its motivation from the same exploration-exploitation of the search space relationship mentioned in the previous experiment. However, an increasing ratio of elites should be used in this experiment to accomplish the same, unline the mutation probability. Selection of elites restricts the genetic variation in the population and saves the genetic materials of them directly to next generation. By first allowing the algorithm to explore the space, then advancing more individuals to the next generation in the latter generations, provided a better convergence. The fraction of elites was used in the experiment is as follows:
  1. Generations 0 - 4000 : 0.05
  2. Generations 4000 - 7000 : 0.1
  3. Generations 7000 - 10000 : 0.2

Compared to earlier experiments, it can be easily observed that convergence to a specific fitness value, say $-2 \cdot 10^8$, occured in significantly early generation for all of the improvement proposals. In the experiment with default parameters, it was observed that the best individual has reached a fitness value of $-2 \cdot 10^8$ in approximately 5000 generations, whereas with the improvements, the best individual attains the same fitness in the first 1000 generations, even in the 4000^{th} for the scheduled mutation probability experiment. Hence, the fitness plots prove that the suggested improvements worked successfully. In addition to these, the selection method of elites could also be change to the **Roulette Wheel Selection** by randomizing the selection of individuals with the highest fitness to provide wider exploration of the search space.

APPENDIX

The Evolutionary Algorithm Main Simulation

```
# Imports
import cv2
import numpy as np
from matplotlib import pyplot as plt
from random import randint, random, shuffle, uniform
from copy import deepcopy as dcopy
from multiprocessing import Pool

from numpy.lib.utils import source
source_img = cv2.imread("painting.png")
HEIGHT, WIDTH, CHANNELS = source_img.shape

# Helpers


def ramp(x):
    return max(0, x)


def create_blank_img():
    img = np.ones((HEIGHT, WIDTH, 3), np.uint8)
    img[:] = 255
    return dcopy(img)


def Pythagorean(x, y):
    return (x**2+y**2)**0.5


def create_empty_list(empty_list=None):
    if empty_list == None:
        empty_list = []
    return empty_list

# Classes


class Gene():
    global HEIGHT, WIDTH

    def __init__(self, x=0, y=0, r=0, R=0, G=0, B=0, A=0):
        self.x = x
        self.y = y
        self.r = r
        self.R = R
        self.G = G
        self.B = B
        self.A = A

    def random_initialize(self):
```

```
        self.x = randint(0, HEIGHT)
        self.y = randint(0, WIDTH)
        self.r = randint(15, 60)
        self.R = randint(0, 255)
        self.G = randint(0, 255)
        self.B = randint(0, 255)
        self.A = random()

    def check_if_in_image(self):
        if self.x < self.r or abs(self.x-WIDTH) < self.r or abs(self.y-HEIGHT) < self.r or
                                                self.y < self.r or (0 < self.x < WIDTH and 0 <
                                                 self.y < HEIGHT):
            return True
        else:
            return False

# Since an individual has only one chromosome, I avoided introducing extra classes and used
                                        genes directly under individiual.


class Ind():

    def __init__(self, num_genes=None, genes=None, fitness=0):
        # Population is initialized with random genes if genes argument is not given, until
                                                all the genes have the corresponding phenotype
                                                 lying in the image.
        i = 0
        if genes is None:
            genes = create_empty_list()
            while i < num_genes:
                gene = Gene()
                gene.random_initialize()
                if gene.check_if_in_image():
                    genes.append(gene)
                    i += 1
        self.genes = genes
        self.calculate_fitness()

    def draw(self):
        # Individuals genes are sorted based on the radius, and regarding phenotypes are
                                                drawn over the blank image.
        sorted_genes = sorted(
            self.genes, key=lambda item: item.r, reverse=True)
        res_img = create_blank_img()
        for gene in sorted_genes:
            overlay = res_img
            cv2.circle(overlay, center=(gene.x, gene.y), radius=gene.r,
                       color=tuple(map(lambda item: item/1, (gene.B, gene.G, gene.R))),
                                                            thickness=-1)
            res_img = cv2.addWeighted(overlay, gene.A, res_img, 1-gene.A, 0)
        return res_img

    def calculate_fitness(self):
        # Return Frobenius Norm of the difference.
        fitness = -1 * np.linalg.norm(np.int64(source_img)-self.draw())**2
```

```
        self.fitness = fitness


def evaluate_population(population):
    # Calls fitness calculator on every individual of population.
    for ind in population:
        ind.calculate_fitness()
    return population


def create_population(num_individuals, num_genes):
    Population = create_empty_list()
    for _ in range(num_individuals):
        ind = Ind(num_genes)
        ind.calculate_fitness()
        Population.append(ind)
    return Population


def choose_n_elites(population, frac_of_elites):
    # Population is sorted in decreasing order and the n best individuals are returned.
    n = int(len(population) * frac_of_elites)
    sorted_pop = sorted(
        dcopy(population), key=lambda item: item.fitness, reverse=True)
    return sorted_pop[0:n], sorted_pop[n:]


def tournament_selection(Remaining, tm_size, parent_size):
    # First the population are shuffled to draw some number of random individuals, selected
                                        individuals are sorted and best of them is
                                        returned.
    Winners = create_empty_list()
    for _ in range(parent_size):
        shuffle(Remaining)
        Remaining = Remaining[0:tm_size]
        sorted_remains = sorted(
            Remaining, key=lambda item: item.fitness, reverse=True)
        Winners.append(dcopy(sorted_remains[0]))
    return Winners


def unguided_mutation(gene: Gene):
    gene.random_initialize()
    return gene


def guided_mutation(gene: Gene):

    # Applied the specified mutation rules to the genes, paying attention on the clipping
                                        and corrections.
    gene.x += randint(int(-0.25*gene.x), int(0.25*gene.x))
    gene.y += randint(int(-0.25*gene.y), int(0.25*gene.y))

    # Negative values are not allowed, hence used ramp
    gene.r = randint(ramp(gene.r-10), gene.r+10)
```

```
    gene.R = randint(ramp(gene.R-64), gene.R+64) if gene.R+64 < 255 else 255
    gene.G = randint(ramp(gene.G-64), gene.G+64) if gene.G+64 < 255 else 255
    gene.B = randint(ramp(gene.B-64), gene.B+64) if gene.B+64 < 255 else 255

    # [0,1]->[-1,1]->[-0.25,0.25]
    shifted_rn = uniform(-0.25, 0.25)
    gene.A += shifted_rn
    gene.A = gene.A if gene.A <= 1 and gene.A >= 0 else (
        0 if gene.A < 0 else 1)
    return gene


def mutate_individual(mutation, mutation_prob, ind: Ind):

    # Randomly selects a gene and mutates it if the mutation probability allows
    if random() < mutation_prob:
        genes = ind.genes
        gene_idx = randint(0, num_genes-1)
        if mutation.lower() == "guided":
            gene = guided_mutation(genes[gene_idx])
        else:
            gene = unguided_mutation(genes[gene_idx])
        ind.genes[gene_idx] = gene
    return ind


def draw_Pop(ind_list):
    # Creates the phenotype of the population
    genes = create_empty_list()
    for ind in ind_list:
        genes = genes + ind.genes
    IND = Ind(len(genes), genes=genes)
    img = IND.draw()
    return img


def crossover(parents: list):

    # Swaps genes of the parents with a certain probability, 0.5 here
    p1_genes = parents[0].genes
    p2_genes = parents[1].genes

    # initialize children genes
    ch1 = Ind(genes=p1_genes)
    ch2 = Ind(genes=p2_genes)

    # Exchange of genes with probability 0.5
    for i in range(len(ch1.genes)):
        if random() > 0.5:
            ch1.genes[i], ch2.genes[i] = dcopy(
                ch1.genes[i]), dcopy(ch2.genes[i])

    return [ch1, ch2]
```

```
def crossover_population(Parents: list):
    # Returns children of the parent population
    child_list = [crossover([Parents[2*i], Parents[2*i+1]])
                  for i in range((len(Parents)//2))]
    children = [x for l in child_list for x in l]
    return children


def round_to_even(a: int):
    if int(a) % 2 == 0:
        return int(a)
    else:
        return int(a)-1


# Set params
frac_of_elites = 0.2
num_individuals = 20
num_genes = 50
frac_parents = 0.6
mutation_prob = 0.2
mutation_type = "guided"
tm_size = 16
generations = 10000

curve_freq = 100
directory = "OUTPUTs//MUTPROB_new//"
EXP_NAME = ""

plt.figure(figsize=(15, 10))

#####################          EVOLUTION           ##########################

legends = [0.1, 0.5, 0.8]
for mutation_prob in legends:

    # First create population
    Pop = create_population(
        num_individuals=num_individuals, num_genes=num_genes)

    # Fitness Curve
    Fitness_Curve = create_empty_list()

    for generation in range(generations+1):

        Pop = evaluate_population(Pop)

        Elites, NonElites = choose_n_elites(Pop, frac_of_elites)
        R = len(NonElites)

        num_parents = int(num_individuals*frac_parents)
        to_be_mutated = R - num_parents

        Winners = tournament_selection(
            NonElites, tm_size, R)
```

```
        # Parents are selected among the Non Elite Population
        Parents, Remainings = Winners[0:num_parents], Winners[-1*to_be_mutated:]

        # Children are produced with parents
        Children = crossover_population(Parents)

        # Children are mutated
        Children = list(map(lambda child: mutate_individual(
            mutation_type, mutation_prob, child), Children))

        # Remainings are mutated
        Remainings = list(map(lambda remaining: mutate_individual(
            mutation_type, mutation_prob, remaining), Remainings))

        # Next Population is created with Elites, Children, and mutated remainings
        Next_Pop = dcopy(evaluate_population(Elites + Remainings + Children))
        Next_Pop = sorted(
            Next_Pop, key=lambda item: item.fitness, reverse=True)

        # Phenotypes are saved in every 1000 generation
        if generation % 1000 == 0:
            temp_img = Next_Pop[0].draw()
            cv2.imwrite(directory + EXP_NAME + "mut_prob" + str(mutation_prob) + "
                                                    _generation" +
                        str(generation)+".png", temp_img)

        # Fitness value is recorded
        if generation % curve_freq == 0:
            best_fitness = Next_Pop[0].fitness
            Fitness_Curve.append(best_fitness)

        # Next population is assigned to population
        Pop = dcopy(Next_Pop)

    plt.subplot(1, 2, 1)
    plt.plot(Fitness_Curve[0:10])
    plt.ylabel("Fitness score")
    plt.xlabel("x" + str(curve_freq) + " Generations")
    plt.title("Fitness of the best individual")

    plt.subplot(1, 2, 2)
    plt.plot(np.linspace(10, 100, len(Fitness_Curve[10:])), Fitness_Curve[10:])
    plt.ylabel("Fitness score")
    plt.xlabel("x" + str(curve_freq) + " Generations")
    plt.title("Fitness of the best individual")

plt.legend(legends)
plt.suptitle("Fitness Plot for Mutation Probability Experiment")
plt.savefig(directory + EXP_NAME + "Default_Params" + "_Fitness.png")
```

Scheduled Experiment Code

```
# Imports
import cv2
import numpy as np
from matplotlib import pyplot as plt
from random import randint, random, shuffle, uniform
from copy import deepcopy as dcopy
from multiprocessing import Pool

from numpy.lib.utils import source
source_img = cv2.imread("painting.png")
HEIGHT, WIDTH, CHANNELS = source_img.shape

# Helpers


def ramp(x):
    return max(0, x)


def create_blank_img():
    img = np.ones((HEIGHT, WIDTH, 3), np.uint8)
    img[:] = 255
    return dcopy(img)


def Pythagorean(x, y):
    return (x**2+y**2)**0.5


def create_empty_list(empty_list=None):
    if empty_list == None:
        empty_list = []
    return empty_list

# Classes


class Gene():
    global HEIGHT, WIDTH

    def __init__(self, x=0, y=0, r=0, R=0, G=0, B=0, A=0):
        self.x = x
        self.y = y
        self.r = r
        self.R = R
        self.G = G
        self.B = B
        self.A = A

    def random_initialize(self):
        self.x = randint(0, HEIGHT)
        self.y = randint(0, WIDTH)
```

```
        self.r = randint(15, 60)
        self.R = randint(0, 255)
        self.G = randint(0, 255)
        self.B = randint(0, 255)
        self.A = random()

    def check_if_in_image(self):
        if self.x < self.r or abs(self.x-WIDTH) < self.r or abs(self.y-HEIGHT) < self.r or self.
                                                y < self.r or (0 < self.x < WIDTH and 0 < self.y <
                                                 HEIGHT):
            return True
        else:
            return False

# Since an individual has only one chromosome, I avoided introducing extra classes and used
                                        genes directly under individiual.


class Ind():

    def __init__(self, num_genes=None, genes=None, fitness=0):
        # Population is initialized with random genes if genes argument is not given, until all
                                                the genes have the corresponding phenotype lying
                                                in the image.
        i = 0
        if genes is None:
            genes = create_empty_list()
            while i < num_genes:
                gene = Gene()
                gene.random_initialize()
                if gene.check_if_in_image():
                    genes.append(gene)
                    i += 1
        self.genes = genes
        self.calculate_fitness()

    def draw(self):
        # Individuals genes are sorted based on the radius, and regarding phenotypes are drawn
                                                over the blank image.
        sorted_genes = sorted(
            self.genes, key=lambda item: item.r, reverse=True)
        res_img = create_blank_img()
        for gene in sorted_genes:
            overlay = res_img
            cv2.circle(overlay, center=(gene.x, gene.y), radius=gene.r,
                       color=tuple(map(lambda item: item/1, (gene.B, gene.G, gene.R))),
                                                            thickness=-1)
            res_img = cv2.addWeighted(overlay, gene.A, res_img, 1-gene.A, 0)
        return res_img

    def calculate_fitness(self):
        # Return Frobenius Norm of the difference.
        fitness = -1 * np.linalg.norm(np.int64(source_img)-self.draw())**2
        self.fitness = fitness
```

```
def evaluate_population(population):
    # Calls fitness calculator on every individual of population.
    for ind in population:
        ind.calculate_fitness()
    return population


def create_population(num_individuals, num_genes):
    Population = create_empty_list()
    for _ in range(num_individuals):
        ind = Ind(num_genes)
        ind.calculate_fitness()
        Population.append(ind)
    return Population


def choose_n_elites(population, frac_of_elites):
    # Population is sorted in decreasing order and the n best individuals are returned.
    n = int(len(population) * frac_of_elites)
    sorted_pop = sorted(
        dcopy(population), key=lambda item: item.fitness, reverse=True)
    return sorted_pop[0:n], sorted_pop[n:]


def tournament_selection(Remaining, tm_size, parent_size):
    # First the population are shuffled to draw some number of random individuals, selected
                                        individuals are sorted and best of them is returned.
    Winners = create_empty_list()
    for _ in range(parent_size):
        shuffle(Remaining)
        Remaining = Remaining[0:tm_size]
        sorted_remains = sorted(
            Remaining, key=lambda item: item.fitness, reverse=True)
        Winners.append(dcopy(sorted_remains[0]))
    return Winners


def unguided_mutation(gene: Gene):
    gene.random_initialize()
    return gene


def guided_mutation(gene: Gene):

    # Applied the specified mutation rules to the genes, paying attention on the clipping and
                                        corrections.
    gene.x += randint(int(-0.25*gene.x), int(0.25*gene.x))
    gene.y += randint(int(-0.25*gene.y), int(0.25*gene.y))

    # Negative values are not allowed, hence used ramp
    gene.r = randint(ramp(gene.r-10), gene.r+10)
    gene.R = randint(ramp(gene.R-64), gene.R+64) if gene.R+64 < 255 else 255
    gene.G = randint(ramp(gene.G-64), gene.G+64) if gene.G+64 < 255 else 255
    gene.B = randint(ramp(gene.B-64), gene.B+64) if gene.B+64 < 255 else 255
```

```
    # [0,1]->[-1,1]->[-0.25,0.25]
    shifted_rn = uniform(-0.25, 0.25)
    gene.A += shifted_rn
    gene.A = gene.A if gene.A <= 1 and gene.A >= 0 else (
        0 if gene.A < 0 else 1)
    return gene


def mutate_individual(mutation, mutation_prob, ind: Ind):

    # Randomly selects a gene and mutates it if the mutation probability allows
    if random() < mutation_prob:
        genes = ind.genes
        gene_idx = randint(0, num_genes-1)
        if mutation.lower() == "guided":
            gene = guided_mutation(genes[gene_idx])
        else:
            gene = unguided_mutation(genes[gene_idx])
        ind.genes[gene_idx] = gene
    return ind


def draw_Pop(ind_list):
    # Creates the phenotype of the population
    genes = create_empty_list()
    for ind in ind_list:
        genes = genes + ind.genes
    IND = Ind(len(genes), genes=genes)
    img = IND.draw()
    return img


def crossover(parents: list):

    # Swaps genes of the parents with a certain probability, 0.5 here
    p1_genes = parents[0].genes
    p2_genes = parents[1].genes

    # initialize children genes
    ch1 = Ind(genes=p1_genes)
    ch2 = Ind(genes=p2_genes)

    # Exchange of genes with probability 0.5
    for i in range(len(ch1.genes)):
        if random() > 0.5:
            ch1.genes[i], ch2.genes[i] = dcopy(
                ch1.genes[i]), dcopy(ch2.genes[i])

    return [ch1, ch2]


def crossover_population(Parents: list):
    # Returns children of the parent population
    child_list = [crossover([Parents[2*i], Parents[2*i+1]])
```

```
                for i in range((len(Parents)//2))]
    children = [x for l in child_list for x in l]
    return children


def round_to_even(a: int):
    if int(a) % 2 == 0:
        return int(a)
    else:
        return int(a)-1


# Set params
frac_of_elites = 0.2
num_individuals = 20
num_genes = 50
frac_parents = 0.6
mutation_prob = 0.2
mutation_type = "unguided"
tm_size = 5
generations = 10000

curve_freq = 100
directory = "OUTPUTs//INCREASINGELITES//"
EXP_NAME = "increasing_elites"

plt.figure(figsize=(15, 10))

fracs_of_elites = [0.05]*4000 + [0.1]*3000 + [0.2]*3001

#####################          EVOLUTION           ##########################

# First create population
Pop = create_population(
    num_individuals=num_individuals, num_genes=num_genes)

# Fitness Curve
Fitness_Curve = create_empty_list()

for generation in range(generations+1):

    frac_of_elites = fracs_of_elites[generation]

    Pop = evaluate_population(Pop)

    Elites, NonElites = choose_n_elites(Pop, frac_of_elites)
    R = len(NonElites)

    num_parents = int(num_individuals*frac_parents)
    to_be_mutated = R - num_parents

    Winners = tournament_selection(
        NonElites, tm_size, R)
```

```
    # To first explore, then exploit the space, the only variation mechanism in the experiments,
                                        mutation, can be altered thorough generations.



    # Parents are selected among the Non Elite Population
    Parents, Remainings = Winners[0:num_parents], Winners[-1*to_be_mutated:]

    # Children are produced with parents
    Children = crossover_population(Parents)

    # Children are mutated
    Children = list(map(lambda child: mutate_individual(
        mutation_type, mutation_prob, child), Children))

    # Remainings are mutated
    Remainings = list(map(lambda remaining: mutate_individual(
        mutation_type, mutation_prob, remaining), Remainings))

    # Next Population is created with Elites, Children, and mutated remainings
    Next_Pop = dcopy(evaluate_population(Elites + Remainings + Children))
    Next_Pop = sorted(
        Next_Pop, key=lambda item: item.fitness, reverse=True)

    # Phenotypes are saved in every 1000 generation
    if generation % 1000 == 0:
        temp_img = Next_Pop[0].draw()
        cv2.imwrite(directory + EXP_NAME + "_generation" +
                    str(generation)+".png", temp_img)

    # Fitness value is recorded
    if generation % curve_freq == 0:
        best_fitness = Next_Pop[0].fitness
        Fitness_Curve.append(best_fitness)

    # Next population is assigned to population
    Pop = dcopy(Next_Pop)

plt.subplot(1, 2, 1)
plt.plot(Fitness_Curve[0:10])
plt.ylabel("Fitness score")
plt.xlabel("x" + str(curve_freq) + " Generations")
plt.title("Fitness of the best individual")

plt.subplot(1, 2, 2)
plt.plot(np.linspace(10, 100, len(Fitness_Curve[10:])), Fitness_Curve[10:])
plt.ylabel("Fitness score")
plt.xlabel("x" + str(curve_freq) + " Generations")
plt.title("Fitness of the best individual")

plt.suptitle("Fitness Plot for Increasing Fraction of Elites Experiment")
plt.savefig(directory + EXP_NAME + "" + "_Fitness.png")
```

Visualization

```
    import numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec
import cv2

##########################################
# SET PARAMS HERE TO PLOT THE EXPERIMENT
nrow = 11
ncol = 1
fig = plt.figure(figsize=(15, 10))
DIRECTORY = "INCREASINGELITES\\"
EXP_NAME = " Experiment with Increasing Fraction of Elites"
exp_prefix = "increasing_elites"
legends = [""]
fig_name = "increaselites"
##########################################

gs = gridspec.GridSpec(nrow, ncol, height_ratios=[1]*11,
                       wspace=0.0, hspace=0.0, top=0.95, bottom=0.05, left=0.17, right=0.845)


for j in range(ncol):
    for i in range(nrow):
        file_path = "OUTPUTs\\"+DIRECTORY+exp_prefix + \
            str(legends[j])+"_generation"+str(1000*i)+".png"
        print(file_path)
        im = cv2.imread(file_path)
        im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
        ax = plt.subplot(gs[i, j])
        plt.ylabel("Iteration\n"+str(i*1000))
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.imshow(im)
        if i == nrow-1:
            plt.xlabel(exp_prefix + " = " + str(legends[j]))

fig = plt.gcf()
fig.suptitle(EXP_NAME, fontsize=14)

plt.savefig("OUTPUTs\\FINAL_FIGS\\"+fig_name+".png")
```