Ahmet Akman 2442366
Middle East Technical University
Electrical and Electronics Engineering

April 28, 2024

HOMEWORK 2 — Report

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
```

```
141
142         for i, gene in enumerate(self.chromosome):
143             #check if the circle is visible in the image, center does not have to be in
    the image but the circle should be visible
144             while True:
145                 if not self.is_visible(gene):
146                     # randomly initialize the gene and check again
147                     gene[0] = random.randint(-2*self.radius_max, self.image_size[0]+2*
    self.radius_max)
148                     gene[1] = random.randint(-2*self.radius_max, self.image_size[1]+2*
    self.radius_max)
149                     gene[2] = random.randint(0, self.radius_max*2)
150                 else:
151                     break
152             self.chromosome[i] = gene
153             overlay = image.copy()
154             cv2.circle(overlay, (gene[0], gene[1]), gene[2], gene[3], -1)
155             image = cv2.addWeighted(overlay, gene[4], image, 1 - gene[4], 0)
156         return image
157
158     def calculate_fitness(self, target):
159         image = self.draw()
160         # Calculate the difference between the target image and the generated image
161         target_np = np.array(target, dtype=np.int64)
162         image_np = np.array(image, dtype=np.int64)
163         diff = np.subtract(target_np, image_np)
164         # take the square of the difference
165         diff = np.square(diff)
166         # sum of the squared differences
167         # print(np.sum(diff))
168         self.fitness = -np.sum(diff)
169
170     def crossover(self, partner):
171         child1 = Individual(self.num_genes, self.image_size)
172         child2 = Individual(self.num_genes, self.image_size)
173
174         for i in range(self.num_genes):
175             if random.random() < 0.5:
176                 child1.chromosome[i] = copy.deepcopy(self.chromosome[i])
177                 child2.chromosome[i] = copy.deepcopy(partner.chromosome[i])
178             else:
179                 child1.chromosome[i] = copy.deepcopy(partner.chromosome[i])
180                 child2.chromosome[i] = copy.deepcopy(self.chromosome[i])
181         return child1, child2
182
183 # Popoulation class definition
184
185 class Population:
186     def __init__(self, num_individuals, num_genes, image_size, frac_elites, frac_parents,
     tm_size, target_image, guidance):
187         self.num_individuals = num_individuals
188         self.num_genes = num_genes
189         self.image_size = image_size
190         self.num_elites = int(frac_elites*self.num_individuals)
191         self.num_parents = int(frac_parents*self.num_individuals)
192         if self.num_parents % 2 != 0:
193             self.num_parents += 1
194         self.tm_size = tm_size
195         self.guidance = guidance
```

```
196
197         self.target = target_image
198         self.individuals = []
199         self.parents = []
200
201         for i in range(self.num_individuals):
202             self.individuals.append(Individual(self.num_genes, self.image_size))
203
204     def selection(self):
205
206         self.individuals.sort(key=lambda x: x.fitness, reverse=True)
207
208         new_individuals = []
209         # Mark the best individuals as elite
210         for i in range(self.num_elites):
211             self.individuals[i].elite = True
212             new_individuals.append(self.individuals[i])
213         # Select the rest of the individuals using tournament selection.
214         non_elite_individuals = self.individuals[self.num_elites:]
215
216         parentable_individuals = []
217
218         # update the non elite group by adding the best individual from each group to
    parentable individuals
219
220         for i in range(len(non_elite_individuals)):
221             group = random.sample(non_elite_individuals, min(self.tm_size, len(
    non_elite_individuals)))
222
223             group.sort(key=lambda x: x.fitness, reverse=True)
224
225             parentable_individuals.append(copy.deepcopy(group[0]))
226
227         # select the best parents from the parentable individuals
228         parentable_individuals.sort(key=lambda x: x.fitness, reverse=True)
229         self.parents = parentable_individuals[:self.num_parents]
230         # 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
```

```
        self.individuals.extend(children)

    def mutation(self, mutation_probability):
        #check if the individual is an elite, if so do not mutate
        for individual in self.individuals:
            individual.mutate(mutation_probability, guidance = self.guidance)

    def evaluate(self):
        for individual in self.individuals:
            individual.calculate_fitness(self.target)

    def get_best(self):
        self.individuals.sort(key=lambda x: x.fitness, reverse=True)
        return self.individuals[0]

    def get_average_fitness(self):
        return sum([x.fitness for x in self.individuals]) / self.num_individuals




# default parameters in dictionary
parameters_list = [{
    "num_individuals": 20,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "mutation_probability": 0.2,
    "mutataion_type": "guided"
}, {
    "num_individuals": 5,
    "num_genes": 50,
    "tournament_size": 5,
    "frac_elites": 0.2,
    "frac_parents": 0.6,
    "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 = "output_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")
```

Submitted by Ahmet Akman 2442366 on April 28, 2024.