

HOMEWORK 2 — Report

Appendix

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

```
1 # # Homework 2 Evolutionary Algorithms.
2 # 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)"
3 #
4 # ## Pseudo code
5 # Initialize population with <num_inds> individuals each having <num_genes> genes
6 # While not all generations (<num_generations>) are computed:
7 # Evaluate all the individuals
8 # Select individuals
9 # Do crossover on some individuals
10 # Mutate some individuals
11
12 # "Individual" class definition for evolutionary algorithm. There will be on chromosome
   and N number of genes. Each gene will have center coordinates (x,y), Radius, and RGB
   color.
13
14 import random
15 import numpy as np
16 import cv2
17 import copy
18 import h5py
19 from tqdm import tqdm
20
21
22 # Individual class definition
23 class Individual:
24     def __init__(self, num_genes, image_size):
25         self.num_genes = num_genes
26         self.image_size = image_size
27         self.chromosome = []
28         self.fitness = np.float128(0.0)
29         self.elite = False
30         self.radius_max = image_size[0]//4
31
32         for i in range(num_genes):
33             x = random.randint(-1*self.radius_max, self.image_size[0]+1*self.radius_max)
34             y = random.randint(-1*self.radius_max, self.image_size[1]+1*self.radius_max)
35             r = random.randint(1, self.radius_max)
36             color = [random.randint(0, 255), random.randint(0, 255), random.randint(0,
37                 255)]
38             alpha = random.random()
```

```

39         self.chromosome.append([x, y, r, color, alpha])
40
41
42     def is_visible(self, gene=None):
43         x = gene[0]
44         y = gene[1]
45         r = gene[2]
46         if ((x + r) < 0) or (x > self.image_size[0]+r) or ((y + r) < 0) or (y > self.
image_size[1]+r):
47             return False
48         else:
49             return True
50     def mutate(self, mutation_probability, guidance = None):
51         if not self.elite:
52             if random.random() < mutation_probability:
53                 while True:
54                     i = random.randint(0, self.num_genes-1)
55
56                     if guidance is "unguided":
57                         while True:
58                             if not self.is_visible(self.chromosome[i]):
59                                 # randomly initialize the gene and check again
60                                 self.chromosome[i][0] = random.randint(-2*self.radius_max
, self.image_size[0]+2*self.radius_max)
61                                 self.chromosome[i][1] = random.randint(-2*self.radius_max
, self.image_size[1]+2*self.radius_max)
62                                 self.chromosome[i][2] = random.randint(0, self.radius_max
*2)
63
64                                 break
65
66                                 self.chromosome[i][3][0] = random.randint(0, 255)
67                                 self.chromosome[i][3][1] = random.randint(0, 255)
68                                 self.chromosome[i][3][2] = random.randint(0, 255)
69
70                                 self.chromosome[i][4] = random.random()
71         else:
72             #Guided mutation, deviate x,y, radius, color and alpha around
their previous values
73
74             temp_x = copy.deepcopy(self.chromosome[i][0])
75             temp_y = copy.deepcopy(self.chromosome[i][1])
76             temp_r = copy.deepcopy(self.chromosome[i][2])
77
78             # mutate under the condition of visibility again and again until
the gene is visible
79             while True:
80                 self.chromosome[i][0] = int(temp_x + (self.image_size[0]/4)*
random.uniform(-1,1))
81                 self.chromosome[i][1] = int(temp_y + (self.image_size[1]/4)*
random.uniform(-1,1))
82
83                 if self.chromosome[i][0] < -2*self.radius_max:
84                     self.chromosome[i][0] = -2*self.radius_max
85                 elif self.chromosome[i][0] > self.image_size[0]+2*self.
radius_max:
86                     self.chromosome[i][0] = self.image_size[0]+2*self.
radius_max
87

```

```

88         if self.chromosome[i][1] < -2*self.radius_max:
89             self.chromosome[i][1] = -2*self.radius_max
90         elif self.chromosome[i][1] > self.image_size[1]+2*self.
radius_max:
91             self.chromosome[i][1] = self.image_size[1]+2*self.
radius_max
92
93         self.chromosome[i][2] = int(temp_r + 10*random.uniform(-1,1))
94         if self.chromosome[i][2] < 0:
95             self.chromosome[i][2] = 1
96         elif self.chromosome[i][2] > self.radius_max*2:
97             self.chromosome[i][2] = self.radius_max*2
98
99         if self.is_visible(self.chromosome[i]):
100             break
101
102         self.chromosome[i][3][0] = int(self.chromosome[i][3][0] + 64*
random.uniform(-1, 1))
103         if self.chromosome[i][3][0] < 0:
104             self.chromosome[i][3][0] = 0
105         elif self.chromosome[i][3][0] > 255:
106             self.chromosome[i][3][0] = 255
107
108         self.chromosome[i][3][1] = int(self.chromosome[i][3][1] + 64*
random.uniform(-1, 1))
109         if self.chromosome[i][3][1] < 0:
110             self.chromosome[i][3][1] = 0
111         elif self.chromosome[i][3][1] > 255:
112             self.chromosome[i][3][1] = 255
113
114         self.chromosome[i][3][2] = int(self.chromosome[i][3][2] + 64*
random.uniform(-1, 1))
115         if self.chromosome[i][3][2] < 0:
116             self.chromosome[i][3][2] = 0
117         elif self.chromosome[i][3][2] > 255:
118             self.chromosome[i][3][2] = 255
119
120         self.chromosome[i][3] = [int(x) for x in self.chromosome[i][3]]
121
122         self.chromosome[i][4] = self.chromosome[i][4] + 0.25*random.
uniform(-1, 1)
123         if self.chromosome[i][4] < 0:
124             self.chromosome[i][4] = 0.001
125         elif self.chromosome[i][4] > 1:
126             self.chromosome[i][4] = 1
127
128         if random.random() > mutation_probability:
129             break
130     else:
131         pass
132     else:
133         # print("Cannot mutate elite individual")
134         pass
135 def draw(self):
136     # First sort the genes by radius
137     self.chromosome.sort(key=lambda x: x[2], reverse=True)
138
139     # Create a blank image white background
140     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     self.target = target_image
197     self.individuals = []
198     self.parents = []
199
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
219         # parentable individuals
220
221         for i in range(len(non_elite_individuals)):
222             group = random.sample(non_elite_individuals, min(self.tm_size, len(
223                 non_elite_individuals)))
224
225             group.sort(key=lambda x: x.fitness, reverse=True)
226
227             parentable_individuals.append(copy.deepcopy(group[0]))
228
229         # select the best parents from the parentable individuals
230         parentable_individuals.sort(key=lambda x: x.fitness, reverse=True)
231         self.parents = parentable_individuals[:self.num_parents]
232         # non elite non parent elements will be added to the new individuals
233         new_individuals.extend(parentable_individuals[self.num_parents:])
234         # The new generation except the children will be the new individuals
235         self.individuals = new_individuals
236
237     def crossover(self):
238         # parents will create new individuals by crossover. Two parents will create two
239         # children
240         children = []
241         random.shuffle(self.parents)
242
243         for i in range(0, self.num_parents, 2):
244             parent1 = self.parents[i]
245             parent2 = self.parents[i+1]
246
247             child1, child2 = parent1.crossover(parent2)
248
249             child1.calculate_fitness(self.target)
250             child2.calculate_fitness(self.target)
251
252             children.append(child1)
253             children.append(child2)

```

```

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,
289     "mutataion_type": "guided"
290 }, {
291     "num_individuals": 10,
292     "num_genes": 50,
293     "tournament_size": 5,
294     "frac_elites": 0.2,
295     "frac_parents": 0.6,
296     "mutation_probability": 0.2,
297     "mutataion_type": "guided"
298 }, {
299     "num_individuals": 40,
300     "num_genes": 50,
301     "tournament_size": 5,
302     "frac_elites": 0.2,
303     "frac_parents": 0.6,
304     "mutation_probability": 0.2,
305     "mutataion_type": "guided"
306 }, {
307     "num_individuals": 60,
308     "num_genes": 50,
309     "tournament_size": 5,
310     "frac_elites": 0.2,

```

```

311     "frac_parents": 0.6,
312     "mutation_probability": 0.2,
313     "mutataion_type": "guided"
314 }, {
315     "num_individuals": 20,
316     "num_genes": 15,
317     "tournament_size": 5,
318     "frac_elites": 0.2,
319     "frac_parents": 0.6,
320     "mutation_probability": 0.2,
321     "mutataion_type": "guided"
322 }, {
323     "num_individuals": 20,
324     "num_genes": 30,
325     "tournament_size": 5,
326     "frac_elites": 0.2,
327     "frac_parents": 0.6,
328     "mutation_probability": 0.2,
329     "mutataion_type": "guided"
330 }, {
331     "num_individuals": 20,
332     "num_genes": 80,
333     "tournament_size": 5,
334     "frac_elites": 0.2,
335     "frac_parents": 0.6,
336     "mutation_probability": 0.2,
337     "mutataion_type": "guided"
338 }, {
339     "num_individuals": 20,
340     "num_genes": 120,
341     "tournament_size": 5,
342     "frac_elites": 0.2,
343     "frac_parents": 0.6,
344     "mutation_probability": 0.2,
345     "mutataion_type": "guided"
346 }, {
347     "num_individuals": 20,
348     "num_genes": 50,
349     "tournament_size": 2,
350     "frac_elites": 0.2,
351     "frac_parents": 0.6,
352     "mutation_probability": 0.2,
353     "mutataion_type": "guided"
354 }, {
355     "num_individuals": 20,
356     "num_genes": 50,
357     "tournament_size": 8,
358     "frac_elites": 0.2,
359     "frac_parents": 0.6,
360     "mutation_probability": 0.2,
361     "mutataion_type": "guided"
362 }, {
363     "num_individuals": 20,
364     "num_genes": 50,
365     "tournament_size": 16,
366     "frac_elites": 0.2,
367     "frac_parents": 0.6,
368     "mutation_probability": 0.2,
369     "mutataion_type": "guided"

```

```

370 }, {
371     "num_individuals": 20,
372     "num_genes": 50,
373     "tournament_size": 5,
374     "frac_elites": 0.04,
375     "frac_parents": 0.6,
376     "mutation_probability": 0.2,
377     "mutataion_type": "guided"
378 }, {
379     "num_individuals": 20,
380     "num_genes": 50,
381     "tournament_size": 5,
382     "frac_elites": 0.35,
383     "frac_parents": 0.6,
384     "mutation_probability": 0.2,
385     "mutataion_type": "guided"
386 }, {
387     "num_individuals": 20,
388     "num_genes": 50,
389     "tournament_size": 5,
390     "frac_elites": 0.2,
391     "frac_parents": 0.15,
392     "mutation_probability": 0.2,
393     "mutataion_type": "guided"
394 }, {
395     "num_individuals": 20,
396     "num_genes": 50,
397     "tournament_size": 5,
398     "frac_elites": 0.2,
399     "frac_parents": 0.3,
400     "mutation_probability": 0.2,
401     "mutataion_type": "guided"
402 }, {
403     "num_individuals": 20,
404     "num_genes": 50,
405     "tournament_size": 5,
406     "frac_elites": 0.2,
407     "frac_parents": 0.75,
408     "mutation_probability": 0.2,
409     "mutataion_type": "guided"
410 }, {
411     "num_individuals": 20,
412     "num_genes": 50,
413     "tournament_size": 5,
414     "frac_elites": 0.2,
415     "frac_parents": 0.6,
416     "mutation_probability": 0.1,
417     "mutataion_type": "guided"
418 }, {
419     "num_individuals": 20,
420     "num_genes": 50,
421     "tournament_size": 5,
422     "frac_elites": 0.2,
423     "frac_parents": 0.6,
424     "mutation_probability": 0.4,
425     "mutataion_type": "guided"
426 }, {
427     "num_individuals": 20,
428     "num_genes": 50,

```



```

429     "tournament_size": 5,
430     "frac_elites": 0.2,
431     "frac_parents": 0.6,
432     "mutation_probability": 0.75,
433     "mutataion_type": "guided"
434 },{
435     "num_individuals": 20,
436     "num_genes": 50,
437     "tournament_size": 5,
438     "frac_elites": 0.2,
439     "frac_parents": 0.6,
440     "mutation_probability": 0.2,
441     "mutataion_type": "unguided"
442 }
443 ]
444
445 for i, parameters in enumerate(parameters_list):
446     num_individuals = parameters["num_individuals"]
447     num_genes = parameters["num_genes"]
448     tournament_size = parameters["tournament_size"]
449     frac_elites = parameters["frac_elites"]
450     frac_parents = parameters["frac_parents"]
451     mutation_probability = parameters["mutation_probability"]
452     mutataion_type = parameters["mutataion_type"]
453
454     print("Parameters for run ", i+1)
455
456     print("Summary of parameters")
457     print("Number of individuals: ", num_individuals)
458     print("Number of genes per individual: ", num_genes)
459     print("Tournament size: ", tournament_size)
460     print("Fraction of elites: ", frac_elites)
461     print("Fraction of parents: ", frac_parents)
462     print("Mutation probability: ", mutation_probability)
463     print("Mutation type: ", mutataion_type)
464
465     # load input image
466     target = cv2.imread("hw2/painting.png")
467
468     image_size = (target.shape[1], target.shape[0])
469
470
471     pop = Population(num_individuals, num_genes, image_size, frac_elites, frac_parents,
472                     tournament_size, target, guidance=mutataion_type)
473
474     # iterate over generations
475     average_fitness = []
476     best_fitness = []
477     image_of_best = []
478
479     num_generations = 10000
480
481     for i in tqdm(range(num_generations)):
482         pop.evaluate()
483         pop.selection()
484         pop.crossover()
485         pop.mutation(mutation_probability)
486         pop.evaluate()

```

```

487         #reset elite status
488         for individual in pop.individuals:
489             individual.elite = False
490         # print("num individuals left",len(pop.individuals))
491         average_fitness.append(pop.get_average_fitness())
492         best_fitness.append(pop.get_best().fitness)
493         if (i+1) % 1000 == 0:
494             image_of_best.append(pop.get_best().draw())
495         print("Generation: ", i, "Average fitness: ", pop.get_average_fitness(), "Best
496         fitness: ", pop.get_best().fitness)
497
498         # set filename using the parameters
499
500         filename = "hw2/out_sweep/output_" + str(num_individuals) + "_" + str(num_genes) + "_"
501         + str(tournament_size) + "_" + str(frac_elites) + "_" + str(frac_parents) + "_" +
502         str(mutation_probability) + "_" + mutataion_type + ".h5"
503
504         # save images and average best fitness values to a file
505         with h5py.File(filename, "w") as f:
506             f.create_dataset("average_fitness", data=average_fitness)
507             f.create_dataset("best_fitness", data=best_fitness)
508             for i, image in enumerate(image_of_best):
509                 f.create_dataset("image_"+str(i-1000), data=image)
510
511
512 # this file will be used to analyze the data from the h5py data files located in the data
513 folder
514
515 import h5py
516 import numpy as np
517
518 import matplotlib.pyplot as plt
519 import matplotlib.colors as colors
520 import matplotlib.cm as cm
521 import matplotlib.patches as patches
522 import matplotlib.gridspec as gridspec
523 import matplotlib.ticker as ticker
524 import matplotlib.colors as colors
525
526 import os
527 import cv2
528
529 # function to read the data from the h5py file
530 def read_data(file_name):
531     # open the file
532     file = h5py.File(file_name, 'r')
533     # get the data called "average_fitness"
534     average_fitness = file['average_fitness'][:]
535     # get the data called "best_fitness"
536     best_fitness = file['best_fitness'][:]
537     # get the data called "image_-1000"
538     image_0 = file['image_-1000'][:]
539     # get the data called "image_-999"
540     image_1 = file['image_-999'][:]
541     # get the data called "image_-998"
542     image_2 = file['image_-998'][:]
543     # get the data called "image_-997"
544     image_3 = file['image_-997'][:]
545     # get the data called "image_-996"
546     image_4 = file['image_-996'][:]

```

```

35     # get the data called "image_-995"
36     image_5 = file['image_-995'][:,]
37     # get the data called "image_-994"
38     image_6 = file['image_-994'][:,]
39     # get the data called "image_-993"
40     image_7 = file['image_-993'][:,]
41     # get the data called "image_-992"
42     image_8 = file['image_-992'][:,]
43     # get the data called "image_-991"
44     image_9 = file['image_-991'][:,]
45     # put them into the data array
46     data = [image_0, image_1, image_2, image_3, image_4, image_5, image_6, image_7,
47             image_8, image_9]
48
49     # return the data and labels
50     return data, average_fitness, best_fitness
51
52 # get list of files with h5 extension in the data folder
53 data_folder = "hw2/out_sweep/" #
54 files = os.listdir(data_folder)
55 files = [file for file in files if file.endswith(".h5")]
56
57 # loop through the files and read the data
58 for file in files:
59     # file = "output_20_50_5_0.2_0.6_0.2_guided.h5"
60     data, average_fitness, best_fitness = read_data(data_folder + "/" + file)
61     # get the parameter values from the file name
62     parts = file.split("_")
63
64     # get the population size
65     num_individuals = parts[1]
66     # get the number of genes
67     num_genes = parts[2]
68     # get the tournament size
69     tournament_size = parts[3]
70     # get the fraction of elites
71     fraction_elites = parts[4]
72     # get the fraction of parents
73     fraction_parents = parts[5]
74     # get the mutation probability
75     mutation_probability = parts[6]
76     # get the mutation type
77     mutation_type = parts[7].split(".")[0]
78
79     # create the text info
80     text_info = "Population Size: " + num_individuals + "\nNumber of Genes: " + num_genes
81     + "\nTournament Size: " + tournament_size + "\nFraction of Elites: " +
82     fraction_elites + "\nFraction of Parents: " + fraction_parents + "\nMutation
83     Probability: " + mutation_probability + "\nMutation Type: " + mutation_type
84
85     # plot the average fitness
86     # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
87     fig = plt.figure(figsize=(8.3, 5.8))
88     gs = gridspec.GridSpec(1, 1)
89     plt.rcParams["font.family"] = "sans-serif"
90     plt.rcParams["font.sans-serif"] = ["DeJaVu Sans"]
91     ax0 = plt.subplot(gs[0, 0])
92     ax0.plot(average_fitness, linestyle='-', color='#F8670', linewidth=3.0, alpha=1)
93     ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)

```

```

90 ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
91 ax0.grid()
92 ax0.set_title('Average Fitness', fontsize=20, pad=20)
93 ax0.tick_params(axis='both', which='major', labelsize=16)
94 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'))
95 plt.tight_layout()
96 #plt.show()
97 # save the figure
98 fig.savefig("hw2/out_sweep/average_fitness_" + file.split(".h")[0] + ".png")
99
100 # plot the best fitness
101 # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
102 fig = plt.figure(figsize=(8.3, 5.8))
103 gs = gridspec.GridSpec(1, 1)
104 plt.rcParams["font.family"] = "sans-serif"
105 plt.rcParams["font.sans-serif"] = ["DejaVu Sans"]
106 ax0 = plt.subplot(gs[0, 0])
107 ax0.plot(best_fitness, linestyle='-', color='#B80000', linewidth=3.0, alpha=1)
108 ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)
109 ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
110 ax0.grid()
111 ax0.set_title('Best Fitness', fontsize=20, pad=20)
112 ax0.tick_params(axis='both', which='major', labelsize=16)
113 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'))
114 plt.tight_layout()
115 # plt.show()
116 fig.savefig("hw2/out_sweep/best_fitness_" + file.split(".h")[0] + ".png")
117
118
119 # plot the best fitness
120 # Plot the data on A5 paper use Arial font. Use 4:3 aspect ratio.
121 fig = plt.figure(figsize=(8.3, 5.8))
122 gs = gridspec.GridSpec(1, 1)
123 plt.rcParams["font.family"] = "sans-serif"
124 plt.rcParams["font.sans-serif"] = ["DejaVu Sans"]
125 ax0 = plt.subplot(gs[0, 0])
126 ax0.plot(range(1000, 10000), best_fitness[1000:], linestyle='-', color='#7570b3',
linewidth=3.0, alpha=1)
127 ax0.set_xlabel('# Generations', fontsize=18, labelpad=10)
128 ax0.set_ylabel('Fitness', fontsize=18, labelpad=10)
129 ax0.grid()
130 ax0.set_title('Best Fitness from 1000th Gen', fontsize=20, pad=20)
131 ax0.tick_params(axis='both', which='major', labelsize=16)
132 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'))
133 plt.tight_layout()
134 # plt.show()
135 fig.savefig("hw2/out_sweep/best_fitness_1000_" + file.split(".h")[0] + ".png")
136
137
138 # plot the images for the 10 generations

```

```

139 fig, axs = plt.subplots(2, 5, figsize=(8.3, 5.8))
140 fig.suptitle("Evolution of Images", fontsize=20)
141 for i in range(10):
142     ax = axs[i//5, i%5]
143     # convert the image BGR to RGB
144     data[i] = cv2.cvtColor(data[i], cv2.COLOR_BGR2RGB)
145
146     ax.imshow(data[i])
147     ax.set_title("#Gen" + str((i+1)*1000), fontsize=12, fontweight='bold')
148     ax.axis('off')
149 # add text to the figure
150 fig.text(0.9, 0.845, text_info, ha='right', fontsize=8, fontweight='bold', color='
black', alpha=0.75)
151 fig.tight_layout(rect=[0, 0.03, 1, 0.95])
152 # plt.show()
153 fig.savefig("hw2/out_sweep/images_" + file.split(".h")[0] + ".png")
154
155 print("Done")

```

Submitted by Ahmet Akman 2442366 on April 28, 2024.