EE496 Homework 2 - Evolutionary Algorithms

Berken Utku Demirel - 2166221

March 31, 2020

Experimental Work

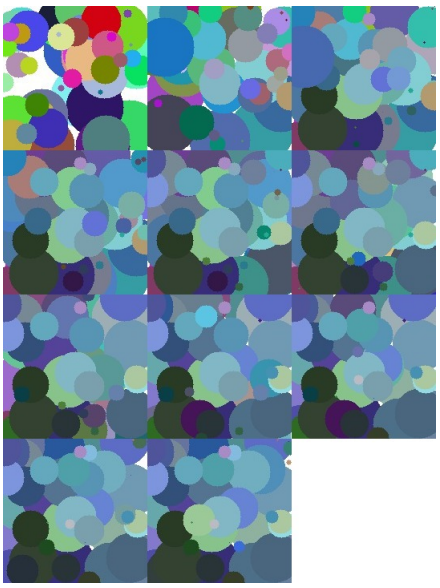


Figure 1: The best of individuals for the **default** parameters

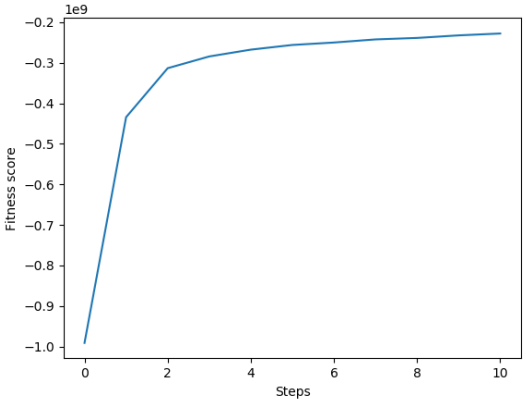
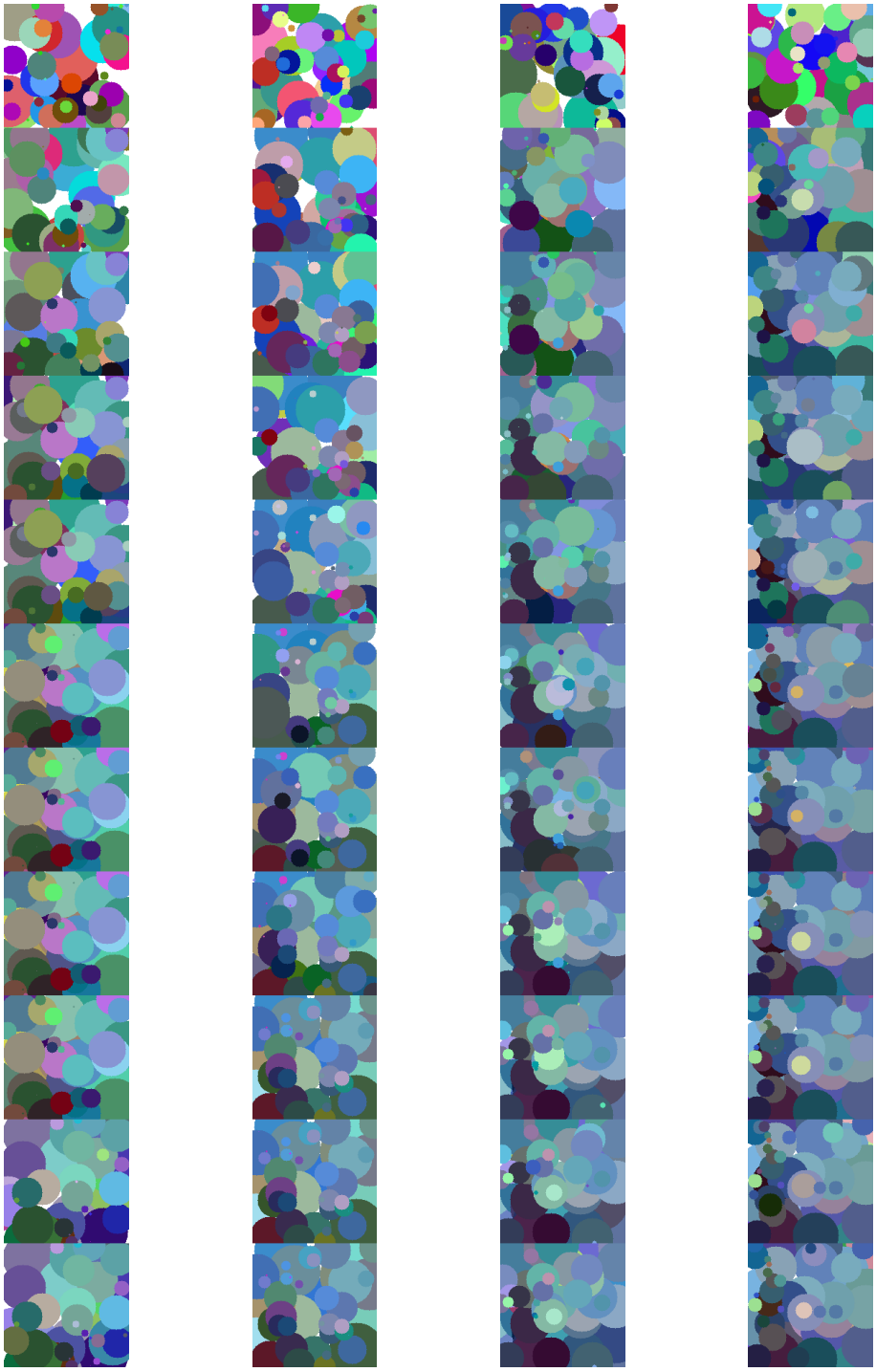
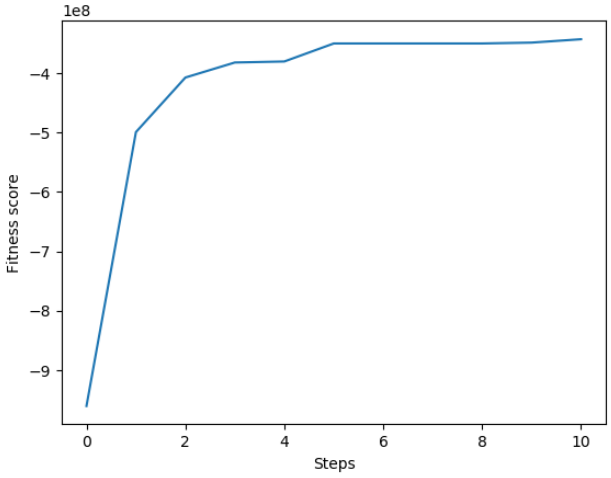


Figure 2: The fitness curve for the **default** parameters

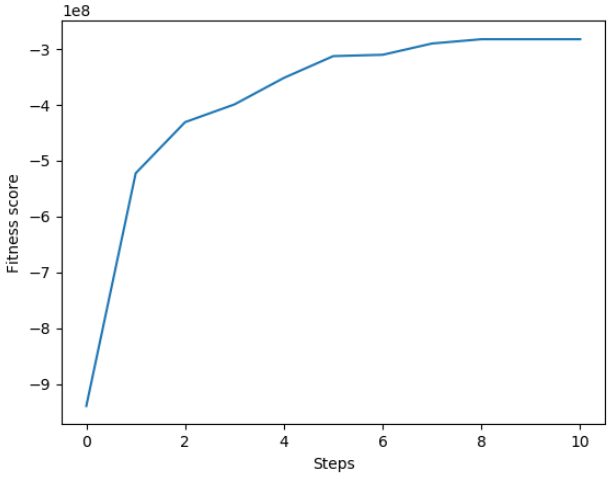


(a) num_inds=5 (b) num_inds=10 (c) num_inds=50 (d) num_inds=75

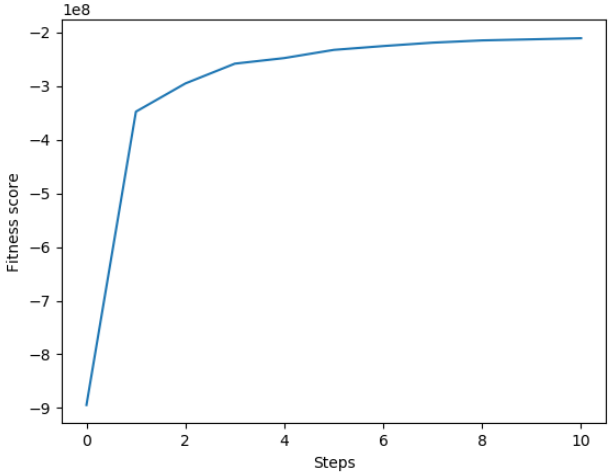
Figure 3: The image of the best individual in the population at every 1000th generation for (<num_inds>)=5,10,50,75



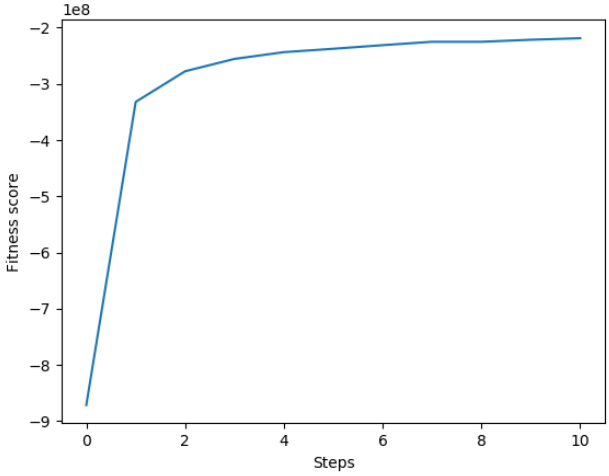
(a) num_inds=5



(b) num_inds=10

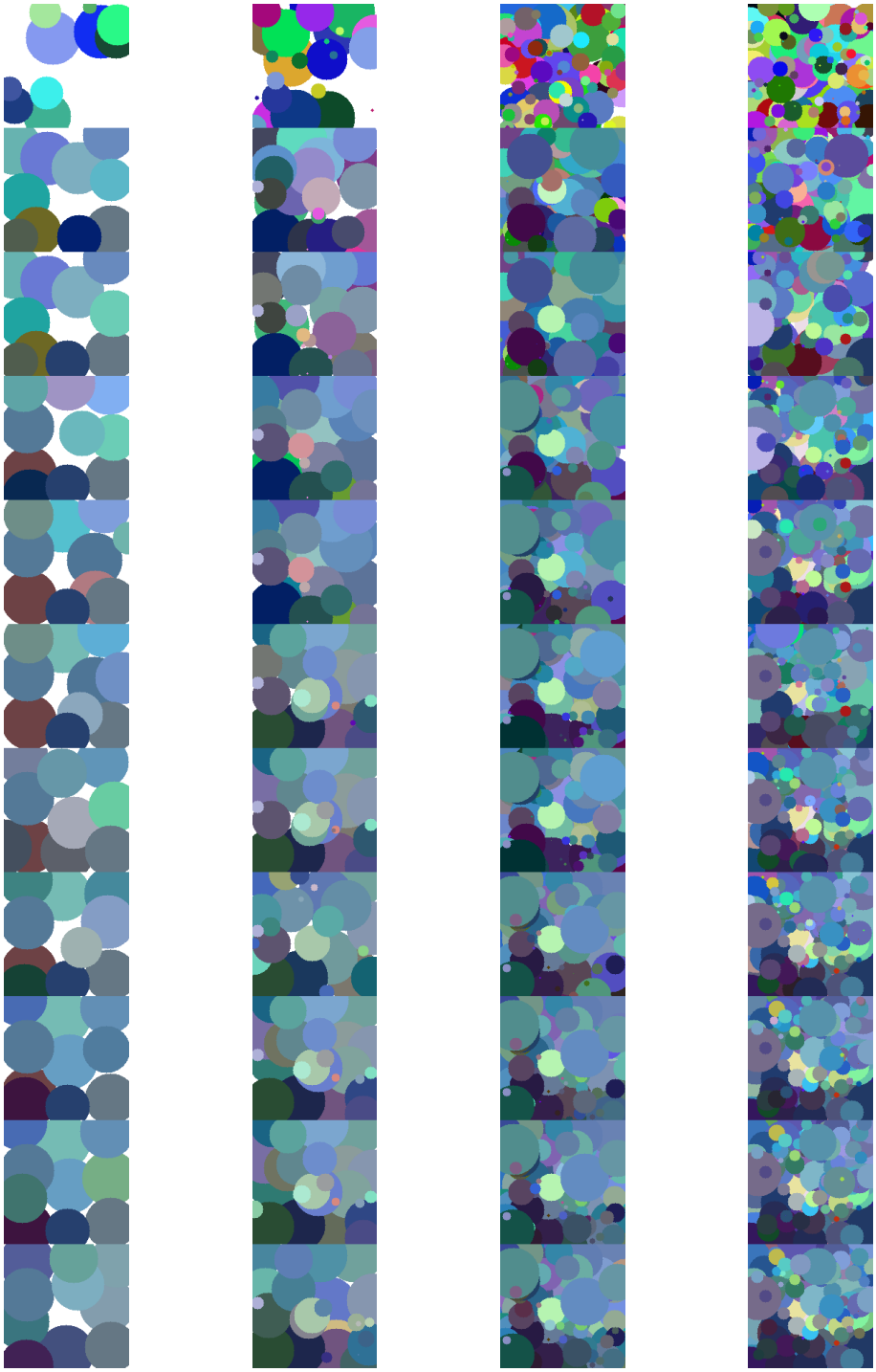


(c) num_inds=50



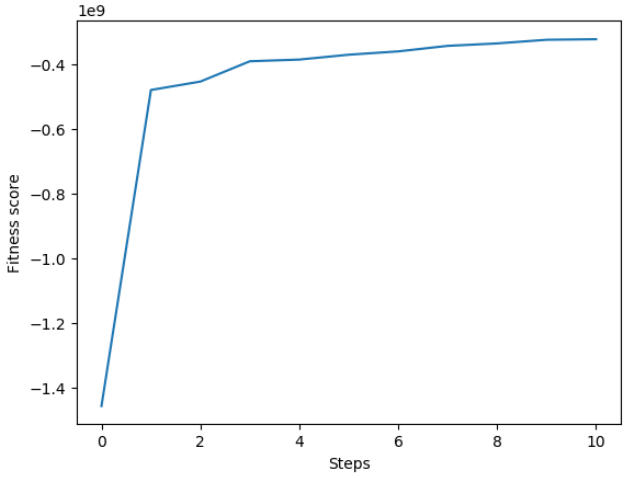
(d) num_inds=75

Figure 4: The fitness plot from generation 1 to generation 10000 for (<num_inds>)=5,10,50,75

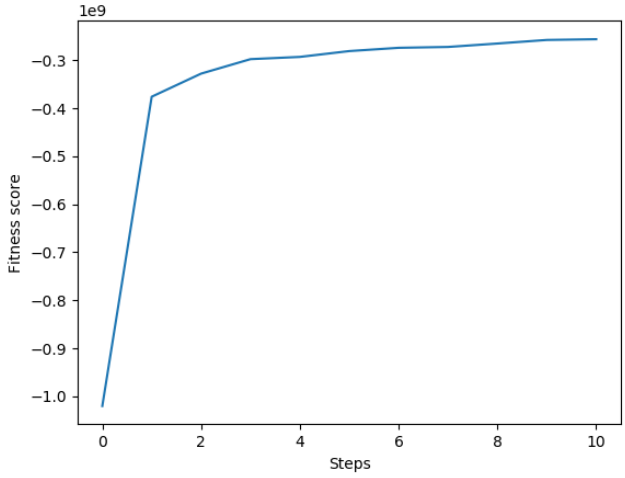


(a) num_genes=10 (b) num_genes=25 (c) num_genes=100 (d) num_genes=150

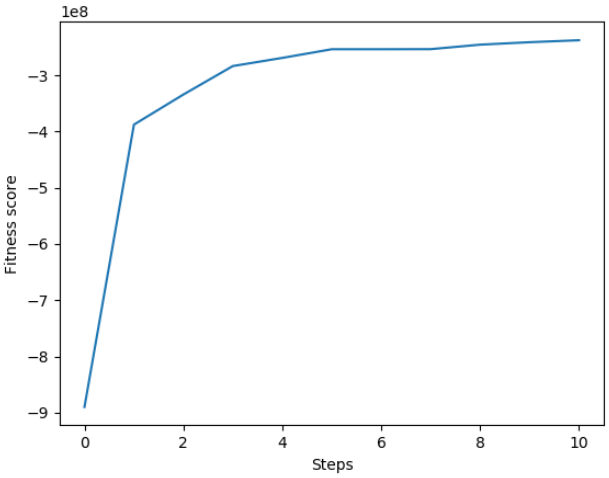
Figure 5: The image of the best individual in the population at every 1000th generation for (<num_genes>)=10,25,100,150



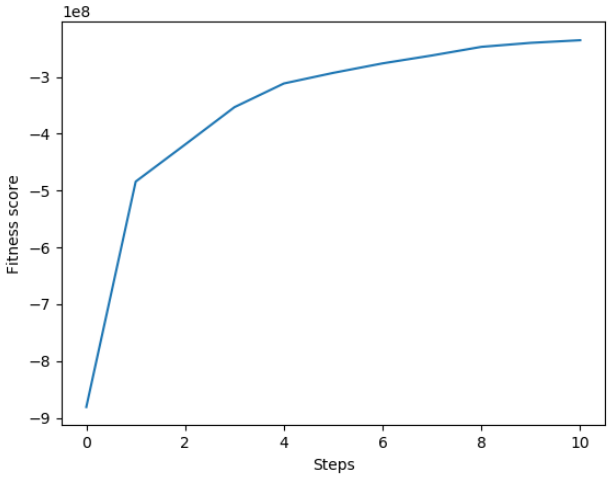
(a) num_genes=10



(b) num_genes=25

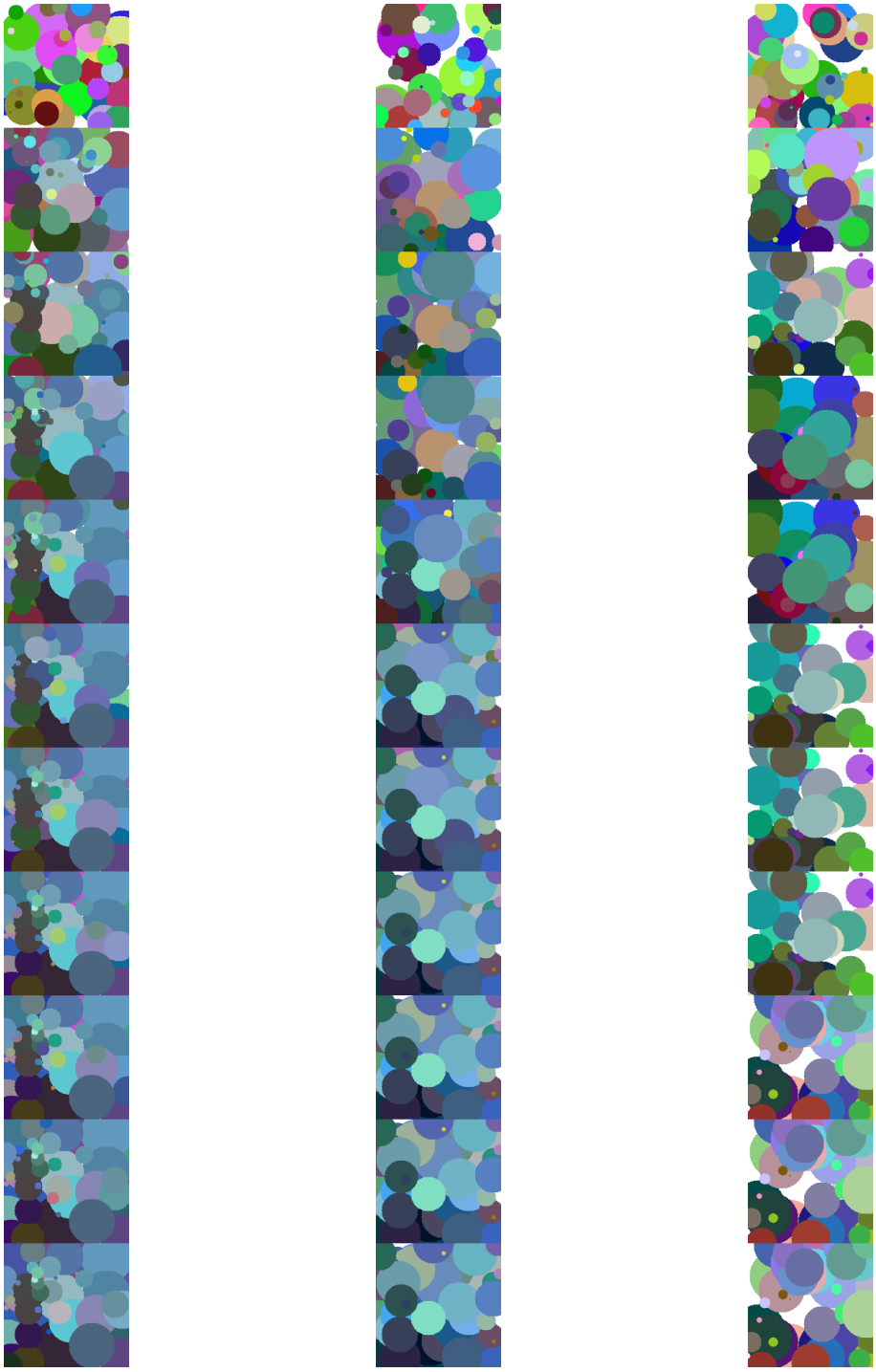


(c) num_genes=100



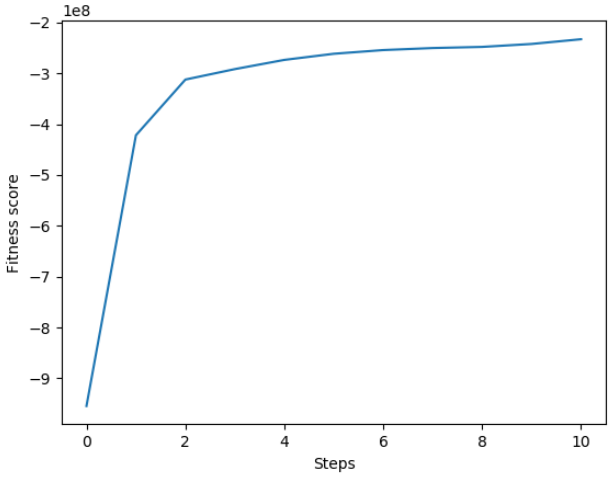
(d) num_genes=150

Figure 6: The fitness plot from generation 1 to generation 10000 for (<num_genes>)=10,25,100,150

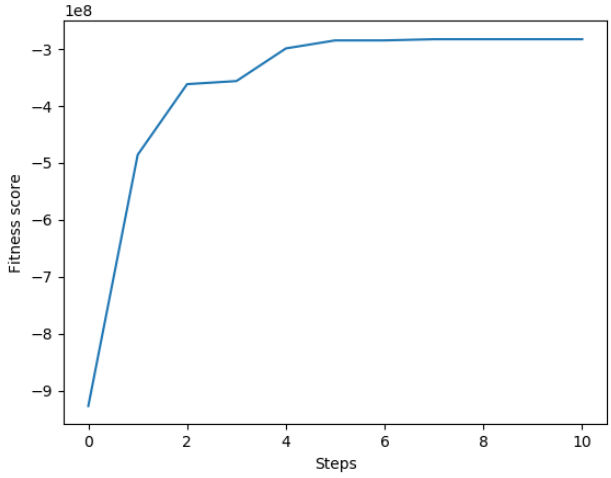


(a) tm_size=2 (b) tm_size=10 (c) tm_size=20

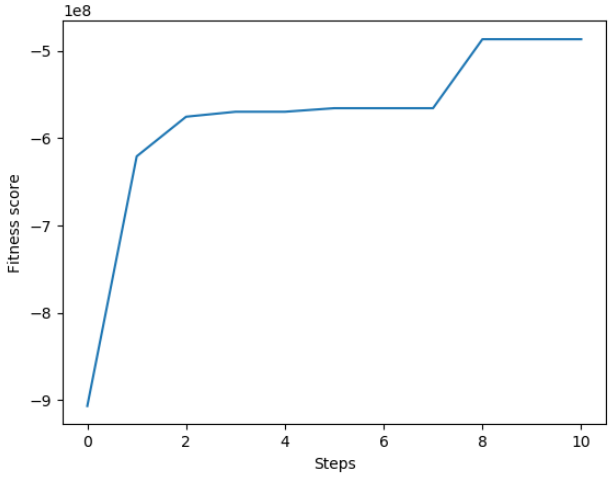
Figure 7: The image of the best individual in the population at every 1000th generation for (<tm_size>)=2, 10, 20



(a) tm_size=2



(b) tm_size=10



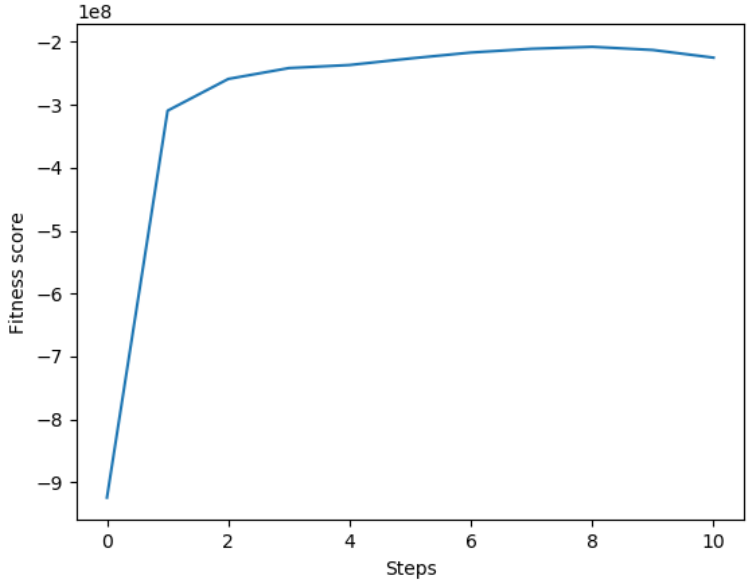
(c) tm_size=20

Figure 8: The fitness plot from generation 1 to generation 10000 for (<tm_size>)=2, 10, 20

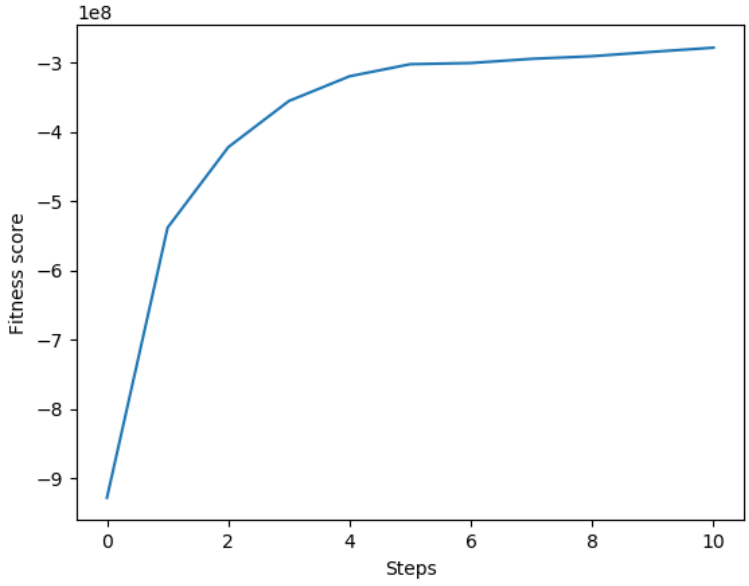
(a)
frac_elites=0.05

(b)
frac_elites=0.4

Figure 9: The image of the best individual in the population at every 1000th generation for (<frac_elites>)=0.05, 0.4

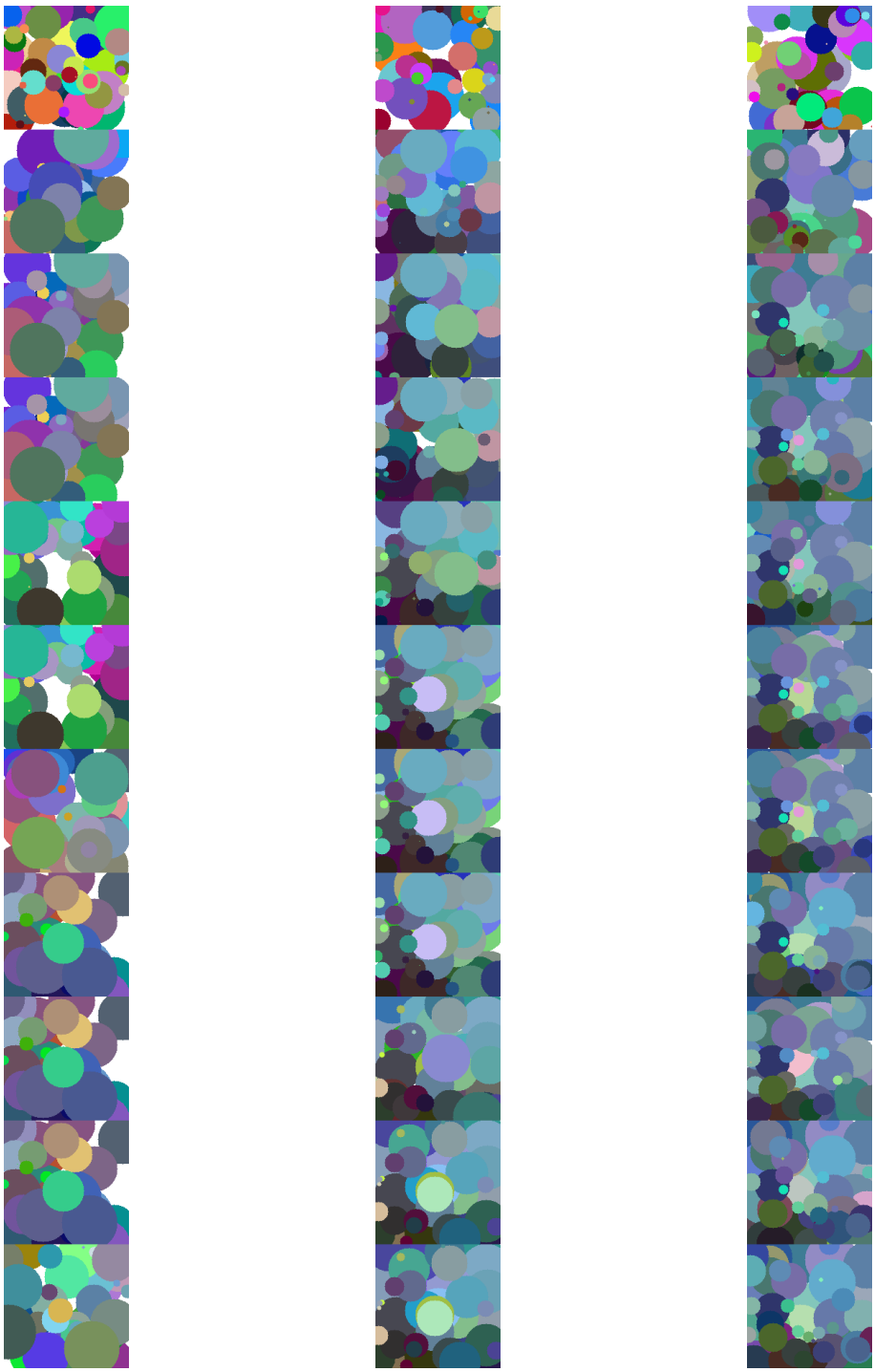


(a) frac_elites=0.05



(b) frac_elites=0.4

Figure 10: The image of the best individual in the population at every 1000th generation for (<frac_elites>)=0.05, 0.4

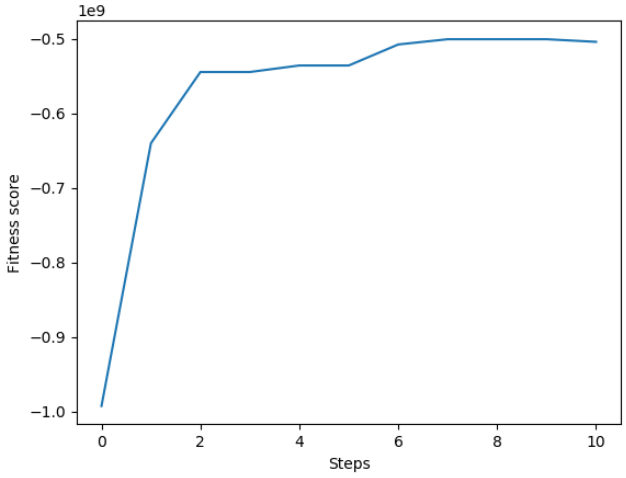


(a)
frac_parents=0.2

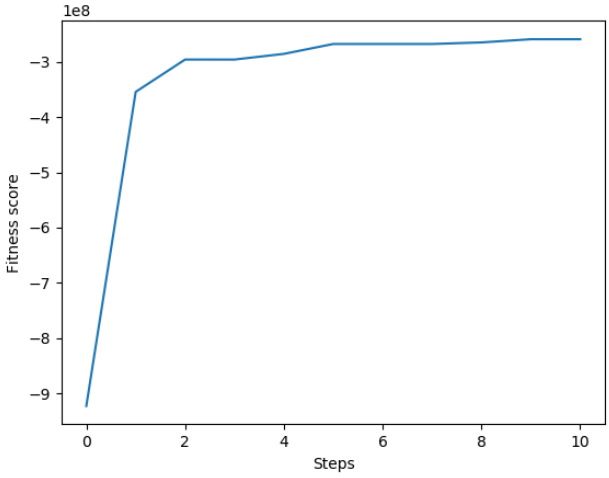
(b)
frac_parents=0.4

(c)
frac_parents=0.8

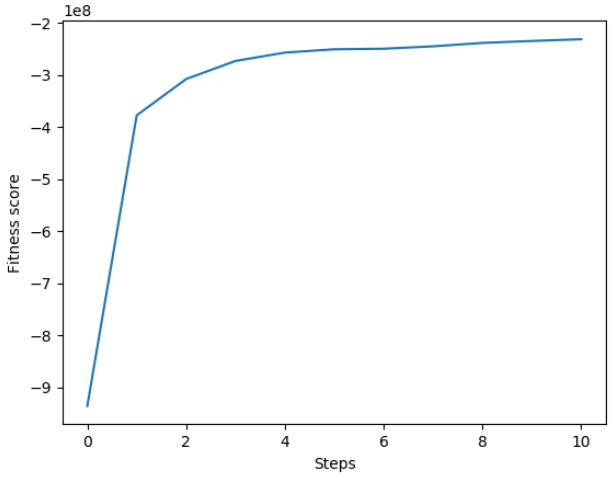
Figure 11: The image of the best individual in the population at every 1000th generation for (<frac_parents>)=0.2, 0.4, 0.8



(a) frac_parents=0.2

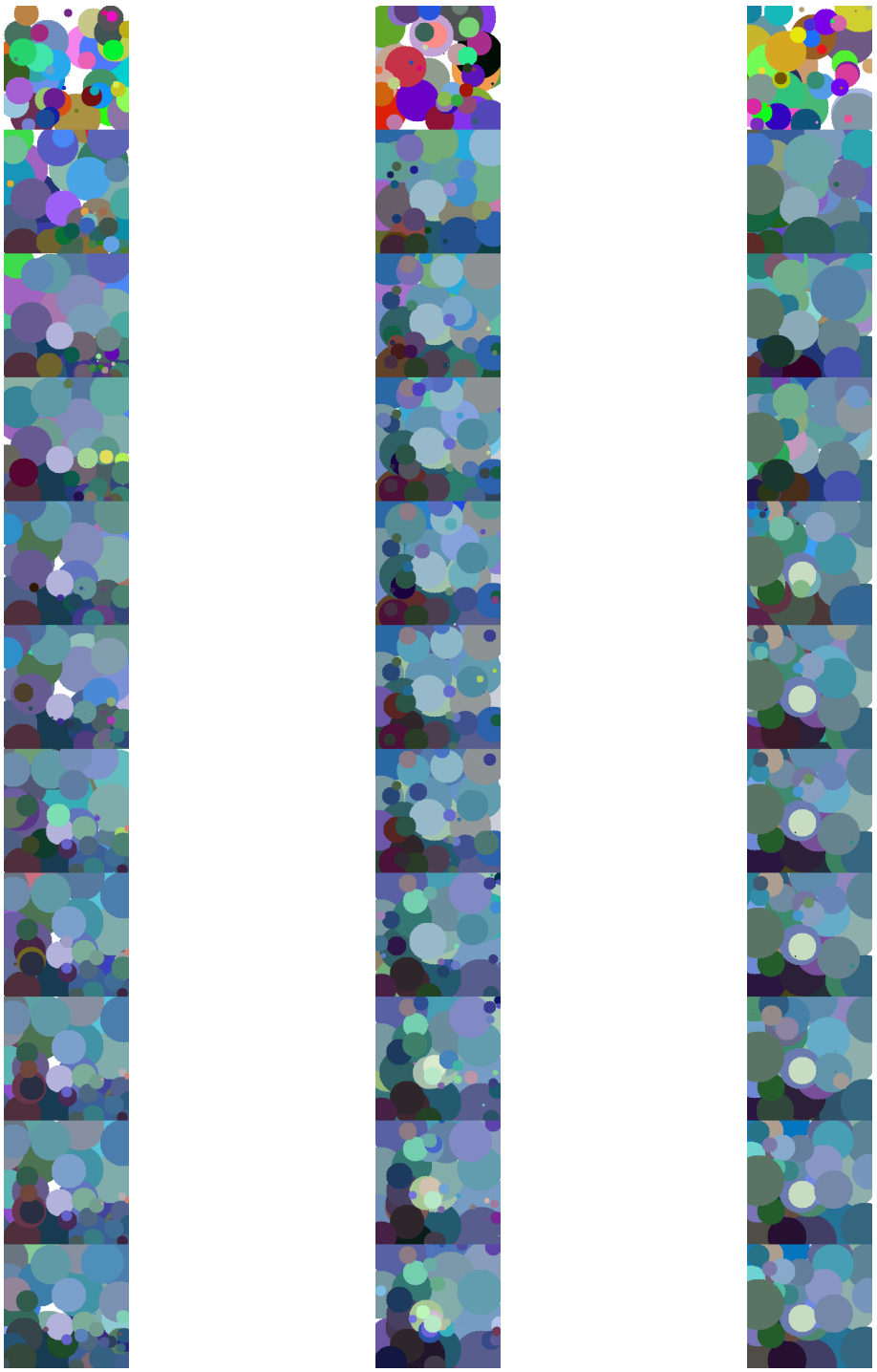


(b) frac_parents=0.4



(c) frac_parents=0.8

Figure 12: The fitness plot from generation 1 to generation 10000 for (<frac_parents>)=0.2, 0.4, 0.8

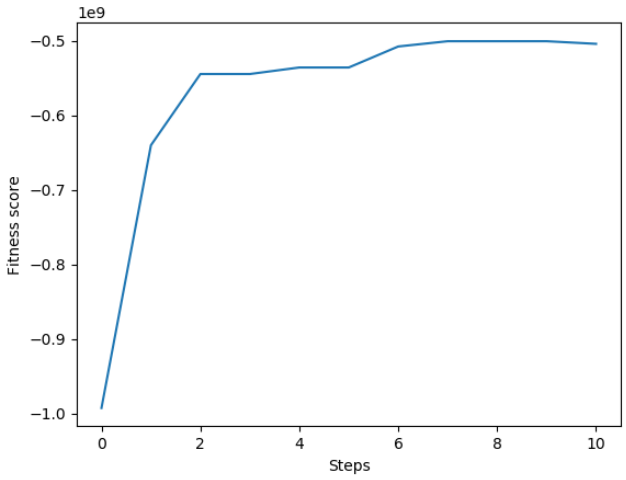


(a) mutation_prob=0.1

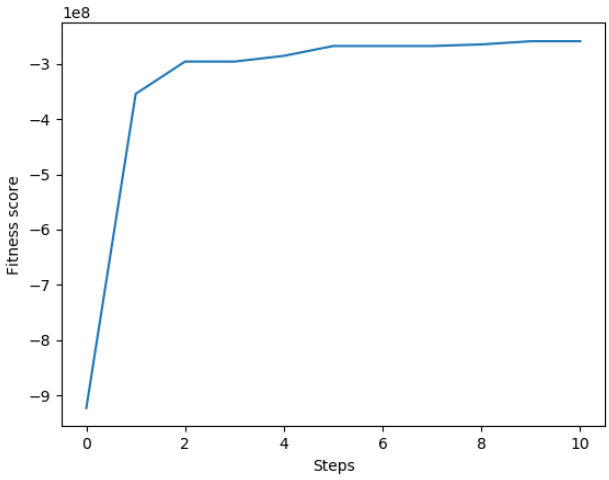
(b) mutation_prob=0.5

(c) mutation_prob=0.8

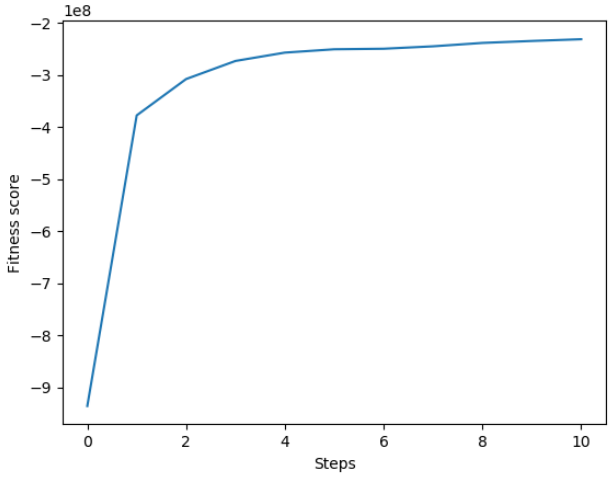
Figure 13: The image of the best individual in the population at every 1000th generation for (<mutation_prob>)=0.1, 0.5, 0.8



(a) mutation_prob=0.1



(b) mutation_prob=0.5



(c) mutation_prob=0.8

Figure 14: The fitness plot from generation 1 to generation 10000 for (<mutation_prob>)=0.1, 0.5, 0.8

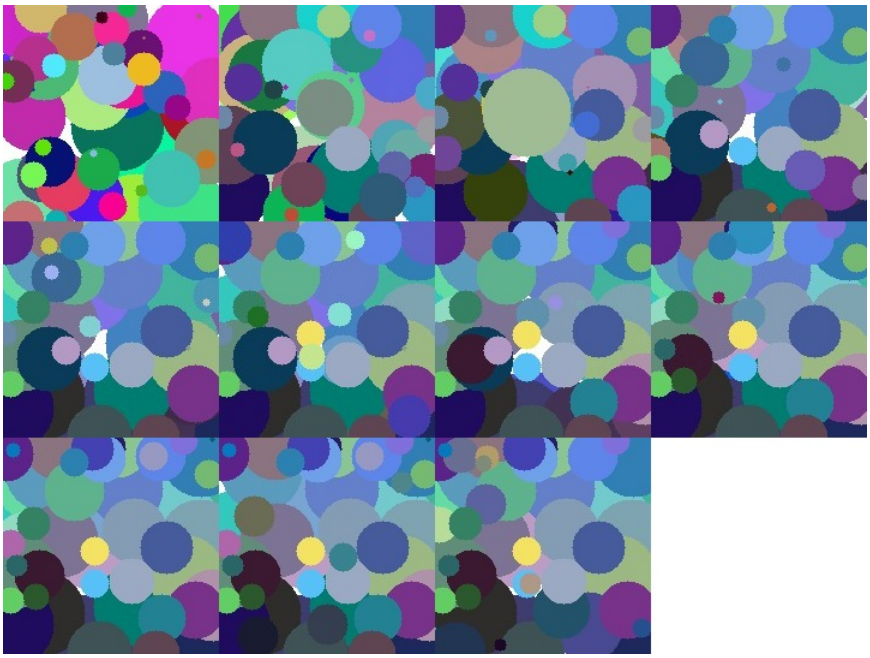


Figure 15: The best of individuals for the **unguided** mutation type

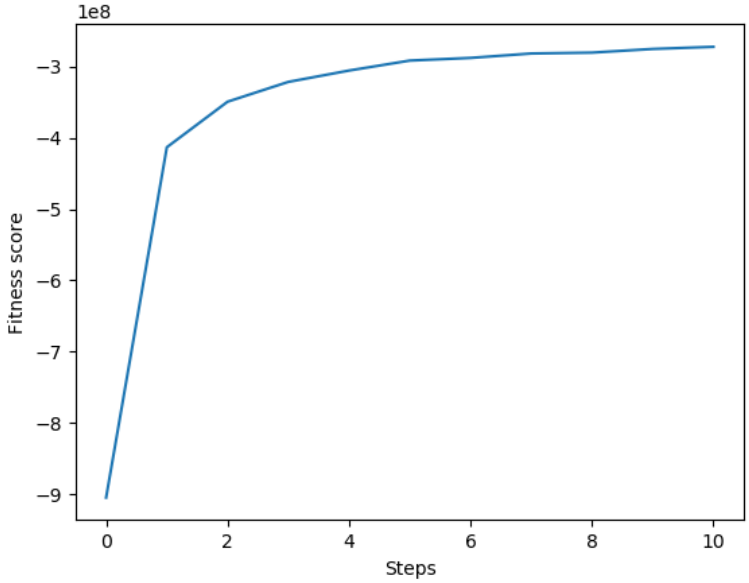


Figure 16: The fitness curve for the **unguided** mutation type

Discussion

1.

As a first change, we can use the **Roulette-wheel** selection method instead of using the tournament. When we look at the fitness score curves of the tm_size parameters, it is clear that its convergence speed is low in bigger tm_sizes. By using the Roulette-wheel selection, we also avoid this problem. As can be seen from the figure 18, the convergence speed of the Roulette-wheel is better than a tournament.

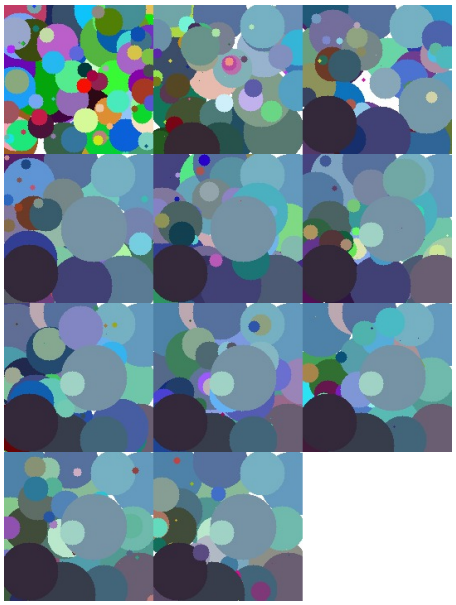


Figure 17: The best of individuals for the **Roulette-wheel** selection method

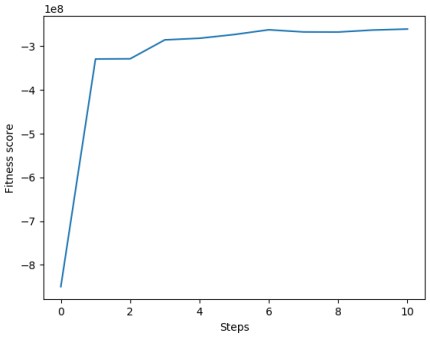


Figure 18: The fitness curve for the **Roulette-wheel** selection method

2.

When we observe the mutation probability parameter, we can see that when the mutation probability is higher, the convergence speed is higher, so as a second suggestion, the scheduled mutation probability can be used.
In this method, the mutation probability is 0.8 for the 2500 generations. Then it is set as 0.5 for the rest of the generations; the following two figures (19 and 20) show the result of the scheduled mutation probability.

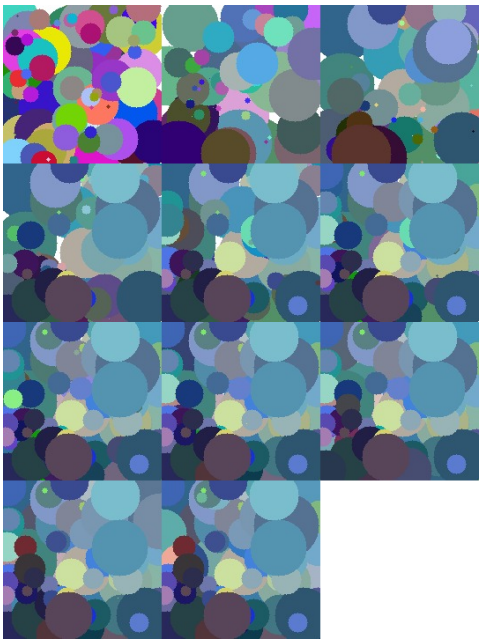


Figure 19: The best of individuals for the **scheduled** mutation prob method

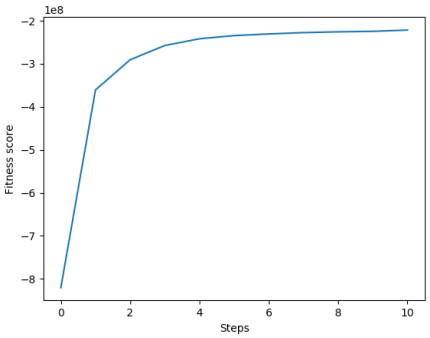


Figure 20: The fitness curve for the **scheduled** mutation prob method

3.

Lastly, in order to provide faster and better convergence, the hyper-parameters can be optimized. When we look at the seven different parameters separately, we can observe that each parameter affects the algorithm performance. If we choose the (<num_inds<),(<num_genes<), (<tm_size<), (<frac_elites<), (<frac_parents<), (<mutation_prob<) and (<mutation_type<) as **50, 100, 2, 0.05, 0.8, 0.8 and "guided"** respectively, we would end up with following result. As can be seen from these two figures (figure. 21 and 22), when the parameters are optimized, the results are much better than most of them.

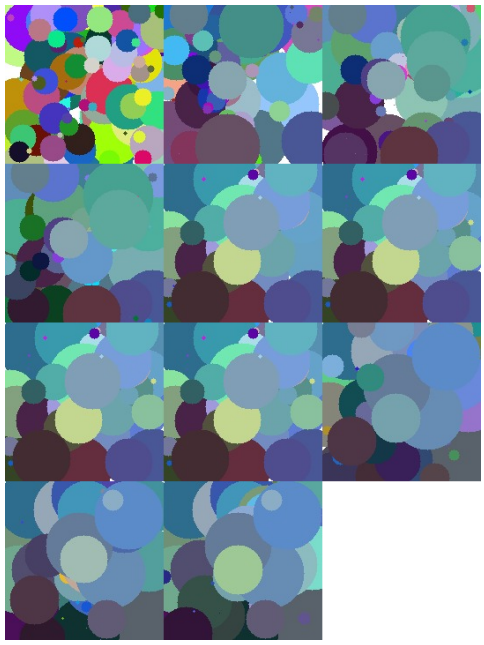


Figure 21: The best of individuals for the **optimized** hyper-parameter

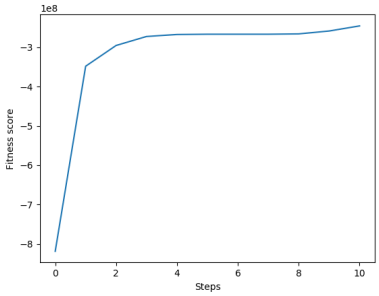


Figure 22: The fitness curve for the **optimized** hyper-parameter

Appendix

File - C:\Users\berkenutku\Desktop\496_hw2\hw2.py

```python
import cv2
import numpy as np
import random
from operator import itemgetter


# Class for each Individual
class Individual:

    def __init__(self, num_genes):
        self.Genes = self.create_genes(num_genes)
        self.Gene = self.Gene()

    # Each individual is composed of Gene which has 7 features
    class Gene:
        def __init__(self):
            random.seed(random.random())
            # Until the circle is within the image, change the x,y and radius values.
            while True:
                self.x = random.randint(1, 180)
                self.y = random.randint(1, 180)
                self.radius = random.randint(1, 40)
                if abs(self.x - self.radius) < 180 or abs(self.y - self.radius < 180):
                    break
            self.R = random.randint(0, 255)
            self.G = random.randint(0, 255)
            self.B = random.randint(0, 255)
            self.A = random.random()

    # Guided mutation, change the gene values according to its previous values
    # This code is quite long since we have to change the features if they are not valid.
    def guided_mutate(self, guided_gene):
        output = self.Gene
        if guided_gene.x + 45 > 180:
            output.x = random.randint(guided_gene.x - 45, 180)
        elif guided_gene.x - 45 < 0:
            output.x = random.randint(0, guided_gene.x + 45)
        else:
            output.x = random.randint(guided_gene.x - 45, guided_gene.x + 45)

        if guided_gene.y + 45 > 180:
            output.y = random.randint(guided_gene.y - 45, 180) if guided_gene.y - 45 < 180 else random
.randint(135, 180)
        elif guided_gene.y - 45 < 0:
            output.y = random.randint(0, guided_gene.y + 45)
        else:
            output.y = random.randint(guided_gene.y - 45, guided_gene.y + 45)

        if guided_gene.radius + 10 > 40:
            output.radius = random.randint(guided_gene.radius - 10, 40)
        elif guided_gene.radius - 10 < 0:
            output.radius = random.randint(0, guided_gene.radius + 10)
        else:
            output.radius = random.randint(guided_gene.radius - 10, guided_gene.radius + 10)

        if guided_gene.R + 64 > 255:
            output.R = random.randint(guided_gene.R - 64, 255)
        elif guided_gene.R - 64 < 0:
            output.R = random.randint(0, guided_gene.R + 64)
        else:
            output.R = random.randint(guided_gene.R - 64, guided_gene.R + 64)

        if guided_gene.G + 64 > 255:
            output.G = random.randint(guided_gene.G - 64, 255)
        elif guided_gene.G - 64 < 0:
            output.G = random.randint(0, guided_gene.G + 64)
        else:
            output.G = random.randint(guided_gene.G - 64, guided_gene.G + 64)

        if guided_gene.B + 64 > 255:
            output.B = random.randint(guided_gene.B - 64, 255)
        elif guided_gene.B - 64 < 0:
            output.B = random.randint(0, guided_gene.B + 64)
        else:
            output.B = random.randint(guided_gene.B - 64, guided_gene.B + 64)

        if guided_gene.A + 0.25 > 1:
```

File - C:\Users\berkenutku\Desktop\496_hw2\hw2.py

```
77              output.A = random.uniform(guided_gene.A - 0.25, 1)
78          elif guided_gene.A - 0.25 < 0:
79              output.A = random.uniform(0, guided_gene.A + 0.25)
80          else:
81              output.A = random.uniform(guided_gene.A - 0.25, guided_gene.A + 0.25)
82
83          return output
84
85      def create_genes(self, num_genes):
86          genes = {}
87          output = []
88          # Create Gene according to parameter num_genes
89          while num_genes != 0:
90              d = self.Gene()
91              # Create a dictionary, add the created gene to dict with its radius
92              inserted_element = {d: d.radius}
93              genes.update(inserted_element)
94              num_genes -= 1
95          # Sort the genes according to their radius. (This sorting is used for the first creation of
    an individual)
96          a = sorted(genes.items(), key=lambda x: x[1], reverse=True)
97          # Add the genes an individual in the sorted order
98          for a_tuple in a:
99              output.append(a_tuple[0])
100
101         return output
102
103     # This function is used for ordering genes after creating an individual i.e mutation,crossover
104     def order_my_genes(self):
105         genes = {}
106         counter = 0
107         while len(self.Genes) != counter:
108             d = self.Genes[counter]
109             inserted_element = {d: d.radius}
110             genes.update(inserted_element)
111             counter += 1
112         # Sort the genes according to their radius
113         a = sorted(genes.items(), key=lambda x: x[1], reverse=True)
114         counter = 0
115         for a_tuple in a:
116             self.Genes[counter] = (a_tuple[0])
117             counter += 1
118
119     def draw_image(self, source_image):
120         # Order the genes according to radius
121         self.order_my_genes()
122         # Initialize <image> completely white with the same shape as the <source_image>.
123         shape_of_image = source_image.shape[0]
124         image = np.zeros((shape_of_image, shape_of_image, 3), np.uint8)
125         image[:] = (255, 255, 255)
126         output = image.copy()
127         for k in self.Genes:
128             overlay = output
129             cv2.circle(overlay, (k.x, k.y), k.radius, (k.B, k.G, k.R), -1)
130             output = cv2.addWeighted(overlay, k.A, output, 1 - k.A, 0)
131
132         return output
133
134     def do_crossover(self, parent2):
135         num_gene = len(self.Genes)
136         # Create two child with the equal number of gene with parents
137         child1 = Individual(num_gene)
138         child2 = Individual(num_gene)
139         counter = 0
140         while counter != num_gene:
141             # Exchange of each gene is calculated individually with equal probability
142             prob = random.randint(0, 1)
143             if prob == 0:
144                 child1.Genes[counter] = self.Genes[counter]
145                 child2.Genes[counter] = parent2.Genes[counter]
146             else:
147                 child1.Genes[counter] = parent2.Genes[counter]
148                 child2.Genes[counter] = self.Genes[counter]
149             counter += 1
150         return child1, child2
151
152     def do_mutation(self, mutation_type):
```

File - C:\Users\berkenutku\Desktop\496_hw2\hw2.py

```
153         selected_gene = random.randint(0, len(self.Genes) - 1)
154         if mutation_type == 0:
155             new_gene = self.Gene
156             self.Genes[selected_gene] = new_gene
157         else:
158             new_gene = self.guided_mutate(self.Genes[selected_gene])
159             self.Genes[selected_gene] = new_gene
160 
161 
162 def initialize_population(num_inds=20, num_genes=50):
163     individuals = []
164     while num_inds != 0:
165         individuals.append(Individual(num_genes))
166         num_inds -= 1
167 
168     return individuals
169 
170 
171 def calculate_fitness(individual, source_image):
172     # Initialize <image> completely white with the same shape as the <source_image>
173     # In order to avoid the overflows the calculation is done with np.int64
174     image_of_individual = np.int64(individual.draw_image(source_image))
175     source_image = np.int64(source_image)
176     f = 0
177     # For each RGB channel calculate the pixel difference between two images
178     for i in range(0, 3):
179         individual_single_channel = image_of_individual[:, :, i].flatten()
180         source_single_channel = source_image[:, :, i].flatten()
181         f += -1 * sum((source_single_channel - individual_single_channel) ** 2)
182         i += 1
183     return f
184 
185 
186 def selection(Fitness_function, individuals, frac_elites=0.2):
187     # Calculate the number of individuals which go to next generation directly
188     number_of_next_generation = np.ceil(len(individuals) * frac_elites)
189     output_individuals = []
190     output_fitness = []
191     while number_of_next_generation != 0:
192         # Find the best in the population
193         index = [i for i, x in enumerate(Fitness_function) if x == max(Fitness_function)]
194         index = int(index[0])
195         # Add the best individual for the next generation with their fitness values
196         output_individuals.append(individuals[index])
197         output_fitness.append(Fitness_function[index])
198         # Delete the added individuals from the population with their fitness values
199         del individuals[index]
200         Fitness_function = np.delete(Fitness_function, [index, index])
201         number_of_next_generation -= 1
202 
203     return output_fitness, output_individuals, Fitness_function, individuals
204 
205 
206 # Comment for the Roulette-wheel selection method for the discussion
207 
208 """
209     number_of_next_generation = np.ceil(len(individuals) * frac_elites)
210     output_individuals = []
211     output_fitness = []
212 
213     # Calculate the normalized probabilities for selection
214 
215     prob = Fitness_function / sum(Fitness_function)
216     while number_of_next_generation != 0:
217         draw = choice(individuals, 1, p=prob)
218         index = individuals.index(draw)
219         output_individuals.append(individuals[index])
220         output_fitness.append(Fitness_function[index])
221         number_of_next_generation -= 1
222 """
223 
224 
225 def tournament(individuals, Fitness_function, tm_size=5):
226     output_individuals = []
227     output_fitness = []
228     counter = 0
229     # Until the number of output is equal to the number of individuals, continue to tournament
```

File - C:\Users\berkenutku\Desktop\496_hw2\hw2.py

```
230     while counter != len(individuals):
231         # Choose the individuals according the tm_size.
232         # If the number of individuals are less than tm_size, add all individuals to tournament.
233         if len(individuals) <= 5:
234             participants = [0, 1, 2, 3]
235         elif len(individuals) <= tm_size:
236             participants = np.arange(0, len(individuals), 1)
237         # Otherwise take the random samples from individuals according to tm_size
238         else:
239             participants = random.sample(range(0, len(individuals)), tm_size)
240 
241         tournament_fitness = list(Fitness_function[participants])
242         tournament_individuals = itemgetter(*participants)(individuals)
243         # Choose the best of the tournament
244         best_of_tournament = np.argmax(tournament_fitness)
245         # Add the champion to the output of the tournament
246         output_individuals.append(tournament_individuals[best_of_tournament])
247         output_fitness.append(tournament_fitness[best_of_tournament])
248         counter += 1
249 
250     return output_fitness, output_individuals
251 
252 
253 def crossover(fitness, population, frac_parents=0.6):
254     # Decide the number of parents which involve the crossover
255     number_of_parents = np.floor(len(population) * frac_parents)
256     number_of_parents = number_of_parents if number_of_parents % 2 == 0 else number_of_parents - 1
257     childs = []
258     while number_of_parents != 0:
259         # Amongst the parents choose the best two of them for crossover
260         best_of_parent_index = np.argmax(fitness)
261         parent1 = population[best_of_parent_index]
262         # Since the chosen parents go to crossover, delete them from the population with their
    fitness values
263         del population[best_of_parent_index]
264         fitness = np.delete(fitness, [best_of_parent_index, best_of_parent_index])
265 
266         best_of_parent_index = np.argmax(fitness)
267         parent2 = population[best_of_parent_index]
268         del population[best_of_parent_index]
269         fitness = np.delete(fitness, [best_of_parent_index, best_of_parent_index])
270 
271         # Create two children from two parents and add them to childs
272         child1, child2 = parent1.do_crossover(parent2)
273         childs.append(child1)
274         childs.append(child2)
275         number_of_parents -= 2
276 
277     return childs, population
278 
279 
280 def mutation(individuals, mutation_prob, mutation_type):
281     # Mutate the all individuals except the elites
282     number_individuals = len(individuals)
283     counter = 0
284     output = []
285     while counter != number_individuals:
286         processed_individual = individuals[counter]
287         #  While the generated random number is smaller than
288         # <mutation prob> a random gene is selected to be mutated
289         if random.random() < mutation_prob:
290             processed_individual.do_mutation(mutation_type)
291 
292         output.append(processed_individual)
293         counter += 1
294     return output
295 
296 
297 # Load image
298 source_image = cv2.imread("painting.png")
299 
300 # Initialize population with <num_inds> individuals each having <num_genes> genes
301 population = initialize_population(num_inds=20, num_genes=50)
302 
303 num_generations = 10001
304 i = 0
305 fitness_plot = np.zeros(11)
```

File - C:\Users\berkenutku\Desktop\496_hw2\hw2.py

```
306 while i != num_generations:
307     Fitness_of_individuals = np.zeros(len(population))
308     counter = 0
309     # Evaluate all the individuals
310     for x in population:
311         Fitness_of_individuals[counter] = calculate_fitness(x, source_image)
312         counter += 1
313 
314     # Record the best individual and the fitness for every 1000 epoch
315     if i % 1000 == 0:
316         best_individual_index = np.argmax(Fitness_of_individuals)
317         best_individual = population[best_individual_index]
318         fitness_plot[int(i / 1000)] = Fitness_of_individuals[best_individual_index]
319         best_individual_image = best_individual.draw_image(source_image)
320         cv2.imwrite('img' + str(i) + '.png', best_individual_image)
321 
322     # Select individuals
323     best_fitness, best_population, to_tournament_fitness, to_tournament_individuals = selection(
    Fitness_of_individuals,
324 
    population,
325 
    frac_elites=0.2)
326 
327     # Go to Tournament
328     other_fitness, other_population = tournament(to_tournament_individuals, to_tournament_fitness,
    tm_size=5)
329     # Do Crossovers
330     childs, old_population = crossover(other_fitness, other_population, frac_parents=0.8)
331     # Mutation ( For the sake of easiness, I used "1" for guided mutation and "0" for the unguided
    mutation)
332     mutation_population = childs + old_population
333     mutated_population = mutation(mutation_population, mutation_prob=0.2, mutation_type=1)
334     # New Population
335     population = best_population + mutated_population
336     i += 1
337 
338 # Plot the fitness
339 import matplotlib.pyplot as plt
340 
341 plt.ylabel("Fitness score")
342 plt.xlabel("Steps")
343 plt.plot(fitness_plot)
344 
```