Gülce Önder
2305159

METU EE 449 Computational Intelligence

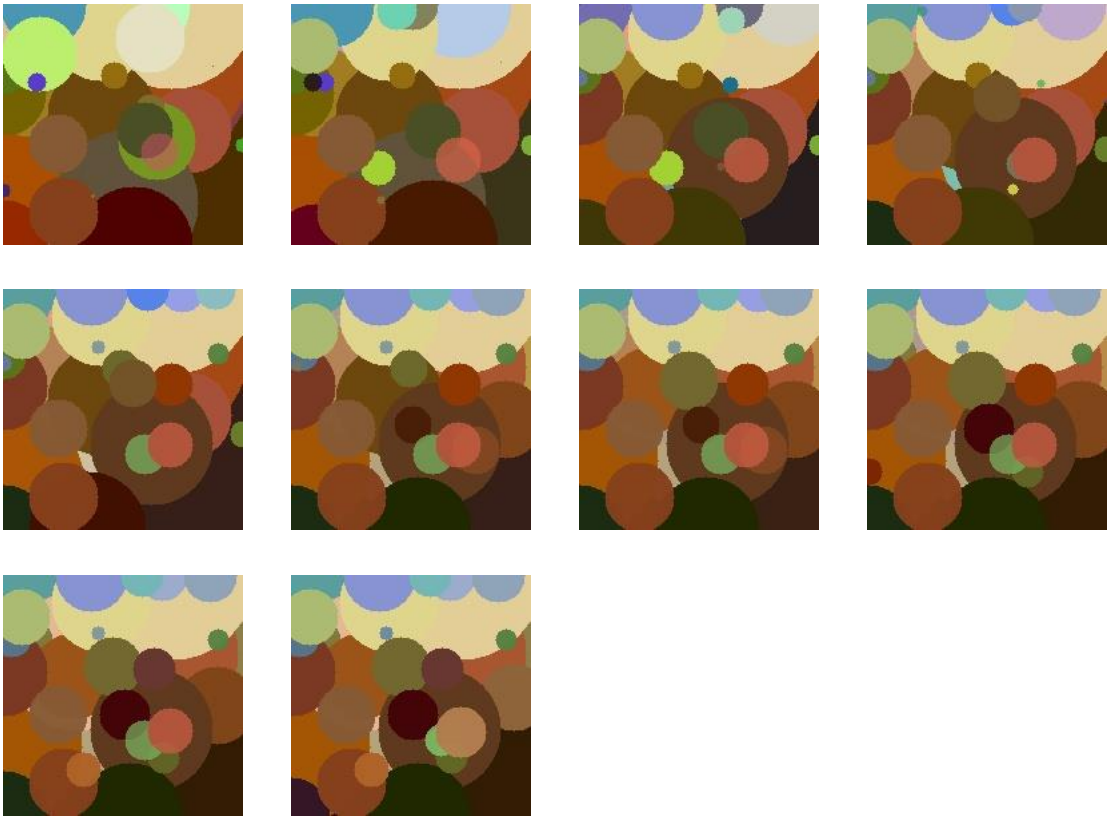
Homework 2 - Evolutionary Algorithms

Parameter: <num_inds> :

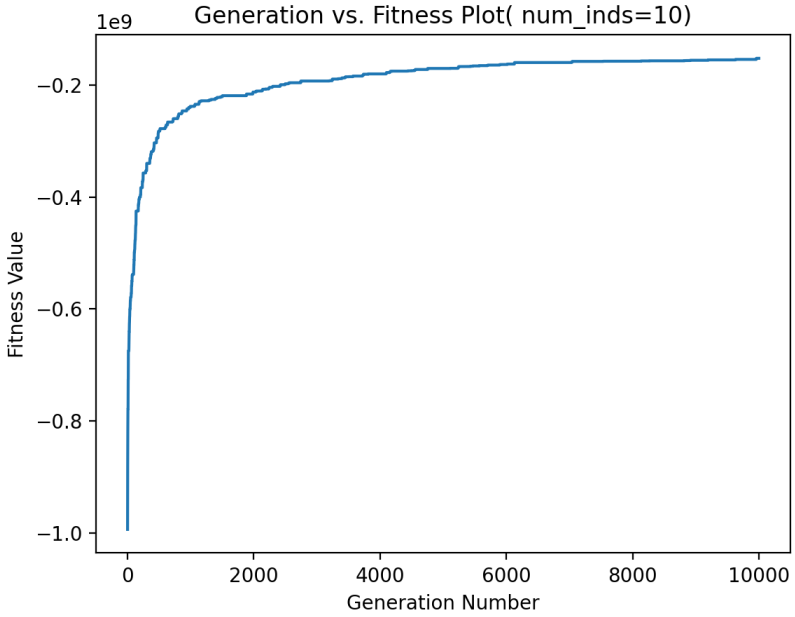
<num_inds> parameter represents the number of individuals (images containing circles in our case) each population contains. Having a greater number of individuals in a certain population increases the amount of best fitness candidates hence a high value given to this parameter means better chances of producing the required output. Below figures also support this claim and the best result is achieved when the parameter is equal to **75**.

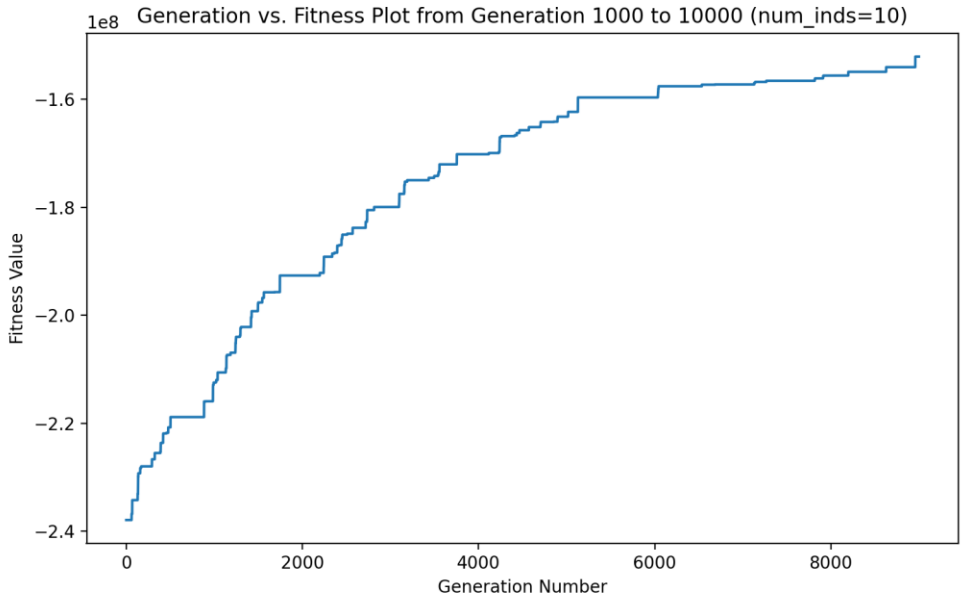
a. 5

b. 10



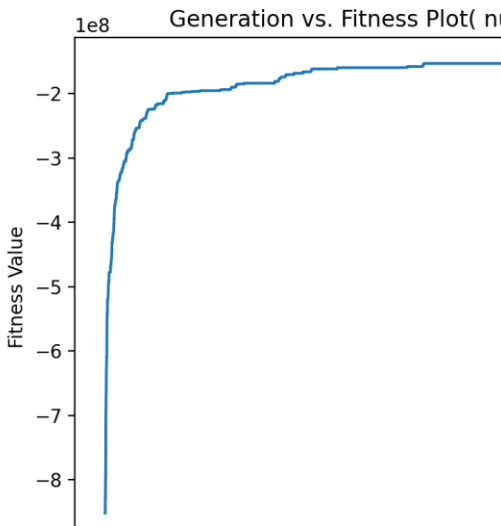
Gülce Önder
2305159





Gülce Önder
2305159

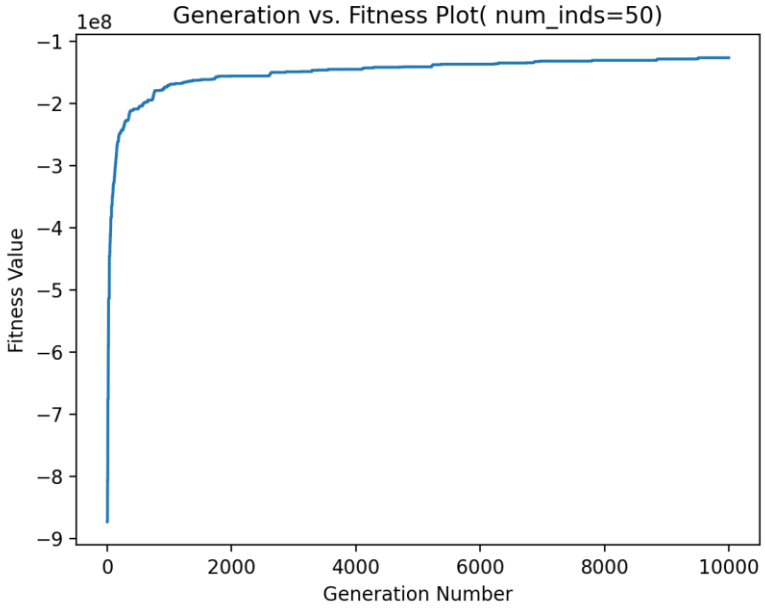
c. 20 (DEFAULT)

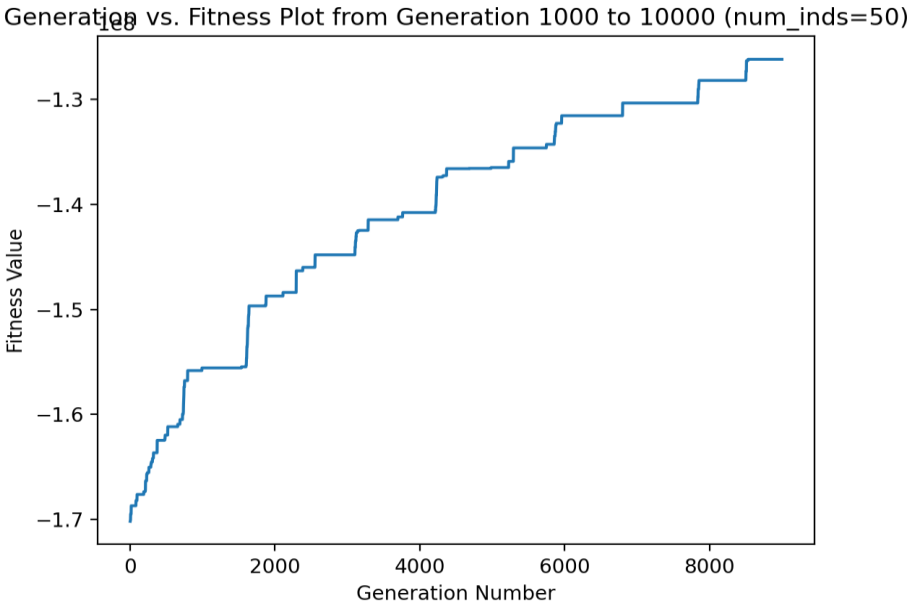


Gülce Önder
2305159

d. 50

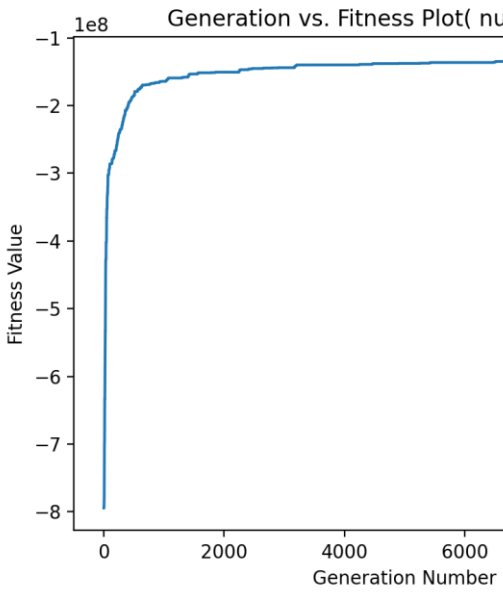
Gülce Önder
2305159



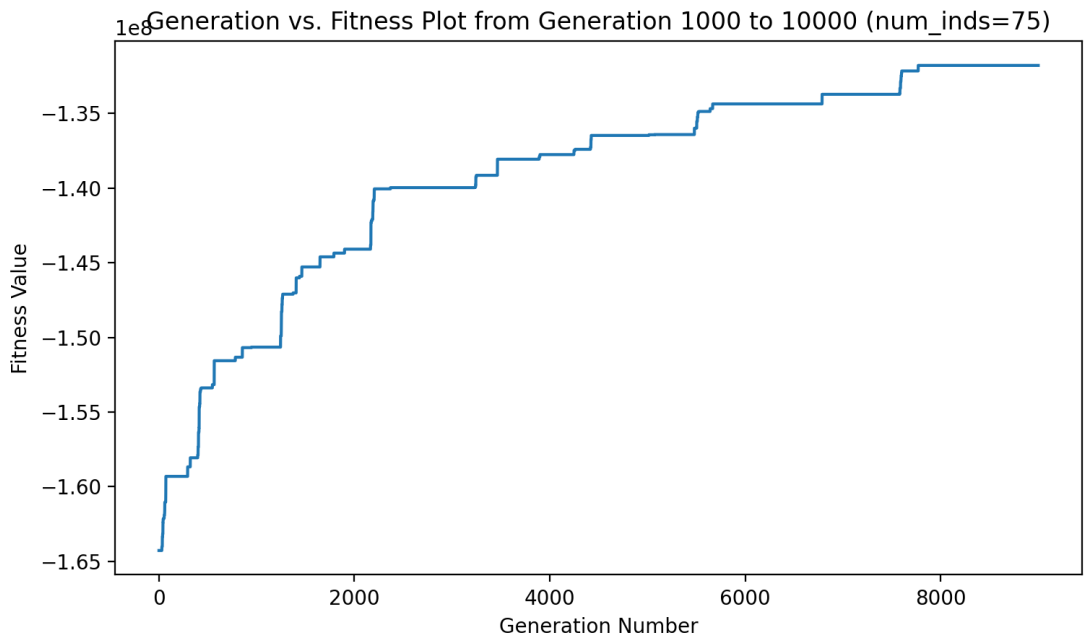


Gülce Önder
2305159

e. 75



Gülce Önder
2305159

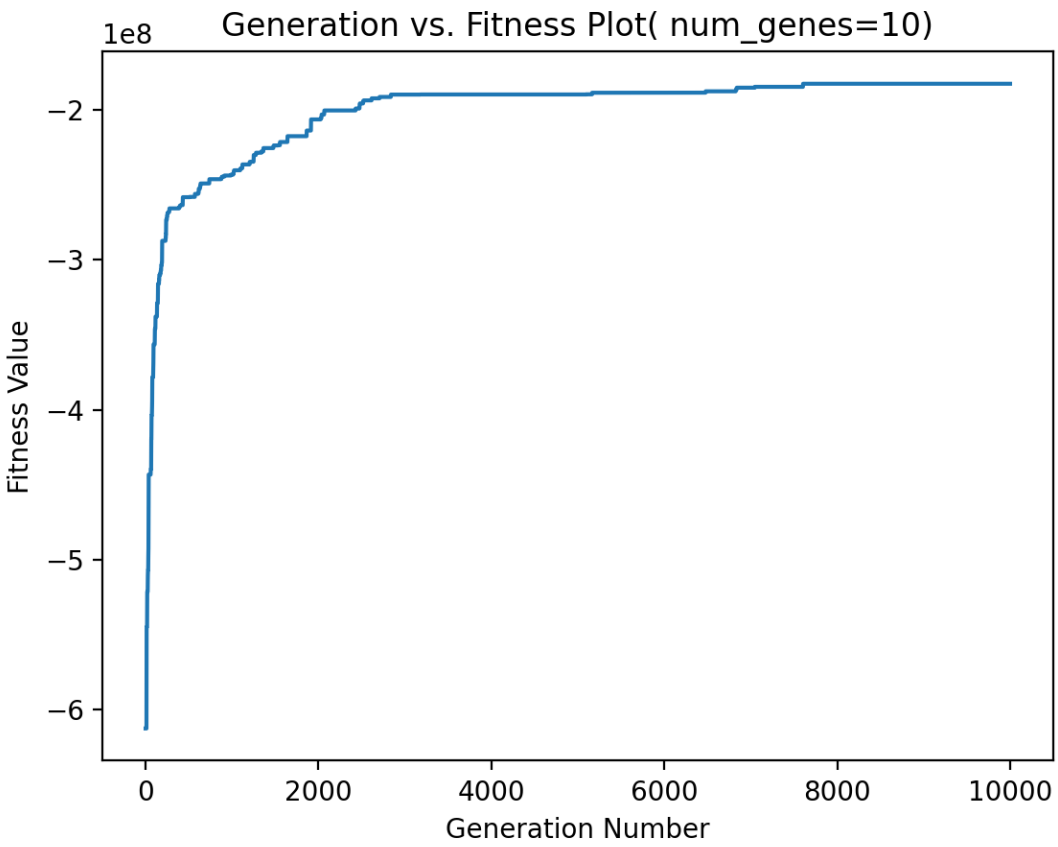


Parameter: <num_genes>:

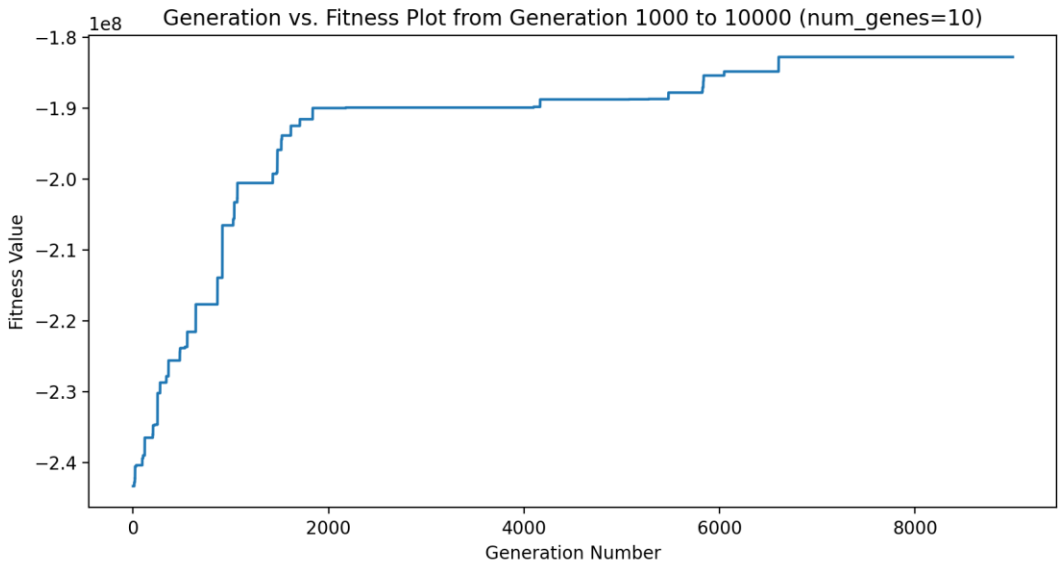
<num_genes> parameter represents the number of genes each individual contains in a given population. In our case, the number of circles to be drawn on each image. Higher number of genes means more circles that might have different colors drawn and it increases the possibility for our output image to resemble the given painting. Hence, higher numbers mean better outputs and the best result is achieved when this parameter equals **150**. (Although difference between 100 and 150 is not dramatically big.)

a.10

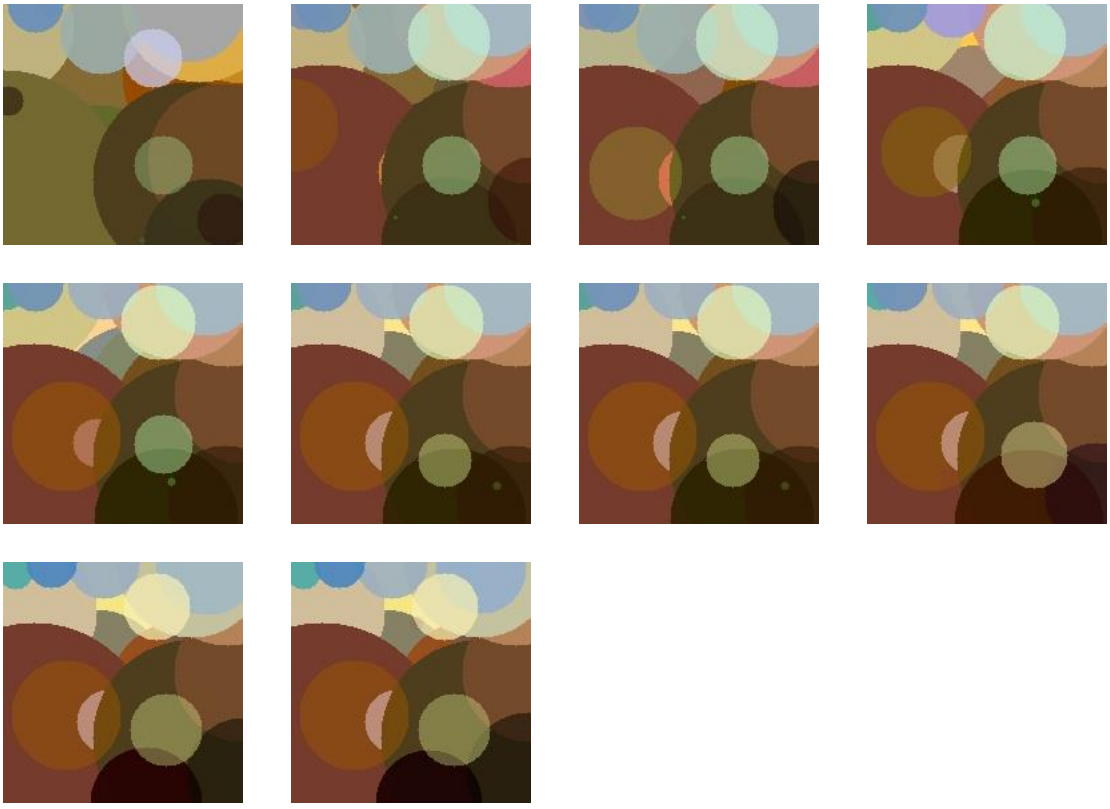
Gülce Önder
2305159



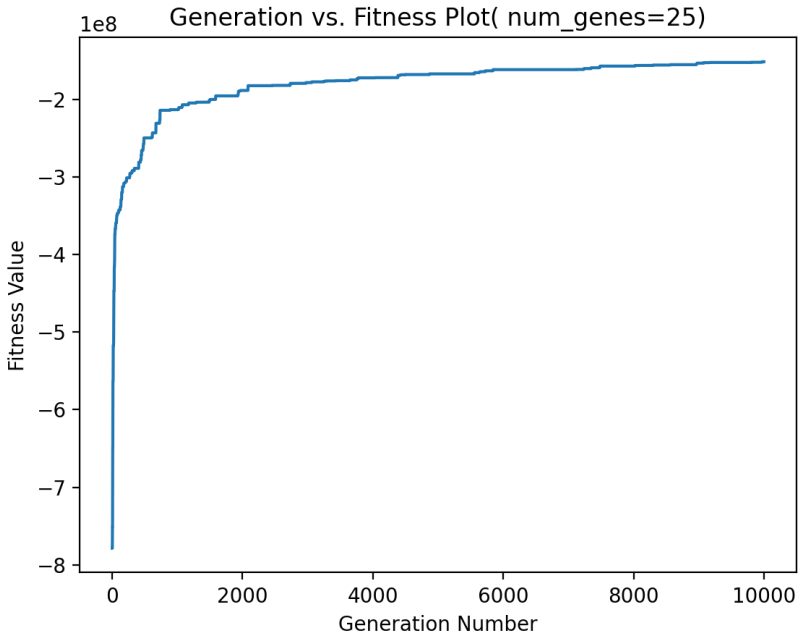
Gülce Önder
2305159

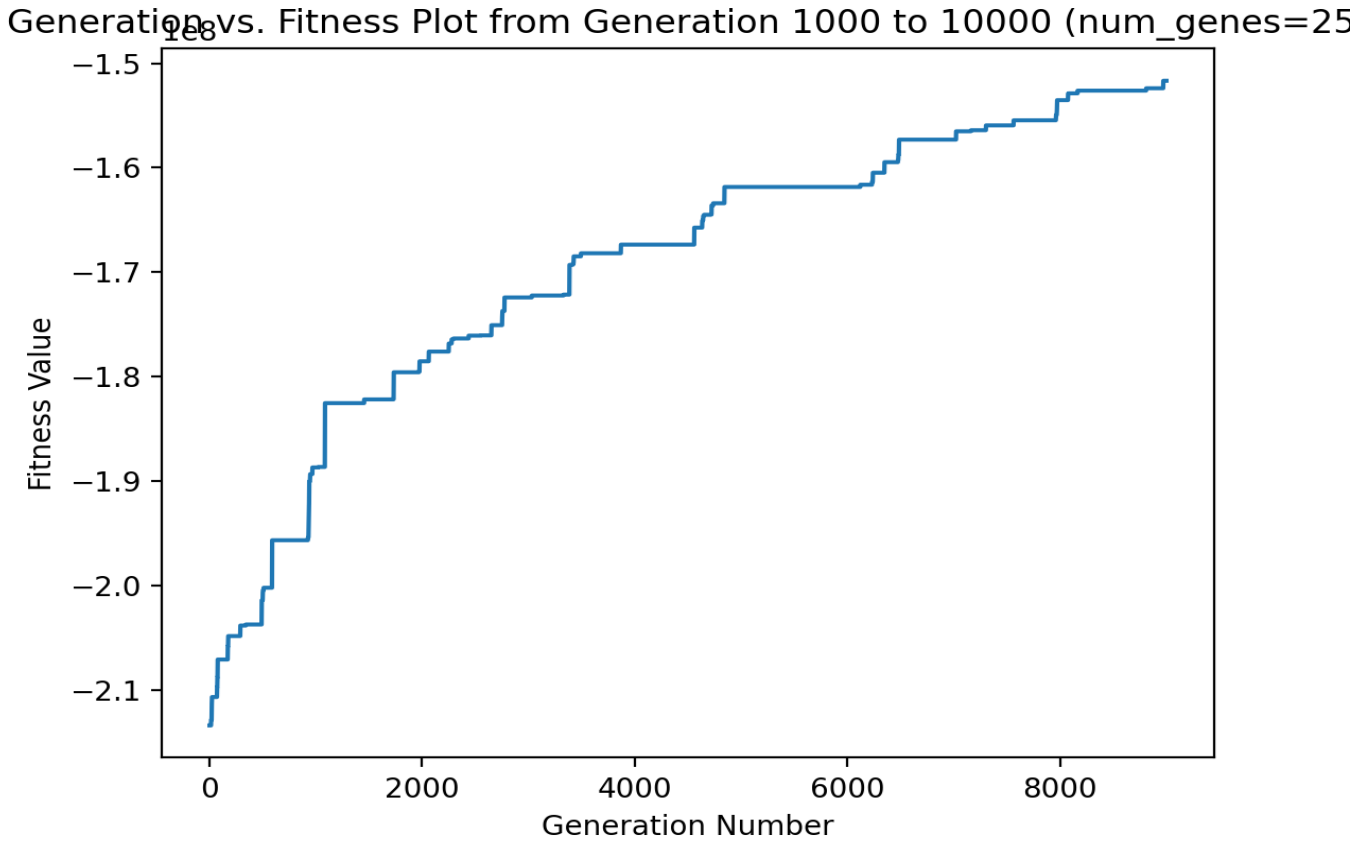


b.25



Gülce Önder
2305159





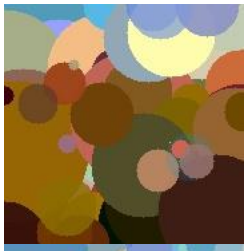
Gülce Önder
2305159

c.50(default)

d.100

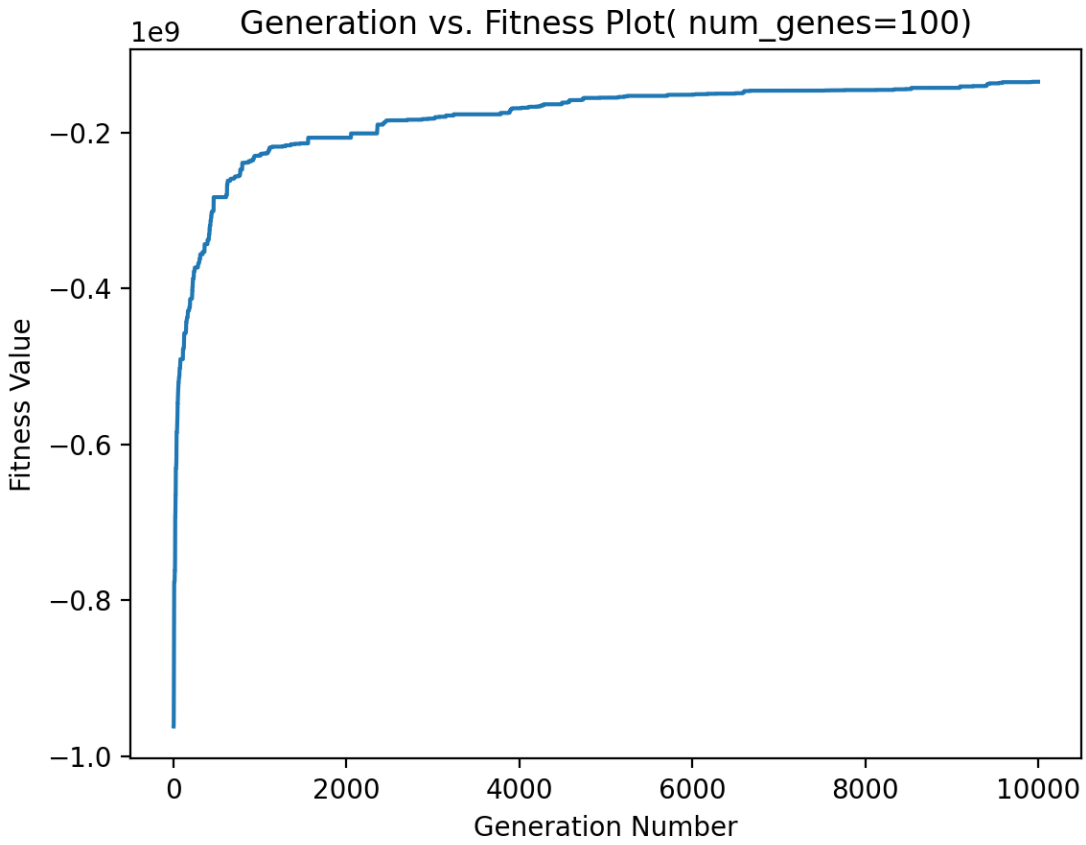




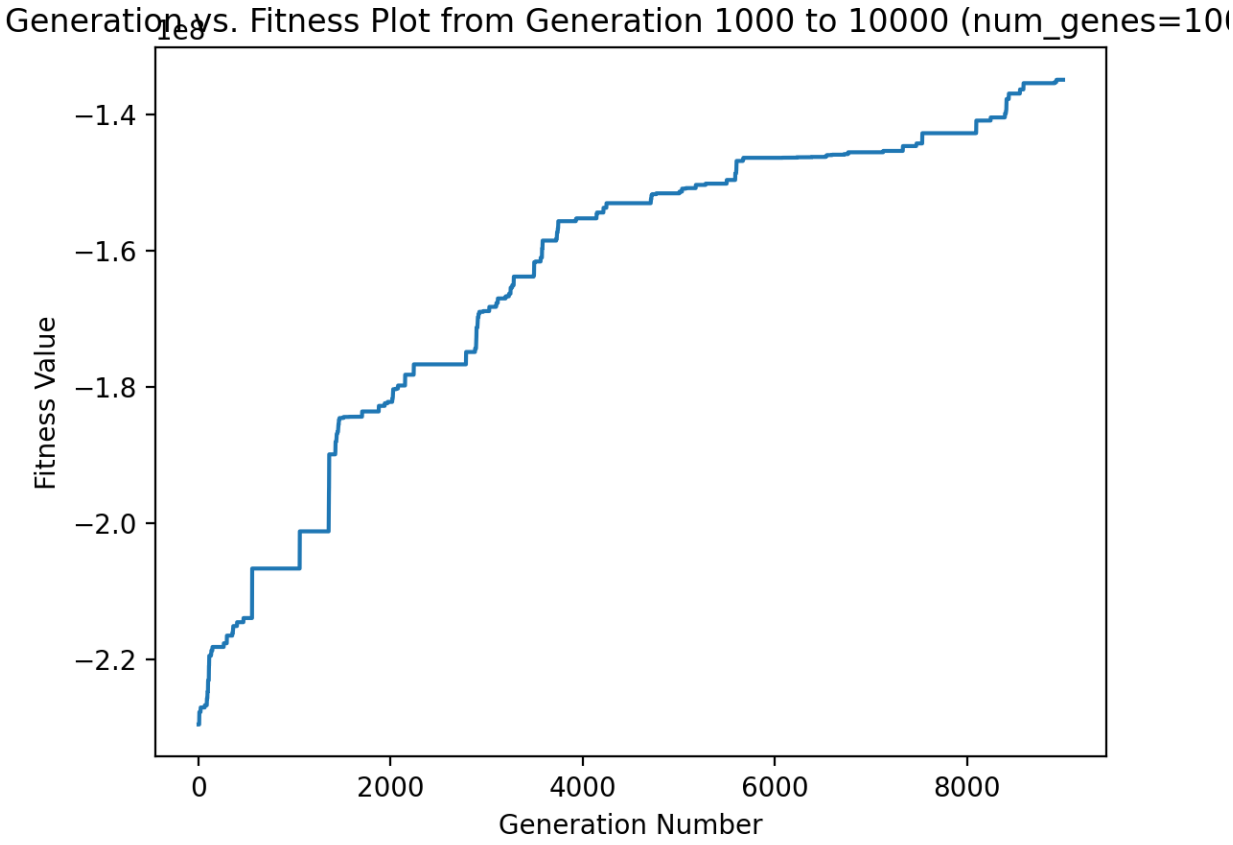





Gülce Önder
2305159

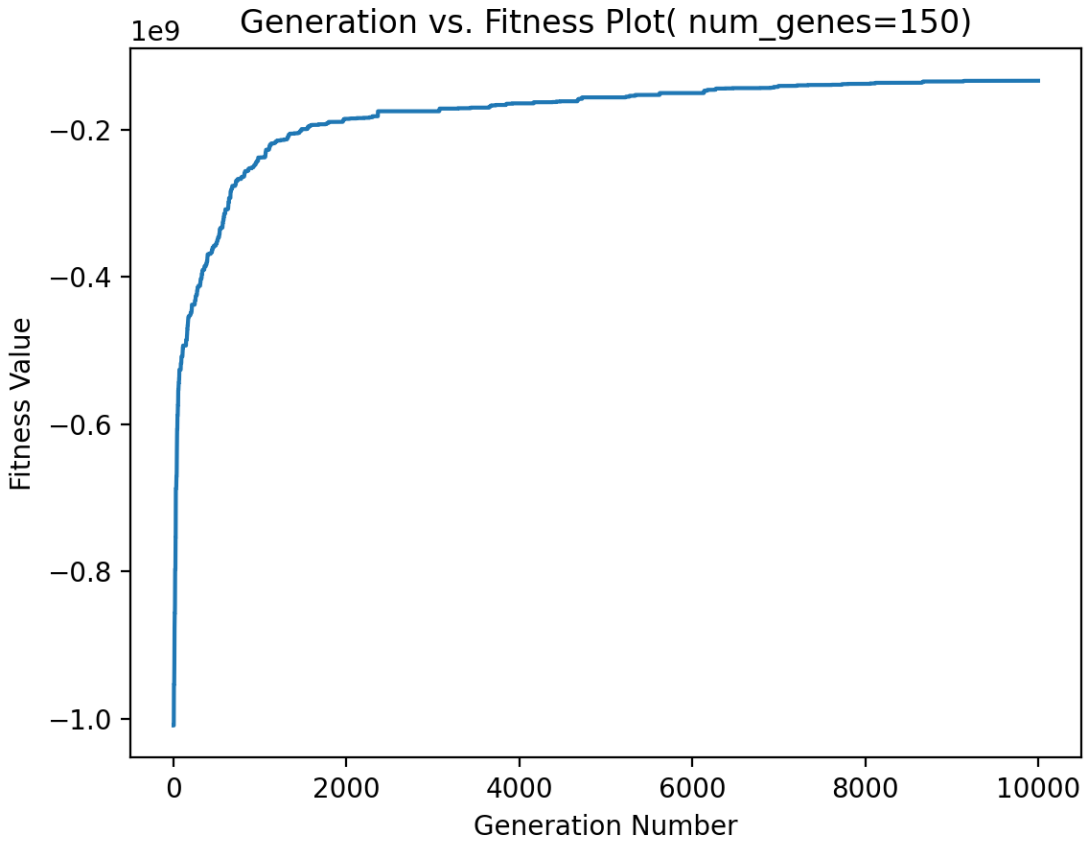


Gülce Önder
2305159

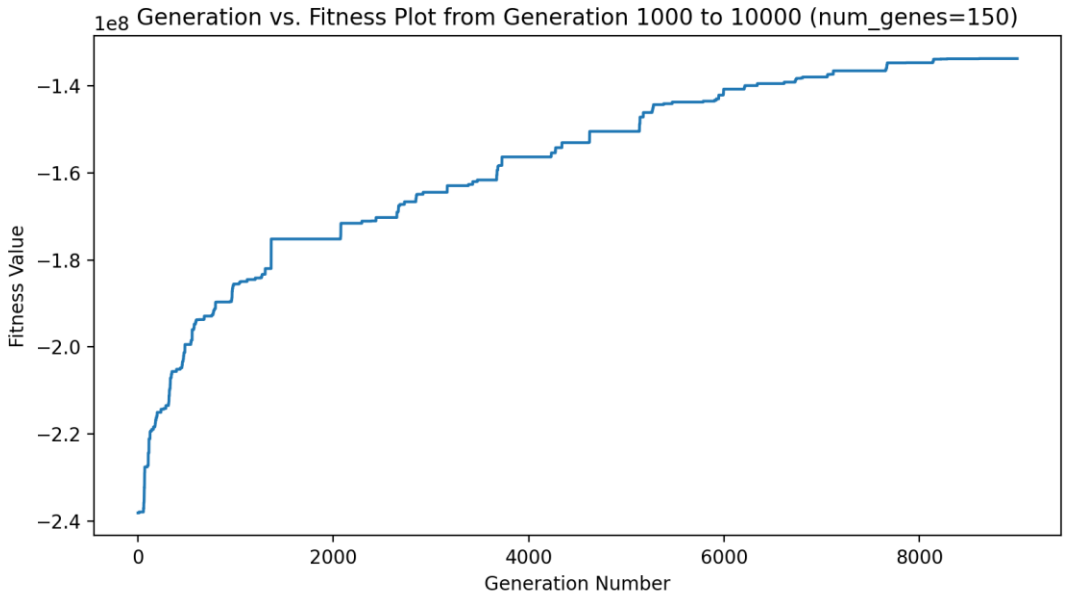


e.150

Gülce Önder
2305159



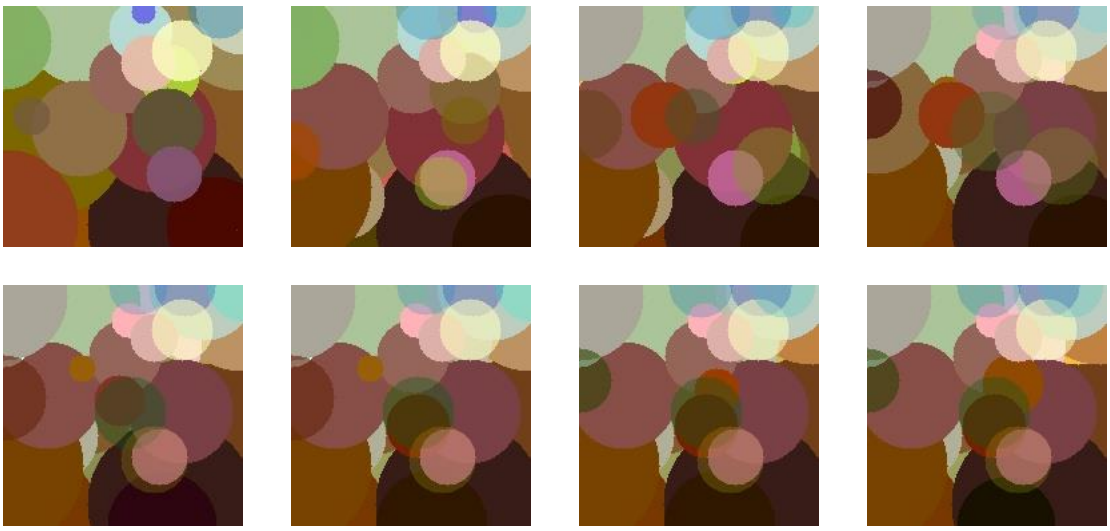
Gülce Önder
2305159



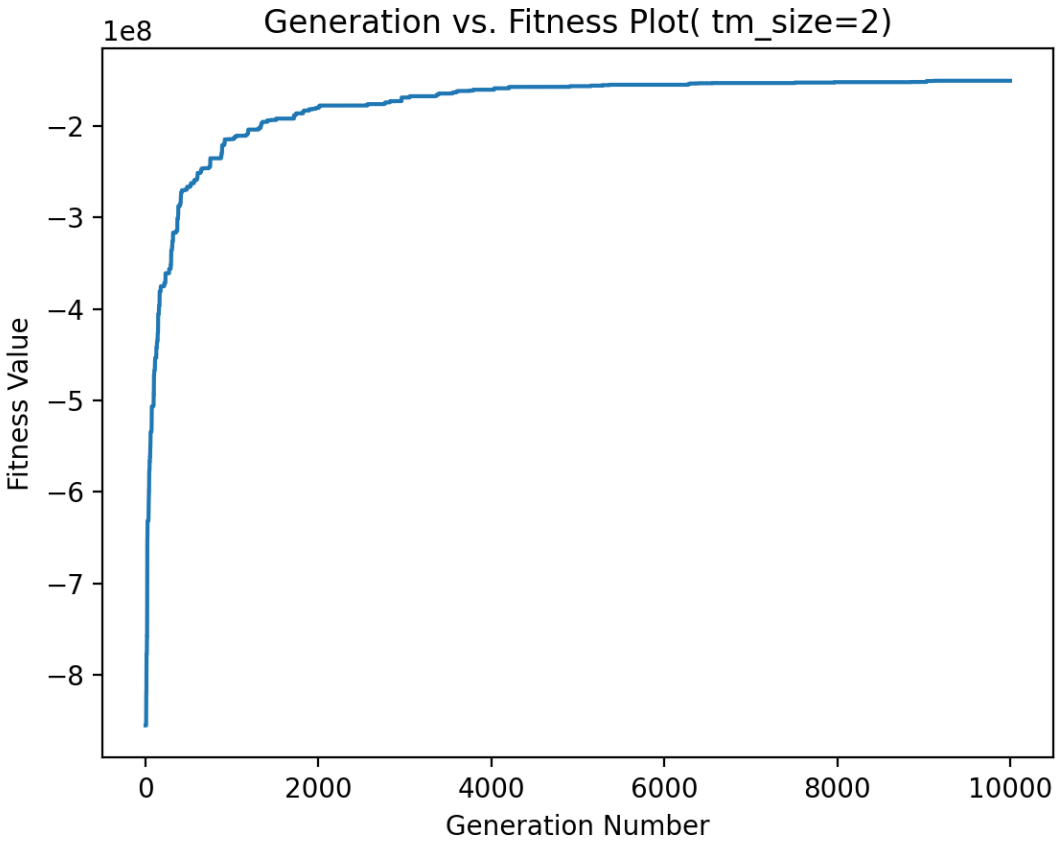
Parameter: <tm_size>:

Parameter <tm_size> represents the number of individuals to enter a tournament. When <tm_size> is too low, individuals with small fitness values may advance to next generation, so, higher values for <tm_size> is preferred. Below figures show that in our case, output is the best when this parameter equals to **20.** However, we must be aware that if number of tournaments to be done were to be higher this could cause an overflow problem since in case of 20, the tournament winner is always the fittest hence tournaments produce the same individuals repetitively.

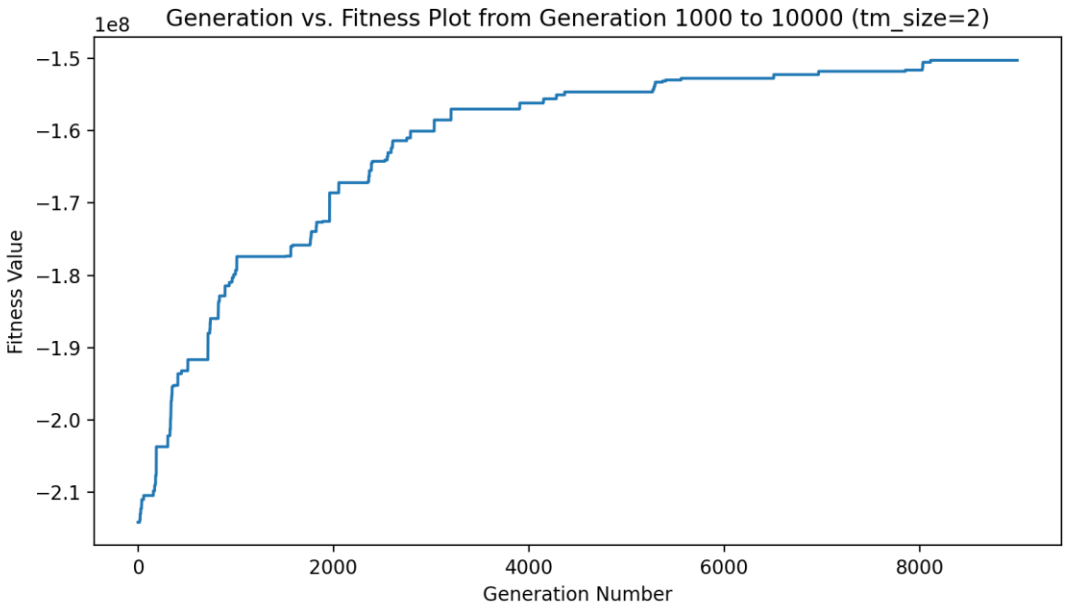
a.2



Gülce Önder
2305159

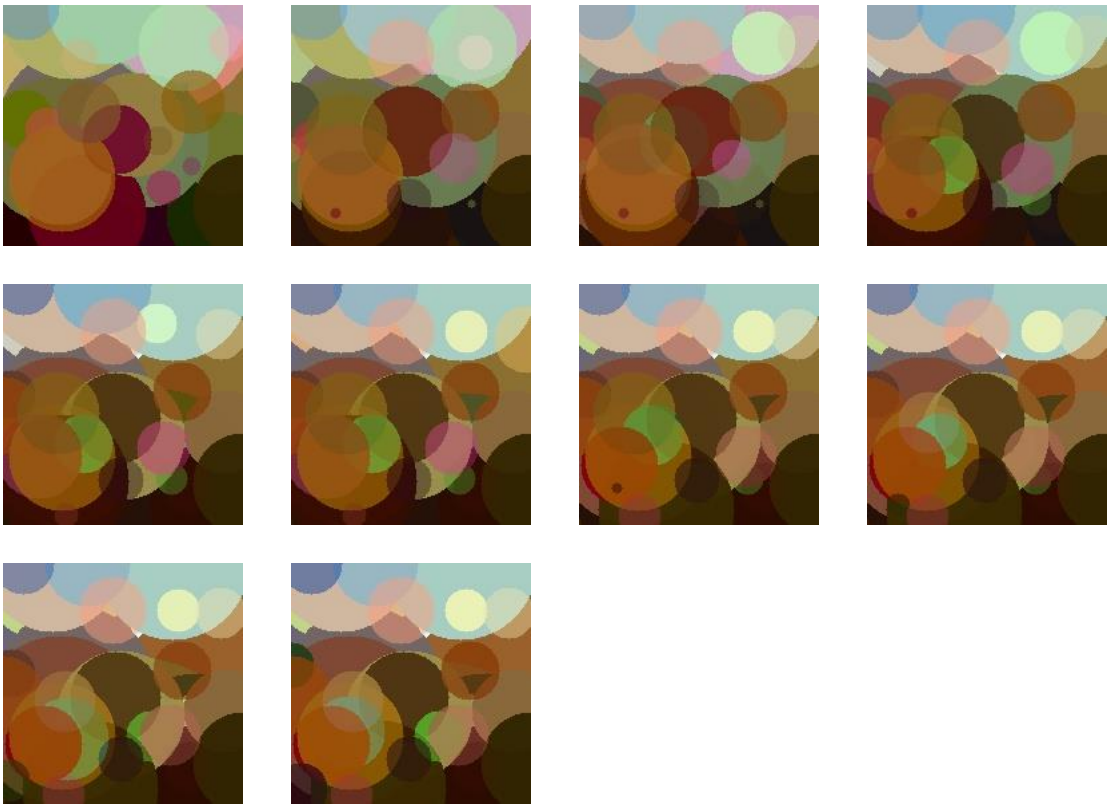


Gülce Önder
2305159

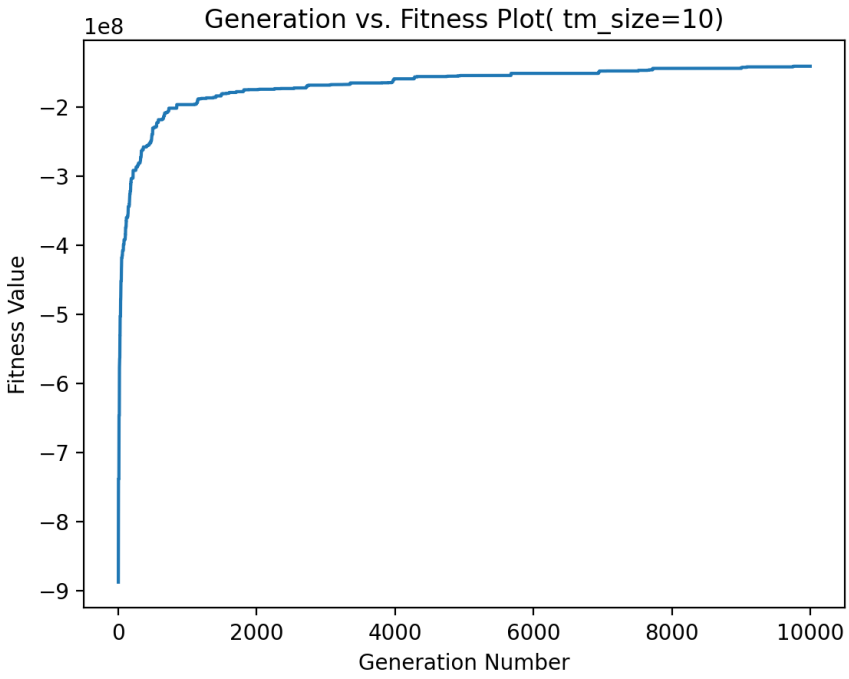


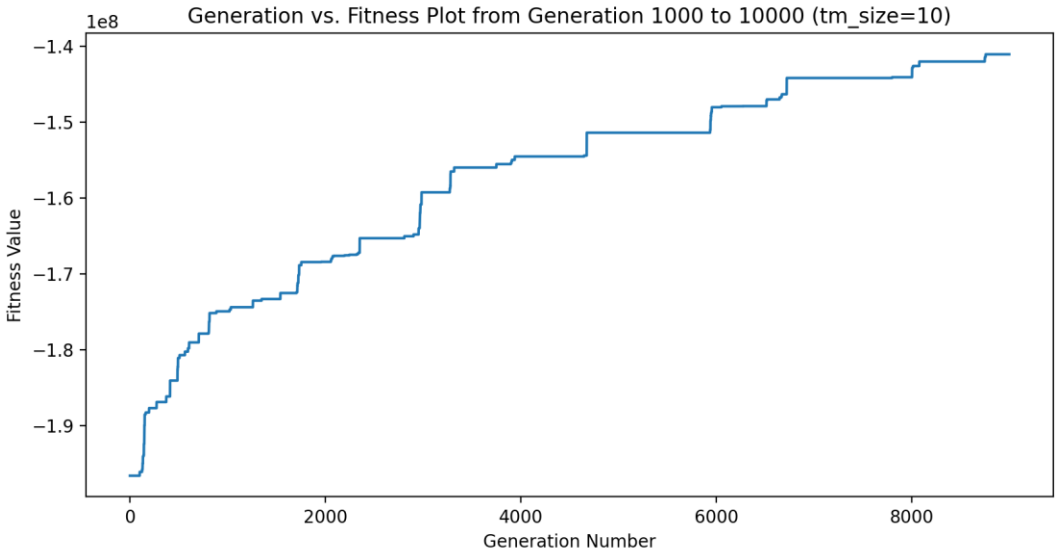
b. 5 (Default)

c. 10



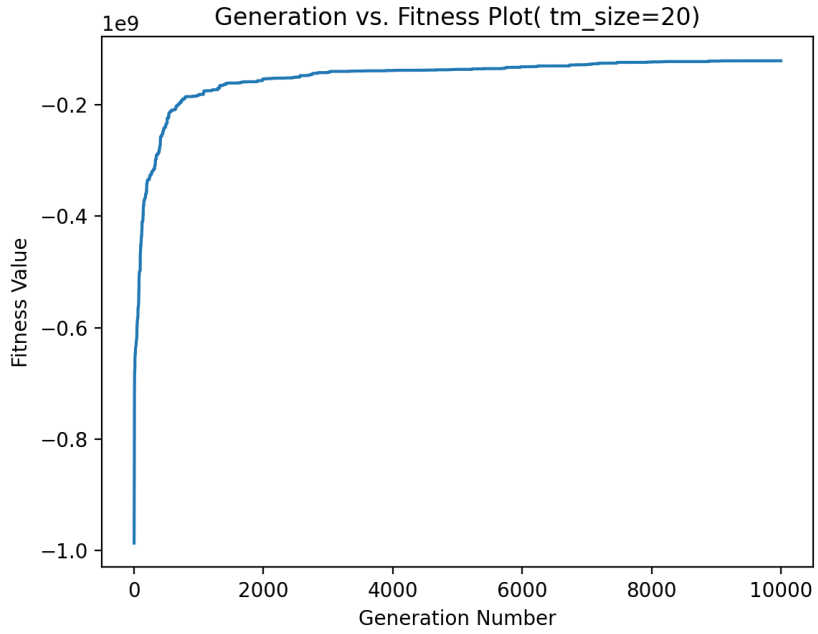
Gülce Önder
2305159



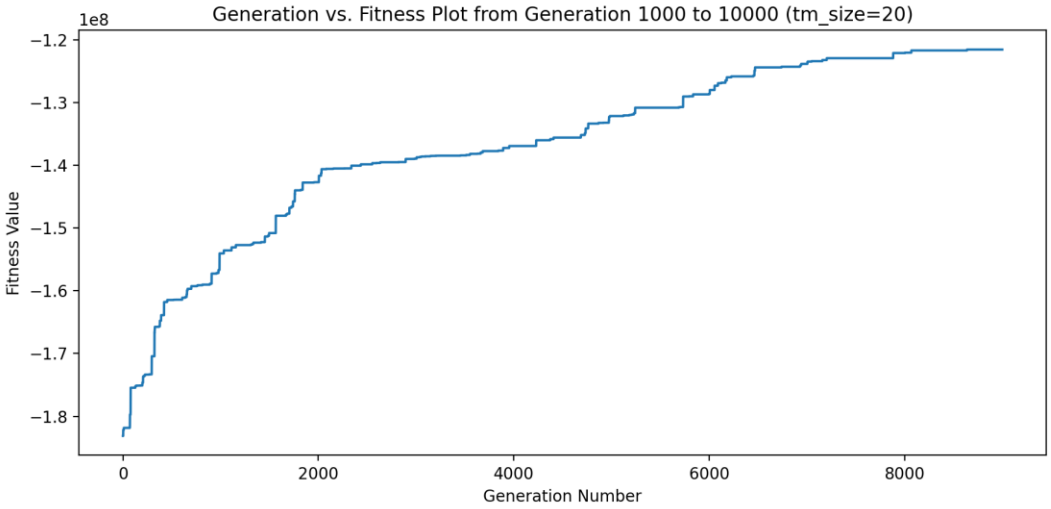


d. 20

Gülce Önder
2305159



Gülce Önder
2305159

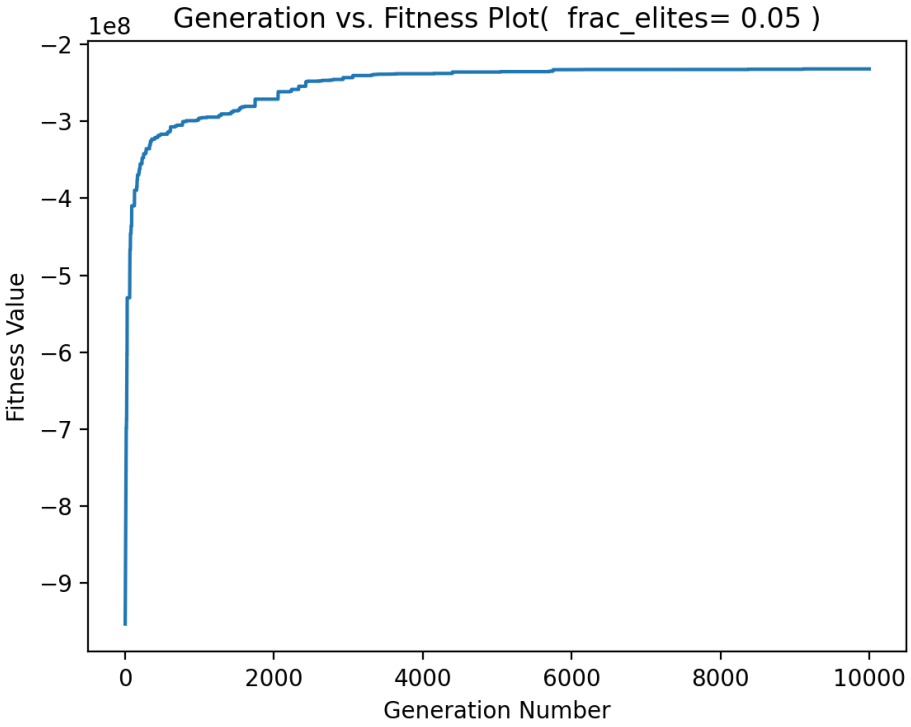


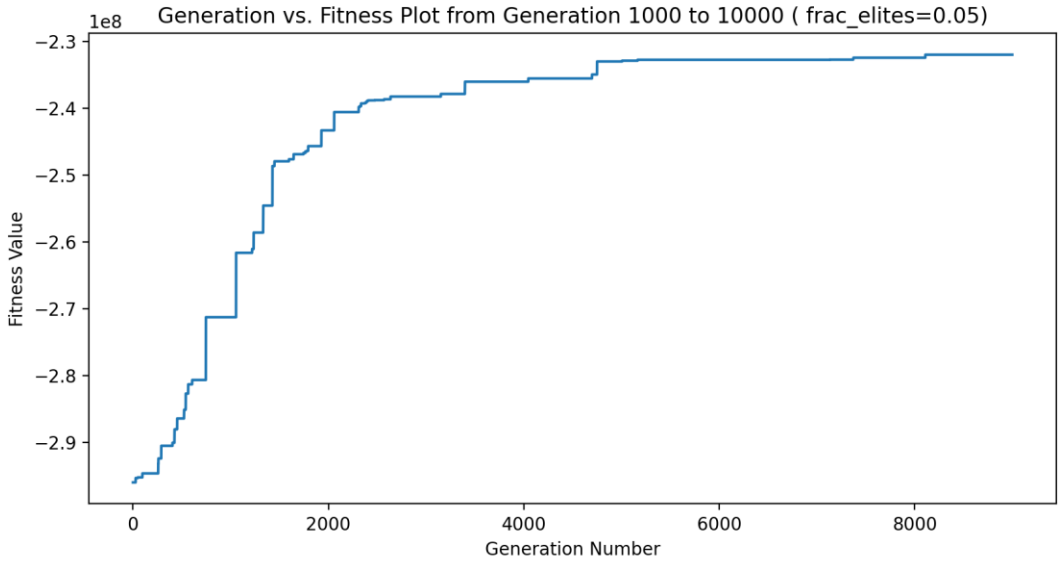
Parameter: <frac_elites>:

<frac_elites> represents the fraction of the population which will remain the same and advanace to the next generation, which are the best fitness individuals. This number must also be choosen carefully as not too high or low. As shown from figures below, **0.2** gave the best results. High value for his parameter means a lot of individuals advance unchanged and this drops efficiency dramatically.

a.0.05

Gülce Önder
2305159

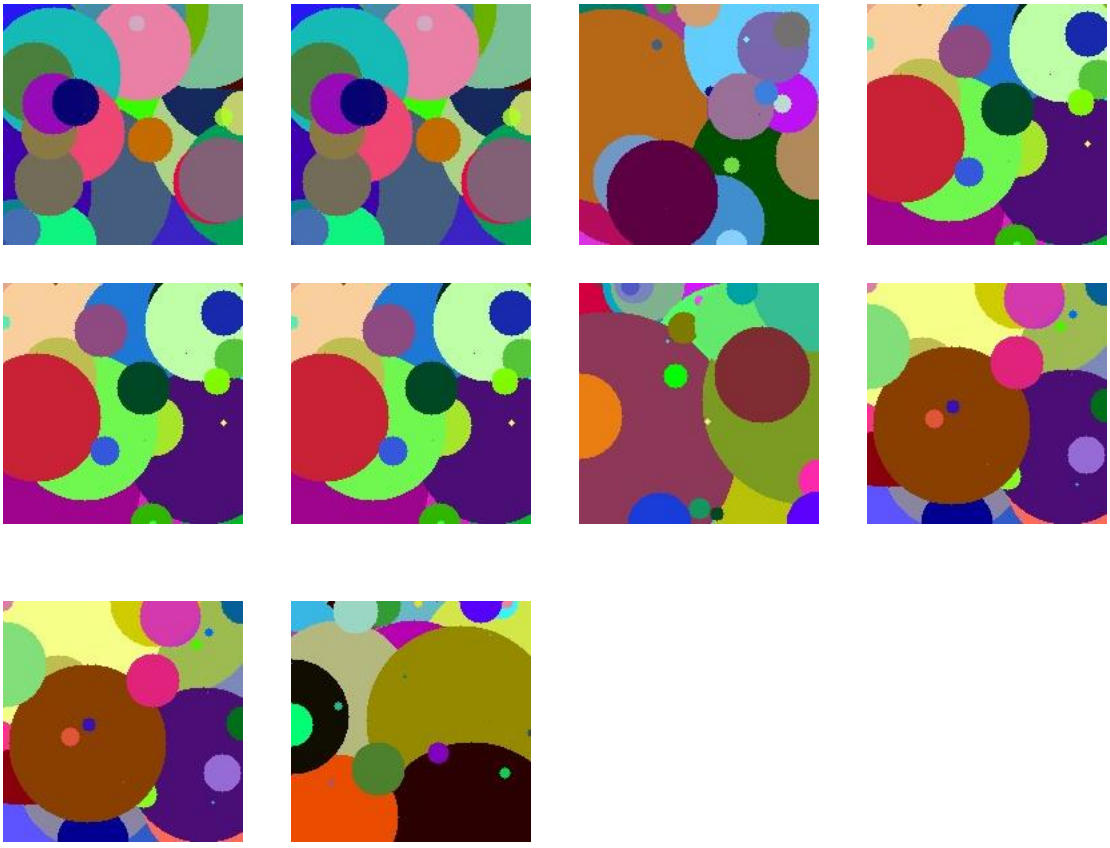




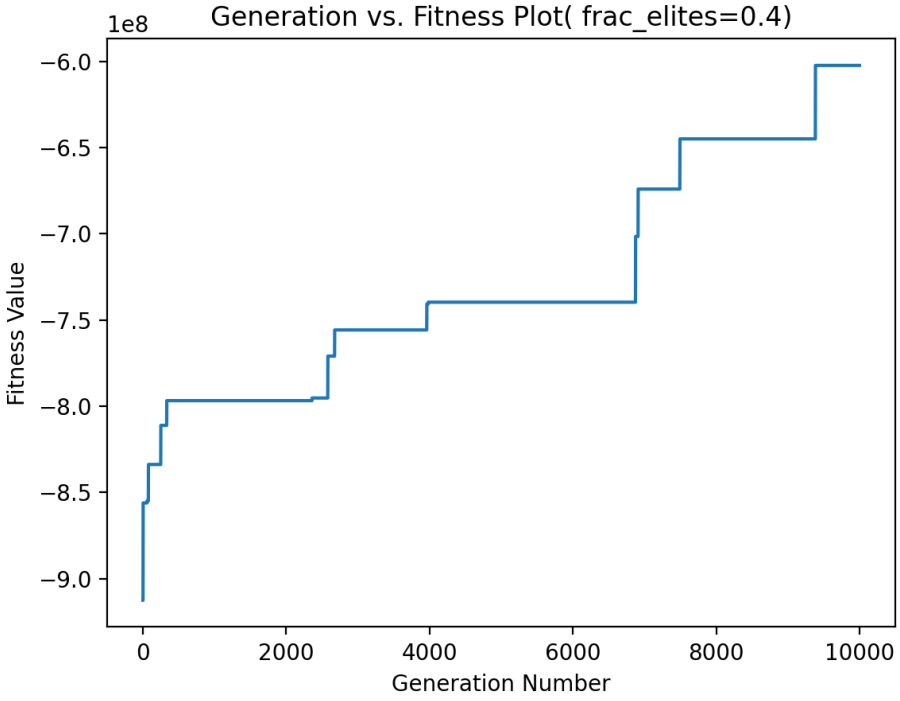
Gülce Önder
2305159

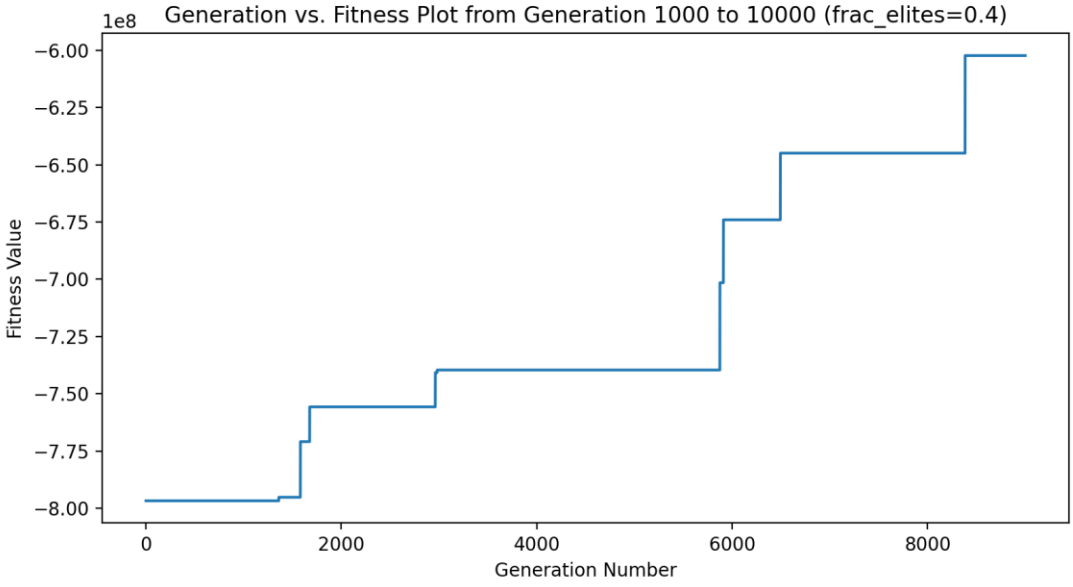
b. 0.2 (Default)

c. 0.4



Gülce Önder
2305159





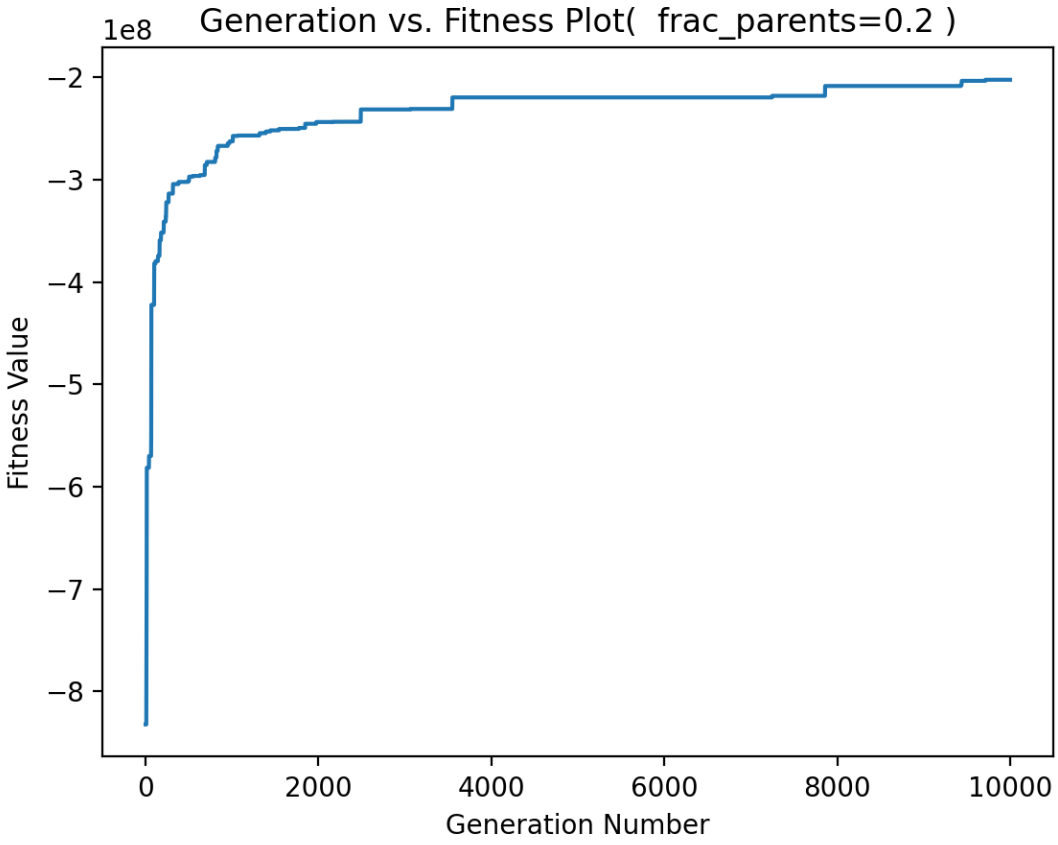
Gülce Önder
2305159

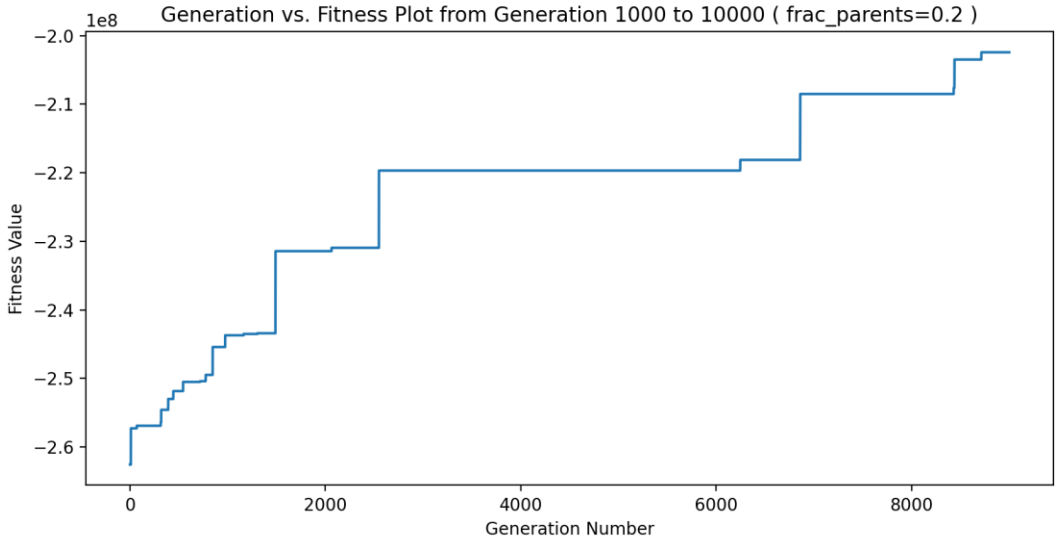
Parameter: <frac_parents>:

<frac_parents> parameter represents the fraction of individuals which will be chosen as parents to create children. In our case the best result is achieved when this parameter equals to **0.4.** This number must be chosen carefully as not too high or too low to avoid killing fit individuals and still keep the benefits of cross-over.

a.0.2

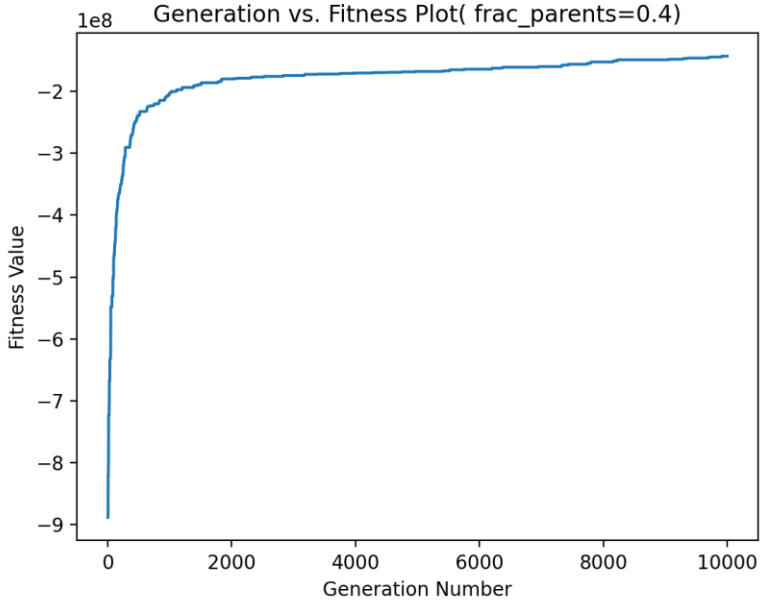
Gülce Önder
2305159



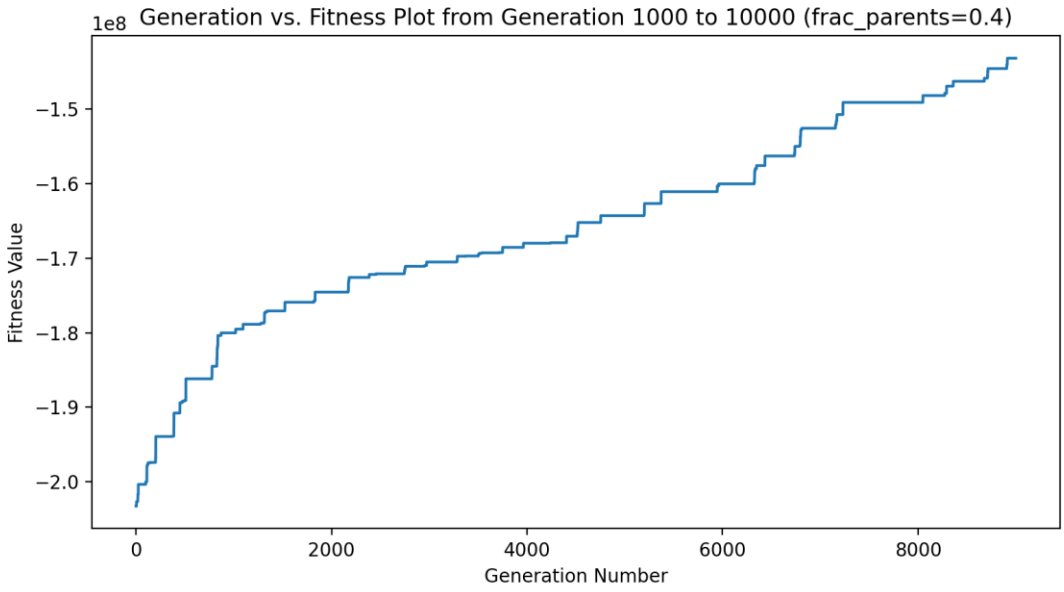


Gülce Önder
2305159

b. 0.4



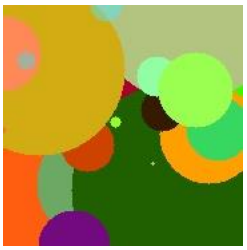
Gülce Önder
2305159

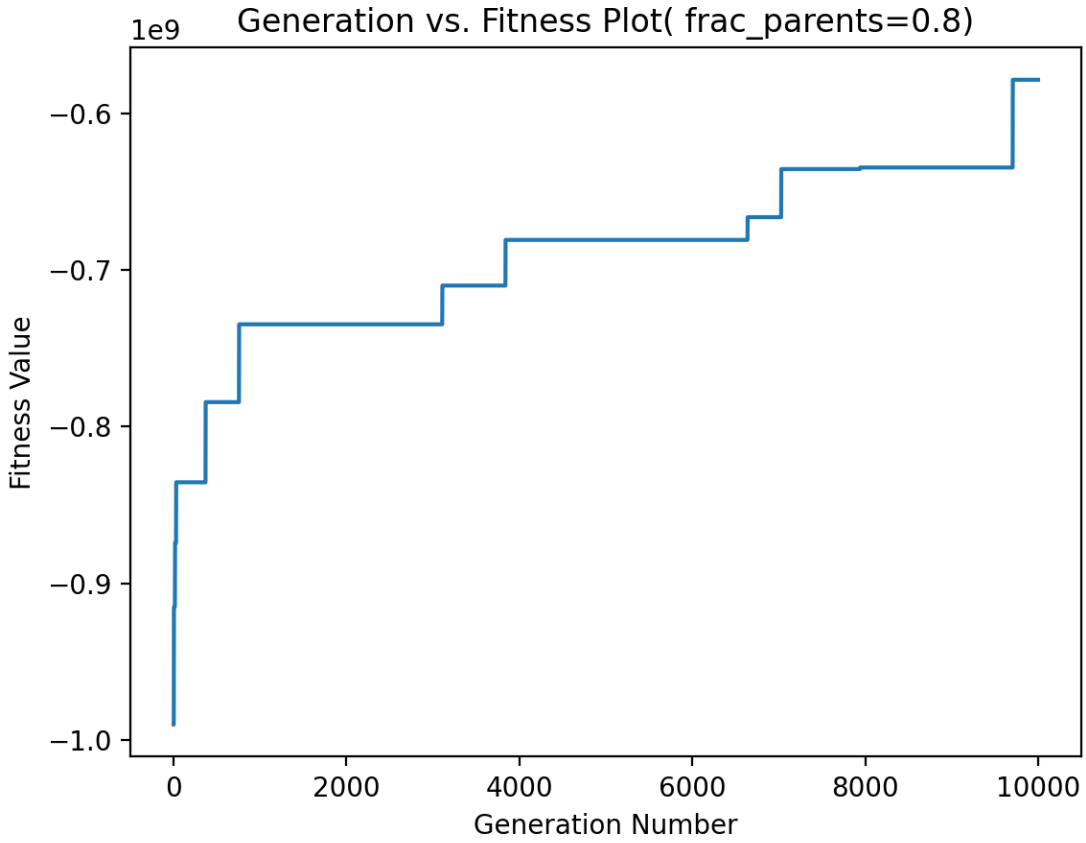


c. 0.6 (Default)

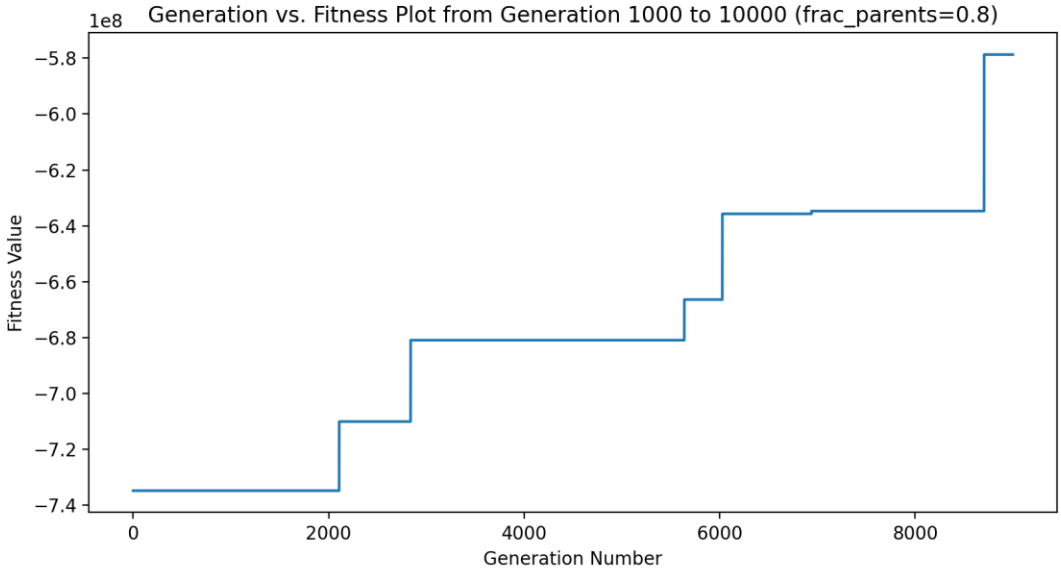
d. 0.8

Gülce Önder
2305159





Gülce Önder
2305159

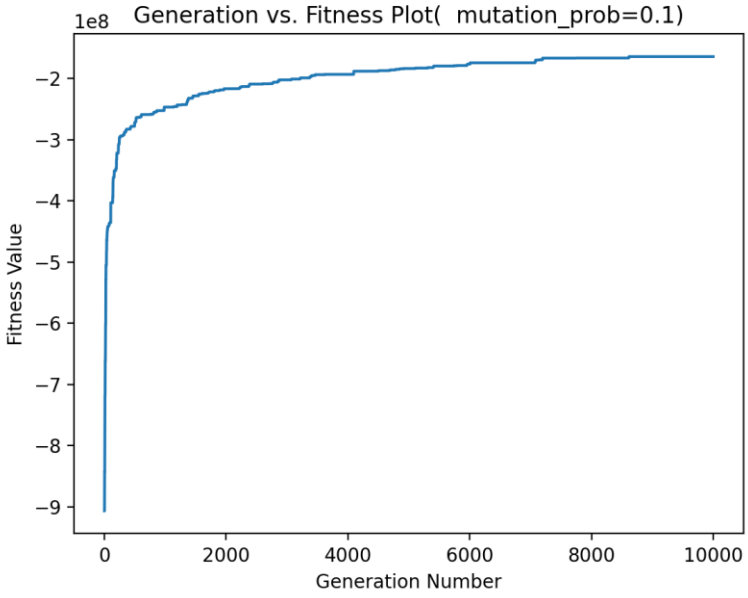


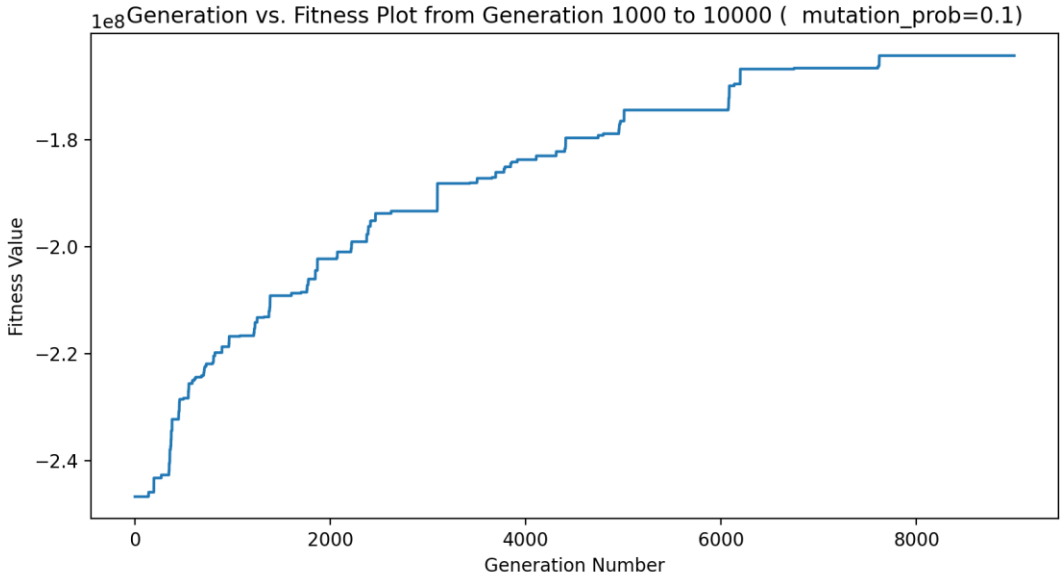
Parameter: <mutation_prob>:

This parameter gives the probability that an individual will be exposed to mutation. As shown from figures below, **0.5** gives the best result. When higher, cross-over and tournament selection benefits are lost. When lower, required variety between individuals can be provided.

a.0.1

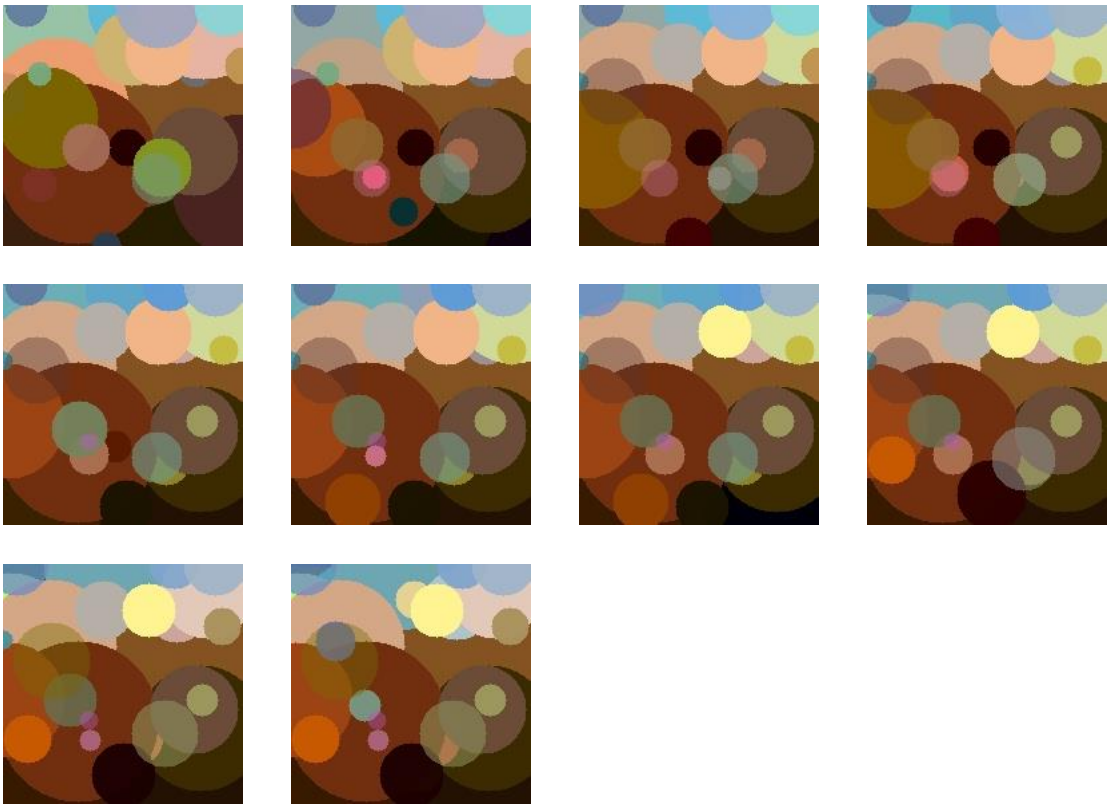
Gülce Önder
2305159



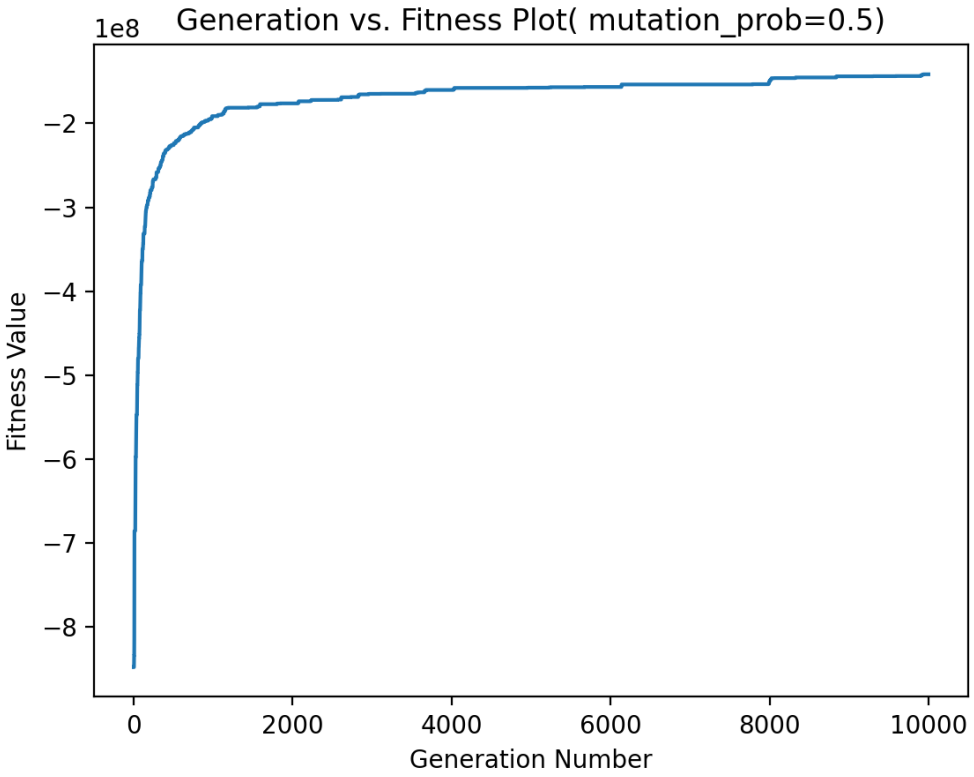


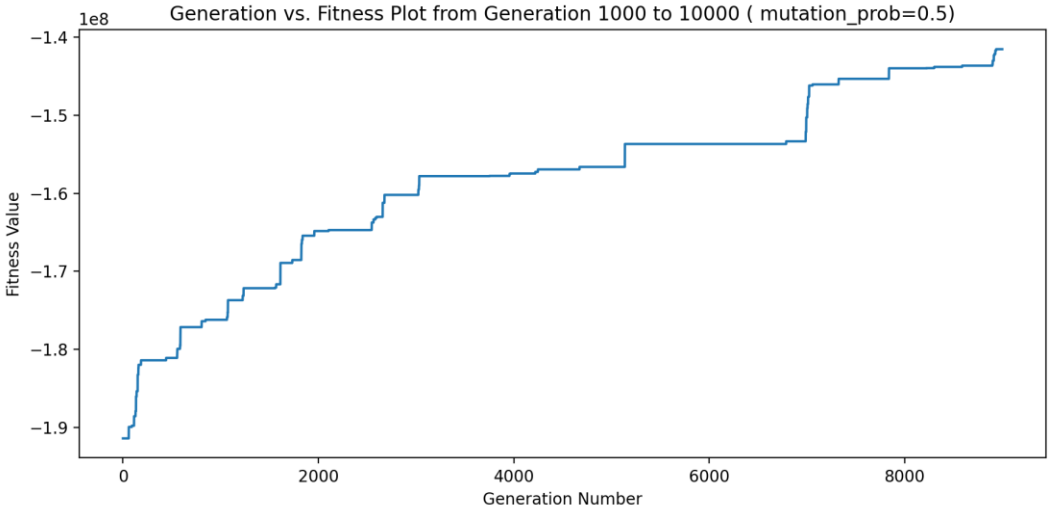
Gülce Önder
2305159

b. 0.2 (Default)
c. 0.5



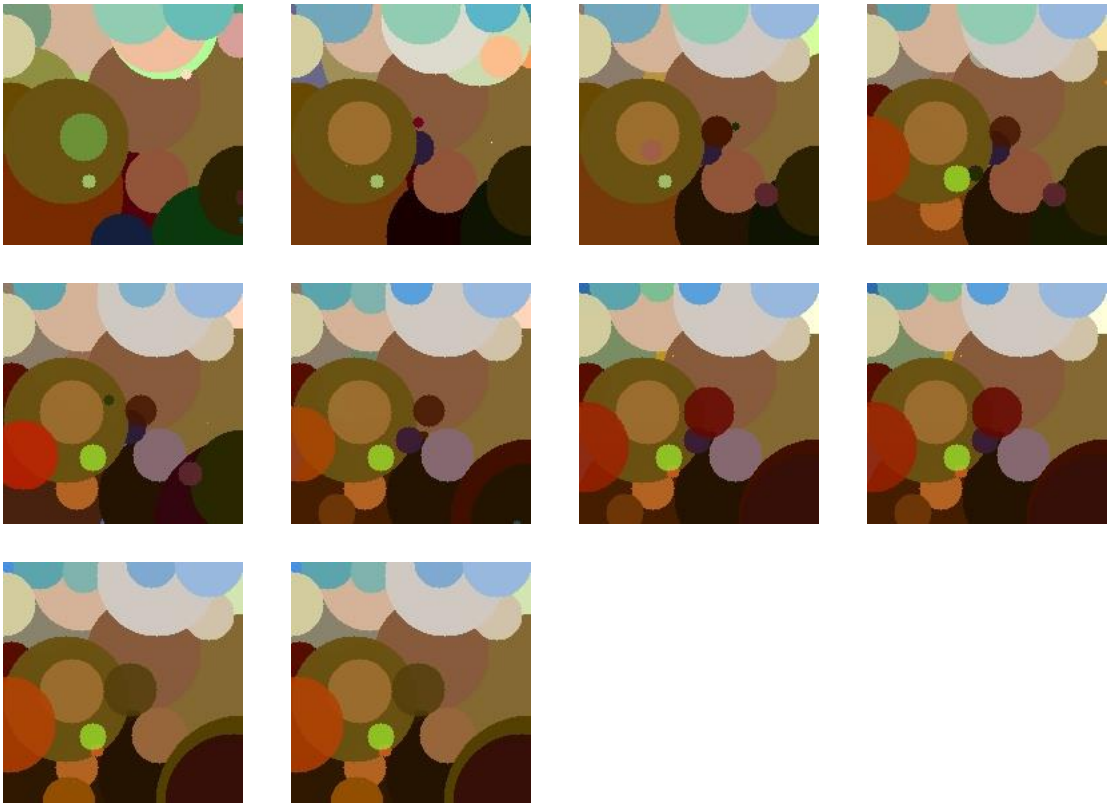
Gülce Önder
2305159

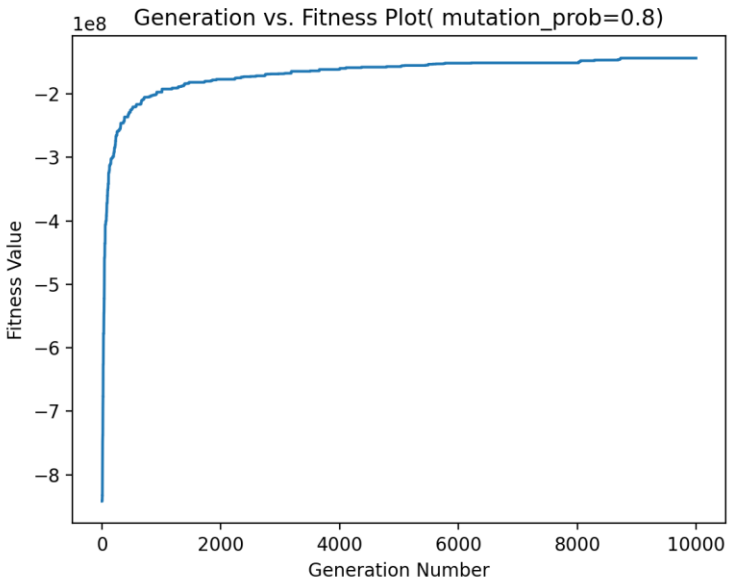




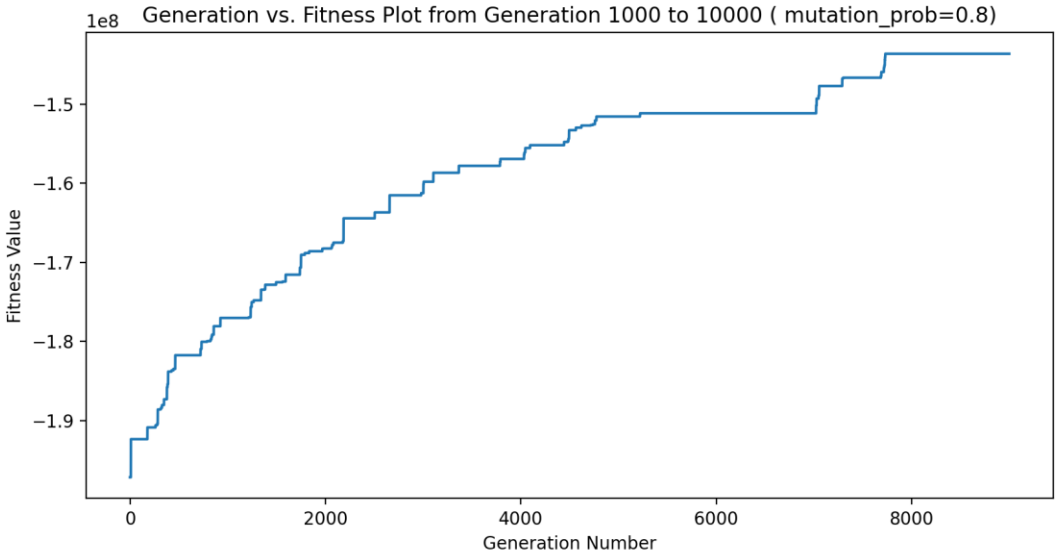
Gülce Önder
2305159

d.0.8





Gülce Önder
2305159



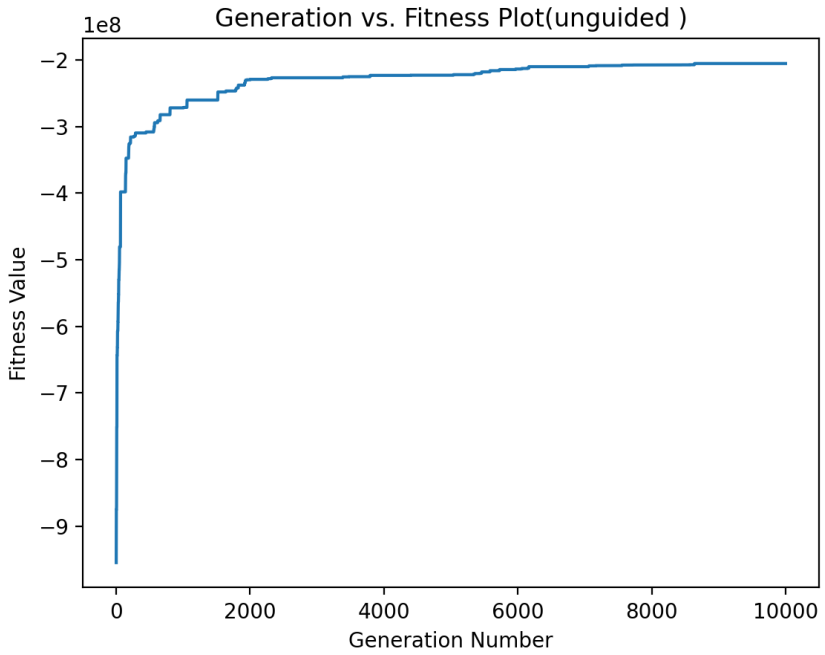
Parameter: ‘guided’ / ‘unguided’

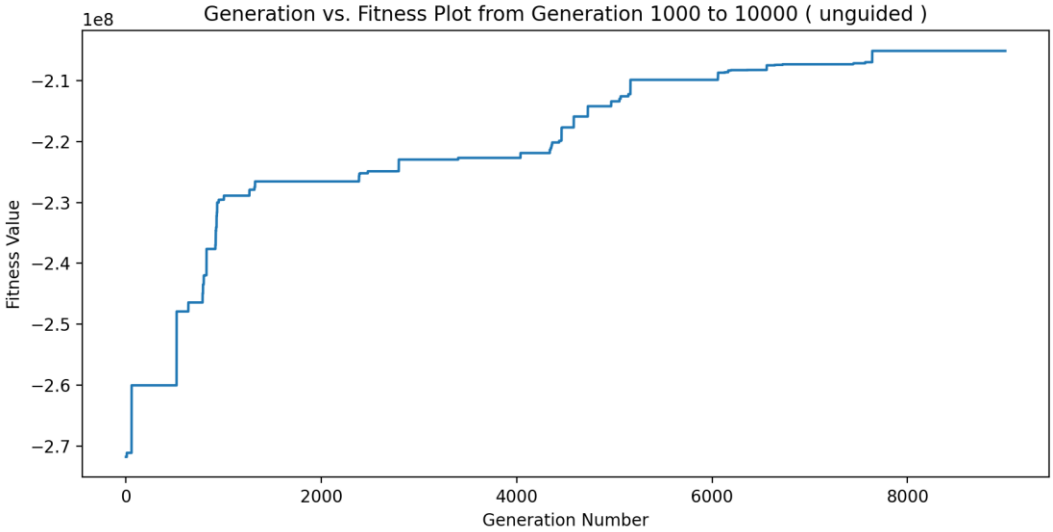
This parameter decides whether the mutation is all random or applied according to individuals current state. As shown from figures guided mutations is more favorable. This is because it helps keep the beneficial traits gained by cross-over and tournaments.

a. ‘guided’ (Default)

b. ‘unguided’

Gülce Önder
2305159





Gülce Önder
2305159

Improving Algorithm

1. New selection methods may be added to the algorithm such as Roulette-wheel Selection
2. Tournament winners could not return to the pool where individuals are selected for another tournament.
3. Gene and individual numbers may be increased. Although it will take more time to run, would give better results significantly.

Gülce Önder
2305159

CODE

```
#include <Python.h>
from __future__ import print_function
import cv2
import copy
from random import uniform
from random import sample
from random import seed
from random import randint
from random import random, randrange
import numpy as np
from operator import attrgetter
from matplotlib import pyplot as plt


# path
path = '/Users/gulceonder/Desktop/evolutionary/painting.png'

# Reading an image in default mode
image = cv2.imread(path)

# Window name in which image is displayed
window_name = 'Image'


blank_image = np.zeros((180,180,3), np.uint8)
blank_image[:,:]= [255,255,255]

def sort_pop(pop):
    pop.ind_list.sort(key=lambda x: x.fitness,reverse=True)
#function to sort the gene list that is genotype according to radius magnitude
def sort_list(ind):
    ind.genotype.sort(key= lambda x: x.r, reverse= True)

#function to draw the circles
def draw_circles(pop,i,k):
    # create two copies of the original image -- one for
    # the overlay and one for the final output image


        # Center coordinates
    center_coordinates = (pop.ind_list[i].genotype[k].x, pop.ind_list[i].genotype[k].y)

    # Radius of circle
    radius = pop.ind_list[i].genotype[k].r
```

Gülce Önder
2305159

```
    # Red color in BGR
    color = (pop.ind_list[i].genotype[k].R, pop.ind_list[i].genotype[k].G, pop.ind_list[i].genotype[k].B)


    # Using cv2.circle() method
    # Draw a circle of red color of thickness -1 px
    cv2.circle(pop.ind_list[i].overlay, center_coordinates, radius, color, -1)

    cv2.addWeighted(pop.ind_list[i].overlay, pop.ind_list[i].genotype[k].A, pop.ind_list[i].output, 1 -
pop.ind_list[i].genotype[k].A,
            0, pop.ind_list[i].output)
    return pop.ind_list[i].output

def check_fitness(output): #check fitness of an individuals output image
    fitness=0


    squared_dist = np.int64(output)-np.int64(image)
    squared_dist = squared_dist*squared_dist
    fitness=np.sum(squared_dist,dtype=np.int64)

    # print("FITNESS1 : " ,fitness)

    # for i in range(180):
    #     for j in range(180):
    #         for k in range(3):
    #             x=(np.int64(output[i,j,k])- np.int64(image[i,j,k]))**2 #convert uint8 to int64 to avoid overflow
    #             fitness2= fitness2+x


    # print("FITNESS2 : " ,fitness2)

    return fitness*(-1)

def evaluate_pop(pop, num_inds):
        for i in range(num_inds):
            #print("enter evaluation")

            pop.ind_list[i].overlay = blank_image.copy()
            pop.ind_list[i].output = blank_image.copy()

            for k in range(pop.ind_list[i].num_genes):
            #self.genotype[i].print_gene()
                #pop.ind_list[i].output=draw_circles(pop.ind_list[i].genotype[k],pop.ind_list[i].overlay,pop.ind_list[i].output)
                pop.ind_list[i].output=draw_circles(pop,i,k)
```

Gülce Önder
2305159

```
        pop.ind_list[i].fitness= check_fitness(pop.ind_list[i].output)

    #     cv2.imshow("Output", output)

    #     cv2.waitKey(0)

    # # closing all open windows

    #     cv2.destroyAllWindows()
    #     cv2.waitKey(1)

    #     #print("indv DONE")

def evaluate_ind2(ind):
        #print("enter evaluation")
        ind.overlay = blank_image.copy()
        ind.output = blank_image.copy()

        for k in range(ind.num_genes):
        #self.genotype[i].print_gene()
            output=draw_circles2(ind,k)

        ind.fitness= check_fitness(ind.output)

def draw_circles2(ind,k):
    # create two copies of the original image -- one for
    # the overlay and one for the final output image


        # Center coordinates
    center_coordinates = (ind.genotype[k].x, ind.genotype[k].y)

    # Radius of circle
    radius = ind.genotype[k].r


    # Red color in BGR
    color = (ind.genotype[k].R, ind.genotype[k].G, ind.genotype[k].B)


    # Using cv2.circle() method
    # Draw a circle of red color of thickness -1 px
    cv2.circle(ind.overlay, center_coordinates, radius, color, -1)

    cv2.addWeighted(ind.overlay, ind.genotype[k].A, ind.output, 1 - ind.genotype[k].A,
        0, ind.output)
    return ind.output
```

Gülce Önder
2305159

```
def cross_over(mom,dad,child1,child2,num_genes):
    #child.num_genes=10
    #assign genes from mom or dad with prob 0.5 for each gene
    for i in range(num_genes):
        x= random()
        if(x<0.5):#do with 0.5 prob
            child1.genotype.append(mom.genotype[i])
            child2.genotype.append(dad.genotype[i])

        else:
            child1.genotype.append(dad.genotype[i])
            child2.genotype.append(mom.genotype[i])


def unguided_mutant(pop,i, gene_index):
    pop.ind_list[i].genotype[gene_index].x = randint(0,180)
    pop.ind_list[i].genotype[gene_index].y = randint(0,180)
    pop.ind_list[i].genotype[gene_index].r = randint(0,100)
    pop.ind_list[i].genotype[gene_index].R = randint(0,255)
    pop.ind_list[i].genotype[gene_index].G = randint(0,255)
    pop.ind_list[i].genotype[gene_index].B = randint(0,255)
    pop.ind_list[i].genotype[gene_index].A = random()


def guided_mutant(pop,i, gene_index):
    #print("new mutant RGB")
    mutantgene=copy.deepcopy(pop.ind_list[i].genotype[gene_index])
    number = randrange(-64, 64)
    if(number+mutantgene.R<0):mutantgene.R=0
    elif(number+mutantgene.R>255):mutantgene.R=255
    else:mutantgene.R+= number
    #print(mutantgene.R)
    number = randrange(-64, 64)
    if(number+mutantgene.G<0):mutantgene.G=0
    elif(number+mutantgene.G>255):mutantgene.G=255
    else:mutantgene.G+= number
    #print(mutantgene.G)
    number = randrange(-64, 64)
    if(number+mutantgene.B<0):mutantgene.B=0
    elif(number+mutantgene.B>255):mutantgene.B=255
    else:mutantgene.B+= number
    #print(mutantgene.B)
    number = randrange(-45, 45)
    if(number+mutantgene.x<0):mutantgene.x=0
    elif(number+mutantgene.x>180):mutantgene.x=180
```

Gülce Önder
2305159

```
    else:mutantgene.x+= number
    #print(mutantgene.x)

    number = randrange(-45, 45)
    if(number+mutantgene.y<0):mutantgene.y=0
    elif(number+mutantgene.y>180):mutantgene.y=180
    else:mutantgene.y+= number
    #print(mutantgene.y)

    number = randrange(-10, 10)
    if(number+mutantgene.r<0):mutantgene.r=0

    else:mutantgene.r+= number
    #print(mutantgene.r)

    number = uniform(-0.25, 0.25)
    if(number+mutantgene.A<0):mutantgene.A=0
    elif(number+mutantgene.A>1):mutantgene.A=1
    else:mutantgene.A+= number
    #print(mutantgene.A)

    pop.ind_list[i].genotype[gene_index]=mutantgene



class gene:

#call constructor according to the given argument
    def __init__(self, *args): #if no argument initiliaze to 0
        if len(args) == 0:
            self.x= 0
            self.y= 0
            self.r= 0
            self.R= 0
            self.G= 0
            self.B= 0
            self.A= 0
        elif len(args) == 7:#assign values to each gene
            self.x= args[0]
            self.y= args[1]
            self.r= args[2]
            self.R= args[3]
            self.G= args[4]
            self.B= args[5]
            self.A= args[6]
        elif args[0] == 'rand': #initiliaze gene randomly
```

Gülce Önder
2305159

```
        seed(random())
        self.x = randint(0,180)
        self.y = randint(0,180)
        self.r = randint(0,100)
        self.R = randint(0,255)
        self.G = randint(0,255)
        self.B = randint(0,255)
        self.A = random()

    def print_gene(self):
        print(self.x)
        print(self.y)
        print(self.r)
        print(self.R)
        print(self.G)
        print(self.B)
        print(self.A)



class ind :

    def __init__(self, num_genes):
        if(num_genes):
            self.num_genes=num_genes
            self.genotype = []#list of genes for individual
            self.fitness = 0
            for i in range(num_genes):
                randomgene= gene('rand')
                self.genotype.append(randomgene) #check radius first

            sort_list(self)
            self.overlay = blank_image.copy()
            self.output = blank_image.copy()

            for i in range(num_genes):
            #self.genotype[i].print_gene()
                draw_circles2(self,i)

            self.fitness= check_fitness(self.output)
            #print(self.fitness)
            cv2.imshow("Output", self.output)

            cv2.waitKey(0)

        # closing all open windows
```

Gülce Önder
2305159

```
            cv2.destroyAllWindows()
            cv2.waitKey(1)

            #print("indv DONE")
        else:
            self.genotype=[]
            self.fitness=0
            self.num_genes=num_genes
            self.overlay = blank_image.copy()
            self.output = blank_image.copy()



class pop:
    def __init__(self,num_inds,num_genes):
        if(num_genes):
            self.num_inds = num_inds
            self.ind_list = []
            for i in range(num_inds):
                random_ind = ind(num_genes)
                self.ind_list.append(random_ind)
        else:
            self.num_inds = num_inds
            self.ind_list = []



#main function to run
def main():
    #enter parameters
    fittest_list=[]
    num_inds=20
    num_genes=50
    tm_size=5
    frac_elites = 0.05
    frac_parents=0.2
    mutation_prob=0.2
    num_generations=10001
    mutation_type='unguided'
    num_elites= int(num_inds*frac_elites)
    num_parents=int(num_inds*frac_parents)
    num_birth=int(num_parents/2)
    num_tm= num_inds- num_elites-num_parents#number of tournaments to be done

    #initialize  adn evaluate random population
    pop1 = pop(num_inds,num_genes)
```

Gülce Önder
2305159

```
gen=1
found= False


while not (gen==num_generations):
    sort_pop(pop1)


    pop2= pop(num_inds,0)#generation containing children and tournament winners
    pop3=pop(num_inds,0)#new gen

    pop2.num_inds= num_inds-num_elites

    pop3.num_inds= num_inds


    #apply elitism :pick the best num_elites and insert into the next generation
    for i in range(num_elites):

        # print("elite ",i ,", fitness:  ",pop1.ind_list[i].fitness)
        pop3.ind_list.append(copy.deepcopy(pop1.ind_list[i]))#elites advance directlty to new gen

    print("report the best fitness of gen ", gen," : ",pop1.ind_list[0].fitness )
    fittest_list.append(pop1.ind_list[0].fitness)
    if(gen%1000==0):
        # Filename
        filename = 'savedImage' + str(gen)+ '.jpg'




        # Using cv2.imwrite() method
        # Saving the image
        cv2.imwrite(filename, pop1.ind_list[0].output)
    #     cv2.imshow("Output", pop1.ind_list[0].output)

    #     cv2.waitKey(0)
    # # # closing  open windows

    #     cv2.destroyAllWindows()
    #     cv2.waitKey(1)


    #enter tournament
    for i in range(num_tm):
        tournament_list=sample(pop1.ind_list[num_elites:],tm_size) #randomly pick (tournament size) number of
individuals from population
```

Gülce Önder
2305159

```
        tournament_list.sort(key=lambda x: x.fitness,reverse=True)#sort tournmnt list
        tournament_winner= max(tournament_list, key=attrgetter('fitness'))

        # print("gen: ", gen, " tournament winner: ",tournament_winner.fitness)
        pop2.ind_list.append(copy.deepcopy(tournament_winner))
        #print(len(pop2.ind_list))

    #initliaze cross-over phase
    for i in range(num_birth):
        parents=pop1.ind_list[num_elites:(num_elites+num_parents)]#pick parents from the fittest individuals which
were not picked as elites
        mom= parents[i]#pick two parents for the current child-to-be
        dad = parents[num_parents-i-1]
        child1 = ind(0) #initiliaze new individual child with empty genotype
        child2 = ind(0)
        child1.num_genes=num_genes
        child2.num_genes=num_genes

        cross_over(mom,dad,child1,child2,num_genes)
        pop2.ind_list.append(copy.deepcopy(child1))#add child to new pop
        pop2.ind_list.append(copy.deepcopy(child2))#add child to new pop


    #enter mutation phase
    #population 2 holds the population to be exposed to mutation which excludes elite individuals
    #genes are picked randomly with gene_index



    if(mutation_type=='unguided'):
        for i in range(len(pop2.ind_list)):
            mut=random()
            #mutant= pop2.ind_list[i].genotype#extract genotype of thr current individual

            if(mut<mutation_prob):
                gene_index=randint(0, num_genes-1)


                unguided_mutant(pop2,i,gene_index)



    elif(mutation_type=='guided') :
        for i in range(len(pop2.ind_list)):
            mut=random()
```

Gülce Önder
2305159

```
            if(mut<mutation_prob):
                gene_index=randint(0, num_genes-1)

                guided_mutant(pop2,i,gene_index)



#evaluate population when all individuals are done
        evaluate_pop(pop2,pop2.num_inds)
        # print("pop2 after mutation: ")
        # for k in range(len(pop2.ind_list)):
        #     #evaluate_ind(pop2.ind_list[i])
        #     print(pop2.ind_list[k].fitness)

        for k in range(len(pop2.ind_list)):
            #evaluate_ind(pop2.ind_list[i])
            pop3.ind_list.append(copy.deepcopy(pop2.ind_list[k]))



        pop1=pop(num_inds,0)
        pop1 = copy.deepcopy(pop3)
        # for i in range(num_inds):


        #     print("last pop indv ", i)
        #     print(pop1.ind_list[i].fitness)

        del pop2
        del pop3
        gen +=1


#PLOT THE FITNESS LISTS

    plt.plot(fittest_list)
    plt.xlabel('Generation Number')
    plt.ylabel('Fitness Value')
    plt.title('Generation vs. Fitness Plot(  frac_elites= 0.05 )')
    plt.show()

    fit2= fittest_list[999:]
    plt.plot(fit2)

    plt.xlabel('Generation Number')
```

Gülce Önder
2305159

```
    plt.ylabel('Fitness Value')
    plt.title('Generation vs. Fitness Plot from Generation 1000 to 10000 ( frac_elites=0.05)')
    plt.show()
    #     sort pop according to fitness
    #     carry first num_elites to next pop.
    #     randomly pick from sorted population to create offspring
    #     recomb for offsprings
    #     mutate some of offsprings
    #     evaluate new gen
    #
    #return best indv of pop.
if __name__ == '__main__':
    main()
```