Alper Kutay ÖZBEK
2375467

# 1. Experimental Results

## 1.1. Number of Individuals

Number of generations = 1000

**Figure 1** : Number of Individuals is 5

**Figure 2** : Number of Individuals is 10

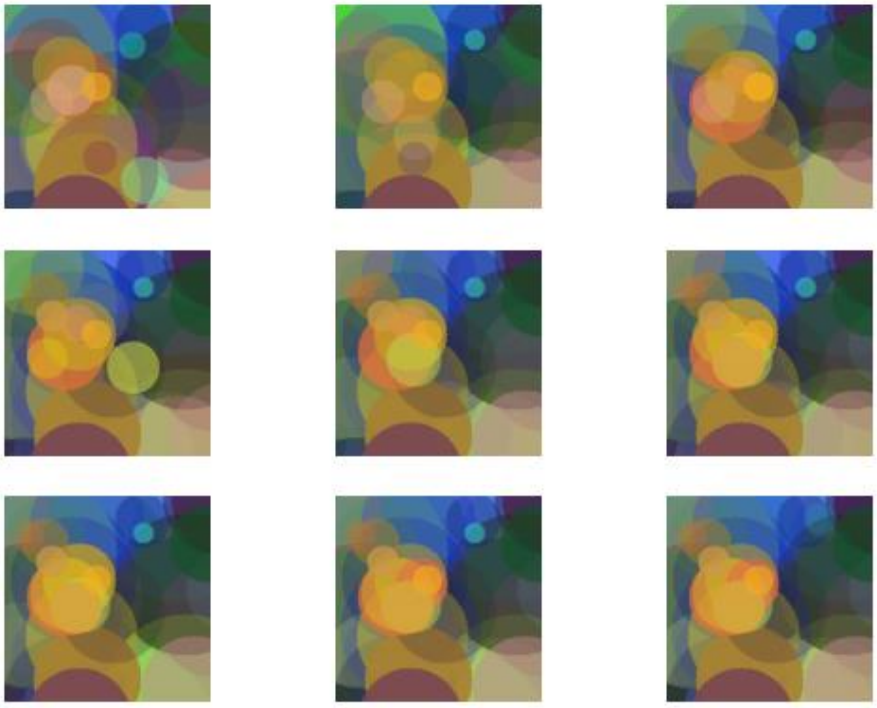
**Figure 3** : Default values

**Figure 4** : Number of Individuals is 40

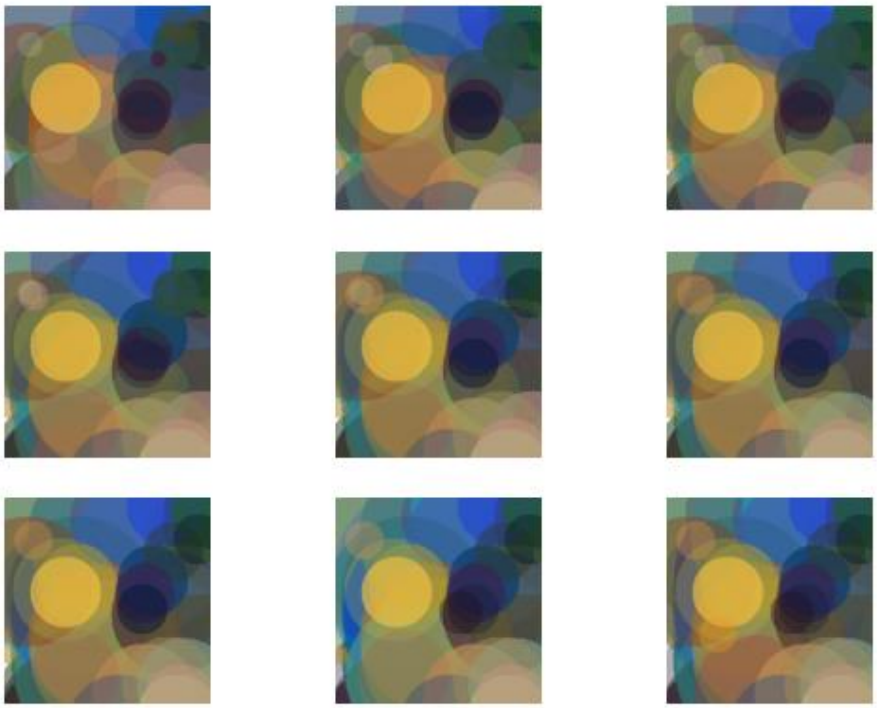
**Figure 5** : Number of Individuals is 60

Alper Kutay ÖZBEK
2375467

num_inds/5/Iterations: 2000 - 10000

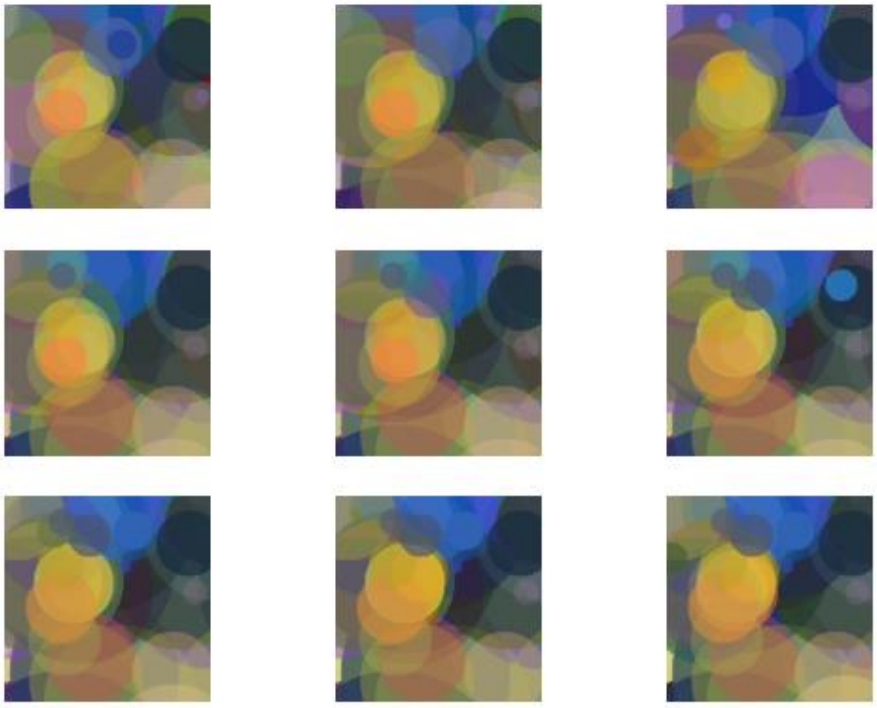


num_inds/10/Iterations: 2000 - 10000

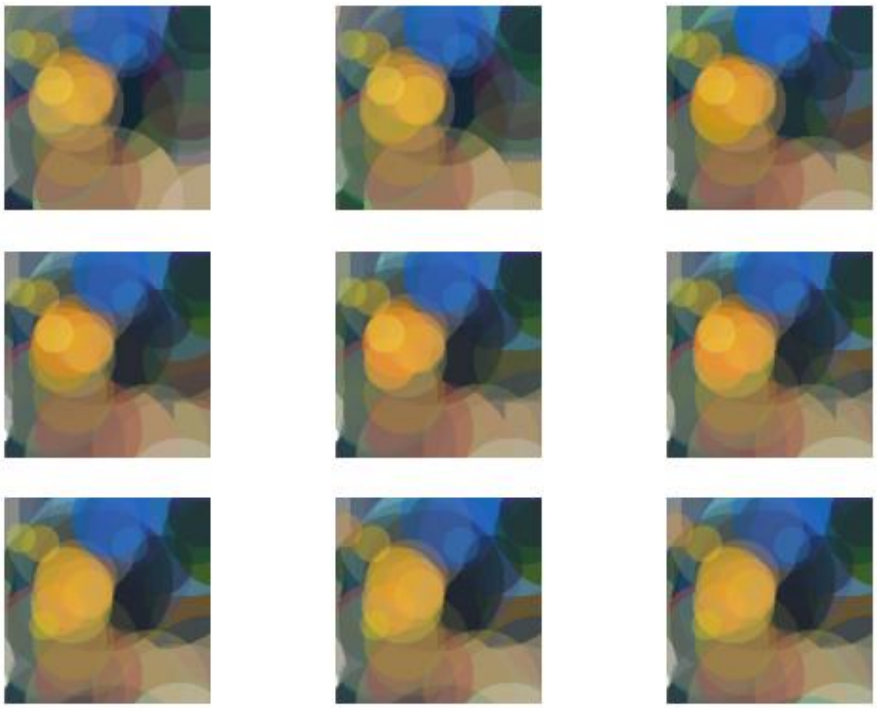


Alper Kutay ÖZBEK
2375467

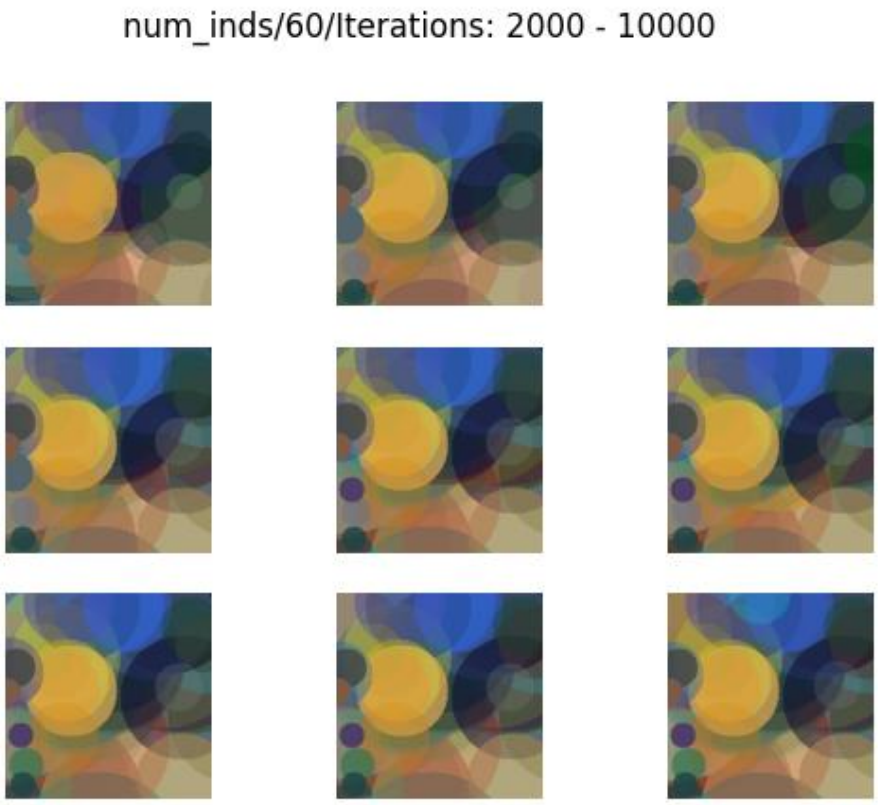
num_inds/20/Iterations: 2000 - 10000

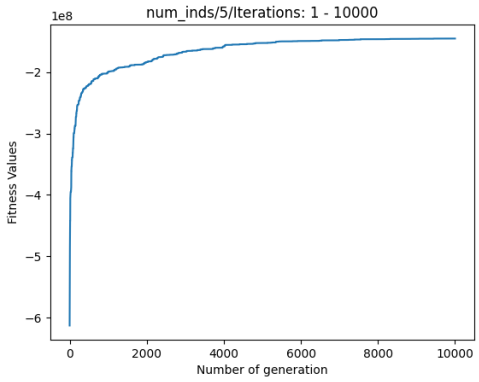


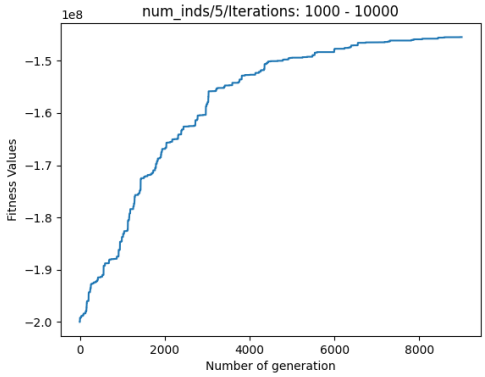
num_inds/40/Iterations: 2000 - 10000

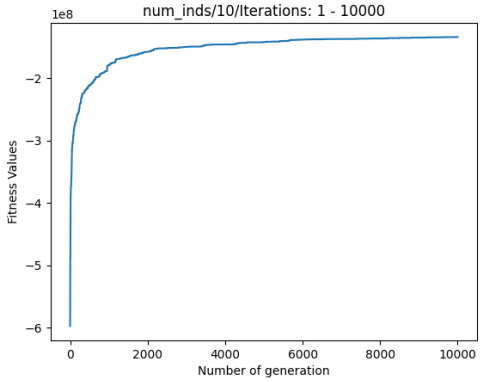


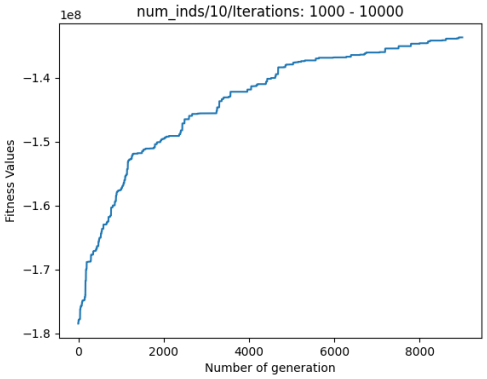
Alper Kutay ÖZBEK
2375467



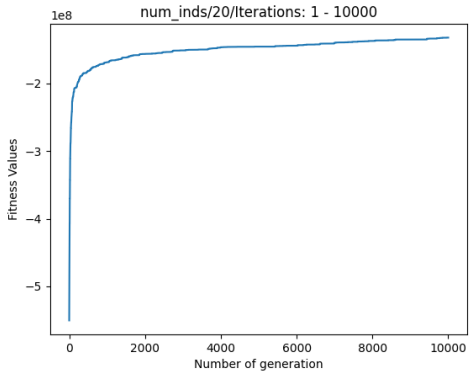



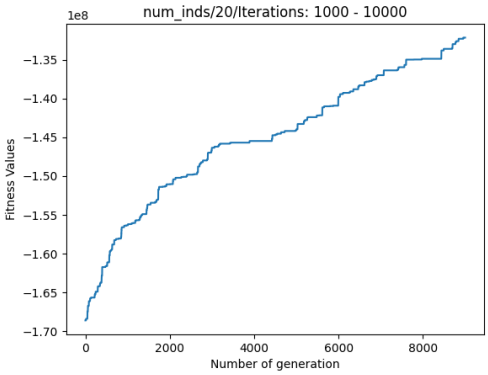



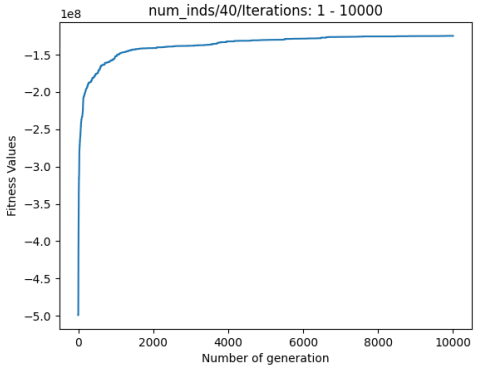



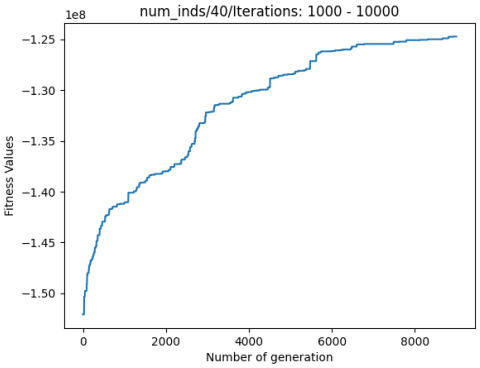



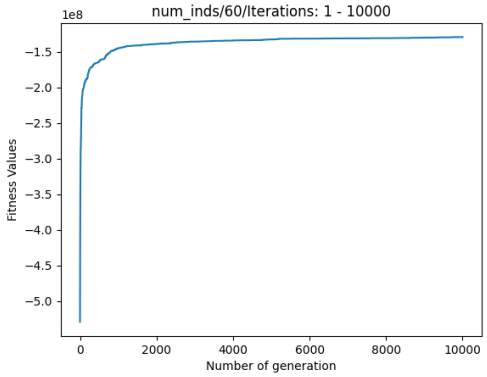
Alper Kutay ÖZBEK
2375467

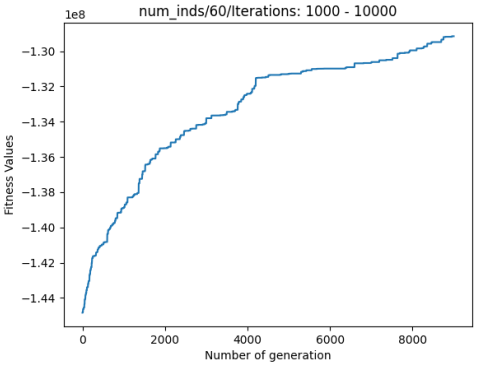


Alper Kutay ÖZBEK
2375467

1.2. Number of Genes

Number of generations = 1000

**Figure 6** : Number of Genes is 15

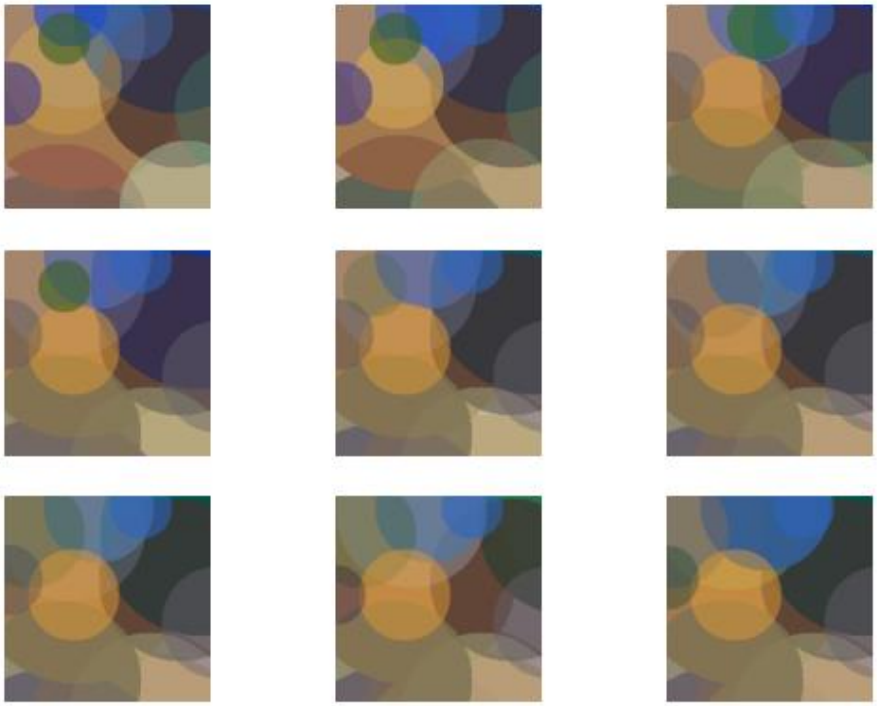
**Figure 7** : Number of Genes is 30

**Figure 8**: Number of Genes is 80

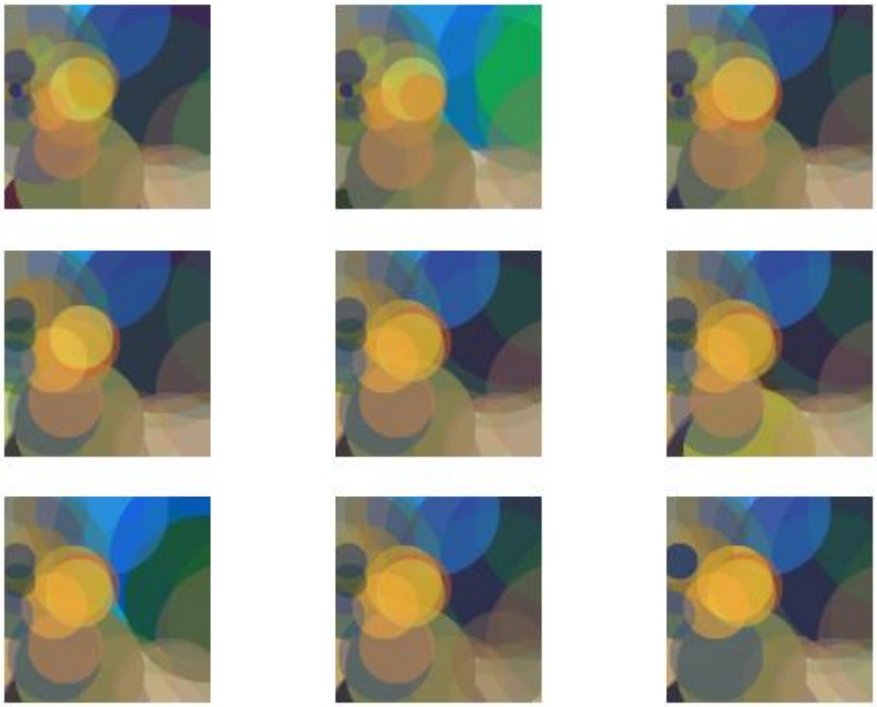
**Figure 9** : Number of Genes is 120

Alper Kutay ÖZBEK
2375467

num_genes/15/Iterations: 2000 - 10000

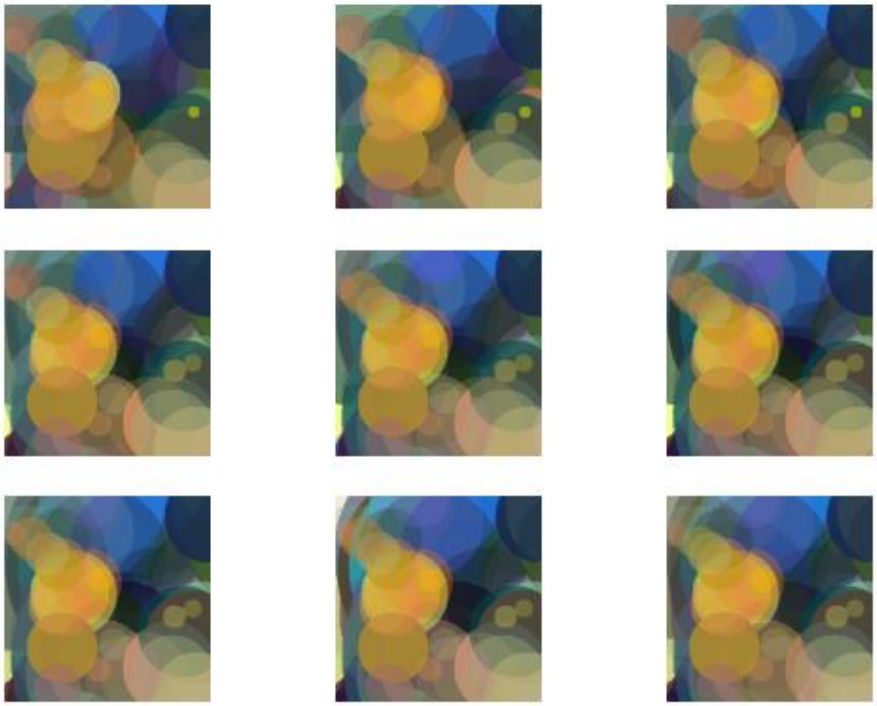


num_genes/30/Iterations: 2000 - 10000

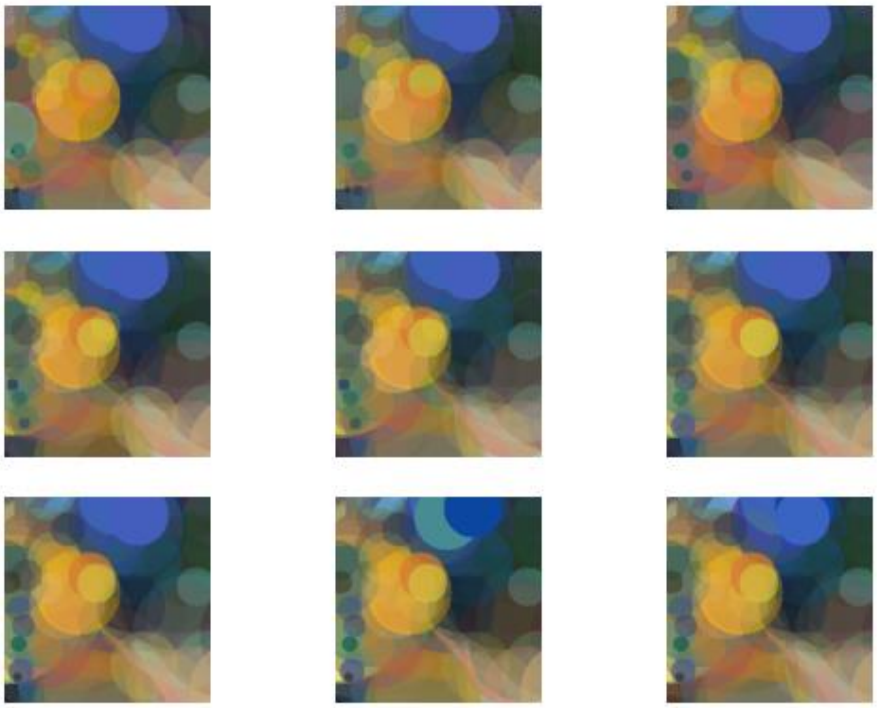


Alper Kutay ÖZBEK
2375467

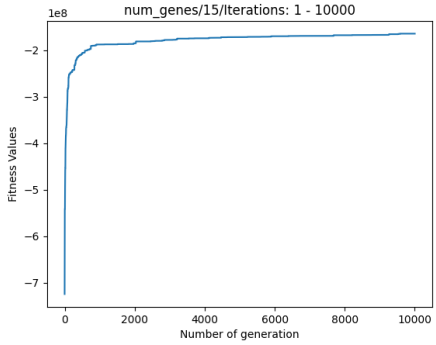
num_genes/80/Iterations: 2000 - 10000

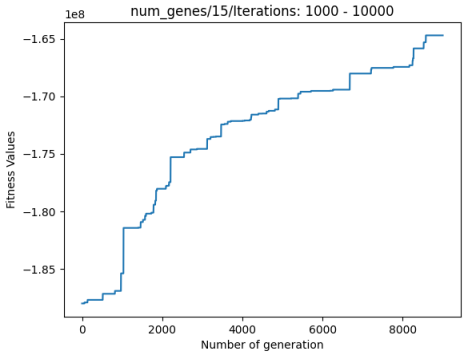


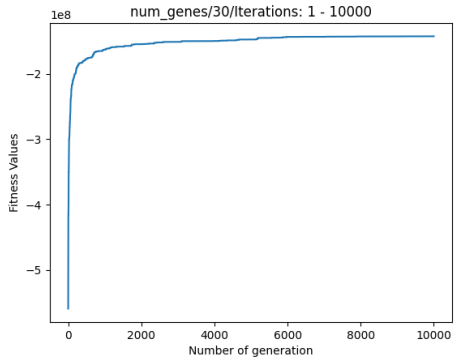
num_genes/120/Iterations: 2000 - 10000

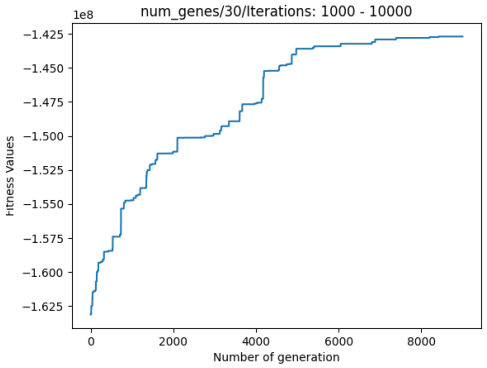


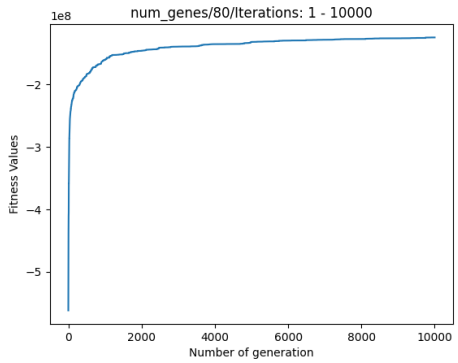
Alper Kutay ÖZBEK
2375467

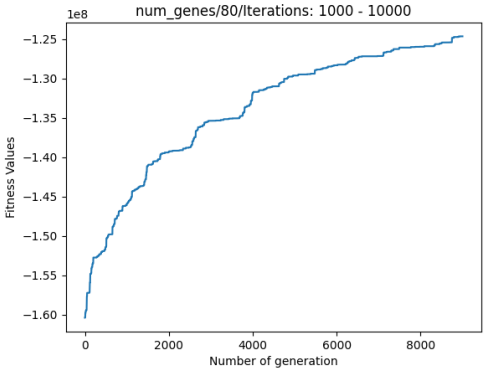


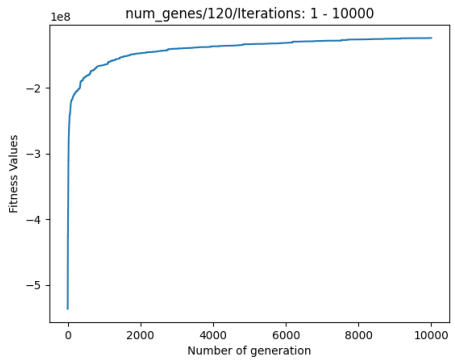



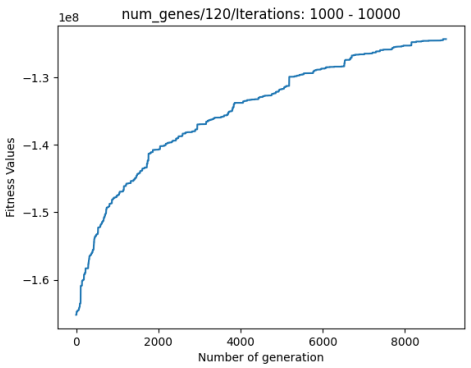



Alper Kutay ÖZBEK
2375467

1.3. Tournament Size

Number of generations = 1000

**Figure 10**: Tournament Size is 2

**Figure 11** : Tournament Size is 8

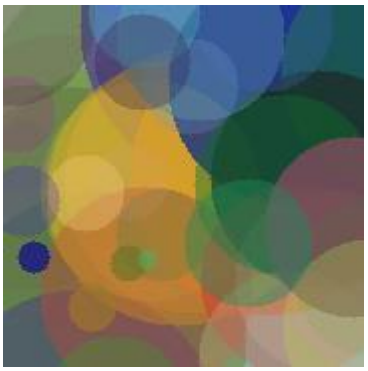
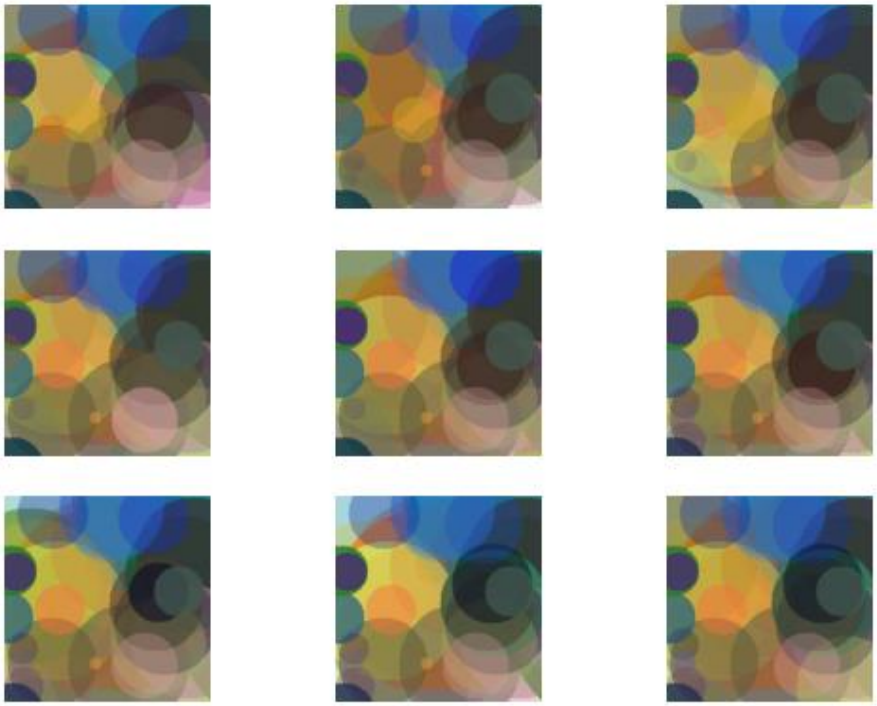


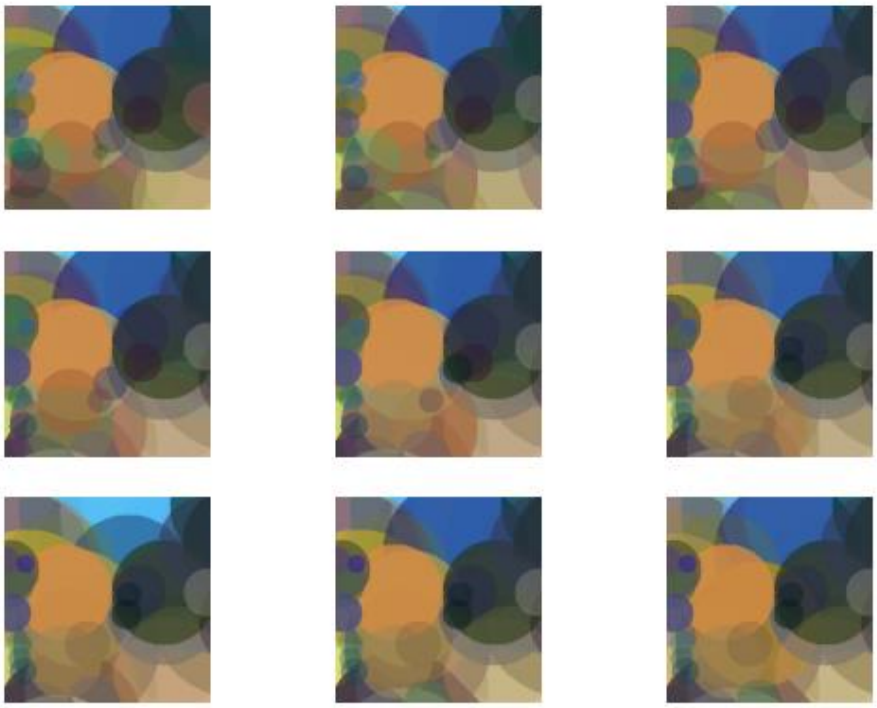
**Figure 12** : Number of Genes is 16

Alper Kutay ÖZBEK
2375467

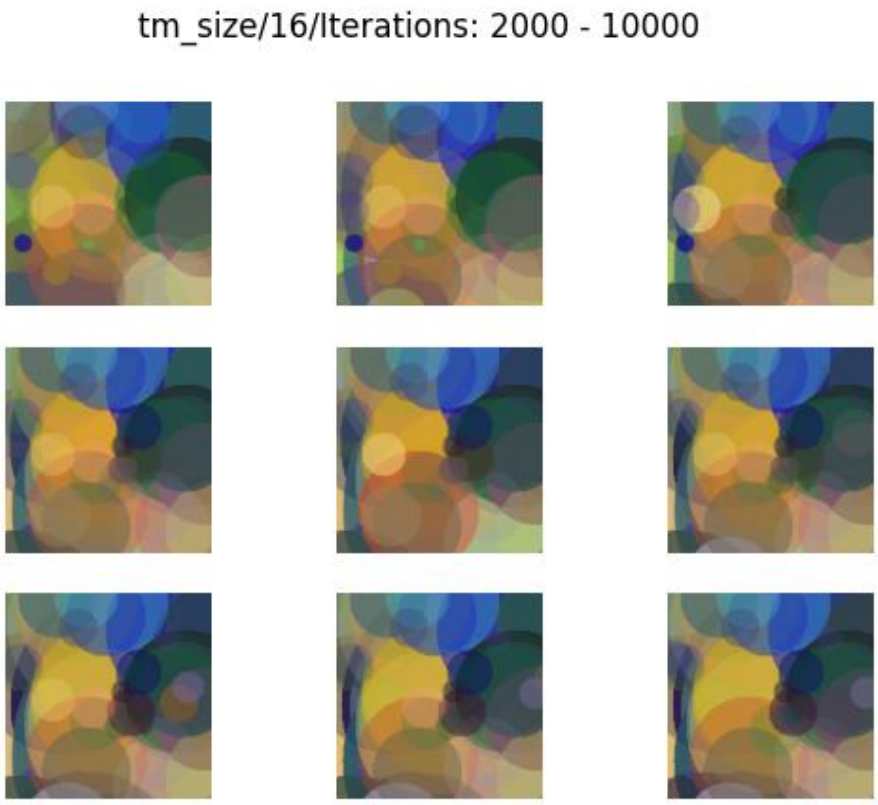
tm_size/2/Iterations: 2000 - 10000

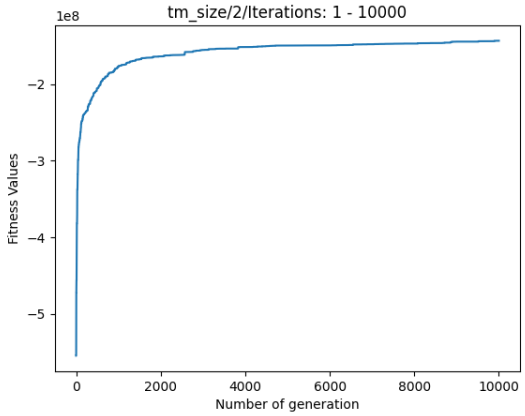


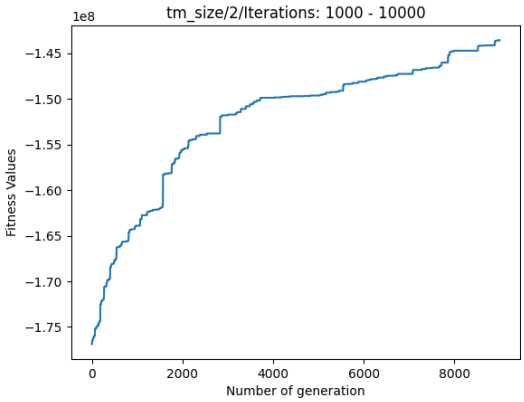
tm_size/8/Iterations: 2000 - 10000

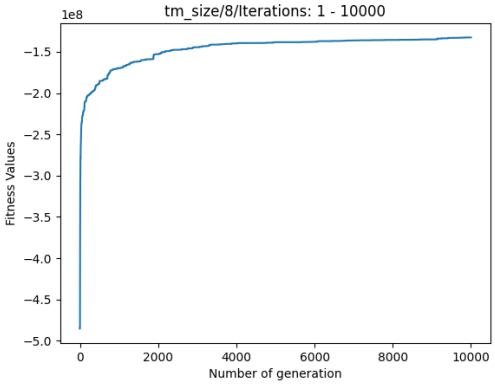


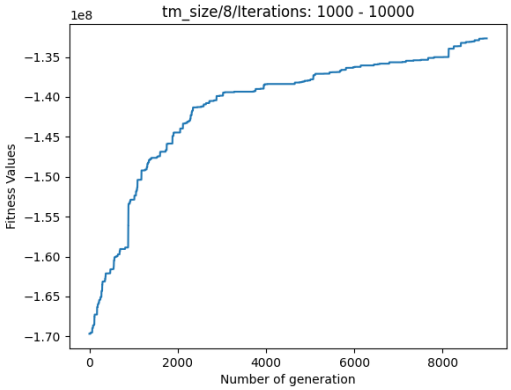
Alper Kutay ÖZBEK
2375467



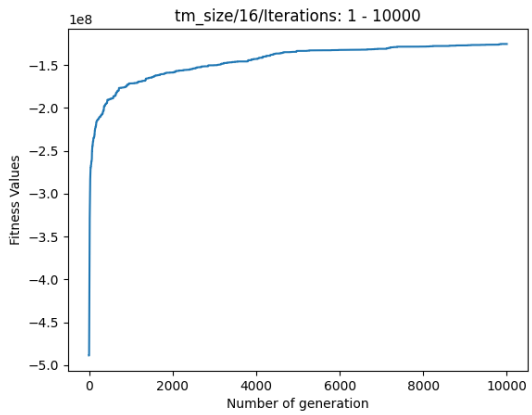



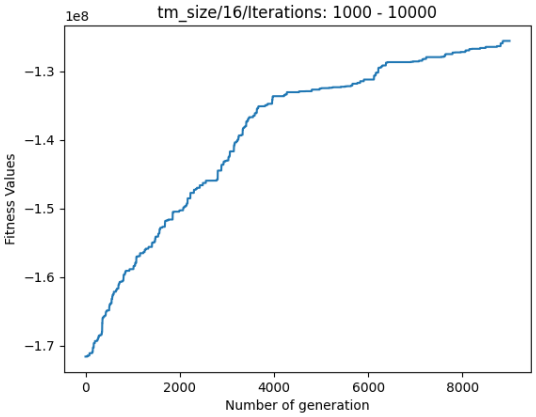



Alper Kutay ÖZBEK
2375467




1.4. Number of Elites

Number of generations = 1000

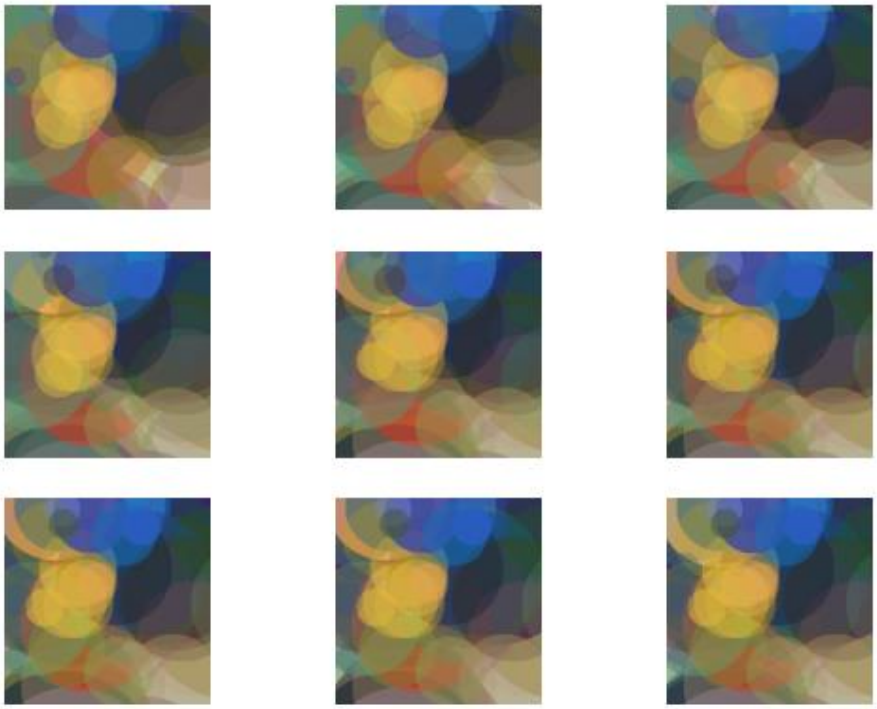
**Figure 13** : Number of Elits is 0.04

**Figure 14** : Number of Elits is 0.35

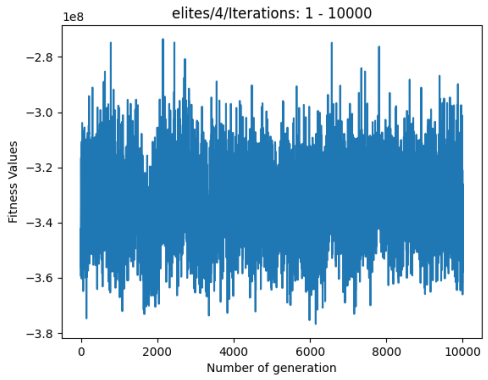
Alper Kutay ÖZBEK
2375467

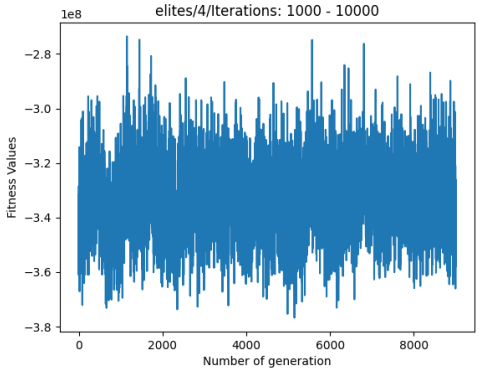
elites/4/Iterations: 2000 - 10000

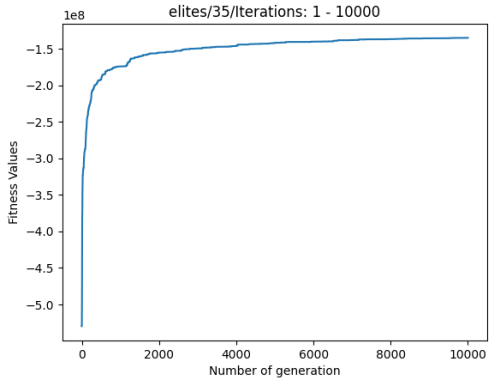
elites/35/Iterations: 2000 - 10000

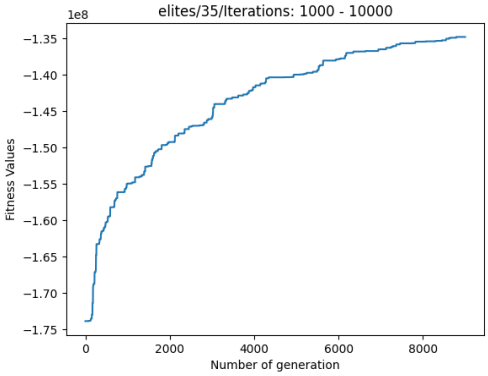


Alper Kutay ÖZBEK
2375467




1.5. Number of Parents

Number of generations = 1000

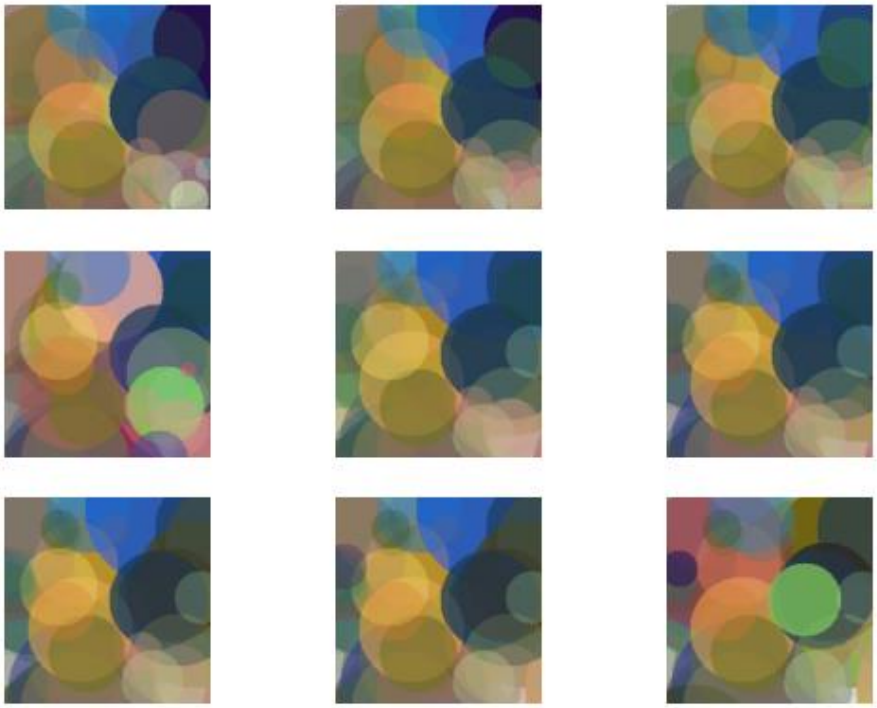
**Figure 15** : Number of Parents is 0.15

**Figure 16** : Number of Parents is 0.30

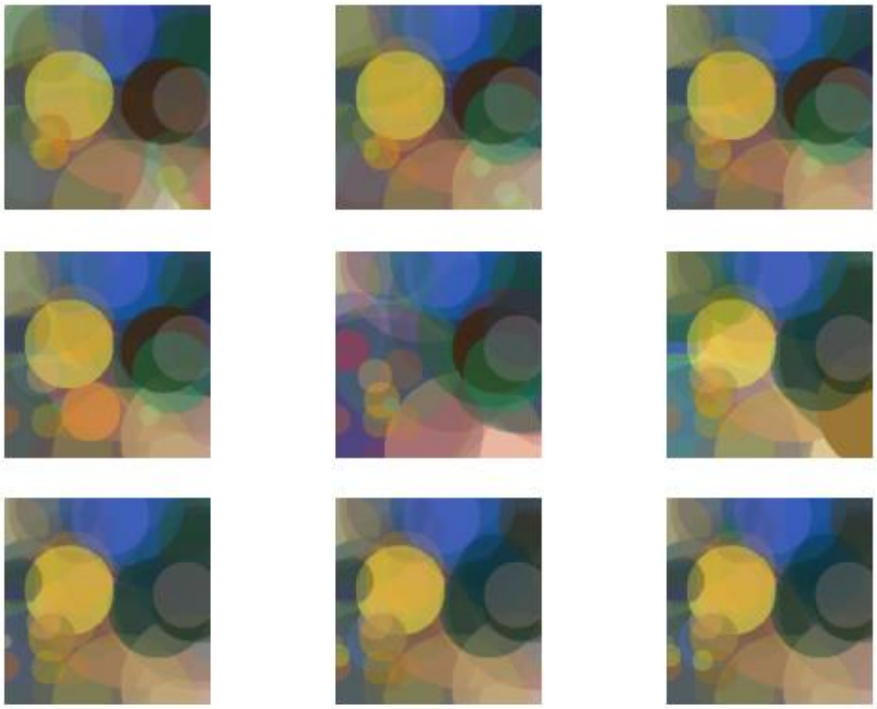
**Figure 17**: Number of Parents is 0.75

Alper Kutay ÖZBEK
2375467

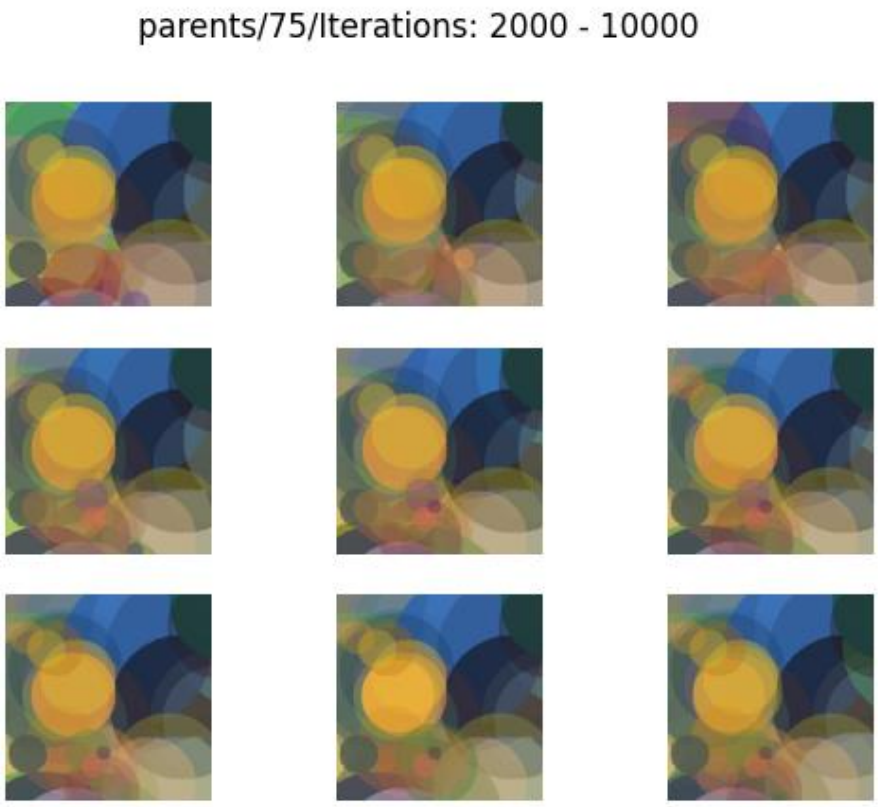
parents/15/Iterations: 2000 - 10000

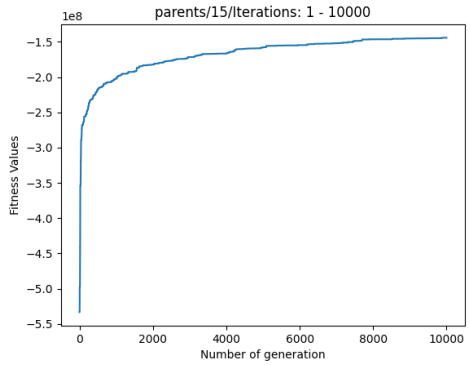


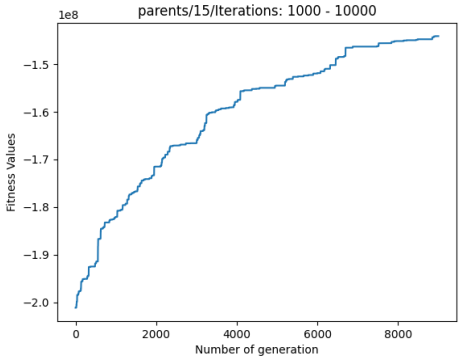
parents/30/Iterations: 2000 - 10000

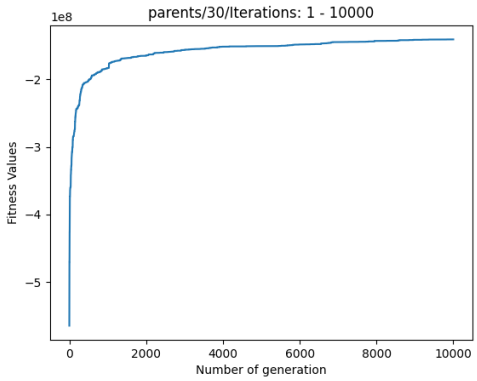


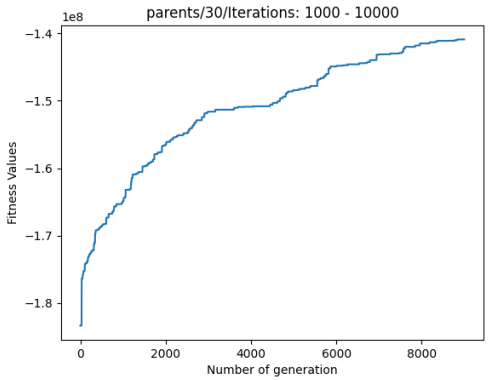
Alper Kutay ÖZBEK
2375467



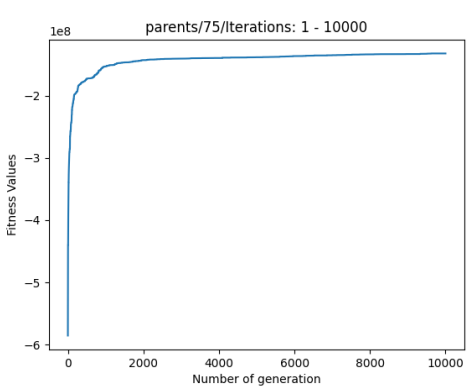



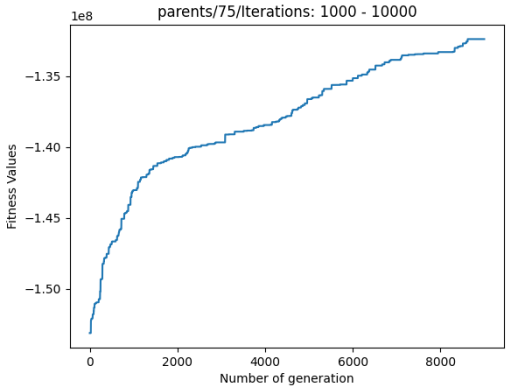



Alper Kutay ÖZBEK
2375467




1.6. Mutation Probability

Number of generations = 1000

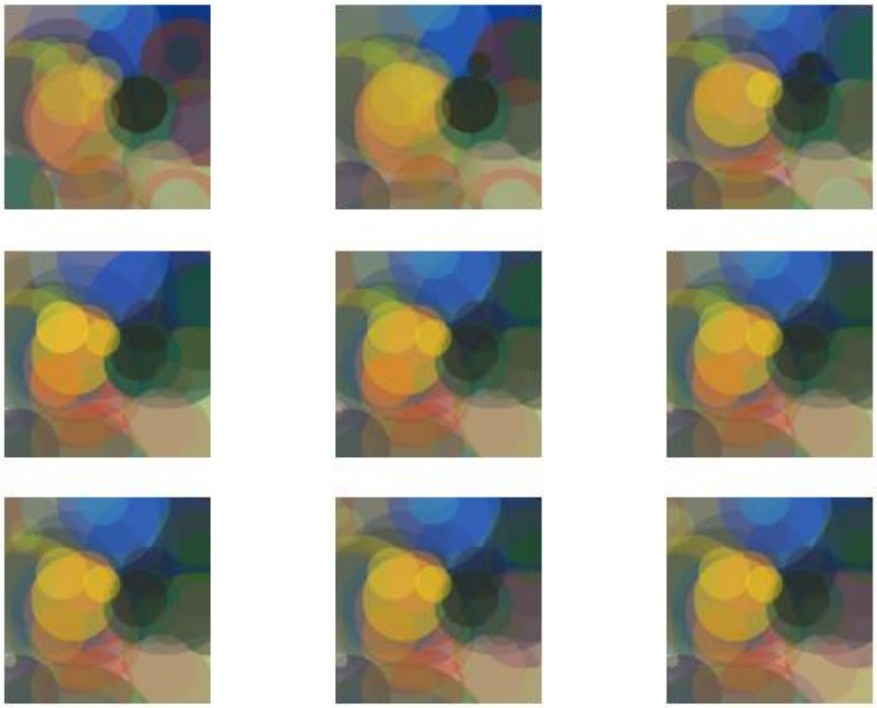
**Figure 18** : Mutation Probability is 0.1

**Figure 19** : Mutation Probability is 0.4

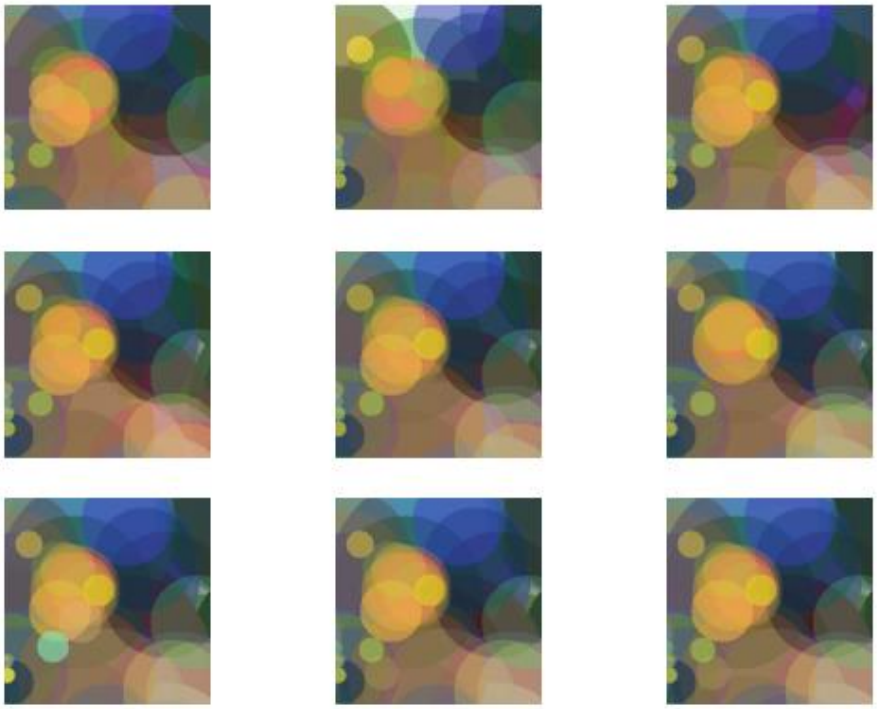
**Figure 20** : Mutation Probability is 0.75

Alper Kutay ÖZBEK
2375467

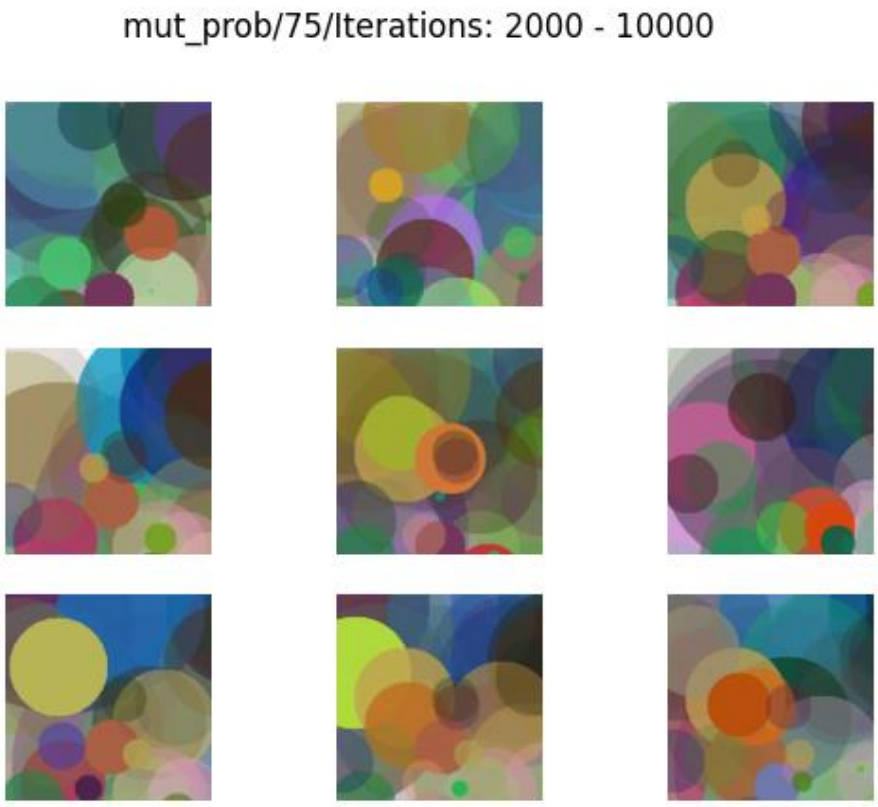
mut_prob/10/Iterations: 2000 - 10000

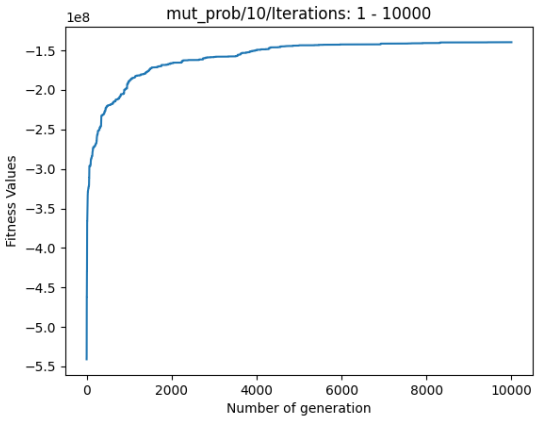


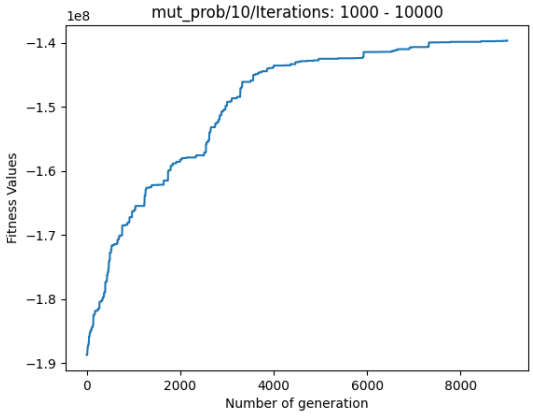
mut_prob/40/Iterations: 2000 - 10000

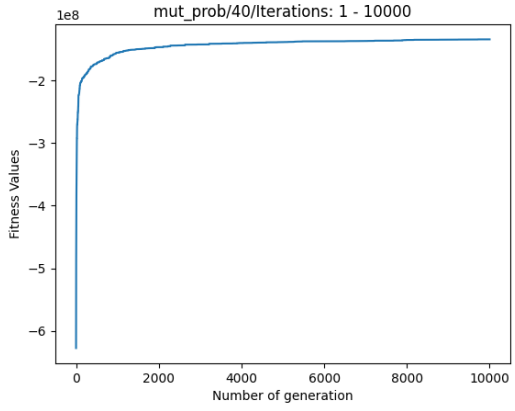


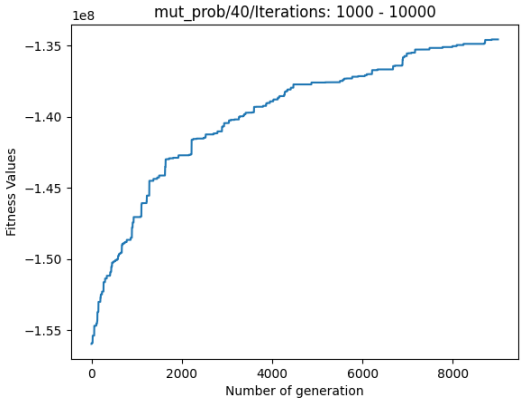
Alper Kutay ÖZBEK
2375467



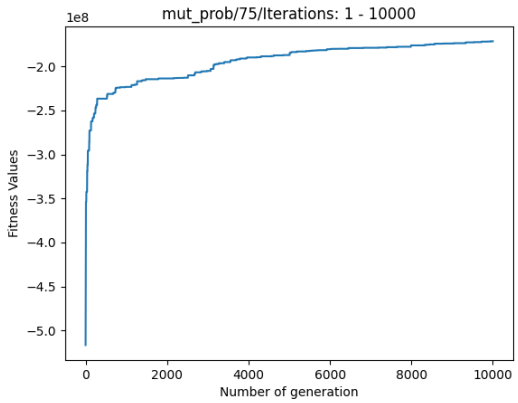



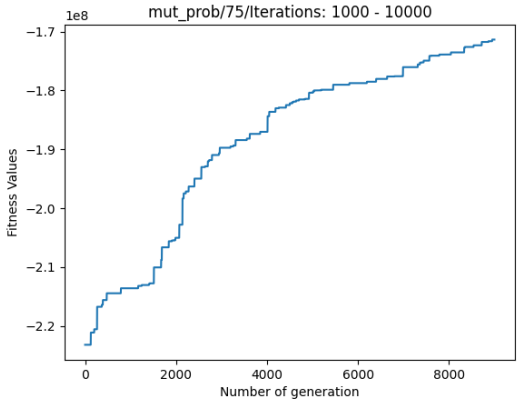



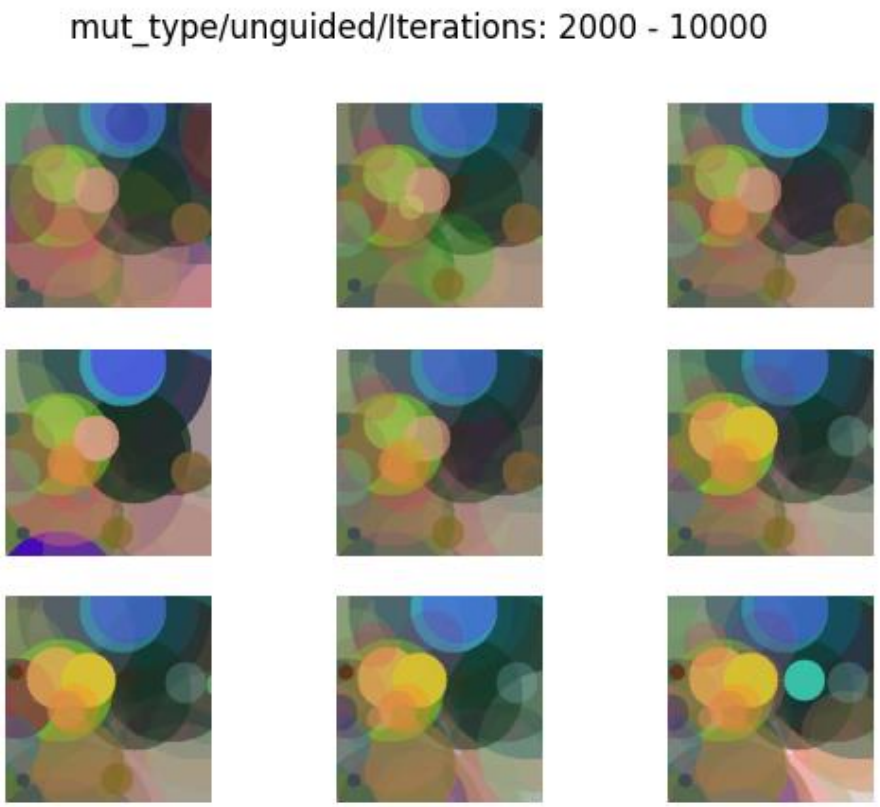
Alper Kutay ÖZBEK
2375467



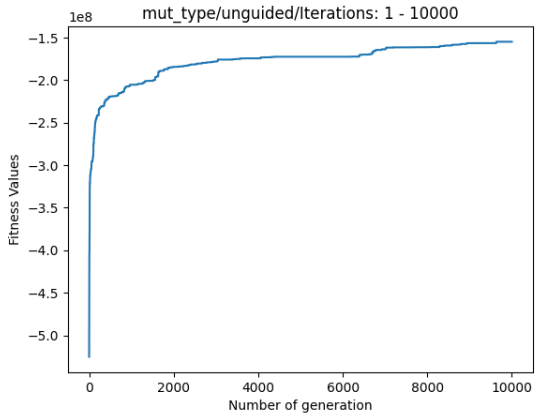



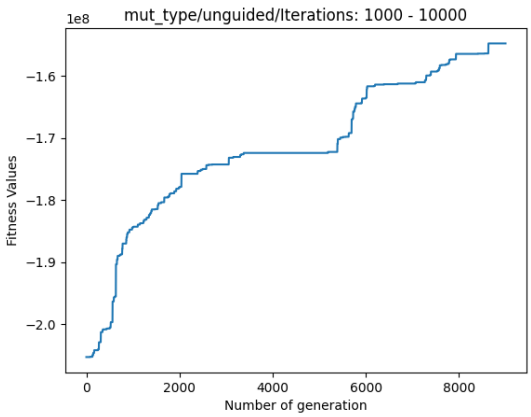

## 1.7. Mutation type

**Figure 21** : Mutation Type is unguided



Alper Kutay ÖZBEK
2375467



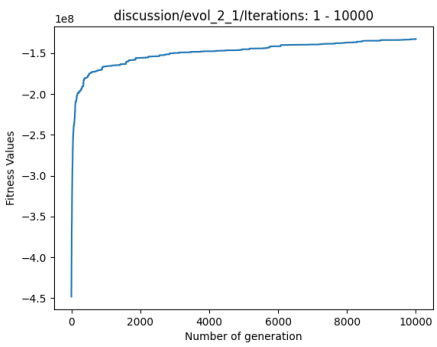



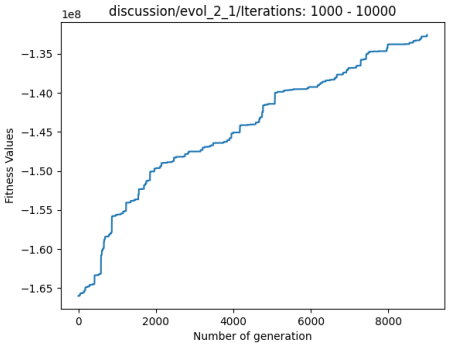

## 2. Discussions

### 2.1. Suggestion 1

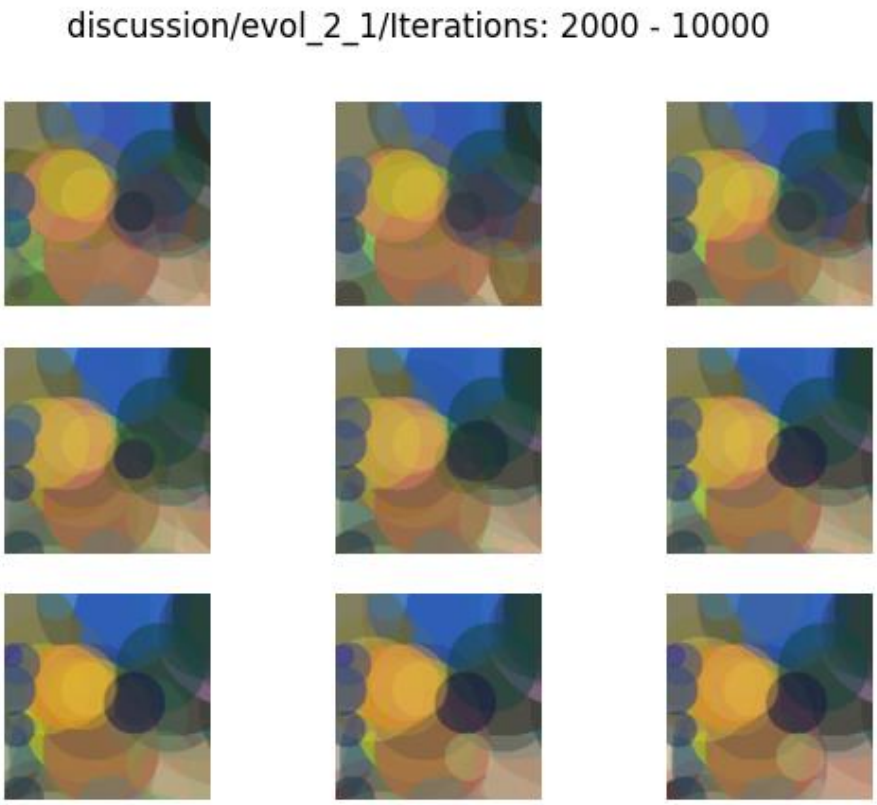
In this suggestion, I propose selecting the two best members from each family: a father, a mother, and two children. We ensure continuous progress and improvement with each iteration by choosing the most promising individuals from each family during the crossover.

Figure 22: 1000th generation of Suggestion 1



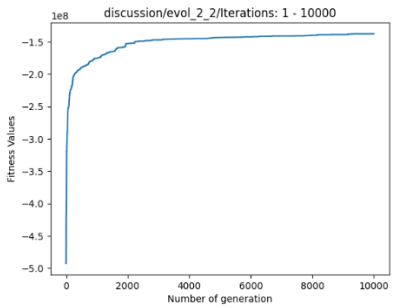
Alper Kutay ÖZBEK
2375467

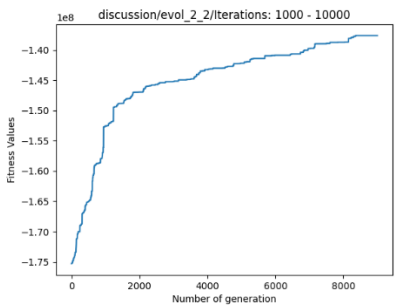



## 2.2. Suggestion 2

In this suggestion, we propose removing the last two individuals from the population during selection. These individuals have the lowest chances of improving themselves to become elites. By replacing them with new individuals, we provide fresh opportunities for progress and advancement within the population.

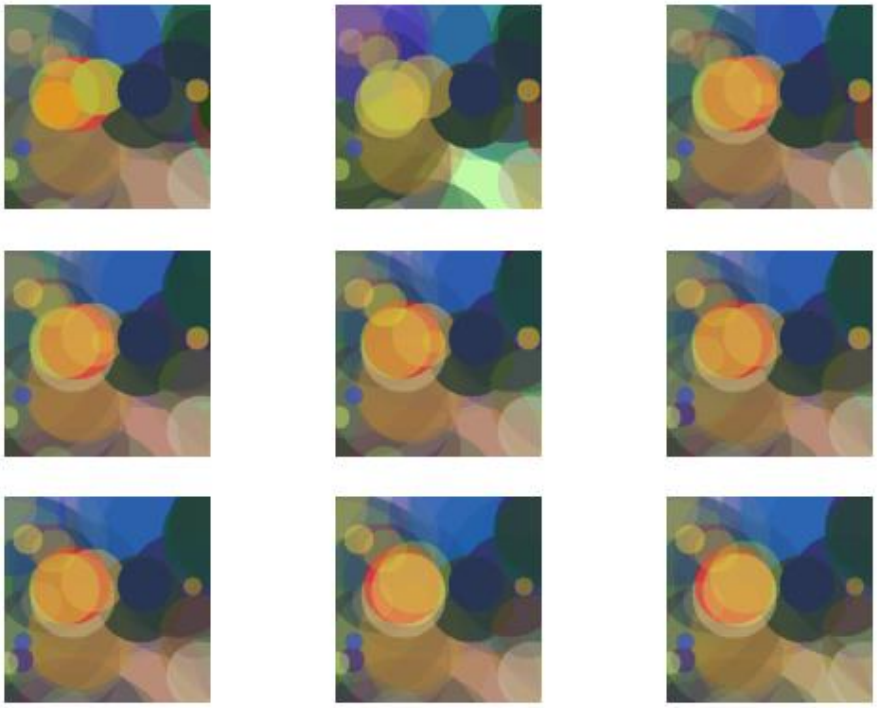
Figure 23: 1000th generation of Suggestion 2



Alper Kutay ÖZBEK
2375467

discussion/evol_2_2/Iterations: 2000 - 10000



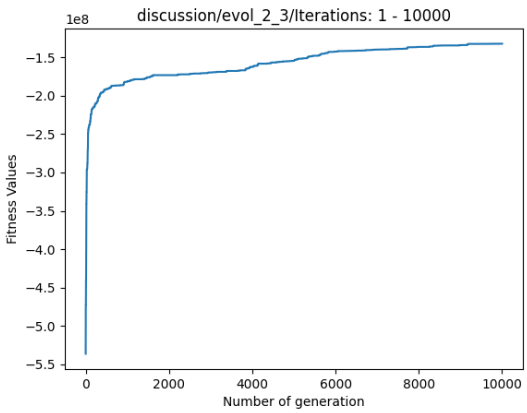
2.3. Suggestion 3

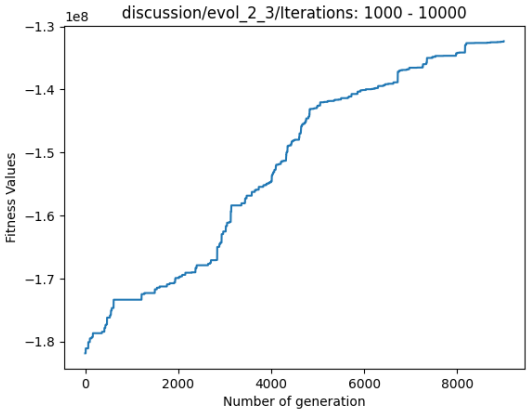
We propose adjusting mutation types and probabilities based on generations in this suggestion. Initially, for the first 5000 generations, we increase the mutation probability to prioritize the rapid improvement of individuals. However, after 5000 generations, we shift to a guided mutation type to ensure the preservation of successful individuals while still introducing variability.

```
def adjust_mut_parameters(self,iteration):
    if iteration < 1000:
        self.mutation_type = 'unguided'
        self.mutation_prob = 0.8
    elif  iteration >5000:
        self.mutation_type = 'guided'
        self.mutation_prob = 0.5
```

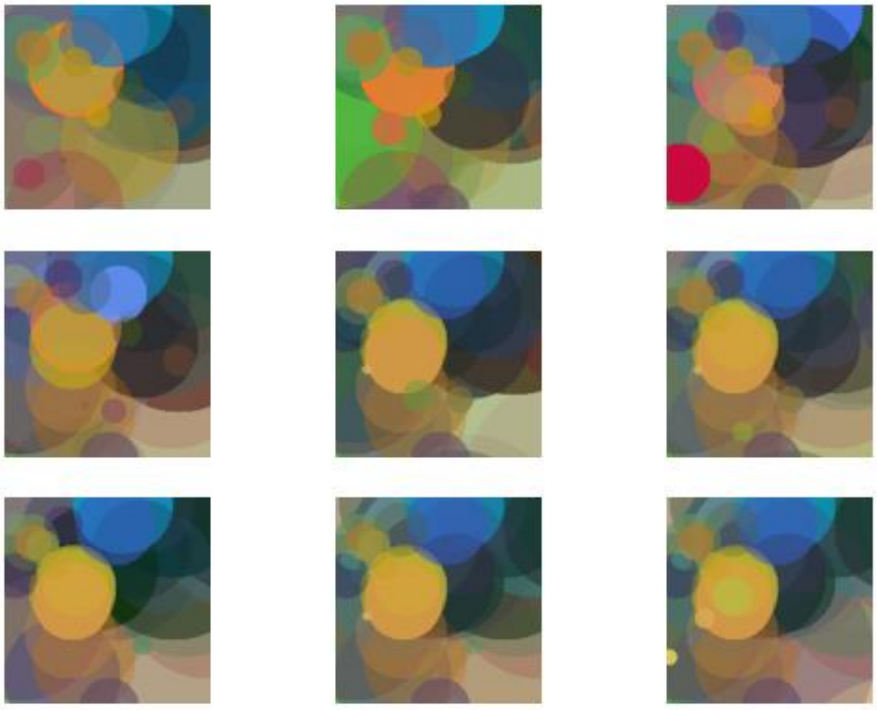
Alper Kutay ÖZBEK
2375467

Figure 24 : 1000th generation of Suggestion 3




discussion/evol_2_3/Iterations: 2000 - 10000



Alper Kutay ÖZBEK
2375467

HW2 code

```
import cv2
import numpy as np
import random as rnd
import json
import matplotlib.pyplot as plt
from copy import deepcopy
source_image = cv2.imread("painting.png")

WIDTH, HEIGHT = source_image.shape[0], source_image.shape[1]



num_inds = [5, 10, 20, 40, 60]
num_genes = [15, 30, 50, 80, 120]
tm_size = [2, 5, 8, 16]
frac_elites = [0.04, 0.2, 0.35]
frac_parents = [0.15, 0.3, 0.6, 0.75]
mutation_prob = [0.1, 0.2, 0.4, 0.75]
mutation_type = ["guided", "unguided"]
num_generation = 10000
dictionary = {"fitness" : None,}


class HyperParameters:
  def __init__(self, num_inds, num_genes, tm_size, num_elites, num_parents, mutation_prob,
mutation_type):#collect parameters in a class
    self.num_inds = num_inds
    self.num_genes = num_genes
    self.tm_size = tm_size
    self.frac_elites = num_elites
    self.frac_parents = num_parents
    self.mutation_prob = mutation_prob
    self.mutation_type = mutation_type

class Gene:
    def __init__(self, index=-1, x=0, y=0, rad=1, R=0, G=0, B=0, A=0):
        self.index = index
        self.x = x
        self.y = y
        self.rad = rad
        self.R = R
        self.G = G
        self.B = B
        self.A = A

    def create_gene(self, index):#Create genes
        self.R = rnd.randrange(256)
        self.G = rnd.randrange(256)
```

Alper Kutay ÖZBEK
2375467

```
        self.B = rnd.randrange(256)
        self.index = index
        self.A = rnd.random()
        temp_x = rnd.randrange(int(1.5 * WIDTH))
        temp_y = rnd.randrange(int(1.5 * HEIGHT))
        temp_rad = rnd.randrange(int(max(WIDTH, HEIGHT) / 2))
        while not self.isIntersects(temp_x, temp_y, temp_rad):
            temp_x = rnd.randrange(int(1.5 * WIDTH))
            temp_y = rnd.randrange(int(1.5 * HEIGHT))
            temp_rad = rnd.randrange(int(max(WIDTH, HEIGHT) / 2))
        self.x = temp_x
        self.y = temp_y
        self.rad = temp_rad

    def isIntersects(self, x, y, r):#it checks whether genes are valid or not
        dist_x = abs(x - WIDTH / 2)
        dist_y = abs(y - HEIGHT / 2)

        if dist_x > (WIDTH / 2 + r): return False
        if dist_y > (HEIGHT / 2 + r): return False

        if dist_x <= (WIDTH / 2): return True
        if dist_y <= (HEIGHT / 2): return True

        cornerDistance_sq = (dist_x - WIDTH / 2) ** 2 + (dist_y - HEIGHT / 2) ** 2
        return cornerDistance_sq <= (r ** 2)

    def guided_Mutation(self):#it provides guided_mutation according to information given
in manual
        #it actually does same things with create_gene() but there is some limitations
which is written
        temp_x = rnd.randrange(max(0, int(self.x-WIDTH/4)), int(self.x+WIDTH/4)+1)
        temp_y = rnd.randrange(max(0, int(self.y-HEIGHT/4)), int(self.y+HEIGHT/4)+1)
        temp_rad = rnd.randrange(max(0, self.rad-10), self.rad+11)
        while not self.isIntersects(temp_x, temp_y, temp_rad):
            temp_x = rnd.randrange(max(0, int(self.x-WIDTH/4)), int(self.x+WIDTH/4)+1)
            temp_y = rnd.randrange(max(0, int(self.y-HEIGHT/4)), int(self.y+HEIGHT/4)+1)
            temp_rad = rnd.randrange(max(0, self.rad-10), self.rad+11)
        self.x = temp_x
        self.y = temp_y
        self.rad = temp_rad
        self.R =  rnd.randrange(max(0, self.R-64), min(self.R+65, 255))
        self.G =  rnd.randrange(max(0, self.G-64), min(self.G+65, 255))
        self.B =  rnd.randrange(max(0, self.B-64), min(self.B+65, 255))
        rnd_a = rnd.random()/2.0 - 0.25
        self.A =  max(0, min(1.0, rnd_a + self.A))

    def printGene(self):
```

Alper Kutay ÖZBEK
2375467

```
        print(f"Gene - {self.index}: x:{self.x}, y:{self.y}, RAD:{self.rad}, R:{self.R},
G:{self.G}, B:{self.B},A:{self.A}")

class Individual:
    def __init__(self,index = -1 ,fitness = 0,chro = []):
        self.index = index
        self.fitness = fitness
        self.chro = chro

    def create_Ind(self,index,num_of_genes):
        self.chro = []
        self.index = index
        for i in range(num_of_genes):#create_genes and append to indvs
            gene = Gene()
            gene.create_gene(i+1)
            self.chro.append(gene)

    def created_image(self):

        self.chro = sorted(self.chro, key=lambda ind: ind.rad, reverse=True)
        white_image = np.zeros((HEIGHT,WIDTH,3),dtype=np.int8 )#firstly create white image
and then make circle of each gene
        cv2.rectangle(white_image, (0, 0), (WIDTH, HEIGHT), (255, 255, 255), -1)
        for gene in self.chro:
            overlay = white_image.copy()
            cv2.circle(overlay,(gene.x,gene.y),gene.rad,(gene.B,gene.G,gene.R),-1)#make
circles on overlayb of white image
            white_image = cv2.addWeighted(overlay, gene.A, white_image, 1 - gene.A,
0)#after that circles are added to white image
        return white_image

    def calculate_fitness(self):
        #Firstly we should make order the chromosome
        white_image = self.created_image()#firstly create white image with circle on it
        diff = source_image.astype(float) - white_image.astype(float)#then calculate
fitness with source_image and white_image
        diff_squared = np.square(diff)
        f = -np.sum(diff_squared)
        self.fitness = f
        return f

    def printIndividual(self):
        #print("Individual -",self.index)
        print("Fitness: ", self.fitness)
        #print("Chrosome:")
        #for gene in self.chro:
        #    gene.printGene()

    def Mutation(self,mut_type,mut_prob):
```

Alper Kutay ÖZBEK
2375467

```
        index_of_mut_genes = rnd.randrange(len(self.chro))#according to probability
        if mut_type == "unguided":#if it is unguieded
            self.chro[index_of_mut_genes].create_gene(index_of_mut_genes)#then create new
gene without any restiriction
        else:
            self.chro[index_of_mut_genes].guided_Mutation()#if it is guided then run guided
muation
        mutation_points = np.random.randint(100,size=len(self.chro))
        mutation_points[index_of_mut_genes]=100
        for i in range (len(self.chro)):
            if mutation_points[i] < mut_prob*100:
                if mut_type == "unguided":#if it is unguieded
                    self.chro[i].create_gene(i+1)#then create new gene without any
restiriction
                else:
                    self.chro[i].guided_Mutation()#if it is guided then run guided muation

    def Indv_cross(self,Indv2):
        crossover_points = np.random.randint(2,size=len(self.chro))#create an array that
takes random values which are 0,1. This array decides crossover genes
        chro1 = []#crossover genes
        chro2 = []
        for i in range(len(crossover_points)):
            if crossover_points[i] == 0:#if crossover_points is 0 then chro1 takes gene
from indv1
                chro1.append(self.chro[i])
                chro2.append(Indv2.chro[i])
            else:#if crossover_points is 0 then chro1 takes gene from indv2 and vice versa
                chro1.append(Indv2.chro[i])
                chro2.append(self.chro[i])
        child1 = Individual(index=self.index,chro=chro1)#then create indvs with crossovered
chromosomes
        child2 = Individual(index=Indv2.index,chro=chro2)
        return child1,child2

    def crossover_suggestion(self,parents):#suggestion 1 : choose best two from family
        children = []#save each childer in an array
        for i in range(0,len(parents),2):#run crossover between each double parents
            crossover_points =
np.random.randint(2,size=self.hyp_parameters.num_genes)#create an array that takes random
values which are 0,1. This array decides crossover genes
            chro1 = []#crossover genes
            chro2 = []
            family = []
            family.append(parents[i])
            family.append(parents[i+1])
            for j in range(0,len(crossover_points)-1):
                if crossover_points[j] == 0:#if crossover_points is 0 then chro1 takes gene
from indv1
```

Alper Kutay ÖZBEK
2375467

```
                    chro1.append(deepcopy(parents[i].chro[j]))
                    chro2.append(deepcopy(parents[i+1].chro[j]))
                else:#if crossover_points is 0 then chro1 takes gene from indv2 and vice
versa
                    chro1.append(deepcopy(parents[i+1].chro[j]))
                    chro2.append(deepcopy(parents[i].chro[j]))
            child1 = Individual(index=parents[i].index,chro=chro1)#then create indvs with
crossovered chromosomes
            child2 = Individual(index=parents[i+1].index,chro=chro2)
            child1.calculate_fitness()
            child2.calculate_fitness()
            family.append(child1)
            family.append(child2)
            family.sort(key=lambda item: item.fitness, reverse=True)#sort the family
            children.append(family[0])#and choose the best two from four
            children.append(family[1])
        children = self.sortIndividuals(children) #we created children but do not forget
all children have index=1
        return children

class Population:
    def __init__(self,hyp_parameters,name,num_of_generations):
        self.hyp_parameters = hyp_parameters
        self.indvs = []
        self.name = name#I assign a name variable because it facilitates to giving name to
png files
        self.num_of_generations = num_of_generations
    def create_population(self):#create population according to given parameters
        self.indvs = []
        #self.best_indvs.create_Ind()
        for i in range(self.hyp_parameters.num_inds):
            indv = Individual()
            indv.create_Ind(i+1,self.hyp_parameters.num_genes)
            self.indvs.append(indv)

    # sort the population according to fitness value of indv
    def sortIndividuals(self,indvs):
        return sorted(indvs, key=lambda item: item.fitness, reverse=True)

    def printPopulation(self):
      for ind in self.indvs:
        ind.printIndividual()

    def eval_population(self):
        for indv in self.indvs:#calculate each one of the indvs in population
            indv.calculate_fitness()

    def selection(self):
        num_of_elits = int(self.hyp_parameters.frac_elites * self.hyp_parameters.num_inds)
```

Alper Kutay ÖZBEK
2375467

```
        num_of_parents = int( self.hyp_parameters.frac_parents *
self.hyp_parameters.num_inds)
        #we need even number of parents
        if num_of_parents % 2 == 1:
            num_of_parents = num_of_parents + 1
        #sort the Indviduals in the populations
        self.indvs = self.sortIndividuals(self.indvs)
        #now we can select the elits

        elits = deepcopy(self.indvs[:num_of_elits])
        others = deepcopy(self.indvs[num_of_elits:])
        parents = []
        #tournament between others
        for i in range(num_of_parents):
            best_one = rnd.randrange(len(others))
            for j in range(self.hyp_parameters.tm_size):
                index = rnd.randrange(len(others))
                if others[index].fitness > others[best_one].fitness:
                    best_one = index
            parents.append(others.pop(best_one))#best one does not contain in the others
because it is selected for parents now
        return(elits,parents,others)

    def selection_suggestion(self):
        num_of_elits = int(self.hyp_parameters.frac_elites * self.hyp_parameters.num_inds)
        num_of_parents = int( self.hyp_parameters.frac_parents *
self.hyp_parameters.num_inds)
        num_of_others = self.hyp_parameters.num_inds -num_of_elits-num_of_parents
        #we need even number of parents
        if num_of_parents % 2 == 1:
            num_of_parents = num_of_parents + 1
        #sort the Indviduals in the populations
        self.indvs = self.sortIndividuals(self.indvs)
        #now we can select the elits

        elits = deepcopy(self.indvs[:num_of_elits])
        others = deepcopy(self.indvs[num_of_elits:])
        parents = []
        #tournament between others
        for i in range(num_of_parents):
            best_one = rnd.randrange(len(others))
            for j in range(self.hyp_parameters.tm_size):
                index = rnd.randrange(len(others))
                if others[index].fitness > others[best_one].fitness:
                    best_one = index
            parents.append(others.pop(best_one))#best one does not contain in the others
because it is selected for parents now
        new = Individual()
        new.create_Ind(others[-1].index,self.hyp_parameters.num_genes)
```

Alper Kutay ÖZBEK
2375467

```
        new2 = Individual()
        new2.create_Ind(others[-2].index,self.hyp_parameters.num_genes)
        others.pop(len(others)-1)
        others.pop(len(others)-1)
        new.calculate_fitness()
        new2.calculate_fitness()
        others.append(new)
        others.append(new2)
        return(elits,parents,others)

    def crossover(self,parents):
        children = []#save each childer in an array
        for i in range(0,len(parents),2):#run crossover between each double parents
            child1, child2 = parents[i].Indv_cross(parents[i+1])
            children.append(child1)
            children.append(child2)
        #children = self.sortIndividuals(children) #we created children but do not forget
all children have index=1
        return children

    def crossover_sugg(self,parents):
        children = []#save each childer in an array
        for i in range(0,len(parents),2):#run crossover between each double parents
            child1, child2 = parents[i].crossover_suggestion(parents[i+1])
            children.append(child1)
            children.append(child2)
        #children = self.sortIndividuals(children) #we created children but do not forget
all children have index=1
        return children

    def pop_mutation(self,population):
        for indv in population:
            if rnd.random() < self.hyp_parameters.mutation_prob:

indv.Mutation(self.hyp_parameters.mutation_type,self.hyp_parameters.mutation_prob)#mutate
each indivudual
        self.eval_population()

    def evolution(self):#we are going to make evolution to our population by specific
number of generation
        self.create_population()#create population
        fitness_values = []#record all fitness values and at the end of the funciton save
it by using json
        for i in range(self.num_of_generations):
            self.eval_population()#after creation of population evaluate the population
which means we calculate the fitness values of each member in the beginning
            elits,parents,others = self.selection()#then do selection and get elits,parents
and others
            children = self.crossover_sugg(parents=parents)#get children from parents
```

Alper Kutay ÖZBEK
2375467

```
            self.pop_mutation(deepcopy(others+children))#skip the elits and make mutation
in rest of population
            self.indvs = deepcopy(others+children+elits)#now our popultaion consists of
 others+children+elits
            fitness_values.append(elits[0].fitness)#lastly sort the individuals and save
the best one and do it every generations
            if i % 10 == 9:
                print(f"Loading %{(i+1)/100},{fitness_values[i]}")
                #print(f"Children:{len(children)} , elits {len(elits)}, others
{len(others)} , parents {len(parents)}")
            if i % 1000 == 999:#pring png in each 1000 generations
                name = self.name+'iteration_'+str(i+1)+'.png'
                print("Name : "+name)
                cv2.imwrite(name,self.indvs[0].created_image())
        dictonary ={
            "fitness" : fitness_values
        }
        with open(self.name+"fitnes_values.json","w") as outfile:
            json.dump(dictonary,outfile)
        del fitness_values

    def evolution2_2(self):
        self.create_population()#create population
        fitness_values = []#record all fitness values and at the end of the funciton save
it by using json
        for i in range(self.num_of_generations):
            self.eval_population()#after creation of population evaluate the population
which means we calculate the fitness values of each member in the beginning
            elits,parents,others = self.selection_suggestion()#then do selection and get
elits,parents and others
            children = self.crossover(parents=parents)#get children from parents
            self.pop_mutation(deepcopy(others+children))#skip the elits and make mutation
in rest of population
            self.indvs = deepcopy(others+children+elits)#now our popultaion consists of
 others+children+elits
            fitness_values.append(elits[0].fitness)#lastly sort the individuals and save
the best one and do it every generations
            if i % 10 == 9:
                print(f"Loading %{(i+1)/100},{fitness_values[i]}")
            if i % 1000 == 999:#pring png in each 1000 generations
                name = self.name+'iteration_'+str(i+1)+'.png'
                print("Name : "+name)
                cv2.imwrite(name,self.indvs[0].created_image())
        dictonary ={
            "fitness" : fitness_values
        }
        with open(self.name+"fitnes_values.json","w") as outfile:
            json.dump(dictonary,outfile)
        del fitness_values
```

Alper Kutay ÖZBEK
2375467

```
    def evolution2_2(self):
        self.create_population()#create population
        fitness_values = []#record all fitness values and at the end of the funciton save
it by using json
        for i in range(self.num_of_generations):
            self.eval_population()#after creation of population evaluate the population
which means we calculate the fitness values of each member in the beginning
            elits,parents,others = self.selection_suggestion()#then do selection and get
elits,parents and others
            children = self.crossover(parents=parents)#get children from parents
            self.pop_mutation(deepcopy(others+children))#skip the elits and make mutation
in rest of population
            self.indvs = deepcopy(others+children+elits)#now our popultaion consists of
 others+children+elits
            fitness_values.append(elits[0].fitness)#lastly sort the individuals and save
the best one and do it every generations
            if i % 10 == 9:
                print(f"Loading %{(i+1)/100},{fitness_values[i]}")
            if i % 1000 == 999:#pring png in each 1000 generations
                name = self.name+'iteration_'+str(i+1)+'.png'
                print("Name : "+name)
                cv2.imwrite(name,self.indvs[0].created_image())
        dictonary ={
            "fitness" : fitness_values
        }
        with open(self.name+"fitnes_values.json","w") as outfile:
            json.dump(dictonary,outfile)
        del fitness_values

    def evolution2_3(self):#changable mut type and probability
        self.create_population()#create population
        fitness_values = []#record all fitness values and at the end of the funciton save
it by using json
        for i in range(self.num_of_generations):
            self.adjust_mut_parameters(i)
            self.eval_population()#after creation of population evaluate the population
which means we calculate the fitness values of each member in the beginning
            elits,parents,others = self.selection_suggestion()#then do selection and get
elits,parents and others
            children = self.crossover(parents=parents)#get children from parents
            self.pop_mutation(deepcopy(others+children))#skip the elits and make mutation
in rest of population
            self.indvs = deepcopy(others+children+elits)#now our popultaion consists of
 others+children+elits
            fitness_values.append(elits[0].fitness)#lastly sort the individuals and save
the best one and do it every generations
            if i % 10 == 9:
                print(f"Loading %{(i+1)/100},{fitness_values[i]}")
```

Alper Kutay ÖZBEK
2375467

```
            if i % 1000 == 999:#pring png in each 1000 generations
                name = self.name+'iteration_'+str(i+1)+'.png'
                print("Name : "+name)
                cv2.imwrite(name,self.indvs[0].created_image())
        dictonary ={
            "fitness" : fitness_values
        }
        with open(self.name+"fitnes_values.json","w") as outfile:
            json.dump(dictonary,outfile)
        del fitness_values

    def adjust_mut_parameters(self,iteration):
        if iteration < 1000:
            self.mutation_type = 'unguided'
            self.mutation_prob = 0.8
        elif  iteration >5000:
            self.mutation_type = 'guided'
            self.mutation_prob = 0.5

"""
#suggestion 1 : choose best of two from family
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
popu(hyp_parameters=params,name="discussion/evol_2_1/",num_of_generations=num_generation)
pop.evolution2_1()


#suggestion 2 : get rid of the last two members of others(not elits and children) and
create new Indiviuals
params = HyperParameters(num_inds[4], num_genes[4], tm_size[3], frac_elites[2],
frac_parents[3], mutation_prob[3], mutation_type[1])
pop =
Population(hyp_parameters=params,name="discussion/evol_2_2/",num_of_generations=num_generat
ion)
pop.evolution2_2()
#?

#suggestion 3 adjustable mut type and probability
params = HyperParameters(num_inds[4], num_genes[4], tm_size[3], frac_elites[2],
frac_parents[3], mutation_prob[3], mutation_type[1])
pop =
Population(hyp_parameters=params,name="discussion/evol_2_2/",num_of_generations=num_generat
ion)
pop.evolution()
```

Alper Kutay ÖZBEK
2375467

```
#default
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_inds/"+str(num_inds[2])+"/",num_of_generations=n
um_generation)
pop.evolution()

#inds
params = HyperParameters(num_inds[0], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_inds/"+str(num_inds[0])+"/",num_of_generations=n
um_generation)
pop.evolution()

params = HyperParameters(num_inds[1], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_inds/"+str(num_inds[1])+"/",num_of_generations=n
um_generation)
pop.evolution()

params = HyperParameters(num_inds[3], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_inds/"+str(num_inds[3])+"/",num_of_generations=n
um_generation)
pop.evolution()

params = HyperParameters(num_inds[4], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_inds/"+str(num_inds[4])+"/",num_of_generations=n
um_generation)
pop.evolution()

#genes
params = HyperParameters(num_inds[2], num_genes[0], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_genes/"+str(num_genes[0])+"/",num_of_generations
=num_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[1], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
```

Alper Kutay ÖZBEK
2375467

```
pop =
Population(hyp_parameters=params,name="num_genes/"+str(num_genes[1])+"/",num_of_generations
=num_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[3], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_genes/"+str(num_genes[3])+"/",num_of_generations
=num_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[4], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="num_genes/"+str(num_genes[4])+"/",num_of_generations
=num_generation)
pop.evolution()


#tm_size
params = HyperParameters(num_inds[2], num_genes[2], tm_size[0], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="tm_size/"+str(tm_size[0])+"/",num_of_generations=num
_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[2], tm_size[2], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="tm_size/"+str(tm_size[2])+"/",num_of_generations=num
_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[2], tm_size[3], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="tm_size/"+str(tm_size[3])+"/",num_of_generations=num
_generation)
pop.evolution()

#elits
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[0],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="elites/"+str(int(frac_elites[0]*100))+"/",num_of_gen
erations=num_generation)
pop.evolution()
```

Alper Kutay ÖZBEK
2375467

```
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[2],
frac_parents[2], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="elites/"+str(int(frac_elites[2]*100))+"/",num_of_gen
erations=num_generation)
pop.evolution()


#parents
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[0], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="parents/"+str(int(frac_parents[0]*100))+"/",num_of_g
enerations=num_generation)
pop.evolution()

params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[1], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="parents/"+str(int(frac_parents[1]*100))+"/",num_of_g
enerations=num_generation)
pop.evolution()


params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[3], mutation_prob[1], mutation_type[0])
pop =
Population(hyp_parameters=params,name="parents/"+str(int(frac_parents[3]*100))+"/",num_of_g
enerations=num_generation)
pop.evolution()


#mut_prob
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[0], mutation_type[0])
pop =
Population(hyp_parameters=params,name="mut_prob/"+str(int(mutation_prob[0]*100))+"/",num_of
_generations=num_generation)
pop.evolution()



params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[2], mutation_type[0])
pop =
Population(hyp_parameters=params,name="mut_prob/"+str(int(mutation_prob[2]*100))+"/",num_of
_generations=num_generation)
```

Alper Kutay ÖZBEK
2375467

```
pop.evolution()
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], 0.95, mutation_type[0])
pop =
Population(hyp_parameters=params,name="mut_prob/"+str(int(mutation_prob[3]*100))+"/",num_of
_generations=num_generation)
pop.evolution()

#type
params = HyperParameters(num_inds[2], num_genes[2], tm_size[1], frac_elites[1],
frac_parents[2], mutation_prob[1], mutation_type[1])
pop =
Population(hyp_parameters=params,name="mut_type/"+str(mutation_type[1])+"/",num_of_generati
ons=num_generation)
pop.evolution()
"""
```

Alper Kutay ÖZBEK
2375467

Function of Print fitness:

```
import os
import matplotlib.pyplot as plt
import json
import numpy as np
x_list_9000 = []
x_list_10000 = []
for i in range(9000):#
    x_list_9000.append(i)
for i in range(10000):
    x_list_1000.append(i)
def plot_fitness_9000(file_name,folder):
    with open(file_name+'.json', 'r') as f:
        data = json.load(f)
    y = list(data.values())
    plt.figure()
    plt.plot(x_list_9000, y[0][1000:10000])
    # Customize the plot as desired
    plt.xlabel('Number of generation')
    plt.ylabel('Fitness Values')
    plt.title(folder+"/Iterations: 1000 - 10000")
    # Save the plot as a PNG file
    plt.savefig(file_name+'1000-10000.png')
    plt.close()
    f.close()
def plot_fitness_10000(file_name,folder):
    with open(file_name+'.json', 'r') as f:
        data = json.load(f)
    y = list(data.values())
    plt.figure()
    plt.plot(x_list_10000, y[0][0:10000])
    # Customize the plot as desired
    plt.xlabel('Number of generation')
    plt.ylabel('Fitness Values')
    plt.title(folder+"/Iterations: 1 - 10000")
    # Save the plot as a PNG file
    plt.savefig(file_name+'1-10000.png')
    plt.close()
    f.close()
folder_names =
["num_genes/","num_inds/","parents/","tm_size/","mut_type/","mut_prob/","elite
s/","discussion/"]
for i in folder_names:
    folders = [f for f in os.listdir(i) if os.path.isdir(os.path.join(i, f))]
    for folder in folders:
plot_fitness_10000(str(i)+str(folder)+"/fitnes_values",str(i)+str(folder))
plot_fitness_9000(str(i)+str(folder)+"/fitnes_values",str(i)+str(folder))
        #enter each folder and find all folders in each defined folder
```

Alper Kutay ÖZBEK
2375467

Print nine images in one image:

```
import os
import matplotlib.pyplot as plt
import numpy as np
import cv2

def plot_images(file_names,folder_name):
  # Generate some random images for demonstration
  images = [cv2.cvtColor(cv2.imread(file_name), cv2.COLOR_BGR2RGB) for
file_name in file_names]
  # Create a 3x3 grid of subplots
  fig, axs = plt.subplots(3, 3)

  # Iterate over the axes and images, and plot each image on a subplot
  for ax, img in zip(axs.flat, images):
      ax.imshow(img)
      ax.axis('off')

  # Set the title of the figure
  fig.suptitle(folder_name+"/Iterations: 2000 - 10000 ")

  # Show the plot
  plt.savefig(folder_name+"/nine_images.png")
  plt.close()

files= []
folder_names =
["num_genes/","num_inds/","parents/","tm_size/","mut_type/","mut_prob/","elite
s/","discussion/"]
for i in folder_names: #enter each folder and find all folders in each defined
folder


    folders = [f for f in os.listdir(i) if os.path.isdir(os.path.join(i, f))]
    for folder in folders:
      for count in range(2000,11000,1000):
        folder_name = str(i)+str(folder)
        files.append(folder_name+"/iteration_"+str(count)+'.png')
      plot_images(files,folder_name)
      files=[]
```