I did my best to complete the code. However, due to my illness I am not able to perform the tasks. I could not update the chromosomes for some reason that I could not find. I just want to obtain as much as possible points from the code that I wrote. Therefore, I uploaded only the code. I have no intention to disrespect. Thank you for your consideration.

```python
# -*- coding: utf-8 -*-
"""EE449_HW2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1cWjpemWtyFsWq5Od7wm5tT-
8JO2etROM
"""

from google.colab import drive
drive.mount('/content/gdrive')

#import necessary libraries and methods
import numpy as np
import cv2
from random import randint,uniform,random
from copy import deepcopy
from cv2 import imwrite, addWeighted, circle, imread
from google.colab.patches import cv2_imshow
import random

#Read the image and get the size of the image.
img = cv2.imread('/content/gdrive/MyDrive/EE449_HW2/painting.png')
img = img[:,:,0]
print(img.shape)
w = img.shape[0]
h = img.shape[1]
#image is 180x180

class Gene:
  def __init__(self, R=0,G=0,B=0,A=0,x=0,y=0,r=0):
    self.R = R
    self.G = G
    self.B = B
    self.A = A
    self.x = x
    self.y = y
    self.r = r
  def init_genes(self):
    #Initialize the genes with random parameters
    R = randint(0,255)
    G = randint(0,255)
    B = randint(0,255)
    A = uniform(0,1)
    x = randint(0,180)
    y = randint(0,180)
    r = randint(1,30)
    #Create new circle parameters until it is valid
    while not self.check_intersection(x,y,r):
      x = randint(0,180)
      y = randint(0,180)
      r = randint(1,30)
    return [R,G,B,A,x,y,r]
  #Method to check if there is an intersection between image and circle
  def check_intersection(self,x,y,r):
    x_distance = abs(w/2 - x)
    y_distance = abs(h/2 - y)
    #if both x and y distances are bigger than r, then
    if(x_distance > r and y_distance > r):
      return False
    else:
      return True
  def mutation_gene(self,mutation_type):
    self.mutation_type = mutation_type
    if mutation_type == 'unguided':
      #Initialize the genes with random parameters
      R = randint(0,255)
      G = randint(0,255)
      B = randint(0,255)
      A = uniform(0,1)
      x = randint(0,180)
      y = randint(0,180)
      r = randint(1,30)
      #Create new circle parameters until it is valid
      while not self.check_intersection(x,y,r):
        x = randint(0,180)
        y = randint(0,180)
        r = randint(1,30)
      return [R,G,B,A,x,y,r]
    else:
      #Guided mutation. Parameters are reviewed according to the
      #explanation in the homework manual.
      #int and max(0,int()) functions are added.
      #Otherwise I got "non-integer stop value" error for randint()
function.
      x = randint(max(0,int(self.x - w/4)) , int(self.x + w/4))
      y = randint(max(0,int(self.y - h/4)) , int(self.y + h/4))
      r = randint(max(0,int(self.r - 10)) , int(self.r + 10))
      R = randint(max(0,int(self.R-64)) , int(min(255,self.R+64)))
      G = randint(max(0,int(self.G-64)) , int(min(255,self.G+64)))
      B = randint(max(0,int(self.B-64)) , int(min(255,self.B+64)))
      A = randint(max(0,int(self.A-0.25)) , int(min(1,self.A+0.25)))
      while not self.check_intersection(x,y,r):
        x = randint(max(0,int(self.x - w/4)) , int(self.x + w/4))
        y = randint(max(0,int(self.y - h/4)) , int(self.y + h/4))
        r = randint(max(0,int(self.r - 10)) , int(self.r + 10))
      return [R,G,B,A,x,y,r]

class Individual:
  def __init__(self,ind_id,chrom=[],num_genes=50):
    self.num_genes = num_genes
    self.fitness = 0
    self.ind_id = ind_id
    self.chrom = chrom

  def assign_gene(self,num_genes=50):
    self.num_genes = num_genes
    self.chrom = []
    for i in range (0,num_genes):
      gene_1 = Gene()
      self.chrom.append(gene_1.init_genes())
  def draw_circle(self):
    blank = np.zeros([180, 180], dtype= np.uint8)#Create an image (180,180)
    blank.fill(255)#Make the image full white
    overlay = blank.copy()#Copy the image to draw a circle
    #Draw circle for each gene and combine the pictures.
    for genes in self.chrom:
      alpha = genes[3]
      beta = 1-alpha
      colour =(genes[2],genes[1],genes[0])
      #colour = (255,0,0)
      radius = genes[6]
      center = (genes[4],genes[5])
      #https://www.geeksforgeeks.org/numpy-ones-python/ is used as
reference.
      cv2.circle(overlay,center,radius, colour,-1)#Draw circle
      #cv2.circle(overlay,(genes[4],genes[5]),genes[6], (150,120,100),-
1)#Draw circle
      #https://docs.opencv.org/3.4/d5/dc4/tutorial_adding_images.html is
used as reference
      image = addWeighted(overlay,alpha,blank, beta,0.0)#Sum the blank
image and circle
    return image

  def mutation_ind(self,mutation_prob = 0.2,mutation_type = 'guided'):
    self.mutation_prob = mutation_prob
    self.mutation_type = mutation_type
    #I multiplied the probabilities with 100. Otherwise I get
    #ValueError: non-integer stop for randrange() error
    #Check mutation probability
    prob_checker = 1.1
    while(prob_checker < mutation_prob):
      #Update the value of probability checker randomly.
      prob_checker = random.uniform(0, 1)
      #Choose a random gene to mutate in the chromosome
      pick_gene= randint(1,self.num_genes)
      #Guided mutation
      if(mutation_type =="guided"):
        #Create a temp gene object to assign after mutation
        temp_gene = Gene()
        temp_gene = temp_gene.mutation_gene("guided")
        self.chrom[pick_gene] = temp_gene
      #Unguided mutation
      else:
        #Create a temp gene object to assign after mutation
        temp_gene = Gene()
        temp_gene = temp_gene.mutation_gene("unguided")
        self.chrom[pick_gene] = temp_gene

class Population:
  def __init__(self,population = []):
    self.population = []
  def add_to_population(self, Individual):
    #Add individuals to the population
    self.population.append(Individual)
  def evaluate(self):
    #self.population = sorted(self.population, key=lambda x: x.fitness,
reverse = True)
    for individual in self.population:
      #Create the drawing with respect to
      #Genes of the individual
      drawing = individual.draw_circle()
      #Evaluation according to the formula given in homework manual
      fitness = np.sum(-1*np.power(img-drawing,2))
      individual.fitness = fitness
      #print(fitness)
      self.population = sorted(self.population, key=lambda x: x.fitness,
reverse = True)
  def sort_by_radius(self,pop):
    self.pop = pop
    #Initialize population and add the individuals to the population to
return
    pop_sorted = Population()
    pop_sorted.population.clear()
    #Sort individuals with respect to their fitness
    pop_sorted.population = sorted(pop.population, key=lambda x:
x.chrom[6], reverse = True)
    #Place the sorted population to return population
    #for i in range(len(sorted_by_fitness)):
     # pop_sorted.population.append(deepcopy(sorted_by_fitness[i]))
    return pop_sorted

  def sort_by_fitness(self,pop):
    self.pop = pop
    #Initialize population and add the individuals to the population to
return
    pop_sorted = Population()
    pop_sorted.population.clear()
    #Sort individuals with respect to their fitness
    pop_sorted.population = sorted(pop.population, key=lambda x: x.fitness,
reverse = True)
    #Place the sorted population to return population
    #for i in range(len(sorted_by_fitness)):
     # pop_sorted.population.append(deepcopy(sorted_by_fitness[i]))
    return pop_sorted

  def select(self,pop,frac_elites = 0.2 ,num_inds=20,tm_size=5):
    self.tm_size = tm_size
    self.pop = pop
    self.frac_elites = frac_elites
    self.num_inds = num_inds
    #Calculate the number of elites
    num_elites = int(frac_elites*num_inds)
    #Create a parent population
    next_gen = Population()
    next_gen.population.clear()
    #In below for loop, elites are appended to the parents
    #when the num_elites are exceeded, remaining parts of the
    #parent population is filled with tournament selection
    for i in range(0,num_inds):
      if(i <= num_elites):
        temp = pop.population[i]
        next_gen.population.append(deepcopy(temp))
      else:
        winner, winner_index = pop.tournament(pop,tm_size)
        temp = pop.population[winner_index]
        next_gen.population.append(deepcopy(temp))

    return next_gen

  def tournament(self,pop,tm_size):
    self.pop = pop
    self.tm_size = tm_size
    #Create a population for the attendants of the tournament
    attend = Population()
    for i in range(1,tm_size+1):
      #https://www.edureka.co/blog/arrays-in-
```

```
python/#:~:text=Length%20of%20an%20array%20is,elements%20present%20in%20tha
t%20array.&text=This%20returns%20a%20value%20of,the%20number%20of%20array%2
0elements.
      #Above website is used as reference to find the number of individuals
in population
      t = randint(5,len(pop.population)-1)
      #Create a population for tournament
      attend.population.append(deepcopy(pop.population[t]))
    #Sort attendants with respect to their fitnesses.
    attend = attend.sort_by_fitness(attend)
    #After sorting, winner will be in the index [0] of the population
    winner = attend.population[0]
    place_of_winner = 0
    attend.population.clear()
    for individual in pop.population:
      if individual.ind_id == winner.ind_id:
        #Find the original location of the winner
        place_of_winner = place_of_winner
        return winner,place_of_winner
      else:
        place_of_winner = place_of_winner + 1
  def Crossover(self,parents,num_genes):
    self.parents = parents
    children = Population()
    children.population.clear()
    self.num_genes = num_genes
    par_length = int(len(parents.population)/2)
    for i in range(0,par_length+1):
      rand_chooser = randint(0,len(self.parents.population)-1)
      parent_1 = self.parents.population[rand_chooser]
      rand_chooser = randint(0,len(self.parents.population)-1)
      parent_2 = self.parents.population[rand_chooser]
      child_1 = deepcopy(parent_1)
      child_2 = deepcopy(parent_2)

      for i in range(0,num_genes):
        crossover_checker = uniform(0,1)
        if crossover_checker < 0.5:
          child_1.chrom[i] = deepcopy(parent_1.chrom[i])
          child_2.chrom[i] = deepcopy(parent_2.chrom[i])
        else:
          child_1.chrom[i] = deepcopy(parent_2.chrom[i])
          child_2.chrom[i] = deepcopy(parent_1.chrom[i])
      children.population.append(deepcopy(child_1))
      children.population.append(deepcopy(child_2))
    return children

#Create population with respect to the size.
pop = Population()
for i in range(1,51):
  ind = Individual(i)#Add num_genes parameter here
  ind.assign_gene()#Add num_genes parameter here
  pop.add_to_population(ind)
#Create population with respect to the size.

num_of_generations = 10000
num_inds = 50
frac_parents = 0.6 #adjust frac_parents here.
for i in range(0,10001):
  print(i)
  #Sort population with respect to their radius values.
  sorted_radius = pop.sort_by_radius(pop)

  #Evaluate the sorted population.
  sorted_radius.evaluate()

  #After drawing, sort the population with respect to their fitness values.
  sorted_fitness = sorted_radius.sort_by_fitness(sorted_radius)

  #Select the parents of the next generation
  parents = pop.select(pop, 0.2, num_inds, 5) #adjust tm_size and
frac_elites here.

  #Create next generation population with respect to the fitness values
  next_gen = Population()
  sum_parents = int(frac_parents * num_inds)
  sorted_parents = parents.sort_by_fitness(parents)
  for j in range(0,sum_parents):
    next_gen.population.append(deepcopy(sorted_parents.population[j]))

  #Create children from the parents selected
  Children = parents.Crossover(parents,num_inds)

  #Combine the children and the first population created
  combined = deepcopy(parents)
  for i in range (0,len(Children.population)):
    combined.population.append(Children.population[i])

  #Apply mutation to the combined population
  for individuals in combined.population:
    individuals.mutation_ind(0.2, 'guided') #adjust mutation_prob and
mutation_type here.

  #Sort the combined population with respect to the fitness after mutation
  sorted_fitness_combined = combined.sort_by_fitness(combined)
  sorted_fitness_combined.evaluate()

  for i in range(0, num_inds-sum_parents):

next_gen.population.append(deepcopy(sorted_fitness_combined.population[i]))

  #Clear the intermediate populations
  sorted_radius.population.clear()
  sorted_fitness.population.clear()
  parents.population.clear()
  Children.population.clear()
  combined.population.clear()
  next_gen_sorted = next_gen.sort_by_fitness(next_gen)
  pop = deepcopy(next_gen_sorted)
  next_gen.population.clear()
  next_gen_sorted.population.clear()
  for ind in pop.population:
    print(ind.fitness)
```