```
#Importing necessary libraries
import cv2
from random import randint
from random import random
from random import uniform
from copy import deepcopy
import numpy as np
import matplotlib.pyplot as plt

#Initialize variables with an input image
input_path='inputs/example.png'
im = cv2.imread(input_path)
width,height,_ = im.shape
# Reference to :https://note.nkmk.me/en/python-opencv-pillow-image-size/
print("Widht is ", width)
print("Height is ",height)

#Gene Class
#-----------------------------------------------------------------------------------------------
----------------
class gene:
    def __init__(self,ID):
        #Constructor Method
        while True:
            #random intialize of radius x and y
            self.r=randint(1,40)
            self.x= int(1.5 * width * random())#int function is used
            self.y= int(1.5 * height * random())#otherwise cirlce function is giving me
an error
            if self.intersects() == True:#If intersects does not validate, break the loop
                break
            #If it does not intersect with our graph, then randomize again
        #Directly randomize values
        self.red=randint(0,255)
        self.blue=randint(0,255)
        self.green=randint(0,255)
        self.alpha=random()
        self.ID = ID


    def intersects(self): #This method is checking whether circle is intersecting with
graph or not
        if(self.x < width and self.y < height):
            return True#Inside the image
        elif(self.x < width and self.y -height < self.r):
            return True#Upper region of image
        elif(self.y < height and self.x -width < self.r):
            return True#Right region of image
        else:#At the right upper corner

            hipo = (self.x-width)* (self.x-width) +(self.y-height) *(self.y-height)
            # If it is intersects with right upper corner
            if(hipo < self.r*self.r ):
                return True
            else:
                return False
    #Apply mutation on the gene
    def mutate(self,ref):
        if ref =="G":#Guided
            #Took the original values
            radii=self.r
            xii=self.x
            yii=self.y
            while True:
                offset=randint(-10,10)#Randomize offset value
                if(radii+offset > 0): #If it does not go to out of region, then assign
offset
                    self.r= radii+offset    #Assign
                    break
                #If it goes out of region, then randomize offset value again
            while True:
                self.x=randint(max(0,int(xii-width/4)),int(xii+width/4))#Randomize x value
                self.y=randint(max(0,int(yii-width/4)),int(yii+width/4))#Randomize yvalue
                if self.intersects() == True:#If it does not go to out of region, then
```

```
assign these values
                    break
                #If it does not go to out of region, then randomize again
            while True:
                offset=randint(-64,64)#Randomize offset value
                if ((self.red+offset > -1) and(self.red+offset < 256)):#If it does not go
to out of definition boundaries, then assign offset
                    self.red= self.red+offset   #Assign
                    break
                #If it goes out of  definition boundaries, then randomize offset value
again
            while True:
                offset=randint(-64,64)#Randomize offset value
                if ((self.blue+offset > -1) and(self.blue+offset < 256)):#If it does not
go to out of definition boundaries, then assign offset
                    self.blue= self.blue+offset   #Assign
                    break
                #If it goes out of  definition boundaries, then randomize offset value
again
            while True:
                offset=randint(-64,64)#Randomize offset value
                if ((self.green+offset > -1) and(self.green+offset < 256)):#If it does
not go to out of definition boundaries, then assign offset
                    self.green= self.green+offset   #Assign
                    break
                #If it goes out of  definition boundaries, then randomize offset value
again
            while True:
                #Reference to:
                #https://www.geeksforgeeks.org/python-number-uniform-method/
                offset=uniform(-0.25,0.25)#Randomize offset value
                if ((self.alpha+offset >= 0) and(self.alpha+offset <= 1)):#If it does not
go to out of definition boundaries, then assign offset
                    self.alpha= self.alpha+offset   #Assign
                    break
                #If it goes out of  definition boundaries, then randomize offset value
again
        else: #Unguided
            #Apply same randomize with __init__ method
            while True:
                self.r=randint(1,40)
                self.x= int(1.5 * width * random())#int function is used
                self.y= int(1.5 * height * random())#otherwise cirlce function is giving
me an error
                if self.intersects() == True:
                    break
            self.red=randint(0,255)
            self.blue=randint(0,255)
            self.green=randint(0,255)
            self.alpha=random()

#-------------------------------------------------------------------------------------------
------------
#Individual Class
class indv:
    def __init__(self,ID=-1,num_genes=50):
        #Constructor Method
        self.ID=ID
        self.num_genes=num_genes
        #Create empty list for choromosomes
        self.chromosome = list()
        for i in range(1,num_genes+1):
            bi=gene(i)#create temp gene for chromosome
            self.chromosome.append(bi)

    def evaulation(self):
        #Reference to:

#https://www.techiedelight.com/sort-list-of-objects-python/#:~:text=A%20simple%20solution%2
0is%20to,only%20arguments%3A%20key%20and%20reverse.
        self.chromosome.sort(key=lambda x: x.r, reverse=True)

        #Reference to:
        #https://www.geeksforgeeks.org/create-a-white-image-using-numpy-in-python/
```

```
        image = np.full((width, height, 3),255, dtype = np.uint8)
        for i in self.chromosome:
            #overlay <- image
            #Avoid shallow copy issues, assign it with deep copy
            #Reference to :
            #https://docs.python.org/3/library/copy.html
            overlay=deepcopy(image)

            # Draw the circle on overlay.
            # Reference to:
            # https://www.geeksforgeeks.org/python-opencv-cv2-circle-method/
            # Center coordinates
            center_coordinates = (i.x, i.y)
            # Radius of circle
            radius = i.r
            # color in BGR
            color = (i.blue, i.green, i.red)
            # Line thickness of -1 px
            thickness = -1
            cv2.circle(overlay, center_coordinates, radius, color, thickness)

            # image <- overlay x alpha + image x (1-alpha)
            # Reference to:
            # https://www.educba.com/opencv-addweighted/
            cv2.addWeighted(overlay, i.alpha, image, (1-i.alpha), 0.0,image)

        #Calculating fitness of INDV
        diff=np.subtract(np.array(im, dtype=np.int64), np.array(image, dtype=np.int64))
        self.fitness = np.sum(-1*np.power(diff, 2))
        if self.fitness >0:
            print("ERROR-------------------------------------------------------------")
            #Undesired situtation
            #Code should not be entered this region

    def takeImage(self):#This method will be called when a image is regiured
        #Reference to:

#https://www.techiedelight.com/sort-list-of-objects-python/#:~:text=A%20simple%20solution%20is%20to,only%20arguments%3A%20key%20and%20reverse.
        self.chromosome.sort(key=lambda x: x.r, reverse=True)

        #Reference to:
        #https://www.geeksforgeeks.org/create-a-white-image-using-numpy-in-python/
        image = np.full((width, height, 3),255, dtype = np.uint8)
        for i in self.chromosome:
            #overlay <- image
            #Avoid shallow copy issues, assign it with deep copy
            #Reference to :
            #https://docs.python.org/3/library/copy.html
            overlay=deepcopy(image)

            # Draw the circle on overlay.
            # Reference to:
            # https://www.geeksforgeeks.org/python-opencv-cv2-circle-method/
            # Center coordinates
            center_coordinates = (i.x, i.y)
            # Radius of circle
            radius = i.r
            # color in BGR
            color = (i.blue, i.green, i.red)
            # Line thickness of -1 px
            thickness = -1
            cv2.circle(overlay, center_coordinates, radius, color, thickness)

            # image <- overlay x alpha + image x (1-alpha)
            # Reference to:
            # https://www.educba.com/opencv-addweighted/
            cv2.addWeighted(overlay, i.alpha, image, (1-i.alpha), 0.0,image)
        return image

    def mutation(self,prob,guide):
        while(random()<prob):#Do it with a given probablity
            #mutate random gene in the chromosome
            genMutated= randint(0,self.num_genes-1)
```

```
            if(guide=="guided"):#guided
                self.chromosome[genMutated].mutate("G")

            else:#unguided
                self.chromosome[genMutated].mutate("U")

#-------------------------------------------------------------------------------------------
#Population Class
class popu:
    def
__init__(self,ID,num_idv,num_genes,num_iteration,frac_elites,frac_parents,tm_size,mutation_
prob,mutation_guide):
        #Constructor method for population
        self.mutation_prob=mutation_prob
        self.mutation_guide=mutation_guide
        self.num_iteration=num_iteration
        self.frac_elites=frac_elites
        self.frac_parents=frac_parents
        self.ID=ID
        self.num_idv=num_idv
        self.num_genes=num_genes
        self.tm_size=tm_size
        #Clear individual lists
        self.indvs = list()
        for i in range(0,num_idv):
            bi=indv(i,num_genes)#Temp Indv to add to population
            self.indvs.append(bi)
            #Create individual

    def evaluation(self):#This method is our main method to train, it will train our
model and record all necessary information
        print(f"Evaloation{self.ID} is started...")#Informative printing
        number=0
        #Clear all necessary lists
        self.allfitness=list()#This will hold all fitness values corresponds to population
        self.parents=list()#Parents will hold the individuals who will have children
        self.children=list()#Children will hold the individuals who is created with
crossover
        self.elites=list()#Elites will hold the individuals who can go to next generation
directly
        for i in range(1,self.num_iteration+1):
            #---------------------------------------SUGGESTION
            # #-------------------------------------TO SEE MY SUGG1 UNCOMMMENT THIS SIDE
            # if(self.ID==27):
            #     if (i<300):
            #         self.mutation_prob=0.8
            #     elif(i<400):
            #         self.mutation_prob=0.5
            #     elif(i<600):
            #         self.mutation_prob=0.2
            #     else:
            #         self.mutation_prob=0.1
            # #-------------------------------------TO SEE MY SUGG2 UNCOMMMENT THIS SIDE
            # if(self.ID==29):
            #     if (i<500):
            #         self.frac_elites=0.05
            #         self.frac_parents=0.8
            #     elif(i<750):
            #         self.frac_elites=0.2
            #         self.frac_parents=0.6
            #     else:
            #         self.frac_elites=0.4
            #         self.frac_parents=0.4

            #-------------------------------------------------------------------------


            #Calculate fitness value for each individual in the generation
            for j in self.indvs:
                j.evaulation()

            #Calculate number of elites and parents
            self.num_elites=int(self.frac_elites * self.num_idv)
```

```
            self.num_parent=int(self.frac_parents * self.num_idv)


            #if it is even number add 1
            if self.num_parent % 2 == 1:
                self.num_parent = self.num_parent + 1

            #Sort the individuals according to their fitness values
            #Reference to:

#https://www.techiedelight.com/sort-list-of-objects-python/#:~:text=A%20simple%20solution%20is%20to,only%20arguments%3A%20key%20and%20reverse.
            self.indvs.sort(key=lambda x: x.fitness, reverse=True)

            #Choosing elites list
            #Clear at the beggining
            self.elites.clear()
            for z in range(self.num_elites):
                #Since we sorted the indvs (.:Fitness Values)
                #The first z ones will be z amount individual with highest fitness
                #Avoid shallow copy issues, assign it with deep copy
                #Reference to :
                #https://docs.python.org/3/library/copy.html
                self.elites.append(deepcopy(self.indvs[z]))

            #Start Tournament
            #Since we want to get num_parents amount of parents, and every tournament
will give us one winner
            #we should do tournament num_parents times.
            for _ in range(self.num_parent):
                self.parents.append(self.tournament())
            #Tournament is finished -----------
            #Record the best fitness
            self.allfitness.append(self.elites[0].fitness)

            #Remove elites from individuals
            for _ in range(self.num_elites):
                self.indvs.pop(0)


            #Crossover Starts
            #Clear the children lists
            self.children.clear()
            #Call the crossover method
            self.crossover()
            #Crossover Finishes --------------

            #Mutation Starts
            self.mutation(i)
            #Mutation finishes ---------------

            #Print informative message
            if i% 200 == 0:
                print(f"Iteration {i}")
                print(self.elites[0].fitness)
            #Record the image for each 1000th generation
            if i% 1000 ==0:
                #Self.ID is our population ID
                print("Population ID is ",self.ID)
                #Record each image for differnet folder locaion
                if self.ID in [0,1,2,3,4]:

cv2.imwrite("Output/NUM_OF_INDV/Res_NI_"+str(self.num_idv)+"_I_"+str(number)+".png",self.in
dvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [5,6,7,8,9]:

cv2.imwrite("Output/NUM_OF_GENES/Res_NG_"+str(self.num_genes)+"_I_"+str(number)+".png",self
.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [10,11,12,13]:

cv2.imwrite("Output/TM_SIZE/Res_TS_"+str(self.tm_size)+"_I_"+str(number)+".png",self.indvs[
0].takeImage())
```

```
                    print("TOOK IMAGE")
                if self.ID in [14,15,16]:

cv2.imwrite("Output/FRAC_ELITE/Res_FE_"+str(self.frac_elites)+"_I_"+str(number)+".png",self
.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [17,18,19,20]:

cv2.imwrite("Output/FRAC_PARENT/Res_FP_"+str(self.frac_parents)+"_I_"+str(number)+".png",se
lf.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [21,22,23,24]:

cv2.imwrite("Output/MUT_PROB/Res_MP_"+str(self.mutation_prob)+"_I_"+str(number)+".png",self
.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [25,26]:

cv2.imwrite("Output/MUT_GUI/Res_MG_"+str(self.mutation_guide)+"_I_"+str(number)+".png",self
.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [27]:

cv2.imwrite("Output/SUGG_1/Res_updated_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [28]:

cv2.imwrite("Output/SUGG_1/Res_old_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [29]:

cv2.imwrite("Output/SUGG_2/Res_updated_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [30]:

cv2.imwrite("Output/SUGG_2/Res_old_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [31]:

cv2.imwrite("Output/SUGG_3/Res_updated_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                if self.ID in [32]:

cv2.imwrite("Output/SUGG_3/Res_old_I_"+str(number)+".png",self.indvs[0].takeImage())
                    print("TOOK IMAGE")
                number+=1
        print(f"Evaloation{self.ID} is ended...")
        #Reference to:

#https://chartio.com/resources/tutorials/how-to-save-a-plot-to-a-file-using-matplotlib/
        plt.figure()
        plt.plot(self.allfitness[999:])#Print the fitness values from 1000-10000
        #Record each image for differnet folder locaion
        if self.ID in [0,1,2,3,4]:
            plt.savefig("Output/NUM_OF_INDV/Res_Fitness1000_NI_"+str(self.num_idv)+".png")
        if self.ID in [5,6,7,8,9]:

plt.savefig("Output/NUM_OF_GENES/Res_Fitness1000_NG_"+str(self.num_genes)+".png")
        if self.ID in [10,11,12,13]:
            plt.savefig("Output/TM_SIZE/Res_Fitness1000_TS_"+str(self.tm_size)+".png")
        if self.ID in [14,15,16]:

plt.savefig("Output/FRAC_ELITE/Res_Fitness1000_FE_"+str(self.frac_elites)+".png")
        if self.ID in [17,18,19,20]:

plt.savefig("Output/FRAC_PARENT/Res_Fitness1000_FP_"+str(self.frac_parents)+".png")
        if self.ID in [21,22,23,24]:

plt.savefig("Output/MUT_PROB/Res_Fitness1000_MP_"+str(self.mutation_prob)+".png")
        if self.ID in [25,26]:

plt.savefig("Output/MUT_GUI/Res_Fitness1000_MG_"+str(self.mutation_guide)+".png")
        if self.ID in [27]:
            plt.savefig("Output/SUGG_1/Res_Fitness1000_updated.png")
```

```
        if self.ID in [28]:
            plt.savefig("Output/SUGG_1/Res_Fitness1000_old.png")
        if self.ID in [29]:
            plt.savefig("Output/SUGG_2/Res_Fitness1000_updated.png")
        if self.ID in [30]:
            plt.savefig("Output/SUGG_2/Res_Fitness1000_old.png")
        if self.ID in [31]:
            plt.savefig("Output/SUGG_3/Res_Fitness1000_updated.png")
        if self.ID in [32]:
            plt.savefig("Output/SUGG_3/Res_Fitness1000_old.png")
        plt.figure()
        plt.plot(self.allfitness[:])#Print the fitness values from 1-10000
        #Record each image for differnet folder locaion
        if self.ID in [0,1,2,3,4]:
            plt.savefig("Output/NUM_OF_INDV/Res_FitnessAll_NI_"+str(self.num_idv)+".png")
        if self.ID in [5,6,7,8,9]:

plt.savefig("Output/NUM_OF_GENES/Res_FitnessAll_NG_"+str(self.num_genes)+".png")
        if self.ID in [10,11,12,13]:
            plt.savefig("Output/TM_SIZE/Res_FitnessAll_TS_"+str(self.tm_size)+".png")
        if self.ID in [14,15,16]:

plt.savefig("Output/FRAC_ELITE/Res_FitnessAll_FE_"+str(self.frac_elites)+".png")
        if self.ID in [17,18,19,20]:

plt.savefig("Output/FRAC_PARENT/Res_FitnessAll_FP_"+str(self.frac_parents)+".png")
        if self.ID in [21,22,23,24]:

plt.savefig("Output/MUT_PROB/Res_FitnessAll_MP_"+str(self.mutation_prob)+".png")
        if self.ID in [25,26]:

plt.savefig("Output/MUT_GUI/Res_FitnessAll_MG_"+str(self.mutation_guide)+".png")
        if self.ID in [27]:
            plt.savefig("Output/SUGG_1/Res_FitnessAll_updated.png")
        if self.ID in [28]:
            plt.savefig("Output/SUGG_1/Res_FitnessAll_old.png")
        if self.ID in [29]:
            plt.savefig("Output/SUGG_2/Res_FitnessAll_updated.png")
        if self.ID in [30]:
            plt.savefig("Output/SUGG_2/Res_FitnessAll_old.png")
        if self.ID in [31]:
            plt.savefig("Output/SUGG_3/Res_FitnessAll_updated.png")
        if self.ID in [32]:
            plt.savefig("Output/SUGG_3/Res_FitnessAll_old.png")

    #Mutation method is to mutate population
    def mutation(self,ind=0):
        #Create indvs list for who are applied to mutation
        #Children and other individuals(we excluded elites already )
        mutationTeam=self.children + self.indvs
        for i_indv in mutationTeam:
            #-----------------------------------------------------SUGGESTION 3
            #-------------------------------------------------FORCED MUTATION
            if(self.ID==31 and (ind > 0) and (ind<5)):#At the beggingaing ,it isforced
                i_indv.evaulation()#Calculate fitness value
                prevFitness=i_indv.fitness#Hold the previous fitness value
                i_indv.mutation(self.mutation_prob,self.mutation_guide)#Mutate
                i_indv.evaulation()#Evalute again
                afterFitness=i_indv.fitness
                while True:
                      if(afterFitness > prevFitness):#If there is a upgrade on fitness,
                          #Finish the mutation
                          break
                      else:
                          #If not,
                          #Mutate again untill there is a upgrade on fitness
                          i_indv.mutation(self.mutation_prob,self.mutation_guide)
                          i_indv.evaulation()
                          afterFitness=i_indv.fitness
            #-----------------------------------------------------SUGGESTION 3  ENDS
            else:
            #-----------------------------------------------DIRECT MUTATION
                i_indv.mutation(self.mutation_prob,self.mutation_guide)
        #Assign new generation list to indvs
```

```
        #Avoid shallow copy issues, assign it with deep copy
        #Reference to :
        #https://docs.python.org/3/library/copy.html
        self.indvs=deepcopy( self.children +self.elites + self.indvs)
    #Crossover method
    #This will update children list with newly created
        #individuals crossovering parents
    def crossover(self):
        for _ in range(0,self.num_parent,2):#Iterate the amount of parents divided by two
                                            #Since each children has two parents
            father=self.parents.pop(randint(0,len(self.parents)-1))#Randomly assign father
            mother=self.parents.pop(randint(0,len(self.parents)-1))#Randomly assign mother
            childrenA=indv(father.ID,self.num_genes)#Create new children
            childrenB=indv(mother.ID,self.num_genes)#Create new children
            #For each gen, randomize a number between 0 and 1
            # if it is smaller than 0.5
            #father will give the gene to children
            #if not
            #mother will give the gene to children
            for i in range(0,self.num_genes):
                res=uniform(0,1)
                if res<0.5:
                    childrenA.chromosome[i]=father.chromosome[i]
                    childrenB.chromosome[i]=mother.chromosome[i]
                else:
                    childrenA.chromosome[i]=mother.chromosome[i]
                    childrenB.chromosome[i]=father.chromosome[i]
            #Update the children list to
            self.children.append(childrenA)
            self.children.append(childrenB)
    #Tourname method will give us a winner
    def tournament(self):
        #Randomly choose one of them
        #Assign it as temporary winner
        bestInd=randint(0,len(self.indvs)-1)
        bestFitness=self.indvs[bestInd].fitness

        #Since we initialize with a random assignment
        #One of the warrior is decided
        #Therefore, we should iterate tm_size - 1 times
        for _ in range(self.tm_size-1):
            #Randomly choose one of them as a warrior
            currentInd=randint(0,len(self.indvs)-1)
            #Take the warrior's fitness value
            currentFitness=self.indvs[currentInd].fitness
            #Compare it with the best one
            if(currentFitness > bestFitness):
                #If the iterated warrior wins
                #Label him as best
                bestFitness=currentFitness
                bestInd=currentInd
        #end of for loop
        #Winner is decided
        #Temporary indv to be added to parent since it is the winner.
        temp=self.indvs[bestInd]
        self.indvs.pop(bestInd)#Delete the winner from current generation
        return temp#Return it so that parents will be updated correctly

#Hyperparameters----------------------------------
num_of_indv=[5,10,20,50,75]
num_of_genes=[10,25,50,100,150]
tm_sizes=[2,5,10,20]
frac_elites=[0.05,0.2,0.4]
frac_parents=[0.2,0.4,0.6,0.8]
mut_probs=[0.1,0.2,0.5,0.8]
mut_gui=["guided","unguided"]
num_generation=10000
#num_generation = 2000 #-----------------------------------------UNCOMMENT TO SEE MY
SUGGESTION 2-3 IN A FASTER WAY


#What do you want??
#Choose one of them and uncomment to see the results
#Do not organize folder tree so that recording can be done!!!!
```

```
#------------------------NUM OF INDV-----------------------------------------------------
# tempPop
=popu(0,num_of_indv[0],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop.evaluation()
# tempPop1
=popu(1,num_of_indv[1],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop1.evaluation()
# tempPop2
=popu(2,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop2.evaluation()
# tempPop3
=popu(3,num_of_indv[3],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()
# tempPop4
=popu(4,num_of_indv[4],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop4.evaluation()
# playsound("cak.mp3")
#------------------------NUM OF GENES----------------------------------------------------
# tempPop
=popu(5,num_of_indv[2],num_of_genes[0],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop.evaluation()
# tempPop1
=popu(6,num_of_indv[2],num_of_genes[1],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop1.evaluation()
# # tempPop2
=popu(7,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# # tempPop2.evaluation() SAME WITH DEFAULT CASE
# tempPop3
=popu(8,num_of_indv[2],num_of_genes[3],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()
# tempPop4
=popu(9,num_of_indv[2],num_of_genes[4],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop4.evaluation()

# #------------------------TM SIZE---------------------------------------------------------
# tempPop
=popu(10,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[0],mut_probs[1],mut_gui[0])
# tempPop.evaluation()
# # tempPop1
=popu(11,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# # tempPop1.evaluation() SAME WITH DEFAULT CASE
# tempPop2
=popu(12,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[2],mut_probs[1],mut_gui[0])
# tempPop2.evaluation()
# tempPop3
=popu(13,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[3],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()

# #------------------------FRAC ELITE------------------------------------------------------
# tempPop
=popu(14,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[0],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop.evaluation()
# # tempPop1
=popu(15,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
# # tempPop1.evaluation() SAME WITH DEFAULT CASE
# tempPop2
=popu(16,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[2],frac_parents[2],tm_sizes[1],mut_probs[1],mut_gui[0])
```

```
# tempPop2.evaluation()


# #------------------------FRAC PARENTS-----------------------------------------------------
# tempPop
=popu(17,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[0],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop.evaluation()
# tempPop1
=popu(18,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[1],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop1.evaluation()
# # tempPop2
=popu(19,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# # tempPop2.evaluation() SAME WITH DEFAULT CASE
# tempPop3
=popu(20,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[3],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()

# #------------------------MUT PROB---------------------------------------------------
# tempPop
=popu(21,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[0],mut_gui[0])
# tempPop.evaluation()
# # tempPop1
=popu(22,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# # tempPop1.evaluation() SAME WITH DEFAULT CASE
# tempPop2
=popu(23,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[2],mut_gui[0])
# tempPop2.evaluation()
# tempPop3
=popu(24,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[3],mut_gui[0])
# tempPop3.evaluation()

# #------------------------MUT GUIDE---------------------------------------------------
# tempPop
=popu(25,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop.evaluation()   SAME WITH DEFAULT CASE
# tempPop4
=popu(26,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[1])
# tempPop4.evaluation()


# #------------------------SUG 1---------------------------------------------------
# tempPop1
=popu(27,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop1.evaluation()
# tempPop2
=popu(28,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop2.evaluation()

#------------------------SUG 2---------------------------------------------------
# tempPop2
=popu(29,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
# tempPop2.evaluation()
#
tempPop3=popu(30,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[
2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()

#------------------------SUG 3---------------------------------------------------
# tempPop2
=popu(31,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[2],tm_si
zes[1],mut_probs[1],mut_gui[0])
```

```
# tempPop2.evaluation()
#
tempPop3=popu(32,num_of_indv[2],num_of_genes[2],num_generation,frac_elites[1],frac_parents[
2],tm_sizes[1],mut_probs[1],mut_gui[0])
# tempPop3.evaluation()
```