```python
## Pycasso, an evolutionary algorithm for the automatic generation of paintings.
#      ________
#    /          \  __     __    _____       _        _____     _____    ___
#   |            | \ \   / /  /       \   /   \    / ____|   / ____|  /     \
#   |            |  \ \_/ /  /    ____|  /  ^  \  | (____   | (____  |       |
#   |    _______/    \   /   |   |      |  /_\  |  \____ \   \____ \ |       |
#   |  /              | |    |   |      | /   \ |  ____| |   ____| | \___/
#   |/                |_|     \____ \   |/     \| |______|  |______|
#                                   \)
#
# https://www.petercollingridge.co.uk/blog/evolving-images/


import cv2
import numpy as np
import random

import math
import time

import matplotlib.pyplot as plt
from matplotlib import rc
import copy
import os


# Global variables
source_image_path = "images/"
source_image_name = "cafe_terrace_at_night.png"
source_image = cv2.imread(source_image_path + source_image_name)
# test = cv2.imread("images/test.png")
# test_image = cv2.imread("images/test_son.png")
h = source_image.shape[0]
w = source_image.shape[1]
s_max = int(math.sqrt(h**2+w**2)*0.4)   # Maximum circle size, diagonal of the image, radius  int(math.sqrt(h**2+w**2)*0.35)
h_margin = 1*h               # Horizontal margin
w_margin = 1*w               # Vertical margin

# Information Display---------------------------------------------------------##
print_info = True
print_intervals = 10000
save = True

# Variables-------------------------------------------------------------------##
num_inds = 20 #20  # Individual Number
num_genes = 50 #50 # Gene Number
num_generations = 10000 # Generation Number

tm_size = 5 # Tournament size
frac_elites = 0.2 # Fraction of elites
frac_parents = 0.6 # Fraction of parents
mutation_prob = 0.2  # Mutation probability
mutation_type = 0 # Mutation type 0 = unguided(random), 1 = guided (within limits)

output_path =
"outputs/"+str(num_inds)+"_"+str(num_genes)+"_"+str(tm_size)+"_"+str(frac_elites)+"_"+str(frac_parents)+"_"+str(mutation_prob)+
"_"+str(mutation_type)+"/"

# rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
# rc('text', usetex=True)

plt.rcParams['figure.figsize'] = [8, 4]

# Class for individual
class Individual:
    def __init__(self, genes, fitness):
        self.genes = genes
        self.fitness = fitness

# Class for gene
class Gene:
    def __init__(self, x, y, s, r, g, b, a):
        self.x = x
        self.y = y
        self.s = s
        self.r = r
        self.g = g
        self.b = b
        self.a = a
```

```python
# Population initialization
def init_population():
    population = []
    for i in range(num_inds):
        genes = []
        for j in range(num_genes):
            x = random.randint(-h_margin, h+h_margin)
            y = random.randint(-w_margin, w+w_margin)
            s = random.randint(0, s_max)
            r = random.randint(0, 255)
            g = random.randint(0, 255)
            b = random.randint(0, 255)
            a = random.uniform(0,1)
            genes.append(Gene(x, y, s, r, g, b, a))
        population.append(Individual(genes, 1))
    if print_info == True:
        print("Population initialized, checking collisions...")

    # Check for collisions
    for individual in population:
        for gene in individual.genes:
            while not check_circle(gene):
                gene.x = random.randint(-h_margin, h+h_margin)
                gene.y = random.randint(-w_margin, w+w_margin)
                gene.s = random.randint(0, s_max)
    if print_info == True:
        print("Collisions checked, Evaluating population...")

    # Evaluate population
    for individual in population:
        evaluate_individual(individual)
    if print_info == True:
        print("Population evaluated, returning population...")

    return population

# Check if a circle is inside the image
# https://stackoverflow.com/questions/75231142/collision-detection-between-circle-and-rectangle

# Second method is more accurate, but slower. (distance)
def check_circle(gene):
    # nearest point on rectangle to circle
    nearest_x = max(0, min(gene.x, h))
    nearest_y = max(0, min(gene.y, w))

    # distance from circle center to nearest point
    dx = nearest_x - gene.x
    dy = nearest_y - gene.y

    # if distance is less than circle radius, there is a collision
    return (dx**2 + dy**2) < (gene.s**2)

# Individual Evaluation
def evaluate_individual(individual):
    # Sort genes by size
    individual.genes.sort(key=lambda x: x.s, reverse=True)

    image = np.zeros([source_image.shape[0],source_image.shape[1],3],dtype=np.uint8)
    image.fill(255)
    for gene in individual.genes:
        overlay = image.copy()
        cv2.circle(overlay, (gene.x, gene.y), gene.s, (gene.r, gene.g, gene.b), -1)
        image = cv2.addWeighted(overlay, gene.a, image, 1 - gene.a, 0)
        # cv2.imshow("image", image)
        # cv2.waitKey(0)
        # cv2.destroyAllWindows()

    fitness = 0
    fitness = np.sum(np.square(image.astype(np.int64)-source_image.astype(np.int64)))

    individual.fitness = -1*fitness
    return

# Tournament selection
def tournament_selection(population):
    tournament = []
    candidates = population.copy()

    # for high number of individuals, tournament size is limited to the number of individuals
    size = tm_size
    if tm_size > len(candidates):
        size = len(candidates)

    # main tournament loop
    for i in range(size) :
        tournament.append(random.choice(candidates))
        candidates.remove(tournament[i])
    best = tournament[0]
    for individual in tournament:
        if individual.fitness > best.fitness:
            best = individual
    return best

# Elitism
def elitism(population):
    attendees = copy.deepcopy(population)
    elites = []
    best_index = []
    for i in range(int(frac_elites*num_inds)):
        j = 0
        k = 0
        best = attendees[0]
        for individual in attendees:
            if individual.fitness > best.fitness:
                k = j
                best = individual
            j+=1
        best_index.append(k)
        elites.append(best)
        attendees.remove(best)
    # print("best index: ", best_index)
    # for j in sorted(best_index, reverse=True):
    #     del population[j]
    return elites


def natural_selection(population):
    parents = []
    testants = population
    for i in range(int(num_inds - int(frac_elites*num_inds)- int(frac_parents*num_inds))):
        parents.append(tournament_selection(testants))
        testants.remove(parents[i])
        ### population.remove(parents[i])
    return parents

def parent_selection(population):
    parents = []
    testants = population
    for i in range(int(frac_parents*num_inds)):
        parents.append(tournament_selection(testants))
        testants.remove(parents[i])
        ### population.remove(parents[i])
    return parents

# Crossover
def crossover(parents):
    children = []
    for i in range(int(frac_parents*num_inds/2)):
        child1 = []
        child2 = []
        for j in range(num_genes):
            if random.random() < 0.5:
                child1.append(parents[i].genes[j])
                child2.append(parents[i+1].genes[j])
            else:
                child1.append(parents[i+1].genes[j])
                child2.append(parents[i].genes[j])
        children.append(Individual(child1, 2))
        children.append(Individual(child2, 2))
    return children

#check the negative limits and correct them
def within_limits(phenotype, upper_lim, lower_lim,range):
    while True:
        phenotype_calc = phenotype + random.uniform(-range, range)
        if phenotype_calc < upper_lim and phenotype_calc > lower_lim:
            phenotype = phenotype_calc
            break
    return phenotype


def mutation(population):
    population = copy.deepcopy(population)
    for individual in population:
        individual.fitness = 1    # mutated individuals are not evaluated
        for gene in individual.genes:
            if random.random() < mutation_prob:
                if mutation_type == 1:
                    while not check_circle(gene):
                        gene.x = int(within_limits(gene.x, w+w_margin, -w_margin, w/4))
                        gene.y = int(within_limits(gene.y, h+h_margin, -h_margin, h/4))
                        gene.s = int(within_limits(gene.s, s_max, 0, 10))
                    gene.r = int(within_limits(gene.r, 255, 0, 64))
                    gene.g = int(within_limits(gene.g, 255, 0, 64))
                    gene.b = int(within_limits(gene.b, 255, 0, 64))
                    gene.a = within_limits(gene.a, 1, 0, 0.25)
                if mutation_type == 0:
                    while not check_circle(gene):
                        gene.x = random.randint(-w_margin, w+w_margin)
                        gene.y = random.randint(-h_margin, h+h_margin)
                        gene.s = random.randint(0, s_max)
                    gene.r = random.randint(0, 255)
                    gene.g = random.randint(0, 255)
                    gene.b = random.randint(0, 255)
                    gene.a = random.uniform(0,1)
    return population

# # # Mutation : single phenotype
# def mutation(population):
#     population = copy.deepcopy(population)
#     for individual in population:
#         for gene in individual.genes:
#             if mutation_type == 0:
#                 while not check_circle(gene):
#                     if random.random() < mutation_prob:
#                         gene.x = random.randint(-w_margin, w+w_margin)
#                     if random.random() < mutation_prob:
#                         gene.y = random.randint(-h_margin, h+h_margin)
#                     if random.random() < mutation_prob:
#                         gene.s = random.randint(0, s_max)
#                 if random.random() < mutation_prob:
#                     gene.r = random.randint(0, 255)
#                 if random.random() < mutation_prob:
#                     gene.g = random.randint(0, 255)
#                 if random.random() < mutation_prob:
#                     gene.b = random.randint(0, 255)
#                 if random.random() < mutation_prob:
#                     gene.a = random.uniform(0,1)
#             elif mutation_type == 1:
#                 while not check_circle(gene):
#                     if random.random() < mutation_prob:
#                         gene.x = int(within_limits(gene.x, w+w_margin, -w_margin, w/4))
#                     if random.random() < mutation_prob:
#                         gene.y = int(within_limits(gene.y, h+h_margin, -h_margin, h/4))
#                     if random.random() < mutation_prob:
#                         gene.s = int(within_limits(gene.s, s_max, 0, 10))
#                 if random.random() < mutation_prob:
#                     gene.r = int(within_limits(gene.r, 255, 0, 64))
#                 if random.random() < mutation_prob:
#                     gene.g = int(within_limits(gene.g, 255, 0, 64))
#                 if random.random() < mutation_prob:
#                     gene.b = int(within_limits(gene.b, 255, 0, 64))
#                 if random.random() < mutation_prob:
#                     gene.a = within_limits(gene.a, 1, 0, 0.25)
#     return population
# DONE : mutate only one phenotype, eg color, size, position, etc.

# Main
def main():
    if not os.path.exists(output_path):
        os.mkdir(output_path)
    init_fit = []
    later_fit = []
    start_time = time.time()
    population = init_population()
    best = population[0]
    best_temp = population[0]
    for i in range(num_generations):
        a = 0
        if print_info == True and i%print_intervals == 0:
            print("START")
        for individual in population:
            a += 1
            evaluate_individual(individual)
            if print_info == True and i%print_intervals == 0:
                print("individual:",a,"fitness:",  individual.fitness)

        for individual in population:
            if (individual.fitness > best.fitness) and (individual.fitness < 0):
                best = individual

        if print_info == True and i%(print_intervals/100) == 0:
            print("Generation: ", i, "Best fitness: ", best.fitness)

        if i<10000:
            init_fit.append(best.fitness)
            plt.plot(init_fit)
            # plt.draw()
            plt.title("Fitness Plot from Generation 1 to 10000")
            plt.ylabel('Fitness')
            plt.xlabel('Generation')
            if i%100 == 0 and save == True:
                plt.savefig(output_path+source_image_name[:-4]+"_fitness.png",dpi=200)
            # plt.pause(0.1)
            plt.clf()
        if i>1000 and i<10000:
            later_fit.append(best.fitness)
            plt.plot(later_fit)
            # plt.draw()
            plt.title("Fitness Plot from Generation 1000 to 10000")
            plt.ylabel('Fitness')
            plt.xlabel('Generation')
            if i%100 == 0 and save == True:
                plt.savefig(output_path+source_image_name[:-4]+"_fitness_1000.png",dpi=200)
            # plt.pause(0.1)
            plt.clf()


        if best.fitness > best_temp.fitness:
            best_temp = best
            best.genes.sort(key=lambda x: x.s, reverse=True)
            image = np.zeros([source_image.shape[0],source_image.shape[1],3],dtype=np.uint8)
            image.fill(255)
            for gene in best.genes:
                overlay = image.copy()
                cv2.circle(overlay, (gene.x, gene.y), gene.s, (gene.r, gene.g, gene.b), -1)
                image = cv2.addWeighted(overlay, gene.a, image, 1 - gene.a, 0)
                # cv2.imshow("image", image)
            # cv2.waitKey(1)

        if i%1000 == 999 or i == 0:
            best.genes.sort(key=lambda x: x.s, reverse=True)
            image = np.zeros([source_image.shape[0],source_image.shape[1],3],dtype=np.uint8)
            image.fill(255)
            for gene in best.genes:
                overlay = image.copy()
                cv2.circle(overlay, (gene.x, gene.y), gene.s, (gene.r, gene.g, gene.b), -1)
                image = cv2.addWeighted(overlay, gene.a, image, 1 - gene.a, 0)
            # cv2.waitKey(1)
            if save == True:
                cv2.imwrite(output_path+source_image_name[:-4]+"_gen_"+str(i)+".png", image)
                cv2.imwrite(output_path+source_image_name[:-4]+"_best.png", image)


        # Elites selected, isolated from the population
        elites = elitism(population)

        # Parents selected, isolated from the population
        parents = parent_selection(population)

        # Crossover
        children = crossover(parents)

        # Natural selection
        population = natural_selection(population)

        # Mutation
        children = mutation(children)
        population = mutation(population)

        population = elites + children + population

    end_time = time.time()
    print("Elapsed time: ", end_time - start_time)
    best = population[0]
    for individual in population:
        if individual.fitness > best.fitness:
            best = individual
            print(best.fitness)
    return best




# Run
best_case = main()
# image = np.zeros([source_image.shape[0],source_image.shape[1],3],dtype=np.uint8)
# image.fill(255)
# best_case.genes.sort(key=lambda x: x.s, reverse=True)
# for gene in best_case.genes:
#     overlay = image.copy()
#     cv2.circle(overlay, (gene.x, gene.y), gene.s, (gene.r, gene.g, gene.b), -1)
#     image = cv2.addWeighted(overlay, gene.a, image, 1 - gene.a, 0)
#     cv2.imshow("image", image)
#     cv2.waitKey(100)
# cv2.waitKey(0)
# cv2.destroyAllWindows()
#
#       _____                _____            __  __  __
#      /\    \              /\    \          /\ \/ / /\ \
#     /::\    \            /::\____\        /::\  / /  \ \
#    /::::\    \          /:::/    /       /:/\:\/ /    \ \
#   /::::::\    \        /:::/    /       /:/  \:\_\____\ \
#  /:::/\:::\    \      /:::/    /       /:/    \:\/___/ \
# /:::/  \:::\    \    /:::/    /       /:/     \:\__\   _
# \::/    \:::\    \  /:::/    /       /:/      \/__/  /\ \
#  \/____/ \:::\    \/:::/    /        \/_____/      /::\ \
#           \:::\____\/:::/    /                      /:/\:\ \
#            \::/    /\::/    /                      /:/__\:\ \
#             \/____/  \/____/                       \:\   \:\ \
#                                                      \:\   \:\ \
#                                                       \:\   \:\_\
#                                                        \:\__\::/
#                                                         \/__\:/
```