CENG463 – MACHINE LEARNING
HOMEWORK I

STUDENT INFORMATION

DEPARTMENT: SOFTWARE ENG.
STUDENT ID: 21050911019
NAME: AHMET GÖRKEM
SURNAME: ÇİÇEK

PARAMETERS OF MY SOLUTION	
IMAGE SIZE	150x100
IMAGE CHANNEL	Grayscale
POPULATIONS	50
GENERATION NUMBER FOR SOLUTION	200
PATCH SIZE	8

EXPLANATION OF HOW I CONSTRUCTED THE CROSSOVER, FITNESS, MUTATION, AND SELECTION

Before everything I have resize a bird picture that has name real_image on folder to 150x100 resolution. Then, I was able to divide it to eight pieces.

After I got every pieces of picture I appended them to a list that holds the true locaion of each piece for real image. I setted the list into a numpy array. Unless you did not that you will never ensure the fitness function because it can not get true indexes because of differences of list type(normal&numpy).

```
image = cv2.imread('real_image.jpg', cv2.IMREAD_GRAYSCALE)
h,w=image.shape
print(h,w)
centerX=w//2
centerY=h//4
t1L=image[0:centerY,0:centerX]
t1R=image[0:centerY,centerX:w]
t2L=image[centerY:2*centerY,0:centerX]
t2R=image[centerY:2*centerY,centerX:w]
t3L=image[2*centerY:3*centerY,0:centerX]
t3R=image[2*centerY:3*centerY,centerX:w]
t4L=image[3*centerY:h,0:centerX]
t4R=image[3*centerY:h,centerX:w]


real_values=[t1L,t1R,t2L,t2R,t3L,t3R,t4L,t4R]
np.array(real_values)
```

Above all I set my algorithm parameters on the top of my file as you can see. Genome length equals to the number of pieces of the Picture that we divided before. Population size is the number of pictures that will be generated for each new generations. I setted mutation rate to very low, and crossover rate to medium. You are able to change them to see new results.

```
population_size=50
genome_length=8
mutation_rate=0.01
crosoverrate_=0.5
generations=200
```

On random_genome function I have a numpy array list that hold a new shuffled list from the real_values list. By the way, I got different pictures from each other. I will generate that function on init_population to shuffle real value to get randomized list.

```
def random_genome(length):
    mylist=[]
    np.array(mylist)
    for a in real_values:
        mylist.append(a)
    random.shuffle(mylist)
    return mylist
```

On init_population for each members of population we generates new shuffled list to get random pictures and we return it to access each population.

```
def init_population(population,genome_length):
    mylist=list()
    for i in range(population):
        mylist.append(random_genome(genome_length))
    print(mylist)
    return mylist
```

On fitness function, we check the values of shuffled list that holds actually unique random pieces from real picture, and compare them with values of real_values that holds true locations of each pieces. If each pair holds then we add one to fitness value of that genome. After all, we return the fitness values of each genomes from population (there will be more than one population).

```
def fitness(genome):
    fitnessVar=0
    for a in range(genome_length):
        if (np.array_equal(real_values[a],genome[a])):
            fitnessVar+=1
    return fitnessVar
```

Before crossover, I have a select_parent function. It gets a random number between 0 and sum of each value of total fitness list. By getting each genome and its fitness values, we sum its fitness value with curren variable that holds zero, and we check if that current value is bigger that random number which we got before we return that genome to assign as parent.

```
def select_parent(population,fitness_values):
    total_fitness=sum(fitness_values)
    print("total fitness", total_fitness)
    random_picked_fitness=random.uniform(0,total_fitness)
    print("ra",random_picked_fitness)
    current=0
    for individual, fitness_value in zip(population,fitness_values):
        current+=fitness_value
        if current>random_picked_fitness:
            return individual
```

For crossover function we get a random number between 0.0-1.0. If that number smaller than our crossover rate, we select a random index between one and genome length-1, we get new two pictures that exchanged their pieces from that index.

```
def crossover(parent1,parent2):
    if(random.random()< crosoverrate_):
        crossover_point=random.randint(1,len(parent1)-1)
        return
parent1[:crossover_point]+parent2[crossover_point:],parent2[:crossover_point]+pare
nt1[crossover_point:]
    else:
        return parent1,parent2
```

For mutation, we get an sample piece from real_values list that hold true locations of each pieces, and Exchange it with a piece of our genome with a random probability.

```
def mutate(genome):
    for i in range(genome_length):
        if(random.random()<mutation_rate):
            sampleViewFromRealJPG=random.sample(real_values,1)
            genome[i]=sampleViewFromRealJPG[0]
    return genome
```

On main algorithm, for each generation we generate a new population, we calculate fitness values for each genome in population. Then, for each generation we create new population that is empty. After that, we initialize the crossover and mutate methods. We append the results that we got from crossover and mutation methods into new population.Because of new generation we calculate fitness values again. After all of these, we find the genome that has best fitness and save it to saved_generations folder to inspect sequence of execution. After too many generations, on last generation, we assume that generation will consist the best result. We get the genome which has best fitness value on last generation to show it in a window.

```
def mainAlgorithm():
    population=init_population(population_size,genome_length)
    for generation in range(generations):
        fitness_values=list()
        for genome in population:
            fitness_values.append(fitness(genome))
        newPopulation=list()
        for a in range(population_size//2):
            parent1=select_parent(population,fitness_values)
            parent2=select_parent(population,fitness_values)
            offspring1,offspring2=crossover(parent1,parent2)
            newPopulation.extend([mutate(offspring1),mutate(offspring2)])
        population=newPopulation
        fitness_values=[fitness(genome) for genome in population]
        best_fitness=max(fitness_values)
        print(fitness_values)
        print(f"Generation {generation}: Best fitness ={best_fitness}")
        temp_max=0
        temp_index=0
        for i in range(len(fitness_values)):
            if(fitness_values[i]>=temp_max):
```

```
            temp_max=fitness_values[i]
            temp_index=i
    best_solution=population[temp_index]
    if(generation==generations-1):
        im_h1 = cv2.hconcat([best_solution[0],best_solution[1]])
        im_h2 = cv2.hconcat([best_solution[2],best_solution[3]])
        im_h3 = cv2.hconcat([best_solution[4],best_solution[5]])
        im_h4 = cv2.hconcat([best_solution[6],best_solution[7]])
        imfinal=cv2.vconcat([im_h1,im_h2,im_h3,im_h4])
        s=f"saved_generations/generation{generation}.jpg"
        cv2.imwrite(s, imfinal)
        cv2.imshow("real_image.jpg", image)
        cv2.imshow(s, image)
        cv2.waitKey(0)
    else:
        im_h1 = cv2.hconcat([best_solution[0],best_solution[1]])
        im_h2 = cv2.hconcat([best_solution[2],best_solution[3]])
        im_h3 = cv2.hconcat([best_solution[4],best_solution[5]])
        im_h4 = cv2.hconcat([best_solution[6],best_solution[7]])
        imfinal=cv2.vconcat([im_h1,im_h2,im_h3,im_h4])
        s=f"saved_generations/generation{generation+1}.jpg"
        cv2.imwrite(s, imfinal)
```

Finally, we have to just call mainAlgorithm function.

```
mainAlgorithm()
```

For the saved pictures that has best fitness value from each generation you can check saved generations folder.

Result: