SUBMITTED TO:

DR. DILSHAD SABIR

SUBMITTED BY:

DANISH ZULFIQAR

REGISTRATION NO.:

FA21-BEE-042

SEMESTER-SECTION:

5-B

COURSE:

Artificial Intelligence

EEE 462

Lab Report 06

BS Electrical Engineering

Lab Tasks:

Activity 1:

Code:

```
import random
import math

class ga:
    def __init__(self,individualsize, populationsize):
        self.population = dict()
        self.individualsize = individualsize
        self.populationsize = populationsize
        self.totalfitness = 0
        i = 0
        while i < populationsize:
            listofbits = [0]*individualsize
            listoflocations = list(range(0, individualsize))
            numberofones = random.randint(0, individualsize-1)
            oneslocations = random.sample(listoflocations, numberofones)
            for j in oneslocations:
                listofbits[j] = 1
                self.population[i] = [listofbits, numberofones]
                self.totalfitness = self.totalfitness + numberofones
                i = i+1

    def updatepopulationfitness(self):
        self.totalfitness = 0
```

```
        for indvidual in self.population:
            individualfitness = sum(self.population[indvidual][0])
            self.population[indvidual][1] = individualfitness
            self.totalfitness = self.totalfitness + individualfitness

    def selectparents(self):
        roulettewheel = []
        wheelsize = self.populationsize*5
        h_n = []
        for individual in self.population:
            h_n.append(self.population[individual][1])
        j = 0
        for individual in self.population:
            individuallength = round(wheelsize*(h_n[j]/sum(h_n)))
            j = j+1
            if individuallength > 0:
                i = 0
                while i < individuallength:
                    roulettewheel.append(individual)
                    i = i+1
        random.shuffle(roulettewheel)
        parentindices = []
        i = 0
        while i < self.populationsize:
            parentindices.append(roulettewheel[random.randint(0, len(roulettewheel)-1)])
            i = i+1
        newgeneration = dict()
        i = 0
```

```
        while i < self.populationsize:
            newgeneration[i] = self.population[parentindices[i]].copy()
            i = i+1
        del self.population
        self.population = newgeneration.copy()
        self.updatepopulationfitness()

    def generatechildren(self, crossoverprobability):
        numberofpairs = round(crossoverprobability*self.populationsize/2)
        individualindices = list(range(0, self.populationsize))
        random.shuffle(individualindices)
        i = 0
        j = 0
        while i < numberofpairs:
            crossoverpoint = random.randint(0, self.individualsize-1)
            child1 = self.population[j][0][0:crossoverpoint]\
                    + self.population[j+1][0][crossoverpoint:]
            child2 = self.population[j+1][0][0:crossoverpoint]\
                    + self.population[j][0][crossoverpoint:]
            self.population[j] = [child1, sum(child1)]
            self.population[j+1] = [child2, sum(child2)]
            i = i+1
            j = j+2
        self.updatepopulationfitness()

    def mutatechildren(self, mutationprobability):
        numberofbits = round(mutationprobability*\
                        self.populationsize*self.individualsize)
```

```
        totalindices = list(range(0, self.populationsize*self.individualsize))
        random.shuffle(totalindices)
        swaplocations = random.sample(totalindices, numberofbits)

        for loc in swaplocations:
            individualindex = math.floor(loc/self.individualsize)
            bitindex = math.floor(loc % self.individualsize)

            if self.population[individualindex][0][bitindex] == 0:
                self.population[individualindex][0][bitindex] = 1
            else:
                self.population[individualindex][0][bitindex] = 0
        self.updatepopulationfitness()

individualsize, populationsize = 8, 10
i = 0
instance = ga(individualsize, populationsize)
while True:
    instance.selectparents()
    instance.generatechildren(0.8)
    instance.mutatechildren(0.03)
    print(instance.population)
    print(instance.totalfitness)
    print(i)
    i = i+1
    found = False
    for individual in instance.population:
        if instance.population[individual][1] == individualsize:
```

```
            found = True
            break
    if found:
        break
```

Output:

```
{0: [[0, 1, 1, 0, 0, 0, 0, 0], 2], 1: [[1, 0, 1, 0, 0, 0, 0, 0], 2], 2: [[0, 1, 1, 0, 0, 0, 1, 1], 4], 3: [[0
, 0, 0, 0, 0, 0, 1, 0], 1], 4: [[1, 1, 0, 0, 1, 0, 1, 0], 4], 5: [[0, 1, 1, 0, 0, 0, 0, 0], 2], 6: [[0, 1, 1,
 0, 1, 0, 0, 0], 3], 7: [[1, 0, 0, 0, 0, 0, 1, 0], 2], 8: [[1, 1, 0, 0, 1, 0, 0, 0], 3], 9: [[1, 0, 1, 0, 0,
0, 0, 0], 2]}
25
0
{0: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 1: [[0, 1, 1, 0, 0, 0, 0, 0], 2], 2: [[1, 0, 0, 0, 1, 0, 0, 0], 2], 3: [[1
, 1, 0, 0, 0, 0, 1, 1], 4], 4: [[0, 1, 1, 0, 0, 0, 1, 1], 4], 5: [[0, 1, 1, 0, 0, 0, 1, 1], 4], 6: [[0, 1, 1,
 0, 1, 0, 1, 0], 4], 7: [[0, 0, 0, 0, 0, 0, 0, 0], 0], 8: [[1, 1, 1, 0, 1, 0, 0, 0], 4], 9: [[1, 0, 1, 0, 0,
0, 0, 0], 2]}
31
1
{0: [[1, 0, 0, 0, 1, 0, 0, 0], 2], 1: [[1, 0, 0, 0, 1, 0, 0, 0], 2], 2: [[0, 1, 1, 0, 0, 0, 1, 0], 3], 3: [[1
, 1, 0, 0, 0, 0, 0, 0], 2], 4: [[0, 1, 1, 0, 0, 0, 1, 0], 3], 5: [[1, 0, 0, 0, 1, 0, 0, 1], 3], 6: [[1, 0, 0,
 0, 1, 0, 1, 0], 3], 7: [[0, 1, 1, 0, 0, 0, 0, 0], 2], 8: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 9: [[1, 1, 0, 0, 1,
0, 1, 1], 5]}
30
2
{0: [[1, 0, 0, 0, 1, 0, 1, 0], 3], 1: [[1, 0, 0, 1, 1, 0, 1, 0], 4], 2: [[1, 1, 0, 0, 0, 0, 0, 0], 2], 3: [[1
, 0, 0, 0, 1, 0, 0, 0], 2], 4: [[1, 1, 0, 0, 0, 0, 1, 0], 3], 5: [[0, 1, 1, 0, 1, 0, 0, 0], 3], 6: [[1, 1, 0,
 0, 1, 0, 1, 1], 5], 7: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 8: [[0, 1, 1, 0, 0, 0, 0, 0], 2], 9: [[1, 1, 0, 0, 1,
0, 1, 1], 5]}
34
3
{0: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 1: [[1, 1, 0, 0, 0, 0, 1, 0], 3], 2: [[1, 1, 0, 0, 1, 0, 0, 0], 3], 3: [[1
, 0, 0, 0, 0, 0, 0, 0], 1], 4: [[1, 1, 0, 0, 1, 1, 1, 0], 5], 5: [[1, 0, 0, 0, 1, 0, 1, 1], 4], 6: [[1, 1, 0,
 0, 1, 0, 1, 0], 4], 7: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 8: [[1, 1, 0, 0, 0, 0, 0, 0], 2], 9: [[1, 1, 0, 0, 1,
0, 1, 1], 5]}
37
4
{0: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 1: [[1, 0, 0, 0, 1, 0, 1, 1], 4], 2: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 3: [[1
, 1, 0, 1, 1, 1, 1, 0], 6], 4: [[1, 1, 0, 0, 0, 0, 1, 0], 3], 5: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 6: [[1, 1, 0,
 0, 1, 1, 1, 0], 5], 7: [[1, 1, 0, 0, 1, 0, 1, 0], 4], 8: [[1, 1, 0, 0, 1, 0, 1, 1], 5], 9: [[1, 0, 0, 0, 1,
0, 1, 1], 4]}
```

Activity 2:

Code:

```
import random

import math


class queensGA:

    def updateIndividualFitness(self, individualArray):

        i = 0

        fitnessValue = 0
```

```
        while i < self.individualSize:
            j = 0
            while j < self.individualSize:
                if i != j:
                    if individualArray[j] == individualArray[i]:
                        fitnessValue = fitnessValue + 1
                    elif individualArray[j] == individualArray[i] - abs(j - i):
                        fitnessValue = fitnessValue + 1
                    elif individualArray[j] == individualArray[i] + abs(j - i):
                        fitnessValue = fitnessValue + 1
                j = j + 1
            i = i + 1
        return fitnessValue

    def selectParents(self):
        rouletteWheel = []
        wheelSize = self.populationSize * 5
        h_n = []
        for individual in self.population:
            h_n.append(1 / (self.population[individual][1] + 1e-6))  # Add a small constant like 1e-6
        j = 0
        for individual in self.population:
            individualFitness = round(wheelSize * (h_n[j] / sum(h_n)))
            j = j + 1
            if individualFitness > 0:
                i = 0
                while i < individualFitness:
                    rouletteWheel.append(individual)
```

```
            i = i + 1
    random.shuffle(rouletteWheel)
    parentIndices = []
    i = 0
    while i < self.populationSize:
        parentIndices.append(rouletteWheel[random.randint(0, len(rouletteWheel) - 1)])
        i = i + 1
    newGeneration = dict()
    i = 0
    while i < self.populationSize:
        newGeneration[i] = self.population[parentIndices[i]].copy()
        i = i + 1
    del self.population
    self.population = newGeneration.copy()
    self.updatePopulationFitness()

def updatePopulationFitness(self):
    self.totalFitness = 0
    for individual in self.population:
        individualFitness = self.updateIndividualFitness(self.population[individual][0])
        self.population[individual][1] = individualFitness
        self.totalFitness = self.totalFitness + individualFitness

def __init__(self, individualSize, populationSize):
    self.population = dict()
    self.individualSize = individualSize
    self.populationSize = populationSize
    self.totalFitness = 0
```

```
        i = 0
        while i < populationSize:
            individualArray = [0] * individualSize
            j = 0
            while j < individualSize:
                value = random.randint(0, individualSize - 1)
                individualArray[j] = value
                j = j + 1
            self.population[i] = [individualArray.copy(), 0]
            i = i + 1
        self.updatePopulationFitness()


    def generateChildren(self, crossoverProbability):
        numberOfPairs = round(crossoverProbability * self.populationSize / 2)
        individualIndices = list(range(0, self.populationSize))
        random.shuffle(individualIndices)
        i = 0
        j = 0
        while i < numberOfPairs:
            crossoverPoint = random.randint(0, self.individualSize - 1)
            child1 = self.population[j][0][0:crossoverPoint] + self.population[j +
1]][0][crossoverPoint:]
            child2 = self.population[j + 1][0][0:crossoverPoint] +
self.population[j][0][crossoverPoint:]
            self.population[j] = [child1, 0]
            self.population[j + 1] = [child2, 0]
            i = i + 1
            j = j + 2
        self.updatePopulationFitness()
```

```
    def mutateChildren(self, mutationProbability):
        numberOfBits = round(mutationProbability * self.populationSize * self.individualSize)
        totalIndices = list(range(0, self.populationSize * self.individualSize))
        random.shuffle(totalIndices)
        swapLocations = random.sample(totalIndices, numberOfBits)
        for loc in swapLocations:
            individualIndex = math.floor(loc / self.individualSize)
            bitIndex = math.floor(loc % self.individualSize)
            value = random.randint(0, individualSize - 1)
            while value == self.population[individualIndex][0][bitIndex]:
                value = random.randint(0, individualSize - 1)
            self.population[individualIndex][0][bitIndex] = value
        self.updatePopulationFitness()

individualSize, populationSize = 8, 16
i = 0
instance = queensGA(individualSize, populationSize)
while True:
    instance.selectParents()
    instance.generateChildren(0.5)
    instance.mutateChildren(0.03)
    if i % 20 == 0:
        print(instance.population)
        print(instance.totalFitness)
        print(i)
    i = i + 1
    found = False
```

```
for individual in instance.population:
    if instance.population[individual][1] == 0:
        found = True
        break
    if found:
        print(instance.population)
        print(instance.totalFitness)
        print(i)
        break
```

Output:

```
24
29680
{0: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 1: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 2: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 3: [[3
, 5, 0, 4, 1, 7, 2, 6], 0], 4: [[4, 5, 0, 4, 1, 7, 2, 6], 4], 5: [[3, 5, 0, 4, 1, 3, 2, 6], 6], 6: [[3, 5, 0,
 4, 1, 7, 2, 6], 0], 7: [[3, 4, 2, 4, 1, 7, 2, 6], 8], 8: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 9: [[3, 5, 0, 4, 1,
7, 2, 6], 0], 10: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 11: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 12: [[3, 5, 0, 4, 1, 7, 2
, 6], 0], 13: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 14: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 15: [[3, 5, 0, 4, 1, 7, 2, 6]
, 0]}
18
29700
{0: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 1: [[3, 5, 0, 7, 1, 7, 2, 6], 4], 2: [[3, 5, 0, 4, 1, 7, 7, 6], 6], 3: [[3
, 5, 0, 4, 1, 7, 2, 6], 0], 4: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 5: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 6: [[3, 5, 0,
 4, 1, 7, 2, 6], 0], 7: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 8: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 9: [[3, 5, 0, 4, 1,
7, 2, 6], 0], 10: [[3, 5, 0, 4, 2, 7, 2, 6], 6], 11: [[3, 5, 0, 4, 2, 7, 2, 6], 6], 12: [[3, 5, 0, 4, 1, 7, 2
, 6], 0], 13: [[3, 5, 0, 4, 2, 7, 2, 6], 6], 14: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 15: [[3, 5, 0, 4, 1, 7, 0, 6]
, 4]}
32
29720
{0: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 1: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 2: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 3: [[3
, 5, 0, 4, 1, 7, 2, 6], 0], 4: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 5: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 6: [[3, 5, 0,
 7, 2, 7, 2, 6], 10], 7: [[3, 5, 0, 4, 1, 7, 0, 6], 4], 8: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 9: [[3, 5, 0, 4, 1,
 7, 2, 6], 0], 10: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 11: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 12: [[3, 5, 0, 4, 1, 7,
2, 6], 0], 13: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 14: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 15: [[3, 5, 6, 4, 1, 7, 2, 6
], 6]}
20
29740
{0: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 1: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 2: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 3: [[3
, 5, 0, 4, 1, 7, 2, 6], 0], 4: [[0, 5, 0, 4, 1, 7, 2, 6], 2], 5: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 6: [[6, 5, 0,
 4, 1, 7, 2, 6], 4], 7: [[3, 5, 0, 4, 1, 7, 2, 3], 4], 8: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 9: [[3, 5, 0, 4, 1,
7, 2, 6], 0], 10: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 11: [[3, 5, 0, 4, 1, 7, 2, 6], 0], 12: [[3, 5, 0, 4, 1, 7, 2
, 6], 0], 13: [[3, 4, 0, 4, 1, 7, 2, 6], 6], 14: [[3, 4, 0, 4, 1, 7, 2, 6], 6], 15: [[3, 5, 0, 4, 1, 7, 2, 6]
, 0]}
```

Home Activity:

Activity 1:

Code:

```
import random
import itertools
import math

def generate_points(num_points=15):
    return [(random.uniform(0, 0.5), random.uniform(0, 0.5)) for _ in range(num_points)]

def distance(point1, point2):
    return math.sqrt((point1[0]-point2[0])**2 + (point1[1]-point2[1])**2)

def heuristic(individual):
    return sum(distance(individual[i], individual[i+1]) for i in range(len(individual)-1))

def crossover(parent1, parent2):
    idx = random.randint(0, len(parent1)-2)
    child1 = parent1[:idx] + parent2[idx:]
    child2 = parent2[:idx] + parent1[idx:]
    return child1, child2

def GA():
    points = generate_points()
    population = [random.sample(points, len(points)) for _ in range(45)]

    for generation in range(100):
        population.sort(key=heuristic)
```

```
        population = population[:len(population)//2]

        children = []
        for i in range(0, len(population) - 1, 2):
            child1, child2 = crossover(population[i], population[i+1])
            children.append(child1)
            children.append(child2)

        for child in children:
            if random.uniform(0, 1) < 0.1:
                idx1, idx2 = random.sample(range(len(child)), 2)
                child[idx1], child[idx2] = child[idx2], child[idx1]

        population.extend(children)

    population.sort(key=heuristic)
    return population[0], heuristic(population[0])

solution, cost = GA()
print("Best Route:", solution)
print("Total Distance:", cost)
```

Output:

```
bash-3.2$ python3 Home_Task.py
Best Route: [(0.17184584318275137, 0.48370197144859306), (0.2585973759252741, 0.4606654008545039), (0.3053136
353544071, 0.2641244029733375), (0.24795731439647523, 0.008015342265402348), (0.24795731439647523, 0.00801534
2265402348), (0.24795731439647523, 0.008015342265402348), (0.004143157911437545, 0.041708608270937864), (0.00
4143157911437545, 0.041708608270937864), (0.045790028507780356, 0.08478870666903537), (0.045790028507780356,
0.08478870666903537), (0.0903489654318379, 0.1794743429682858), (0.0903489654318379, 0.1794743429682858), (0.
20391237263842266, 0.17059087581338578), (0.25757434680031627, 0.06343502431206044), (0.25757434680031627, 0.
06343502431206044)]
Total Distance: 1.19867691152687
bash-3.2$
```

Critical Analysis and Conclusion:

While I performed the in-depth exploration of genetic algorithms in this lab, there were some nuances that stirred confusion. For instance, the process of encoding chromosomes into alphabets was introduced rapidly without ample foundational context. I found that selecting survivors based on the fitness function was a crucial component, but its application could have been elucidated more. The reference to Chapter 4 from Norvig's book provided valuable insights, yet integrating more hands-on examples during the session could bridge the gap between theory and application for many like us.