LAB # 6

Statement Purpose:

This lab will introduce students to genetic algorithms. Students will get the opportunity to get into details of genetic concepts of computation including crossover, mutation and survivor selection. This lab will also introduce students into different schemes of chromosome encoding.

Activity Outcomes:

This lab teaches you the following topics:

- How to encode chromosomes into alphabets
- How to select survivors based upon fitness function
- How to find global maxima using genetic algorithms

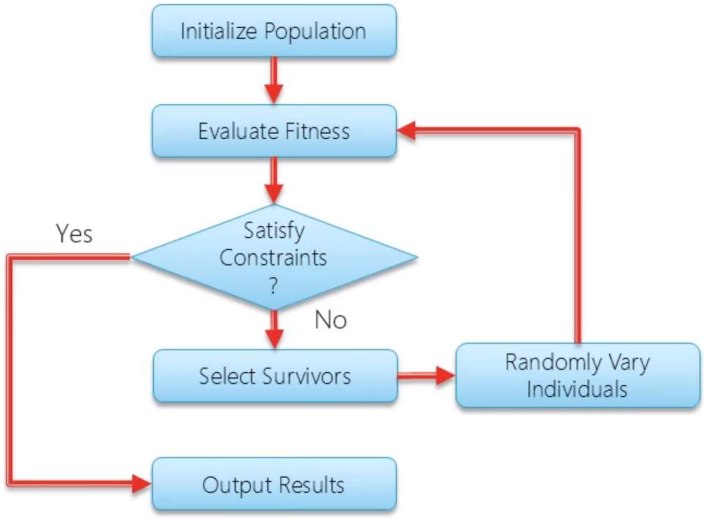
Instructor Note:

As pre-lab activity, read Chapters 4 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of genetic algorithms.

1) Stage J (Journey)

Introduction:

In previous lab we implemented hill climbing and saw that it can stuck at local maxima. A possible solution to avoid local maxima is to use genetic algorithms. A flowchart of generic algorithm is given below,

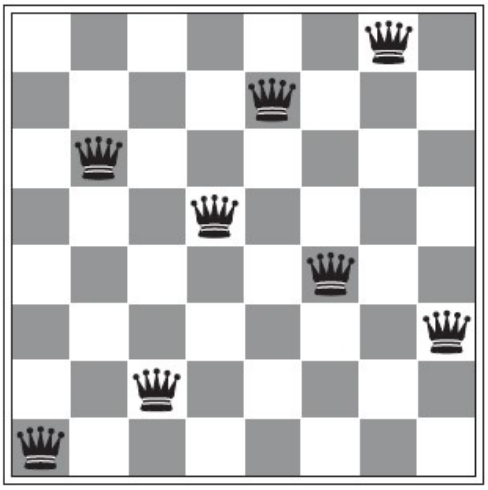


2) Stage a1 (apply)

Lab Activities:

Activity 1:

Imagine an 8 queen problem, where the goal is to place 8 queens on an 8 X 8 board such that no two queens are on the same row or column or diagonal. (Before proceeding, kindly refer to lectures). A sample state is shown below.



Solution:

We start by encoding the state in the form of 8 alphabets. We then define a heuristic function that evaluates how close a given state is with the goal. For this function we sum up number of queens that are in the same row or diagonal within a given queen and then we add up this measure for all 8 queens. What follows is the function definition,

```
import random
import math

class queensGA:

    def updateIndividalFitness(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
```

We now calculate fitness of whole population and after that initialize the population.

```
def updatePopulationFitness(self):
    self.totalFitness = 0
    for individual in self.population:
        individualFitness=self.updateIndividalFitness\
                          (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()
```

We now define the survivor selection method. Here we invert the probability of selection, i.e., the state with lower heuristic value should be given higher probability of selection.

```
def selectParents(self):
    rouletteWheel=[]
    wheelSize=self.populationSize*5
    h_n=[]
    for individual in self.population:
        h_n.append(1/self.population[individual][1])
    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()
```

The children generation is the same as we saw in last activity,

```
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()
```

For mutation, we generate a random number between 0-7 and replace a particular location based upon a mutation probability.

```
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()
```

Finally we define our main function,

```
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
```

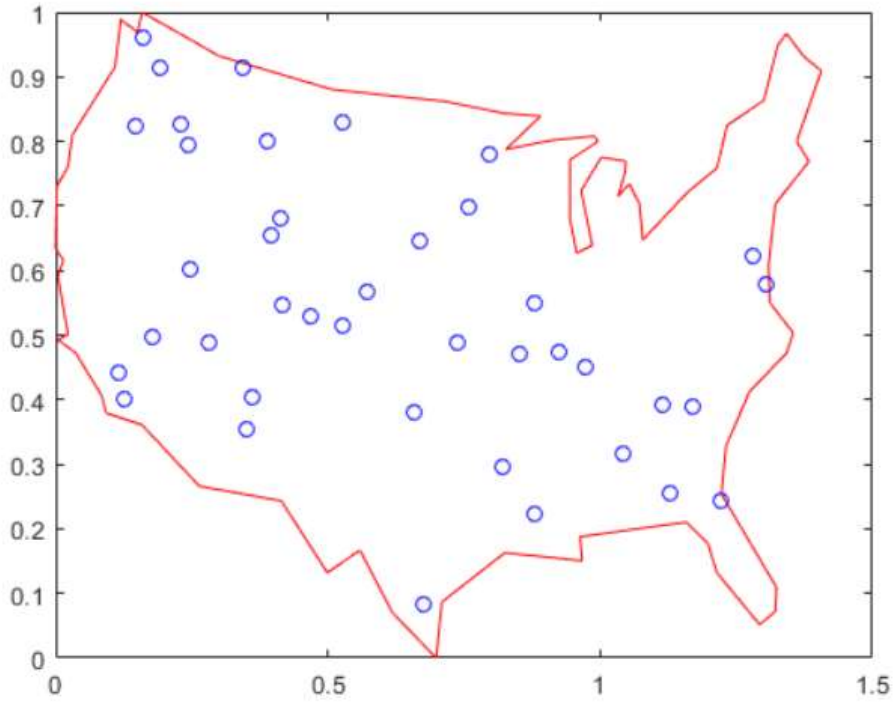
3) Stage v (verify)

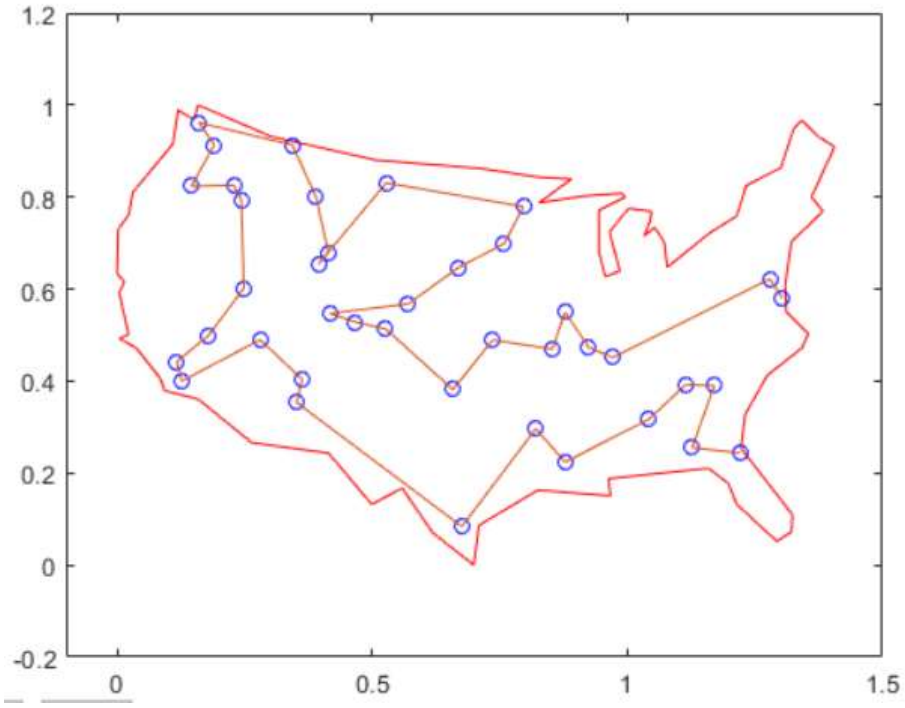
Home Activities:

Activity 1:

The traveling salesman problem is an optimization problem where there is a finite number of cities, and the cost of travel between each city is known. The goal is to find an ordered set of all the cities for the salesman to visit such that the cost or total distance traveled by the salesman is minimized.

The following shoes some cities in US and the route that is optimal for the salesman to follow to minimize the total distance.





Generate 15 random points or cities in python. Each point can be represented by a location tuple (x,y). Assume x ranges from 0-5 and y ranges from 0-5. Select an ordered set of these 15 points e.g., {(x4,y4), (x9,y9),…., 15 points}. This will represent a single

individual/chromosome which basically tells the sales person that he/she should visit 4th city first, then 9th city and so on. The heuristic value of this individual will be the total distance covered by the salesman or the sum of distance between consecutive pair of those 15 points in that individual. Likewise generate a pollution of 45 individuals (possibly by randomly permuting this set). Keep in mind that the salesman is not allowed to visit any city twice i.e., the list must not contain any point multiple times. Use set() function in python for this purpose. See how the total fitness values of the population decreases with each generation. Note that in this case we don't know the minimum heuristic value of the ideal solution. Actually, we don't have an ideal solution beforehand i.e., we don't know how the ideal sequence of cities looks like (unlike other problems where we already know the solution beforehand e.g., how 4 queens should be placed in order to obtain the heuristic value of 0 or how all bits should be 1 to obtain the ideal heuristic value).

4) Stage a2 (assess)

Assignment:

Submit the previous activity within two weeks.

Annex1:
GA class (8 queens)

```
import random
import math

class queensGA:

    def updateIndividalFitness(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 updatePopulationFitness(self):
        self.totalFitness = 0
        for individual in self.population:
            individualFitness=self.updateIndividalFitness\
                              (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 selectParents(self):
        rouletteWheel=[]
        wheelSize=self.populationSize*5
        h_n=[]
        for individual in self.population:
            h_n.append(1/self.population[individual][1])
        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 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
```