ECE 449 - Intelligent Systems Engineering

Lab 5-D42: Genetic Algorithms

Lab date: *Thursday, December 2, 2021 -- 2:00 - 4:50 PM*
Room: *ETLC E5-013*
Lab report due: *Tuesday, December 7, 2021 -- Midnight*

1. Objectives

The objective of this lab is to become familiar with the principles of genetic algorithms (GA), and implement them in some typical applications

2. Expectations

Complete the pre-lab, and hand it in before the lab starts. A formal lab report is required for this lab, which will be the completed version of this notebook. There is a marking guide at the end of the lab manual. If figures are required, label the axes and provide a legend when appropriate. An abstract, introduction, and conclusion are required as well, for which cells are provided at the end of the notebook. The abstract should be a brief description of the topic, the introduction a description of the goals of the lab, and the conclusion a summary of what you learned, what you found difficult, and your own ideas and observations.

3. Pre-lab

1. Describe and compare roulette wheel (fitness proportional) and ranked selection mechanisms.

4. Introduction

A genetic algorithm is a type of evolutionary algorithm based on the concept *evolution*. The essence of evolution are variation and selection, that can be seen as a series of mapping functions connecting genetics and behavior. Selection and mutation are two of those functions. A common application is optimization problems that involve many variables acting simultaneously.

Unlike classical search and optimization methods, a GA begins its search with an initial set of randomly generated candidate solutions to the problem, referred to as *individuals* in a *population* of solutions, where each individual competes for survival. Also, GA utilizes random variation to find new solutions.

Typically, an individual is represented by binary strings, but other encodings can be used (e.g. integers or real numbers). The method of representation scheme has a major impact on the performance of the GA, as different schemes may cause different performance in terms of accuracy and computation time.

Once a random population is **initialized**, each individual is **evaluated and assigned a fitness value**, according to a fitness function defined by the user. This marks the completion of a generation worth of individuals, leading to ***crossover* and *mutation*** of individuals in the current generation. These operations are performed only on selected members of the population, *parents*, typically based on fitness. Crossover is analogous to reproduction, and involves the mixing of two parents' genetic information. Mutation consists of changing an individual's representation (e.g. flipping 0 to 1). Both operations are used to introduce new genetic information into the population such that other solutions are explored, and the algorithm does not settle with a local minimum/maximum. The modified individuals are then **evaluated**, the low-fit individuals are **discarded**. This procedure is repeated until the maximum number of generations is reached, or a stopping criteria is met, and the best fit individual is chosen as the solution.

 alt text

5. Experimental Procedure

If you have not yet installed the pyeasy library, run the cell below.

In [1]:
```
%%bash
# "--user" is essential to install in local environment"
pip install --user -U pyeasyga
```

```
Collecting pyeasyga
  Downloading pyeasyga-0.3.1.tar.gz (20 kB)
Requirement already satisfied: six in /opt/conda/lib/python3.9/site-package
s (from pyeasyga) (1.16.0)
Building wheels for collected packages: pyeasyga
  Building wheel for pyeasyga (setup.py): started
  Building wheel for pyeasyga (setup.py): finished with status 'done'
  Created wheel for pyeasyga: filename=pyeasyga-0.3.1-py2.py3-none-any.whl
size=6804 sha256=f6bfa67f8ff912f6e72fa03d0e70075402aa4bb6fec4db55834c3b9698
f69650
  Stored in directory: /home/jupyter/.cache/pip/wheels/4d/b5/ac/484fc9ef665
b0699a2d0e64d5075282b73d3ae73e5ac6c3c98
Successfully built pyeasyga
Installing collected packages: pyeasyga
Successfully installed pyeasyga-0.3.1
```

Run the cell below to import the libraries required to complete this lab.

In [2]:
```
import numpy as np              # General math operations
import matplotlib.pyplot as plt # Data visualization
from pyeasyga import pyeasyga   # Genetic algorithms
import random                   # RNG for GA implementation
```

Exercise 1: Mathematical genetic algorithm

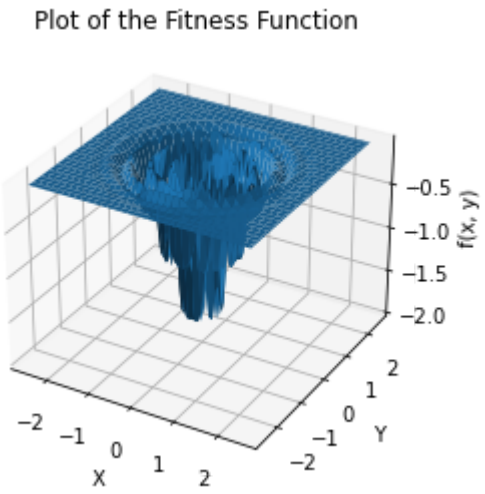
Create a simple genetic algorithm to determine the global minimum of the function:

$$f(x, y) = -[1 + cos(15r)]e^{-r^2}, \text{ where } r = \sqrt{x^2 + y^2}$$

The cell below plots the fitness function to illustrate that there are several local minima, and so traditional gradient descent algorithms could easily get stuck in one of these trenches.

In [3]:
```
import matplotlib.pyplot as plt             # Data visualization
from mpl_toolkits.mplot3d import Axes3D     # 3D data visualization

x = np.linspace(-2.5, 2.5, num = 101)
y = np.linspace(-2.5, 2.5, num = 101)
[gX, gY] = np.meshgrid(x, y)
fcn = -(1+np.cos(15*np.sqrt(gX**2 + gY**2))) * np.exp(-gX**2 - gY**2)
fig = plt.figure()
ax = fig.gca(projection = '3d')
ax.plot_surface(gX, gY, fcn)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('f(x, y)')
ax.set_title('Plot of the Fitness Function')
plt.show()
```

```
/tmp/ipykernel_76/3339539888.py:9: MatplotlibDeprecationWarning: Calling gc
a() with keyword arguments was deprecated in Matplotlib 3.4. Starting two m
inor releases later, gca() will take no keyword arguments. The gca() functi
on should only be used to get the current axes, or if no axes exist, create
new axes with default keyword arguments. To create a new axes with non-defa
ult arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection = '3d')
```



The execution of any GA requires the definition of multiple functions: *create_individual*, *selection*, *crossover*, and *mutate*. The cell below provides the individual creation, mutation, and crossover functions to be used, but the fitness function, along with the GA creation, needs to be programmed. The GA should be initially built with the following parameters:

- Population size: 30
- Generations: 50
- Crossover probability: 0.8
- Mutation probability: 0.005
- Selection: tournament

As for the fitness function, look at the *create_individual* function and understand how each individual is represented. In addition, think about whether you wish to maximize or minimize the fitness value in your function, and program the GA according to this answer. It is worth noting that there is no need for input data in this application, so you can initialize the GA with an arbitrary variable.

1. Complete the GA, according to the above parameters, and run the GA a few times to confirm your results. What was the best fitness value and solution that the GA found?

In [30]:
```
def create_individual(data):
    """ Create a candidate solution representation
        Represented as an array of x and y floating-point values from -10 to
    """
    individual = np.zeros((2,))
    individual[0] = random.uniform(-10, 10)  # X value
    individual[1] = random.uniform(-10, 10)  # Y value
    return individual

def crossover(parent_1, parent_2):
    """ Crossover two parents to produce two children
        Performs a weighted arithmetic recombination
    """
    ratio = random.uniform(-1, 1)  # Generate a number from -1 to 1
    crossIndices = np.random.choice([0, 1], size=(len(parent_1),), p=[0.5, 0.
    child_1 = parent_1
    child_2 = parent_2
    for i in range(len(crossIndices)):
        if (crossIndices[i] == 1):
            child_1[i] = child_1[i] + ratio * child_2[i]  # Perform weighted
            child_2[i] = child_2[i] + ratio * child_1[i]
    return child_1, child_2

def mutate(individual):
    """ Mutate an individual to introduce new genetic information to the popu
        Adds a random number from 0 to 9 to each allele in the individual (up
    """
    mutateIndices = np.random.choice([0, 1], size=(4,), p=[0.8, 0.2])
    for index in range(len(mutateIndices)):
        if(mutateIndices[index] == 1):
            individual[0] += random.randint(1, 9) * (10**(index - 2))
            individual[1] += random.randint(1, 9) * (10**(index - 2))

def fitness(individual, data):
    """ Calculate fitness of a candidate solution representation
    """
    x = individual[0]   #Since the index zero of individual is the x coordina
    y = individual[1]   #Since the index one of individual is the x coordinat
    r = np.sqrt((x**2) + (y**2))
    fitness = - ((1+ np.cos(15*r))*(np.exp(-(r**2))))
    return fitness
```

```
The best fitness value that the GA found was -2.0 and the best solution was
(-3.30709058e-10, -3.84317199e-10).
```

In [50]:
```
inputData = np.random.rand(10,10)

ga = pyeasyga.GeneticAlgorithm(
inputData,
population_size = 30,
generations = 50,
crossover_probability = 0.8,
mutation_probability = 0.005,
maximise_fitness = False  # Minimizes fitness if False
)


ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
ga.selection_function = ga.tournament_selection # pyeasyga's implemented sele
ga.fitness_function = fitness

ga.run()
bestSoln = ga.best_individual()
print("Fitness = ", bestSoln[0])
print("Solution = ", bestSoln[1])
```

```
Fitness =  -2.0
Solution =  [-3.30709058e-10 -3.84317199e-10]
```

2. Change both the crossover probability and mutation probability to 0, and run the GA a few times again. Comment on how this affects the results, and provide a possible explanation as to why this GA setup does not return the optimal solution.

In [56]:
```
inputData = np.random.rand(10,10)

ga = pyeasyga.GeneticAlgorithm(
inputData,
population_size = 30,
generations = 50,
crossover_probability = 0,   #Crossover probability changed to 0
mutation_probability = 0,    #Mutation probability changed to 0
maximise_fitness = False  # Minimizes fitness if False
)


ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
ga.selection_function = ga.tournament_selection # pyeasyga's implemented sele
ga.fitness_function = fitness

ga.run()
bestSoln = ga.best_individual()
print("Fitness = ", bestSoln[0])
print("Solution = ", bestSoln[1])
```

```
Fitness =  -0.1483960705753585
Solution =  [0.23614729 0.07375762]
```

As can be seen above, when the crossover and mutation probability is changed to 0, the fitness value obtained is -0.148 and the best solution is [0.23614729 0.07375762]. When the probability for both is changed to 0, this implies that there are no variation operators in the GA and thus no variation would be carried out. This means that there won't be any further exploration in the search space and a whole new generation will be made from exact copies of chromosomes from old population due to lack of diversity. However,this does not mean that the new generation is the same.
Hence, the GA does not return optimal solutions as there is no variation in the GA because evolution by selection cannot occur if there is no variation within a species as variation within the solution candidates causes better adaptation to new environments, i.e., better fitness. Thus, if there is no variation we can't aim for better fitness or an optimal solution.

3. Perform the previous task again, except with the crossover and mutation probability changed to 1.

In [62]:
```
inputData = np.random.rand(10,10)

ga = pyeasyga.GeneticAlgorithm(
inputData,
population_size = 30,
generations = 50,
crossover_probability = 1, #Crossover probability changed to 1
mutation_probability = 1,  #Mutation probability changed to 1
maximise_fitness = False  # Minimizes fitness if False
)


ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
ga.selection_function = ga.tournament_selection # pyeasyga's implemented sele
ga.fitness_function = fitness

ga.run()
bestSoln = ga.best_individual()
print("Fitness = ", bestSoln[0])
print("Solution = ", bestSoln[1])
```

```
Fitness =  -1.999689842562077
Solution =  [ 0.00144702 -0.00078426]
```

As can be seen above, when the crossover and mutation probability is changed to 1, the fitness value obtained is -1.999.. and the best solution is (0.00144702, -0.00078426). If crossover probability is 1 or 100%, then this implies that all the offsprings will be made by crossover. Similarly, if the mutation probability is 1 or 100%, then this implies that all the offsprings will be made by mutation. In our case, both mutation and crossover probability is changed to 1. In a case when both mutation and crossover probability is being implied in variation their sum should equal to 1, in this case that is not true.
This implies that the variation operator will be unguided in this case as there is no balance between crossover and mutation. Keeping a balance between crossover and mutation rates is an issue of controlling parameters in GA. When there is no balance, there will be excessive variation which means that there will be random searching/exploration within the search space of candidate solutions. Thus, in this case the GA will perform like a blind random search and so, we wouldn't have a gaurantee of obtaining optimal solutions.

```
<h4>Exercise 2:   WSN genetic algorithm</h4>
```

A team of climatologists is trying to optimize the energy usage of their wireless sensor network (WSN) of weather monitoring stations. Their current setup involves all stations sending their data directly to the base station (BS). However, they would like to explore the option of assigning cluster heads (CH) to some of these stations. These CH's would collect the data from nearby regular stations (RS) and send it to the BS such that not every station has to communicate with the BS, thereby optimizing the total communication distance of the network. In order to determine the optimal setup of both number and location of CH's, they are designing a genetic algorithm with the following parameters:

- Individual representation: binary string
- Population size: 80
- Generations: 100
- Crossover: one-point with probability 0.7
- Mutation: bitwise with probability 0.05
- Selection: tournament

The GA has been built to solve the WSN routing optimization problem in the cell below. The base station is located in the centre of a (250, 250) map, with 80 stations assigned randomly around it.

An individual is represented as a binary string, with length equal to the number of stations in the network. A 0 represents a regular station, whereas a 1 represents a cluster head. Parents are selected using tournament selection, in which individuals are randomly chosen to be compared to another individual, and the most fit individual "wins". For crossover, a single-point method is used, where a point is chosen in a parent's string, and the genetic information is swapped with another parent starting at that point. Finally, the mutation function flips on average two bits of a chosen parent anywhere in its string.

The GA aims to maximize a fitness function based on the communication distance difference between the previous method (all stations to BS) and the new method (RS to CH, and CH to BS), as well as the difference between the total number of stations and the number of CH's used.

Once it has undergone the specified number of generations, the GA determines what the best solution is, and the optimal routing scheme is displayed.

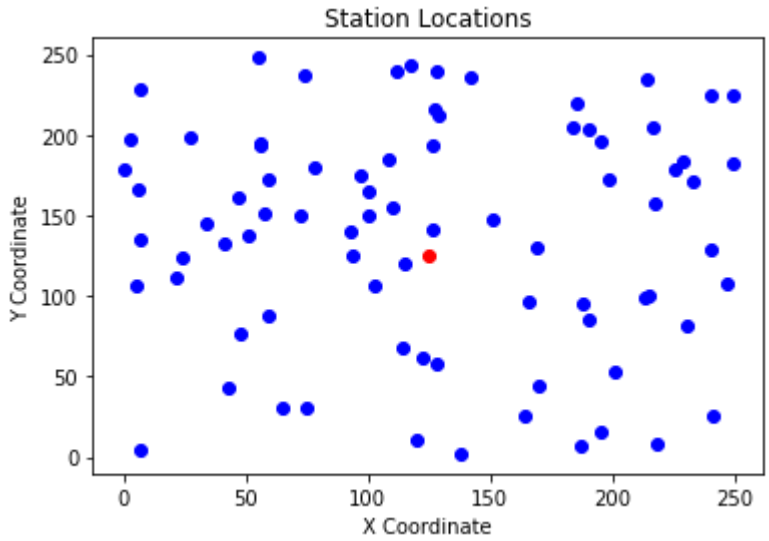
1. Run the script to call the GA and determine the optimal clustering and routing for the WSN. Include the plot of the final results in your report.

In [4]:

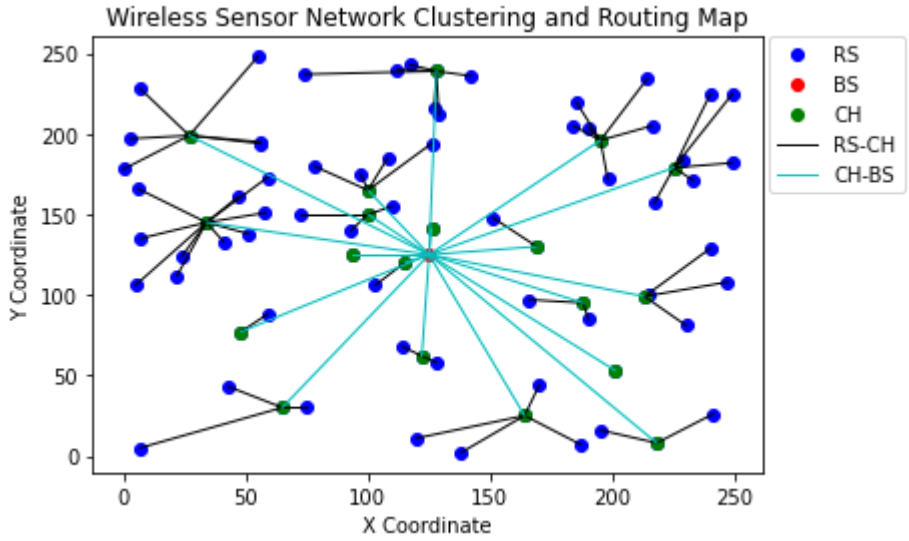
```
def create_individual(data):
    """ Create a candidate solution representation
        0 = regular station; 1 = cluster head
        Represented as a binary sequence with ~25% 1's
    """
    individual = np.random.choice([0, 1], size = (len(data),), p = [0.75, 0.2
    return individual

def crossover(parent_1, parent_2):
    """ Crossover two parents to produce two children
        Implements single point crossover
    """
    index = random.randrange(1, len(parent_1))
    child_1 = np.append(parent_1[:index], parent_2[index:])
    child_2 = np.append(parent_2[:index], parent_1[index:])
    return child_1, child_2

def mutate(individual):
    """ Mutate an individual to introduce new genetic information to the popu
        Flips on average 2 bits in the individual
    """
    noStations = len(individual)
    for i in range(noStations):
        if (random.randint(0, noStations) % noStations == 0):  # ~2 bits muta
            individual[i] ^= 1  # Swap the current bit using XOR operator

def fitness(individual, data):
    """ Calculate fitness of a candidate solution representation
        Based on the difference between no clustering and clustering
    """
    totDist = np.sum(data[:, 2])  # Total distance of all stations to the bas
    noStations = len(individual)  # Total number of stations [N]
    noCH = np.sum(individual)  # Total number of cluster heads (CH) [H_i]

    # If no CH's are assigned, return a fitness value of 0
    if (noCH == 0):
        fitness = 0
        return fitness

    chIndices = np.transpose(np.nonzero(individual))  # Find indices of the s
    minDist = np.zeros((noStations, 1))
    chBSDist = np.zeros((noCH, 1))

    # Get distance of each CH to the BS
    for k in range(noCH):
        chBSDist[k] = data[chIndices[k], 2]

    # Calculate the distance between each station and the CH's to determine t
    temp = np.zeros((noCH, 1))
    for i in range(noStations):
        for j in range(noCH):
            temp[j] = distMap[i, chIndices[j]]  # Store distance between a st
        minDist[i] = np.amin(temp)  # Determine the closest CH to the current

    newDist = np.sum(chBSDist) + np.sum(minDist)  # Sum of distances from sta
```

```
    fitness = (totDist-newDist) + (noStations-noCH)  # Fitness value to be ma
    return fitness

def mapRoute(individual, data):
    """ Displays the routed results given an individual and the input data
    """
    chIndices = np.transpose(np.nonzero(individual))  # Find indices of the s
    noCH = len(chIndices)
    noStations = len(individual)

    stationConnectivity = np.zeros((noStations+1, noStations+1))  # 0 = not c

    # Determine station-CH connectivity
    temp = np.zeros((noCH, 1))
    for i in range(noStations):
        for j in range(noCH):
            temp[j] = distMap[i, chIndices[j]]  # Store distance between a st
        if (np.amin(temp) == 0):  # Ignore if the current station is a CH
            continue
        chIndex = chIndices[np.argmin(temp)]
        stationConnectivity[i,chIndex] = 1
        stationConnectivity[chIndex, i] = 1

    # Begin plotting the data
    fig, ax = plt.subplots()
    stationHandle ,= plt.plot(data[:, 0], data[:, 1], 'bo', label = 'RS')  #
    bsHandle ,= plt.plot(bsCoords[0], bsCoords[1], 'ro', label = 'BS')  # Bas

    # Determine CH -> BS connectivity
    for k in range(noCH):
        chIndex = chIndices[k]
        stationConnectivity[chIndex, -1] = 2
        stationConnectivity[-1, chIndex] = 2
        chHandle ,= plt.plot(data[chIndex, 0], data[chIndex, 1], 'go', label

    # Plot station connections
    for i in range(1, len(stationConnectivity)):
        for j in range(i):
            if (stationConnectivity[i, j] == 1):
                rsCHHandle ,= plt.plot([data[i, 0], data[j, 0]], [data[i, 1],
            if (stationConnectivity[i, j] == 2):
                chBSHandle ,= plt.plot([bsCoords[0], data[j, 0]], [bsCoords[1
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title('Wireless Sensor Network Clustering and Routing Map')
    plt.legend(bbox_to_anchor = (1.01, 1), loc = 2, borderaxespad = 0., handl
    plt.show()


noStations = 80  # Number of stations to route

stationInfo = np.zeros((noStations, 3))  # [x, y, dist to BS]
distMap = np.zeros((noStations, noStations))  # Distance between stations
bsCoords = np.array([125, 125])  # Base station coordinates at (125, 125)

random.seed(1)  # Set seed for consistent coordinates
```

```
# Assign random station coordinates and calculate the distances between stati
for i in range(len(stationInfo)):
    stationInfo[i, 0] = random.randint(0, 250)
    stationInfo[i, 1] = random.randint(0, 250)
    stationInfo[i, 2] = np.linalg.norm(stationInfo[i, 0:2] - bsCoords)

    for j in range(i + 1):
        distMap[i, j] = np.linalg.norm(stationInfo[i, 0:2] - stationInfo[j, 0
        distMap[j, i] = distMap[i, j]

# Show location of stations
stationPlt = plt.scatter(stationInfo[:, 0], stationInfo[:, 1], c = 'b')
bsPlt = plt.scatter(bsCoords[0], bsCoords[1], c = 'r')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.title('Station Locations')
plt.show()

# Create the GA
ga = pyeasyga.GeneticAlgorithm(stationInfo,  # Input data
                               population_size = 80,
                               generations = 100,
                               crossover_probability = 0.7,
                               mutation_probability = 0.005,
                               maximise_fitness = True
                               )

# Set the appropriate parameters for the GA
ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
ga.selection_function = ga.tournament_selection
ga.fitness_function = fitness

# Run the network and print the best individual
ga.run()
bestSoln = ga.best_individual()
print("Fitness = ", bestSoln[0])
print("Solution = ", bestSoln[1])

# Display routing results
mapRoute(ga.best_individual()[1], stationInfo)
```




```
Fitness =  4668.523516020534
Solution =  [1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0
 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0 1 1 0 0 1 0 0 0 0
 0 1 0 0 0 0]
```



2. Summarize the pros and cons of using a GA as a problem solving technique.

Your summary goes here

A GA is employed in several areas such as problem solving techniques, gaming, scheduling problems etc. However, a GA has both pros and cons to its implementation. The pros of using a GA as a problem solving technique are that a GA implements parallelism, thus it is easy to modify and adapt for different problems. A GA also provides a large and wide solution space search ability and uses a non-knowledge based optimisation process of a fitness function for evaluation. By using a GA, it is easy to discover global optimum and we can avoid getting trapped in a local optima.Lastly, a GA can optimize various problems such as discrete functions, multi-objective problems, and continuous functions.
However, a GA has some cons attached to it as well. The optimization time of using a GA could be very high. There could be a very slow convergence when using a GA beacause of an unguided mutation, i.e., a random addition of an individual parameter to the population.Since a GA uses random variation, we must define appropriate crossover and mutation operations to obtain efficient results. If the variation operators are inaccurately defined, the GA can potentially behave like a blind random search. The method of representation scheme has a major impact on the performance of a GA, as different schemes may cause different performance in terms of accuracy and computation time, thus, any mistake in defining the representation scheme can afect the results. Finally, there is no gaurantee of obtaining an optimal solution when using a GA unlike an exhaustive search.

Abstract

Genetic algorithms apply the ideas of genetic evolution to solving optimization problems. Selection and variation are considered as the essence of evolution. A GA is different from classical search and optimization methods. It works in the following stages:

1. Chromosomal Representation - Typically, an individual is represented by binary strings, but other encodings can be used like integers or real numbers.

2. Initial Population - Once a suitable representation has been found,an initial population is randomly created to serve as the starting point for the genetic algorithm.

3.Fitness Evaluation- Once a random population is initialized, each individual is evaluated and assigned a fitness value, according to a fitness function defined by the user and as the GA proceeds we could expect that the individual fitness of the best chromosome will increase along with the total fitness of the population as a whole.

4. Selection- Members of the population i.e., parents are selected at this stage, typically based on fitness.The higher the fitness value the higher the probability of that chromosome/parent being selected for reproduction.

5. Crossover and Mutation - Crossover and Mutation are then performed on the selected individuals/chromosomes.A crossover operation involves the mixing of two parents' genetic information while mutation consists of changing an individual's representation like, flipping 0 to 1. It is essential that both of these operations are used, so that we can introduce new genetic information into the population so that we are not trapped at a local minimum/maximum.

Finally, the modified individuals are evaluated and the low-fit individuals are discarded.

This whole procedure is repeated until either the maximum number of generations has been reached or a stopping criteria is met.Then,the best fit individual is chosen as the solution.

Introduction

In this lab, we became familiar with the principles of genetic algorithms - GA and implemented them in some typical applications. Firstly, we created a simple genetic algorithm to determine the global minimum of the function: $f(x,y)$=-[1+cos(15r)]e^(-r^2) , where r= sqrt(x^2 + y^2). This GA was created by first defining multiple functions like - create_individual, selection, crossover, and mutate and fitness function. The GA was initially built with the parameters, population size =30, generations = 50, crossover probability = 0.8, mutation probability = 0.005 and tournament selection.We minimized the fitness function and ran the GA to find the best fitness value and solution. We also analyzed how the fitness value and solution change when both the crossover and mutation probabilities are changed to 0 and 1.

In the second exercise, we built a GA to solve the WSN routing optimization problem using the following parameters: individual representation = binary string, population size = 80, generations = 100, crossover = one-point with probability 0.7, mutation = bitwise with probability 0.05 and tournament selection. Our GA aimed at maximizing the fitness function based on the communication distance difference between the previous method (all stations to BS) and the new method (RS to CH, and CH to BS), as well as the difference between the total number of stations and the number of CH's used. After this, the GA determined the the best solution and dsiplayed the optimal routing scheme.

Conclusion

The objective of this lab was to familiarize ourselves with the principles of genetic algorithms. Genetic algorithms apply the ideas of genetic evolution to solve optimization problems.They are different from classical search and optimization methods. A GA works in the following stages - representation, initializing the population, fitness evaluation & selection, and corssover and mutation. Finally, the modified individuals are evaluated and the low-fit individuals are discarded and the whole procedure is repeated until a stopping condition is met.

In the first exercise, we created a simple genetic algorithm to determine the global minimum of the function:
$f(x,y)$=-[1+cos(15r)]e^(-r^2) , where r= sqrt(x^2 + y^2). This GA was created by first defining multiple functions like - create_individual, selection, crossover, and mutate and fitness function. The GA was initially built with the parameters, population size =30, generations = 50, crossover probability = 0.8, mutation probability = 0.005 and tournament selection. We minimized the fitness function and ran the GA to find the best fitness value and solution. The best fitness value was found to be -2.0 and the solution as (-3.30709058e-10, -3.84317199e-10). We also analyzed how the fitness value and solution change when both the crossover and mutation probabilities are changed to 0 and 1.

When the crossover and mutation probability is changed to 0, the fitness value obtained is -0.148 and the best solution is [0.23614729 0.07375762]. There are no variation operators in the GA when the probabilities are changed to 0 and thus, there won't be any further exploration in the search space and a whole new generation will be made from exact copies of chromosomes from old population due to lack of diversity.Thus, if there is no variation we can't aim for better fitness or an optimal solution.

Also, when the crossover and mutation probability is changed to 1, the fitness value obtained is -1.999.. and the best solution is (0.00144702, -0.00078426). In this case, the variation operator will be unguided as there is no balance between crossover and mutation.So, there will be excessive variation which means that there will be random searching/exploration within the search space of candidate solutions. Thus, in this case the GA will perform like a blind random search and so, we wouldn't have a gaurantee of obtaining optimal solutions.

We also built a GA to solve the WSN routing optimization problem where our GA aimed at maximizing the fitness function based on the communication distance difference between the previous method (all stations to BS) and the new method (RS to CH, and CH to BS), as well as the difference between the total number of stations and the number of CH's used. After this, the GA determined the the best solution and dsiplayed the optimal routing scheme.

We also observed the pros and cons of using a GA as a problem solving technique. Certain pros of using a GA that were identified are parallelism, a large and wide solution space search ability provided by a GA and its use of a non-knowledge based optimisation process of a fitness function for evaluation. While using a GA, it is easier to discover global optimum and we can avoid getting trapped in a local optima. However, certain cons of using a GA were also identified, like the optimization time of using a GA could be very high and there could be a very slow convergence when using a GA beacause of an unguided mutation. Also, appropriate crossover and mutation operations must be defined to obtain efficient results as a GA uses random variation.

Lab 5 Marking Guide

Exercise	Item	Total Marks	Earned Marks
	$Pre-lab$	10	
	$Abstract$	3	
	$Introduction$	3	
	$Conclusion$	4	
1	$Mathematical\ geneticalgorithm$	50	
2	$WSN\ geneticalgorithm$	30	
	TOTAL	100	