ECE 449 - Intelligent Systems Engineering

Lab 5: Genetic Algorithms

Lab date: *Thursday, November 21, 2019 -- 2:00 - 4:50 PM*
Room: *ETLC E5-013*
Lab report due: *Friday, November 29, 2019 -- 3:50 PM*

---

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 an approach to machine learning that mimics evolution. 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*. 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. 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.

5. Experimental Procedure

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

In [3]:
```
import sys
!{sys.executable} -m pip install pyeasyga
```

```
Collecting pyeasyga
Requirement already satisfied: six in /opt/conda/lib/python3.6/site-packag
es (from pyeasyga) (1.12.0)
Installing collected packages: pyeasyga
Successfully installed pyeasyga-0.3.1
```

In [4]:
```
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
```

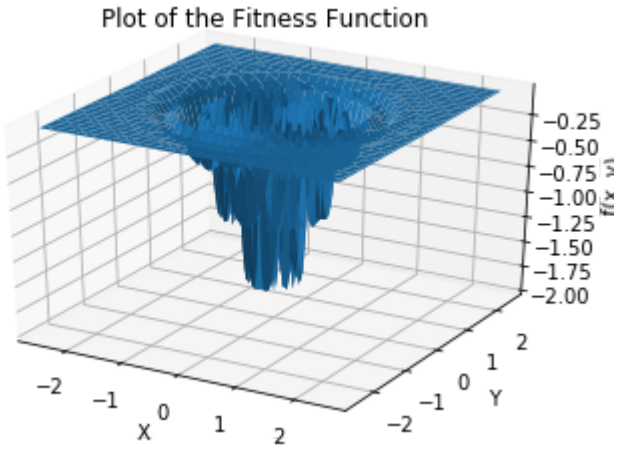
Exercise 1: Mathematical genetic algorithm

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 [5]:
```
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()
```



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 [6]:

```
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, (
    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 weighte
            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 pop
        Adds a random number from 0 to 9 to each allele in the individual (u
    """
    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]
    y = individual[1]
    return -(1+np.cos(15*np.sqrt(x**2 + y**2))) * np.exp(-x**2 - y**2)
```

Write your answer to question 1 below:

In [7]:
```
def run_ga(g):
    for i in range(10):
        ga.run()
        best_solution = ga.best_individual()
        print("Fitness = {}".format(best_solution[0]).ljust(40), "Solution =

# default is tournament selection
ga = pyeasyga.GeneticAlgorithm(
None,
population_size = 30,
generations = 50,
crossover_probability = 0.8,
mutation_probability = 0.005,
maximise_fitness = False # Minimizes fitness if False, max if it's true
)

ga.create_individual = create_individual
ga.crossover_function = crossover
ga.mutate_function = mutate
# ga.selection_function = ga.tournament_selection
ga.fitness_function=fitness

run_ga(ga)
```

```
Fitness = -1.9999999999987077            Solution = [ 5.45362557e-08 -9.1
1701243e-08]
Fitness = -2.0                           Solution = [-1.01419615e-09 -2.7
5699545e-10]
Fitness = -1.9999999986200392            Solution = [ 1.99567968e-06 -2.8
4065553e-06]
Fitness = -2.0                           Solution = [-7.23479956e-10 -1.7
9357547e-10]
Fitness = -2.0                           Solution = [-5.03316652e-10 -6.1
0937695e-11]
Fitness = -2.0                           Solution = [1.36678252e-10 5.378
72515e-10]
Fitness = -2.0                           Solution = [ 2.12245750e-10 -4.7
8218467e-11]
Fitness = -1.9999999999998856            Solution = [-1.83941042e-09  3.1
5713818e-08]
Fitness = -1.9999999927022927            Solution = [3.67455823e-07 7.974
98665e-06]
Fitness = -2.0                           Solution = [-9.05618098e-10 -1.0
8671485e-10]
```

What was the best fitness value and solution that the GA found?

The best solution the GA found had a fitness value around -2, and the solution it found is (x,y) being close to (0, 0)

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 [9]:
```
ga.mutation_probability = 0
ga.crossover_probability = 0

run_ga(ga)
```

```
Fitness = -0.003708779158596691              Solution = [-2.47530363 -0.4
0163158]
Fitness = -0.00022001740184833066            Solution = [-2.5688209  -1.5
2712927]
Fitness = -0.011812661871050664              Solution = [ 1.79895714 -1.2
2376426]
Fitness = -1.4192252874181064                  Solution = [0.18593113 0.4
2701173]
Fitness = -0.0052408051165751994             Solution = [-0.95926918 -0.3
8999325]
Fitness = -0.025504514073937806              Solution = [-0.81942695  1.8
5778299]
Fitness = -0.12247270198538122               Solution = [-1.66520129 -0.1
3748905]
Fitness = -0.06281757401112374               Solution = [-0.92776545  1.4
8989437]
Fitness = -0.7318384005420594                  Solution = [0.68717206 0.3
0696397]
Fitness = -0.0004271715373357015               Solution = [2.0683121  2.0
1340623]
```

```
Changing the crossover and mutation probability to 0 means that variation
is no longer introduced in the GA.
We see that the results obtained after changing these values are not as
good as the first run above where these values were nonzero.

Because no variation is introduced over the generations, the fitness of the
individuals depends on where they originally started, since the GA will
not explore new solutions. (Note: this does not mean that the new
generations are exactly the same as the old generations, but they are made
from exact copies of the chromosomes of the old population.
```

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

In [11]:
```
ga.mutation_probability = 1
ga.crossover_probability = 1

run_ga(ga)
```

```
Fitness = -1.9725915470737594                Solution = [ 0.01532121 -0.0
0239924]
Fitness = -1.99999927166391               Solution = [ 7.63979912e-05 -2.2
8989435e-05]
Fitness = -1.9997847685944539                Solution = [-0.00133712  0.0
0030321]
Fitness = -1.9765314986548954               Solution = [ 1.43448975e-02 -9.8
7708160e-05]
Fitness = -1.6832817331054801                Solution = [-0.40952206  0.0
3775916]
Fitness = -1.6832892021654484                  Solution = [0.38176338 0.1
5340302]
Fitness = -1.6832748633320629                Solution = [ 0.31277984 -0.2
668738 ]
Fitness = -1.9990254090779886                  Solution = [0.00034207 0.0
028976 ]
Fitness = -1.6832909244052126                Solution = [ 0.02054781 -0.4
110205 ]
Fitness = -1.683279557784876                 Solution = [-0.17635994  0.3
7225074]
```

Changing the crossover probability to 1 means that the individuals will always crossover their genes for each generation. This on its own is not that harmful, with a low probability of mutation.

However, setting the mutation probability to 1 means that the genes are always mutated. This means that now, the GA is essentially randomly searching for a solution. Even good solutions are mutated with a 100% probability

This results in too much variation in the offspring, and like mentioned, this means that the GA is essentially now doing a random search. It seems to do better than having no variation, but with too much variation, it is doing slightly worse than with the original parameters.

Exercise 2: WSN genetic algorithm

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.

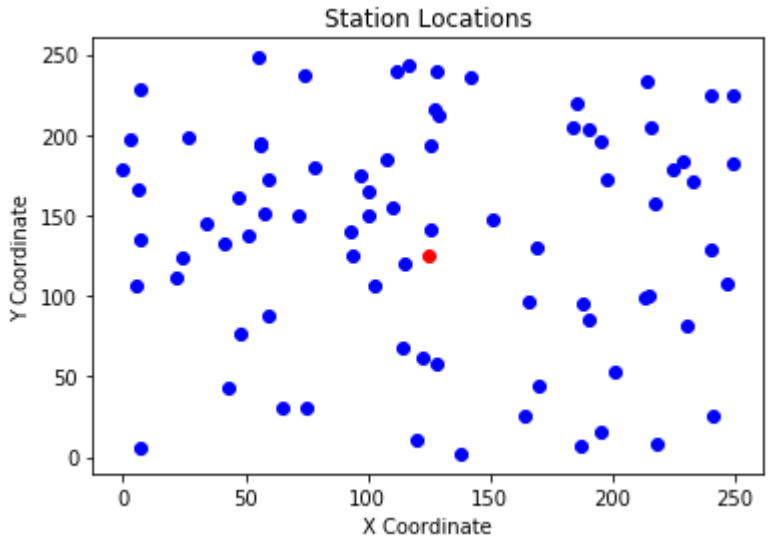
1. Run the script to call the GA and determine the optimal clustering and routing for the WSN. Include the plot of the final results in your report.

In [57]:

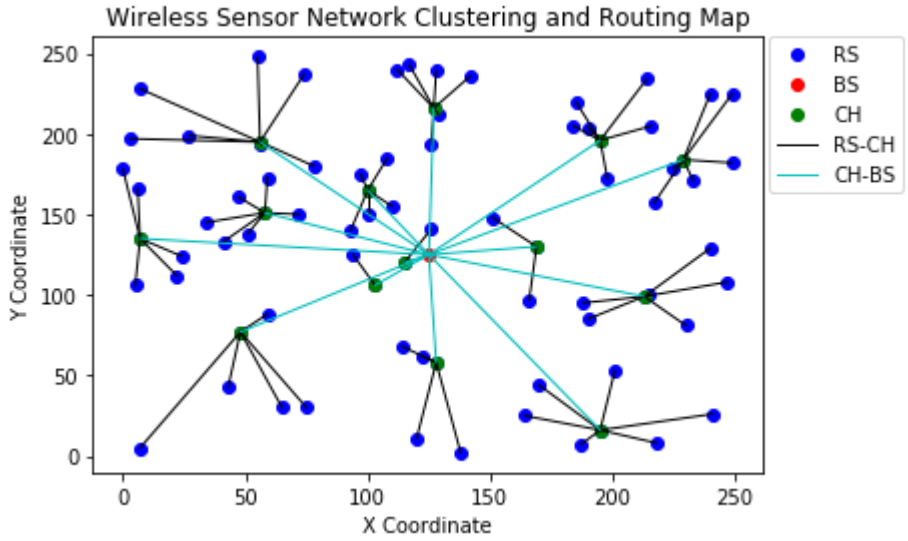
```
def create_individual(data):
    """ Create a candidate solution representation
        0 = regular station; 1 = cluster head
        Represented as a binary sequence with ~25% 1's
    """
    individual = np.random.choice([0, 1], size = (len(data),), p = [0.75, 0
    return individual

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 pop
        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 mut
            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 ba
    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
    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
    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 s
        minDist[i] = np.amin(temp)  # Determine the closest CH to the curre

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

```
    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
    noCH = len(chIndices)
    noStations = len(individual)

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

    # 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 s
        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')   # Ba

    # 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
    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., hand
    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 sta
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,
        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 =  4804.948033425915
Solution =  [0 0 1 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0
 0 1 0 1 0 0]
```



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

```
Pros:
+ Generally faster compared to other learning algorithms
+ Variation means that we can escape local minima or maxima
+ Relatively easy to implement compared to other learning algorithms
+ Can find fit solutions relatively quickly
+ no understanding of how the system works is needed. We just have to know
how to measure fitness

Cons:
- the random heuristics sometimes does not find the optimal solution
- no guaranteed convergence
- the fit solutions found are not guaranteed to be optimal
```

- results will not be good if your fitness algorithm is not good. This can be hard to get right

Abstract

The purpose of this lab was to become familiar with genetic algorithms. Genetic algorithms are an approach to machine learning inspired by the natural process of evolution found in nature. In this lab, we gain a feeling for how changing parameters like the mutation probability and crossover probability can affect the solutions found by the genetic algorithm. The python library pyeasyga was used to implement the genetic algorithms. We also consider the pros and cons of genetic algorithms. We use genetic algorithms in this lab to attempt to find a global minimum in a search space with many localized minima, and a genetic algorithm is also used to determine the optimal locations of cluster heads in weather stations, which collect data from nearby weather stations then send the data to a base station. This allows us to optimize the communication distance by determining the best number of cluster heads, and their locations needed for routing. The fitness function used here was set up so that the genetic algorithm needed to maximize the fitness.

Introduction

In the lab we discover the usefulness of genetic algorithms. Genetic algorithms are inspired by the evolution process found in nature. We begin with a initial random population. Each individual in a population is a candidate solution represented by chromosomes, which are made up of genes, which are typically repreented by binary strings. (Other representations may be used such as integers, real numbers, etc.) THe way in which we choose to represent an individual can have signficant consequences on the performance of a genetic algorithm, since different representation methds can result in varying performance in terms of the accuracy of the genetic algorithm and the computation time. Once we have initialized a population, the individuals are evaluated using the fitness function we define, and we choose to either maximize or minimize the fitness. For creating the next generation, we begin by selecting the most fit individuals, similar to how in nature, the most fit individuals get to survice. Many methods can be used for selection, but for this lab, tournament selection was used, where a tournament style tree is created and the individuals face-off against each other much like how sports brackets are set up in say, soccer. Then, based on the parameters chosen for crossover probability and mutation, the surviving individuals get the opportunity to 'mate' or more technically for genetic algorithms, they get to have their genes crossed over. A higher crossover probability, means that the crossover happens more often. With a crossover probablity of 100%, every individual gets a chance to crossover their genes. Based on the probability for mutation, a surviving individual may have their genes altered by random chance, much like how a populaiton can evolve through useful mutations in nature. Having too high of a mutation probability can be detrimental, however, since for example, with a mutation probability of 100% the genetic algorithm is basically doing a random search since all individuals get their genes mutated, even the individuals that have an optimal solution, whereas with a lower mutation probability, these better performing individuals have a chance to mutate or not, since sometimes a mutation can make the fitness better or worse. Like many other learning algorithms, we want to carefully think about each parameter we want to adjust and how it will affect the finding of the solution. In this lab, we explored how changing the mutation probablity and crossover probablity parameters affects the genetic algorithm's ability to find the global minimum of a function with other localized minima, and a genetic algorithm is also used to determine the optimal locations of cluster heads in weather stations, which collect data from nearby weather stations and then send the data to a base station. This allows us to optimize the communication distance by determining the best number of cluster heads, and their locations needed for routing. The fitness function used here was set up so that the genetic algorithm needed to maximize the fitness.

Conclusion

In this lab, we experimented with genetic algorithms. We used a genetic algorithm to search for a global minimum in a search space with many local minima. We also used a genetic algorithm to look at an actual problem, which is to determine the optimal locations of cluster heads in weather stations. These cluster heads collect data from nearby weather stations and then send the data to a base station. That way, there is no need for all weather stations to communicate with the base station directly. This allows us to optimize the communication distance by determining the best number of cluster heads, and their locations needed for routing. The fitness function used here was set up so that the genetic algorithm needed to maximize the fitness. For this problem, an individual is represented by a binary string with a population size of 80 and 100 generations. The crossover probability at one point is 0.7, and the mutation probability was 0.05. Tournament selection was used. The genetic algorithm for this problem attempted to maximize a fitness function, which depends on the difference in communication distance between all stations individually contacting the base station versus the new approach with cluster heads collecting data from nearby weather stations and then relaying the data to the base station. A pretty plot was generated to display the best solution the genetic algorithm game up with after it has simulated all generations.

Compared the the rest of the lab, the first task, although not difficult at all was relatively speaking more complex than the other tasks in the lab, since some thought had to be put into how changing the mutation probability and crossover probability affects the genetic algorithm's performance in finding a fit solution. Some thought also had to be put into figuring out why a mutation probability of 0% does not generate good results, as well as a mutation probability of 100%. The same goes for a crossover probability of 0% and 100%. In the end, it turns out that we want some variation so that we can escape local minima, but also we do not want too much variation, otherwise we are just doing a random search. If I'm being honest here, writing the Abstract, Introduction and Conclusion was probably the most difficult part of the lab, since writing and coming up with words is hard

In conclusion, this lab served as a solid introduction to genetic algorithms and their uses. We experimented with changing some parameters, and using a genetic algorithm to solve a practical problem.
Evolution in nature works pretty well, so it makes sense and is only natural to apply this logic to artificial systems by attempting to mimic naure and apply the same strategy that was worked for years in the natural world. It is of course, not a silver bullet but like any other learning algorithm excels in some cases, while for other use cases, one may want to consider other learning algorithms. Genetic algorithms can be fun too, since the offspring can produce entertaining results.

Examples of entertaining results:
https://tenor.com/WTgg.gif
https://www.youtube.com/watch?v=K-wIZuAA3EY

Lab 5 Marking Guide

Exercise	Item	Total Marks	Earned Marks
	$Pre-lab$	10	
	$Abstract$	3	
	$Introduction$	3	
	$Conclusion$	4	
1	$Mathematicalgeneticalgorithm$	50	
2	$WSNgeneticalgorithm$	30	
	TOTAL	100	