ECE 449 - Intelligent Systems Engineering

Lab 5-D41: Genetic Algorithms

Lab date: *Thursday, November 25, 2021 -- 2:00 - 4:50 PM*
Room: *ETLC E5-013*
Lab report due: *Tuesday , November 30, 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.

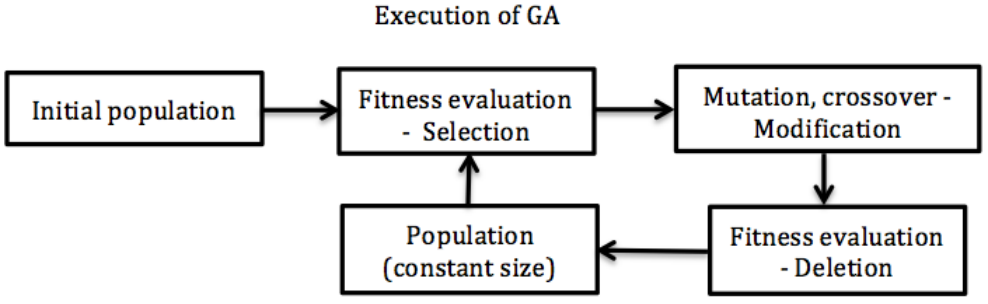
3. Pre-lab
1. Describe and compare roulette wheel (fitness proportional) and ranked selection mechanisms.

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.



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

In [2]:
```
%%bash
# "--user" is essential to install in local environment"
pip install --user -U pyeasyga
```

```
Requirement already satisfied: pyeasyga in /home/jupyter/.local/lib/python3.9/site-packages (0.3.1)
Requirement already satisfied: six in /opt/conda/lib/python3.9/site-packages (from pyeasyga) (1.16.0)
```

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

In [3]:
```
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

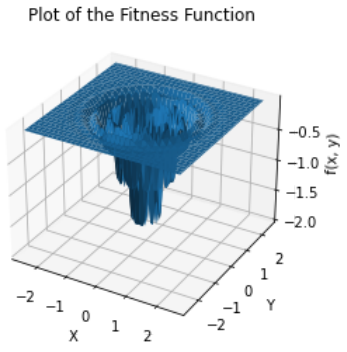
Create a simple genetic algorithm to determine the global minimum of the function:

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

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 [4]:
```
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_75/3339539888.py:9: MatplotlibDeprecationWarning: Calling gca() with keyword arguments was depreca
ted in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword arguments. The gca() functio
n should only be used to get the current axes, or if no axes exist, create new axes with default keyword argument
s. To create a new axes with non-default 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 [5]:

```python
def create_individual(data):
    """ Create a candidate solution representation
        Represented as an array of x and y floating-point values from -10 to 10
    """
    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, 0.5])  # Determine if x and/or y value
    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 sum
            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 population
        Adds a random number from 0 to 9 to each allele in the individual (up to two decimal places)
    """
    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]
    fitness_val = -(1+np.cos(15*np.sqrt(x**2 + y**2))) * np.exp(-x**2 - y**2)
    return fitness_val
```

Write your answer to question 1 below:

In [6]:
```
# Code goes here
def genetic_algorithm(iterations, ga_object):
    for i in range(iterations):
        ga_object.run()
        best_score = ga_object.best_individual()
        print("Iteration {}: \nScore = {}, Point = ({},{})".format(i+1 ,best_score[0],best_score[1][0],best_score[
ga_object = pyeasyga.GeneticAlgorithm(
    None,
    population_size = 30,
    generations = 50,
    crossover_probability = 0.8,
    mutation_probability = 0.05,
    # Since this is a minimization problem
    maximise_fitness = False
)
ga_object.create_individual = create_individual
ga_object.crossover_function = crossover
ga_object.mutate_function = mutate
ga_object.fitness_function = fitness

genetic_algorithm(10,ga_object)
```

```
Iteration 1:
Score = -2.0, Point = (5.25471665402866e-10,-1.0798659531226398e-10)
Iteration 2:
Score = -2.0, Point = (-2.8323223103570175e-10,-4.946149577194299e-10)
Iteration 3:
Score = -2.0, Point = (2.673902284585281e-10,5.680193318812134e-12)
Iteration 4:
Score = -2.0, Point = (3.008041395608325e-10,-2.511161756339391e-10)
Iteration 5:
Score = -1.9999999999997449, Point = (-4.704387726426315e-08,-3.949883673974281e-09)
Iteration 6:
Score = -2.0, Point = (-1.0767749783214264e-10,-9.175646381312566e-11)
Iteration 7:
Score = -1.9999999999991387, Point = (3.260586446069715e-08,-8.036923602297061e-08)
Iteration 8:
Score = -1.9999999999999853, Point = (2.563754713270498e-09,-1.1049789575397604e-08)
Iteration 9:
Score = -2.0, Point = (4.9147263904437016e-11,-5.530725062704689e-12)
Iteration 10:
Score = -2.0, Point = (1.9576737723531212e-10,-9.302824634649446e-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 [7]:
```
ga_object.mutation_probability = 0
ga_object.crossover_probability = 0

genetic_algorithm(10,ga_object)
```

```
Iteration 1:
Score = -0.004906983148904471, Point = (-0.9785973589728147,-1.6612699313305725)
Iteration 2:
Score = -0.08256219670347535, Point = (-1.626503589217144,-0.5959280037444863)
Iteration 3:
Score = -2.9495560805278783e-08, Point = (-3.201763963040749,-2.7691435223626115)
Iteration 4:
Score = -0.856575530228097, Point = (0.6844220956811604,-0.35774267524230474)
Iteration 5:
Score = -0.019712867079549595, Point = (-0.9586676934321368,-1.9034670089249488)
Iteration 6:
Score = -0.016106210070882764, Point = (-2.0925341975104317,0.5015667599223974)
Iteration 7:
Score = -0.0038754310824518916, Point = (-0.14734651565236945,2.4424139329362795)
Iteration 8:
Score = -0.005390809668844535, Point = (0.2289847201920363,-2.205142429034961)
Iteration 9:
Score = -0.0010594717985735747, Point = (0.4795559031299206,-2.573500740670518)
Iteration 10:
Score = -0.40154066807783456, Point = (1.0019122926041586,-0.6696314764559048)
```

The genetic algorithm that uses a mutation and crossover probability of zero does not find an optimal solution because there is no variation between the offspring and the parent (offspring replicates the parent). From the above results we can see that the scores are no where near the optimal score (around -2.0) and are varying greatly with each generation. Since there is no variation between the offspring and the parent, the genetic algorithm will not search at all for new solutions.

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

In [8]:
```
ga_object.mutation_probability = 1
ga_object.crossover_probability = 1

genetic_algorithm(10,ga_object)
```

```
Iteration 1:
Score = -1.9733784377823365, Point = (0.0014828199905592386,0.015210555407005714)
Iteration 2:
Score = -1.9999608347618612, Point = (0.00011814109514045394,-0.0005727996461760823)
Iteration 3:
Score = -1.6831687077243116, Point = (0.316261923320153,0.26513279266594836)
Iteration 4:
Score = -1.9999999906500066, Point = (4.954855047517958e-06,-7.557032679699528e-06)
Iteration 5:
Score = -1.996904030953141, Point = (-0.004978421864802092,-0.0015061713368141163)
Iteration 6:
Score = -1.9659383326433304, Point = (-0.008352388582485368,-0.015147716920959165)
Iteration 7:
Score = -1.99999357504014, Point = (-8.333440008766272e-05,-0.000221739995131682)
Iteration 8:
Score = -1.953515699527078, Point = (-0.01930472825812568,-0.006045174899149852)
Iteration 9:
Score = -1.9964913061153597, Point = (-0.0038724541610062322,-0.003958024860237877)
Iteration 10:
Score = -1.9635175088430263, Point = (-0.0019752690640931136,-0.017796436000028316)
```

When changing the crossover and mutation probabilities to zero, this performs well (better than 0 probability) in terms of finding a solution since we find scores close to the optimal score (around -2). However, by setting these probabilities to one, we've essentially transformed this Genetic Algorithm into a Blind Random Search. Although this does better than the zero probability case, it is doing a little worse than the case with the inital 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

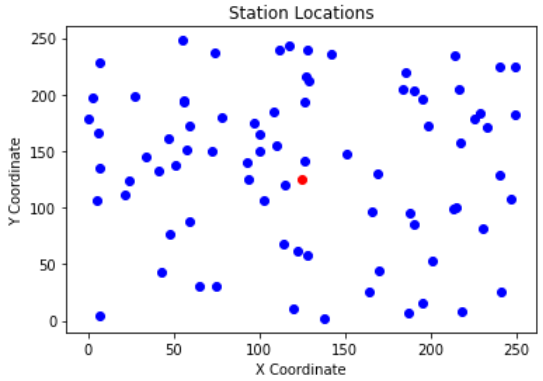
The GA has been built to solve the WSN routing optimization problem in the cell below. The base station is located in the centre of a (250, 250) map, with 80 stations assigned randomly around it.
An individual is represented as a binary string, with length equal to the number of stations in the network. 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.

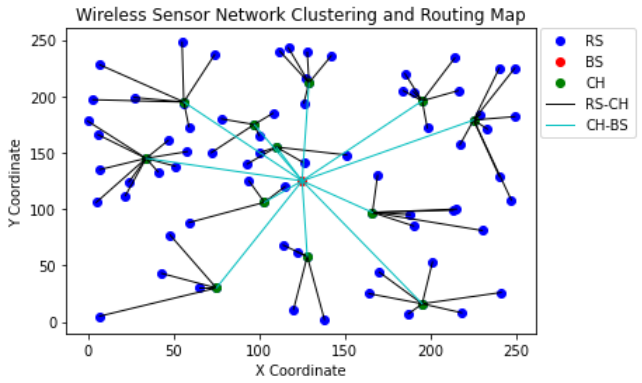
In [9]:

```
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, 0.25])
    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 population
        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 mutated within the individual
            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 base station (BS) [D]
    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 stations that are CH'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 the nearest CH
    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 station and each CH
        minDist[i] = np.amin(temp)  # Determine the closest CH to the current station

    newDist = np.sum(chBSDist) + np.sum(minDist)  # Sum of distances from stations to CH's + CH's to BS [distance_

    fitness = (totDist-newDist) + (noStations-noCH)  # Fitness value to be maximized [D-distance_i + N-H_i]
    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 stations that are CH's
    noCH = len(chIndices)
    noStations = len(individual)

    stationConnectivity = np.zeros((noStations+1, noStations+1))  # 0 = not connected; 1 = station-CH; 2 = CH-BS;

    # 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 station and each CH
        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')  # Regular stations in blue
    bsHandle ,= plt.plot(bsCoords[0], bsCoords[1], 'ro', label = 'BS')  # Base station in red

    # 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 = 'CH')  # Cluster heads in green

    # 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], data[j, 1]], color = 'k', linewidth
            if (stationConnectivity[i, j] == 2):
                chBSHandle ,= plt.plot([bsCoords[0], data[j, 0]], [bsCoords[1], data[j, 1]], color = 'c', linewidt
    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., handles = [stationHandle, bsHandle, chHand
    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 stations
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, 0:2])
        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 =  4683.337235394919
Solution =  [1 0 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 0 0]
```



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

The pros and cons of using a genetric algorithm (GA) as a problem-solving technique depend on the problem that is being solved, however, there are some general pros and cons that we can outline. Using GA's as a problem solving technique is advantageous in that it provides high quality solutions, it uses a broader search space (searches from a population) to find potential solutions, it uses objective functions to determine the quality of solutions instead of gradient methods, it is also unlikely that genetric algorithms would get stuck in local optima (maxima/minima) and they work well with both/mixed discrete and continuous problems. Although there are many advantages to using Genetic algorithms as a problem-solving technique, there are also disadvantages. Utilizing GA's as a problem-solving is computationally expensive, tuning control parameters is time consuming, slow or premature convergence to a solution can occur and coming up with a good heuristic (fitness function) is difficult.

Abstract

The purpose of this lab was to get familiarized with the basics of genetic algorithms, by solving two separate problems. The first problem involved implementing a simple genetic algorithm to find the global minimum of a function. By plotting the function, it was clear that traditional gradient descent methods would not work since there is a very high chance we get stuck in one of the local minima. As a result, we created a genetic algorithm with the a population size of 30, 50 generations, crossover probability of 0.8, mutation probability of 0.005 and a tournament selection method. After creating this genetic algorithm and running it multiple times, we obtain a score (fitness/f(x,y) value) of around -2 consistently. Afterwards, we proceed to change both the mutation and crossover probabilities to 0 and the result of this was that no optimal solution was found. This is because by setting these probabilities to zero, there is no variation between the offspring and the parent. By running this multiple times, it was clear that with each iteration the optimal score and points were varying greatly. Then, we proceeded to set both the mutation and crossover probabilities to 1 and then re-run our genetic algorithm again. The results were an improvement from the 0 probability run, however, they were not better than our original parameters; as there was still some deviation from the optimal score. The reason why this performs relatively well is that since the probabilities have been set to 1, the Genetic Algorithm has been transformed into a blind random search. The next problem involved aiding a team in optimizing the energy usage of their wireless sensor network. Instead of the team having all stations send their data directly to the base station, they aim to have cluster heads to collect data from nearby regular stations. These cluster heads would then be responsible in communicating with the base station so that not every station would have to communicate with the base station. This change is done to reduce the total communication distance. Here, we employ a genetic algorithm to find the optimal number of cluster heads and their respective locations. The parameters used for this genetic algorithm are a population size of 80, 100 generations, crossover probability of 0.7, mutation probability of 0.05, tournament selection method and we use binary string representations to represent the individuals. The result of our run provides a fitness value of 4683 and uses a total of 12 cluster heads, from the generated routing graphical results, it is clear that the distance of communication has been reduced. Lastly, for this lab we list the pros and cons of using Genetic Algorithms as a problem solving technique. Genetic Algorithms use a broader search space and are unlikely to get stuck in local optima (maxima/minima), however, employing them is computationally expensive and premature or slow convergence to a solution may occur.

Introduction

In this lab, our aim is to get familiarised with Genetic Algorithms and utilize them to solve two cases. Genetic Algorithms are heuristic-based search algorithms that take inspiration from Charles Darwin's theory of evolution. The algorithm starts off with an initial population (usually uniformly randomly selected), where each individual (chromosome) in this population is a solution to the given problem. A fitness function is then used to assign a fitness score to each solution in the population to determine how "fit" a solution is. The next stage in this algorithm is to use this fitness score to perform selection of the fittest individuals (individuals with the highest scores). Then crossover and mutation are used to produce offsprings (new solutions) from the parents. These steps (besides intializing a population) are repeated until the population "converges" which means that there is no significant difference between the current generation and the previous one. This process that makes up a Genetic Algorithm is used to solve the two problems in this lab. The first problem involves finding the global minimum of a funtion. This function contains many "trenches" (local minima) that traditional gradient descent methods would get stuck in. Using a genetic algoirithm with tuned parameters such as population size, number of generations, crossover probability, mutation probability and selection methods will allow us to avoid getting stuck in such "trenches". The second part of this lab involves assisting a team in reducing the energy usage of their Wireless Sensor Network (WSN) by optimizing the distance that their communication lines cover. This optimization is done by using cluster heads that regular stations will have to communicate with if they wish to communicate with the base stations. Afterwards, the cluster heads will realy that information to the base station. The implemented genetic algorithm with tuned parameters will help us find the optimum number of cluster heads needed and their respective placement in the 250x250 map.

Conclusion

In conclusion, this lab served as a great introduction to putting our genetic algorithms knowledge to practical use by utilizing the pyeasyga library to solve two problems. For the first problem which involved finding the global minima of a given function, we started by plotting the given function. The plot clearly showed that there were many local minima and that traditional gradient descent methods would not work. By utilizing a genetic algorithm with a population size of 80, 50 generations, crossover probability of 0.8, mutation probability of 0.005 and tournament selection method, we were able to obtain a score of around -2 from multiple runs of this genetic algorithm. Afterwards, the mutation and crossover probabilities were set to 0, after re-running the genetic algorithm, it was clear that doing this resulted in the offspring and parent having no variation. The obtained scores from many runs were varying with each iteration since the offspring replicates the parent. Next, the mutation and crossover probabilities were set to 1 and the genetic algorithm was re-run for 10 iterations. The results were not better than our original parameters, however they were an improvement from the 0-probability run. The reason why this performs relatively well is that since the probabilities have been set to 1, the Genetic Algorithm has been essentially transformed into a blind random search. For the second problem the genetic algorithm used had a population size of 80, 100 generations, crossover probability of 0.7, mutation probability of 0.05, tournament selection method and we use binary string representations to represent the individuals. The result of our run returned a fitness value of 4683 and from the number of 1's in the binary representation, there was a total of 12 cluster heads. Genetic Algorithms are a great tool to solve problems as they use a broader search space and there is a lower chance that they get stuck in local optima, however, employing them is computationally expensive and fine tuning the parameters of the genetic algorithm is time-consuming as the issue may be with the derived fitness function the whole time.

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	