Computational Intelligence Optimisation

Lab 2: Introduction to GAs using Deap

For this exercise we are going to simulation/solve the Moth scenario discussed in the Venables et al. paper shared earlier using the DEAP libraries. The process to do so is as follows:

1. Import libraries and define evolutionary parameters
2. Set up the toolbox
3. Define fitness, individual and population creators
4. Create the fitness function
5. Set up the genetic operators
6. Run the algorithm

Step 1: Import libraries and define evolutionary parameters:

```
##**************************************************
# import libraries
from deap import base
from deap import creator
from deap import tools
from deap import algorithms

import random
import numpy as np

import matplotlib.pyplot as plt

##**********************************************

## parameters
chrom_size = 100
population_size = 200
P_CROSSOVER = 0.9
M_MUTATION = 0.1
MAX_GENERATIONS = 200
RANDOM_SEED = 42

```

Step 2: Set up the toolbox

```
## set up the toolbox
toolbox = base.Toolbox()
```

Step 3: Define fitness, individual and population creators

```
    ## define the structure of the chromosome and register the funct
toolbox.register("binary", random.randint, 0,1)

    ## create a function to evaluate the individual and providing on
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

    ## create a function to create an individual
creator.create("Individual", list, fitness=creator.FitnessMax)

    ## define/register a function to create an individual
toolbox.register("IndividualCreator", tools.initRepeat,
                 creator.Individual, toolbox.binary, chrom_size)

    ## define/register a function to generate a population of indivi
toolbox.register("PopulationCreator", tools.initRepeat,
                 list, toolbox.IndividualCreator)
```

Step 4: Create the fitness function

```
    ## set up a fitness/evaluation function [(
def fitnessFunction(individual):
    return sum(individual), ## return the ind:
```

Step 5: Set up the genetic operators

```
toolbox.register("evaluate", fitnessFunction)

toolbox.register("select", tools.selTournament, tournsize=3)

toolbox.register("mate", tools.cxTwoPoint)

toolbox.register("mutate", tools.mutFlipBit, indpb=1/population_size)
```

Step 6: Run the algorithm

```python
## run the algorithm
def main():

    population = toolbox.PopulationCreator(n=population_size)

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", np.max)
    stats.register("avg", np.mean)

    population, logbook= algorithms.eaSimple(population,
                                              toolbox, cxpb=P_CROSSOVER,
                                               mutpb=M_MUTATION,
                                               ngen= MAX_GENERATIONS,
                                               stats=stats,verbose=True)

    maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")

    plt.plot(maxFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.xlabel('Max/average fitness')
    plt.ylabel('Max/average fitness over generations')
    plt.show()

if __name__ =="__main__":
    main()
```

Exercise 2:

Implement a GA to solve the Guard Scheduling Problem