A Genetic Algorithm from scratch in Python

Syllabus

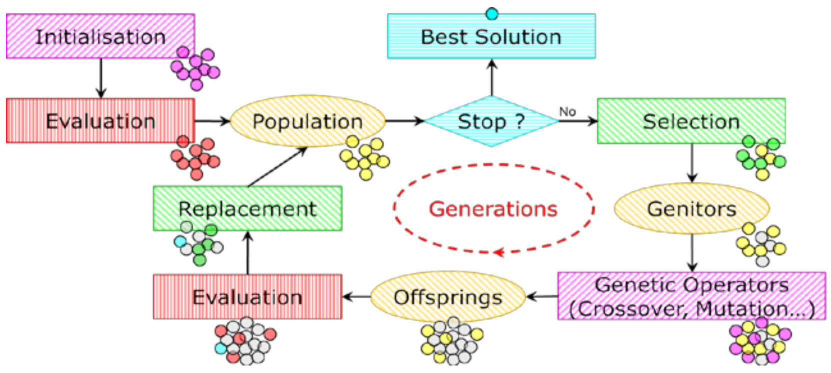
Goal of this lab is to create a genetic algorithm from scratch using Python language. Discrete (OneMax benchmark) and continuous (Ackley) optimizations are tackled.

Exercice 1 Configuring Python

1. Install python language on your computer (https://www.python.org/downloads/). You could also use alternatives packages as WinPython (http://winpython.github.io).
2. Make sure that `pip` command is available. Test with `pip --version`. If not, do not forget to add Python path to your OS.
3. I recommend using any IDE (Integrated Development Environment) dedicated to Python as PyCharm, Spyder or whatever similar.

Exercice 2 GA from Scratch

The different steps of an EA are represented below:



1. Create a new project in your favorite IDE (I suppose it is PyCharm) and create a new python file called `BasicGA.py`. The first thing we need is a pseudo-random number generator (`prng`). Then, in the main program, we affect its seed in order to be reproducible. We define a population size to 10 and consider that each individual in the population is encoded by 20 genes (bits). We then call method `geneticAlgorithm` that populate the population randomly and print all individuals. Execute your program several times. What can you observe? Now, change the seed with `prng.randint(0,10000)`. What is the effect?

```
import random as prng

def geneticAlgorithm(popsize:int, indsize:int):
    population = [[prng.randint(0, 1) for _ in range(indsize)] for _ in range(popsize)]
    for ind in population:
        print(ind)

if __name__ == '__main__':
    prng.seed(0)
    popsize = 10
    indsize = 20
    geneticAlgorithm(popsize=popsize, indsize=indsize)
```

2. Remove printing the population and change the seed to 0. The *initialization* phase is done!
3. Now, we have to define the evaluation function of one individual. Usually, one of the simplest problem is to maximize the number of '1' contained in an individual. This is a benchmark function called *MAX_ONES*. Of course, we already know that the optimum value for this optimization problem equals the size of an individual. It is just a way to observe that the evolution process converges to the real optima and one way to prove that it works. We simply implement it as the sum of all genes contained in the individual. Remark that we give the type of each

parameter in input and output in python:

```python
'''Maximize the number of one in the individual'''
def maxOnes(individual:list[int]) -> float:
    return sum(individual)
```

4. Then, the new function is given as an argument to the genetic algorithm and we save fitness value of all individuals in a list:

```python
def geneticAlgorithm(evaluate,popsize:int, indsize:int):
    ...
    fitness = [evaluate(ind) for ind in population]

if __name__ == '__main__':
    ...
    geneticAlgorithm(evaluate=maxOnes, popsize=popsize, indsize=indsize)
```

5. For selecting individuals considered as genitors, the most used mechanism is *tournament* selection. This selection needs the result of evaluations and the size *T* of the tournament. The principle is to select randomly *T* individuals in the population and keep the best among them as one parent. We define a default value of 2 for the tournament size. Iterate again for having another parent.

```python
''' Select one individual using
 Tournament selection of size size
 Return index of selected individual'''
def tournamentSelection(fitness:list[float], size:int=2)->int:
    selected_index = prng.sample(range(len(fitness)), size)
    best_index = np.argmax([fitness[i] for i in selected_index])
    return selected_index[best_index]
```

6. Once parents are selected, they are crossed for obtaining offspring. For this mechanism, we use one point crossover.

```python
''' One-point XOver on two individuals'''
def onePointXOver(parent1:list[int], parent2:list[int])-> list[list[int]]:
    child1, child2 = parent1.copy(), parent2.copy()
    min_length = min(len(parent1),len(parent2)) #if parents have not same length
    pointcut = prng.randint(1,min_length-1) # to be sure to be inside the genome
    child1[pointcut:] = parent2[pointcut:]
    child2[pointcut:] = parent1[pointcut:]
    return [child1,child2]
```

7. After crossover, mutation is applied according to a certain probability. In the case of string of bits, bits are simply flipped. Using assertion, we check whether given probability is in [0,1].

```python
'''Bit-flip mutation'''
def bitflipMutation(parent:list[int], probability:float=0.05)->list[int]:
    assert probability>=0.0 and probability<=1.0 ,"Probability of mutation should be in [0,1]."
    for indexGene in range(len(parent)):
        if prng.random()<=probability:
            parent[indexGene] = 1 - parent[indexGene]
    return parent
```

8. We just define all operators for creating a new population. Now, we have to compute offspring from current population and their corresponding fitness.

```python
'''create offspring with same size as parent'''
def offspring(parent:list[list[int]],fitness:list[float])->list[list[int]]:
    offspring=[] #create an empty list of offspring
    popsize=len(fitness)
    while len(offspring)<popsize:
        ind1 = tournamentSelection(fitness)
        ind2 = tournamentSelection (fitness)
        off = onePointXOver(parent[ind1], parent[ind2])
        for ind in off:
            ind=bitflipMutation(ind)
        offspring.extend(off)
    return offspring[:len(parent)] #when popsize is odd
```

9. At that point, we should have parents and offspring with corresponding fitness. We started with a population of size 10 and we created 10 offspring, so we have 20 individuals. In order to have a constant population size, the simplest mechanism will replace all parents by offspring. However, imagine that one parent has the best fitness among the parents and the children. By doing so, you may loose the best individual found so far. That is why *elitism* is often introduced in evolutionary computation. Some best individuals from parents are kept for next generation.

Here, we add a new parameter that corresponds to the number of best parents kept.

```
'''Define individuals of next generation based on parents & offspring.
Elitism is the number of best individuals from parents kept for next generation'''
def replacement(parent:list[list[int]],fitness:list[float],
                offspring:list[list[int]],fitnessOff:list[float],elitism:int=1):
    newpop = [] # next generation
    newfit = [] # and corresponding fitness
    popsize = len(parent) # and corresponding fitness
    while elitism>0:
        best_index = np.argmax(fitness) # find best individual in parents
        newpop.append(parent.pop(best_index)) # remove best_index from parent and add to newpop
        newfit.append(fitness.pop(best_index)) # same for fitness
        elitism -= 1
    newpop.extend(offspring[:popsize -len(newpop)])
    newfit.extend(fitnessOff[:popsize - len(newfit)])
    return newpop, newfit
```

10. In order to implement the generational loop, one of the best practice is to consider how many calls have been done to function `evaluate`. This is called *Number of Function Evaluations* (`NFE`). We also need the maximum of NFE (`MAX_NFE`) to know when the algorithm will stop. A good value for `MAX_NFE` could be 10 times the size of one individual. Do not forget to increment NFE when evaluating individuals. Finally, the algorithm could return the best individual with its fitness:

```
def geneticAlgorithm(evaluator,popsize:int, indsize:int, MAX_NFE:int)->(list[int],float):
    NFE = 0
    population = [[prng.randint(0, 1) for _ in range(indsize)] for _ in range(popsize)]
    fitness = [evaluator(ind) for ind in population]
    NFE += len(fitness)
    # generationnal loop
    while NFE < MAX_NFE:
        off = offspring(population, fitness)
        fitnessOff = [evaluator(ind) for ind in off]
        NFE += len(fitnessOff)
        population, fitness = replacement(population, fitness, off, fitnessOff)

    best_index = np.argmax(fitness)
    return population[best_index], fitness[best_index]

if __name__ == '__main__':
    ...
    bestInd,bestFit = geneticAlgorithm(evaluator=maxOnes, popsize=popsize, indsize=indsize,
                                                    MAX_NFE=10*indsize)
    print(bestInd,bestFit)
```

11. Execute your algorithm; it should work. Now, change the value of `MAX_NFE` to `5*indsize`. What can you observe? Explain!
12. The main disadvantage of this program is that we obtain a result without knowing what happens during the execution. We would like to do something (printing, plotting, saving …) during the run. To do so, we add a list parameter `observers` to the genetic algorithm and call all methods included in this list in the generational loop:

```
def geneticAlgorithm(evaluator,popsize:int, indsize:int, MAX_NFE:int, observers=[])->(list[int],
float):
    ...
    # generationnal loop
    while NFE < MAX_NFE:
        for obs in observers:
            obs(population,fitness,NFE)
        ...
```

13. Now, we have to define what to do; for instance, printing the best individual of the population. Execute.

```
''' print to console best individual of population during the run'''
def printBestIndividual(population:list[int], fitness:list[float], NFE:int):
    best_index = np.argmax(fitness)
    print(NFE, population[best_index], fitness[best_index])

if __name__ == '__main__':
    ...
    bestInd,bestFit = geneticAlgorithm(evaluator=maxOnes, popsize=popsize, indsize=indsize,
                                       MAX_NFE=10*indsize, observers=[printBestIndividual])
```

14. And, at the same time, save some statistics on the population in a file? Execute and open `csv` file with Excel.

```python
''' save statistics of current population to a file during run'''
def saveStatistics(population: list[int], fitness: list[float], NFE: int):
    fileName='./statistics.csv'
    if NFE==len(fitness):
        file = open(fileName, 'w')
        file.write('NFE \t avg \t min \t max \t std')
    else:
        file = open(fileName, 'a')
    file.write("\n{0}\t{1}\t{2}\t{3}\t{4}".format(NFE, np.average(fitness), np.min(fitness),
                                                   np.max(fitness), np.std(fitness)))
    file.close()

if __name__ == '__main__':
    ...
    bestInd,bestFit = geneticAlgorithm(evaluator=maxOnes, popsize=popsize, indsize=indsize,
                                       MAX_NFE=10*indsize,
                                       observers=[printBestIndividual, saveStatistics])
```

15. Modify all previous code for changing the mutation probability and the tournament size when calling `geneticAlgorithm` method and add a probability of crossover. You might group all parameters in only one dictionary parameter containing all previous ones.

Exercice 3 Continuous optimization

Optimizing ones in a string of bits is not very interesting. A clever task could be to find the minimum value of Ackley[1] function $f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(\cos(2\pi x+\cos(2\pi y)))} + e + 20$ for *x* and *y* in [-5, 5] which equals (0, 0).

1. As we want to reuse what has been previously done, we will consider that one individual, a string of bits, represents *x* and *y* in this case. We previously defined a problem with 20 bits, and, for this new problem, we have to optimize two variables *x* and *y*: so, we simply consider that each variable is encoded by 10 bits in the genotype. The first thing to do is to define a new evaluation function called `Ackley` with the same prototype as `maxOnes` function defined previously and consider it for optimization:

```python
from math import *

'''Akcley function'''
def Ackley(individual:list[int]) -> float:
    a = 20
    b = 0.2
    c = 2*pi
    number_variables = 2
    bounds = [[-5.0, 5.0], [-5.0, 5.0]]
    values= decode_float_values(bounds, number_variables, individual)
    f = -a * exp(-b * sqrt(1.0/number_variables) * sum([xi**2 for xi in values])) \
           - exp (1.0/number_variables * sum([cos(c*xi) for xi in values])) + a + e
    return f

if __name__ == '__main__':
    ...
    bestInd,bestFit = geneticAlgorithm(evaluator=Ackley, popsize=popsize, indsize=indsize,
                                       MAX_NFE=10*indsize,
                                       observers=[printBestIndividual, saveStatistics] )
    print(bestInd, bestFit)
```

2. As opposed to previously, Ackley function needs reals and not bits; that is why the string of bits is converted using `decode_float_values` method which needs number of variables (2), the bounds of each variable ([-5, 5]) and

[1] https://en.wikipedia.org/wiki/Test_functions_for_optimization

the string to decode:

```python
'''Decode a string of bits (genotype) as a list of number_variables float values.
Bounds for each variable is stored in bounds.'''
def decode_float_values(bounds:list[list[float]], number_variables:int, genome:list[list[int]])->list[float]:
    decoded = list()
    n_bits= len(genome)//number_variables
    largest = 2 ** n_bits
    for i in range(len(bounds)):
        # extract the substring
        start, end = i * n_bits, (i * n_bits) + n_bits
        substring = genome[start:end]
        # convert bitstring to a string of chars
        chars = ''.join([str(s) for s in substring])
        # convert string to integer
        integer = int(chars, 2)
        # scale integer to desired range
        value = bounds[i][0] + (integer / largest) * (bounds[i][1] - bounds[i][0])
        # store
        decoded.append(value)
    return decoded
```

3. Now, you can execute. What are the results? Is it correct? Explain Why! What do we need to change?
4. Increase NFE and look at the results. Can you obtain optimal value? If not, what do you propose to change?

Sources

- Simple genetic algorithm from scratch in Python, https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/, Juillet 2021.