Observing EA behaviors and apply statistical tests with EA library

Syllabus

Goal of this lab is to discover and apply Evolutionary Algorithms (EA) in Python using Platypus library. Firstly, we apply Genetic Algorithm on a 2D benchmark function (Ackley), then we compute some information during evolution and finally we plot results. Next, we work on a classical combinatorial optimization problem (TSP) with one objective.

Exercice 1 Configuring Platypus

There exist many EA libraries (see https://sites.google.com/view/benchmarking-network/home/resources for a more exhaustive list). As mentioned on its website, Platypus is especially dedicated to Multi-Objective evolutionary Optimization (MOO) and provides many up-to-date algorithms (NSGA-II, NSGA-III, SMPSO, MOEA/D…). Like other good EA libraries (like ECJ in Java), it allows both using a special algorithm and defining its own.

1. For installing Platypus, just type `pip install platypus-opt` in a command line.
2. Download `ackley.py` from Moodle and execute it to check whether Platypus is correctly installed. Ackley function from previous lab has been reused and adapted.

Exercice 2 Plotting Evolutionary Algorithm (EA) results

1. Could you look in platypus code to understand to what corresponds integer value of method `run`?
2. In order to have a better understanding of what happens, add a callback to new method `printBestIndividual` during the execution of the algorithm and use following code. What can you see?

```python
def printBestIndividual(self):
    """ Function called during each iteration of the algorithm"""
    # select one individual in population regarding its objective value as minimized 'larger=False'
    bestSolution = truncate_fitness(self.population,1,
                               larger_preferred=self.problem.directions[0]==self.problem.MAXIMIZE,
                               getter=objective_key)[0]
    print ('{0} {1:4f} {2:4f} {3:4f}'.format(self.nfe, bestSolution.variables[0],
                                             bestSolution.variables[1],
                                             bestSolution.objectives[0]))
```

3. Rather than printing results, we would like to store some data (as the *average*, *minimum*, *maximum* and *standard deviation* of the population's fitness) for future usage. To do so, we firstly `import numpy as np` for mathematic computation and add new method `saveStatistics` using following code. Modify algorithm execution.

```python
def saveStatistics(self):
    """ Observer function for saving population statistics
    called during each iteration of the algorithm"""
    # check whether self has attribute 'statistics' ?
    if not hasattr(self, 'statistics'):
        self.statistics = {'nfe':[],'avg':[], 'min':[], 'max':[],'std':[]}
    self.statistics['nfe'].append(self.nfe)
    fitness = [x.objectives[0] for x in self.population]
    self.statistics['avg'].append(np.average(fitness))
    ... # TO BE COMPLETED
```

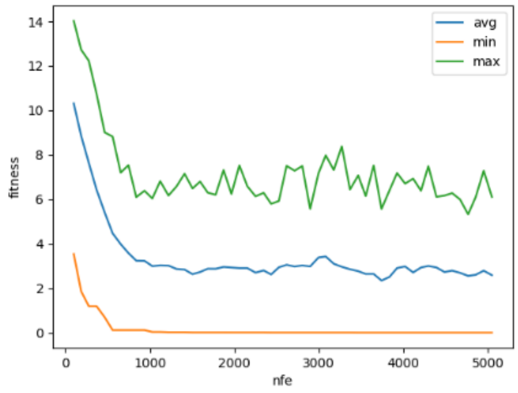
4. Now, the problem is that we can call only one function at a time (print or save)! How to do both or more? To do so, we create a new method that look for an attribute called `observers` and call all methods contained in this attribute. Modify algorithm execution such that you can call both methods `saveStatistics` and

`printBestIndividual`.

```
def Observers(self):
    """ Defines a set of functions to be called"""
    if hasattr(self, 'observers'):
        for obs in self.observers:
            obs(self)
    else:
        raise NameError("Unknown attribute 'observers'. No method to call.")
```

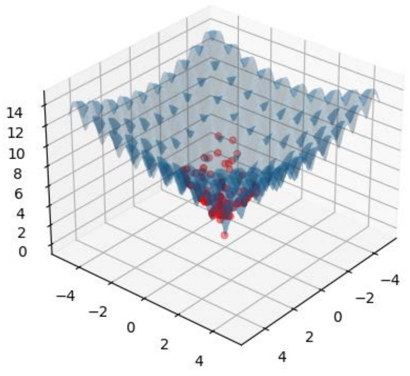
5. It is time to use statistics previously stored during evolution. Once algorithm has ran, plot the results in order to obtain following plot. You should `import matplotlib.pyplot as plt`. It is better to create `plotStatistics` method. What means `nfe`?

```
def plotStatistics(self):
    if hasattr(self, 'statistics'):
        fig=plt.figure(0)
        plt.plot(...,label='avg')
        ... # TO BE COMPLETED
        plt.xlabel('nfe')
        ... # TO BE COMPLETED
        plt.legend()
        #Should we wait action from user ? => block=True
        plt.show(block=True)
    else:
        raise NameError("Unknown attribute 'statistics'
        for plotting statistics. Method 'saveStatistics'
        should be used as observer.")
```



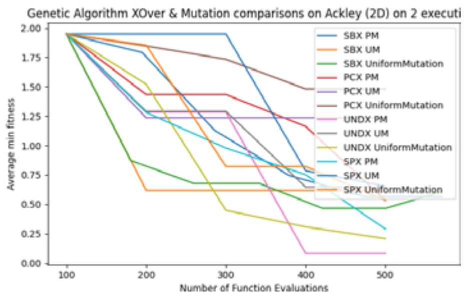
6. Now, we would like to understand how individuals are evolving in the search space. It means that, at each iteration, the search space (Ackley function in blue) and individuals (in red) found by the EA in that search space should be displayed. Complete following observer function and execute your program to know what is going on.

```
def plotSearchSpace(self):
    """ Plot search space and population"""
    fig = plt.figure(1)
    ax = fig.gca(projection='3d') # create a 3D plot
    ax.view_init(30, 40) # change 3D view
    x = np.arange(-5, 5, 0.1) # set of float values between
    y = np.arange(-5, 5, 0.1) # -0.5 and 0.5 step 0.1
    X, Y = np.meshgrid(x, y) # dot product between x & y
    Z = [ackley([a,b]) for a,b in zip(np.ravel(X),np.ravel(Y))]
    Z = np.array(Z).reshape(X.shape)
    surf = ax.plot_surface(X, Y, Z, alpha=0.3)
    ... # TO BE COMPLETED (plot solutions in red)
    surf = ax.scatter(solX,solY,solZ, color='red')
    plt.show()
```



7. If you look at default variation operators used by Platypus for Genetic Algorithm in `config.py` file, you will find `SBX` and `PM` operators. You can obtain exactly the same result as previously by changing the line `algorithm = GeneticAlgorithm(myProblem)`by `algorithm = GeneticAlgorithm(myProblem, variator = GAOperator(SBX(), PM()))` in your code where SBX is the crossover and PM the mutation. Moreover, in `operators.py` file, it is possible to identify other operators for real: `PCX`, `UNDX`, `SPX` for crossover and `UniformMutation`, `NonUniformMutation`, `UM` for mutation. What is interesting, now, is to launch as many executions as there are possible combinations to then deduce which one is the most suitable for this optimization problem.

```
nfe=1000
nexec = 3
Xovers=[SBX()...] # all possible Xovers
Mutations=[PM()...] #all possible Mutations
fig=plt.figure() # a new figure
# for all combinations of Xover and Mutation
for Xover,Mutation in [(x,m) for x in Xovers for m in Mutations]:
   resultNfe, resultMin=[], [] # empty results
   XoverName, MutName=type(Xover).__name__, type(Mutation).__name__
   for seed in range(nexec): # execute same algorithm several times
   random.seed(seed) # modify current seed
      ... # TO BE COMPLETED
      # create a pandas serie for Nfe & Min fitness
      resultNfe.append(pd.Series(algorithm.statistics['nfe']))
      resultMin.append(pd.Series(algorithm.statistics['min']))
      # execution may be long, so print where we are
      print ('run {0} with {1} {2}'.format(seed,XoverName,MutName))
      # mean of all executions for same algorithm using pandas
      X = pd.concat(resultNfe,axis=1).mean(axis=1).tolist()
      Y = pd.concat(resultMin,axis=1).mean(axis=1).tolist()
      ... # TO BE COMPLETED
   plt.title('Genetic Algorithm XOver & Mutation comparisons on
   Ackley ('+ (myProblem.nvars) +'D) on '+str(nexec)+' executions')
   plt.xlabel('Number of Function Evaluations')
   plt.ylabel('Average min fitness')
   plt.legend()
   plt.show()
```



8. Test with increased values of `nfe=10000`, `nexec=30` and number of Ackley variables (2, 10 for instance). What could you deduce? What is the influence of increasing the complexity of Ackley function?

Exercice 3 Real experiments

It is not correct to "visually" deduce which operator or technique beats the others, especially on only one problem (Ackley function). As Platypus is mainly developed for multi-objective optimization problems (MOP), there is few materials for single objective optimization problems (SOP). That is why, we install single-objective benchmark problems using optproblems package and use the 25 benchmark functions from CEC 2005 test problems. We next program some experiments.

1. Install the new package via `pip install optproblems`. Is the previous Ackley function contained in the CEC benchmark test? If yes, what is its name?
2. Python is a dynamic programming language that do not impose to specify types of manipulated objects and variables. In most cases, this is simpler to program; however, it may cause some problems. It is exactly the case here because I discovered that, in optproblems, CEC2005 functions sometimes return *float* and other times return *numpy.float*. It causes an error in platypus. That is why we create a new class for intercepting evaluation function and avoid an error. Put following code in a file called `experiments.py` that creates the class which intercepts results from CEC and normalize them for platypus:

```
import optproblems.cec2005
import numpy as np
from platypus import *
class interceptedFunction(object):
   """ Normalize returned evaluation types in CEC 2005 functions"""
   def  __init__(self, initial_function):
      self.__initFunc = initial_function
   def __call__(self,variables):
      objs = self.__initFunc(variables)
      if isinstance(objs, np.floating):
         objs=[objs]
      return objs
```

3. As previously, we want to execute several times (`nexec`) different algorithms on all CEC functions (`CEC2005`). We just added the fact that each function can be defined on different dimensions (`dims`):

```
if __name__ == '__main__':
    # # USE PLATYPUS EXPERIMENT AND DO STATISTICAL TESTS
    # use single objective functions from optproblems.cec2005
    nexec = 3
    # for all wanted dimensions
    dims = [2]  # 2,10,30,50 are common for all cec functions
    Xovers = [SBX()...]# TO BE COMPLETED
    Mutations = [PM()...]#TO BE COMPLETED
    # build a list of problems for all dimensions
    problems = []
    results = OrderedDict()
    for dim in dims:
        nfe = 500
        for cec_function in optproblems.cec2005.CEC2005(dim):
            #Platypus problem based on CEC functions using our intercepted class
            problem = Problem(dim, cec_function.num_objectives, function=interceptedFunction(cec_function))
            problem.CECProblem = cec_function
            problem.types[:] = Real(-50,50) if cec_function.min_bounds is None else
                               Real(cec_function.min_bounds[0], cec_function.max_bounds[0])
            problem.directions = [Problem.MAXIMIZE if cec_function.do_maximize else Problem.MINIMIZE]
            # a couple (problem_instance,problem_name)
            Mandatory because all functions are instance of Problem class
            name = type(cec_function).__name__ + '_' + str(dim) + 'D'
            problems.append((problem, name))
        # a list of (type_algorithm, kwargs_algorithm, name_algorithm)
        algorithms = [(GeneticAlgorithm, dict(variator=GAOperator(x, m)), 'GA_' + type(x).__name__ + '_' +
type(m).__name__) for x in Xovers for m in Mutations]
        results = results | experiment(algorithms=algorithms, problems=problems, nfe=nfe, seeds=nexec,
display_stats=True)
    print(results)
```

4. Execute previous code and inspect the results. How `results` are structured (problem and algorithm names, best fitness…)? As it is difficult to browse text, it is better to:
    a. Add a breaking point in the code on line that print the results;
    b. Execute the program in debugging mode;
    c. Inspect `results` variable.
5. Previous results contains the final population obtained by the EA but we need to compute the best fitness of this population. To do so, we need to define a new indicator based on class `Indicator` contained in `platypus.core.py`.

```
class bestFitness(Indicator):
    """find best fitness in population"""
    def __init__(self):
        super(bestFitness, self).__init__()

    def calculate(self, set):
        feasible = [s for s in set if s.constraint_violation == 0.0]
        if len(feasible) == 0:
            return 0.0
        elif feasible[0].problem.nobjs != 1:
            raise ValueError("bestFitness indicator can only be used for single-objective problems")
        best = None
        optimum = np.min if feasible[0].problem.directions[0] == Problem.MINIMIZE else np.max
        best = optimum([x.objectives[0] for x in feasible])
        return best
```

6. Then, rather printing results, we would like to compute the best fitness for all final population contained in `results`. For that, we define a list of indicators to be applied and use defined function `calculate` which will compute all indicators on all runs. Execute and have a look. Increase values of `nexec` and `nfe`. What can you observe?

```
    indicators=[bestFitness()]
    indicators_result = calculate(results, indicators)
    display(indicators_result, ndigits=3)
```

7. What is python code for obtaining the best fitness of the Genetic Algorithm based on `SBX` and `PM` operators on benchmark function `F8` in dimension 2 of the second run?

8. What is python code for computing the average best fitness of same algorithm as previously for `F10` function on all executions?
9. From previous structure, we create now a pandas dataframe which allows a lot of manipulations.

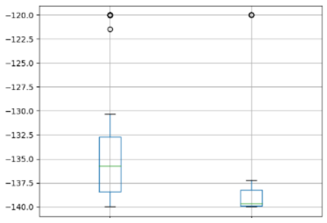
```
import pandas as pd
# create a pandas MultiIndex dataframe
data=dict()
for key_algorithm, algorithm in indicators_result.items():
    for key_problem, problem in algorithm.items():
        data[(key_algorithm,key_problem)] = indicators_result[key_algorithm][key_problem]['bestFitness']
bestFitness = pd.DataFrame(data=data)
print(bestFitness)
```

10. Now, it is possible to manipulate this dataframe as saving to a `csv` file, computing statistics, modifying its structure and plotting.

```
#save dataframe to csv file
bestFitness.to_csv('experiment_%d_runs.csv' % nexec)
# print dataframe statistics
print(bestFitness.describe())
# reverse MultiIndex levels, bestFitness['Algortihm']['Problem'] -> bestFitness['Problem']['Algortihm']
bestFitness=bestFitness.stack(level=0).unstack()
# plot columns concerned by problem F1_2D
bestFitness['F1_2D'].plot()
import matplotlib.pyplot as plt
plt.show(block=True)
```

11. Now, modify and execute previous program by considering 30 different executions on dimensions 2 on SPX and PCX crossovers and PM mutation using 10000 function evaluations with function F8 only. Can you say which algorithm is better?
12. Rename file `experiment_30_runs.csv` to `experiment_30_runs_dim_2.csv` for saving the fact this experiment was done with only 2 variables.
13. Do same experiment as previously but in dimension 10. Do not forget to rename the file. Which algorithm is better?

Exercice 4 Statistical tests



As EAs are stochastic, they may produce different results each time they are executed. To do so, you are supposed to do many simulations on a lot of benchmark problems, store results and "mathematically" find which technique is the best from previous results. To prove it mathematically, you must do *statistical tests*.

1. Rather than executing experiments each time it is needed, remember that we save pandas dataframe after each experiment. Therefore, we can load it without executing corresponding experiment. Create new file called `statiscal_tests.py` and execute following code.

```
import pandas as pd

runs = 30
dim = 2
# load from saved file
bestFitness = pd.read_csv('experiment_%d_runs_dim_%d.csv'%(runs, dim), header=[0, 1], index_col=0)
print(bestFitness.describe())
```

2. Now, using `bestFitness` dataframe, plot boxplots. You should obtain a graph similar to above figure. Can you deduce visually which algorithm is better?
3. Plot boxplot for dimension 10. Now, is it as easier as previously to say which algorithm is better *significantly*?

4. In order to know which algorithm is significantly better, we need to apply statistical tests. For applying those tests, we firstly need to know if the data are normally distributed or not (i.e. data are following a Gaussian distribution).

```
significance_level=0.05
# Test Normality with Shapiro-Wilk test
stat1, p1 = stats.shapiro(bestFitness[bestFitness.columns[0]])
print('stat=%.3f, p=%.3f' % (stat1, p1))
stat2, p2 = stats.shapiro(bestFitness[bestFitness.columns[1]])
print('stat=%.3f, p=%.3f' % (stat2, p2))
if p1 > significance_level and p2 > significance_level:
    print('Normally distributed')
else:
    print('Not Normally distributed')
```

5. Then according to normality, apply adequate statistical test to know whether the means of the two algorithms are significantly different using python code from [Brownlee 2021] webpage.
6. Now, apply statistical tests for same algorithms as previously but in dimension 2.
7. Now, modify and execute previous program by considering 30 different executions on dimensions 2, 10 on two different crossovers and two different mutations. Number of function evaluations is depending on the dimension and defined by $NFE = 2 \times 10^4 \times dimension$ according to [Hellwig *et al.* 2018]. It should take a long long… time.
8. As package optproblems is able to give global optima f^* for each problem, create a new indicator corresponding to the Success Rate (SR). We consider a given simulation a success if it finds some individual x such that $f(x) < f^* + \varepsilon$ where ε is a positive success threshold. SR is the percentage of simulations that are successful.

Exercice 5 Techniques comparisons

By group of 2 or 3 students, compare following techniques using plotting charts, box plots and statistical tests. You also should apply variation on problem dimension (5, 10, 20, 50, 100, 200). If it is a 3-students-group, only consider one proposition where there is at least 3 techniques to compare:

1. When using a genetic algorithm and binary coding, compare classical binary coding and Gray coding.
2. When initializing population, compare random sampling and Latin Hypercube Sampling. If you are 3, consider also orthogonal sampling.
3. When selecting individuals, compare Tournament selection with Roulette Wheel (also called fitness-proportionate) selection. If you are 3, consider also Stochastic universal selection.
4. When selecting individuals, compare Split rank selection and Reward-based selection. If you are 3, consider also Linear ranking selection.
5. Compare Genetic Algorithm and Differential Evolution. If you are 3, consider also SMPSO.

Sources

- Antonio LaTorre, Daniel Molina, Eneko Osaba, Javier Poyatos, Javier Del Ser, Francisco Herrera. *A prescription of methodological guidelines for comparing bio-inspired optimization algorithms*, Swarm and Evolutionary Computation, Volume 67, 10.1016/j.swevo.2021.100973, 2021.
- Shahin Rostami, Using a Framework to Compare Algorithm Performance, 2019, https://datacrayon.com/posts/search-and-optimisation/practical-evolutionary-algorithms/using-a-framework-to-compare-algorithm-performance/.
- COCO: A platform for Comparing Continuous Optimizers in a Black-Box Setting, http://arxiv.org/abs/1603.08785, http://numbbo.github.io/coco-doc/.
- Michael Hellwig, Hans-Georg Beyer, *Benchmarking Evolutionary Algorithms For Single Objective Real-valued Constrained Optimization - A Critical Review*, 2018, https://doi.org/10.1016/j.swevo.2018.10.002.
- Jason Brownlee, Statistical Hypothesis Tests in Python, 2021, https://machinelearningmastery.com/statistical-hypothesis-tests-in-python-cheat-sheet/.