Artificial Intelligence - CET1

Genetic and evolutionary algorithms

Juan José Rodríguez Aldavero
May 16, 2020

Contents

1 Genetic algorithm presentation and single objective optimization

In this deliverable exercise we are going to apply a genetic algorithm to the study and optimization of a real power supply distribution problem. To do this, we will use the DEAP library and we will treat a dataset that represents a raster map (1000×1000).

In particular, the genetic algorithm will be used to optimize the location of the supply stations by satisfying a set of objectives. The single-objective problem (minimizing the average distance between stations and urbanizations) as well as the multiobjective problem (adding other objectives, such as minimizing the standard deviation or the distance to the main station) will be discussed.

Genetic and evolutionary algorithms are very suitable for this type of optimization problem because of their high degree of flexibility. With a convenient representation, it is not necessary to know details of the problem in question to apply an optimization with this type of algorithms. This is because genetic algorithms fall into the category of metaheuristic algorithms, which are characterized by requiring only an evaluation function for the data set (Talbi 1).

We start by exposing and understanding the genetic algorithm from the following questions:

A. **Explain what an individual is in the context of this problem, how it is represented, and fill in the gaps for the configuration functions.**

   In this problem, the individual is the set of 20 positions for each of the service substations. Each of these positions is represented by an integer $n[0, 4832]$ or by a tuple (x, y) where x, y are the positions in the substation raster map. The relationship between these two representations is given by a map defined in the code by the function 'text values' to 'coordinates'. For simplicity, in the genetic algorithm we use the representation by integers.

   The population is a set of $n_{population}$ individuals, who in evolutionary language fight among themselves and against the environment to survive, and where survival is decided on the basis of the objective function or 'fitness' that depends on a set of criteria defined for the problem.

   We proceed to fill in the needed code for the different functions. What we have done is to complete the *toolbox.register* method to create the individuals in representation of permutations, as well as complete the selection, mutation and evaluation procedures. The most interesting change has been the definition of the representation of individuals. We define the attributes associated to each individual as a list of 20 integer indexes that can vary between 0 and 4831. To do this, we use the methods of the DEAP library as seen below.

```
IND_SIZE = 20
POP_SIZE = 4832

toolbox.register('indices', random.sample, range(POP_SIZE), IND_SIZE)
toolbox.register('individual', tools.initIterate, creator.Individual,
toolbox.indices)
toolbox.register('population', tools.initRepeat, list,
    toolbox.individual)
```

This allows us to define individual type objects initialized in the representation previously defined.

B. **Configure the genetic algorithm with the different combinations of the following values for the variables:**
We solve the problem by varying the hyperparameters of the algorithm to study its effect on the solution. In particular, we vary the population sizes, the number of generations and the probabilities of mutation and recombination.

We proceed by studying the changes of the hyperparameters keeping the rest constant, in order to see the isolated influence of each parameter. Only a group of the graphs obtained are attached in this work to improve the clarity of the exposition, but they will be attached as an annex for consultation if desired.

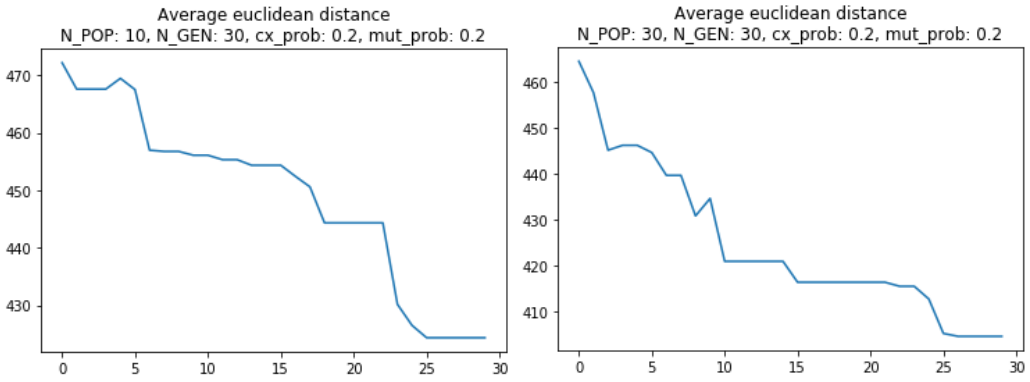


Figure 1.1: *Solution varying the population*

We see how by varying the population we get a faster rate of improvement, as well as a better performance. This makes sense since an increase in population increases evolutionary pressure and decreases the likelihood that poorly adapted individuals will be able to survive and pass on to the next generation.

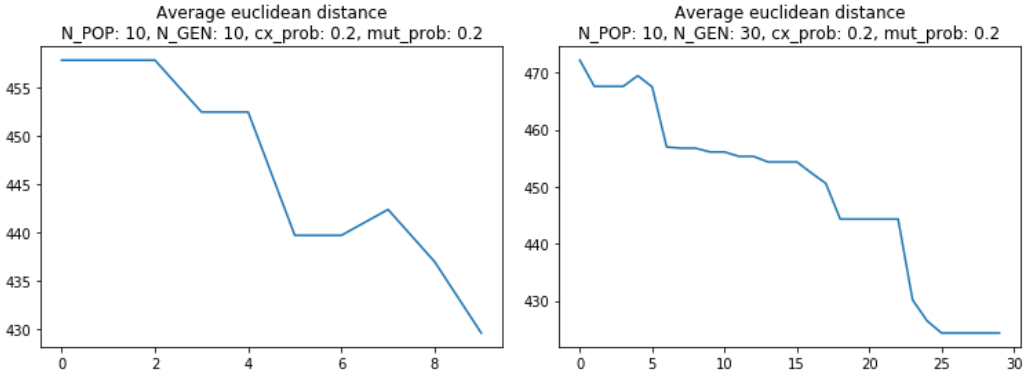


Figure 1.2: *Solution varying the number of generations*

We see how the increase in the number of generations leads to an improvement in the final quality. This is logical since it increases the number of occasions in which the processes of recombination and mutation occur, as well as those of selection by maintaining the evolutionary pressure for longer.

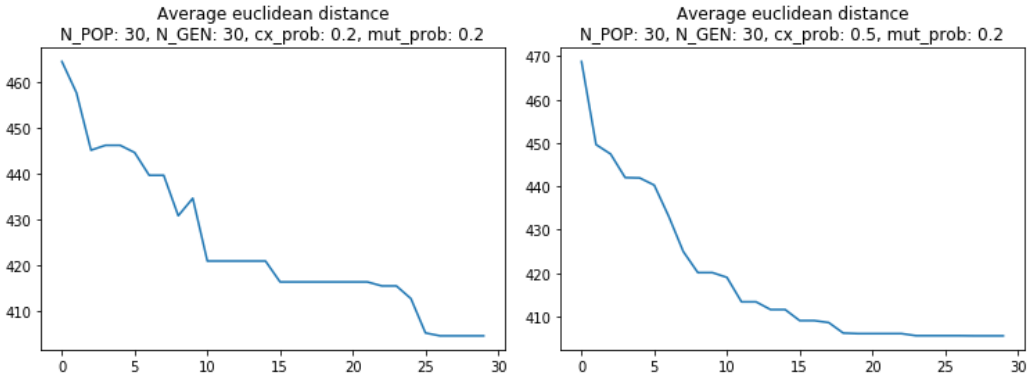


Figure 1.3: *Solution varying the recombination probability*

An increase in the probability of recombination slightly increases the speed of process improvement. In terms of metaheuristic optimization algorithms, an increase in the recombination rate increases the exploitation rate of the algorithm at the expense of explorability (exploration vs exploitation tradeoff). This means that an increase in this rate, unlike the mutation rate, allows a greater exploitation of the set of initial individuals (population) because it increases the rate of combinations between them, at the expense of reducing the exploration of the solution space outside these initial individuals.

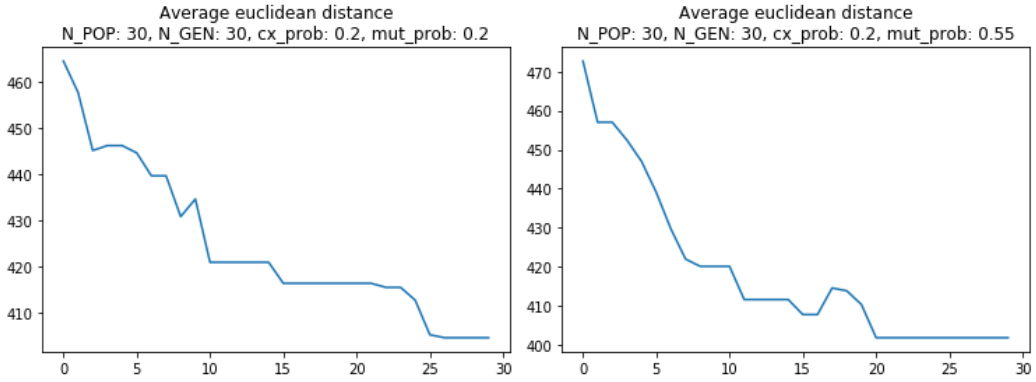


Figure 1.4: *Solution varying the mutation probability*

Again, as in the previous case, we obtain a sensible improvement of the solution when reaching the last generation as well as the speed with which it is reached. The algorithm becomes more explorative at the expense of the exploitation of its solutions, since mutations are random changes in some of the individual's genes, giving rise to completely new individuals. For this particular problem we see that this increases the performance of the algorithm, but it does not seem something easily extrapolated to other cases.

C. **Discuss the effect of each of the variables as observed after the previous section.** From these graphs we see how, in general, the increase in these parameters increases the quality of the solutions. For example, for the case with minimum hyperparameters (*N_POP*: 10, *N_GEN*: 10, *cx_prob*: 0.2, *mut_prob*: 0.2) we obtain a final result of approximately 430. On the other hand, in the case of maximum hyperparameters (*N_POP*: 30, *N_GEN*: 30, *cx_prob*: 0.5, *mut_prob*: 0.55) we obtain a result close to 410 points.

In summary, the effects are as follows:

- **Population**: in this case the convergence speed of the algorithm is improved at the cost, possibly, of greater computational resources.
- **Number of generations:**In all cases the final quality of the solution is increased, but not the speed, at the cost of more computational resources (calculation time).
- **Recombination rate**: increases the degree of exploitation of the initial population, which increases the probability of getting stuck in a local minimum and therefore decreases the probability of reaching good long-term solutions. In return, the performance of the algorithm is improved in the short term.
- **Mutation rate**: increases the degree of exploration of the algorithm by giving rise to new individuals, at the cost of potentially worsening its quality in the short term. It increases the probability of obtaining more adapted individuals in the long term, although it may decrease the performance in the short term.

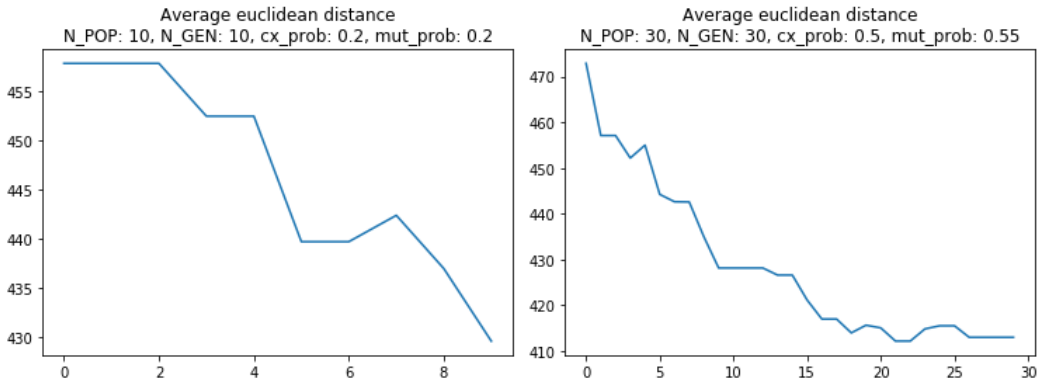


Figure 1.5: *Solution with greater and lesser hyperparameters*

Below we see two particular cases that present interesting properties:

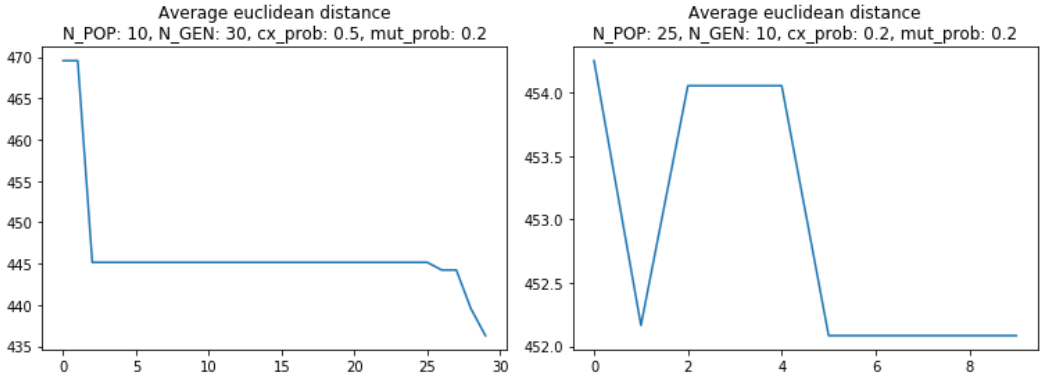


Figure 1.6: *Particular effects of genetic algorithms*

In the first case, we see how the genetic algorithm remains without presenting improvement in the quality of individuals a large number of generations (22) . This is characteristic of these algorithms because it is based on random procedures, and the increase or decrease in the quality of individuals is linked to whether this is achieved from the mutations or recombinations made. In the image on the right we see that not only is no improvement achieved, but on one occasion this is significantly worsened. This is because in this particular algorithm the best solutions are not stored in memory (elitism), and therefore an unfortunate mutation or recombination may worsen the best available solution.

Below we see two of these solutions with respect to the same random set of urbanizations.

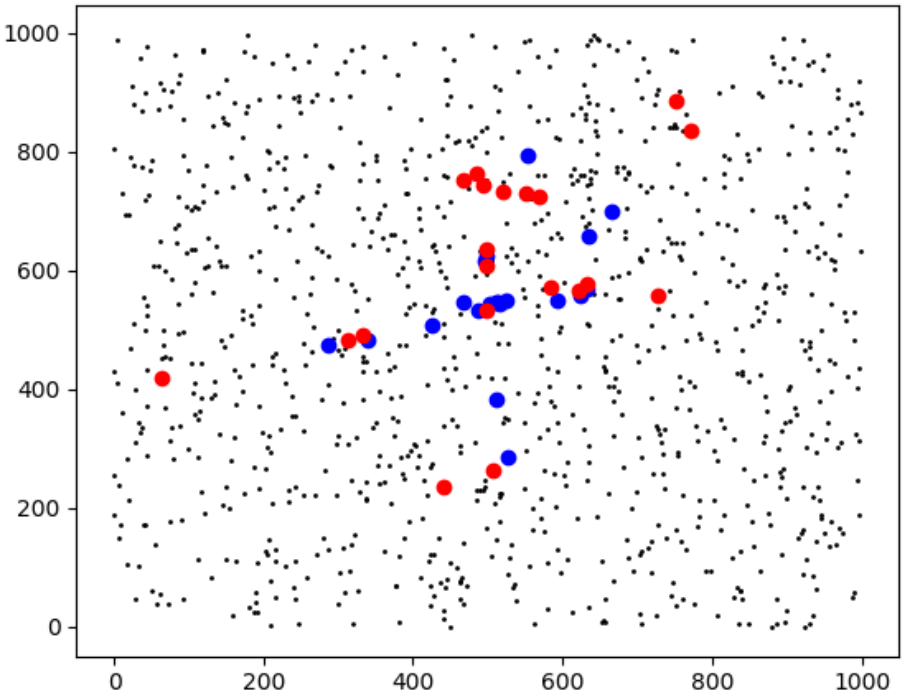


Figure 1.7: *Final distribution of service stations*

The set of black dots represents the random distribution of urbanizations. The set of blue dots represents a good solution, with fitness $fit_1 = 406.41$; while the red dots represent a somewhat worse solution, with $fit_2 = 438.60$. We see how the points are located near the center since any point far from it would increase the average distance considerably.

2 Multiobjective optimization

We continue the study by applying a multi-target optimization for the following extra criteria:

- Minimize the standard deviation of the distances
- Minimize the average distance from all substations to the main station $(0, 0)$.

This will be summarized, essentially, in modifying the objective function to involve these extra criteria. To do this, we define two extra functions *stdev_distance* y *avg_distance_to_main* and incorporate them into the objective function *fitness*. We proceed to adapt the code to support this multi-target optimization.

A. **Complete the functions *stdev_distance*, *avg_distance_to_main* y *fitness* of the file *utils.py***
The function for the standard deviation will take the same form as the function for the average distance except for the use of the *np.std()* instead of *np.mean()*. For the distance to the main station (0.0) it is sufficient to reuse the function for the average distance by replacing the positions of all the developments (vector d) with the zero location. As in Python the union of tuples is done by the + operator, it is enough to add the different functions in the function *fitness*.

```
def fitness(individual, urbanizations):

return avg_distance(individual, urbanizations), +
    stdev_distance(individual, urbanizations), +
    avg_distance_to_main(individual, urbanizations),
```

B. **Modification of the code for the multi-target function and the function weights *avg_distance, stdev_distance* y *avg_distance_to_main***
e have to incorporate the three weights in the *creator.create()* method of the library by incorporating them in the same tuple, since the three criteria are of minimization.

```
creator.create('FitnessMin', base.Fitness, weights=(-1.0,-0.8,-0.5,))
```

All these changes can be checked in the attached codes. Once done, we check that the code works correctly and show a result for 10 generations with a population of 10 individuals and a probability of mutation and recombination of 20%:

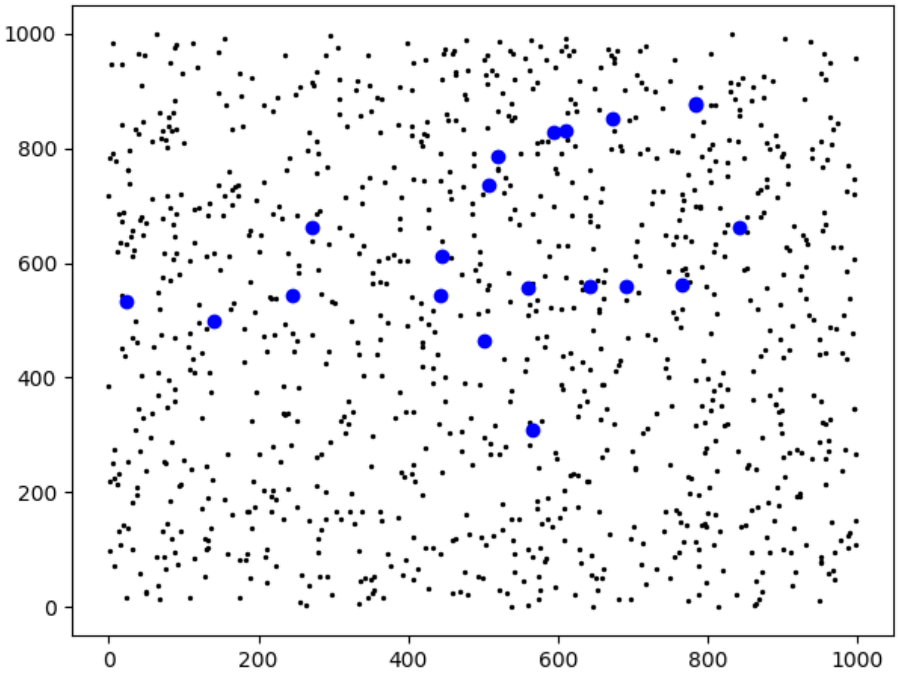


Figure 2.1: *Final distribution of service stations for the multi-target function*

Looking at the graph we see how the distribution of stations has a similar shape to the distributions for the single-target function, although curiously they seem to be more dispersed and far from the center (even though one of the criteria introduced in the evaluation function is to minimize this distance). The values of the multi-target function obtained are an average distance *avg_dist* = 469.55, a standard deviation *stdev_distance* = 217.41, and an average distance to the center of *avg_dist_to_main* = 851.02.

3 Questions about the algorithm

We examine the details of the genetic algorithm used from the following questions:

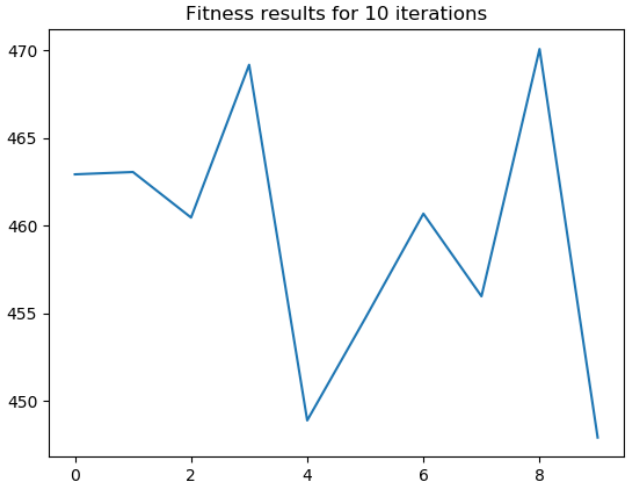
A. **Considerations on the construction of the distribution of urbanizations**
Given the code for building the locations of the 1000 urbanizations:

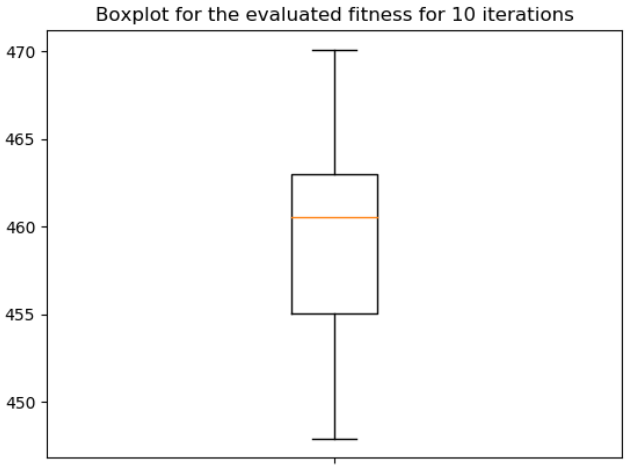
```
seq = [i for i in range (0,999999)]
rand_urbanization_map = sample(seq,1000)
```

we are asked about the implications of changing the distribution of urbanizations when evaluating the results of the algorithm. It is clear that a change in the locations of the urbanizations would have direct effects on the results of the optimizations, since the evaluation of the fitness function is based on the average distance of the stations from these urbanizations. For example, there may be some distributions for which there are solutions with a very small average distance, while other distributions may not have solutions with such a small distance.

This may make it appear that some genetic algorithms have a better performance than others when it is not, so it seems advisable to fix the distribution of urbanizations in advance when comparing genetic algorithms (or any other optimization algorithm).

B. **Causes of the low variance for the speed of convergence and fitness found between iterations** If we run the same algorithm configuration several times, we can see the variance found in the optimization results. To check it, we run the simplest configuration (N_POP = 10, N_GEN = 10) 10 times and represent in a graph the variation of the final quality of the solution.





We obtain a final variance of $\sigma = 51.29$, which is around 11% of variance with respect to the mean of the solutions. This is not much, although it is not neglible. We have done all this with the same distribution of urbanizations, although for each execution the initial distribution of individuals is random. This makes the origin of this low variance a bit mysterious, since the speed of convergence as well as the final result of the solution should be highly dependent on this initial distribution. The cause for this low variance might be that given a random set of initial individuals, the probability of improvement given a random variation through mutation and recombination is almost constant.

C. **Advantages and disadvantages of using binary coding for individuals** If binary coding were chosen instead of integer coding, each individual would be represented by a matrix of size (4832×20). This would make operations with these elements very difficult as they would require more computational effort as well as a larger amount of memory for storage. On the other hand, it is possible that it would facilitate the operations of mutation and recombination

D. **Explain the main differences between the functions *toolbox.initIterate* and *toolbox.initRepeat*** These two methods are integrated into the DEAP libray in order to specify the behavior of each data type (individuals, populations, etc.). In other words, each data type has a set of operations associated and this two methods specify how to perform the operations.

toolbox.initIterate(container, func, n) calls the function *func n* times and then stores the *n* executions in the container. For example:

```
initRepeat(list, random.random, 2)
#Output
>>> [0.6394..., 0.0250...]
```

On the other hand, *toolbox.initIterate(container, generator)* calls the function *generator* and then stores its output in the container only once. Therefore, the output of the *generator* function has to be an iterable (list, tuple, etc.). For example:

```
gen_idx = partial(random.sample, range(10), 10)
```

```
initIterate(list, gen_idx)
#Output
>>> [1, 0, 4, 9, 6, 5, 8, 2, 3, 7]
```

E. **Decoration of individuals**

We see how the algorithm incorporates in a preventive way a decoration procedure applied on the recombination and mutation operations. This is especially useful when there are constraints on individuals in the population that can be violated by these procedures. For example, if our representation is a list of floats $x \in (a, b)$, and we apply a Gaussian mutation procedure, then there is a considerable probability of obtaining elements that may be outside the a and b limits. The decoration allows to correct these occasions without the need to explicitly modify the mutation and recombination procedures (which are included in the library itself and are not explicitly seen).

In our case, due to the representation (list of integers between 0 and 4831) and the recombination and mutation procedures used, decoration is not necessary because it is not possible to obtain values outside these limits.

4 Decoration function

We proceed to apply the decoration function *fulfill_constraints*, which is initially empty, to the problem. To do this, we proceed to complete it.

A. **Before completing the function, reason out what aspects of our individuals make decoration necessary.** Decoration is a procedure by which you can change the result of a function without explicitly changing it. This is a very common and convenient procedure in Python, and for this project it can have useful applications e.g. when controlling with a high degree of precision the operations of the genetic algorithm.

   For example, if we are dealing with a problem whose representation for the genetic algorithm presents constraints, we can apply decorators to restrict the operation of some methods to respect these constraints. This may be useful, for example, to prevent mutation and recombination procedures from violating the constraint imposed on individuals.

B. **Complete the function *fulfill_constraints*.**
   We proceed to illustrate this example by completing the decorator called *fulfill_constraints* with respect to a constraint imposed on the individuals. In particular, we consider the constraint imposed on the offspring of the population, so that each of the genes of the offspring can take on a minimum and a maximum value.

   For the case at hand, essentially what we have done is copy the information contained in each child, impose upper and lower bounds (a, b) on each gene of the new children and then copy back this new information in the original children. We do it redudantly so that it is easier to read and modify.

```
#Copy the information of the children into a new variable
new_child = np.array([c for c in child])

#Modify the new children imposing the lower and upper bounds on each
    gene
for i in range(len(new_child)):
if new_child[i] > b:
new_child[i] = b
elif child[i] < a:
child[i] = a

#Copy back the information into the original children
for i in range(0, len(child)):
child[i] = new_child[i]
```

   In our case, as we have commented before at first sight it seems that the algorithm works correctly without the need of decoration. If, for example, we were to use another type of mutation method (e.g. Gaussian mutation) then we would have to take into account the binding of minimum and maximum values for each gene.