Achraf Aït Sidi Hammou - Jérome Roche - Zoubaïr Tazakkati

Gitlab : https://gitlab.data-ensta.fr/ait-sidi-hammou/in104_ga_tazakkati_zoubair-roche_jerome-aitsidihammou_achraf

Problem: In this project, we try to solve an NP-complete problem: **Given a set of integers S**, is there a subset whose sum of the elements is 0 ? If it exists such a subset, what is the largest subset verifying this property ? We answer this problem by implementing a genetic algorithm.

A / First implementation

According to the formalism of genetic algorithms, we call:

-Population: a group of individuals
-Individual: any subset of S. For instance, if S = [7, -6,9,5, -3, -7] then [5, -3] is an individual.
-Allele: an element of S
-gene: an index of S

I) Main idea of the first implementation

We give ourselves **a size L** for the individuals. We generate a population of individuals of this size : It is the generation 0. We test if one of the individuals of this generation is a solution. If it is, we stop the algorithm and the problem is solved. Otherwise, we use the evolve method to create a new population : the generation 1. This new generation is constructed so that its individuals are more likely to be a solution. We retest all the individuals in this population and so on.

If at the MAX_GEN-th generation, there is still no solution, then we consider that there is no individual of size L wich is solution. Then we repeat exactly the same process but with individuals of size L-1. By starting with individuals of maximum size (ie the same size as S), we thus check all possible sizes of individuals

II) Study of the Code

a) The class of individuals: Individual

It is the class that defines the attributes and methods of our individuals.

Methods of the Individual class

Selection : calc_fitness(self)

For our fitness function we want that evaluate_fitness (A) = 1 if A is a solution and that evaluate_fitness (A) -> 0 when sum (A.set) -> +/- infinite. Therefore, we chose the function :

$$\text{Individual} : --> \frac{1}{1 + sumOfElementOf(Individual)^2}$$

It immediately checks our requirements. We also chose this function for its algorithmic simplicity. Indeed this method will be used a very large number of times, thus it must be inexpensive in terms of operations.

Cross-over : crossover_individuals (par1, par2)

We have implemented this function quite simply. From two individuals (called) P1 and P2 (necessarily of the same size), we create a new individual C (called child) such that the allele number i of C is either the allele number i of P1 either the allele number i of P2 with probability 0.5 in both cases. However, we had to be careful: If we had i and j such that P1.set [i] = P2.set [j] then the child could poentially have had the same allele twice, which should be impossible. Thus, we implement the quickSort method (self, low, high) (itself using the partition_indiv method (self, low, high)) in order to sort the lists of the allèles of P1 and P2. Indeed if the lists are sorted, the problem we just noticed is impossible.

Mutation : mutate_individuals (self)

It is a method which randomly modify an allele of an individual. Once again,we should to be careful and to not replace an allele by an other allele that was already present. For this we simply check that the allele that we are trying to add is not already present, otherwise we are looking for another.

b) The class of populations: Population

There is one main method in this Class :

Evolution : evolve (self)

This is the fundamental method of our algorithm and the code is essentially based on it. If the current population does not contain an individual that is a solution to the problem, evolve generates a new population in which we will be more likely to find the solution. It is based on 4 steps:

***) First, we choose the 10% best individuals (it is not difficult because they are sorted according to their fitness, that's why we sorted them) to which we apply mutations. Thus we get 10% of the new population.

***) Then we create a "roulette". Let’s explain how we create it.

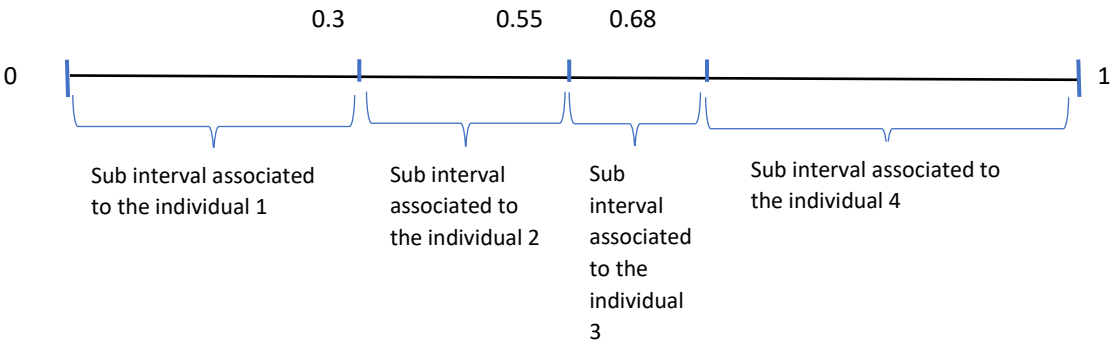
To each individual A of the population, we associate an interval [Xa, Ya] contained in [0,1] according to the following rules:

-If A and B are two different individuals then [Xa, Ya] and [Xb, Yb are disjointed.

-if calc_fitness (A) <calc_fitness (B) then Ya-Xa < Yb-Xb

In other words, roulette is a partition of the interval [0,1] : the interval is separated into subintervals and each of these sub intervals is associated with an individual according to its fitness : the better it is the greater the subinterval is.

For instance, this is a roulette for 4 individuals such as calc_fitness(Indiv_3) < calc_fitness(Indiv_2) < calc_fitness(Indiv_1) < calc_fitness(Indiv_4) :

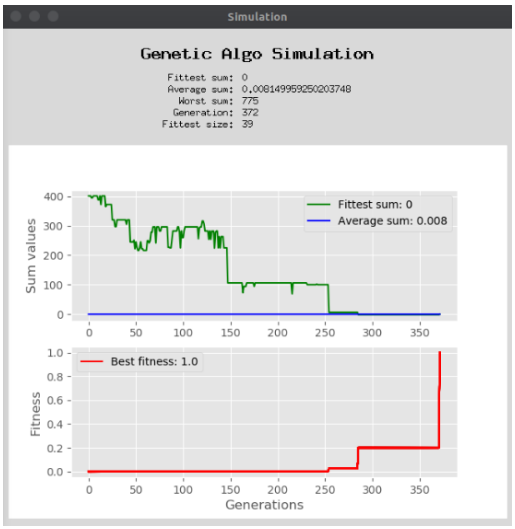


Thus, from this roulette we generate 80% of individuals of the new population thanks to the cross-over. We randomly draw 2 numbers a and b between 0 and 1, thanks to the roulette we can know the sub intervals associated with which individuals they fell. Then we do a cross over between these two individuals to get a new individual

***) Finally, the last 10% are generated completely randomly; they are individuals unrelated to those of previous generations

III) Results with this algorithm

This algorithm gives quite good results for sets S that are not too large. However, if S becomes too large, it becomes very slow and may not end. Indeed, the fact of having to sort each individual before the crossover over notably slows down the program. And if S is too large (tipically for 500 elements it is already very slow and greater than 1000 elements the algorithm crash), we can even have a stack overflow error. Here is the result of our simulation for the small fichier (40 elements). It considers that the greatest subset is a 39 elements subset.



B / Second implementation

To improve and to solve the weakness of the previous algorithm, we have adopted a new way to implement the algorithm.

I) Main idea of the 2nd implementation

Our individuals are no longer real subsets of S. Now, these are lists of same size that L and:

If S[i] is an allele of our individual A, then A[i]=1, else A[0]=0. For instance if S = [7, -6,9,5, -3, -7] then [0,1,0,0,1,1] is an individual which represent the subset [-6,-3,-7].

Thus, we no longer need to iterate over the size of the individuals. It is much more efficient. Indeed, we just generate generations and we search if one of its individuals is a solution, if it is, we solve the problem, else we generate another generation. We also no longer need to sort our lists to avoid some issues during the cross-over (we mentioned earlier that it was because of our way of representing our individuals that we had to sort our individuals to do the cross-over).

We also modified some methods.

II) Study of the new Code

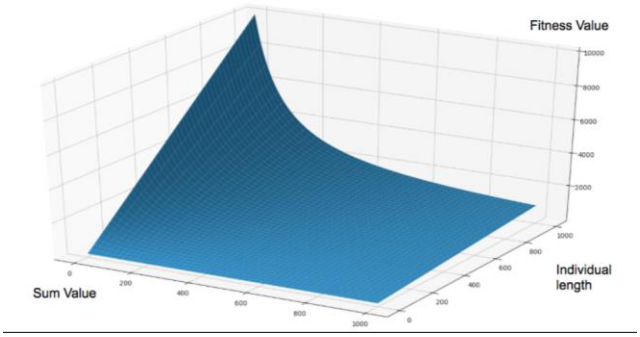
a) The class of individuals: Individual

Methods of the Individual class

Selection : evaluate_fitness (self)

Now, our individuals **are all the same size**, we must adapt the fitness function to make it considers the size of the individual. Indeed, a good individual is an individual whose sum is close to 0 but also (and above all) an individual who represents a large subset of S. Thus, we have chosen the function

$\text{Individual} : --> \frac{len(input_{array}) * numberOfElementOf(Individual)}{100+|sumOfElementOf(Individual)|}$ (the constant 100 at denominators is chosen empirically). We can represent this function :



We notice that this fitness function is more related to the length than to the value of the sum : that's what we wanted.

Cross-over : crossover_individuals (par1, par2)

Crossovers are no longer done randomly for each index. Now we choose randomly "cross-over points", and we choose randomly large portions of alleles on each side of these points to git it to the child. Let us clarify this with an example:

Individuals P1 : 0 1 0 1 1 1 1 1 0 0 0 1 0 0 1 0 1 0 0 1
Individuals P2 : 1 0 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 1 0 1

We choose cross over point randomly :

Individuals P1 : 0 1 0 1 1 1 * 1 1 0 0 0 1 0 * 0 1 0 1 0 0 1
Individuals P2 : 1 0 0 0 1 1 * 1 0 0 1 0 1 0 * 1 0 0 0 1 0 1

Then we cross over to get a child :

Individual C : : 0 1 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 1

Doing that, our model of cross-over is closer to the real cross-overs that take place in the ADN. Also,
we have a better control of the variation of alleles between parents and child.
Also, thanks to the new representations that we have of individuals, there is no longer any risk of taking 2 times the same allele if the 2 parents have one allele in common and without having to sort

b) The class of populations: Population

Evolution : evolve (self)

First we define two way to choose randomly an individuals from a population (this is one of the parameter of our simulation : we choose which one we want when launching the simulation) :

-The roulette method : we pick a number randomly between 0 and 1 and associated it to an individual thanks to the roulette we defines earlier.

- **The system of « tournament » :** we pick randomly 2 individuals from the population. However, among these two individuals, we keep the one with the best fitness **with a probability win_rate** : he is the winner.

Then we define a probability p less than 0.5 that we call **renew rate**. It is this parameter which will define the different ways of creating new individuals. Now we can renew our generations following these steps:

***) (renew_rate*100)% of our new_population is taken from the previous one using the methode of selection we chose when we launched the simulation (roulette or tournament)

***) (renew_rate*100)% are generated completely randomly; they are individuals unrelated to those of previous generations

***) we also keep the individual with the best fitness (the fittest) of the previous generation

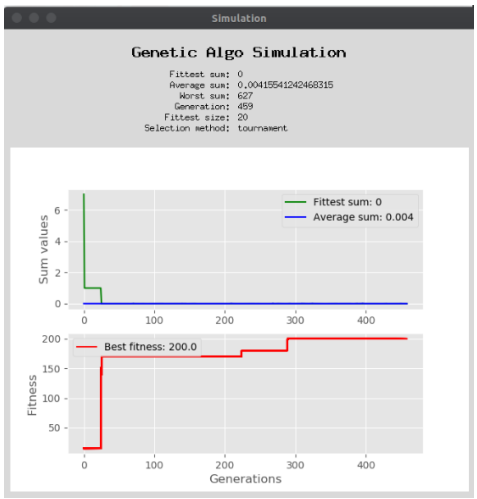
***) Then we complete our new population thanks to cross over : we use one of the two methods of selection (roulette or tournament) to choose the parents of the child. Then we apply mutations to the child.

III) Results with this algorithm

Thanks to the modifications that we have made, the new algorithm is now much faster even for large sizes of S.

For instance, we show that for a set S of 100 elements, the previous algorithm takes 9.4s to generate a new generation whereas the new algorithm takes only 0.25s. Moreover, it is now able to manage with great set S with for instance set S that contains more than 100 000 elements and to gives us solutions (it took 70s to generates a new generation while the previous algorithm crashed for a set of 1000 elements).

However, it loses accuracy. Indeed, he finds many solutions, but these are not always the optimal solutions. For instance, here is the results for the small fichier:



We can see that it considers that the solution is a 20 elements subset whereas the previous algorithm found a 39 elements solution. It shows us that this algorithm does not always find the best solution, particularly if the set S is small.