TP7 : Genetic Programming

1 Introduction

In this TP we are working on genetic algorithms. Those algorithms are based the way a population will adapt to an environment through genome crossing and mutations. The genetic algorithms uses the same process to find a solution to an optimisation problem.

We have a dataset with each value composed of five binary values and we want to find an optimal function to this dataset. The four firsts values of an element are the data x_1, x_2, x_3 and x_4 and the last one is the expected output of the function applied on the data.
The function uses a stack and are represented by a list (of fixed length) of operations that will modify the stack.Those operations picked from the following list :

X_i : those are simply operations that put the binary value x_i on the stack.

AND : this operation reads the two first values on the stack and pushes the result of the logical AND between those values on the stack.

OR : this operation reads the two first values on the stack and pushes the result of the logical OR between those values on the stack.

XOR : this operation reads the two first values on the stack and pushes the result of the logical XOR between those values on the stack.

NOT : this operation reads first value on the stack and pushes the result of the logical NOT on the stack.

2 Implementation

The implementation of this problem can be found in the file `gp.py`

The genetic algorithm repeats the same three steps for a fixed number of generations (variable gen). Those three steps are selection, crossover and mutations.

2.1 Selection

This step acct a bit like natural selection, we choose the individuals best adapted to the environment and let die the less adapted. In this TP we use the 2-Tournament method (K-Tournaments with $k = 2$), we choose randomly two individuals and keep the one with the best fitness. We repeat this N times to have a population of the same size as the previous.

2.2 Crossover

This step corresponds to the crossing of genomes. We take the individuals two by two and exchange their second half with a probability p_c.

2.3 Mutation

This step corresponds to genome mutations. For each element of each individual we have a probability $1 - p_m$ to replace the element by an other operator.

The crossover and the mutation steps allow to create new individuals and give a chance to find an individual with a better fitness, but the problem is that it could also create individuals that are not executable.

2.4 Initial population

Before applying the above steps we have to create an initial population by creating N random individuals.

3 Results and

My function *random_prog* does not always work (due to a RecursionError) and since I could not create a random initial population and didn't have time to fix it I could not test my implementation. In order to fix that I should modify my function, instead of creating a completely random program I should put restrictions to make sure that the program created is valid. For example [$"X1", "OR", "AND"$] is not valid since $"OR"$ operation need to use two values on the stack and only $"X1"$ was push (and we have the same problem for $"AND"$).

3.1 Parameters impact

$progLength$: The length of the programs impacts the number of operators that will construct our function. A too short length will allow us to create only basic functions and in most case our best fitness will still be low. But a too large length won't allow us to create compact functions and we would tend to have over complicated boolean expressions. In the next sub section we discuss the possibility to use programs of variable length.

p_c : This parameter impacts the number of crossover that we will have. Crossover can destroy interesting individuals as well as create new unexplored possibilities. We don't want too much crossover to occur to avoid destroying all the interesting individuals, but with a too low probability we won't create much new individuals and have less chances to really improve the best fitness over the generations.

p_m : As for the crossover mutation can both destroy interesting individuals and create new ones. Without them we would not be able to increase the best fitness and they provide a lower bound of the growth of this fitness over the generations. But as before too much mutations would tend to destroy too much the interesting individuals and risk to make the best solution disappear from the population.

3.2 Programs of variable sizes

Here we use programs of a fixed length but using variable program sizes could also be interesting since it could allow us to find more compact solutions or in other case to create a longer program that could have a better fitness. To allow this possibility we should slightly modify the implementation. First we should create a parameter *max_length* to make sure that the program length does not increase to much. It could also be interesting to try to add a simplification function that would reduce the programs (but without modifying the result of the boolean expression) by removing useless terms (for example $"X1", "X1", "AND"$ could be simplified as $"X1"$). But we would have to compare results with and without this function to make sure that this kind of simplifications does not affect negatively the performance of the algorithm.

We also would have to slightly modify the algorithm to allow individuals of various length. The selection step can stay the same since we are only interested in the fitness of individuals and their length does not affect the 2-Tournament.
For the crossover step we have different possibilities. We can decide to continue to exchange the second half of both programs, taking into account that both programs might not have the same length and therefore not the same half length. But this would shorten the longest individual and expand the shortest and over time the individuals will tend to all have similar length. An other option would be to preserve each individual length by exchanging only a part as long as the half of the shortest program of both.
For the mutation step the implementation would steel work on a population with various individual lengths (since for each individual p we have a for loop in range length of p). But we could try to adapt the probability p_m to the size of p so that longer programs doesn't have more chance to have a mutation.

Finally we would have to decide how to create the initial population, with completely random sizes of individuals or with various fixed length. We could also start with individuals of the same length and introduce the size variations by randomly choosing the size of the parts that we exchange during crossover.