Université de Genève

Metaheuristics for optimization

TP 7: Genetic Programming

Author: Joao Filipe Costa da Quinta

E-mail: Joao.Costa@etu.unige.ch

December 31, 2021

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Département d'informatique

1 Introduction

During this TP we will be working with an algorithm that tries to simulate genetic evolution. In biology a population adapts to an environment by pure luck. This luck is due to the crossing of genomes, and by mutations in this genome. This is a very long process. Genetic algorithms try and apply the same logics and methods, but faster!

2 Problem to optimize

During this TP we will try to put together a simple function, that is the most optimal to a given dataset.
This dataset is made up of 16 values. Each value is an array of 5 binary values. The first 4 binary values represent the data itself, and the 5th value represents the expected output.
Let's say that $[1, 0, 1, 0, 1]$ is a value of the dataset, then the first 4 values $1, 0, 1, 0$ are the input to the function, and the 5th value 1 is the expected output.
This function will be represented by and array of given length, and will only allow certain simple operations to take place.
the results of each operation will be saved in a stack.
The operations are the following:

"X_i" pushes to ith bit of the data to the stack

"AND" consumes the 2 last values of the stack, and performs AND logic operation on the consumed values. Then pushes the result onto the stack

"OR" consumes the 2 last values of the stack, and performs OR logic operation on the consumed values. Then pushes the result onto the stack

"XOR" consumes the 2 last values of the stack, and performs XOR logic operation on the consumed values. Then pushes the result onto the stack

"NOT" consumes the last value of the stack, and performs NOT logic operation on the consumed value. Then pushes the result onto the stack

3 Implementation of the algorithm

First we need to create an individual, or a function, this is done randomly, however we have to be careful as we have to make sure the function is doable, for instance if the first operation of a function is "AND" it wont be possible, first we need to push vales from data into the stack to be able to perform the operation "AND".
Let's set the size of the program to 5 operations, then the following program is valid:

$$Program := [\text{"}X1\text{"}, \text{"}X2\text{"}, \text{"}AND\text{"}, \text{"}X3\text{"}, \text{"}XOR\text{"}]$$

We push the value X_1 then the value X_2, we perform the operation "AND" on the last 2 values pushed to the stack, and push the result of the operation, we then push "$X3$", and finish by performing "XOR" in the 2 last values, and push the result. We have to make sure that the stack now contains only 1 value, if it contains more than one, then the program isn't valid.

3.1 Selection

The goal is to choose the best individuals that get to 'live on', and the worst ones we let them 'die'. There are many ways of doing this. The way we choose to do this in this TP is the k-Tournament, this is a really easy step to selection method to understand. From the whole population at generation t, we chose k individuals at random (all have the same probability of being chosen), from the individuals, we keep the one with the best fitness. We do this N times, so that the number of individuals is constant through the generations. For this TP k = 2.

3.2 Crossing and Mutation

These functions were given to us.
Crossing cuts an individual in half, and mixes it with the other half of another individual.
Where as Mutation just changes an operation by another.
I would say that both of these are bad ways of doing it, simply because these new individuals might not represent an executable program. Which means that the population of a new generation might have individuals that are simply not useful.

Personally, I would have skipped the crossing step, and kept the mutation step, by allowing only specific modifications, mutations that are interchangeable, for instance we can change any "X_i" by any other "X_j" where $j! = i$, whilst keeping the validity of a program. (operations "AND", "OR", "XOR" are also interchangeable).

3.3 Algorithm

(1) Generate random population of size N

(2) For $generation\_max$ do the following steps

(3) Select Individuals, as show in section 2.1

(4) Compute crossing, as show in section 2.2

(5) Do mutations, as show in section 2.2

(6) After doing this for $generation\_max$, return best fit individual.

4 Results

The general algorithm works.

For our tests: N=1000, $P_c = 0.6$, $P_m = 0.1$, $max\_generation = 100$.

First I tried with a program of length 5. For a program of length 5, the best result was 13 of fitness consistently. With 100 or 500 iterations.

Then I attempted an longer program, for instance of length 10, the results were better. With 10 attempts the average was of fitness 15.1, the best programs reaching the maximum of 16 fitness.
This program has the perfect fitness for our dataset:

$$['X4','X1','X2','AND','X2','X3','XOR','OR','NOT','AND']$$