Methods and Heuristics for Learning and Optimization

Series 7: Genetic Programming

Return no later than January 1, 2024 (23h59)

1 Introduction

The goal of this exercise is to use Genetic Programming (GP) techniques to learn *boolean* expressions which reproduce a look-up binary table. More precisely, having a binary table one searches an expression (as compact as possible) which allows computing the outputs based on the input entries.

In this exercise, a genetic program is a set of instructions encoded in a *post-fix* language, in which the operators follow their operands. For example the following (non-boolean) expression:

$$-2\left(\frac{\sin(x)}{y}\right)$$

can be translated into:

```
2 NEG X SIN Y DIV MUL
```

The execution of such a program relies on a *stack* machine, where each instruction can remove elements from a stack and put other elements (e.g. the operator “MUL” removes two elements, multiplies them, and puts the result back on the stack).

2 How to proceed

During this exercise, you must complete the provided Python code `gp.py`, where some genetic operators of selection, mutation, and crossover are already implemented.

N.B. The class CPU in `gp.py` represents the *stack*.

2.1 Representation of individuals

The solutions (programs) are represented by a sequence of boolean operations (NOT, OR, XOR, AND) and variables (X_i).

Debate: which is the research space in this case?

In the code, a program is a list of strings, where each string is the name of a Python function, e.g. *prog* = [”X1”, ”X2”, ”AND”, ”X3”, ”OR”].

You can consider that all the individuals have the same constant size *progLength*.

Debate: what to do with invalid sequences?

2.2 Fitness

The fitness function should take as a parameter the exhaustive look-up table of the wanted boolean expression.

The look-up table consists of n rows. Each row is made up of $m+1$ boolean values; the first m values are the values of the m boolean variables $x_1, ..., x_m$ and the last value is the corresponding correct output $f(x_1, ..., x_m)$.

For instance, the look-up table of a function f taking two input variables

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

can be represented as [[0,0,0],[0,1,0],[1,0,0],[1,1,1]].

The fitness function should execute the learned expression on each line of the table. The fitness represents the number of entries in the table where the program gave a correct answer.

2.3 Wanted boolean expression

The optimal boolean expression (programs) to look after takes as input four variables x_1, x_2, x_3, x_4 and returns 0 unless

$$[x_1, x_2, x_3, x_4] \in \{[0,0,0,1], [0,1,1,1], [1,0,0,1]\}$$

.

2.4 Genetic operators

In addition to the selection, mutation and crossover operators that are already implemented, you can implement new operators (ex. K-tournament selection with variable value of K, etc.) to check if it provides better results.

Clearly indicate the parameters you used, such as the mutation probability p_m, crossover probability p_c, size of tournament K, population size N, number of generations gen, etc.

3 Work to do

Implement the genetic programming within the given code `gp.py`. This includes the creation of an initial population of random programs, their execution using a stack, the evaluation of their fitness, and the evolution of the population over generations. Do not forget to keep at generation $i+1$, the best individual (with the best fitness) of generation i.

Compare the performance of different operators and parameter settings, to generate a program reproducing the provided lookup table. The performance must be done based on the statistics (e.g. average, standard deviation, and best fitness).

N.B. You have some freedom here, show that you know how to investigate.

List the boolean expressions allowing to better reproduce the provided look-up table, and highlight the more compact ones.

Finally, discuss the case where the programs have variable sizes (no need to implement it); the way you should adapt the crossover operator, the mutation operators you would implement, and the way you would control that your program length do not increase too much during evolution.

4 Report

Each student is required to give back a *personal* work consisting of a code and a concise but precise report in PDF format (4-5 pages) displaying an introduction of the problem to be solved, a description of employed metaheuristics, experiments carried through with corresponding results, and a discussion. Both report and code (*Surname Name TP number*) have to be uploaded on Moodle (TP7).