TP7: Genetic Programming

by Martino Ferrari

1 Expression Search

?? Non ? che sono sicuro di avere capito l'incipit ... forse: Si puo' andare bene, a te non va ancora bene??

Genetic Programming is effectively used for building new mathematical expressions, providing solution to a problem, through evolution. Any new expression is built from two predefined sets, one of operators and functions (called *function set*), and a second of constants and variables (called *terminal set*). The expression length can be fixed or variable but, in our exercise, we will consider the fixed-length case only.

In particular, in our TP we will study the problem of a Boolean expression of four variables using only the $[\wedge, \vee, \dot{\vee}, \neg]$ operators in the function set and the variables $[x_1, x_2, x_3, x_4]$ in the terminal set. The truth table of the desired expression, that will be used as *test set*, is shown in table 1.

x_1	x_2	x_3	x_4	$f(x)$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Table 1. Truth table of $f(x)$

Searching for a minimized expression by means of a Karnaugh map brought me to the following possible solution:

$$f(x) = x_4 \wedge ((x_1 \wedge \neg(x_2 \wedge x_3)) \vee \neg(x_2 \dot{\vee} x_3))$$

In this case the symbol set is composed by $[\wedge, \vee, \dot{\vee}, \neg, x_1, x_2, x_3, x_4]$. The length of the solution will be set to the minimum found, i.e. 13 symbols. Each solution $s \in S$ will be a sequence of 13 symbols taken from the symbol set, meaning a solution space S composed by 13^{10} ($\sim 10^{12}$) possible combinations of the symbol set.

2 Genetic Programming

The Genetic Programming (**GP**) technique was first developed in the 90's from the Genetic Algorithm (**GA**) by *J.R. Koza et al.*. The logic behind the **GP** is the same as in the **GA**: through a process of selection, crossover and mutation, generation by generation, the research space S will be efficiently explored and an optimal (or acceptable) solution found. The individuals in **GP** are not just values or data but simple programs.

In order to deal with the need of a reliable syntax capable to resist to crossover and mutation processes, the proposed solution are two:

1. **S-Expression**: where the program is represented as a tree

2. **Post-Fix**: where the program is represented as a list and the execution is based on a stack machine

In our case, we will use the second option that gives us the possibility to simplify the implementation by fixing the size of the program.

A program written in post-fix syntax will look something like `"2 NEG X SIN Y DIV MUL"` that represent $-2 \cdot \frac{\sin(x)}{y}$. The program is built taking the symbols from 2 sets:

1. **Function-Set**: the ensemble of all operators and functions
2. **Terminal-Set**: the ensemble of all constants and variables

All the operators and functions implemented in the function-set must be robust to errors, such that any randomly generated program could always be executed. As policy I choose to skip functions and operators that don't have enough operands or terms (e.g. `"X1 AND"`$\rightarrow$`"X1"`), and to return *false* if at the end of the execution the stack is empty (e.g. `"AND AND"`$\rightarrow$*false*).

The processes of selection, crossover and mutation are very similar to the ones in the **GA**.

The **selection** strategies are the same proposed for the **GA**, and in our case we will use a 2-Tournament.

The **crossover** will be done using the fix-point crossover technique.

The **mutation** will be done by exchanging each gene with a random symbol, with a given probability (and equal probability of being taken from either the function or the terminal set).

To recap, the **GP** has 4 main driving parameters:

- The population size p_{size}
- The program length p_l
- The crossover probability p_c
- The mutation probability p_m

The value of each of them will be discussed in the result section.

3 GP and Boolean Expression

As for any other metaheurstic we have to define few elements:

- The research space S
- The start solution s_0
- The fitness function $f(s_i)$
- The neighbourhood of a solution $N(s_i)$
- The exploration operator U
- The end condition

The research space S was already defined in Section 1. The start solutions (as the **GP** is a population-based algorithm) are randomly generated from S.

To compute the fitness $f(s_i)$ of a solution s, I choose to take into account 3 factors:

1. The number of results that differ from the expected ones taken from the *test-set* ($d_c(s_i)$).
2. The number of errors encountered in the execution (with a penalty of the size of the program for the programs with empty stack at the end of the execution) of the programs ($e_c(s_i)$), e.g. the program `"AND AND NOT"` contains 3 errors and will have a penalty of 3 for having the stack empty.
3. The number of elements left on the stack after the execution ($p_c(s_i)$), e.g. the program `"X1 X2 X3"` will leave 2 elements in the stack.

Finally the fitness is computed as follows:

$$f(s_i) = d_c(s_i) \cdot (1 + e_c(s_i) + p_c(s_i))$$

In this way the selection will not only favour the programs with more correct results but also the ones more synthetically correct. The range of this function is from 0 (all the results as in the test set) down to $-n_{\text{variables}}^2 \cdot (p_l^2 + p_l)$.

The neighbourhood $N(s_i)$ of a solution s_i is virtually the full research space S, as the research operator may create any solution s_{i+1}.

The research operator U, as in the **GA**, is the composition of the *Selection, Crossover* and *Mutation* operations.

Finally the end condition will be $g_{\max}$ (a 5th driving parameter), the number of generations to reach.

4 Implementation

The implementation is based on the Python skeleton provided, where the research operator was already implemented. To speed up the execution of the code I choose to change the program representation from a string list to a function list, to avoid the use of a dictionary.

Since, however, the performance of the code was still quite poor, I choose to re-implement the code in C++ (maintain the same structure, `main.cpp`), gaining a factor of about ~25 in the execution time.

A further variable-length variant will be discussed in Section 6.

5 Results

Before analyzing the performance and results of the algorithm, I tried to better understand the problem and its research space S.

To do so, I generated a landscape with 1000 randomly-generated programs of length 13, with the results shown in Figure 1.

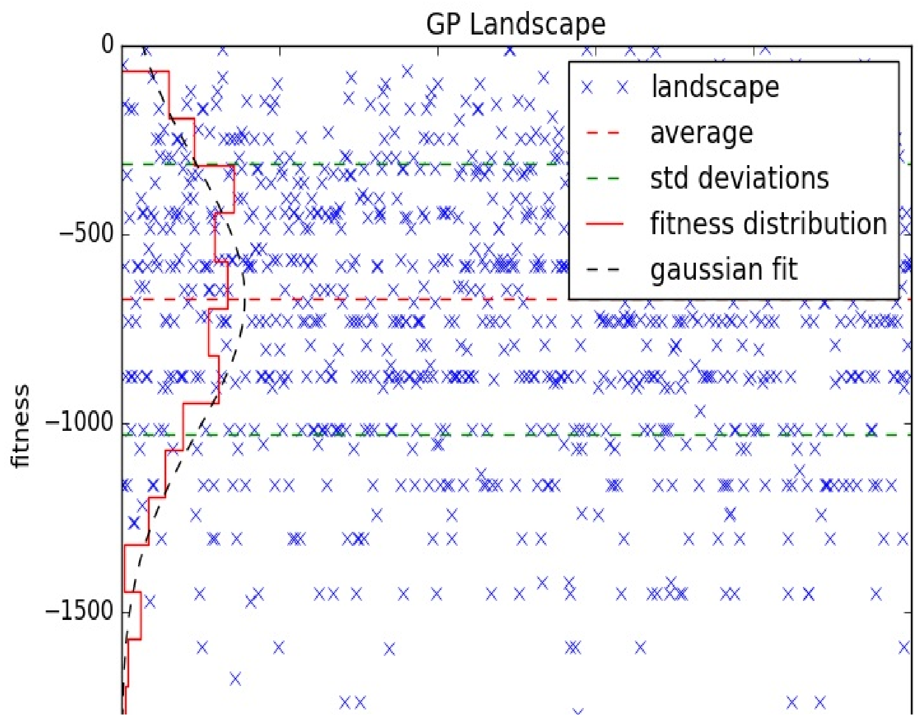

Figure 1. 1000 random-sample landscape

In Fig. 1, it's possible to see that the landscape follow a Gaussian distribution with some asymmetry close to the upper limit. (non è per niente strano ... è limitata superiormente e il limite superiore è in linea di principio un valore osservabile ...) si ma e' limitata anche inferiormente.. In particular these 1000 events have average -670.5, STD 357.7, best value -5 and worst -1885.

After this first observation, I decided to study the influence of the parameters p_m and p_c on the performance of the **GP**: I varied both and for each variation I executed 100 times, with $p_{\text{size}} = 30$, $g_{\max} = 100$ and $p_{\text{length}} = 13$, the algorithm. The results are shown in Figures 2 and 3.

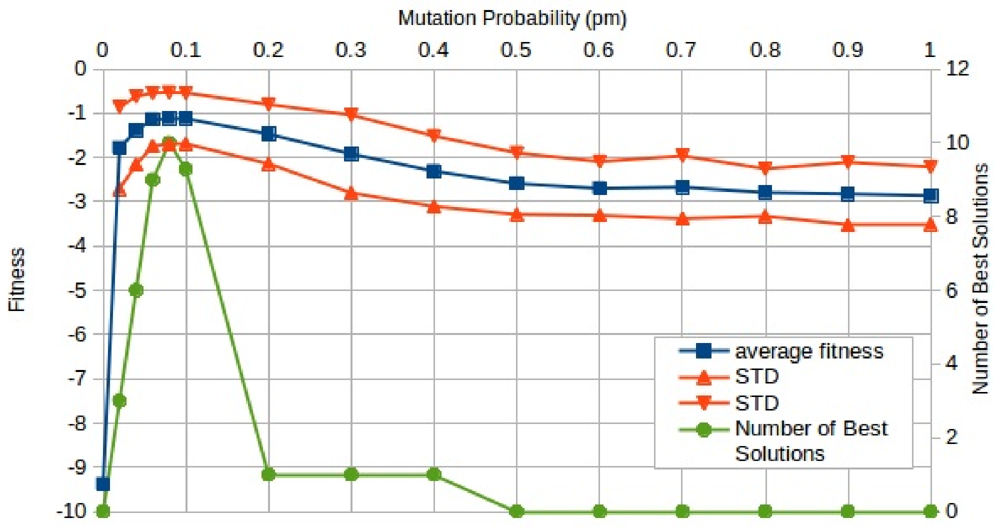

Figure 2. Fitness and Number of Best Solutions Vs p_m ($p_c = 0.5$)

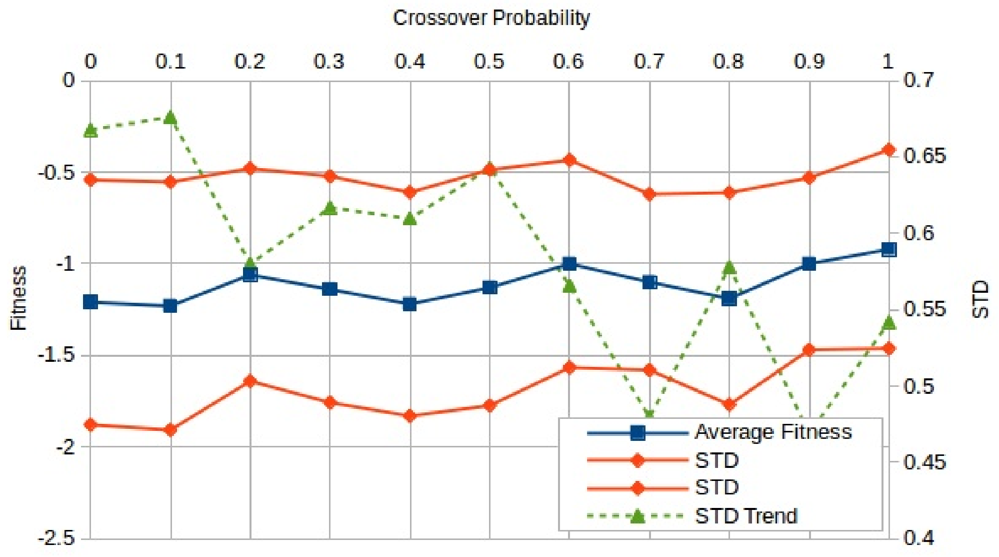

Figure 3. Fitness and STD Vs p_c ($p_m = 0.08$)

The plots enlighten interesting features: the p_m factor strongly affects the performance of the

GP, even very small variations of it may radically change the results. The best performance in both average fitness and number of optimum solutions is found around 0.08 and, with $p_m > 0.4$, it becomes hard or close to impossible to find any optimum solution. It's important to notice that it is not possible to find optimum solutions also with $p_m = 0$.

Quite the opposite, the influence of p_c is much less important, the trend of the average fitness is flat ?? non capisco ma mancano i grafici ??. Guarda il pdf per i grafici The only small difference is on the fitness STD that decreases with increasing p_c (from $\sim 0.7 \rightarrow \sim 0.45$).

The best found combination of parameters is $p_m = 0.08$ and $p_c = 0.7$.

Finally I chose to observe the impact of the program length on the performance, by varying the length from 6 to 20 and observing the results of 100 executions. Figures 4 and 5 summarize the experiment.

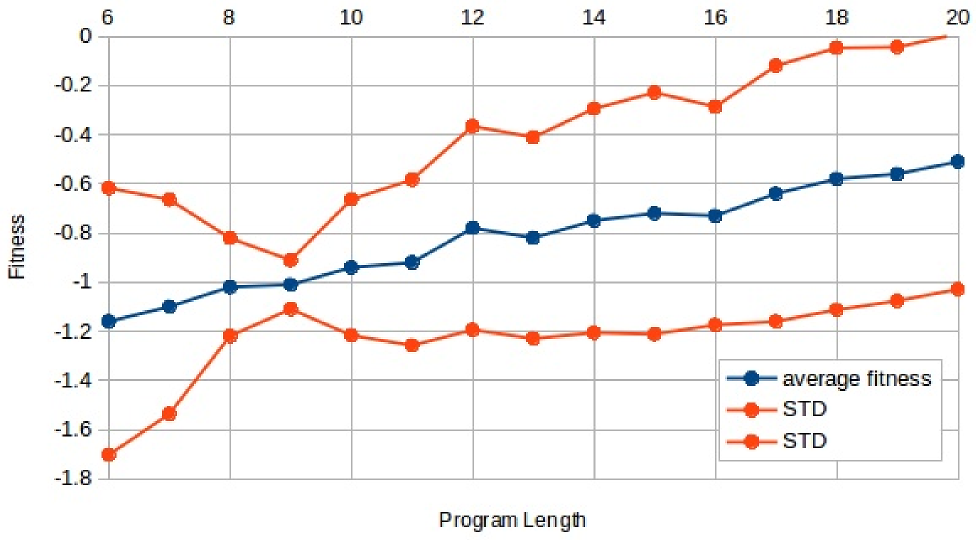

Figure 4. Fitness Vs Program Length ($p_m = 0.1, p_c = 0.5$)

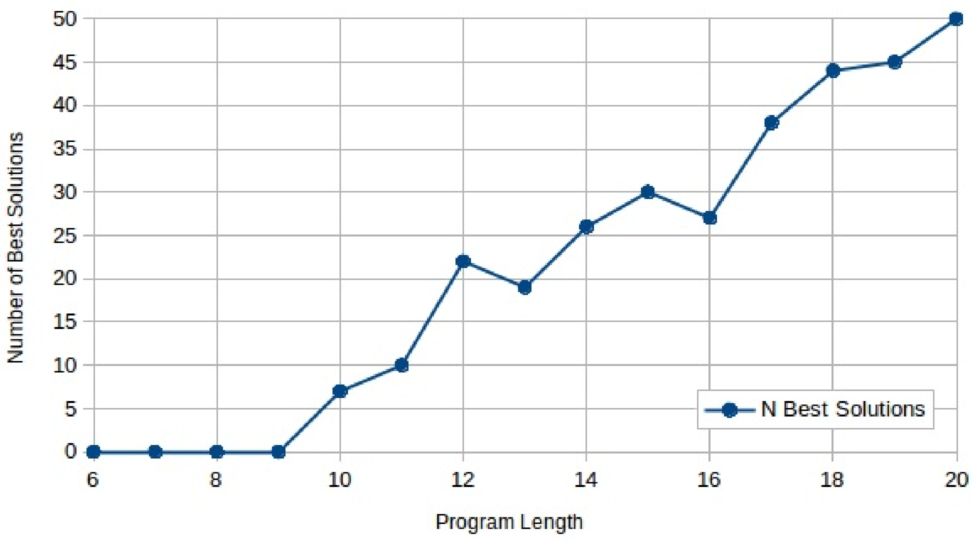

Figure 5. Number of Optimal Solution Vs Program Length ($p_m = 0.1, p_c = 0.5$)

The first notable results of this analysis is that my implementation of the **GP** is capable to find minimal (shorter) solution to the problem than the classical Karnaugh map, as it finds solutions with length 10, 11 and 12 (while using the Karnaugh map gave me program of length 13).

Moreover from the figures, it's possible to see that, even if the research space S increases with the program length p_l (the research space size is 8^{p_l}, 8 being the number of possible symbols), the result quality increases (?? perché dici "even if" ? Se cerchi bene in uno spazio più grande avrai la possibilità di trovare migliori risultati ?? SI Ma essendo lo spazio molto piu' grande dovrebbe essere in teoria + difficile trovare le soluzioni, invece diventa + facile. Io forse direi: 'to see that, having a much larger research space, ..."). In particular, Figure 3 show a linear trend for the number of optimal solutions found (meaning solutions with fitness $= 0$). It's also noticeable the fact that the **GP** is not capable to find any optimal solution with programs shorter than 10. However, for length $= 9$, a large number (close to 90%) of the executions finds solutions with fitness -1.

The reason why the quality of solution increases with the program length is the fact that, with more symbols, it's easier to find combinations giving the correct results with redundant (e.g. `"NOT NOT X1"`) or wrong (e.g. `"NOT X1 AND"`) operations, that due to the fitness function are not taken into account if the results match the test set ??? mi pare abbastanza incomprensibile ? volevo dire che come ho costrutio la funzione di fitness se un programma se pure ha degli errori ma metcha il test-set avra' fitness 0, quindi con una lunghezza + grande sara' in grado trovare sequenze con qualche errore in + ma che comunque metchano il test-set?. This results in a higher number of correct possibilities and the trend shown in figure 5. ??? and can explain the trend. guarda il plot

I performed, as well, tests about the impact of p_{size} and $g_{\max}$ on the performance of the **GP**. As expected, increasing p_{size} or $g_{\max}$ will increase the quality of the results (as the overall number of fitness evaluations will increase as well as the research space exploration).

Consequently, I chose as best parameters: $p_{\text{size}} = 100$, $g_{\max} = 100$, $p_c = 0.5$, $p_m = 0.08$ and I analyzed the algorithm performance over 100 execution and with $p_{\text{length}} = 6, 10, 15$.

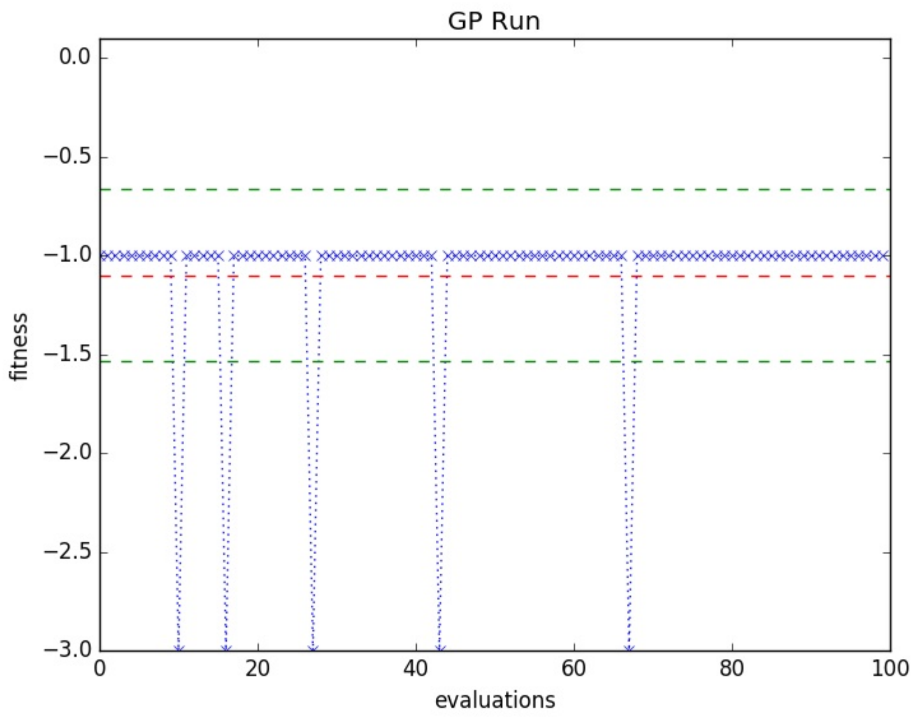

Figure 6. Evaluation with $p_l = 6$

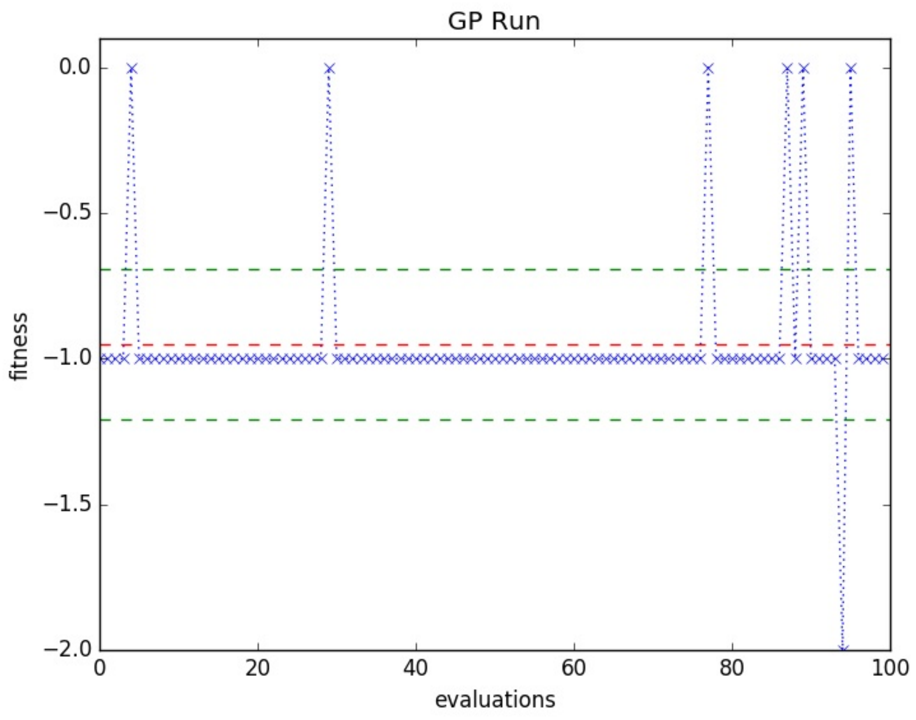

Figure 7. Evaluation with $p_l = 10$

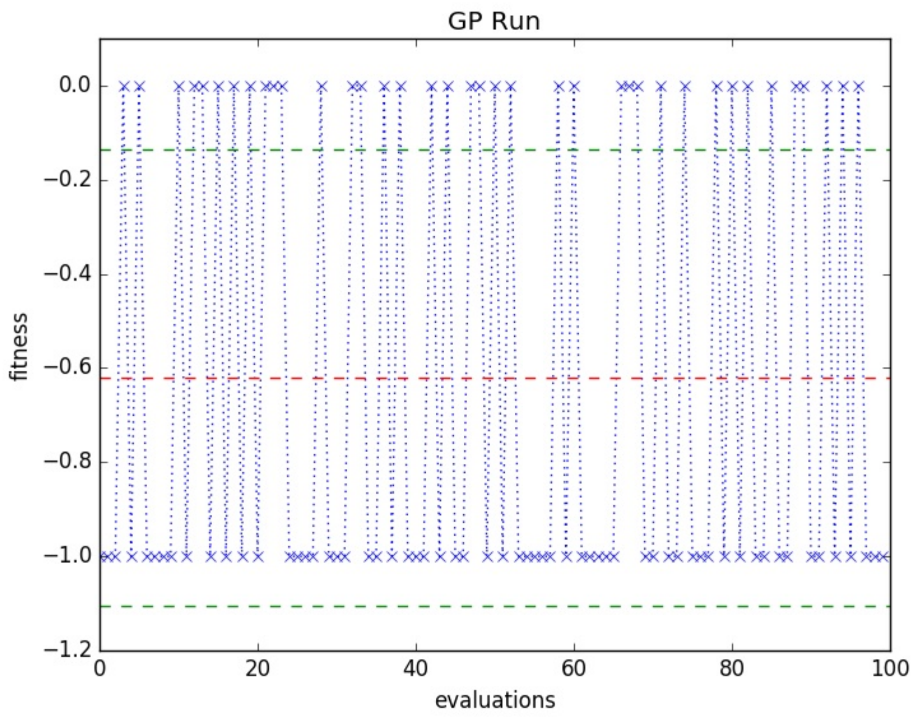

Figure 8. Evaluation with $p_l = 15$

As shown in Fig. 6, 7 and 8, my **GP** implementation performs pretty well, the overall worst result having fitness -3, and it is capable to find solutions with minimum program length ($p_l = 10$) smaller than the ones obtained with classical method. Tables 2 and 3 show the numerical results of the previous test and the best programs created.

p_l	p_c	p_m	p_{size}	$g_{\max}$	Average Fitness	STD	Best
6	0.7	0.08	100	100	-1.13	0.44	-1
10	0.7	0.08	100	100	-0.95	0.22	0
15	0.7	0.08	100	100	-0.62	0.45	0

Table 2. Fitness Table

p_l	Program and Expression	$f(x)$
6	`X2 X3 OR NOT X4 AND`	-1
	$\neg(x_2 \vee x_3) \wedge x_4$	
10	`X4 X2 X1 AND X2 X3 XOR OR NOT AND`	0
	$\neg((x_2 \wedge x_1) \vee (x_2 \dot{\vee} x_3)) \wedge x_4$	
15	`X2 X3 X2 X1 AND AND X3 X4 XOR NOT XOR XOR NOT X4 AND`	0
	$\neg((x_2 \wedge x_3) \dot{\vee} ((x_2 \wedge x_1) \dot{\vee} \neg(x_3 \dot{\vee} x_4))) \wedge x_4$	

Table 3. Program Table

6 Variable Length Implementation

An interesting variation could be a variable program length implementation. In this way the algorithm should be capable of finding the minimal program length without a previous analysis of the problem. For that, I suggest two modification:

- A non length-conservative crossover (o "A length non-conservative crossover" ?? bo, cosa suona meglio?)
- A fitness function modification

To be able to have different program lengths within the same population, I suggest that the crossover be modified by exchanging the portion of the code with same size, e.g.:

fix-point : 2

p_1: `"X1 X2 | AND NOT X4 OR"` , p_2: `"X3 NOT | X4 XOR X3 AND"`

p_1': `"X1 X2 X3 NOT"`, p_2': `"AND NOT X4 OR X4 XOR X3 AND"`

In this way it would be possible to generate a population with variable length starting from a uniform population.

Now the fitness function should take into account the program length, to promote the shortest programs:

$$f(s_i) = d_c(s_i) \cdot (1 + e_c(s_i) + p_c(s_i)) \cdot l(s_i)$$

being $l(s_i)$ the length of s_i.

7 Project structure

gp.py	**GP** implementation in python
compile.sh	is the compilation script
src/	contains all the source code (above the core components)
→/main.cpp	**GP** implementation in C++
→/main_plot.cpp	main file with plot
→/landscape.cpp	random landscape generator
bin/	contains all the binary
→/gp	runs the **GP**

The code and the reports can also be found on my github repository: Metaheuristic.