

**Genetic programming types comparison**

**Matteo Gianvenuti**

Summary

[Project description 3](#_Toc190516656)

[Changes made to the reference codes 3](#_Toc190516657)

[data\_loader.py 4](#_Toc190516658)

[utils.py 4](#_Toc190516659)

[gp\_types.py 5](#_Toc190516660)

[Common changes 5](#_Toc190516661)

[Changes to Cella method 5](#_Toc190516662)

[Changes to Stefano method 5](#_Toc190516663)

[user\_interface.py 5](#_Toc190516664)

[user\_interface\_charts.py 6](#_Toc190516665)

[Representation complexity analysis 6](#_Toc190516666)

[Complexity of *extraction\_tree* 6](#_Toc190516667)

[Complexity of *get\_modules* 7](#_Toc190516668)

[Complexity of *extraction\_list* 7](#_Toc190516669)

[Final complexity comparison 7](#_Toc190516670)

[Results comparison 8](#_Toc190516671)

# Project description

Compare classic/normal genetic programming (GP) with the modified/enhanced GP starting from the Cella-Stefano reference code (possibly improving the code) on the ECG dataset by Arianna Cella:

The comparison should be made at the level of:

1. performance (e.g., MSE for a regression problem).

2. 'effective' size of the solution measured in number of equivalent nodes (i.e. transforming the added functions into the equivalent subtree using the basic functions defined at the beginning and measuring the number of nodes of that solution). (It is expected that equivalent performing solutions can be found with fewer nodes).

3. speed of convergence (by evolving smaller trees, the search space is reduced, and the search should be faster).

At the algorithmic/complexity and code level, it is interesting to evaluate what advantages can be obtained by using the classical representation through trees and the linear one that Francesca Stefano worked on.

Definitions:

"Normal GP" is a normal single run of GP (whatever its purpose) in which the parameters that regulate it, the function set and the terminal set remain constant throughout the execution.

"Modified GP" is the method implemented by Arianna Cella and subsequently modified by Francesca Stefano. In this method, trees of depth Pmax are evolved that are more limited than in the 'standard' case, thus limiting the size of the search space and therefore facilitating the search itself and limiting the generation of bloat.

To compensate for the limited depth, and therefore the limited expressiveness, of the trees that evolve, every X generations the subtrees of depth 2 or 3 at most that appear most frequently in the population are transformed into functions and inserted into the function set. In this way, the average depth of the trees of the current population is limited and it is possible to continue to evolve trees that continue to respect the constraint of the limited maximum depth (or, at most, slightly increasing the depth) with all the advantages that I have previously listed, but are actually equivalent to much deeper and therefore more 'expressive' trees.

The difference between Arianna Cella's version and Francesca Stefano's version consists in the use of a different representation for the trees, which can be found documented in the article to which Francesca Stefano's report refers and which should facilitate the generation of the histograms of the frequencies of the subtrees.

# Changes made to the reference codes

Here I report the most important changes made to bot the reference codes.

General changes:

* Switch to English.
* Use of Python best practices.
* Switch to f strings, they are more efficient because avoid concatenation and calls to the “str” function.

## data\_loader.py

The original file was called “utils\_import\_data.py”. I removed the function “shuffle\_in\_union”. It is sufficient to use the shuffle=True parameter in “train\_test\_split” function already available from scikit-learn.

## utils.py

The original file was called “utils\_functions.py”. For clarity, I added the postfix *\_tree* to all the functions from the Cella’s reference code that works on a tree representation and *\_list* to all the functions from the Stefano’s reference code that works on a list representation (interested functions: extraction and get\_modules). Modified functions:

* **training\_rf**: Originally the mean F1 score was manually calculated after the call to “f1\_score” available from scikit-learn. Now it is directly calculated with the parameter average='macro' in the function f1\_score. I modified the usage of “LabelEncoder“ to directly encode all the labels, this avoid multiple encoding errors. I also removed the save and restore of the numpy random state that is useless/irrelevant because for repeatability is use the “random\_state” parameter of “RandomForestClassifier”. Both refence codes had these “errors”. I modified also the number of estimators from 50 to 100, 100 is the default value.
* **extraction\_tree**: I modified the last past of the function for a strong optimization trick, check the complexity section for more details.
* **extraction\_list**: I noticed the parsing of the tree was wrong, terminals were considered as internal nodes, this build an incorrect tree structure. In fact, commas separate children of the same parent, but the Stefano reference code nests subsequent nodes under the previous terminal. So, I had to rewrite it.
* **get\_modules\_list** and **get\_modules\_tree**: I generalized it in one function that take as argument the extraction mode (extraction\_tree or extraction\_list) because they do the same thing.
* **get\_modules\_individual\_tree** and **get\_modules\_individual\_list**: I removed these functions, they were unused.
* **depth\_tree** and **depth\_list**: I removed these functions because you can already obtain the individual depth with the tools provided by DEAP (individual.height), reducing the complexity.
* **view\_hist1** and **view\_hist2**: These two functions were different only in one print. So, I generalized them into one by passing the difference as parameter.
* **varAnd**: This function was exactly the same provided by DEAP then I simply imported it.

Unused library removed: “convolve” and “cross\_val\_score” from both the reference codes.

## gp\_types.py

The original file was called “GPmodular.py”. Unused library removed from both the reference codes: “collections”, “KMeans” and “PCA”.

### Common changes

* I generalized the two functions *evalTestSet* and *evalValidationSet* into a sigle one, *evalSet*, by passing the type of test set as parameter.

### Changes to Cella method

### Changes to Stefano method

* I removed the library “warnings” because warnings must be resolved not ignored.

## user\_interface.py

The original UI relied on just one process (the graphics one) that had to handle the graphics and run the program when launched, so if it ran the program, the graphics would get stuck and become unresponsive.

I used multiprocessing to run the program through a dedicated process. So now it is possible to launch in parallel different configurations that run at the same time (if the hardware supports). You can launch all three types of GP at the same time (classic, Cella and Stefano methods).

I modified the “run\_script” function to measure the running time and to save all the run data in a more compact and structured mode avoiding unnecessary cycles that save the same information multiple times. I improved the efficiency of calculating the mean F1 by calculating it with "map" and "sum" functions rather than manually scrolling the list with a for loop.

I also added support for the “verbose” choice in the graphics.

A screenshot of a computer

AI-generated content may be incorrect.

I removed the function “graph” because no longer useful, I did a dedicate UI to see results.

## user\_interface\_charts.py

I created this “extra” UI to parse and show the run results saved. With this UI is possible to plot in multiple windows results of different runs to compare them. It shows the parameters configurations, the F1 performance on test and validation set along the iterations for each run, the number of equivalent nodes of the best individual, the run time and average F1 on the test set. Finally, it shows average F1 on all runs and total time.

# Representation complexity analysis

The “representation functions” are in *utils.py*. For the following analysis, “n” is the general length of a string (individual).

## Complexity of *extraction\_tree*

The “replace” function have a complexity of O(n).

*regex\_depth1* complexity: *(?:add|sub|neg|mul|div|execTree\d+)* is a direct comparison so it is O(1), the part *\((...)\)* is a series of alternatives:

* -?\d+ search for a positive/negative number, worst case O(n)
* [A-Za-z0-9\_]+ search one or more objects in the specified ranges, worst case O(n)
* \([^()]+\) search for something between parenthesis but not parenthesis, worst case o(n)
* -?\d+,-?\d+ search for a pair of integer, worst case O(n)
* [-A-Za-z0-9\_]+,-?\d+ search for a pair “string” in the range and a number, worst case O(n)
* [-A-Za-z0-9\_]+,[A-Za-z0-9\_]+ search for a pair of “strings” in the range, worst case O(n)

All the other regex are combinations of these or similar objects like “(?:-?\d+|ARG\d+|[A-Za-z0-9\_]+)”, “(?:,-?\d+|,ARG\d+|,[A-Za-z0-9\_]+){0,3}” this is O(3n) but it is always O(n), etc. So, all these have a complexity of O(n). As consequence the re.findall(str\_a, str\_b) has a complexity O(m\*n) where m is the number of matches but it is always O(n). We know that O(n) + O(n) + … + O(n) = O(n). For the last part of the function is better to see the code with comments.

A computer screen with text

AI-generated content may be incorrect.

The original code was the one with worst case complexity O(n3), with my optimization the final complexity of this function is O(n).

## Complexity of *get\_modules*

This function is composed by a for loop (on the population of size p) with a call to *extraction\_*tree and two for loop (one for module with size m1 and m2) nested at the same level, so the complexity is O(p)\*(O(n)+O(m1)+O(m2)) = O(p\*n). It could be considered as O(n) with n → ∞ and a limited population or O(p) with p → ∞ and a limitation on the size of the individuals.

## Complexity of *extraction\_list*

The complexity analysis is done always for the worst case. It starts with a “replace” that is O(n), after there some calls to support functions, all of these are bounded by O(n):

* parse\_expr: entirely scan the individual to build a representation (‘func’, [(‘func2’, [‘arg0’, ‘arg1’]), ‘arg1’]), so it is O(n)
* extract\_nodes: entirely scan the previous representation (so the nodes) to build the submodules depth lists, so it is O(b) with b the number of nodes

Support functions used:

* is\_operator: it directly checks for “mul”, “add”, etc. It is O(1)
* is\_flat: it checks if a node is a terminal and return it, if not return the operator and its arguments. Then, it is O(k) with k the number of arguments for a node.
* node\_to\_str: rebuild the submodule as a single string by joining the node and its children. It is O(m) with m the total number of characters involved.

Since all of these are bounded by O(n) the final complexity is O(n).

## Final complexity comparison

Both the representations use the *get\_modules*, so they are different only in the “extraction” (*extraction\_tree* and *extraction\_list*). I showed the two extraction modes have the same complexity O(n), then they are practically comparable or equal.

Overall, the Stefano method can be a bit slower than Cella method because it dynamically calculates the crossover and mutation probabilities at every iteration.

# Results comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **F1** | **Eq. nodes** | **Time (min)** |
| modularGP\_Cella |  |  |  |
| modularGP\_Stefano |  |  |  |
| classicalGP |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **F1** | **Eq. nodes** | **Time (min)** |
| modularGP\_Cella |  |  |  |
| modularGP\_Stefano |  |  |  |
| classicalGP |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **F1** | **Eq. nodes** | **Time (min)** |
| modularGP\_Cella |  |  |  |
| modularGP\_Stefano |  |  |  |
| classicalGP |  |  |  |