



**UNIVERSITÀ
DI PARMA**

Genetic programming types comparison

Matteo Gianvenuti

Summary

Project description	3
Changes made to the reference codes.....	4
data_loader.py.....	4
utils.py	4
gp_types.py.....	5
Changes to both methods	5
Changes to Cella's method	6
Changes to Stefano's method	6
user_interface.py.....	6
user_interface_charts.py	7
terminal_interface.py.....	8
best_ind_f1.py.....	8
Representation complexity analysis	9
Complexity of <i>extraction_tree</i>	9
Complexity of <i>get_modules</i>	9
Complexity of <i>extraction_list</i>	10
Final complexity comparison	10
Results comparison	10
Experiment one	11
Experiment two	11
Experiment three	11
Conclusion.....	12
Figure 1 User Interface	7
Figure 2 Some graphs of experiment three, using user_inerface_charts.py	8
Figure 3 Terminal Interface, example of usage.....	8
Figure 4 Python complexity comparison	9

Project description

Compare classic/normal genetic programming (GP) with the modified/enhanced GP starting from the Cella-Stefano reference code (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 P_{max} 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.

Changes made to the reference codes

Here I report the most important changes made to both 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*, *get_modules*, *get_modules_individuals* and *depth*). 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. I modified also the number of estimators from 50 to 100. 100 is the default value, I think 50 are few.
- **extraction_tree**: I modified the last part 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 completely.
- **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”.

Changes to both methods

- I generalized the two functions *evalTestSet* and *evalValidationSet* into a single one, *evalSet*, by passing the type of test set as parameter.
- The functions *mul* and *protectedDiv* were not necessary as defined, this because if the situation NaN happen for them should happen also for the other operators. So, I used the *mul* operator provided by the “operator” library as is done for the other operators. The unique real risk is the division by zero, so I maintained the *protectedDiv* (fixed), and I renamed it to *div* this because the regexes search for *div*, so previously the *protectedDiv* was not seen as a valid function in the search for submodules.
- the definition of the ephemeral constant was done via a *lambda* function, this prevented the serialization of the best individual in the final save. I replaced the *lambda* with *functools.partial*.
- I rewrite the function *sostituisci* (now called *replace*), to cycle the index efficiently by exploiting the mathematical operation of the module.
- I avoided the possible risk of overwriting between best individual and identity at the beginning of the iteration.
- At each iteration the MAX_DEPTH variable was increased by two, but this operation has no effect. The maximum depth is defined with the “register” on the toolbox and therefore the variable is considered at that moment. Furthermore, a control limit is added to the toolbox to not exceed this depth. I therefore removed this operation also because it is contrary to the methodology of keeping the depth limited to work on the submodules. Check the code for more details.
- At the end of the iteration, the individuals to be kept were added to the psets. The operation was done in two cycles, it could be done with just one reducing the complexity. I fixed it.
- I added the function *count_nodes* to count the number of equivalent nodes of an individual by expanding the submodules through pset.

Changes to Cella's method

I moved the *replace* function definition out of the loop on iterations to avoid the overhead of continuous redefinition.

Changes to Stefano's method

- I removed the library "warnings" because warnings should be resolved not ignored.
- In the individual evaluation function (*evalTrainingSet*) the number of nodes was defined as the depth of the individual. Also, as k-value (in the parsimony pressure formula to avoid bloat) $1e-2$ was used which is too high. In fact, the best individual was practically always a terminal. I corrected these things and brought the k-value back to the same one used by Cella in Cella's method ($1e-4$) also to have the fairest comparison possible between the two methods.
- In the function *get_individuals_to_keep* the fitness normalization formula was incorrect, I brought it back to the formula originally used by Cella.
- I fixed the *adjust_probabilities* function to dynamically vary the *cspb* and *mutpb* probabilities (what it should do I think), originally it was swapping the mutation operators.
- An identity individual was inserted into the population in all iterations. This should only be done at the first iteration (as Cella's method does), in subsequent iterations the population must evolve with the operators (elitism aside). I fixed it.
- The evolutionary algorithm originally started with *varAnd*, then evaluated fitness, made a selection, and repeated the process for all generations. This is not in line with the general evolutionary algorithm (already implemented by *eaSimple*) where before starting the generations the fitness is evaluated, and then a selection is made at the beginning of the generation, etc. So, I switched it back to using *eaSimple_elite* also for as fair a comparison as possible with the Cella's method, maintaining the function *adjust_probabilities*.

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's 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.

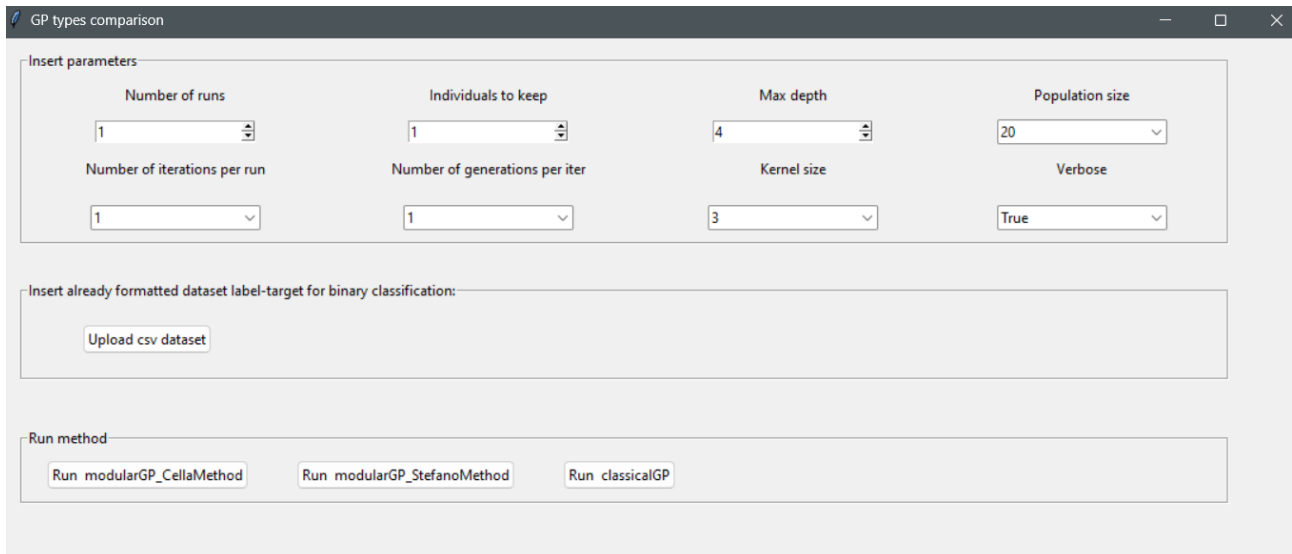


Figure 1 User Interface

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/iterations and total time.

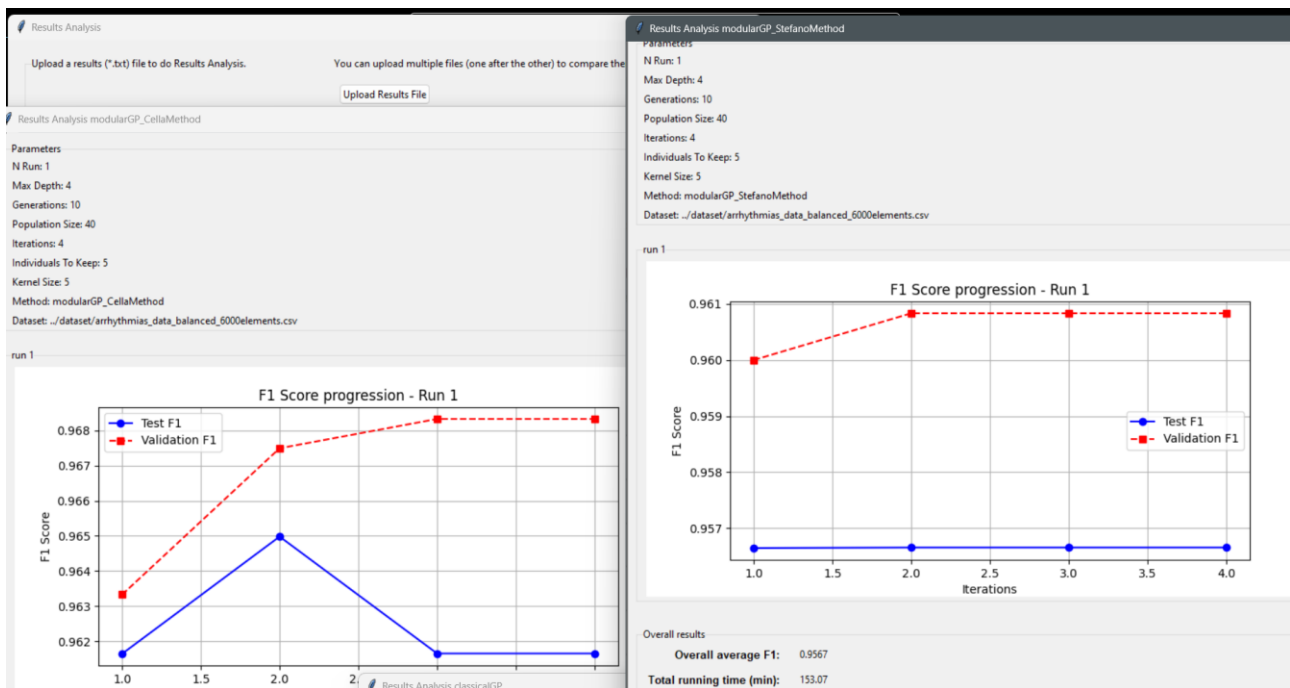


Figure 2 Some graphs of experiment three, using `user_interface_charts.py`

terminal_interface.py

This is an equivalent version of “`user_interface.py`” but it is used from a terminal, it does not have a graphical interface. As consequence, there is less overhead because there is no longer the parent process that manages the UI, this version has allowed to significantly reduce execution times (see experiment one). It is based on the library “`argparse`”. It is always possible to run several methods in parallel using multiple terminals. I recommend using this interface.

```
PS C:\Users\MATTEO\Desktop\ML\project\code> python ./terminal_interface.py --generations 5 --pop_size 50 --iterations 3
--inds_to_keep 3 --kernel_size 5 --method modularGPCella --dataset ../dataset/arrhythmias_data_balanced_6000elements.csv
```

Figure 3 Terminal Interface, example of usage

(You can also change the default value of the arguments in the code, and then start it without writing them in the terminal)

best_ind_f1.py

The original file was called “`script.py`”. This file loads a dataset, a best individual with its kernel size and its pset to evaluate the F1 score.

I used the functions already defined in the other files (before they were redefined), and removed useless operations (e.g., the pset was set from the saved parameters but then read from the file in which it is saved).

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 $O(n) + O(n) + \dots + O(n) = O(n)$. For the last part of the function is better to see the code with comments.

```
# optimized check # O(n)
submodules_depth1_set = set(submodules_depth1) # "not in"/"in" is O(1) for a set, O(n) for a list
submodules_depth2 = [module for module in submodules_depth2 if module not in submodules_depth1_set]

# original check # p, b <= n, then it is O(n**3)
for module in submodules_depth1: # O(p) with p = len(submodules_depth1)
    if module in submodules_depth2: # worst case O(b) with b = len(submodules_depth2)
        submodules_depth2.remove(module) # worst case O(b)
```

Figure 4 Python complexity comparison

The original code was the one with worst case complexity $O(n^3)$, 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 $O(p)*(O(n)+O(m_1)+O(m_2)) = O(p*n)$. It could be considered as $O(n)$ with $n \rightarrow \infty$ and a limited population or $O(p)$ with $p \rightarrow \infty$ 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'])]`), 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’s method can be a bit slower than Cella’s method because it dynamically calculates the crossover and mutation probabilities at every iteration.

Having optimized as shown and removed unnecessary operations as listed above, this version of the code will have less execution time and overhead than the original one.

Results comparison

For the classical/normal GP the number of runs and iterations are always one. While individuals to keep (number of submodules) is not applicable, it is zero. So, the classical GP may obviously take less time when the other methods have much more runs or iterations.

At each iteration, the individuals to be retained are chosen and placed in the pset as functions to be used for subsequent iterations. So, increasing the number of iterations or generations can be important. While the run is a completely new run that starts from scratch with the same parameter configuration. So, I preferred to keep one run and increase iterations/generations. What is called the number of runs in the project description corresponds to iterations.

Note: results are subject to variation due to probabilities and **hardware** (the used hardware is dated 2020, so I did what the hardware allowed me to do). Here are some experiments and comparison.

Experiment one

Max depth = 4 (6 for classicalGP). Generations = 5. Population = 50. Iterations = 3. Individuals to keep = 3. Kernel size = 3. Dataset = arrhythmias_data_balanced_3000elements.

Method	F1 (test set)	Eq. nodes	Time (minutes)
modularGP_CellaMethod	0.9018	20	75.4414
modularGP_StefanoMethod	0.8892	12	24.7755
classicalGP	0.9100	11	9.7079

In this experiment the Cella's method is much slower than Stefano's method because for it I used the user interface (after this test I developed the terminal interface).

Experiment two

Max depth = 4 (6 for classicalGP). Generations = 10. Population = 40. Iterations = 3. Individuals to keep = 3. Kernel size = 3. Dataset = arrhythmias_data_balanced_6000elements.

Method	F1 (test set)	Eq. nodes	Time (minutes)
modularGP_CellaMethod	0.9574	13	75.0966
modularGP_StefanoMethod	0.9444	17	85.3785
classicalGP	0.9549	29	30.0474

Experiment three

Max depth = 4 (6 for classicalGP). Generations = 10. Population = 40. Iterations = 4. Individuals to keep = 5. Kernel size = 5. Dataset = arrhythmias_data_balanced_6000elements.

Method	F1 (test set)	Eq. nodes	Time (minutes)
modularGP_CellaMethod	0.9625	16	129.8882
modularGP_StefanoMethod	0.9566	21	153.0693
classicalGP	0.9483	20	29.0836

Conclusion

I note that in general increasing the iterations/generations leads the "modified" GP to a better result than the "normal" GP, as expected. For the number of equivalent nodes of the "modified" GP, it is possible that in some cases it is greater than those of the "normal" GP, this is because there are submodules that once expanded can greatly increase the number of nodes but in general, as expected, the "modified" GP gives a better result (less or equal nodes with a higher F1 performance). For the convergence time, as mentioned above, the "modified" GP will practically always have a higher time because it has more iterations. It is therefore necessary to evaluate on the basis of the application case whether it is worth a higher overhead of the "modified" GP for an improvement in results, even if "small". To better compare the results of the methods (to see graphs/charts) it might be useful to use "user_interface_charts.py".