Generating Object-Oriented Source Code Using Genetic Programming

Vicente Illanes
vicente.illanes.v@gmail.com

Alexandre Bergel
abergel@dcc.uchile.cl

ISCLab, Department of Computer Science (DCC), University of Chile

***Abstract*—Using machine learning to generate source code is an active and highly important research area. In particular, it has been shown that genetic programming (GP) efficiently contributes to software repair. However, most of the published advances on applying GP to generate source code are limited to the C programming language, a statically-typed procedural language. As a consequence, applying GP to object-oriented and dynamically-typed languages may represent a significiant opportunity.**

This paper explores the use of genetic programming to generate objected-oriented source code in a dynamically-typed setting. We found that GP is able to produce missing one-line statements with a precision of 51%. Our preliminary results contributes to the state of the art by indicating that GP may be effectively employed to generate source code for dynamically-typed object-oriented applications.

I. Introduction

Generating source code using machine learning is a challenging and promising research area. In particular, machine learning has the potential to significantly reduce the programmer's burden to manually fix software errors [1], [2].

Genetic programming. In the area of artificial intelligence, genetic programming (GP) is a computational technique, based on the biological principles of the Darwinian evolution theory, which allows one to find a computational program that solves a specific task in an automated and iterative way [3], [4]. Throughout iterations, also called generations, the execution of GP maintains a population of individuals, where each one of them is a computational program represented by its abstract syntax tree (AST). At each generation, GP uses genetic operators to transform individuals into variants, and thus, producing new programs. The initial population is generated at random using the universe of variables and functions of the particular problem. The quality of the individuals is measured using a fitness function, which is evaluated by compiling and executing the AST. The algorithm stops when a particular condition is met.

Code generation. GP has been successfully employed to generate software patches. Most of the proposed approaches, pioneered by Weimer et al. [1], operate with C, a statically-typed and procedural programming language. The 2021 edition of Tiobe's ranking[1] reveals that nine of the ten most popular programming languages support object-orientation and six are dynamically typed. As such, all indicate that studying the expressiveness of genetic programming for dynamically-typed object-oriented programming language is a valuable and relevant research area to explore. Our paper makes a significant step in this field by evaluating how well does GP generate code for a dynamically-typed object-oriented programming language.

Single statement. A software failure may be caused by one single line of code. Such failures are called *single-statement bugs* and their syntactic constructions follow relatively simple patterns [5]. Software repair techniques therefore consider single-statement bugs as a relevant target.

To assess the expressiveness of GP to produce code, we employ GP to generate the body of an arbitrarily designed method. We only consider methods that have exactly one statement in their body to solely focus on the code generation while avoid bug localization.

Methodology. For a given application that has unit tests, we iteratively use GP to produce the body of each single-statement method. The fitness function reflects the number of tests that successfully pass after replacing a method's body with the GP-produced body. The performance of GP is then assessed by comparing the produced code with the original method body. The traditional GP algorithm and genetic operations are complemented with a weight mechanism that takes into account particularities of a dynamically-typed object-oriented programming language.

We design our experiment using the Pharo programming language [6][2], which has the property to be homogeneous in its syntax (e.g., all the computation is expressed using a unique syntactic construct, the message send) and it has an extremely reduced set of involved concepts (e.g., a program under execution is expressed using objects and objects interact with each other by sending messages).

We use a *set of weights* that associate probabilities to different values that may be used when generating a node in the abstract syntax tree. In particular, the weights reflect programming style and convention used in class hierarchy, class reference, instance variables, and method locations. Each weight represents a probability to pick a particular value for the AST node under generation (e.g., variable name, method call). This set of weights is calibrated using a large codebase.

[1]https://www.tiobe.com/tiobe-index/

[2]http://pharo.org

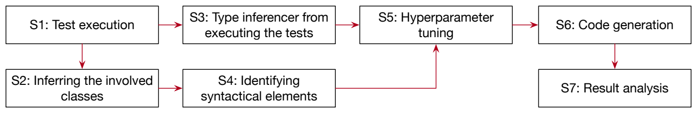
S1: Test execution
S3: Type inferencer from executing the tests
S5: Hyperparameter tuning
S6: Code generation
S2: Inferring the involved classes
S4: Identifying syntactical elements
S7: Result analysis

Fig. 1: Our methodology in 7 steps.

Results. We have run our approach on Roassal [7], a medium sized visualization engine written in Pharo. Our implementation of GP, using the set of weights, is able to produce 51 methods from a total of 99 (= 51%).

Furthermore, using the set of weights to favor particular values used in the AST nodes help improve identifying methods that are more complex in their structure.

Paper outline. The paper is structured as follows: Section II details the research question and the methodology we have adopted to answer the question; Section III explains how we represent Pharo code using GP; Section IV describes our experiment and the results it has produced; Section V presents how we expressed similarity between methods using weights; Section VI concludes and outlines our future work.

II. Methodology

The research question we are investigating is:

How suitable is genetic programming to generate code for an object-oriented, class-based, dynamically typed programming language?

To answer this question we defined a methodology presented in Figure 1. Our methodology is composed of seven different steps, each describing a particular experimental phase. To illustrate the methodology, consider the method called `maxValueX` that belongs to the class `RSAbstractPlot`, itself belonging to the package named `Roassal3-Charts` in the Roassal application:

```
RSAbstractPlot >> maxValueX
   "Return the maximum X value of the plot, excluding NaN and infinite"
   ^ self definedValuesX max
```

The body of the method is highlighted in **bold** and it returns the maximum of the defined values for X. It uses the `self` variable[3] to which it sends the message `definedValuesX`. The expression `self definedValuesX` returns a collection of numbers. The message `max` returns the maximum value of this collection. Since computation is almost exclusively expressed using method calls in Pharo, a source code generation must determine which other methods (defined in the class or not) must be called. As such, the sequence of calls must fulfill non-explicit type requirements as in any programs written using a dynamically-typed programming language.

This section details our methodology and illustrates how a single-statement method body is generated, `^ self definedValuesX max` in our example. Our methodology assumes the presence of unit tests that cover (either directly or indirectly) the method whose body has to be generated.

[3]The `self` keyword is used in Pharo to reference the current object, equivalent to `this` in Java and C++.

S1 - Test execution. The first step of our methodology is to run the unit tests of the project that comprise the method to generate. Tests are executed in a controlled environment in which each method call is monitored. We use the Spy profiling framework [8] to (i) monitor all the passing values as arguments and return values, and (ii) to extract the call-graph.

The model obtained from monitoring the unit tests are key to infer the classes that may be involved in the method body (S2) and identify method signature (S3).

S2 - Inferring the involved classes. The second step of our methodology is to determine which classes may be involved in the method body to be generated. These classes are useful to determine all the possible methods that may be invoked in the method body generation. For example, the original body definition of `maxValueX` involves two classes:

- `RSAbstractPlot` since the `self` keywords reference an instance of one of the subclass of `RSAbstractPlot`, and
- `OrderedCollection` since the method `definedValuesX` returns a collection of numerical values. The maximum value of an `OrderedCollection` is obtained by sending the message `max` to it.

We deduce the classes that may be involved in the method body from the model obtained from Step S1. The coverage of the tests refers to the portion of the base code to be executed by the tests. Classes that are marked as fully or partially covered by the tests are considered as potential classes involved in the method body to be generated.

S3 - Deducing method signature. The signature of a method consists of a method name, the number of arguments, the types of these arguments, and the return type. Dynamically typed languages, including Pharo, do not explicitly state method signature. As such, programmers rely on knowledge and documentation to deduce the types of the argument and return values.

From the result of Step S1, method signatures are deduced based on the values during the test executions. We therefore built a simple type model that simply consists in identifying the type of a variable from the classes of the object held in that variable during the execution.

Method signatures will be employed in the subsequent steps to calibrate the weights and to generate type-safe sequence of method calls.

S4 - Identifying syntactical elements. Genetic programming is about generating adequate abstract-syntax trees, themselves formed with node representing syntactic elements. We identify such elements from various sources:

- instance variables defined in the class and its superclasses of the method under search;
- all the methods defined in the class and its superclasses of the method under search;
- methods that are covered during the test execution (S1);
- built-in variable such as `self`;

S5 - Hyperparameter tuning. Our approach involves numerous hyparameters. Beside the classical hyperparameter associated with the genetic programming algorithm (e.g., population size, probability to perform a genetic operation), our approach associates a weight to each kind of syntactic element the algorithm can pick during the genetic operations and when creating the initial population. We consider theses weights as hyperparameter since they are used to control the generation and evolution process.

Relevant syntactical elements, identified in S4, are accompanied with weights that are determined through empirically mining a large code base of Pharo. In particular, we have found that methods defined in a class are more likely to access the variables defined in that class than variables defined in its superclass chain. Similarly, a method is more likely to invoke a method within the same class rather than invoking located up in the superclass chain.

Empirically, we have found that a method having only one statement performs between 0 and 2 calls. About 73% of single-statement methods do not perform any calls (e.g., accessor and mutator methods), oppositely, only 2% of single-statement methods performs 6 or more calls. The method body we generate uses a chain of invocation that does not exceed the maximal length of method call chain that we empirically found.

During the tests executions we determine the number of times each method is invoked. The number of time a method is invoked is also used when generating method nodes in the AST. The rational is that a method that is frequently executed is more likely to appear in the method to be generated.

S6 - Code generation. The code generation is driven by the genetic programming algorithm. The fitness the algorithm tries to optimize is the number of passing tests. It may happens that some individual takes a long time to execute. The execution happens within a dedicated thread. In case the thread takes an excessive amount of time, it is killed by a supervisor.

Consider the following method possible individual:

```
RSAbstractPlot >> maxValueX
  "Return the maximum X value of the plot, excluding NaN and infinite"
    ^ self maxValueX
```

Listing 1: Example solution with error

Such a method creates an infinite loop. The supervisor thread stops the execution and a comparatively bad fitness is given to the individual.

S7 - Result analysis. In the previous step, the genetic programming algorithm produces a method body that maximizes the amount of passing tests. Several outcomes are possible:

- The generated method body is identical to the original one. In this case, the algorithm produced the very same code that the author produced;
- The generated method body is syntactically different but semantically equivalent. On our example, the genetic algorithm could produce the body `^ xValues max` (listing 2), which has exactly the same semantics than the original definition, `^ self definedValuesX max`.
- The generated method body is syntactically and semantically different than the original method body:
  - If the fitness of the best individual did not reach 0, meaning that some tests are failing, then the method is too complex for the algorithm. In this case, augmenting the population or adjusting some hyparameters may solve the case;
  - If the fitness indicates that all the test passes, then the method is not executed by the tests, which means that the method may not be a good candidate to be run by our approach since it is covered by the test. In this case, new tests have to be added.

```
RSAbstractPlot >> maxValueX
   "Return the maximum X value of the plot, excluding NaN and
   infinite"
  ^ xValues max
```

Listing 2: Example solution found by technique

Summary. The methodology this section presents is designed to use genetic programming to generate a method body for a program that (i) is written in a dynamically-typed programming language, and (ii) is accompanied with a set of tests. We applied our methodology to the Pharo programming language, however it is not tied to it and we believe it may be applied to a different language.

The following section discusses some details about our implementation of the genetic programming algorithm.

III. Genetic Programming

A. Program representation

We represent each candidate program as an abstract syntax tree (AST), which includes a generated statement to be compiled as the body of the searched method. The different ways in which this line of code can be composed and constructed directly influences the way in which genetic operators operate. At the time of constructing an individual, GP must select the type of statement in a random way, not necessarily uniform. Our approach considers three types of statements:

- *Return Statement*: the program returns a value or the call to another method, as is the case with `maxValueX`.
- *Assignment Statement*: programs that allow mutation of instance variables. In this case, care is taken to not generate an assignment that reassigns `self`, since it is illegal.
- *Void Statement*: a call to a method that indirectly performs an assignment (return `self`).

B. Selection, crossover and mutation

The roulette algorithm is used to select individuals for the next generation, where the probability of choosing an individual is directly proportional to their fitness within the population. We always select the best individual of each generation (elitism). So, if the population size is P, we apply $P/2$ times the selection algorithm where in each case two individuals are selected, which will be used as parents to apply the genetic operators on the AST and generate two new individuals.
The technique considers the two traditional genetic operators: the crossover will produce two new individuals, while the

mutation will have a small chance of changing a subtree of the AST. However, these operators must be careful not to generate irrelevant statements, such as assigning to self, or making too many calls to other methods. Therefore, each time an individual is added to the population, it must fulfill certain predicates, in case it does not fulfill any, the individual is discarded and the operators are applied again until individuals that fulfill the invariants are obtained.
The crossover is calculated using a cutoff point in the AST in one of the parents, and then we swap the subtrees based on this point. This generates two new ASTs. In the case of mutation, it is defined in terms of the crossover between the individual and another completely new one.

C. Fitness function

Given an individual, the fitness function measures how acceptable and effective the solution offered by an individual is. These values are important for the continuation of the algorithm in future generations, since the selection algorithm requires the fitness of each program. The value is calculated as the absolute value of the difference between the total number of tests and the number of tests that pass positively. Note that GP's goal is to minimize this value to zero. Those methods that do not compile, generate errors, make too many calls are penalized with fairly high fitness values, so that our algorithm discards it. It should be noted that the quality of this function depends exclusively on the set of tests, so if there are too many successful tests from the beginning, it will be more difficult for the technique to find a solution. In addition, it is important that the tests cover the searched methods, so that the functionalities that are required in the application are rescued.

IV. Experiments with Uniform Weight System

We have run our experiment as described in our methodology on the Roassal3-Chart library. This library has more than 180 methods, for which 99 methods (i) have a body length of one line of code and (ii) are tested by some unit tests. We will run GP to iteratively generate the body of these 99 methods. By iteratively, we refer to assuming the original definition of 99 methods, we will produce the body of one designed method.

Also, abstract methods are not considered, since it is trivial that tests pass when redefined by a minor class, and also those methods that use blocks (In Pharo these are loops statements). As mentioned above, the added statement only has three possible forms: return, assignment, or indirect mutation.

A. Calibration algorithm

Sample and Tests. The library has its own test package called Roassal3-Chart-Tests with 65 unit tests. We extract the Roassal methods that participate in some way in the tests to form the sample of the methods that the technique will look for. We have 34 methods that participate statically, that is, that are used directly in the source code. On the other hand, these methods can make calls to other methods during the execution flow that are not necessarily statically present. We have 65 different methods that are called indirectly. We make this distinction, since the technique needs a good coverage from the tests cases, because the fitness calculation is used to determine which individuals are better. For example, if 90% has already been met since the beginning of the 65 tests, it will be more difficult to find the correct code. The ideal case is that the first generation of individuals does not pass too many tests, to have greater expectations of improvement through the generations.

Subdivision Sample. We have 99 methods that technique will look for. These methods can be subdivided into the functionalities that comply with the application:

- Accessors: methods that return an instance variable. For example, GP is able to found the method `shape` of the `RSLinePlot` class. This method simply returns shape, a variable that belongs to the same class. We have 33 accessors methods.

```
RSLinePlot >> shape
  "Return instance variable shape"
  ^ shape
```

Listing 3: Example accessor method found.

- Setters: methods that mutate the value of an instance variable. This methods take an argument that will be the new value of the variable. In total, we have 19 setters methods.

```
RSLinePlot >> shape: aShape
  shape := aShape
```

Listing 4: Example setter method found.

- Complex: methods that make calls to other methods. For example `maxValueX` is designed as a complex method. In the sample, we have 43 complex methods.

Weight System. The need to take into account the context in which the searched method is used arises from the fact that GP needs to create the individuals according to the universe of variables and functions, but this set grows considerably considering all the methods and variables accessible by the participating classes. Consequently, GP may explore a restricted small part of the universe or worse, never use the relevant components to generate the expected result. Therefore, it is important to build a technique that assigns values (weights) to each method and variable to influence the way GP selects elements for a new candidate AST. These weights can be considered as probabilities with respect to the total sum of the assigned weights.

For a first approximation, we consider a uniform system weight where all methods have the same probability to be chosen. On the other side, variables have different weights according to the class hierarchy. Previously, it was mentioned that the variables within child classes will have greater weights than the parent classes:

- Variables in the same class, have the same weight w.
- If A is the parent class of B and weight of variables of B is w, then the variables of A have weight equal to $w/2$.

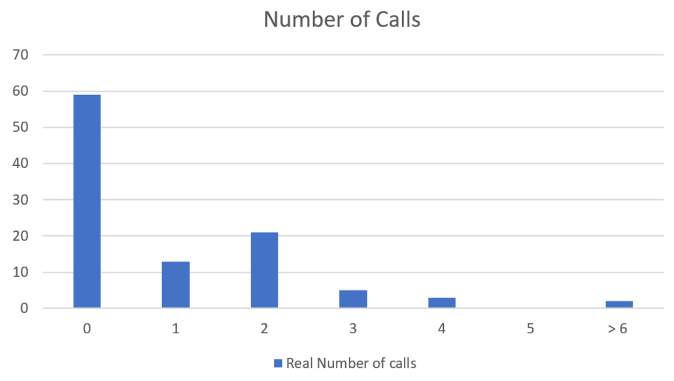

Fig. 2: Number of real calls of the methods in Roassal

Hyper-parameters. Our technique needs these to control execution algorithm. Also, there are some parameters chosen before with prior research. For example, building individuals must consider the OOP context to build consistent programs. A priori, a line of code can generate M arbitrary calls to other methods. In reality, the number of calls is quite limited. In the graph in the figure 2, the methods are distributed according to the number of actual calls they make. These percentages are used in GP, where if $p\%$ of methods make n calls, then with probability p, GP creates an expression that makes n calls to other methods. On the other hand, when performing genetic operations it is possible that the number of calls increases. Therefore, it is arbitrarily decided that it is reviewed whether or not the program complies with a certain number of calls (also decided through probabilities) if the individual does not comply, it is discarded and the genetic ops are applied again. Within the Roassal methods, 57% did not call other methods, but instead returned a value. 33% make between 1 and 2 calls. Therefore, it is sensible for an individual to have very little chance of generating an AST that makes more than 2 calls.

The size of the population used is 25. While the maximum number of generations is 30. The mutation rate is 15%, when GP creates a new generation.

Optimizations. When GP calculates an individual's fitness, it is cached for future reference. Then those variants that are simply copied to the next generation are not recalculated their associated fitness. Logically if an individual changes its AST, the fitness must be calculated.

B. *Experimental Results*

Table I summarizes experimental results where we use uniform system weight to search for 99 Roassal methods. We mentioned that we separate methods in two categories:

- 34 methods directly called by tests (i.e., the test contains a reference to the method), and
- 65 methods that are indirectly called by the tests (i.e., the tests do not contain reference of the method).

The intuition we are here exploring, is that greater the distance in terms of intermediary calls between the method to be generated and the test, more difficult the generation is. Correctly generated methods are those that managed to have a fitness 0. The Time column is the estimate according to the sum of all search processes.

Methods	Total	System Weight	Corrects	Time (Hours)
Directly Calling	34	Uniform	16	34:17:32
Indirectly Calling	65	Uniform	28	65:13:17

TABLE I: Experiments Results with uniform weight system.

C. *Characterization of methods found*

Using uniform weights (i.e., all the methods have the same probability to be chosen when generating the AST), GP correctly generated 44 methods from the 99 methods ($= 44\%$). Table II details the number of correct generation for accessors, setter, and complex methods.

Methods	Directly	Indirectly	%
Accessors	11	19	91%
Setters	0	9	47%
Complex	5	0	12%

TABLE II: Characterization of methods found correctly by Uniform

From this information it is observed that generating accesses does not represent a great challenge for GP even with a uniform selection in. It is only necessary to select the variable (which is part of the search universe) and form an AST of the return type, which has a high probability of occurring. On the other hand, there is the possibility of finding assemblers with success but with less precision. This should be due to the fact that there are more restrictions when constructing an assignment, as the instance variable needs to be matched with the value in question (again, there is a high probability of using the argument).

In the case of complex methods, it is remarkable to consider that it was only possible to construct the semantics necessary to fulfill the tests, without achieving the expected syntax. This implies breaking an important OOP rule about delegating responsibility to the method or object as appropriate. For example, the code in listing 2 is passed to carry the delegation that would correspond to `definedValuesX`.

Another example that confirms a research question is that the solution obtained is sensitive to the quality of the set of tests. The `isVerticalBarPlot` method of the `RSAbstractPlot` class returns false. Then GP will look for a line of code that satisfies this, the solution offered by GP is to call another method call `isLinePlot` of the same class and that also returns false. This could cause problems in child classes when redefining parent methods. The problem is solved by adding a test that ensures that the result of both methods does not have to match according to a context.

V. Experiments with Similarity Weight System

We use a second weight system that consider other particular aspect of OOP. The intuition we are here exploiting is *structural elements, such as class and methods, having similar name are* *likely to have similar code definition*. The rationale is that names given to structural elements may expression aspect of the collaboration the element should have, and therefore, have some similarity in their definition.

The similarity between two strings w_1 and w_2, is calculated using the Levenshtein algorithm [9], which counts the number of inserts, exchanges or deletions that must be done to transform w_1 to w_2. Where the best similarity is 0. In this case, to calculate the weight of an individual I is calculated as:

$$Weigth(I) = C - \alpha * Similarity(I)$$

Where for the experiments we consider $C = 1000$ and $\alpha = 10$. While closer to zero the similarity, the greater the weight assigned to the method or variable, and therefore it is more likely to be chosen.

A. Experimental Results

Table III summarizes our experimental results where we use similarity as system weight for methods and variables.

Methods	Total	System Weight	Corrects	Time (Hours)
Directly Calling	34	Similarity	17	32:18:07
Indirectly Calling	65	Similarity	34	53:16:39

TABLE III: Experiments Results with Similarity System Weight.

B. Characterization methods found

If the information between Table II and IV is compared, it is observed that both systems behave similarly in the case of directly called methods. However, using similarity considerably improves the search in some cases. In particular, they considerably increase the number of setters found, this probably because the similarity is maximum between the formal name of the method and the variable to be mutated. On the other hand, despite having obtained quantitatively equivalent results in the case of complex methods, results were obtained that maintain the logic of delegation of application, unlike the uniform system.

Methods	Directly	Indirectly	%
Accessors	11	21	97%
Setters	3	11	61%
Complex	3	2	12%

TABLE IV: Characterization of methods found correctly by Similarity

VI. Conclusion and Future Work

We present a first version of an automated technique for generating code with wide room for improvement. From the context in which a method lives, it is possible to extract important information to find the required code. This shows the feasibility so that in the future corrective patches can be generated in programs written in Pharo. Our algorithm uses GP combined with a weight system that takes this context into account. This allows scaling to methods where your body is not trivial (return of a value). A fundamental part of the search is the use of test cases that show the functional requirements that the code returned by the algorithm must meet. We are able to generate correctly for 51 different methods of a single line in Pharo Smalltalk out of a total of 99. We note that each weight system used has its own advantages. What motivates to find a harmony between these two systems and to consider other aspects of the context, in such a way that it increases the precision and the technique is able to find both accessors and setters, as well as more complex methods that make calls.

Acknowledgments

We thanks Lam Research and the ANID FONDECYT Regular 1200067 for partially sponsoring the work presented in this paper.

References

[1] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, Automatically finding patches using genetic programming, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 364–374. doi:10.1109/ICSE.2009.5070536.
URL http://dx.doi.org/10.1109/ICSE.2009.5070536

[2] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, 2012, p. 3–13.

[3] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[4] J. R. K. I, R. Poli, A Genetic Programming Tutorial. Introductory Tutorials in Optimization, Search and Decision Support, Chapter 8, 2003.

[5] R.-M. Karampatsis, C. Sutton, How often do single-statement bugs occur? the manysstubs4j dataset, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 573–577. doi:10.1145/3379597.3387491.
URL https://doi.org/10.1145/3379597.3387491

[6] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep Into Pharo, Square Bracket Associates, 2013.
URL http://books.pharo.org/deep-into-pharo/

[7] A. Bergel, Agile Visualization, LULU Press, 2016.
URL http://AgileVisualization.com

[8] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, *SPY: A flexible code profiling framework.* Computer Languages, Systems & Structures (Volume 38) (April, 2012).

[9] G. Hicham, Introduction of the weight edition errors in the Levenshtein distance, IJARAI, 2012.