ANTIS: Another Tool for Identical or Similar Code Detection

Automatic Control and Computers Faculty, University "*Politehnica*" of Bucharest, Romania
bogdan.ghit@gmail.com, matei.gruber@gmail.com

***Abstract*—The goal of this project is to develop an application able to detect the plagiarism in programming assignments by identifying similar code sections. The evaluation process is realized by comparing source code in the abstract syntax trees (AST) representation. Detecting similarity between programs involves finding the maximum sub-graph that can be matched in both trees. The AST is a relational structure that provides a better understanding of the code. We may easily reduce this challenge to the well known tree matching problem, which has been proved to be NP-complete, as there is no deterministic algorithm able to compute the maximum match between two trees. In consequence, we developed several algorithms based on structural analysis of the abstract syntax trees and certain heuristics which are going to be explained in the following sections of the article. The aim of this work is to provide new approaches in the field of code source plagiarism detection that are going to increase both efficiency and accuracy of the matching process.**

Keywords

Code Plagiarism, Abstract Syntax Tree, Tree Matching Problem, Longest Common Sequence, Hash Encoding

I. Introduction

Verifying source code similarity has an important role in the evaluation of programming assignments in universities. The fact that many programs are graded based on the results produced, with optionally a brief analysis of the source code, provides a relatively low risk environment for a student to conduct plagiarism. But, for a typical entry level class, there may be over 300 submissions per assignment which means that checking for plagiarism can easily become an extremely time consuming task.

There are several methods adopted by current tools which we are going to discuss, but none of them offers a complete and suitable application. Our intention is to analyze the features and the characteristics of these algorithms and to explore and develop some new approaches in this field.

During the following sections of this paper we describe the techniques used by existing software in this field, we present our system's architecture with details regarding each major component and then we move the focus over the algorithms that we have developed in order to implement our metrics for comparison. A various set of test cases is analyzed and compared for the metrics that we have implemented. In the end, we draw the conclusions and we establish the next target of the project.

II. Related Work

This section contains a presentation of existing plagiarism detection tools and emphasizes their strengths and weaknesses.

There are many methods by which a program can be transformed into another with similar runtime behaviour. The most common ones are represented by lexical changes and structural changes. The former, does not require knowledge of the language and is related to adding or removing comments, modifying identifier names or changing formatting. The latter is much more interesting, and it is easy to notice that it is language dependent, because it requires modifying functions, by adding, removing or reordering instructions, but with the requirement of keeping the implementation's consistency.

The most common techniques used for plagiarism detection in source code use the *n-gram* algorithm. The idea is to divide the initial document into contiguous substrings of length n (with each character of the document as the starting point of a new *n-gram*) and to compute hashes for each of them. Then a certain subset of the hashes will be selected in order to define the program's fingerprint. If there some identical values for fingerprints are detected in the two documents, it is very likely that they share at least one *n-gram* element. This idea led to many approaches with the goal of increasing both the accuracy and efficiency.

MOSS is a software tool for plagiarism detection that goes further with this algorithm by offering an improvement to ensure that any substring is matched as long as a certain guaranteed threshold t is detected. The idea is to define a window of hashes of size $w = t - n - 1$ and to chose the minimum value or the rightmost occurrence if there is more than one hash value equal with the minimum. A complete description and evaluation of this algorithm is presented by Saul Schleimer, Daniel S. Wilkerson and Alex Aiken in [7].

Even though it has remarkable results, MOSS has its limitations which are more obvious in object oriented languages where the textual analysis cannot cover anymore particular programming techniques like encapsulation, polymorphism, inner classes etc. Moreover, the tool is not really useful when checking programs implemented in different programming languages.

JPLAG was designed by Lutz Prechelt, Guido Malpohl and Michael Philippsen and it is based on a known algorithm called *Greedy String Tiling*. All programs are converted into token

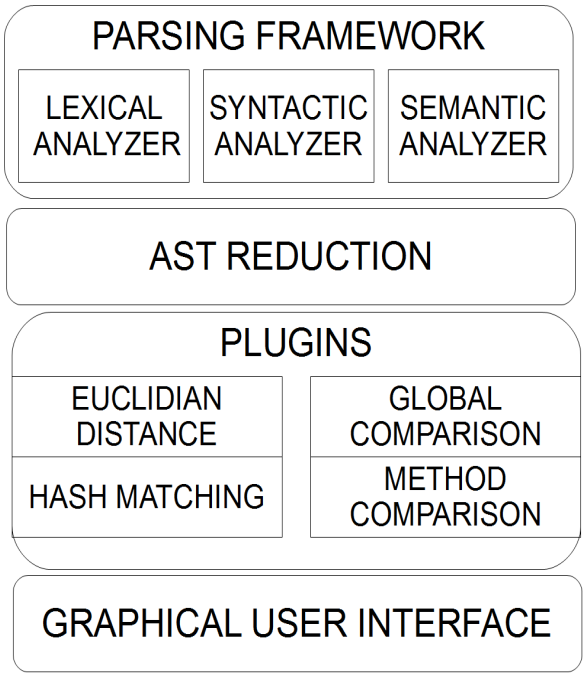

Fig. 1. ANTIS Architecture

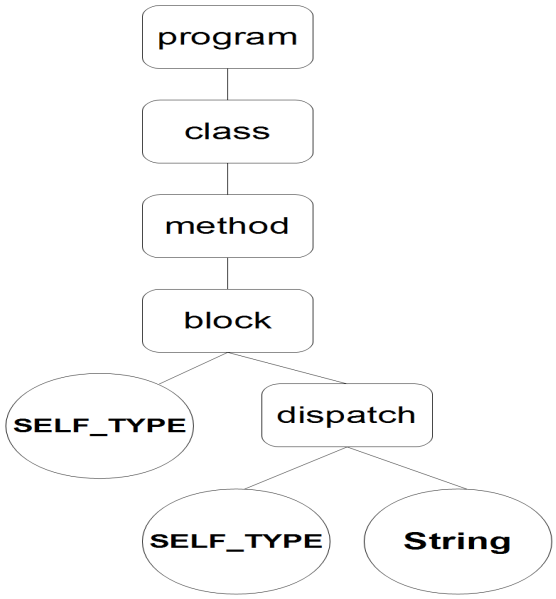

Fig. 2. AST for Hello World program

strings which are going to be compared by trying to cover one token string with substrings called *tiles* taken from the other as well as possible.

A detailed pseudocode and also a critical analysis of performance are provided by its creators in [5]. Although the experiments done for Java programs proved to be successful, the authors confess that the results obtained might apply to a lesser degree for C and C++ and they might also be weaker for differently structured or larger programs.

In [9] Wuu Yang describes an algorithm for tree matching based on dynamic programming which can be used for identifying syntactic differences between two programs. It is inspired from the well known *Longest Common Subsequence* algorithm which is generalized from a sequence of tokens to a sequence of trees. Based on a matrix W with values of 0 or 1, depending on whether the tree at index i from the first sequence is similar to the tree at index j from the second one, a dynamic programming scheme is applied to find the number of pairs in a maximum matching between the two trees. The problem is that this algorithm does not handle well transformations such as method or instructions reordering and the comparison time depends on the amount of differences between files and the places where they occur. The heuristics used for reducing the processing time, such as comparing only methods with identical names, are very simple and most probably less efficient, since one of the most common code transformation is renaming identifiers.

In article [3] Matt G. Ellis and Claude W. Anderson describe a method of coding the parse tree associated to a program by generating a string during a pre-order traversal. The plagiarism can be identified by using the *Greedy String Tiling* algorithm.

Jason T. L. Wang and Bruce A. Shapiro propose in [8] an algorithm for finding the largest approximately common substructures of two trees. An interesting approach for trees comparison based on bio-inspired algorithms is presented by Olfa Sammoud, Christine Solnon and Khaled Ghedira in [6] and a performance analysis of three algorithms for maximum common sub-graph detection on a wide database of graphs is realized in [2] by D. Conte, P. Foggia and M. Vento.

III. System Architecture

This section presents the architecture of our plagiarism detection tool and offers a detailed description of each component illustrated in figure 1. As we have developed the detection tool based on abstract syntax trees comparison, we make a brief description of the first stages of any compiler, we introduce the programming language that we have used for testing our algorithms, we define the targets of the detection engine and we end this section with the description of the interface used for displaying the results.

Our interest was to study and explore new ideas for the problem of plagiarism detection in source code, with the aim of reaching some language agnostic techniques. Considering the fact that our main concern was related to the algorithmic part of the project, we decided to develop this software for a small programming language designed for teaching the basics of compiler construction to undergraduate CS majors.

In spite of its simplicity, the Classroom Object Oriented Language, presented by Alexander Aiken in [1] as COOL, retains many of the features of modern programming languages including objects, inheritance, static typing and automatic memory management. Another reason for choosing

COOL was the fact that it is object oriented, because we also wanted to offer a better solution to other plagiarism detection tools which proved to be inefficient with this programming paradigm.

Listing 1. Hello World program in COOL

```
class Main inherits IO {
    main(): SELF_TYPE {
        out_string("Hello World!\n")
    };
};
```

We illustrate a simple program written in COOL and the associated abstract syntax tree generated according to the COOL grammar in figure 2. Further details will be given during the following sections of the article.

A. Parsing Framework

Lexical analysis breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name.

The token syntax is a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.

Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax.

The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

Semantic Analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings.

Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

B. AST Reduction

One of the project challenges is to ensure a stable framework able to detect similarities between programs implemented in different programming languages. In order to accomplish this we have to introduce an intermediate stage between the parsing process and the actual detection layer of the architecture, which will be able to bring the abstract syntax trees generated from different grammars (for example COOL and C) to a simplified canonical representation. This means, that after applying a certain set of transformations, the two trees will have the same layout, which will make the detection tool a language independent application. This is a theoretical layer, so in this article we will focus on the algorithms that we propose for plagiarism detection and we present the experiments that we have done with COOL programs.

C. Detection Layer

The system architecture is composed of several plug-ins which represent a powerful detection tool able to identify various structural modifications of the source code. We have developed and analyzed five methods: three of them may be considered blind, because they rely on a pure structural comparison of the AST, one of them is a brute-force algorithm which led us to the last and our most important algorithm representing a new approach in the field of tree structural comparison: hashing tree matching.

As we have mentioned before, the detection tool will identify plagiarism in sources that use the following basic methods of transformation from simple to more complicated techniques:

1) *Comments*. Add, remove or change comments is a very simple transformation which must not be neglected since the plagiarist could manipulate the size of the program only by editing the explanations attached to different parts of the original code.
2) *Renaming*. A plagiarist will certainly change the name of identifiers including class names, method names and also variable names, transformations which do not change at all the structure of the original program.
3) *Reordering*. Moving field variable declarations from the top of the source to the bottom, changing the order of the methods, represent a type of transformation which modify the appearance of the code source and might induce in error a naive analyzer.
4) *Splitting*. A big chunk of code can be easily broken into two or even more methods, each of them implementing a small part of the desired functionality. Experienced plagiarists may restructure the program's layout by splitting the content of different methods, which means that the original code is kept without significant transformations, but its position in the program is changed.
5) *Inlining*. The opposite of the splitting method is method combination or in-lining methods. Grouping code together in separate methods makes a program easy to understand and follow, but replacing method invocations with the proper code, will definitely complicate programs structure.
6) *Rearranging*. This transformation implies that different independent program statements can be rearranged without altering the functionality of the original code. Even though it is similar with the reordering transformation, code rearranging is a much more difficult technique, since it is applied to a lower level regarding the program's structure (*reordering* operates with methods, while *rearranging* moves instructions) and the plagiarist should be very careful in order to ensure the functionality of the program.

7) *Inserting code.* A plagiarist may add pieces of code to the program with irrelevant side effects to the main algorithm, in order to make sections of the program to look different. This may include adding assignments to new variables, adding invariant mathematical operations and so on.

Besides these basic types of plagiarism techniques, also mentioned by J. Hamilton in [4], our algorithm will be able to detect other language specific techniques which we are going to present and explain in the results section.

D. GUI Layer

A convenient and conventional way of comparing source code is using a visual diff tool which shows two files at once in vertically split window. We have decided to follow this approach and present the results in a browser window, with each source file in its own half.

For a friendly display of the identical code, we have exported the compared files in HTML and we have marked the portions of code of the reference program that have at least one match in the other one. When selecting a marked expression, the corresponding expressions from the other file are emphasized with a different color. This component represents the graphical user interface illustrated in the system's architecture picture.

IV. Algorithms and Metrics

In this section we present the evolution of our project, by analyzing each feature that we have developed in order to achieve the previous mentioned purpose. We describe several algorithms and metrics and we prove their impact and relevance in detecting similar source code. We also make a critical analysis, by demonstrating the complexity of each algorithm used, by emphasizing the strenghts and weaknesses for each technique and by illustrating their behavior on different test cases.

In the following sections we consider the comparison between two programs P_1 and P_2, with the associated ASTs T_1 and T_2, having n_1, respectively n_2 number of nodes and having the heights h_1, respectively h_2.

A. Euclidian Distance

Considering the fact that our analysis is mostly based on structural comparison, we will introduce several representations of the source code.

A straightforward approach would be to consider programs as points in a vector space, where each component of the vector is represented by a software metric for the given program:

- number of methods
- number of variables
- number of loops
- number of conditionals
- number of methods calls

Then, we'll expect that the more similar programs the programs will be, the closer they'll be to each other in terms of a metric norm.

We quantify the similarity of the sources by computing the Euclidian distance between the two vectors associated to the compared programs.

Having $\mathbf{x}$ and $\mathbf{y}$ the representations of two programs,

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{1}$$

$$\mathbf{y} = (y_1, y_2, \ldots, y_n) \tag{2}$$

we calculate the distance between them with the following formula:

$$d = \sqrt{\sum_{1}^{n}(x_i - y_i)^2} \tag{3}$$

We interpret the value obtained as the syntactic difference between the programs. The lower the distance is, the higher is the probability that the programs are similar and vice-versa, the higher the distance is, the lower is the probability that the programs are similar.

Complexity is proportional to the length of the input, as the metrics can be computed in a single pass, after AST reduction. This leads to an optimal algorithm in terms of speed.

This method succeeds when the programs are very similar and most of the transformations used by the plagiarist are variables, methods and independent statements reordering. The following techniques cannot be detected with this simple metric: declaration of dummy variables, splitting methods in two or more other methods with same functionality as the original one, insertion of dead code.

The limitation of this method is obvious now, considering the fact that a simple transformation as adding several dummy variables (for instance, 100), will lead to a high value of the distance between two programs which share most of the code.

We may also state that two programs might have the same structure in terms of number of variables, number of methods, number of loops etc, but the semantic and even the statements are very different. In the context of programming assignments, there is a good chance that some homeworks are designed after a certain pattern which the students will follow, so in this situation this metric will detect a huge set of identical sources, but most of them will probably prove to be false positives.

Another disadvantage is that we can not detect and indicate the segments of code that are identical in both programs so this method offers only a blind global estimation of the similarity between two programs.

B. Global AST Comparison

A dynamic programming approach for this problem is motivated by the apparent resemblance of the task of comparing two source files to the task of computing the Levenstein distance between two strings. The major difference though, is that the Levenstein distance, which is trivially computed by dynamic programming doesn't apply directly to the more complex changes used by plagiarists.

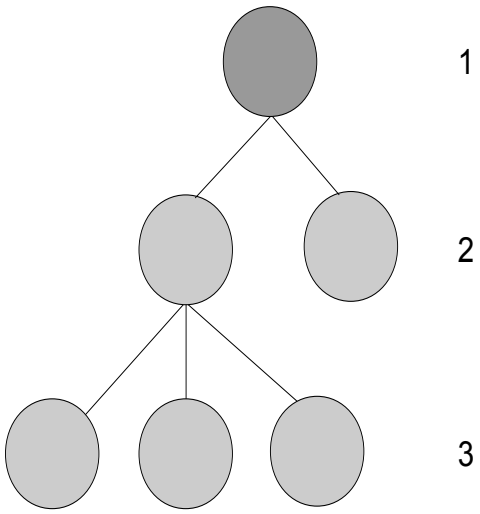

Fig. 3. Tree String Representation

The first algorithm that we have designed for identifying structural similarities between the abstract syntax trees of two programs is based on the intuitive observation that *two trees can be considered similar if they have similar number of nodes per each level of its height*.

There are three basic steps of this algorithm:

Step 1. First, we compute the number of nodes per each level of the compared trees with a level-order traversal algorithm which has a linear complexity: $O(n_i)$.

Step 2. We thus generate a vector representation for each AST, where for a vector $\mathbf{v}$, $\mathbf{v}_i$ is the number of nodes in the i-th level of the three.

Step 3. Third we apply the *LCS* algorithm for the vectors generated at step 2 $\mathbf{v}_1$ and $\mathbf{v}_2$ of lenghts l_1 and l_2 and the result is a vector that represents the best match between the compared trees regarding their levels layout.

This vector quantifies the similarity between the two trees according to the formula 4:

$$s = \frac{\text{len}\left(\text{LCS}\left(\mathbf{v}_1, \mathbf{v}_2\right)\right)}{\max\left(l_1, l_2\right)} \tag{4}$$

For instance, in figure 3 the associated representation of the tree is the the vector $\mathbf{v} = (1, 2, 3)$.

LCS algorithm's complexity is also polynomial depending on the lengths of the two strings: $O(l_1 l_2)$.

The final complexity is composed of the complexity of computing the number of nodes per each level of the AST for both programs and the complexity of finding the longest common sequence between the two trees codifications:

$$C = O(n_1) + O(n_2) + O(l_1)O(l_2) \tag{5}$$

Since l_1 and l_2 are bounded by the number of nodes in each tree, the complexity is trivially $O(n_1 n_2)$

The previous inequality results from the observation that the number of levels of a tree is always less or equal than the number of nodes: the limit case is a tree with one node per each level, so that the total number of nodes is equal with the tree's height.

This algorithm has the ability of detecting similar patterns between two tree layouts. A tree layout is defined as the number of nodes per each level and a pattern represents the maximum set of tree levels with identical layouts.

Even though this algorithm provides a better analysis of the syntactic structure of the programs, it is still a blind method, since it does not make any difference between the nature and type of AST nodes, and this may lead to false positives.

C. Local AST Comparison

In order to reduce the false positives that we have mentioned before, we decided to apply the same algorithm at a lower level in the AST structure. In this manner, we apply the previous algorithm for each pair of methods of the compared programs.

Suppose that each program has m_1 and m_2 methods, we build a matrix W of size $m_1 m_2$, where the element w_{ij} represents the percentage of similarity between the method i from program P_1 and the method j from program P_2.

In this manner we determine the best fit for each method k as the maximum value in line k of matrix W. A global value of similarity of the two programs can be calculated as the medium value of the highest percentages on each line of the matrix.

$$s = \frac{\sum_{i=1}^{m} \max(w_i)}{m_2} \tag{6}$$

A property of this approach is that a method in the second program may be potentially matched against several methods in the first one. This might seem like a disadvantage at first, but it addresses the plagiarism technique of splitting a method into pieces.

This is a better approximation of the structural similarity between trees T_1 and T_2 because the false positives are not propagated between different methods and they remain limited only to the current compared method.

D. Brute-force Tree Matching

As we have noticed, the previous algorithms are global and they do not provide methods for identifying the similar segments of code, they can only estimate a percentage of similarity of the programs regarding their AST layouts.

The purpose of a plagiarism detection tool is to identify those portions of code that are identical or very similar in both programs.

Adding the restriction that two expressions are matched if and only if they are exactly the same, a brute force algorithm will try to compare each expression from P_1 with each expression from P_2 recursively until two different elements are found. If there are no differences, that pair of expressions is a match.

Each node of the AST is a pointer to a COOL expression and also a root of a sub-tree. In other words, a node that contains a terminal (object, integer, string etc.) will match only a node with the same type of terminal, and a non-terminal node will match another node if two conditions are met:

1) both must have the same type (*conditionals, loops, assigns etc.*)
2) their child trees must be identical

For instance, let's analyze the case of comparing two conditional expressions. According to the COOL grammar, a conditional expression can be represented as a tree with the root containing a keyword that indicates the expression's type and three children: *condition*, *then-clause* and *else-clause*. So, two conditional expressions match if they have the same *condition*, the same *then-clause* and the same *else-clause*.

This algorithm's complexity is exponential, and this makes it unacceptable, but there are actually three conditions that have to be respected in order to make this algorithm fail: the programs should be very big, a high percentage of their code should be identical and there should be many levels of imbrications.

As we have described the algorithm, it is recursive and it stops when two different nodes occur, which means in certain situations when the programs don't share a big portion of their code and the copied segments are not represented by expressions with many levels of imbrications, the algorithm succeeds.

In spite of this, the many restrictions that we have indicated are difficult to control and the exponential complexity does not recommend this algorithm. The importance of this algorithm is given by the idea of matching identical expressions, which represents the base of the next algorithm.

E. Hashing Tree Matching

This algorithm guarantees the fact that our tool is an invariant to all the plagiarism techniques that we have mentioned. In addition, it allows us to identify exactly the portions of code that are identical in both sources and it is very efficient, fact which is going to be proved soon.

The tree matching problem, also known as the maximum common sub-tree problem, is NP-complete, so there is no deterministic algorithm able to compute the maximum match between two trees.

As we have seen in the *global AST comparison*, a good heuristic is to associate a codification to the trees and to try to identify similar patterns in those representations.

The codification that we propose at this point is to apply a hash function for the tree so that the resulted value will be an identifier for that kind of tree. We have agreed that each node of the AST is actually a pointer to a COOL expression and can be also viewed as a root of a sub-tree that dominates another expression.

The idea is to assign to each node of the AST a hash value which represents an identifier for that particular kind of tree. The comparison between two trees is immediately reduced to a simple comparison between hash values. *Two trees are going to be considered identical or similar if they have the same hash value stored in their roots.*

The algorithm signals those expressions from program P_1 that have at least one match in program P_2 and has the following steps:

Step 1. The first step is to compute a hash function for each tree node, and then store these values in a hash table. A node in an AST is a COOL expression.

The hash function should be chosen in such manner that it reflects both the relationship between tree nodes (parent-children) and also the type of the expression that it points to. We associate for each type of AST node (*class, method, conditional, loop, assign, let, object, integer, string, bool etc.*) an unique code, which is an integer value.

The hash value of a tree t is obtained by applying a function that combines the hashes of the children, and another function that combines the resulted value with the code of the hashed node as we can see in formula 7.

$$hv(t) = f(code(t), g(child_1, child_2, .., child_n)) \quad (7)$$

where t is a node of the tree, $\mathrm{code}(t)$ is a function that returns an integer value acording to the node's type and $child_1, child_2, \ldots, child_n$ are the node's children.

In order to compute the hash values for each node, a preorder traversal of the AST is required.

Algorithm 1 hash(n)

```
h := e // identity element of g

for all c in children(n) do
  h := g(hash(c), h)
end for

h := f(code(n), h)
T[h] = T[h] ∪ n

return h
```

Two types of collisions may occur in the hash table during generation:

1) Identical sub-trees with the same hash value (trivial case). This cannot be avoided and we do not want that anyway, because they belong to different expressions from the tree.
2) Different sub-trees with the same hash value. This type is less probably to appear and can be avoided with a proper hash function.

After executing these operations for both T_1 and T_2 we obtain two hash tables H_1 and H_2 representing the codifications associated to the compared programs.

Collisions are solved by chaining, as it is shown in the algorithm.

Step 2. At this point, the natural move is to compare the codifications generated at previous stage for the programs.

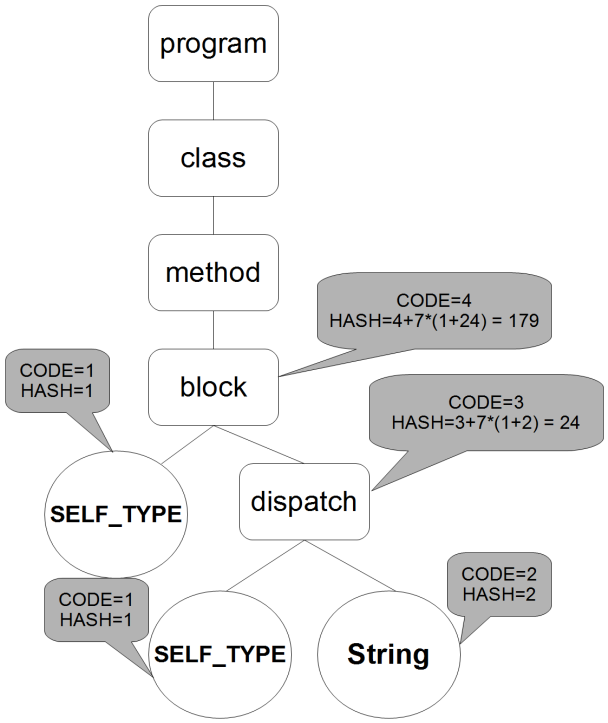

Fig. 4. Hash Encoding Example

This means that the algorithms iterates one of the hash tables and tries to find identical key values in the second one.

The number of key values in the hash table is lower or equal than the number of nodes in the tree, because there can be at most n different kinds of sub-trees in a tree with n nodes.

We therefore need to match trees with same hash values in both the tables. A tree in the the first table may match one in the second if they have the same hash values and:

1) The trees should be identical, which means that if the second type of collisions appear, we have to reject the tree as a suspect. A very simple heuristic help us manage this situation: a pair is marked as match if both trees have the same number of nodes and the same type of roots.
2) If the current list of suspects already contains a tree which is an ancestor of the analyzed tree, it will be ignored, because we want to keep only the maximum match between two expressions.
3) If the current list of suspects contains children of the analyzed tree, it will be marked as suspect, but not before removing all its children.

All basic methods of code plagiarism and more others are detected by this algorithm, fact which is going to be analyzed in the *Experimental Results* section.

An important observation is that by having g be a commutative function, the plagiarism technique of reordering pieces of code is addressed

Another interesting feature of this algorithm is that is allows us to define equivalent expressions which may be very useful in a complex programming language like *C*: for instance, a *while* with a *for*. This can be achieved by ensuring that the hash function provides the same value when it evaluates those types of expressions.

The efficiency of the algorithm can be easily proved by demonstrating the complexity:

- the phase of generating hash values for each node of the AST is done during a pre-order traversal of the tree, which means the complexity is linear $O(n_1) + O(n_2)$.
- the matching process between hash tables keys needs maximum $n_1 n_2$ comparisons, because a table's size is lower than the number of nodes of the AST.

The final complexity is the same as the one of the *Global AST Comparison* algorithm: $O(n_1 n_2)$. We succeeded in providing a better heuristic, with more accurate results on various types of code plagiarism, but more important, the algorithm is also very efficient.

V. Implementation Details

In this section we provide several details regarding the implementation of the tool, including the technologies used, information about the AST structure, an example of hash function that we have used for testing and the way we display the results in html.

The parsing framework was developed in ANTLR and JAVA, the detection algorithms in JAVA and the GUI in HTML and JAVASCRIPT.

The hash values are computed during a post-order traversal of the AST, as we can notice from the pseudocode illustrated in algorithm 1. Each distinct hash value becomes the key of a table which stores a list of nodes that are identified by that hash value.

The hash function that we have used for testing has the following form:

$$hv(t) = code(t) + M \sum_{1}^{n} code(child_i) \tag{8}$$

where M is a prime number which ensures the dispersion of the table.

In figure 4 it is illustrated the hash encoding for the AST associated with the *Hello World* program. For instance, the hash values of the terminals *SELFTYPE* and *String* are identical with their associated codes: 1 and 2.

For the *dispatch* node we obtain the hash value by calculating the sum of the children's hashes which is 3, multiplying it by 7 (which is the prime number from the above formula) and finally by adding the node's code which is 3. The hash value resulted is 24. In the same manner we obtain the *block* node's hash equal to 179 and so on.

In order to mark a tree as a suspect, we need the whole list of its ancestors, because according to the last conditions specified in the previous section, we have to check if either its ancestors, or its children have already been marked.

TABLE I
LIST OF TESTED TRANSFORMATIONS

Test	Transformation
1.	Change variable names
2.	Reorder independent instructions
3.	Insert dead code
4.	Change loop condition
5.	Split a method in more methods
6.	Change method prototype, by adding dummy parameters
7.	Add dummy variables as class members
8.	Extend a class with a dummy interface
9.	Exchange two variables names
10.	Split a complex expression in multiple expressions
11.	Replace equivalent operators
12.	Change methods order in class
13.	Change classes order in program
14.	Code inlining
15.	Additional instructions to an identical segment of code
16.	Identical programs
17*.	Alternative algorithm: iterative and recursive versions

For this, a level-order traversal is required and at each step, the visited node stores a list containing an identifier and the list of identifiers from its direct parent. In this way, the chain of ancestors are propagated to the leaves of the tree.

Regarding, the abstract syntax tree, this is implemented with an array of pointers to their children, fact which makes all tree traversals that we have used to execute in linear time. An AST node contains complete information about its type (what kind of COOL expression represents), the line number where it occurs in the parsed file, the list of its ancestors and the list of its children and also the most important element of our last algorithm, the hash codification.

We also developed a very convenient visual tool for comparing source files. The tool takes as input the annotated token stream along with the hash values computed for each program element.

Since the hash matching algorithm is local, we already have information about the matching pieces of code, which can be displayed by simply hovering the mouse over the interesting sections of code. This is big advantage over global metrics, since it allows manual inspection of the reported plagiarism cases.

The pretty-printing is realized by intercepting the tokens flow resulted from the parser phase of the program. The tree structure of the parsed program is simulated with html *span* tags having as *id* attribute the unique integer value associated to each AST node during the pre-order traversal. In the same time, the identifiers of the matched nodes are passed to a javascript that handles mouse events and highlights the identical elements of the programs.

VI. TEST SCENARIOS

In this section of the article we present a large set of tests that we have used for the evaluation of ANTIS and we illustrate and explain the results obtained.

We have tested the application on programs containing a large list of transformation usually used by plagiarists. These are enumerated in table I. ANTIS has an excellent behavior on this these methods of plagiarism and detects the copied instructions even if the code is rearranged or is masked with equivalent operators.

The last record of the table is marked with a star because it represents a special case of test, when we tried to implement algorithms in different ways, for example an iterative method and a recursive one. These tests prove that the tool is well guarded against false positives, since they cannot be considered similar because they use a different programming technique.

The results obtained on a set of 24 tests with programs of different sizes transformed according to the techniques specified in I are presented in table II.

The file size is given in number of lines (*nol*) and the similarity between two files is given as a percentage which represents the number of matched nodes from the AST of the reference file.

In test 1 we modified a program by reordering the clauses of two conditional expressions. The rest of the file remains the same, so that the similarity is almost 90% and the execution time is 114 ms.

Two identical programs with large sizes (over 400 lines), but with methods order changed represent test number 2. The similarity is maximum and the execution time is around 700 ms. The same technique is applied in test 16 too with the same results.

Test 3 sustains the fact that the tool is not tricked by false positives, as we coded the same algorithm with two different techniques: recursive and iterative. The similarity is around 47% because of the small size of the programs 24 and 32 lines, but the only matched segments of code are the *Main* methods.

Test 4 contains another common transformation, which changes the loop direction. In spite of this, ANTIS detected the identical code placed in the loop instruction in 89 ms, so that the percentage of similarity is very high: 81%.

Test 5 starts with a method that generates the prime numbers and modifies it by splitting different parts of it in other methods such as: *checkCondition, executeLoop*. The execution time is insignificant and the similarity is almost 90%.

A similar strategy is applied in test 9 with the same good results. The same operations are realized on a *bubblesort* algorithm in test 14.

In test 6 we combine three techniques: we add a dummy method, we change the methods order and we insert a method invocation into an identical segment of code. The execution time is very small and the similarity is exactly how we expect to be: 79%.

Test 7 inserts a set of dummy class variables, transformation that is detected by ANTIS which highlights the whole program as being identical with the other one.

Test 8 uses an object-oriented technique called *upcast*: instead of instantiating a certain class, we instantiate a dummy class that inherits from the original one. ANTIS returns

TABLE II
PERFORMANCE

Test	Files Size (nol)	Execution Time (ms)	Similarity (%)
1.	48/48	114	89
2.	427/407	710	100
3.	24/32	10	47
4.	83/83	89	81
5.	41/54	12	89
6.	24/32	57	79
7.	50/70	11	100
8.	353/368	114	100
9.	24/31	9	75
10.	70/90	15	79
11.	265/320	15	79
12.	103/105	23	82
13.	427/435	214	97
14.	166/166	84	93
15.	416/420	122	100
16.	196/197	45	97
17.	144/143	31	100
18.	53/58	10	85
10.	416/420	163	100
20.	11/20	6	53
21.	52/52	8	100
22.	35/44	8	87
23.	957/1078	277	82
24.	556/1078	446	95

a maximum similarity between the two programs and the execution time is around 350 ms.

Test 10 inserts dead code in the original program, but this proves to be an unsuccessful method too, because our tool marks immediately the identical segments, returning a similarity of 80%.

In test 11 we applied another object-oriented method, by adding an unused interface to the original class, while in test 12 we added some dummy parameters to the methods. In both cases, ANTIS displays a similarity close to 100% with a very small execution time too.

Test 13 we modify a large program with more than 400 lines, by using a recursive algorithm for computing factorial, instead of an iterative one in a certain method. The rest of the program remains unchanged and ANTIS highlights it as identical, while the factorial function remains unmarked. The similarity is close to 100% and the execution time is 214 ms.

In tests 16 and 17 we rename different variables and methods and the similarity is maximum.

In test 18, instead of having a method and calling it, we simply replace the invocation with its code. In this case the method's code is still matched and the similarity is 85%.

Other tests using equivalent operations, expression split, loop split or even identical code had the same good results as the previous ones.

Tests 23 and 24, which represent comparisons between large programs that combine the complete set of transformations that we have suggested in the previous tests, are meant to confirm the efficiency of the algorithm.

VII. Conclusions and Future Work

We have developed a set of heuristics that offer a new perspective in the field of source code plagiarism. Our ideas and algorithms are analyzed in terms of correctitude, complexity and efficiency and are supported by a large set of tests that cover the most common techniques used for modifying the programs.

We also present the evolution of our project, by analyzing each feature that we have developed in order to achieve the previous mentioned purpose. We describe several algorithms and metrics and we prove their impact and relevance in detecting similar source code. A critical analysis is done by demonstrating the complexity of each algorithm used, by emphasizing the strengths and weaknesses for each technique and by illustrating their behavior on different test cases.

After studying other tools for detecting plagiarism, we realized that most of them make a textual analysis of the sources, fact which might not reflect the complex and non-linear structure of a programming language. The AST is a relational structure that provides a better understanding of the code, fact which convinced us to make a tree based comparison of the programs.

At this point, we have easily reduced the challenge to the well known tree matching problem, which has been proved to be NP-complete, as there is no deterministic algorithm able to compute the maximum match between two trees. In consequence, we developed several algorithms based on structural analysis of the abstract syntax trees and certain heuristics to identify identical sub-trees.

As future work we plan to extend the parsing framework in order to test the tool on C and JAVA programs and we also want to explore the idea of finding a canonical AST representation that will allow us to compare programs written in different programming languages.

References

[1] A. Aiken. Cool: A portable project for teaching compiler construction. *ACM Sigplan Notices*, 31(7):19–24, 1996.

[2] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[3] M.G. Ellis and C.W. Anderson. Plagiarism Detection in Computer Code. 2005.

[4] J. Hamilton. Static Source Code Analysis Tools and their Application to the Detection of Plagiarism in Java Programs.

[5] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[6] O. Sammoud, C. Solnon, and K. Ghedira. Ant algorithm for the graph matching problem. *Lecture Notes in Computer Science*, 3448:213–223, 2005.

[7] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 85. ACM, 2003.

[8] J.T.L. Wang, B.A. Shapiro, D. Shasha, K. Zhang, and K.M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.

[9] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.