Detection of Source Code Similitude in Academic Environments

ANDRÉS M. BEJARANO, LUCY E. GARCÍA, EDUARDO E. ZUREK

Universidad del Norte, Ingeniería de Sistemas, Km. 5 Antigua vía a Puerto Colombia, Barranquilla, Atlántico, Colombia

Received 12 June 2012; accepted 2 December 2012

ABSTRACT: This article presents a proposal for the detection of programming source code similitude in academic environments. The objective of this proposal is to provide support to professors in detecting plagiarism in student homework assignments in introductory computer programming courses. The developed tool, CODESIGHT, is based on a modification of the Greedy String Tiling algorithm. The tool was tested in one theoretical and three real scenarios, obtaining similitude detections for assignments ranging from those that contained code without modifications to assignments containing insertions of procedural instructions inside the main code. The results verified the efficiency of the tool at the first five levels of the plagiarism spectrum for programming code, in addition to supporting suspicions of plagiarism in real scenarios. © 2013 Wiley Periodicals, Inc. Comput Appl Eng Educ 23:13–22, 2015; View this article online at wileyonlinelibrary.com/journal/cae; DOI 10.1002/cae.21571

Keywords: plagiarism; computer programming; source code

INTRODUCTION

Software development requires each programmer to generate creative ideas that provide computational solutions for problems with certain degrees of complexity. There are many academic programs that require an introductory course in computer programming. The purpose of these courses is to develop the students ability to implement logical solutions to a problem (represented via a certain algorithmic scheme) using a particular programming language. Such skills are evaluated through the assignment of exercise problems that the student has to solve in an algorithmic fashion using programming code. During the academic evaluation process, the course instructor must verify that the student's solution code complies with the required restrictions and that the provided solution is unique and clearly differentiable from the solutions provided by his fellow classmates.

The inherent complexity of the course described above, as well as the conceptual, ethical, and pedagogical limitations of such courses, produces situations in which plagiarism can occur. To obtain a satisfactory grade, certain students commit this type of fraud, which is initially manifested in the modification of certain lines of code. However, as the student's knowledge increases, this type of behavior becomes more elaborate and therefore more difficult to detect for the class instructor. To detect these cases, a one by one comparison of students' assignments is required, which is a grueling task with a high degree of complexity. Additionally, the procedure must take into account which of the different conditions that define a certain specific situation allow the case to be considered as plagiarism.

To control plagiarism in computer code developed in academic environments, the use of simple Unix commands such as grep, diff, and wc [1] up to more complex tools such as GPLAG [2], JPLAG [3], MOSS [4], SHERLOCK [5], and YAP3 [6] has been implemented. Several implementations have been introduced for software plagiarism detection using token-based analysis [7], algorithm filter [8], and the analysis from the point of view of the teacher [9,10]. Different techniques have been implemented like Natural Language Processing (NPL) [11], Singular Value Decomposition (SVD) over algorithms [12], fingerprint based distance measure [13], analysis from the generated assembler code [14], program characterization [15], and adaptive local alignment of keywords [16]. Although these tools are aimed at detecting similarity in computer programs, this detection is focused on the lexical similarity (word for word) of the codes. However, this type of similarity is insufficient when identifying those cases where copying involves the essence of the program, with a more than superficial modification of the original source code [1]. For this reason, there is a need to develop a tool that facilitates the detection of those cases where the similarity is present at both

Correspondence to E. E. Zurek (ezurek@uninorte.edu.co).

the lexical and syntactic levels. The proposed solution should be evaluated in a manner that would determine the following:

- The ability of the tool to detect copies within the first five levels of plagiarism according to the spectrum of plagiarism in programming codes 6, regardless of complexity or code size.
- Responses to real cases of plagiarism reported previously in a certain course.
- The ability to facilitate the identification of those cases in which false positives could appear.

The complexity of the problem in question is due to the different forms in which a student can cheat, as well as the pedagogical conditions under which programming courses facilitate learning within a controlled environment as proposed by the Jarpeb system authors [17], the CTPracticals module authors [18], or the Technical University of Madrid (UPM) laboratory work management framework [19]. An analysis of these restrictions [20] has determined that the degree of similarity among the students' assignments can arise from different causes and cannot be attributed exclusively to a specific type of plagiarism. The class instructor can substantially influence the degree of similitude among the returned assignments via some of the following factors: examples described in class, problems defined with strong restrictions that limit the number of possible distinct solutions, instances in which the answers may be clearly defined from the beginning, and cases where the students can work out solutions developed by the instructor or by a text guide.

The cases listed above differ from those where students copy the code, modify it, and present it as if it were of their own authorship. These changes range from the modification of comments, text strings, and variable names to changing the program structure. Among the most common modifications [21,22] are the following:

- Code textual copying.
- Adding, modifying, and/or eliminating comments.
- Changing blank spaces and formatting (sangria).
- Changing the name or identifier types.
- Reordering operators and operands.
- Reordering blocks of code and sentences within blocks of code.
- Replacing sentences with their equivalents.
- Adding sentences or redundant variables.
- Reordering independent sentences.
- Replacing control structures with their equivalents.
- Replacing procedure calls by the procedure body and including non-structured sentences.

The variety of modifications that can raise suspicion of plagiarism can be organized into two groups [23]:

- *Lexical changes*: those that do not require a grammatical analysis or a deep understanding of programming to be effective. Some examples are the removal of comments, changes to the format of the source code, and changing the names of variables.
- *Structural changes*: these require a greater knowledge of both programming techniques and the language itself. These changes are highly dependent on the language; this group includes changing the control structures, rearranging blocks and sentences, embedding the body of a procedure within the main code, and the rearranging of operators and operands within sentences.

As more elaborate techniques are used to modify the code, it becomes more difficult to detect similitude. In Ref. 1, a classification that includes seven levels of increasing complexity is defined, ranging from an exact copy of the code to changes to the program logic. These levels are as follows:

- *Level zero*: The plagiarized code is an exact copy of the original source code.
- *Level one*: Only comments are modified (annexed, modified, and/or eliminated).
- *Level two*: Variable names are changed.
- *Level three*: Some variables are transformed, such as a change in the scope (from local to global scope, or vice versa), the implementation of arrays in a set of variables, or any other kind of modification of data structure (e.g., a matrix transformed into a set of vectors).
- *Level four*: The body of a procedure is included in the main code or one or several segments of the main code are transformed into procedures.
- *Level five*: Sentences are modified in the source code. This includes structural changes to the program, reordering of operators and operands, changes in the control sentences (if, switch, while, do, for, etc.), and changes in the data types.
- *Level six*: The logic of the program is modified.

Levels zero to three correspond to lexical modifications, whereas in levels four to six, changes occur on a structural level. In this second group, the code may or may not be considered a copy because its structure is very different. The changes at the structural level are more difficult to detect because it is necessary to know the student's programming skills to determine whether such similarities are caused by plagiarism or as a product of their skill level.

Based on an analysis of the previously raised issues, the development of a screening tool is proposed for detecting code similarities between computer programs generated in academic settings under the paradigm of structured programming. The evaluation of similarity should be performed at both the lexical and syntactic levels and be focused on the identification of cases classified within the first five levels of the previously defined spectrum of plagiarism.

DEVELOPMENT OF THE PROPOSED SOLUTION

Based on the analysis conducted above, the main functionalities that should be included in the tool were determined:

- Analyze the code without taking into account some of the features provided by the developer, such as the names of variables, the order of the blocks of code, and their corresponding format.
- Show the detailed locations of the similitude.
- Detect the code similitude efficiently.
- Because the tool will be applied in an academic setting, it should be easily implemented in such a scenario.

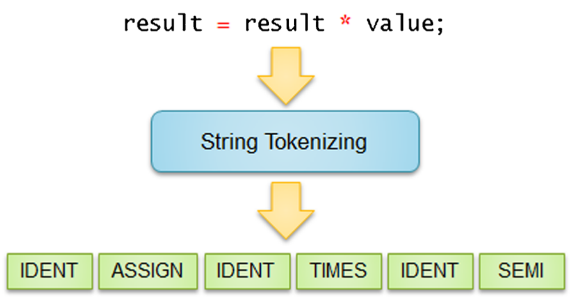


Figure 1 Example of a sentence represented by String Tokenizing. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

In the above context, different techniques were analyzed for detection Halstead profile [1], Software Entropy [24], Conceptual Graphs [25], String Tokenizing [3], Program Dependence Graphs [26,27], Neural Networks [28], and Code Linearization [29], and String Tokenizing was chosen as a strategy for transforming the code into a system that is easier to interpret. This technique allows for comparisons between segments of code based on the essence of the code, leaving aside those factors specific to the developer such as the names of the variables, the order of sentences, or order of the code structure in procedures.

With the String Tokenizing technique, the code is translated into a series of tokens (Fig. 1) representing both the reserved words and any other code element that can be manipulated by the programmer (variables, static text strings, etc.). This process is dependent on the programming language; therefore, for each language, the rules that allow the recognition of such tokens must be defined. In this case, they were defined to be implemented in the C++ programming language.

The tokens characterize the essence of the program structure (which is difficult for the plagiarizer to change) instead of focusing on the superficial aspects of the code [3]. This code representation focuses on what the code does rather than the lexical items introduced by the programmer, which are irrelevant for identifying a copy.

Once the conversion is completed, a comparison of the sequences of tokens is performed using the Greedy String Tiling algorithm [30] (Fig. 2). This algorithm compares two sequences of tokens and determines which are continuous, depending on the specified threshold. The threshold, defined as the allowed number of similar consecutive tokens, is what ultimately determines whether two sequences of exactly equivalent tokens are considered suspected copies or not.

The analysis of token sequences is divided into two steps [3]:

- Find the matches in both sequences; this is done with the three initial cycles contained between lines 5 and 28. The first cycle runs through all the tokens in sequence A, the second runs through sequence B, and the third searches the end of the mutual sequence. At the end of the three cycles, the similar sequences surpassing the input threshold are obtained.
- Mark all of the matching sequences that do not overlap (a token that is inside a matching sequence cannot appear in another sequence).

The measure of similitude between sequences A and B must reflect the number of tokens belonging to the original programs that are inside the matching sequences [3]:

$$sim(A,B) = \frac{2 \times coverage(tiles)}{(|A| + |B|)} \tag{1}$$

where

$$coverage(tiles) = \sum match(a,b,length) \in tiles \tag{2}$$

And, *length* is the length of the matching sequences found between *A* and *B*.

The selected technique has a detection scope up to the fifth level of similarity (combining procedures with the main code and vice versa). The similarity score is a numerical expression of how alike two program codes are. The similarities are detected at different levels: comments, variables (location and use), and control structures (conditionals, loops, and jumps to procedures). The similarity detection depends on a preset threshold, which determines how tight the detection is: the higher the threshold, the higher should be the similarity between two programming sequences to be considered suspicious of being plagiarized; on the other hand, if the threshold is too small, even a programming language keyword could be considered suspicious of being plagiarized. The threshold setting depends on the complexity of the programs to be analyzed, that is, for programs developed by first year students the threshold should be higher than for programs developed by last year students, due that first year students solve simple problems that have a small set of possible solutions. According to the authors of JPlag [3] a threshold set between 30 and 50 seems to approach conclusive results.

For the purpose of developing a simple comparison between distinguishable token sequences in the codes, a modification of the Greedy String Tiling algorithm is proposed. The aspects considered are the following:

- Execute a single overall pass over the two sequences under comparison. In other words, once the sequences are scanned, do not perform a rescanning regardless of whether the maximum number of consecutive identical tokens in both codes is greater than the allowed threshold.
- Organize the matching sequences by length, giving higher priority to those strings having greater lengths, which constitute the best evidence of similitude between a pair of sequences.
- Avoid overlap only when all of the matching sequences of tokens have been identified.

The proposed comparison algorithm is illustrated below (Fig. 3):

This algorithm finds all sequences of tokens shared between a pair of sequences whose length is greater than or equal to the predefined threshold. Finally, the sequences are organized by length and the overlap from the longest to the shortest length is verified. Sequences (equivalent to a fragment of code) of longer length are weighted more heavily. Based on this modification, the CODESIGHT tool was developed, which analyzes a set of source code and identifies those fragments that exhibit similarity at the lexical and syntactic levels (levels from zero to four in the spectrum of plagiarism). The tool allows the user to set the similarity threshold, which determines the minimum

```
Greedy-String-Tiling(String A, String B) {
   tiles = {};
   do {
      maxmatch = M;
      matches = {};
      Forall (unmarked tokens Aa in A) {
         Forall (unmarked tokens Bb in B) {
            j= 0;
            while (Aa + j == Bb + j && unmarked(Aa + j) && unmarked(Bb + j)) {
               j++;
            }
            if (j == maxmatch) {
               matches = matches ⊕ match(a, b, j);
            }
            else {
               if (j >maxmatch) {
                  matches = {match(a, b, j)};
                  maxmatch = j;
               }
            }
         }
         Forall (match(a, b, maxmatch) ∈ matches) {
            For (j = 0. . . (maxmatch- 1)) {
               mark(Aa + j);
               mark(Bb + j);
            }
            tiles = tiles ∪ match(a, b, maxmatch);
         }
      }
   }
   while (maxmatch > M);
   return tiles;
}
```

Figure 2 Greedy String Tiling algorithm [30].

length that a matching sequence should have between the two codes.

EVALUATION METHODOLOGY

Four tests were applied to determine the efficacy of CODESIGHT.

Test 1

The objective of this test is to evaluate the ability of the tool to detect copied sections of program code at the different levels of the spectrum of plagiarism. An iterative version of the factorial number calculation was used as the original program. Using this as a base, modifications were introduced (lexical and syntactic) that corresponded to each of the seven levels of plagiarism, ultimately creating a total of seven modified codes for implementing comparisons. In the case of level 6 (the seventh in the spectrum of 0 to 6), the factorial calculation was performed recursively to establish a change in the program logic.

Test 2

In this case, the tool was tested on 15 programming laboratory assignments (out of 30), chosen at random from an Introduction to Programming course (reflecting the actual environment where the tool would be frequently used).

```
Greedy-String-Tiling-Modificado(String A, String B) {
   tiles = {};
   maxmatch = M;
   matches = {};
   forall (unmarked tokens Aa in A) {
      forall (unmarked tokens Bb in B) {
         j = 0;
         while (Aa + j == Bb + j && unmarked(Aa + j) && unmarked(Bb + j)) {
            j++;
         }
         if (j >= maxmatch) {
            matches = matches ∪ {match(a, b, j)};
            maxmatch = j;
         }

      }
   }
   sort_desc(tiles);
   forall (tokens t in tiles) {
      mark(t);
   }
   return tiles;
}
```

Figure 3 Modified Greedy String Tiling algorithm.

The test was conducted as follows: each of the students was assigned one out of three possible statements. The students proceeded to develop an application that would solve the problem, and then 15 files were randomly selected to be analyzed with the tool. This test was the first in which the tool was used to detect similarities in programs that were developed by different people, with different problems and different logics and interpretations (even in those cases with coincidental problem statements).

Test 3

This test is essentially similar to test 2 presented above, but implemented in a different class for the same course. Unlike in the previous test, all files were compared against each other to find all suspected cases of copying. This test was used to analyze the performance of the tool in a real situation in which 14 cases of similarity were suspected to be plagiarism (in fact, the students admitted cheating when the evidence was presented as a product of the comparisons made by CODESIGHT).

Test 4

This test was performed to identify those rare cases where CODESIGHT can generate false positives due to the restrictions imposed by the threshold value.

The test consisted of a comparison of two programs written in C++, differing both in their solution logic and in the results. One program performs a calculation of the weekly hours an employee works according to a common statement in the assignment exercises handled in the course. The other calculates the greatest common divisor (GCD). Although the codes do not share the same logical structure (as they do not perform the same task), the header statements and some control structures are the same due to language specifications.

RESULTS

Results of Test 1

The codes were compared using the tool with a threshold of 30 consecutive tokens, obtaining the results shown in Table 1.

The obtained data show that as the characteristics of the plagiarism attempt become more complex, a lower percentage of similarity occurs between the original and copy: by omitting the comments, the tool considers files L0.cpp and L1.cpp to be exactly equivalent. The second file (L1.cpp) differs from the first one only in the comments (second level in the spectrum of plagiarism) (Fig. 4).

When comparing files L0.cpp and L1.cpp against L2.cpp, the tool also indicates a 100% similarity between them. This is because file L2.cpp represents the second level of plagiarism (renaming of variables). When performing a syntactic analysis of the code, the name given to the identifiers makes no difference; therefore, the modification of these identifiers does not alter the result of the comparison.

The tool yields a 30% similarity between files L5.cpp and L6.cpp. Although these files have only four lines in common, the percentage is high because these are the programs with the lowest number of code lines; L5.cpp changes one of the control structures (Fig. 5), whereas L6.cpp alters the logic of the algorithm (Fig. 6), changing it from an interactive into a recursive mode.

The comparison between files L0.cpp and L4.cpp (representing levels zero and four of plagiarism, respectively) shows that the data reading segment and the pause segment at the output of results were recognized as similar in both codes. By lowering the threshold to 20 tokens, the percentage of similarity was higher and an additional segment of copied code was identified: the instructions for calculating the factorial (target analysis). This case illustrates the fourth level of plagiarism: inserting the code for a procedure within the main program.

After reducing the threshold to 15 tokens, CODESIGHT detected a new code segment corresponding to the header files declaration. The type of program that was analyzed requires the declaration of the same headers; therefore, the implementation of these headers will be present in all of the programs, and the detection of the similar headers yields a false positive.

The results obtained by CODESIGHT for Test 1 compared to MOSS showed that, in order to generate its results, CODESIGHT takes in account C++ language characteristics such as headers declarations and control structures definition; such considerations make the similarity scores bigger. For this test, the cases where CODESIGHT generates a 100% similarity respect to the results generated by MOSS are related to the aspects mentioned before. The scores obtained after comparing L0.cpp to L4.cpp, L5.cpp and L6.cpp trend to diminish due that these codes have been modified in regard to L0. The results of comparing CODESIGHT and MOSS are show in Table 2 and Figure 7.

Results of Test 2

The initial test was executed with a threshold of 30. The obtained results do not reflect interesting cases, except between A13.cpp and A15.cpp, for which a similar code segment was detected (Fig. 8).

Table 1 Results of Test 1 With a Threshold of 30 Tokens

$U = 30$	L0.cpp (%)	L1.cpp (%)	L2.cpp (%)	L3.cpp (%)	L4.cpp (%)	L5.cpp (%)	L6.cpp (%)
L0.cpp	—	100.0	100.0	83.45324	50.193047	25.498009	0.0
L1.cpp	100.0	—	100.0	83.45324	50.193047	25.498009	0.0
L2.cpp	100.0	100.0	—	83.45324	50.193047	25.498009	0.0
L3.cpp	83.45324	83.45324	83.45324	—	54.05405	25.498009	0.0
L4.cpp	50.193047	50.193047	50.193047	54.05405	—	29.310345	0.0
L5.cpp	25.498009	25.498009	25.498009	25.498009	29.310345	—	30.04292
L6.cpp	0.0	0.0	0.0	0.0	0.0	30.04292	—

```
//
// Programa que calcula el factorial de un número
//

#include <cstdlib>
#include <iostream>

using namespace std;

int factorial(int num) {
   int resultado = 1;
   for(int i = 2;i <= num;i++) {
      resultado = resultado * i;
   }
   return resultado;
}

int main() {
   int numero = 0;
   int valor = 0;
   cout << "Calculo del factorial de un numero." << endl;
   cout << "Ingrese un numero: ";
   cin >> numero;
   valor = factorial(numero);
   cout << "El factorial de " << numero << " es: " << valor << endl;
   system("PAUSE");
}
```

Figure 4 Source code of file L0.cpp.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int number = 0, value = 1, index = 2;

//Programa principal
int main(){
   cout << "Calculo del factorial de un numero.\n";
   cout << "Ingrese un numero: ";
   cin >> number;
   while(index <= number){
      value *= index;
      index++;
   }
   cout << "El factorial de " << number << " es: " << value << endl;
   system("PAUSE");
}
```

Figure 5 Source code of file L5.cpp.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int number = 0;

//Funcion que calcula el factorial
int factorial(int num){
   if(num == 1)
      return 1;
   else
       return num * factorial(num - 1);
}

//Programa principal
int main(){
   cout << "Calculo del factorial de un numero.\n";
   cout << "Ingrese un numero: ";
   cin >> number;
   cout << "El factorial de " << number << " es: " << factorial(number) << "\n";
   system("PAUSE");
}
```

Figure 6 Source code of file L6.cpp.

Table 2 CODESIGHT Versus MOSS, Test for File L0

	MOSS 10 (%)	MOSS 100 (%)	MOSS 1000 (%)	MOSS 10000 (%)	Codesight 20 (%)	Codesight 30
L0–L1	96.00	96.00	96.00	96.00	100.00	100.00
L0–L2	96.00	96.00	96.00	96.00	100.00	100.00
L0–L3	78.00	78.00	78.00	78.00	83.45	83.45
L0–L4	0.00	0.00	0.00	0.00	70.27	50.19
L0–L5	0.00	0.00	0.00	0.00	25.50	25.50
L0–L6	0.00	0.00	0.00	0.00	16.15	0.00

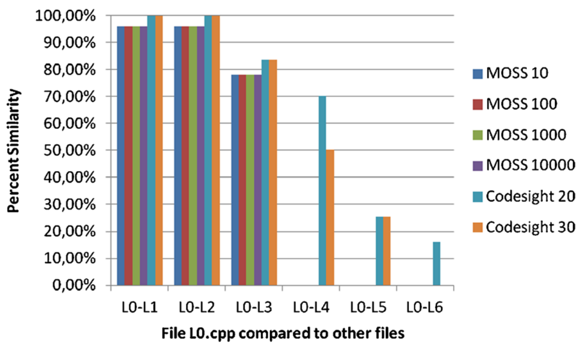
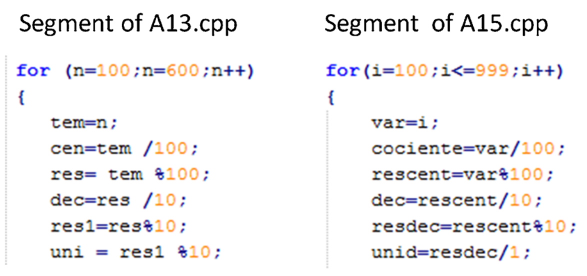


Figure 7 CODESIGHT versus MOSS, test for file L0. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]


Segment of A13.cpp

```
for (n=100;n=600;n++)
{
    tem=n;
    cen=tem /100;
    res= tem %100;
    dec=res /10;
    res1=res%10;
    uni = res1 %10;
```

Segment of A15.cpp

```
for(i=100;i<=999;i++)
{
    var=i;
    cociente=var/100;
    rescent=var%100;
    dec=rescent/10;
    resdec=rescent%10;
    unid=resdec/1;
```

Figure 8 Similar code segments between files A13.cpp and A15.cpp. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Because the programs are running different tasks, it is curious to observe similar patterns between them. When performing the same test with a threshold of 20, the test results were not very different from the higher threshold results. When threshold was lowered to 10, an increase in the similarity percentages could be noticed. Despite this increase, verification indicated similarity in the statements and data output in which no copying existed.

Compared to MOSS, due to the CODESIGHT considerations (headers and control structures), these comparisons are more detailed and rigorous. For these cases, the scores indicate similarities at common functions and header definitions that the students have to work with. These results are shown on Table 3 and Figure 9.

Results of Test 3

First, a threshold of 30 was applied; this resulted in the first formal case of copying so far. With an 88.89% similitude, files B5.cpp and B7.cpp exhibit textual copying of content. Other cases with elevated percentages of similarity only indicated the use of the same libraries in the code headers. After reducing the threshold to 20 tokens, no major changes in similitude appeared. One case in particular stands out in this last comparison: although they have only a 28% similitude, files B7.cpp and B12.cpp include a common condition evaluated with the same restrictions (Fig. 10).

This result indicates that the percentage of similarity is not in itself a decisive value for determining whether there is plagiarism, as in this case the percentage similarity was not

Table 3 CODESIGHT Versus MOSS, Test for File A2

	MOSS 10 (%)	MOSS 100 (%)	MOSS 1000 (%)	MOSS 10000 (%)	Codesight 20 (%)	Codesight 30 (%)
A2–A1	0.00	0.00	0.00	0.00	14.94	14.94
A2–A3	0.00	0.00	0.00	0.00	0.00	0.00
A2–A4	0.00	0.00	0.00	0.00	17.36	17.36
A2–A5	0.00	0.00	0.00	0.00	11.05	0.00
A2–A6	0.00	0.00	0.00	0.00	5.45	0.00
A2–A7	0.00	0.00	0.00	0.00	0.00	0.00
A2–A8	0.00	0.00	0.00	0.00	23.89	14.57
A2–A9	0.00	0.00	0.00	0.00	5.83	0.00
A2–A10	12.00	12.00	12.00	12.00	19.19	13.51
A2–A11	12.00	12.00	12.00	12.00	14.50	8.65
A2–A12	0.00	0.00	0.00	0.00	0.00	0.00
A2–A13	0.00	0.00	0.00	0.00	8.59	0.00
A2–A14	0.00	0.00	0.00	0.00	0.00	0.00
A2–A15	0.00	0.00	0.00	0.00	0.00	0.00
A2–A16	19.00	19.00	19.00	19.00	16.35	11.37
A2–A17	0.00	0.00	0.00	0.00	12.28	12.28
A2–A18	0.00	0.00	0.00	0.00	16.02	0.00
A2–A19	0.00	0.00	0.00	0.00	8.63	8.63

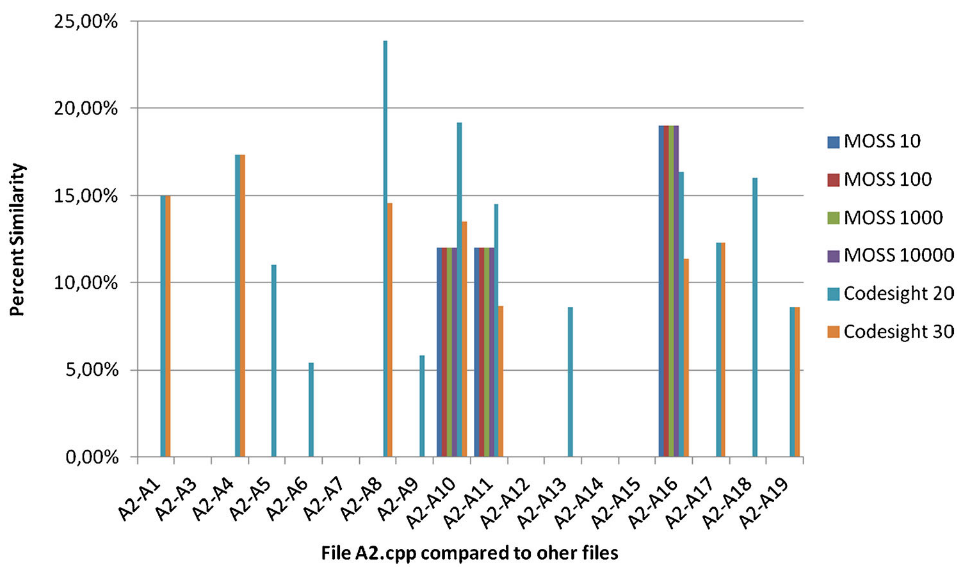
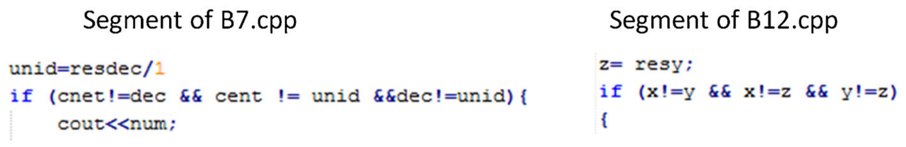


Figure 9 CODESIGHT versus MOSS, test for file A2. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]


Segment of B7.cpp

```
unid=resdec/1
if (cnet!=dec && cent != unid &&dec!=unid){
    cout<<num;
```

Segment of B12.cpp

```
z= resy;
if (x!=y && x!=z && y!=z)
{
```

Figure 10 Similar code segments between files B7.cpp and B12.cpp. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

very large, but the similarity was focused on the restrictions of a condition. Reducing the threshold to 10 tokens, we found that the percentages increase but do not reveal copying. Among other commands, the similar segments represented similar calls to routines and statements, which are inherent to the language syntax.

When comparing CODESIGHT and MOSS for these cases, MOSS finds similarities in data read/write instructions that CODESIGHT dismiss due that CODESIGHT is more rigorous with the contents of the text strings; when CODESIGHT does not find similarities between the strings, the order of the data read/write instructions is not detected as similar. These results are shown in Table 4 and Figure 11.

Results of Test 4

The first analysis with a threshold of 30 tokens indicated that there was no similarity between the files. This result was

Table 4 CODESIGHT Versus MOSS, Test for File B4

	MOSS 10 (%)	MOSS 100 (%)	MOSS 1000 (%)	MOSS 10000 (%)	Codesight 20 (%)	Codesight 30 (%)
B4–B1	0.00	0.00	0.00	0.00	16.92	0.00
B4–B2	0.00	0.00	0.00	0.00	0.00	0.00
B4–B3	39.00	39.00	39.00	39.00	13.72	0.00
B4–B5	0.00	0.00	0.00	0.00	0.00	0.00
B4–B6	0.00	0.00	0.00	0.00	0.00	0.00
B4–B7	0.00	0.00	0.00	0.00	16.67	0.00
B4–B8	0.00	0.00	0.00	0.00	21.74	21.74
B4–B9	0.00	0.00	0.00	0.00	13.79	0.00
B4–B10	0.00	0.00	0.00	0.00	39.53	39.53
B4–B11	0.00	0.00	0.00	0.00	6.78	0.00
B4–B12	0.00	0.00	0.00	0.00	0.00	0.00
B4–B13	0.00	0.00	0.00	0.00	14.65	0.00
B4–B14	39.00	39.00	39.00	39.00	0.00	0.00

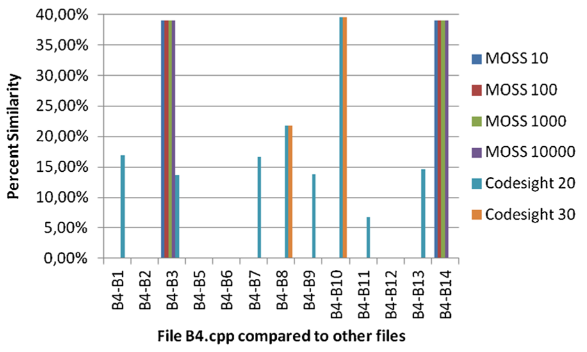


Figure 11 CODESIGHT versus MOSS, test for file B4. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

expected because the two programs being compared were coded for different purposes. A second test with a threshold of 20 also showed no traces of similitude. The third test with a threshold of 10 showed similarities that are inherent to the programming language (declarations of variables and the output of information displayed on the screen) (Table 5).

CONCLUSIONS

Based on the four tests, it was possible to verify the four points that were initially raised. CODESIGHT is capable of identifying those cases where the code has been modified to intentionally mislead an agent who checks for code similarity. According to the tests, the tool successfully detected those cases that were independently reported as plagiarized. Similarly, the tests helped identify those cases where there appears to be copying but that actually represent similarities inherent to the programming language.

However, the percentage of similitude detected by the tool is not a determining factor in the presence of plagiarism because, as we saw in test 4, it can be influenced by the language syntax and the size of the selected threshold. Thus, cases of apparent copying can only be confirmed by the teacher, who knows the true scope of the answers and the skills of the students.

For the instructors, this tool is important because it facilitates the process of reviewing the program files sent by the students. Comparing programming codes is required for detecting plagiarism; the tool generates a similarity score that helps the instructor to make decisions about plagiarism. For the students, this tool facilitates the educational process due that it allows to detect and correct programming malpractices.

The tool still needs to be strengthened through the inclusion of comparison rules that allow for greater efficiency at the time of verification. One of these rules is the conditional validation of a single line with respect to regular conditionals.

Table 5 Results of Test 4 Analysis

$U = 10$	file1.cpp	GCD.cpp
file1.cpp	—	17.75%
GCD.cpp	17.75%	—

Also, it is necessary to find a mechanism to represent, in the system of tokens generated by the String Tokenizing method, those control structures that are similar regardless of their nature (a generic arrangement for the structures "if," "else," and "switch" and one for "while," "do," and "for").

REFERENCES

[1] A. Parker and J. O. Hamblen, Computer algorithms for plagiarism detection, IEEE Trans Educ 32 (1989), 94.

[2] C. Liu, C. Chen, J. Han, and P. S. Yu, Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM 872 (2006).

[3] L. Prechelt, G. Malpohl, and M. Philippsen, Finding plagiarisms among a set of programs with JPlag, J UCS 8 (2002), 1016.

[4] K. W. Bowyer and L. O. Hall, 29th Annual Frontiers in Education Conference, 1999. FIE'99. IEEE 13B3/18 (1999).

[5] The Sherlock Plagiarism Detector. The University of Sydney. [Online]. Available: http://sydney.edu.au/engineering/it/∼scilect/sherlock/ [Last accessed 03 Dec 2012].

[6] M. J. Wise, ACM SIGCSE Bulletin. ACM 130 (1996).

[7] Y. Yuan and Y. Guo, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM 286 (2012).

[8] F. Zhang, Y. C. Jhi, D. Wu, P. Liu, and S. Zhu, Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM 111 (2012).

[9] J. Y. H. Poon, K. Sugiyama, Y. F. Tan, and M. Y. Kan, Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ACM 122 (2012).

[10] U. Inoue and S. Wada, 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), IEEE 2308 (2012).

[11] E. Flores, A. Barrón-Cedeno, P. Rosso, and L. Moreno, NAACL-HLT 2012, 1 (2012).

[12] G. Cosma and M. Joy, An approach to source-code plagiarism detection and investigation using latent semantic analysis, IEEE Trans Comput 61 (2012), 379.

[13] S. Narayanan and S. Simi, 2012 7th International Conference on Computer Science & Education (ICCSE), IEEE 1065 (2012).

[14] V. Juricic, Proceedings of the ITI 2011 33rd International Conference on Information Technology Interfaces (ITI), IEEE 597 (2011).

[15] Y. C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, Proceedings of the 33rd International Conference on Software Engineering, ACM 756 (2011).

[16] J. S. Lim, J. H. Ji, H. G. Cho, and G. Woo, Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication, ACM 24 (2011).

[17] D. Spinellis, P. Zaharias, and A. Vrechopoulos, Coping with plagiarism and grading load: Randomized programming assignments and reflective grading, Comput Appl Eng Educ 15 (2007), 113.

[18] J. Ramos, M. A. Trenas, E. Gutiérrez, and S. Romero, E-assessment of Matlab assignments in Moodle: Application to an introductory programming course for engineers, Comput Appl Eng Educ (2011), Published online on Wiley Online Library; DOI: 10.1002/cae.20520

[19] S. Rodríguez, J. L. Pedraza, A. G. Dopico, F. Rosales, and R. Méndez, Computer-based management environment for an assembly language programming laboratory, Comput Appl Eng Educ 15 (2007), 41.

[20] S. Mann and Z. Frew, Proceedings of the 8th Australasian Conference on Computing Education—Volume 52, Australian Computer Society, Inc. 143 (2006).

[21] E. L. Jones, Metrics based plagarism monitoring, J Comput Sci Coll 16 (2001), 253.

[22] C. Arwin and S. M. M. Tahaghoghi, Proceedings of the 29th Australasian Computer Science Conference—Volume 48, Australian Computer Society, Inc. 277 (2006).

[23] M. Joy and M. Luck, Plagiarism in programming assignments, IEEE Trans Educ 42 (1999), 129.

[24] K. E. M. Sabri and J. J. Al-Ja'afer, The JK system to detect plagiarism, J Comput Sci Technol 6 (2006), 66–72.

[25] J. F. Sowa, Conceptual structures: Current research and practice, Ellis Horwood, New York, London, Toronto, 1992, p 3.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren, The program dependence graph and its use in optimization, ACM Trans Programming Lang Syst (TOPLAS) 9 (1987), 319.

[27] J. Krinke, Identifying similar code with program dependence graphs, Proceedings of the Eighth Working Conference on Reverse Engineering, IEEE 301 (2001).

[28] S. Engels, V. Lakshmanan, and M. Craig, Plagiarism detection using feature-based neural networks, ACM SIGCSE Bulletin, ACM 34 (2007).

[29] J. H. Ji, G. Woo, and H. G. Cho, A source code linearization technique for detecting plagiarized programs, ACM SIGCSE Bulletin 39 (2007), 73.

[30] M. J. Wise, Running karp-rabin matching and greedy string tiling. Basser Department of Computer Science, University of Sydney, 1993.

BIOGRAPHIES

Andres M. Bejarano received a BS degree in Systems Engineering from the Universidad del Norte (Barranquilla, Colombia) in 2009, and a MS degree in Systems Engineering from the Universidad del Norte (Barranquilla, Colombia) in 2012. Since 2010 he has been a Professor in the Department of Systems Engineering (Departamento de Ingeniería de Sistemas) at the Universidad del Norte.

Lucy E. García received a BS degree in Systems Engineering from the Universidad del Norte (Barranquilla, Colombia) in 1987, a MS degree in Engineering Science from the Pontificia Universidad Católica de Chile (Chile) in 2005, and a Ph.D. degree in Engineering Science from the Pontificia Universidad Católica de Chile (Chile) in 2007. Since 2007 she has been a Professor in the Department of Systems Engineering (Departamento de Ingeniería de Sistemas) at the Universidad del Norte.

Eduardo E. Zurek received a BS degree in Systems Engineering from the Universidad del Norte (Barranquilla, Colombia) in 1994, a MS degree in Electrical Engineering from the University of South Florida in 2002, and a Ph.D. degree in Electrical Engineering from the University of South Florida in 2006. Since 1994 he has been a Professor in the Department of Systems Engineering (Departamento de Ingeniería de Sistemas) at the Universidad del Norte.