An Acad Bras Cienc (2023) 95(3): e20220684 DOI 10.1590/0001-3765202320220684
Anais da Academia Brasileira de Ciências | *Annals of the Brazilian Academy of Sciences*
Printed ISSN 0001-3765 I Online ISSN 1678-2690
www.scielo.br/aabc | www.fb.com/aabcjournal

ENGINEERING SCIENCES

BIOPLAG: An Approach to Detect Programming Plagiarism

KAIO P. GOMES, SIMONE N. MATOS & TARCIZIO ALEXANDRE BINI

Abstract: This paper creates an approach to the automatic detection of plagiarism in programming by combining the interdisciplinary knowledge from bioinformatics with techniques such as: tokens of programming language elements, tokens mapping in synthetic biological sequence, and alignment of biological sequences. This solution, named BIOPLAG, contemplates different levels of modifications in source code, and its functioning depends on concepts of computer science as well. Through the realization of three experiments with students and programmers, a total of 168 real examples of plagiarism evaluated the implementation of BIOPLAG with the usage of 336 source codes in C language divided into seven scenarios. The evaluation process compared to other tools known as references in the literature: MOSS and JPLAG. The evaluative metrics used are precision, recall, and fmeasure. As a result, BIOPLAG showed to be the best in four and equal in three out of seven test scenarios. Its average score on the metrics in the evaluation process was 0.95 against MOSS with 0.83 and JPAG with 0.87.

Key words: Automatic detection, bioinformatics, plagiarism in programming, source code.

INTRODUCTION

The plagiarism in programming, specifically in source code, can be defined as unauthorized re-use of a program's implementation by copying its structure and syntax (Burrows et al. 2007). Unlike authorized use, which allows the re-use of source code in software development (Karnalim et al. 2022). For instance, it might be desirable to re-use test implementations of open-source projects to contribute to greater security and agility in the development of software (Makady & Walker 2017).

The concern of this subject has been noticed by the statistics (Flores et al. 2015). A study developed by Sraka & Kaucic (2009) and Cheers et al. (2021, 2023) showed that in the academic environment, a rate of 72.5% of students admitted to plagiarizing at least once in programming assignments. This problem might affect the learning process of students in programming courses (Gomes & Matos 2020). However, this type of plagiarism can interfere in various areas besides the education field (Chuda et al. 2012, Cheers & Lin 2022). According to Ullah et al. (2020), every software system has up to 20% of plagiarized source code in its implementation.

Proposed solutions for the plagiarism problem are created to automate the detection of similarity between source codes, having its usage for the detection of both authorized and unauthorized re-use (Flores et al. 2015, Ragkhitwetsagul 2016, Makady & Walker 2017, Cheers & Lin 2022). Manually performing this task requires much effort and may become unfeasible in certain cases (Prado et al. 2018). The automatic detectors for unauthorized use of source code try to contemplate different modification techniques used by plagiarists. These modifications are addressed by studies in computer science, such as the six levels of Faidhi & Robinson (1987) and the four types of Roy & Cordy (2007).

The automatic detectors created are commonly based on various approaches, for example, graphs, trees, tokens, metrics, textual comparisons, among others (Arwin & Tahaghoghi 2006). Some of the tools developed using these existing techniques are JPLAG, Measure of Software Similarity (MOSS), Yes, Plague, Yet Another Plague (YAP), Plaggie, Fast Plagiarism-Detection System (FPDS) and Marble (Đurić & Gašević 2013).

Many solution tools have been proposed and developed; the two most used as references in the literature are MOSS and JPLAG (Ahadi & Mathieson 2019, Đurić & Gašević 2013). However, these tools, including the most used ones, have limitations regarding the ability to detect various levels or types of modifications in source codes. Besides, these solutions have a limited number of supported programming languages. As a limiting factor, the time complexity has to be regarded to avoid infeasible solutions.

The present paper creates an approach, named BIOPLAG, to automatically detect the six levels of plagiarism in source code following the classification elaborated by Faidhi & Robinson (1987). The BIOPLAG (Gomes 2020) uses a combination of techniques from bioinformatics and computer science. The following techniques are used: tokens of programming language elements, tokens mapping in synthetic biological sequences, and alignment of biological sequences. The applied concepts of bioinformatics are based on the computer virus detection method developed by Pedersen et al. (2012).

BIOPLAG has the characteristic of supporting any programming language. In this work, its implementation used a token generator for the C language to evaluate the created approach. However, other token generators of any programming language can be integrated.

The evaluation process is performed from the application of BIOPLAG in seven test scenarios, of which six aim to evaluate its performance in detecting plagiarism levels in plagiarized source code, and one is directed to find false positives and false negatives. The tests were created from implementations of source codes written in C language representing samples of presence and absence of plagiarism, and they were implemented through three real experiments with volunteers.

Experiments carried out in this paper had the participation of 25 undergraduate students and three graduate students from the Federal University of Technology – Paraná, in addition to six programmers from a software development company in the region. From a public source code dataset, the volunteers produced examples of programming plagiarism following the six specific levels of Faidhi & Robinson (1987) to compose six testing scenarios.

The test scenarios used a total of 336 source code examples implemented in C language to be tested in pairs, resulting in 168 tests to evaluate the detection of six levels of plagiarism in programming and cases of absence of plagiarism. For the cases of absence, 60 non-plagiarized examples were chosen randomly from the source code dataset used in the experiments.

The evaluation of the test scenarios was performed to analyze the following performance metrics: precision, recall, and fmeasure. The MOSS and JPAG tools, considered as references in state of the art, were compared with BIOPLAG regarding their performances in programming plagiarism detection.

The results obtained by BIOPLAG demonstrated that its performance was not inferior in any test scenario for the precision, recall, and fmeasure regarding the compared tools. In sum, it has shown to be greater or equal for every test scenario.

BACKGROUND

The Cambridge dictionary (Cambridge 2020) defines plagiarism as ”the process or practice of using another person’s idea or work and pretending that it is your own.” Regarding the concern of plagiarism in programming, as shown in Ullah et al. (2020), each software system contains from 5% to 20% of plagiarism in its source code.

Due to this ethical problem that affects several fields (Chuda et al. 2012, Cheers & Lin 2022) besides the software industry, plagiarism detection approaches are proposed. The computer science contributes for the elaboration of these solutions (Allyson et al. 2019, Kuo et al. 2018, Meuschke et al. 2018, Prechelt et al. 2002, Cheers & Lin 2022) as well as other interdisciplinary fields, for example, the bioinformatics (Kane & Springer 2007, Pedersen et al. 2012, Revett 2009, Xu et al. 2006).

Bioinformatics

Bioinformatics techniques can solve computer science problems, as shown in the following studies: ”integrating bioinformatics, distributed data management, and distributed computing for applied training in high-performance computing” (Kane & Springer 2007); ”Blast your way through malware analysis assisted by bioinformatics tools” (Pedersen et al. 2012); ”An approach to SOA-based bioinformatics grid” (Xu et al. 2006); ”a bioinformatics based approach to user authentication via keystroke dynamics” (Revett 2009); ”Planogram Compliance Control via Object Detection, Sequence Alignment, and Focused Iterative Search” (Yücel & Ünsalan 2022); among others.

The biological sequences which are used in the alignments can be of different types, for example, DNA and protein (Goad & Kanehisa 1982). According to the chosen bioinformatics tool for executing alignments, any type of sequence can be used (Tang et al. 2009). For bioinformatics, the use of these tools provides support in the investigation of protein functions, evolutionary relationships, among other applications (Junior & Wolmer 2019).

Deoxyribonucleic acid (DNA) is responsible for keeping the genetic information of living organisms. In its composition, there are four different types of nucleotides represented by the following nitrogenous bases: adenine (A), thymine (T), guanine (G), and cytosine (C). The formulation of a DNA sequence is done through a polynucleotide chain (Alberts et al. 2017).

Protein is associated with the metabolism of organisms performing different functions. In its structure, it is defined as a polymer formed by a linear sequence of amino acids. Amino acids are the molecules that make up the protein through 20 different shapes: alanine (A), arginine (R), asparagine (N), aspartic acid (D), cysteine (C), glutamine (Q), glutamic acid (E), glycine (G), histidine (H), isoleucine (I), leucine (L), lysine (K), methionine (M), phenylalanine (F), proline (P), serine (S), threonine (T), tryptophan (W), tyrosine (Y), and valine (V) (Hunter 2009).

The alignment of these biological sequences consists of comparing two or more sequences in search of a series of characters or patterns of characters that are in the same order in the comparison (Mount 2001). This technique aims to detect the level of similarity between sequences. Its functioning belongs to the same class of problems of finding the largest common subsequence problem (LCS) (Coull et al. 2003).

The two main types of sequence alignment are: global and local. The global occurs when the character-to-character comparison is made considering the entire length of the strings, seeking to obtain the largest possible number of comparisons. Unlike the global, the local type seeks only regions with a higher degree of similarity or equality of characters (Haque et al. 2009).

A scoring system is used to evaluate the alignment based on three comparison criteria, which can be: match, mismatch, and gap. A positive value is assigned when characters are equal, and a negative value is assigned in opposite situations when there is no equality. For the gap cases, it assigns a penalty through a negative score when there are absence and presence of characters in comparison (Coull et al. 2003).

All these concepts of sequence alignment are applied in a biologically inspired computing method developed by Pedersen et al. (2012) that solves a computer science problem for the information security domain. The main goal of the proposed solution is to detect computer malware using the methods and tools of bioinformatics. Its functioning depends on four main steps.

The step I is responsible for selecting the files of interest, which are called digital artifacts. In this context of the application, any file suspected of being infected and a known file containing a specific malware of analysis interest. The chosen digital artifacts are inputs for the next step.

In step II, it occurs the mapping of the digital artifacts into DNA sequences. These sequences represent the files, and no longer, the genetic material of organisms. The creation of these biological structures depends on the ASCII (Extended version) binary representation of each character that belongs to the digital artifact's content.

Through a mapping table, every two contiguous bits a nucleotide is generated for creating the DNA sequence of the digital artifact. The method used the following bits mapping into nucleotide: 00 into T, 01 into G, 10 into C, and 11 into A. The order of the nucleotides in the sequence respects the same order that the bits appear in the binary representation.

The DNA sequences generated in step II are inputs for step III. In this step, a local alignment conducted by a bioinformatics tool is performed. The tool adopted by this task is the Basic Local Alignment Search Tool (BLAST), which detects regions of similarity between DNA sequences (Altschul et al. 1990). This tool creates a report with different parameters analyzing the alignment, such as e-value, percent identity, and score (Nilsson et al. 2012).

After performing the alignment, the report is analyzed in step IV. The main objective is to identify the level of similarity among the sequences. By considering the degree of similarity established, it is possible to identify whether there is malware in the suspect file. Due to the nature of the local alignment, the report identifies regions of high similarity, which helps in the identification of files camouflaged with the virus. Intentionally, the virus hides in a small portion of the file, and the rest keeps the same as original not infected. In sum, this bioinformatics proposed method solves a computer science problem and open opportunity to create novel solutions for other computing domains besides information security, for example, programming plagiarism detection as pointed by Pedersen et al. (2012).

Programming plagiarism

Plagiarism consists of re-using the work of other authors without referencing them (Maurer et al. 2006). An example is programming plagiarism, in which source codes are modified or camouflaged with partial or integral pieces of other authors' implementations (Flores et al. 2015).

Plagiarists in programming make different types of modifications to the source code so that plagiarism detectors cannot identify that the implementation was generated from the partial or even full copy of other code snippets. There are studies in computer science that seek to categorize these modifications; for example, one of the leading studies for classification approach is the six levels of Faidhi & Robinson (1987).

According to the six levels of modifications, level 1 represents source code changes in terms of comments and indentations. Level 2 changes the identifiers of source code components, for example, the name of variables. Level 3 contemplates changes in declared components, for instance, by altering the position of declared variables, constants, procedures, and functions. Level 4 are modifications in module components of source codes such as functions and procedures; for example, the creation of new functions by merging two or more existing ones.

The level 5 replaces program repetition statements such as "FOR" and "WHILE". Level 6 applies logic changes by altering control statements such as the expressions in conditions. In sum, the levels are cumulative and present increasing complexity for programming plagiarism detectors, in which the sixth is the most difficult and the first is the easiest.

The two main detectors of plagiarism in source code evaluated as references in the literature are JPLAG and MOSS. Both are widely used for comparison studies since they are free and functional (Đurić & Gašević 2013, Ahadi & Mathieson 2019).

JPLAG is an automatic detection tool developed by Prechelt et al. (2002) to find similarities among source codes. This tool supports the following languages of source code implementation: Java, Scheme, C, C++ and C# (Ahadi & Mathieson 2019).

The functioning of JPLAG depends on two main phases. Phase 1 applies the token technique, in which each element of the source code turns into a sequence of tokens. Phase 2 performs a peer-to-peer comparison among these sequences by using its version of the Running Karp-Rabin Greedy String Tiling (RKR-GST) algorithm (Noh 2003). Regarding these processes, the time complexity in the worst case is $O(n^3)$.

The Measure of Software Similarity (MOSS) is another tool for automating the detection of plagiarism in source codes. According to Araujo & Kyrilov (2020), its usage has been directed in education to assess the authenticity of programming activities. The supported languages for analysis are C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, and HCL2 (MOSS 2020).

Similar to JPLAG in terms of functioning and time complexity, it depends on the token technique and has the complexity of $O(n^3)$ for worst cases. The source codes to be compared are converted into tokens and analyzed with the Winnowing algorithm developed by Schleimer et al. (2003). This algorithm creates fingerprints for representing each previously generated token. The calculation of the distance between these signature structures indicates the level of similarity for the compared source codes.

THE CREATED SOLUTION

A system that detects programming plagiarism deals with different levels of modifications in source code. Besides, its performance cannot demand a high computational cost of time complexity. Otherwise, the solution is not viable. Regarding other potential limitations, a problem could be the lack of language support for plagiarism analysis.

This present paper creates a novel approach named BIOPLAG for solving the source code plagiarism problem. The goal of this solution is to contemplate all the six levels of modifications in plagiarized source codes following the classification of Faidhi & Robinson (1987).

One of the main features presented in BIOPLAG is the flexibility to extend any language of support. Its functioning depends on the combination of bioinformatics and computer science techniques. From a viability aspect, its time complexity is the same as other recognized tools known as references, for instance, JPLAG and MOSS.

In sum, the following combined techniques elaborate the approach: generation of tokens, mapping in biological sequences, and local alignment of biological sequences. The application of these bioinformatics concepts in the created approach is based on the work of Pedersen et al. (2012). Figure 1 presents an overview of how BIOPLAG works through an activity diagram.

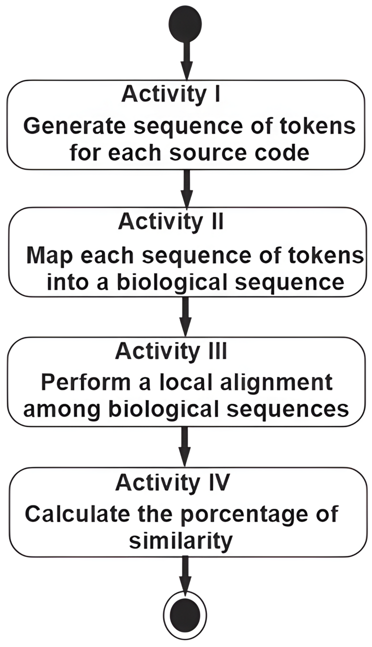

Activity I
Generate sequence of tokens for each source code
Activity II
Map each sequence of tokens into a biological sequence
Activity III
Perform a local alignment among biological sequences
Activity IV
Calculate the porcentage of similarity

Figure 1. BIOPLAG activity diagram.

The diagram is composed of four activities: I) Generation of token sequences for each input source code according to its language of implementation; II) Mapping of each token sequence into biological sequences from the number of different tokens and composition units of the biological sequences;

III) Executing a local alignment among the biological sequences by using a bioinformatics tool; IV) Identification of the rate of similarity from the alignment report provided in the previous activity.

In activity I, the token technique identifies every type of component presented in the source code and categorizes it by assigning a specific token representing its purpose. For example, when there is a declared variable instruction, it assigns a token for the identified variable component.

According to the language of source code implementation, a proper generator of tokens is responsible for the analysis of every instruction. For this reason, BIOPLAG has the flexibility to accept any programming language since it can integrate any generator.

The process of the first activity consists of source codes as inputs, and the outputs are specific token sequences representing each input. Through a hash table, each instruction present in the inputs produces tokens that form the output sequences. The number of different tokens depends on the lexical analyzer used by the chosen generator, as well as the types of instructions in the source codes.

Figure 2 shows an example of functioning for the token generation process. In this case, an input source code in C language turns into a token sequence. The single line of code composed on variable declaration results into three tokens: <TYP> for the data type, <VAR> for the identifier, and <GRA> for the syntax of end of instruction.

The token sequences are the inputs for the activity II, which is responsible for mapping tokens into biological sequences. There are two different ways of performing this mapping, depending on the chosen token generator in the activity I and the units of composition in biological sequence. Firstly, it needs to check the number of different tokens (DT) supported and the number of different biological units (DBU).

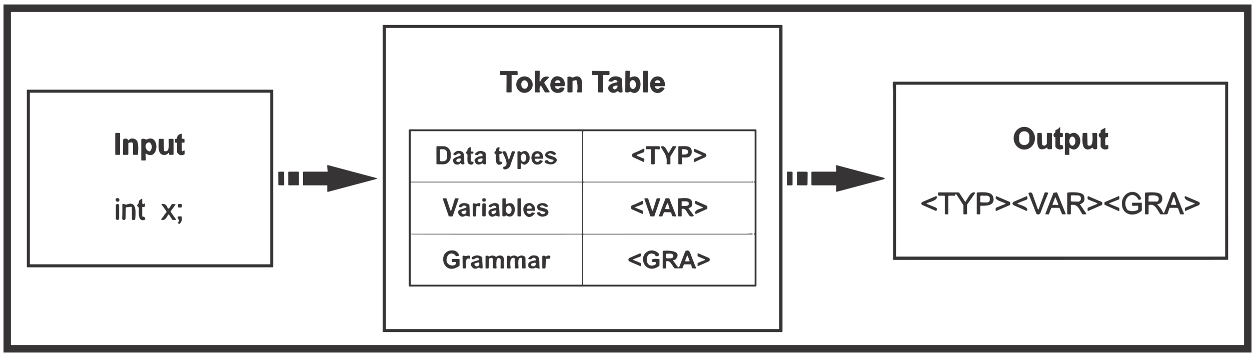
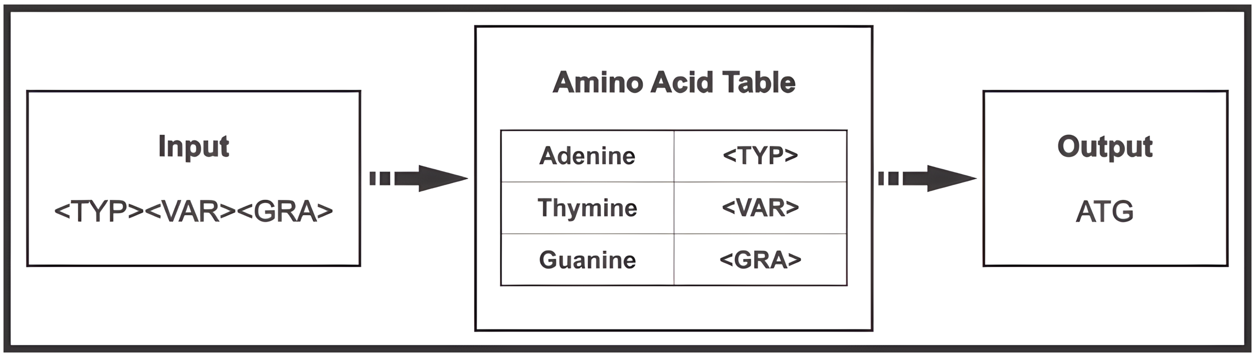


Figure 2. Converting a piece of code into a token sequence.

Formula 1 shows the two possible conditional cases for performing activity II. For the first case, the mapping occurs based on consulting a token table to identify which biological unit will be generated. In Figure 3, it represents this case regarding the chosen biological sequence as DNA, and it has three different tokens. The three inputs convert into three nucleotides.

$$\begin{aligned} DT \leq DBU, forcase: 1 \\ DT > DBU, forcase: 2 \end{aligned} \tag{1}$$

For the second case of performing activity II, the mapping does not occur directly. Firstly, it converts the input into a binary sequence, and then it maps from binary to biological units. The binary representation of the inputs is identified by consulting the ASCII table, and the number of bits that generate one biological unit depends on the type of chosen sequence.



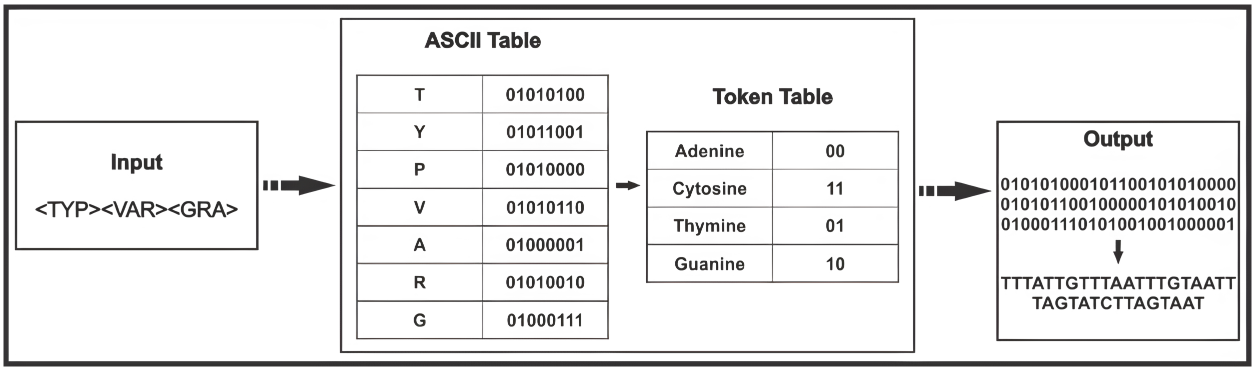
Adenine	<TYP>
Thymine	<VAR>
Guanine	<GRA>

Figure 3. First case of converting a token sequence into a biological sequence.

Formula 2 determines how many contiguous bits are required to convert itself into one unit of the output sequence, where the variable N indicates the number of different units that compose the biological sequence. Once found the result, any binary combination is acceptable since following the number of required bits.

$$\log_2 N \tag{2}$$

Figure 4 illustrates the whole process of performing for the second case in activity II. The input turns into a binary form with the ASCII table, and from this new representation generates the output. The mapping to the new sequence is through the nucleotide table based on bioinformatics. Applying Formula 2 with variable N as 4 since the chosen biological sequence is DNA, it finds that any combination of two bits is enough to complete the mapping.



T	01010100
Y	01011001
P	01010000
V	01010110
A	01000001
R	01010010
G	01000111

Adenine	00
Cytosine	11
Thymine	01
Guanine	10

```
010101000101100101010000
010101100100000101010010
010001110101001001000001
```

```
TTTATTGTTTAATTTGTAATT
TAGTATCTTAGTAAT
```

Figure 4. Second case of converting a token sequence into a biological sequence.

Once the biological sequences are available as output of activity II, they become the inputs for the activity III that execute the alignment among these sequences. BLAST is the bioinformatics tool recommended for performing alignment in this created approach. However, BIOPLAG allows using other solutions for this purpose since it has specific features as supporting biological sequences and local alignments.

Due to the execution of the alignments for each input sequence, the time complexity for this processing is cubic. This technique is the most demanded in terms of the computational cost compared to others in this approach. For this reason, BIOPLAG has time complexity in its worst case of $O(n^3)$ being n the number of source codes for analysis.

The alignment tool provides a report stating the level of similarity based on its result parameters. This report with the complete analysis is the output of the activity III, and at the same time, is the input for the last activity. Figure 5 shows how the alignment process works on BIOPLAG. Note that it is necessary to adjust the tool, in addition to the need to be compatible with the input.

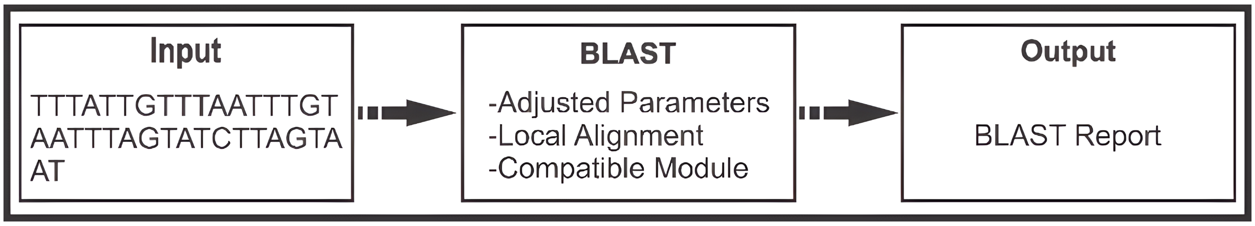


Figure 5. Alignment process by using the BLAST tool.

The activity IV is the last activity, which calculates a rate of similarity ranging from zero to one. As a result of plagiarism detection, BIOPLAG provides this rate in percentage to determine the level of similarity among the input source codes. Figure 6 presents the functioning of this last activity that consists of extracting and processing some of the result metrics founded in the BLAST report. These obtained data compose the elaboration of an own mathematical formula.

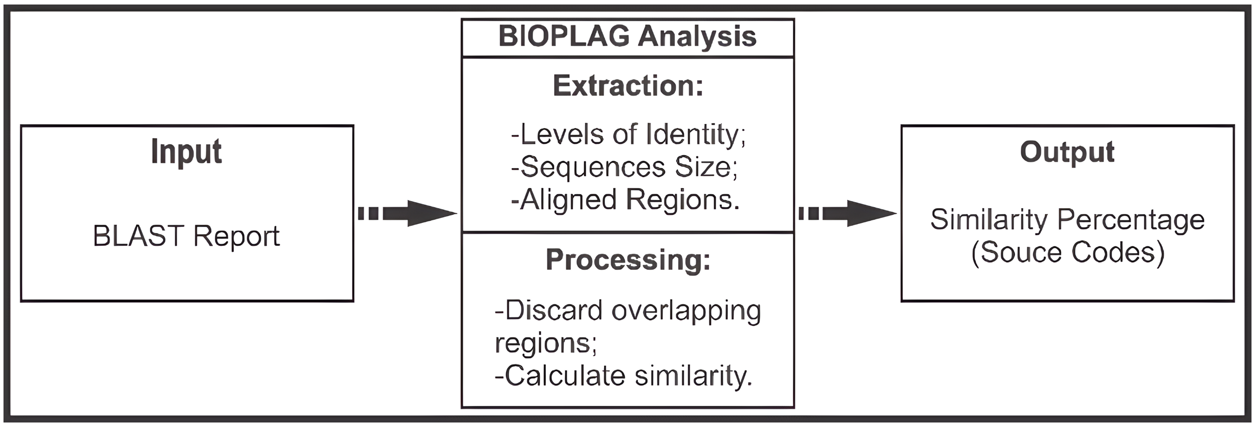


Figure 6. Report analysis process of the BIOPLAG for activity IV.

When BIOPLAG analyses a sequence in the report, it extracts the following corresponded data: size, number of equal units, and positions of each aligned region. Also, it occurs the discarding of overlapping areas in the alignment by keeping only those with high similarity. Finally, it calculates the rate of similarity with the obtained data through Formula 3.

$$sim(Query, Sbjct) = \left(\frac{\sum_{i=1}^{K} n_i}{h}\right), \in [0,1] \tag{3}$$

The variables of the created formula are: n for the number of equal units from the level of identity in a specific region; i for specifying which aligned region, for example, assuming $i = 1$ it refers to the first aligned region; k for the number of different aligned regions; h for the size of the analyzed sequence; is the rate of similarity between $query$ and $sbjct$; $query$ refers to the analyzed sequence, and $sbjct$ is the other compared sequence considering a pairwise comparison.

IMPLEMENTATION

According to the functioning of the BIPLAG to implement the approach is necessary to deal with bioinformatics and computer science techniques such as token generation, biological sequence mapping, and sequence alignment. For this paper, the technology used to develop the solution was Perl for the activities I, II, III, and IV. As for the evaluation process, a shell script was responsible for automating the tests.

For the first activity, the token generator adopted was C Tokenize version 0.18, a Perl module available from the Comprehensive Perl Archive Network (CPAN) (CPAN 2020). This module is responsible for transforming source code into a sequence of tokens, and its usage is only for the C programming language (MetaCPAN 2020). Therefore, in this implementation, the supporting language used is C language.

The chosen token generator can produce up to nine different tokens, as shown in Table I. Each coding command in source code can generate one or more of these symbols, which are the input for the biological sequence mapping technique.

Table I. Description of the tokens.

Token Name	Source Code Component
operator	Different types of operators
grammar	Coding syntax elements
number	Numbers regardless of the type
reserved	Reserved words
word	Identifiers
string	Chain of characters
comment	Single and compound comments
cpp	Inclusions and definitions
char_const	Character constants

In activity II, the chosen type of biological sequence was protein. The hash table used as consulting for the mapping of tokens into amino acids is presented in Table II. It is noteworthy that only 9 out of 20 amino acids available are enough for the mapping process, and any of them can be chosen. However, there are 8 possible mappings instead of 9 since the token for comments is irrelevant for plagiarism detection in programming.

Table II. Mapping table from token to amino acid.

Amino Acid Name	Amino Acid Abbreviation	Token Name
Aspartic acid	D	operator
Cysteine	C	grammar
Glutamine	Q	number
Glutamic acid	E	reserved
Glycine	G	word
Histidine	H	string
N/A	N/A	comment
Arginine	R	cpp
Asparagine	N	char_const

Regarding the chosen biological sequence as protein, it was necessary to use a compatible BLAST module for this type of sequence, by which the choice was Blastp version 2.2.28+ developed by BLAST (2020). In sum, the activity III executed local alignments using this bioinformatics tool with 12 adjusted parameters, considering as inputs: amino acid sequences in FASTA formatting (Pearson & Lipman 1988). Table III shows the listing of the modified settings as well as the new values set for them.

Table III. The values assigned to each parameter in BLAST.

Blastp Parameter	Value
word_size	3
matrix	BLOSUM90, BLOSUM80, BLOSUM62, PAM70 and PAM30
threshold	1
comp_based_stats	0
seg	No
soft_masking	False
db_soft_mask	None
db_hard_mask	None
xdrop_gap_final	25
evalue	1e47
ungapped	N/A
Window_size	40

For the matrix in this new setting, there is more than one value to improve alignment results according to the size of each sequence. Figure 7 presents through a flowchart the mechanism of varying for the parameter values. The remaining parameters not listed kept default values.

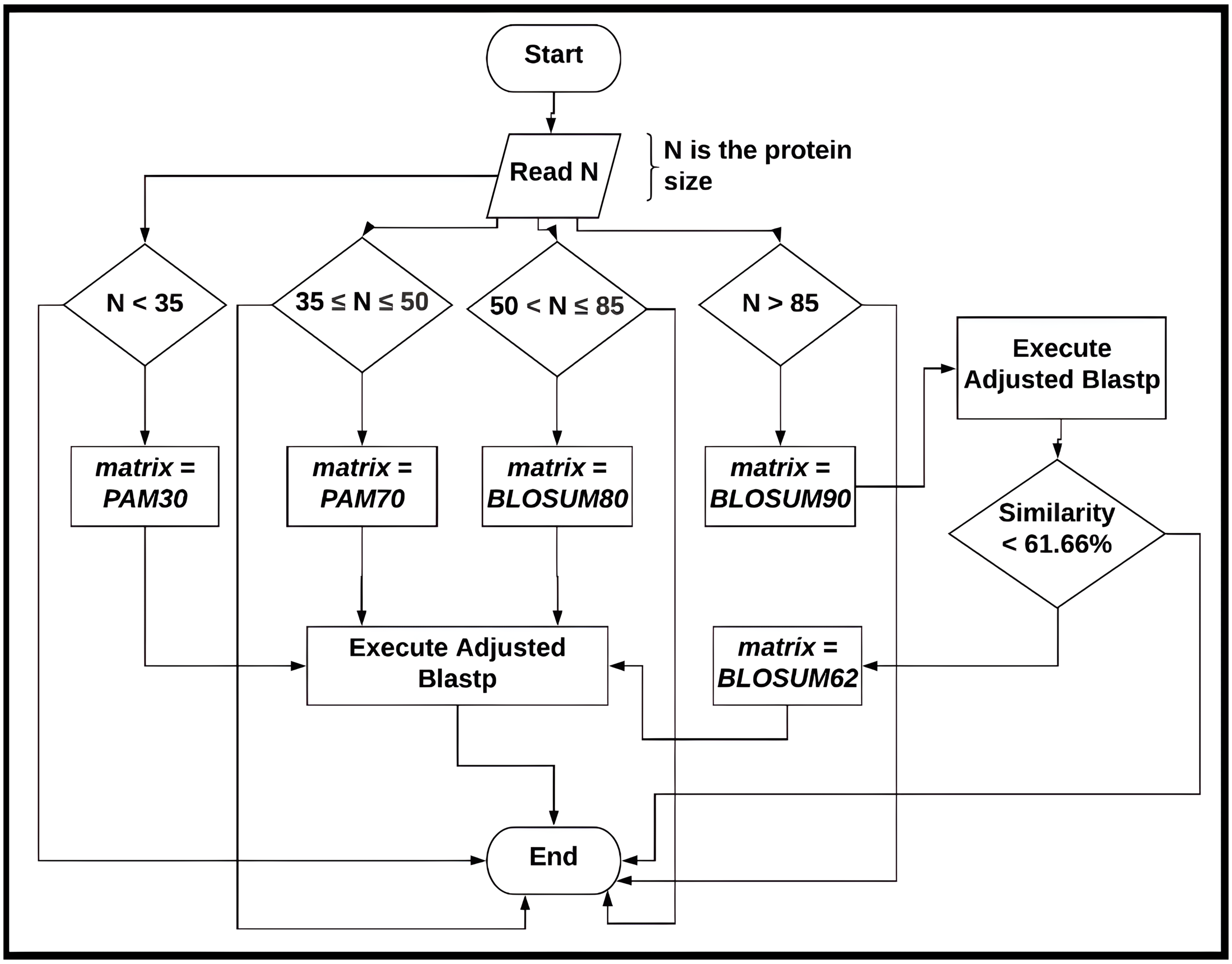


Figure 7. Mechanism of assigning different values for the matrix parameter.

The created mechanism of assigning different values has four cases according to a range of sequence size. One of these cases allows BLAST to use up to two different values instead of just one. This possible second value is used based on a threshold of 61.66% that indicates the presence of plagiarism in source code.

This threshold was calculated by the average rate of six proposed studies in Ajmal et al. (2013), Cosma & Joy (2012), Mason et al. (2019), Pawelczak (2013), Sulistiani & Karnalim (2019) and Xiong et al. (2009). The reason is to avoid false positives e false negatives for alignments with long sequences.

For extracting and processing the BLAST report in the last activity, it was performed a sorting of the aligned regions by crescent positions. The sorting is to deal with the overlaps since it was kept only ranges closer to the beginning of the sequences. For example, by deciding between areas: A with initial position 2 and B with initial position 3, area A is chosen due to localization be nearest to beginning represented through the first unit of composition.

EXPERIMENTS AND EVALUATION

In order to evaluate BIOPLAG's plagiarism detection performance using real examples, three experiments conducted the participation of 28 students and six programmers as volunteers. The main objective of these experiments is to expose the solution in real-world situations with different modification techniques in plagiarized source codes. The participants freely created models of plagiarism for each of the six levels of Faidhi & Robinson (1987). To produce these samples, they receive a dataset used in Mou et al. (2016) with 52.000 source codes to be chosen to plagiarize.

Each plagiarized source code produced by the volunteers went through a check to certify the fulfillment of the following requirements: implementation without errors; usage of the specified programming language, which in this case is C; and correct indication of the techniques used to follow the six levels of plagiarism. Table IV presents a quantitative analysis of the submissions that occurred, considering the three experiments carried out.

Table IV. Quantitative analysis of the submissions for the three experiments.

Experiment	Submissions	Disapprovals	Approvals
1*	128	30	98
2*	39	10	29
3*	21	0	21
Total	188	40	148

Through the approved submissions, seven test scenarios were elaborated to evaluate BIOPLAG. From the first to the sixth scenario, each of the six levels of programming plagiarism is assessed with 148 examples produced by the volunteers, respectively. The seventh scenario is composed of 20 non-plagiarized source codes randomly chosen from the dataset. Its goal is to complement the evaluation with potential situations of false positives and false negatives. Table V shows the composition of each test scenario as well as its evaluation goal.

Altogether, 336 source codes were used in 168 tests through seven different scenarios established for evaluating the BIOPLAG. The following performance metrics were applied to each testing: precision (P), recall (R), fmeasure (F), as shown in Formula 4, 5, and 6.

$$Precision = P = \left(\frac{TP}{TP + FP}\right), \in [0, 1] \tag{4}$$

$$Recall = R = \left(\frac{TP}{TP + FN}\right), \in [0, 1] \tag{5}$$

$$fmeasure = F = 2 * \left(\frac{P * R}{P + R}\right), \in [0, 1] \tag{6}$$

These metrics were considered according to other studies in programming plagiarism domain such as in Acampora & Cosma (2015), Cosma & Joy (2012), Đurić & Gašević (2013), Narayanan & Simi

Table V. Composition of the seven test scenarios.

Scenario	Source Codes	Level of Plagiarism	Tests
A	52	I	26
B	48	II	24
C	50	III	25
D	54	IV	27
E	48	V	24
F	44	VI	22
G	40	False Positive/Negative	20
Total	336	N/A	168

(2012), Nichols et al. (2019), Son et al. (2013), Sulistiani & Karnalim (2019) and Xiong et al. (2009). The variables of these formulas are defined as follows: TP is true positive, FP is false positive, FN is the false negative, P is precision, and R is recall. The positive and negative label is associated with the presence and absence of plagiarism, respectively. The precision metric indicates whether the identified plagiarism cases were expected. On the other side, the recall analysis of whether all plagiarism cases were detected. The fmeasure shows an average performance, considering the precision and recall results.

RESULTS

The implementation of BIOPLAG was evaluated in the defined seven test scenarios from the three experiments, and its results were compared with those obtained by the tools: JPLAG version 2.11.8 and MOSS updated version until December 14, 2018. The default settings of each tool were considered for the 168 tests. Table VI shows the obtained results for the performance metrics.

By analyzing the precision metric, it is identified that all tools achieved the maximum score. This performance indicates that all the existing plagiarisms cases were successfully identified. For the last scenario, the precision was proven by assessing the absence of plagiarism in all its tests. Regarding the recall and fmeasure metric, no solution achieved maximum results in all scenarios. On the other hand, the tool that obtained the maximum result in more scenarios is BIOPLAG being in 3 out of 7 for both metrics.

Among the lowest scores obtained for the recall in each tool, BIOPLAG had the highest value: 0.76. MOSS and JPLAG presented the following values: 0.20833 and 0.41667, respectively. It is observed that BIOPLAG reached a 34% higher performance about the second best-evaluated tool in this case.

The fmeasure performance was higher with BIOPLAG, which had the following scores: an average of 0.94719, a maximum of 1, and a minimum of 0.86364. In comparison, JPLAG had an average of 0.84826, a maximum of 1, and a minimum of 0.58824. With MOSS, an average of 0.78584, a maximum of 1, and a minimum of 0.34483. Table VII presents the overall results regarding all the evaluation metrics.

Table VI. Quantitative analysis of the submissions for the three experiments.

Scenario	Detector	TP	FP	FN	Precision	Recall	fmeasure
A	BioPlag	26	0	0	1	1	1
	JPLAG	25	0	1	1	0.961538	0.980392
	MOSS	25	0	1	1	0.961538	0.980392
B	BioPlag	24	0	0	1	1	1
	JPLAG	24	0	0	1	1	1
	MOSS	23	0	1	1	0.958333	0.978723
C	BioPlag	19	0	6	1	0.76	0.863636
	JPLAG	18	0	7	1	0.72	0.837209
	MOSS	19	0	6	1	0.76	0.863636
D	BioPlag	25	0	2	1	0.925926	0.961538
	JPLAG	19	0	8	1	0.703704	0.826087
	MOSS	18	0	9	1	0.666667	0.80
E	BioPlag	21	0	3	1	0.875	0.933333
	JPLAG	10	0	14	1	0.416667	0.588235
	MOSS	5	0	19	1	0.208333	0.344828
F	BioPlag	17	0	5	1	0.772727	0.871795
	JPLAG	12	0	10	1	0.545455	0.705882
	MOSS	8	0	14	1	0.363636	0.533333
G	BioPlag	20	0	0	1	1	1
	JPLAG	20	0	0	1	1	1
	MOSS	20	0	0	1	1	1

In summary, BIOPLAG performed better results in four scenarios (A, D, E, F) and equal in three (B, C, G). The overall performance analysis of the tools regarding the detection of plagiarism in source codes is presented in Figure 8. The formulation of the analysis considers the following quantitative measures: first quartile, second quartile, third quartile, average, upper limit, lower limit, and outlier. The adopted outliers are data points located 1.5 times above or 1.5 times below the size of the each box.

BIOPLAG presented scores of less than or equal to approximately 0.925926 in 25% of its results. Also, it obtained maximum scores in 50% and values between the inclusive range 0.925926 and 1 in 25%. The average was 0.950665 and the median 1, with the upper limit of 1 and the lower limit of 0.863636. It was found the presence of two discrepant results: 0.76 and 0.772727.

It was found that the lowest score of BIOPLAG was considered to be a discrepant point due to its low frequency, which in this case occurred a single time and was distant from the limit: 0.863636. On

Table VII. Overall results for the compared tools.

Detector	Metric	Minimum	Average	Maximum
BIOPLAG	Precision	1	1	1
MOSS		1	1	1
JPLAG		1	1	1
BIOPLAG	Recall	0.76	0.9048	1
MOSS		0.2083	0.7026	1
JPLAG		0.4166	0.7639	1
BIOPLAG	fmeasure	0.8636	0.9471	1
MOSS		0.3448	0.7858	1
JPLAG		0.5882	0.8482	1

the other hand, JPLAG did not present any discrepant points, but its lower limit was the lowest out of all: 0.416667. The MOSS presented three outliers, and its lower limit was 0.533333.

BIOPLAG achieved the highest scores and with the least variability. In 75% of its results, the scores varied between the inclusive interval 0.925926 and 1. JPLAG and MOSS presented the following inclusive ranges: 0.72 and 1, and 0.76 and 1, respectively.

The variation between the results is smaller with BIOPLAG, considering its difference of 0.049335 between the mean and the median. Comparing with JPLAG and MOSS, this value is 0.1229278 and 0.170504, respectively. The difference between these indicators confirms a higher symmetry in the results.

Summarizing the findings on the boxplot chart, BIOPLAG showed better or equal results for each comparative measure analyzed. Regarding all tests performed, the average value for the three metrics was 0.950665 for the created approach against 0.829496 for MOSS and 0.870722 for JPLAG.

CONCLUSIONS

This paper created an approach, named BIOPLAG, capable of detecting plagiarism levels in source codes. Out of all its features, the flexibility of supporting any programming language stands out. Its output result is a value in percentage that indicates the similarity found between the compared source codes. The elaboration of the approach was designed considering interdisciplinary techniques to improve the detection of plagiarism in programming.

The BIOPLAG is based on a set of combined techniques from bioinformatics and computing, and its functioning essentially depends on four steps: generation of tokens, mapping of tokens into biological sequences, alignment of biological sequences, and calculation of similarity among the source codes compared. The central premise is to convert source code in a biological model in order to use bioinformatics resources. Once this model is combined with an efficient computing technique, a better solution can be proposed.

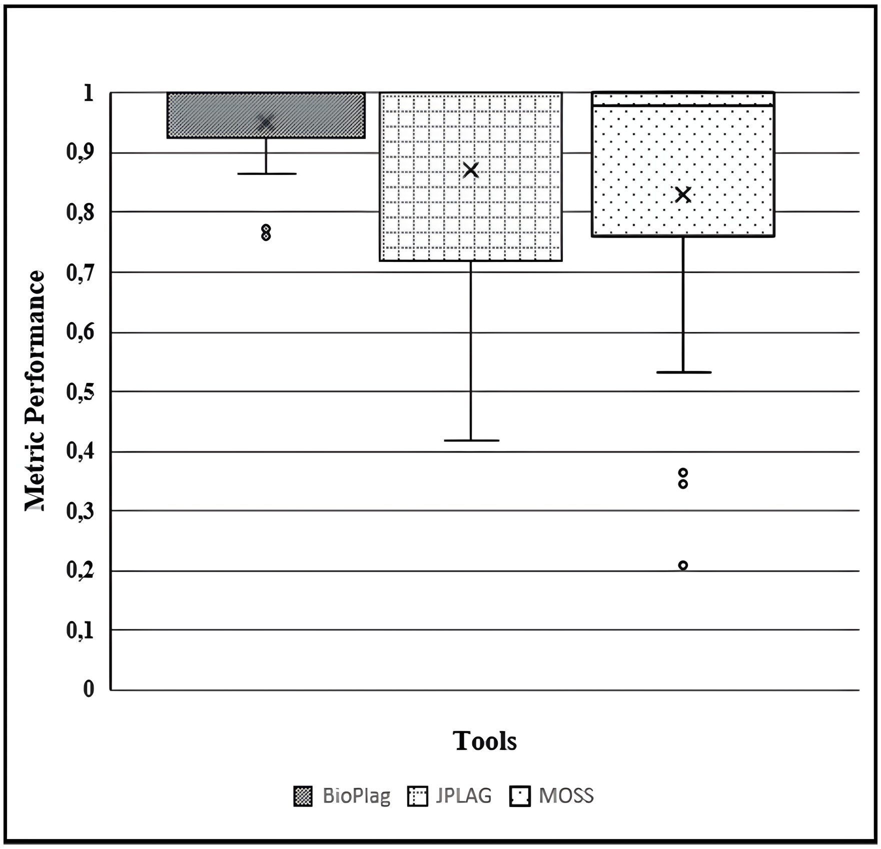


Figure 8. Comparison of the tools through a boxplot chart by analyzing on y-axis the results from 0 to 1 regarding the three performance metrics such as recall, precision and fmeasure.

With the implementation of BIOPLAG, an evaluation process was performed through seven different test scenarios. This evaluation used 296 source codes produced through three experiments having the participation of undergraduate students, graduate students, and programmers from an information technology company. Besides, for the composition of the last test scenario, 40 source codes were chosen randomly from a dataset that served as support for the experiments.

The assessment used a total of 336 source codes implemented in C language and distributed within seven specific test scenarios contemplating the different levels of programming plagiarism. For each test, the following metrics were considered: precision, recall, and fmeasure. The results found with the implementation of the BIOPLAG were compared with other programming plagiarism detectors: MOSS and JPAG.

The results obtained by BIOPLAG indicate that the created approach was able to detect the six levels of plagiarism in source code categorized by Faidhi & Robinson (1987). A total of 152 (90.48%) out of 168 tests were correctly detected with the presence or absence of programming plagiarism. In contrast, the other known tools, such as MOSS and JPLAG presented a total of 118 (70.23%) and 128 (76.19%), respectively.

Comparing the tools, BIOPLAG obtained better performance in 4 test scenarios and equal in 3 out of 7. Its average score on the metrics in the evaluation process was 0.950665 against MOSS with 0.829496 and JPAG with 0.870722. Therefore, the created approach proved that the usage of the combined techniques from bioinformatics and computer science successfully detected plagiarism in programming.

FUTURE WORKS

Regarding the continuity of the development of this work, the following potential subjects can be considered: extend the evaluation process for detection performance, define a new set of metrics to be applied on the tests and implement the BIOPLAG considering other programming languages of support. For instance, conduct new tests aimed at evaluating samples of non-plagiarized source codes. The objective is to incorporate other metrics that deal with false negatives and true negatives, such as the specificity rate. Also, different token generators can be utilized or developed in order to complement the support of new programming languages.

The application of BioPlag for detecting similarities in texts and images is a research to be developed in future works. As shown in the background section of this paper, several computer science domain problems including the object detection on computer vision could be a new topic of study.

Future experiments can be performed with BioPlag to extend the detection of programming plagiarism using source codes from standardized datasets such as ACM International Collegiate Programming Contest (ICPC) and SOurce COde re-use (SOCO) from the international PAN@Fire 2014 competition.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brazil (CAPES) - Finance Code 001. To Dr. Alexandre Rossi Paschoal for review of bioinformatics content the article.

REFERENCES

ACAMPORA G & COSMA G. 2015. A Fuzzy-Based Approach to Programming Language Independent Source-Code Plagiarism Detection. In: 2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE). New Jersey: IEEE Press, p. 1-8. URL https://doi.org/10.1109/FUZZ-IEEE.2015.7337935.

AHADI A & MATHIESON L. 2019. A Comparison of Three Popular Source Code Similarity Tools for Detecting Student Plagiarism. In: Proceedings of the Twenty-First Australasian Computing Education Conference. New York: Association for Computing Machinery, p. 112-117.

AJMAL O, SAAD MISSEN MM, HASHMAT T, MOOSA M & ALI T. 2013. EPlag: A two layer source code plagiarism detection system. In: Eighth International Conference on Digital Information Management (ICDIM 2013). Islamabad: IEEE, p. 256-261. doi:10.1109/ICDIM.2013.6693984.

ALBERTS B, JOHNSON A, LEWIS J, MORGAN D, RAFF M, ROBERTS K, WALTER P, WILSON J & HUNT T. 2017. Biologia Molecular da Celula. Porto Alegre: Artmed Editora. URL https://books.google.com.br/books?id=DlMmDwAAQBAJ.

ALLYSON FB, DANILO ML, JOSÉ SM & GIOVANNI BC. 2019. Sherlock N-overlap: Invasive Normalization and Overlap Coefficient for the Similarity Analysis Between Source Code. IEEE Trans Comput 68(5): 740-751. doi:10.1109/TC.2018.2881449.

ALTSCHUL SF, GISH W, MILLER W, MYERS EW & LIPMAN DJ. 1990. Basic local alignment search tool. J Molec Bio 215(3): 403-410. doi:10.1016/S0022-2836(05)80360-2.

ARAUJO GG & KYRILOV A. 2020. Plagiarism Prevention through Project Based Learning with GitLab. J Comput Sci Coll 35(10): 53-58.

ARWIN C & TAHAGHOGHI SMM. 2006. Plagiarism Detection across Programming Languages. Australia: Australian Computer Society Inc., p. 277-286.

BLAST. 2020. National Library of Medicine. URL https://blast.ncbi.nlm.nih.gov/Blast.cgi.

BURROWS S, TAHAGHOGHI SMM & ZOBEL J. 2007. Efficient Plagiarism Detection for Large Code Repositories. Softw Pract Exper 37(2): 151-175.

CAMBRIDGE. 2020. The Cambridge Dictionary. URL https://dictionary.cambridge.org/us/dictionary/english/plagiarism.

CHEERS H & LIN Y. 2022. Identifying plagiarised programming assignments based on source code similarity scores. Computer Science Education, p. 1-25. doi:10.1080/08993408.2022.2060633.

CHEERS H, LIN Y & SMITH SP. 2021. Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. IEEE Access 9: 50391-50412. doi:10.1109/ACCESS.2021.3069367.

CHEERS H, LIN Y & YAN W. 2023. Identifying Plagiarised Programming Assignments with Detection Tool Consensus. Inform Educ 22(1): 1-19. doi:10.15388/infedu.2023.05.

CHUDA D, NAVRAT P, KOVACOVA B & HUMAY P. 2012. The Issue of (Software) Plagiarism: A Student View. IEEE Trans Educ 55(1): 22-28. doi:10.1109/TE.2011.2112768.

COSMA G & JOY M. 2012. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. IEEE Trans Comput 61(3): 379-394. doi:10.1109/TC.2011.223.

COULL S, BRANCH J, SZYMANSKI B & BREIMER E. 2003. Intrusion detection: a bioinformatics approach. In: 19th Annual Computer Security Applications Conference, 2003. Proceedings, p. 24-33. doi:10.1109/CSAC.2003.1254307.

CPAN. 2020. Comprehensive Perl Archive Network. URL https://www.cpan.org/modules/.

ĐURIĆ Z & GAŠEVIĆ D. 2013. A Source Code Similarity System for Plagiarism Detection. The Computer Journal 56(1): 70-86. doi:10.1093/comjnl/bxs018.

FAIDHI JAW & ROBINSON SK. 1987. An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment. Comput Educ 11(1): 11-19. doi:10.1016/0360-1315(87)90042-X.

FLORES E, BARRÓN-CEDEÑO A, MORENO L & ROSSO P. 2015. Cross-language source code re-use detection using latent semantic analysis. J Univers Comput Sci 21(13): 1708-1725.

GOAD WB & KANEHISA MI. 1982. Pattern recognition in nucleic acid sequences. I.A general method for finding local homologies and symmetries. Nucleic Acids Res 10(1): 247-263. doi:10.1093/nar/10.1.247.

GOMES K. 2020. BIOPLAG: abordagem de detecção de plágio em código-fonte. Master's thesis. Universidade Tecnologica Federal do Paraná, Ponta Grossa-PR.

GOMES K & MATOS S. 2020. Contributions of Bioinformatics for computing education in the detection of programming assignment plagiarism. doi:10.36229/978-85-7042-223-1.CAP.17.

HAQUE W, ARAVIND A & REDDY B. 2009. Pairwise Sequence Alignment Algorithms: A Survey. In: Proceedings of the 2009 Conference on Information Science, Technology and Applications. New York: Association for Computing Machinery, p. 96-103. doi:10.1145/1551950.1551980.

HUNTER LE. 2009. The Processes of Life: An Introduction to Molecular Biology. Cambridge: The MIT Press. doi:10.7551/mitpress/9780262013055.001.0001.

JUNIOR Q & WOLMER D. 2019. ESSEX: identificação de um aminoácido de interesse em sequências biológicas de origens diferentes. Master's thesis. Universidade Federal do Rio Grande.

KANE MD & SPRINGER JA. 2007. Integrating Bioinformatics, Distributed Data Management, and Distributed Computing for Applied Training in High Performance Computing. In: Proceedings of the 8th ACM SIGITE Conference on Information Technology Education. New York: Association for Computing Machinery, p. 33-36. doi:10.1145/1324302.1324311.

KARNALIM O, SIMON, CHIVERS W & PANCA BS. 2022. Educating Students about Programming Plagiarism and Collusion via Formative Feedback. ACM Transac Comput Ed 22(3): 1-31. doi:10.1145/3506717.

KUO JY, CHENG HK & WANG PF. 2018. Program plagiarism detection with dynamic structure. In: 2018 7th International Symposium on Next Generation Electronics (ISNE). Taipei: IEEE, p. 1-3. doi:10.1109/ISNE.2018.8394758.

MAKADY S & WALKER RJ. 2017. Test Code Reuse from OSS: Current and Future Challenges. In: Proceedings of the 3rd Africa and Middle East Conference on Software Engineering. New York: Association for Computing Machinery, p. 31-36. doi:10.1145/3178298.3178305.

MASON T, GAVRILOVSKA A & JOYNER DA. 2019. Collaboration Versus Cheating: Reducing Code Plagiarism in an Online MS Computer Science Program. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. New York: Association for Computing Machinery, p. 1004-1010. doi:10.1145/3287324.3287443.

MAURER H, KAPPE F & ZAKA B. 2006. Plagiarism - A Survey. J Univers Comput Sci 12: 1050-1084.

METACPAN. 2020. C-Tokenize. URL https://metacpan.org/dist/C-Tokenize/view/lib/C/Tokenize.pod\%23VERSION.

MEUSCHKE N, GONDEK C, SEEBACHER D, BREITINGER C, KEIM D & GIPP B. 2018. An Adaptive Image-Based Plagiarism Detection Approach. In: Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries. New York: Association for Computing Machinery, p. 131-140. doi:10.1145/3197026.3197042.

MOSS. 2020. A system for detecting software similarity. URL https://theory.stanford.edu/~aiken/moss/.

MOU L, LI G, ZHANG L, WANG T & JIN Z. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI Press 30(1): 1287-1293.

MOUNT DW. 2001. Bioinformatics - sequence and genome analysis. New York: Cold Spring Harbor Laboratory Press.

NARAYANAN S & SIMI S. 2012. Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In: 2012 7th International Conference on Computer Science and Education (ICCSE), p. 1065-1068. doi:10.1109/ICCSE.2012.6295247.

NICHOLS L, DEWEY K, EMRE M, CHEN S & HARDEKOPF B. 2019. Syntax-Based Improvements to Plagiarism Detectors and Their Evaluations. In: Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education. ITiCSE '19. New York: Association for Computing Machinery, p. 555-561. doi:10.1145/3304221.3319789.

NILSSON RH ET AL. 2012. Five simple guidelines for establishing basic authenticity and reliability of newly generated fungal ITS sequences. MycoKeys 4: 37-63. doi:10.3897/mycokeys.4.3606.

NOH SY. 2003. An XML Plagiarism Detection Model for Procedural Programming Languages. URL https://api.semanticscholar.org/CorpusID:5847809.

PAWELCZAK D. 2013. Online detection of source-code plagiarism in undergraduate programming courses. In: Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS), p. 1-7.

PEARSON W & LIPMAN D. 1988. Improved tools for biological sequence comparison. In: Proceedings of the National Academy of Sciences of the United States of America 85: 2444-2448.

PEDERSEN JG, BASTOLA DR, DICK K, GANDHI R & MAHONEY WR. 2012. BLAST Your Way through Malware Malware Analysis Assisted by Bioinformatics Tools. In: International Conference on Security and Management, p. 1-7. URL https://api.semanticscholar.org/CorpusID:202724024.

PRADO BO, BISPO KA & ANDRADE R. 2018. X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments. In: Proceedings of the 20th International Conference on Enterprise Information Systems, ICEIS. p. 517-524. doi:10.5220/0006668705170524.

PRECHELT L, MALPOHL G & PHILIPPSEN M. 2002. Finding Plagiarisms among a Set of Programs with JPlag. J UCS 8(11): 1016-1038.

RAGKHITWETSAGUL C. 2016. Measuring Code Similarity in Large-Scaled Code Corpora. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), p. 626-630. doi:10.1109/ICSME.2016.18.

REVETT K. 2009. A bioinformatics based approach to user authentication via keystroke dynamics. Int J Cont Automat Sys 7: 7-15.

ROY C & CORDY J. 2007. A Survey on Software Clone Detection Research. School Comput TR 541: 64-68.

SCHLEIMER S, WILKERSON DS & AIKEN A. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. New York: Association for Computing Machinery, p. 76-85. doi:10.1145/872757.872770.

SON JW, NOH TG, SONG HJ & PARK SB. 2013. An Application for Plagiarized Source Code Detection Based on a Parse Tree Kernel. Eng Appl Artif Intell 26(8): 1911-1918. doi:10.1016/j.engappai.2013.06.007.

SRAKA D & KAUCIC B. 2009. Source code plagiarism. In: Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces, p. 461-466. doi:10.1109/ITI.2009.5196127.

SULISTIANI L & KARNALIM O. 2019. ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment. Comput Appl Eng Educ 27: 166-182.

TANG Y, XIAO B & LU X. 2009. Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms. Comput Sec 28: 827-842.

ULLAH F, WANG J, FARHAN M, JABBAR S, WU Z & KHALID S. 2020. Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. Multimed Tools Appl 79: 1-18.

XIONG H, YAN H, LI Z & LI H. 2009. BUAA_AntiPlagiarism: A System To Detect Plagiarism for C Source Code. In: 2009 International Conference on Computational Intelligence and Software Engineering, p. 1-5.

XU G, LUO Y, YU H & XU Z. 2006. An Approach to SOA-Based Bioinformatics Grid. In: 2006 IEEE Asia-Pacific Conference on Services Computing, p. 323-328.

YÜCEL ME & ÜNSALAN C. 2022. Planogram Compliance Control via Object Detection, Sequence Alignment, and Focused Iterative Search. arXiv preprint arXiv: 221201004.

How to cite
PABLO K, MATOS SN & BINI TA. 2023. BIOPLAG: An Approach to Detect Programming Plagiarism. An Acad Bras Cienc 95: e20220684. DOI 10.1590/0001-3765202320220684.

Manuscript received on August 10, 2022;
accepted for publication on February 11, 2023

KAIO P. GOMES
https://orcid.org/0000-0002-0886-1484

SIMONE N. MATOS
https://orcid.org/0000-0002-5362-2343

TARCIZIO ALEXANDRE BINI
https://orcid.org/0000-0002-1320-2387

Universidade Tecnológica Federal do Paraná, Departamento de Ciência da Computação, Rua Doutor Washington Subtil Chueire, 330, Jardim Carvalho, 84017-220 Ponta Grossa, PR, Brazil

Correspondence to: **Kaio Pablo Gomes**
E-mail: kaiopablo@live.com

Author contributions

KPB contributed to the writing and research. SNM was an advisor and reviewer of the work. TB reviewed the manuscript. All authors approved the final version.