IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 24, 2020, accepted September 21, 2020, date of publication September 24, 2020, date of current version October 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3026422

Abstract Syntax Tree Based Source Code Antiplagiarism System for Large Projects Set

MICHAL DURACIK, PATRIK HRKUT, EMIL KRSAK, (Member, IEEE), AND STEFAN TOTH
Department of Software Technologies, Faculty of Management Science and Informatics, University of Žilina, 010 26 Žilina, Slovakia

Corresponding author: Michal Duracik (michal.duracik@fri.uniza.sk)

ABSTRACT The paper deals with the issue of detecting plagiarism in source code, which we unfortunately encounter when teaching subjects dealing with programming and software development. Many students want to simplify the completion of the course and therefore submit modified source codes of their classmates or even those found on the Internet. Some try to modify the source code e.g. by changing the identifiers of classes, methods and variables to different ones, by changing the corresponding loops, by introducing new methods or by changing the order of methods in the source code or in other ways. We focused directly on this problem and designed our own anti-plagiarism system that we describe in this paper. The designed system consists of three parts during which the source code is processed using six designed algorithms. The basis is the processing of the source code and its transformation into an abstract syntax tree, consisting of two types of nodes, which is then vectorized using our modified DECKARD algorithm. The vectors are then clustered and stored in a database from which similar parts of the source code can be searched. The output of the system is then the final report containing a list of matches with similarities of all works that have been added to the database until then. The designed anti-plagiarism system is finally compared with the success of plagiarism detection performed by the two most used anti-plagiarism tools, namely JPlag and MOSS. It is evaluated on assignments elaborated by students from the courses dealing with object-oriented programming at our faculty.

INDEX TERMS Anti-plagiarism system, education, plagiarism, software algorithms, source code plagiarism, source code similarities.

I. INTRODUCTION

In the current global pandemic of Covid-19, teaching has moved even further into the online space. Students complete a number of assignments at home and send them to their teachers for review. It is necessary to objectively ensure that the student prepares the assignments independently and does not copy parts of the work of other students or has not misused the finished works found on the Internet. With several thousand assignments per semester, the manual check of the work originality is very lengthy and, moreover, it is not always possible to easily detect fraudulent student behavior.

At our faculty, we have also encountered many cases of various frauds when some students wanted to make it easier to complete the course. Unfortunately, in the subjects dealing with programming, especially teaching the basics of object-oriented programming, we encounter several such cases every semester. As we require a separate solution for some assignments so that students learn to program, understand the basics and gain the necessary experience, such fraudulent behavior is unacceptable. Despite warnings and many reminders, there are always a few students who submit either the same or much more frequently the modified solution to a classmate's task, or those downloaded from the Internet. This is often a refactoring of the source code, where a student either changes the names of classes, methods and attributes, or modifies comments or the order of declared methods. The more skillful ones also change the algorithm by changing the loop to another type of loop, negating the conditions and dividing the method into several methods, either manually or using the support of the integrated development environment.

In the field of textual works, a number of tools and systems are available that make it possible to detect plagiarism, the same parts of texts already written by other authors. The situation is a bit different for source code. Currently, there

The associate editor coordinating the review of this manuscript and approving it for publication was Daniela Cristina Momete.

is no free tool that allows us to compare a large number of works at the same time. Although there are many tools that can compare two projects with each other, we have not come across a freely available system that incrementally adds projects to a set of source code and provides tools to check new source code against a large base of existing source code. Therefore, we, at our faculty, decided to theoretically describe the starting points, find procedures, design and create such a system. The system can be used not only to check online assignments, but can also be used to check final theses, software projects and the like. We dealt in more detail with the motivation and influence of the availability of such a system on the quality of student work in the paper *Issues with the detection of plagiarism in programming courses on a larger scale* [1].

In this paper, we will describe our lessons learned, problems with detecting the same or similar parts in the source code and describe existing similar systems and tools. Subsequently, we will present our designed and implemented system that we will verify and compare with the two currently most used freely available tools.

II. PROBLEMS OF DETECTION OF THE SOURCE CODE PLAGIARISM

The main task of the anti-plagiarism system (hereinafter referred to as APS) is to identify parts of the source code that could be similar to other source codes in the searched set of projects. The problem is that unlike textual works, where it is exactly defined, what plagiarism is [2]; plagiarism detection is much more complicated in the case of source code. Of course, there are cases where plagiarism is obvious and the source codes or their parts are identical. This is the case when two students submit the same project (work). However, such cases are rare. Due to the fact that programming does not matter the names of identifiers (from a functional point of view), we more often encounter work that differs in the names of identifiers (variables, methods, classes, etc.). Such works can again be declared plagiarized, although in terms of the text they are different works. Another example is the work where the student tries to ''mask'' plagiarism by various other procedures, such as changing the order of parameters, shuffling methods between classes, changing the internal structure of classes, etc. All such work would be comfortable with a text matching system, but from a practical point of view, it is plagiarism.

In the source code, the structure and order of its individual elements are important. Simple replacement of individual elements is often not possible and an attempt to do so would cause the final solution to malfunction. Certain modifications are possible but their application often requires a deeper knowledge of the programming language and also of the problem that the work solves. This means that we must also be able to detect certain modifications. Interesting from this point of view is the question of how big modifications we still have to be able to detect. Because the further we go, the more complicated the situation becomes. The definition of plagiarism refers to ''imitation or literal write-off'' [2], which could ultimately mean that all works on one topic (assignment) will be plagiarism, as they provide the same functionality. When looking for plagiarism, we must therefore focus on the detection of the similarities between the individual algorithms and the overall structure of the source code because each programmer will design different algorithms and a different structure of the program that will help them to achieve the goal.

If we identify plagiarism only on the basis of the similarity of algorithms and structures, we may encounter another problem. The problem is that programming often uses learned and proven principles that lead to structurally identical code, regardless of the programmer. As an example, we can mention design patterns that often prescribe the exact structure of objects or algorithms by which we can solve routine programming tasks. During the evaluation, it is necessary to take this phenomenon into account, either during the machine processing or also during the manual evaluation.

The last problem is the use of source code the license of which allows it. Within the source code, there are various licenses that allow its use in foreign programmes without specifying the original source or licence. From the point of view of copyright law, such use is perfectly fine but within the academic sphere, it may not be acceptable in certain cases.

III. CURRENT STATE

The topic of plagiarism has been and still is popular today. Many papers and applications deal with the search for plagiarism in text documents [3]. There are even many freely available tools and methods [4]. In the case of source code, the situation is similar. The most widely used free plagiarism detection systems include the *Measure of Software Similarity* (**MOSS**) [5] from Stanford University and the **JPlag** system [6] from Germany. Both of these systems were created more than ten years ago and their development continues to the present.

JPlag uses a method called Running–Karp–Rabin Greedy-String-Tiling (hereinafter referred to as RKR-GST) to search for plagiarism. Originally, JPlag was only available as a web service; currently it is available as a standalone desktop application. JPlag supports several programming languages – Java, C, C++, and PHP. It provides the output in a graphical form where it is possible to display a table with the similarities of the assignments and then it is possible to display the matches detected between the individual works. The system can run on a set of jobs, comparing each one with each other and creating a report based on this comparison. Other tools that use RKR-GST method include **YAP3** [7]. Research shows that modified version of RKR-GST can be used to find plagiarism across different programming languages [8]. On the other hand, the **MOSS** system uses a method called windowing based on n-grams and their fingerprinting. MOSS is available for several programming languages as well as JPlag. MOSS is implemented as a web service to which individual solutions can be sent for comparison. The report from the system is

similar to the case of the JPlag system. MOSS does not have an interface for loading assignments directly on the website, but it has various other tools available for this purpose from console to GUI ones. These two methods are used in various variations in other tools as well. Both are based on methods that are also used to search for plagiarism in text documents.

Other tools that can be used to search for plagiarism include the **AC** system [9]. It uses a combination of several algorithms when searching. **Plaggie** is a stand-alone source code plagiarism detection engine purposed for Java programming exercises [10]. Plaggie's functionality and graphical user interface is similar to JPlag. **SIDplag** is based on JPlag, it uses a modified version of GST [11]. **ES-Plag** is a plagiarism detection tool featured with cosine-based filtering and penalty mechanism to handle aforementioned issues [12].

The mentioned tools were based on methods commonly used in text documents. With the source code, there are various algorithms that can search for plagiarism due to the knowledge of the code structure. They use syntax trees [13]. There are also various modifications of these algorithms that use tree-weighting principles [14] to improve detection or deal with efficient comparison of syntax trees using fingerprinting [15] or hashing [16]. Systems that use machine learning, such as neural networks, are also innovative [17], [18].

In addition to the above-mentioned freely available applications, there are also commercial solutions that deal with this issue. The best known are **Codequiry** [19], **Unicheck** [20], **CodeMatch** [21], and **ACNP** [22]. These systems again use well-known algorithms used to detect plagiarism. Their main advantage is a more sophisticated user environment in order to simplify the evaluation of works as much as possible.

In the literature, we can also find tools that seek the opposite goal. The **PerfectPlaggie** system deals with the creation of plagiarism (plagiarized material) that plagiarism detection systems are not able to detect. It uses source code obfuscation [23]. There are many tools for source code obfuscation [24]. Although these can deceive the anti-plagiarism system, the detection of source code obfuscation itself can be left to the human responsibility, as it is a simple task for them. In the literature, we can also find methods that utilize semantic analysis [25] for obfuscation detection in such modified codes.

As it is clear from the analysis of the current situation, there is currently no freely available anti-plagiarism system that would allow effective search for plagiarism with the possibility of incremental addition of work, although some theoretical procedures have been developed and none of them has been completed yet.

IV. DESIGN OF ANTI-PLAGIARISM SYSTEM

Searching for plagiarism, especially in large amounts of data, is not an easy and trivial task but a complex process. As we stated in the overview of the current state, there is no comprehensive system for searching for plagiarism in the source code, which would be to a similar extent as those for text documents. There are works that describe the adaptation of text-oriented systems to source code, but our analysis [26] has shown that source code, even if it is only text, differs significantly from text documents. Over time, various methods of source code processing have emerged. Recently, the advantages of working with source code in the form of a tree have been shown [27].

A. SYSTEM STRUCTURE DESIGN

Before designing the system, we set the following requirements:

- Use of efficient ways to process and represent source code because the amount of data stored grows over time and will be enormous.
- A scalable way to detect matches, as the number of works increases (the difficulty as well).
- Appropriate way of data persistence that the speed and efficiency of the search will depend on.
- Versatility of the system that guarantees use on different platforms, seamless access to data input and evaluation results.

Based on these requirements, we decided to divide the prepared system into the following logical blocks, which can be found in the following diagram (Fig. 1).

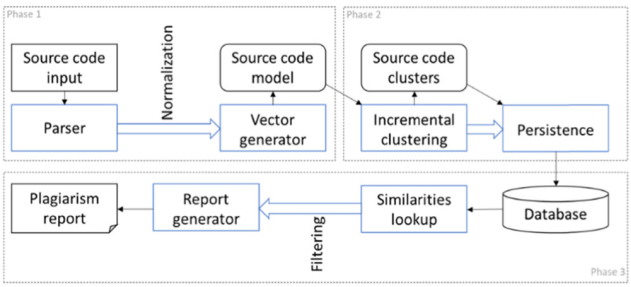

FIGURE 1. Structure of the designed system.

The main advantage of such a designed system is its modularity because it is divided into three separate phases and each one consists of two algorithms (blue rectangles). The individual phases represent:

1. Processing and representation of the source code.
2. Data clustering and persistence.
3. Similarities lookup and generation of reports.

In the first phase, there are algorithms that allow us to process the source code. We divide these algorithms into two groups. The first of these are algorithms that process the source code and create a model of it in the form of a syntax tree. These algorithms will always be specific to each programming language we want to process. The second group consists of algorithms that transform the syntax tree into a form suitable for further processing. Based on the analysis [28], we chose characteristic vectors as a suitable form for the representation of the source code. Among the processing and vectorization, we can include algorithms that will normalize the syntax tree, thanks to which it is possible

to detect plagiarism even in the case of commonly used tricks to deceive APS [27].

In the second phase, the clustering of similar vectors occurs due to higher search efficiency. To cluster similar vectors, we use the known and widely used K-Means algorithm, which we modify for our purposes [29]. The result of this phase will be the data that are ready to be stored in a database in a form that allows them to be easily looked up. The goal of clustering is, in addition to the preparation of data for lookup, also a certain logical division of data into related units. Such a division will be the basis for the scalability of the whole system.

The last phase deals with obtaining individual matches from the database and their evaluation. In this phase, before the actual generation of the report, a filter of non-significant matches can be included which serves to clarify the reports [30]. In the filter, it is possible to define patterns that will not be taken into account in the evaluation – for example, commands for importing packages, generated commands and others.

The individual phases are almost independent of each other. The only element on which they depend is the format of the data transferred between them. Between the first and second phases, a processed source code model is transmitted, which is represented by characteristic vectors. In addition, the bridge between phases 2 and 3 forms a database in which vectors are stored in a suitable form. Such a division brings additional possibilities in scaling the whole system.

B. SOURCE CODE PROCESSING

In this chapter, we will focus on the processing and representation of the source code. We will demonstrate the whole procedure for processing the source code in C# but the same method can be used for other programming languages without a significant modification.

We will not directly parse the source code, as there are a large number of tools that can process the source code. These tools can be provided directly by the language developers (e.g. Roslyn for.NET languages) or by a third party solution (e.g. Javaparser for Java processing). We will focus in more detail on the representation of the source code through a syntax tree and describe the transformation of syntax trees into structures that are easier to be processed by plagiarism search algorithms.

1) ABSTRACT SYNTAX TREE

Abstract Syntax Tree (hereinafter referred to as AST) is one of the most common methods of representing source code in the form of a tree structure. Let us have simple source code (**FIGURE 2**). The syntax tree of algorithm in **FIGURE 2** can be seen in **FIGURE 3**. There are two basic types of nodes in this tree. The first of them are the so-called *SyntaxNode* nodes, shown in blue in the figure. These express an abstract view of the structure of the source code. The second group are the so-called *SyntaxToken* nodes (shown in green, see the figure). These represent the individual tokens.

```
1 for (int i = 0; i < n; i++) {
2     x[i] = 0;
3 }
```

FIGURE 2. Example of an algorithm – for loop.

For each node, we can specify the range that it includes in the source code. An important feature is that each SyntaxNode node has its own type. In the figure, we can see just these types of nodes (*ForStatement, VariableDeclaration, etc.*). A further feature of this tree is that each *SyntaxNode* must have at least one child. *SyntaxToken* nodes are always leaves of this tree. Individual *SyntaxNode* nodes always cover a certain contiguous part of the source code – the block. By post-ordering this tree over *SyntaxToken* nodes, we can get a standard stream of tokens.

Not all nodes in this tree are important to us. *SyntaxNode* nodes will be considered relevant and others irrelevant. This division was introduced because exactly these nodes contain the structural information we require from the source code.

On the other hand, the structure of the syntax tree is rich, and in some cases, we have decided to mark some of the *SyntaxNode* nodes as irrelevant. As an example, we can show the syntax tree of declaring a local variable. In **FIGURE 4**, we can see such a tree. In this case, the *VariableDeclaration* and *VariableDeclarator* nodes are irrelevant to us because they depend directly on their parent and do not add new information on the code structure.

2) VECTORIZATION OF THE SOURCE CODE

Since the comparison of tree structures is a computationally demanding operation, it was necessary to transform such a tree into a linear structure or a collection of structures. One of the possibilities is to use a source code vectorization [28].

As inspiration, we chose the paper *DECKARD: Scalable and Accurate Tree-based Detection of Code Clone* [31]. In this work, the authors introduce an algorithm for creation of characteristic vectors. In addition, they describe a scalable method of clustering these vectors for detection of clones. The source code of this algorithm is publicly available and used in various research papers even today.

During the introduction of the characteristic vectors into our work, we take over the concepts defined by the authors of the DECKARD algorithm. The characteristic vectors are used to capture the structural information of trees or forest of trees. The characteristic vector of a given subtree is thus defined as the point $<c_1, \ldots, c_n>$ in Euclidean space, where each c_i, $i \in 1..n$ represents the frequency of occurrence of a certain pattern in the subtree. The basic pattern that we can follow in this case is the frequency of occurrence of individual types of nodes. Of course, we will not create a component in this vector for each node type. During the analysis, we determined the relevant and irrelevant nodes, and we will use this division in the construction of the vector where we focus only on the relevant nodes.

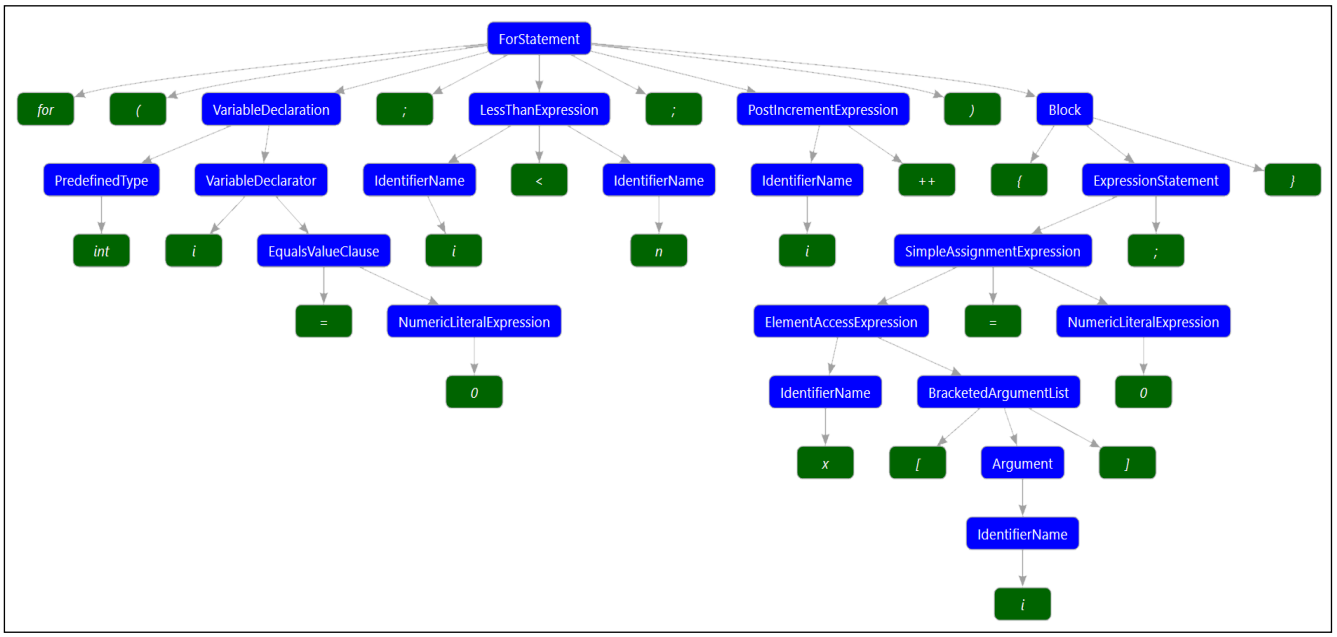

FIGURE 3. Example of a syntax tree from basic for loop.

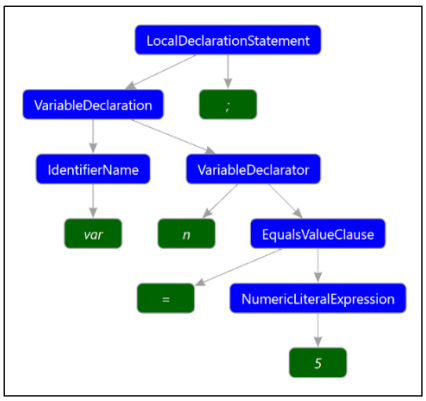

FIGURE 4. Syntactic expression tree – declaration of a variable (var n = 5;).

We generate vectors for all relevant nodes in the syntax tree. When generating, we proceed by post-order inspection of the tree so that for each node we get its characteristic vector by adding the vectors of the children with the vector of the given node (a vector that contains all zeros, except for the element that corresponds to the current node). We adapted the algorithm used in the DECKARD tool to our conditions. This algorithm has got two steps. The first is the generation of vectors for individual nodes of the tree and the second is the connection of vectors. The first step of our system is identical to the DECKARD system. We modified the second step to better suit the needs of plagiarism search. We will describe this modification in the following section.

ALGORITHM 1 shows an example of the generation of vectors within the first phase of the DECKARD algorithm (for simplicity, we consider a vector consisting of seven elements). The algorithm proceeds from the leaves and

Algorithm 1 Vector Merge (Modified Deckard)

```
1: function V_MERGE (T: tree, M: int, S: int): vectors
2:     ST ← Serialize(T); V ← ∅
3:     step ← ∅; front ← ST.first
4:     repeat
5:         if RightStep (step, M) then
6:             V_merged ← V_front
7:             n ← NextNode (front)
8:             while n != ST.Last ∧
               ¬ContainsEnoughTokens(V_merged, M) do
9:                 n ← NextNode(n)
10:                id ← IndexOf(n)
11:                V_merged[id] ← V_merged[id] + 1
12:            end while
13:        end if
14:        front ← NextNode(front)
15:        step ← step + 1
16:    until front = ST.Last
17:    return V
28:end function
```

individual vectors are created gradually. For example (Fig. 5), the vector of the for loop consists of three vectors – loop declaration, loop body and unit vector of the for block - *<0, 0, 0, 0, 0, 0, 1>*

3) MODIFICATION OF THE ALGORITHM FOR VECTOR MERGE

The main reason for the vector merge is the need to cover the source code with more evenly generated vectors. In the initial generation of vectors, the vectors are generated based on the source code structure. These vectors cover logical parts of the

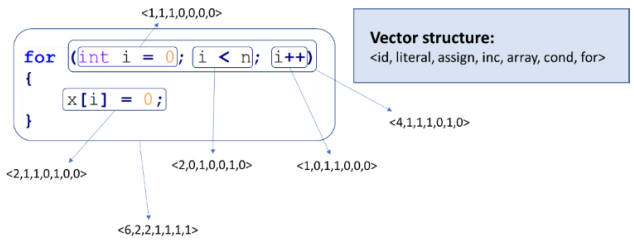

FIGURE 5. Example of possible vectors.

code (expressions, blocks, methods, etc.). The vector merge creates vectors from adjacent pieces of the source code (for example *n* contiguous expressions, or from two methods).

From the code in **FIGURE 6**, vectors for the condition (lines 4 – 6), method (lines 2 – 7) and class (lines 1 – 8) will be generated in the basic algorithm (with optimal selection of parameters). During the vector merge, we will proceed through the source code gradually and (with a suitable selection of parameters) we can obtain vectors, for example for lines 1 – 3 or 3 – 6. Some of the merged vectors may not be very important (lines 1 – 3), whereas others (lines 3 - 6) may be significant.

```
1 class A {
2     void Method() {
3         int a = 5;
4         if (a > 0) {
5             a++;
6         }
7     }
8 }
```

FIGURE 6. Sample of a simple class source code.

The algorithm works with node vectors generated in the previous algorithm. The algorithm used by the DECKARD tool consists of several steps. In the first step, it serializes the parse tree in the post-order. Subsequently, a window is moved along the serialized tree, within which it connects the vectors of nodes that are located in this window. The parameters of this algorithm are the size of the window (defined by the minimum length of the merged vector) and the shift by which the window is moved after each generated vector. Larger window sizes ensure that larger pieces of the code are merged, and shifts allow us to determine how those pieces overlap.

In our case, it was not possible to apply this algorithm directly as we encountered the problem of multiple addition of vectors in the described algorithm. If we only process terminal nodes (leaves) during the post-order inspection of the tree, the merged vectors correctly represent the processed part of the tree. However, if we traverse towards the root during the inspection, then by adding the vectors of the child with the father's vector, we obtain 2 times counted vectors of the children, as these are already included in the father's vector. We solved this problem by using their own vector in addition to the initial node during the merge of the nodes.

The parameters of the **ALGORITHM 1** are T (syntactic tree), M (minimum vector size) and S (step size). The algorithm gradually traverses the serialized tree. At the beginning of each step, it is evaluated whether it is the correct step (the *RightStep* method according to the parameter M). Subsequently, the merge algorithm itself consists of several steps. The first one is the assignment of the initial vector. Next, the algorithm continues by adding the actual vectors of the following nodes to the resulting vector until the resulting merged vector has the sufficient size or we have reached the end. After reaching the desired size, the merged vector is added to the set V and we move on to the next step. At the end, the algorithm returns a list of generated vectors.

Such a merge algorithm can often generate non-significant vectors (e.g. a vector that covers the code, that begins in the middle of one method and ends in the middle of the other method); therefore, we designed its modification. This modification assumes that plagiarized material is created mainly by copying whole blocks (for example the whole method, a loop, etc.) or by copying several commands (method bodies). The first method is well included in the basic vector generator process. The second method is included, inter alia, in the vector merge algorithm. Our modified algorithm therefore aims to merge sibling nodes. It might seem that the merging of sibling nodes is not very important because the vectors of these nodes are usually contained in the node of their parent but we will show by example when such a case is important.

In the case of this code, the basic algorithm (with the appropriate selection of parameters) will generate only two vectors (one vector for class – lines 1 to 10 and one vector for method – lines 2 to 9). No vectors will be generated from the expressions contained in the method, as their vectors are not large enough. In the case of using our designed algorithm, we can generate vectors from lines 3 – 5, 4 – 6, 6 – 8 that will help us in detection of partial plagiarisms. For comparison, **FIGURE 7** also lists the code sections from which the original DECKARD algorithm generates vectors.

Similar to the basic vector merge algorithm, our algorithm (**ALGORITHM 2**) has got the parameters T (syntactic tree), M (minimum vector size) and S (step size). The algorithm sequentially traverses the individual nodes of the syntax tree. If it detects a nonterminal node that has a sufficiently long vector (because it does not make sense to merge children unless their father has a sufficient length of the vector), line 4, it gradually starts to merge children similarly to the previous case. In this algorithm, we do not merge our own vectors as we did in the previous algorithm but we do the vector sum of the vectors of the given nodes (line 13). At the end of the algorithm, we return a list of the detected vectors.

C. CLUSTERING AND PERSISTENCE

After processing the source code, we obtained a list of vectors that characterize the source code. There are many of these vectors and they need to be processed. Using clustering,

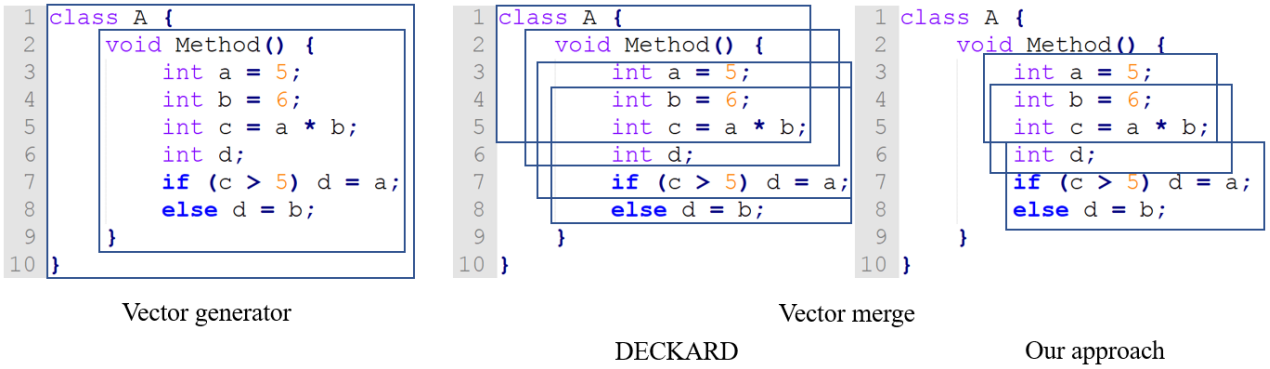

FIGURE 7. Blocks of code from which vectors are generated when merging vectors.

Algorithm 2 Vector Merge (Our Approach)

```
1: function V_SIB_MERGE(T: tree, M: int, S: int): vectors
2:     V ← ∅
3:     for all node N in T do
4:         if IsNonTerminal(N) ∧ ContainsEnoughTokens(V_N) then
5:             CH ← N.childrens
6:             step ← θ; front ← CH.first
7:             repeat
8:                 if RightStep(step, M) then
9:                     V_merged ← V_front
10:                    n ← NextNode(front)
11:                    while n != CH. Last ∧ ¬ContainsEnoughTokens (V_merged, M) do
12:                        n ← NextNode(n)
13:                        V_merged ← V_merged + V_n
14:                    end while
15:                end if
16:                front ← NextNode(front)
17:                step ← step + 1
18:            until front = CH. Last
19:        end if
20:    end for
21:    return V
22: end function
```

we divide the vectors into groups that we will then process separately. Such a division can be used in the search, so we do not have to search in a large group of vectors but we only need to search one cluster when searching for similarities.

For the needs of the source code clustering, we used the well-known **K-Means** method for characteristic vectors and we designed its **extension for the needs of continuous source code processing** [29]. Incremental clustering is based on the fact that if we already have clusters with a sufficient number of vectors, the addition of one vector to this system will not cause a fundamental change in the distribution of clusters. Based on our experiments [29], with a sufficiently large initial dataset, the addition of a single vector will cause a change in cluster distribution in a very small number of cases. With a sufficiently large number of vectors, these small shifts accumulate. The solution designed in such a way will make it possible to perform re-clustering only after a certain number of added vectors.

The K-Means algorithm has got one parameter – the number of clusters. We tried to determine this parameter experimentally. In the experiments, we monitored selected characteristics such as cluster radius, distance between clusters, entropy of clusters, and ''similarity of vectors in clusters''. The result of this experiment was the finding that there is a certain limit while it pays to add more clusters but once this limit is exceeded, it does not make sense to add more. It also proved that this number increases with the total number of vectors. However, using the designed algorithm, this parameter is dynamic. We can always reduce / increase the number of clusters when re-clustering [29].

In addition to the design of an incremental algorithm, we **optimized the K-Means algorithm itself** [32] to be able to process the **required amounts of the source code** and way how these optimizations contribute to the scalability of the algorithm. Unlike conventional methods of optimization of the K-Means algorithm, we did not deal with the initial distribution of centres, but we optimized the algorithm itself. We introduced parallelization and implemented heuristics in order to be able to assign vectors to clusters in a more efficient way.

Finally, we designed **a method of data persistence** [30] based on clustering using a relational database – see **FIGURE 9**. When dealing with data persistence, we also dealt with the efficiency of the search in this data structure. The aim was to find out as efficiently as possible whether the given vector is in the database or not. It was not possible to index a vector that has more than 100 elements, so we detected significant components using conditional entropy and indexed only those. It proved that on average it is enough to select only five elements of the vector and create an index from them [29].

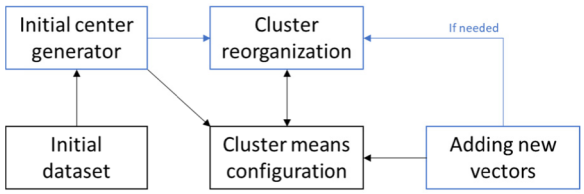

FIGURE 8. Incremental clustering scheme.

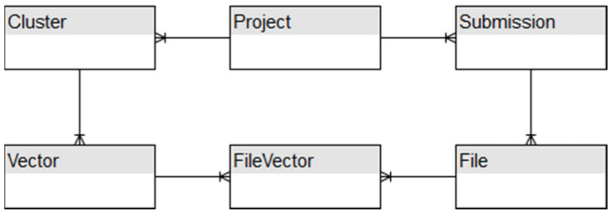

FIGURE 9. Data structure for filing vectors.

D. SEARCH FOR THE MATCHING PARTS OF THE SOURCE CODE

The last part of the designed system for searching for plagiarism in the source code is to search for similar parts of the source code. The basis of our plagiarism search algorithm is the data structure (see **FIGURE 9**) designed in our previous work [30]. The disadvantage of such a design is that we are able to generate a report only for works (students' assignments) that we have already added to the database before. However, from the point of view of the designed method, this approach is completely fine. Our goal is that any code we add to the database can be futher used to search for plagiarism. Unlike commonly used systems, the designed system will not search for plagiarism in the whole data set but will allow obtaining a list of matches for one specific work.

The two source code fragments will be marked as identical if their characteristic vectors are identical. This means that only the code that has the same structure will be considered plagiarized. The plagiarism search algorithm consists of three parts. The first one is to obtain similarities from the database, the second one is to match and filter these similarities, and in the third part, the degree of similarity is calculated for the detected pairs of works [30].

1) SEARCH FOR THE MATCHING VECTORS

The input to this algorithm is the identification number of the works for which we want to find the matching vectors. In addition, the algorithm needs access to the clustering data. In the first step, for each vector from the checked work, the algorithm searches for the same vectors in other works that are in the database. In this way, a list of pairs $<vf_1, vf_2>$ is created, where vf_1 is a vector that comes from the controlled work and vf_2 is a matching vector from another work detected for it. After obtaining the list of pairs, its transformation follows in which we divide this list based on the works from which the vector vf_2 originated. In this way, we get a list of works that can be similar to the controlled work, along with a list of individual similarities.

2) MERGING OF THE DETECTED MATCHES

For better processing and subsequent visualization, it is necessary to process the detected matches between the two works further. In order to determine how similar the individual entries are, we need to obtain the disjunctive parts of the source code in which the match was detected. Since the vectors were generated so that they overlap, we can find a large number of vectors in the list of matches that overlap in some way. The algorithm searches for all disjunctive sets that can be composed of these matches.

3) MATCH CALCULATION FOR A PAIR OF WORKS

The similarity of works A and B is expressed by the formula:

$$\boldsymbol{Sim(A,B)} = \frac{\sum_{\boldsymbol{m}} \in \boldsymbol{matches(A,B)coverage(m_A)}}{\boldsymbol{coverage(A)}}$$

It is important to note that the similarity thus defined is not symmetrical, i.e. $Sim\,(A, B)! = Sim\,(B, A)$. In the formula, the *coverage* (A) function represents the amount of the source code that is covered by vectors (by the term amount we mean the number of characters). When calculating the coverage, care must be taken that not every piece of the source code needs to be covered by a vector. The numerator represents the amount of the code that is contained in the detected matches.

4) FILTERING NON-SIGNIFICANT PARTS OF THE SOURCE CODE

In each source code, we can find some parts that are not significant from the point of view of plagiarism detection. As an example, we can mention various imports at the beginning of the source code or automatically generated code using the development environment. The removal of imports could be done already during the processing of the source code but our goal was to design a universal method. This method consists of two parts.

The first is the manual annotation of non-significant parts of the source code [1]. In this method, we manually went through several works and marked sample vectors that represented various non-significant pieces of the code. Subsequently, we modified the algorithm so that it does not even read these vectors from the database.

The second is the semi-automatic removal of the non-significant vectors based on clustering [33]. Based on the annotation of several vectors, we were able to identify clusters that contained only non-significant vectors. Removal of these clusters greatly increased the number of non-significant vectors we were able to identify.

V. EXPERIMENT

After implementation of the system and verifying the individual parts, we evaluated it. We used real student works in the evaluation. The aim was to compare the implemented system

with other commonly available systems. For comparison, we chose the JPlag and MOSS systems.

A. COMPARISON WITH THE MOSS SYSTEM

For the purpose of evaluation of our algorithm, we have prepared a set of works created in the C# programming language. We added one work to this set twice for verification, so that a 100% match was created, which will serve as a reference indicator. In addition, the set contained several real pieces of the plagiarized material/plagiarism that were created directly by the students.

The set consisted of a total of **227** works from **5**different groups ranging from simple console applications to database management systems with a GUI interface. It contained **3,989** files with **291,862** lines of code and **72,107** lines of comments. A total of **169,013** characteristic vectors were generated and **79,590** of them were unique.

The first step in the analysis of the results of our algorithm was to manually check the detected matches. Due to the relatively large number of the detected matches, we did not evaluate all matches but focused only on some of them. First, we evaluated the plagiarism we produced. That system detected 100% compliance. Other pieces of plagiarism, which we also knew about, also achieved a relatively high match score (55% - 75%). Based on a manual check of these assignments, we can confirm that the assigned percentage corresponds to the amount of matching code in these works. The percentage of compliance between other assignments was relatively low (up to 30%). From the experience with other tools, we must say that such a percentage of compliance certainly does not indicate plagiarism.

We compared the achieved results with the results obtained from the MOSS system. The problem with the MOSS system, which we encountered, was the limitation on the maximum amount of work that the system can process. We had to compare individual assignments in groups, and then combine the individual results. Another problem was that the output generated by the MOSS system does not allow comparing specific matches. In the comparison, we only had to focus on comparing the degree of similarity that the respective systems determined for the given assignments.

FIGURE 10 shows a comparison of the matches detected for the 50 most similar pairs of works. The pairs were selected based on the similarity determined by our system. We can see from the graph that, in general, our system admits a higher degree of similarity than MOSS (except for one case). MOSS even assigned only 91% similarity to our 100% copy. This difference is caused by the different calculation of the resulting match, as the MOSS calculates the degree of similarity based on the number of similar lines. From the report generated by MOSS, it was clear that some blank lines were not marked as identical, and thus the pair of works received a lower score. Based on this comparison, we can declare that our system better handles parts of files that do not directly contain code (free lines, spaces, etc.) and therefore its result is more accurate.

B. COMPARISON WITH THE JPLAG SYSTEM

The second system, that we compared our system with, was the JPlag system. Since JPlag cannot work with the latest version of C# and is suitable for checking the source code written in Java where it also provides support for the current version of this language, we prepared a new data set. This consisted of semester assignments in two subjects where the basics of object-oriented programming in Java are taught at our faculty. When processing the source code, we generated two sets of characteristic vectors. Since we were also inspired by the DECKARD algorithm when designing the vector structure, the first set of vectors was generated by this tool. In this way, we wanted to verify the compatibility of our solution with the generator, which is part of the DECKARD system. The second set was generated by its own algorithm. In addition, the generated vectors were partly manually annotated to show the impact of annotation of non-significant parts on the number of false positive cases [1]. When annotating, we manually marked parts of the code that commonly occur. These included mainly imports at the beginning of classes; parts generated using a GUI generator and simple methods (getters, setters).

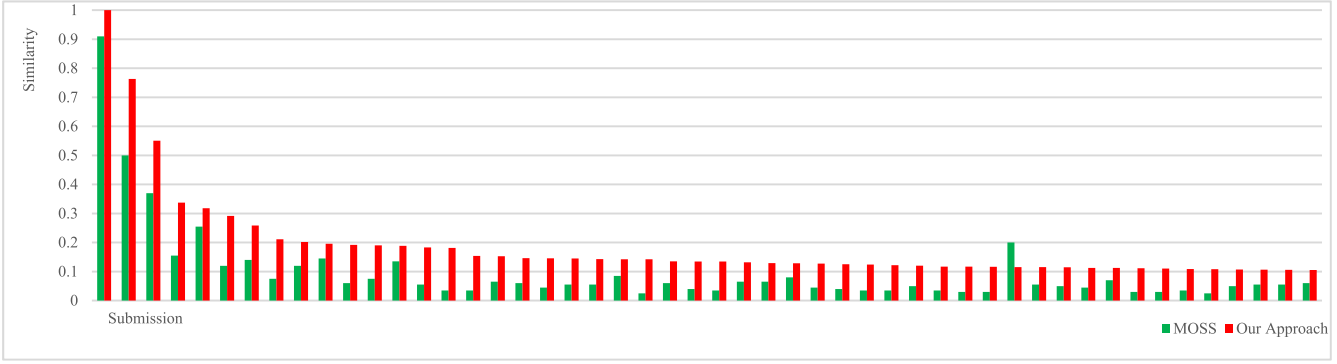

FIGURE 10. Comparison of similarities for the selected 50 tasks.

In total, the set contained **555** assignments. The assignments consisted of **6,889** files with **612,141** lines of code and **216,546** lines of comments. Using the DECKARD algorithm, **425,563** vectors were generated, of which **164,356** were unique. We generated **456,672** vectors with our system and **227,825** of them were unique.

These assignments include several previously known plagiarisms created by students. Some of them are almost 100% copies; others are modified to some extent. The dataset also contained the works of several students who committed the so-called self-plagiarism, when they submitted the same semester work for two years in a row while re-taking the subject.

We proceeded with the evaluation in a similar way as in the previous case. Even the JPlag system does not allow program processing of the detected matches. Again, we relied only on a comparison of the calculated percentage matches. To compare the algorithms, we selected 17 works for which JPlag detected at least one match with a similarity higher than 50%. Using our algorithm, we searched for all plagiarism for the selected works (again, we determined that we would mark works that would have a similarity greater than 50% as possible plagiarism). We manually evaluated all the detected matches and determined which of them we really considered to be plagiarized.

In Table 1, we see a list of selected 17 works. In the first column, there is the identifier of the work, in the second column there is the number of possible plagiarism detected by the JPlag system and behind the slash, there is the number of those that were real plagiarisms. In the next four columns, we can see the number of plagiarisms detected by our system. In the first two of them, the results are based on vectors generated by the DECKARD algorithm. In the last two, the results are based on our source code vectorization algorithm. In both cases, we present the results before and after the manual identification of the non-significant parts of the source code.

In the table, we can see seven cases (marked in green) where our algorithm, unlike JPlag, was able to detect only the relevant matches. In seven cases (not marked), the algorithm effectively eliminated incorrectly reported matches by JPlag. In one case (submission 306), the algorithm failed to detect the real plagiarism that JPlag was able to detect. In this case, only the set, which was generated by our algorithm with annotation of the non-significant parts of the source code, detected plagiarism. In Table 2, we can see a comparison of the number of works that have been falsely marked as plagiarized. The table shows the number of works that matched more than 50% and were not actually plagiarisms.

TABLE 1. Match comparison for selected 17 works.

Submission	JPlag	Our Approach			
		DECKARD		Our vector generator	
		Basic	Annotated	Basic	Annotated
130	3/1	1/1	1/1	1/1	1/1
489	1/1	1/1	1/1	1/1	1/1
382	1/0	0/0	0/0	0/0	0/0
242	1/1	0/0	0/0	0/0	0/0
435	1/0	0/0	0/0	0/0	0/0
272	2/0	0/0	0/0	0/0	0/0
356	1/1	1/1	1/1	1/1	1/1
186	1/1	1/1	1/1	1/1	1/1
446	5/1	1/1	1/1	1/1	1/1
271	1/1	1/1	1/1	1/1	1/1
306	1/1	0/0	0/0	0/0	1/1
546	3/1	1/1	1/1	1/1	1/1
198	2/0	0/0	0/0	0/0	0/0
9	1/0	0/0	0/0	0/0	0/0
179	1/0	0/0	0/0	0/0	0/0
297	1/1	1/1	1/1	1/1	1/1

TABLE 2. Number of false positive cases.

JPlag	DECKARD		Our vector generator	
	Basic	Annotated	Basic	Annotated
26	9	5	14	4

In addition to comparing the number of the detected matches, it is appropriate to focus on the individual similarities of the selected pairs of works. This comparison is shown in Table 3. The values in the table are highlighted with green, for which we consider the work to be plagiarism, and in fact, it was the plagiarism. Blue is falsely marked plagiarism. The values highlighted with orange are real plagiarism pieces that have received less than 50% compliance. In most cases, our system experienced a lower degree of similarity. The reference pair "356-415" (the same work by the same student submitted two times, i.e. 100% compliance) detected complete compliance using both algorithms. This table shows why our system did not detect plagiarism (with a maximum score of 11.5%, we will hardly consider it plagiarism – a line with red letters). In addition, we can see that the pairs that JPlag falsely detected achieved relatively low scores in our algorithm.

In the analysis of why the algorithm could not detect the mentioned plagiarism, the DECKARD system showed that there was a problem in the vectorization of the source code. DECKARD was unable to process all files. 12% of the files were completely missing and for 10 % of the files not enough vectors were generated. This was because DECKARD incorrectly processed the source code and excluded some files from the comparison. Unfortunately, we were not able to influence this behaviour. These complications reduced the overall efficiency of the algorithm using the DECKARD system.

TABLE 3. Comparison of similarities of the selected pair of assignments.

Submission		JPlag	Our Approach				Plagiarism?
			DECKARD		Our vector generator		
A	B		Basic	Annotated	Basic	Annotated	
356	415	100	100	100	100	99,3	Yes
186	163	97,8	99,2	99,8	95,4	95,5	Yes
271	138	79,6	70,4	70,7	68,2	69,3	Yes
446	450	78,6	68,3	49,8	71	69,8	Yes
130	272	77,4	57,2	57,2	52,3	52,3	Yes
271	224	68,7	53,8	47,2	8,2	1,4	No
306	106	68,4	16,4	19,2	46,6	52,8	Yes
297	182	67,7	51,2	30,2	53,8	39,7	Yes
546	520	66	59,4	51,2	70,7	75,9	Yes
489	441	61,8	53,8	53,8	51,2	50,8	Yes
130	259	60,8	28,4	28,4	12,1	10,9	No
198	188	59,9	40,2	16,1	46,8	43,6	No
546	337	58,3	44,1	12,6	43,8	44,8	No
382	111	56,8	3,4	3,4	5,8	5,8	No
242	426	56,5	7,1	7,1	11,5	11	Yes
435	347	56,5	39,1	39,1	20,3	19	No
446	296	54,9	28,9	12,4	45,3	42,6	No
446	133	53,6	33,8	9,1	46,3	44,8	No
272	109	53,1	23,7	23,7	23,9	23,7	No
130	109	52,3	34,2	34,2	18,2	15,1	No
272	259	50,4	12,4	12,4	16,9	16,9	No

Using the algorithm that we implemented in order to vectorize the source code, the results were better, but even that one was not able to detect all the plagiarisms. In this case, the problem was the overly strict representation of the source code using vectors – in one of the works the keyword this was removed from all methods that caused the algorithm to be unable to detect similarities.

Our results show that in the case of the so-called "CRTL + C, CTRL + V" plagiarisms we were able to detect plagiarism with 100 % success both in the case of the student work and in various tested fragments. However, such plagiarism is not commonly found in student works due to its simple detection. We encounter plagiarism with some modifications more often. Basic modification techniques – editing comments, adding / removing whitespaces, changing the name of identifiers, can, by our algorithm, similarly to the previous case, be detected in 100% of cases. For the algorithm, none of the mentioned modifications represents a relevant source of information.

This is not the case if the modification changes the structure of the source code. The methods of pre-preparation or normalization of the source code, which can be found in literature [34], [35] can also be applied to our system. We have not yet dealt with their verification in this research work. It turned out that some of them are not even necessary, as the very nature of the designed algorithm can solve the situations that these methods solve. The results showed that some changes in the structure – especially shuffling parts of the code – can be detected by our system. If a student shuffles larger blocks – for example methods (blocks that are covered by a separate vector), our system can correctly identify these shuffled methods. If very small blocks are shuffled – typically individual expressions (all blocks are covered in one vector), the system can also reliably identify these cases. The problem arises when the blocks we shuffle are small enough not to form separate vectors but large enough to be all covered by a single common vector. This situation can be solved by various settings of the relevant parameters.

The last modification we considered in the evaluation is to add / remove parts of the code. This type of modification is the most complicated to create for the student but also to detect by our system. When generating vectors, these are generated hierarchically, i.e. even a small change at the lowest level of the syntax tree is reflected in all vectors. The main

component, thanks to which it is possible to detect even these modifications, is the phase of the vector merge in the generation of the characteristic vectors. However, our results show that even this is not a sufficient solution to this problem. If the source code is heavily nested, we cannot effectively suppress these types of modifications.

When identifying plagiarism, it must be taken into account that learned techniques as well as design patterns are often used in programming and the development environments generate a lot of code for the programmer. For this reason, commonly used tools identify a large number of false-positive matches. Our system was no exception, and it also marked several works as plagiarism, even though the work was not plagiarized. The designed method of annotation of the non-significant source code proved to be beneficial especially in the elimination of false-positive matches that arose from the automatic generation of code using the development environment.

In the evaluation of the individual algorithms, we do not state computational complexity. Currently, if we are looking for all plagiarism in a large group of works, the system is slower compared to the JPlag system. On the other hand, if we need to evaluate one work against others that we already have in the database, our approach is faster. When designing the system, our goal was not to achieve the highest efficient implementation (we optimized only the parts necessary for the needs of the evaluation) but we focused mainly on the scalability and reliability of the detection. During the research work, we tried to streamline those parts that slowed down its speed the most, such as optimization of the implementation of the K-Means algorithm [32] or storage of data in a relational database. With these adjustments, we have been able to make the search for matches efficient and fast.

VI. CONCLUSION

As we have shown in the experiment, the described system provides us with better results than the currently most used tools MOSS and JPlag. In addition, our system provides the possibility of incremental addition of works and for their evaluation, it is not necessary to repeatedly run the process for the whole set of works but it is always sufficient to compare a new work with works already added to the system.

Of course, there are also suggestions for improvement of the existing features, such as normalization of inputs that would guarantee that minor modifications to the source code, such as changing the logical expression in the conditions, would not affect the masking of the plagiarism. Also, modifying the vector generator so that the change at the end of the tree does not affect the vectors of larger units that in turn would improve the accuracy of the search for the matches and minimize the number of false positives. Another improvement could be a more efficient algorithm for the annotation of the non-significant code parts that would simplify and speed up the search, as we would be able to filter the source code that is not relevant for plagiarism searches in a more efficient way.

Despite these suggestions for improvement, we want to state that the proposed system was not only theoretically designed, but also implemented by us. Moreover, after its successful testing in the subjects taught at our faculty, we plan to make the system available to the public via the web. We hope that this will not only help teachers assess students' assignments but we also wish to encourage students not to resort to unfair practices in the elaboration of their assignments but rather try to elaborate assignments by themselves and learn by doing so as much as possible and thus become more successful in their studies.

REFERENCES

[1] M. Ďuračík, E. Kršak, and P. Hrkút, "Issues with the detection of plagiarism in programming courses on a larger scale," in *Proc. Int. Conf. Emerg. eLearn. Technol. Appl.*, Nov. 2018, pp. 141–148, doi: 10.1109/ICETA.2018.8572260.
[2] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism," *IEEE Trans. Educ.*, vol. 51, no. 2, pp. 195–200, May 2008.
[3] S. Awasthi, "Plagiarism and academic misconduct: A systematic review," *DESIDOC J. Libr. Inf. Technol.*, vol. 39, no. 2, pp. 94–100, 2019.
[4] R. R. Naik, M. B. Landge, and C. N. Mahender, "A review on plagiarism detection tools," *Int. J. Comput. Appl.*, vol. 125, no. 11, pp. 16–22, 2015, doi: 10.1007/s00122-005-2042-4.
[5] A. Aiken. *A System for Detecting Software Similarity*. Accessed: Jul. 15, 2020. [Online]. Available: https://theory.stanford.edu/~aiken/moss/
[6] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. UCS*, vol. 8, no. 11, p. 1016, 2002.
[7] M. J. Wise, "YAP3: Improved detection of similarities in computer program and other texts," in *Proc. 27th SIGCSE Tech. Symp. Comput. Sci. Edu.*, 1996, pp. 130–134.
[8] T. Foltýnek, R. Všianský, N. Meuschke, D. Dlabolová, and B. Gipp, "Cross-language source code plagiarism detection using explicit semantic analysis and scored greedy string tilling," in *Proc. ACM/IEEE Joint Conf. Digit. Libraries*, Aug. 2020, pp. 523–524.
[9] M. Freire and A. Sopan, "Gene similarity uncovers mutation path VAST 2010 mini challenge 3 award: Innovative tool adaptation," in *Proc. IEEE Symp. Vis. Analytics Sci. Technol.*, Oct. 2010, pp. 287–288.
[10] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises," in *Proc. 6th Baltic Sea Conf. Comput. Edu. Res. Koli Calling-Baltic Sea*, 2006, pp. 141–142.
[11] D. Chudá and B. Kováčová, "Checking plagiarism in e-learning," in *Proc. 11th Int. Conf. Comput. Syst. Technol. Workshop PhD Students Comput. Int. Conf. Comput. Syst. Technol. (CompSysTech)*, 2010, pp. 419–424.
[12] L. Sulistiani and O. Karnalim, "ES-plag: Efficient and sensitive source code plagiarism detection tool for academic environment," *Comput. Appl. Eng. Edu.*, vol. 27, no. 1, pp. 166–182, Jan. 2019.
[13] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code comparison system based on abstract syntax tree," in *Proc. 3rd IEEE Int. Conf. Broadband Netw. Multimedia Technol. (IC-BNMT)*, Oct. 2010, pp. 668–673.
[14] D. Fu, Y. Xu, H. Yu, and B. Yang, "WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection," *Sci. Program.*, vol. 2017, pp. 1–8, Feb. 2017.
[15] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 243–247.
[16] G. Tao, D. Guowei, Q. Hu, and C. Baojiang, "Improved plagiarism detection algorithm based on abstract syntax tree," in *Proc. 4th Int. Conf. Emerg. Intell. Data Web Technol.*, Sep. 2013, pp. 714–719, doi: 10.1109/EIDWT.2013.129.
[17] V. Ljubovic and E. Pajic, "Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories," *IEEE Access*, vol. 8, pp. 96505–96514, 2020.
[18] F. Ullah, J. Wang, S. Jabbar, F. Al-Turjman, and M. Alazab, "Source code authorship attribution using hybrid approach of program dependence graph and deep learning model," *IEEE Access*, vol. 7, pp. 141987–141999, 2019.
[19] *Codequiry—Code Plagiarism & Similarity Checker*. Accessed: Jul. 16, 2020. [Online]. Available: https://codequiry.com/

[20] *Unicheck—Plagiarism Checker for Educators and Students*. Accessed: Jul. 16, 2020. [Online]. Available: https://unicheck.com/
[21] *CodeMach—Plagiarism Detection System. SAFE Corporation*. Accessed: Jul. 16, 2020. [Online]. Available: https://www.safe-corp.com/products_codematch.htm
[22] *AntiCutAndPaste—Copied and Pasted Source Code Detector*. Accessed: Jul. 16, 2020. http://www.plagiarism-report.com/anticutandpaste/
[23] J. Petrík, "PerfectPlaggie: Source code plagiarising tool," in *Proc. 12th Student Res. Conf. Inform. Inf. Technol. (IIT SRC)*, Apr. 2016, pp. 38–44.
[24] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The effectiveness of source code obfuscation: An experimental assessment," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 178–187.
[25] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 389–400.
[26] M. Ďuracík, E. Kršák, and P. Hrkút, "Current trends in source code analysis, plagiarism detection and issues of analysis big datasets," *Procedia Eng.*, vol. 192, pp. 136–141, 2017, doi: 10.1016/j.proeng.2017.06.024.
[27] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An AST-based code plagiarism detection algorithm," in *Proc. 10th Int. Conf. Broadband Wireless Comput., Commun. Appl. (BWCCA)*, Nov. 2015, pp. 178–182, doi: 10.1109/BWCCA.2015.52.
[28] M. Ďuracík, E. Kršák, and P. Hrkút, "Source code representations for plagiarism detection," in *Proc. Int. Workshop Learn. Technol. Educ. Cloud*, vol. 870, 2018, pp. 61–69.
[29] M. Ďuracík, E. Kršák, and P. Hrkút, "Searching source code fragments using incremental clustering," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 13, p. e5416, Jul. 2020, doi: 10.1002/cpe.5416.
[30] M. Ďuracík, E. Kršák, and P. Hrkút, "Scalable source code plagiarism detection using source code vectors clustering," in *Proc. IEEE 9th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Nov. 2018, pp. 499–502, doi: 10.1109/ICSESS.2018.8663708.
[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 96–105.
[32] P. Hrkút, M. Duracík, M. Mikušová, M. Callejas-Cuervo, and J. Zukowska, "Increasing K-means clustering algorithm effectivity for using in source code plagiarism detection," in *Smart Technologies, Systems and Applications* (Communications in Computer and Information Science), vol. 1154. Cham, Switzerland: Springer, 2020.
[33] M. Ďuracík, "Semi-automatic identification of non-significant source code parts using clustering," in *Proc. Math. Sci. Technol., MIST Conf.*, 2019, pp. 17–22.
[34] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.
[35] F. B. Allyson, M. L. Danilo, S. M. Jose, and B. C. Giovanni, "Sherlock N-overlap: Invasive normalization and overlap coefficient for the similarity analysis between source code," *IEEE Trans. Comput.*, vol. 68, no. 5, pp. 740–751, May 2019.

MICHAL DURACIK was born in Žilina, Slovakia, in 1993. He received the bachelor's degree (Bc.) in informatics and the master's (Ing.) and Ph.D. degrees in applied informatics from the University of Žilina, Slovakia, in 2014, 2016, and 2019, respectively.

Since 2019, he has been an Assistant Professor with the Department of Software Technologies, Faculty of Management Science and Informatics, University of Žilina. He is teaching subjects focused on algorithmization, object-oriented programming, application design, and web application development in bachelor's and master's degree studies. He has participated in several research projects with international participation, such as INNOSOC, X2Rail, and Optima. His research interests include source code processing, natural language processing, anti-plagiarism, and intelligent transport systems.

PATRIK HRKUT was born in Ružomberok, Slovakia, in 1972. He received the master's (Ing.) and Ph.D. degrees in applied informatics from the University of Žilina, Slovakia, in 1995 and 2010, respectively.

Since 1995, he has been working as an Assistant Professor with the Department of Software Technologies, Faculty of Management Science and Informatics, University of Žilina. He is teaching subjects focused on web and mobile applications development in bachelor's and master's degree studies. He has participated on many international research and educational projects, such as JoinITS, Connect, EasyWay, Intras, Acesyri, and Ceres. His research interests include source code processing, natural language processing, anti-plagiarism, and intelligent transport systems.

Dr. Hrkut is a member of the Centre of Excellence for Systems and Services of Intelligent Transport. He was a recipient of the Werner von Siemens Excellence Award, in 2003.

EMIL KRSAK (Member, IEEE) was born in Námestovo, Slovakia, in 1968. He received the master's degree (Ing.) in applied informatics and the Ph.D. degree in transport and communication systems from the University of Žilina, Slovakia, in 1992 and 2002, respectively.

Since 1992, he has been an Assistant Professor with the Department of Software Technologies, Faculty of Management Science and Informatics, University of Žilina. Since 2014, he has been the Dean of the Faculty of Management Science and Informatics. He is teaching subjects focused on object-oriented programming, such as C# language and.NET and advanced object technologies in bachelor's and master's degree studies. He has participated in several research projects with international participation, such as Connect, EasyWay, and the Centre of Excellence for Intelligent Transportation Systems. His research interests include source code processing, anti-plagiarism, and intelligent transport systems.

STEFAN TOTH was born in Bojnice, Slovakia, in 1986. He received the bachelor's degree (Bc.) in informatics and the master's (Ing.) and Ph.D. degrees in applied informatics from the University of Žilina, Slovakia, in 2008, 2010, and 2013, respectively.

Since 2013, he has been an Assistant Professor with the Department of Software Technologies, Faculty of Management Science and Informatics, University of Žilina. He is teaching subjects focused on object-oriented programming in bachelor's and master's degree studies, such as informatics for beginning programmers, C# language and.NET, and advanced object technologies. He has participated in several research projects with international participation, such as Interreg PL-SK, SK-CZ, University of Science Park, and the Centre of Excellence for Intelligent Transportation Systems. He is the author of more than 20 articles. His research interests include intelligent transport systems, VANET, image recognition, natural language processing, the Internet of Things, and anti-plagiarism.