© The Author 2012. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oup.com
Advance Access publication on 13 March 2012
doi:10.1093/comjnl/bxs018

A Source Code Similarity System for Plagiarism Detection

Zoran Đurić[1,*] and Dragan Gašević[2]

[1]Faculty of Electrical Engineering, University of Banjaluka, Patre 5, 78 000 Banjaluka, Bosnia and Herzegovina
[2]School of Computing and Information Systems, Athabasca University, Athabasca, Canada
*Corresponding author: zoran.djuric@etfbl.net

Source code plagiarism is an easy to do task, but very difficult to detect without proper tool support. Various source code similarity detection systems have been developed to help detect source code plagiarism. Those systems need to recognize a number of lexical and structural source code modifications. For example, by some structural modifications (e.g. modification of control structures, modification of data structures or structural redesign of source code) the source code can be changed in such a way that it almost looks genuine. Most of the existing source code similarity detection systems can be confused when these structural modifications have been applied to the original source code. To be considered effective, a source code similarity detection system must address these issues. To address them, we designed and developed the source code similarity system for plagiarism detection. To demonstrate that the proposed system has the desired effectiveness, we performed a well-known conformism test. The proposed system showed promising results as compared with the JPlag system in detecting source code similarity when various lexical or structural modifications are applied to plagiarized code. As a confirmation of these results, an independent samples *t*-test revealed that there was a statistically significant difference between average values of *F*-measures for the test sets that we used and for the experiments that we have done in the practically usable range of cut-off threshold values of 35–70%.

Keywords: algorithms; plagiarism; similarity detection; software; source code

Received 1 September 2011; revised 7 February 2012
Handling editor: Mariangiola Dezani-Ciancaglini

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

1. INTRODUCTION

Plagiarism is an act of unethical behavior, which involves reuse of someone else's work without explicitly acknowledging the original author [1–3]. A good example of plagiarism is source code plagiarism in programming languages courses, when a student submits a course assignment whose part was copied from another student's course assignment. In such a case, the plagiarized source code has been derived from another piece of source code, with a small number of routine transformations [4, 5]. Plagiarism in university course assignments is an increasingly common problem [3]. Several surveys showed that a high percentage of students have engaged in some form of academic dishonesty, particularly plagiarism [6].

Source code plagiarism is an easy to do task, but it is not so easy to detect. Usually, when students are solving the same problem by using the same programming language, there is a high possibility that their assignment solutions will be more or less similar. Several strategies of source code modification exist that can be used to mask source code plagiarism [5, 7, 8]. Examples of such strategies are renaming identifiers or combining several segments copied from different source code files. Other types of such strategies along with their classification are presented in Section 2. These modifications increase the difficulty in recognizing plagiarism. Tracking source code plagiarism is a time-consuming task for instructors, since it requires comparing each pair of source code files containing hundreds or even thousands of lines of code (LOC). Section 3 gives an overview of some of the existing approaches and discusses their advantages and disadvantages. Most of the existing source code similarity detection systems can become confused when some structural modifications have been applied to the original source code. Besides structural redesign of source code, probably the best techniques for source code plagiarism hiding are modification of data structures and redundancy [5]. On the basis of advantages and disadvantages of existing systems, the objectives for the system reported on this paper—source code similarity detector system (SCSDS)—are created and presented in Section 4. To have an efficient source code similarity detection, the major requirement for SCSDS is an enhanced and efficient tokenization algorithm, and support for multiple algorithms for similarity detection that will operate on the produced token sets. Also, an important requirement for SCSDS is to support algorithm extensibility and exclusion of template code. Section 5 describes the approach that we used for addressing these requirements for source code similarity detection set for SCSDS. This underlying approach of SCSDS consists of the following five phases: pre-processing, tokenization, exclusion, similarity measurement and final similarity calculation. The SCSDS implementation is described in detail in Section 6. Evaluation of the source code similarity detection systems is always challenging. For evaluation of SCSDS, we performed a well-known conformism test [9, 10] and compared its performance to a state-of-the-art system for plagiarism detection. Before concluding the paper, the evaluation process and obtained results are described in detail and presented in Section 7.

2. SOURCE CODE MODIFICATIONS

There have been several attempts to classify the source code modifications that a plagiarizer can do to hide plagiarism. According to [2, 11–14], source code modifications can be classified into two main categories: lexical modifications and structural modifications. Lexical modifications usually do not require any programming knowledge and skills. In most cases, this kind of modifications can easily be detected and ignored. Typical examples of lexical modifications are:

(1) modification of source code formatting (L1),
(2) addition, modification or deletion of comments (L2),
(3) language translations (L3),
(4) reformatting or modification of program output (L4),
(5) renaming of identifiers (L5),
(6) split or merge of variable declarations (L6),
(7) addition, modification or deletion of modifiers (L7) and
(8) modification of constant values (L8).

Modification of source code formatting (L1) includes cases such as insertion or deletion of empty lines, insertion or deletion of extra line breaks, or insertion or deletion of spaces and tabs. Addition, modification or deletion of comments (L2) is very often seen in plagiarized programs. Translations of the text contained in I/O statements or graphical user interface (GUI) components, for example, from English to German (or any other language), are kinds of language translations (L3). Reformatting or modification of program output (L4) includes superficial formatting such as replacing text inside literal strings, addition of characters and extra string concatenations. This modification type usually requires a certain extent of understanding of the program being modified. Thus, educationally, reformatting or modification of program output still involves some level of learning of studied programming languages. Renaming of identifiers (L5) includes changing names of classes, methods and variables. Several identifiers may be combined in the same variable declaration if the variables are of the same type. A declaration can be split into separate declarations, and vice versa (L6). Modifiers that can be added, modified or deleted are usually access modifiers, such as private, protected and public in the Java programming language (L7). For example, modifiers can be changed in such a way that more general accessibility is allowed, without affecting the semantics of a program. Addition, modification or deletion of modifiers requires some programming knowledge and skills, because some changes of modifiers can lead to compile errors, unpredictable behavior or run-time exceptions. Modifications of constant values (L8) include insignificant modifications of constants that will not affect the program output or GUI. An example of this modification is a change of the port number that the server socket is listening on.

Unlike lexical modifications, structural modifications require a higher level of programming knowledge and skills. This kind of modifications makes detection very difficult. Typical examples of structural modifications are:

(1) changing the order of variables in statements (S1),
(2) changing the order of statements within code blocks (S2),
(3) reordering of code blocks (S3),
(4) addition of redundant statements or variables (S4),
(5) modification of control structures (S5),
(6) changing of data types and modification of data structures (S6),
(7) method inlining and method refactoring (S7),
(8) redundancy (S8),
(9) temporary variables and subexpressions (S9),
(10) structural redesign of source code (S10) and
(11) modification of scope (S11).

Changing the order of variables in statements (S1) includes reordering of variables in statements in such a way that the semantics of a program is preserved. Changing the order of statements within code blocks (S2) includes reordering of statements in such a way that the semantics of a program is preserved. Reordering of code blocks (S3) includes reordering of a sequence of statements, local class declarations and local variable declaration statements within curly braces. Addition of redundant statements or variables (S4) includes kinds of additions that will not affect the output of the program. Modification of control structures (S5) covers replacement of control structures with the equivalent ones. One switch statement can be replaced by multiple if statements, for example. Data types changing (S6) is a kind of modification that is very often used by software plagiarists. For example,

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

primitive Java data types like *int* and *long*, can be changed by corresponding wrapper classes *Integer* and *Long*, respectively. Also, in most situations, numeric data types can all be replaced by some other without an essential effect on the behavior of a program. Method inlining (S7) replaces a call to a method with the body of the method[1]. Method refactoring (S7) is the reverse process of method inlining. Redundancy (S8) is the process of addition or removal of methods that are never called, or variables that are never used. The main difference with (S4) is that additions of (S8) are never executed, while additions of (S4) are executed but without any impact on the final output. Temporary variables and subexpressions (S9) are often used to mask source code plagiarism. For example, subexpressions can be translated into declaration of additional temporary variables, or default values of local variables can be set explicitly. Structural redesign of the source code (S10) is, probably, one of the most successful techniques used to hide source code plagiarism. For example, this technique can include moving of a statechanging method into a separate new class. Similar examples are refactorings supported by an Eclipse integrated development environment and similar tools [8], such as extraction of class members into a (probably newly created) parent class, string externalization, generation of additional methods (e.g. getter and setter methods, hashCode, equals and toString in Java) and generation of delegate methods. Modification of scope (S11) can include moving try-catch-finally blocks toward the outer perimeter of methods, or the scope of temporary variables can be extended by declaring them in surrounding blocks.

The success of the techniques for source code plagiarism hiding strongly depends on the weaknesses of tools used for source code similarity detection. Still, it is important to mention that, besides structural redesign of source code, modification of data structures and redundancy are probably among the most effective techniques for source code plagiarism hiding [5].

3. RELATED WORK

There are three main categories of plagiarism-detection approaches: text-based, attribute-oriented code-based and structure-oriented code-based [14, 15]. These approaches are used by source code similarity detection tools that can be divided into two categories [10]: offline and online. Online source code similarity detection tools can check a document for fragments that can also be found through web search engines, while offline source code similarity detection tools operate on documents inside a given collection. Some offline similarity detection tools can also check for historical data saved in a local database.

Two families of methods for text-based plagiarism detection exist: ranking [16] and fingerprinting [17]. Ranking is based on information retrieval concepts, where indexed collection items are ranked by an estimated similarity to a query [15]. In fingerprinting, documents are summarized as a compact collection of integers (fingerprints) representing key aspects of a document [17]. It is important to mention that text-based plagiarism-detection tools ignore coding syntax, and simply compare English words from the provided source code. This is an important drawback, since it is the coding syntax that should be used for similarity detection. The coding syntax should be used for similarity detection because it is a dominant part of every program. Empirical research also shows that text-based plagiarism-detection tools can find only small amounts of plagiarized programs to be similar, particularly method names, comments and output statements [15].

Attribute-oriented plagiarism-detection systems measure key properties of the source code. Some of the first attribute-oriented systems presented by Donaldson *et al.* [18] and Halstead [19, 20], identified only four key properties: the number of unique operators, the number of unique operands, the total number of operators and the total number of operands. Apart from these, other properties can be considered, such as: the number of variables, the number of methods, the number of loops, the number of conditional statements and the number of method invocations. A number of tools that use an attribute-oriented similarity detection approach [12, 21, 22] have so far been developed. It is important to note that two programs with, for example, the same number of variables, methods, loops and/or number of conditional statements might be marked as suspected of plagiarism, if an attribute-oriented similarity detection approach is used. Obviously, this does not necessarily mean that these programs are plagiarized. This can only suggest that they should be further inspected. Even though this approach can be considered to be better than the text-based one, it does not yet take into account the structure of a program. This makes attribute-oriented plagiarism-detection systems highly limited [15]. As stated by Verco and Wise [23], these systems are most effective when very close copies are involved. It is also difficult for these systems to detect copied segments of code in larger programs [15]. Thus, it can be concluded that the attribute-oriented approach is not the best one for plagiarism detection.

One of the approaches suitable for source code similarity detection is the structure-oriented code-based approach, which mostly use tokenization and string-matching algorithms to measure source code similarity [24, 25]. Easily modified elements, such as comments, white-space and variable names, are ignored by systems using this approach.

Structure-oriented systems are relatively robust to various source code modifications, because they perform local comparisons [15]. As stated by Burrows *et al.* [15], some of the most important and most cited state-of-the-art structure-oriented plagiarism-detection systems are JPlag [5] and Measure Of Software Similarity (MOSS) [26]. There are also other systems, such as Sim [27], Plague [28], Yet Another Plague (YAP) [25], Plaggie [29], Fast Plagiarism-Detection System (FPDS) [9] and Marble [8].

[1]Of course, such modification may violate some of the key software quality principles, too.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

JPlag is a system that can be used to detect the similarity of source code written in Java, C, C++, C# and Scheme [5]. Also, JPlag can be used to detect the similarity of plain text files. It is available in the form of a web service, while the client is implemented as a Java Web Start application [30]. For each source code file under comparison JPlag generates the corresponding token string. After that, these token strings are compared with each other using a greedy string tiling (GST) algorithm [31]. The worst case complexity for this algorithm is $O(n^3)$ as stated by Prechelt *et al.* [5]. In the best case, when two completely different strings are compared, the complexity is reduced to $O(n^2)$. The Karp–Rabin algorithm is used to further improve efficiency [15, 32]. Using these improvements, the complexity becomes almost linear—$O(n^{1.12})$. JPlag offers a simple, efficient and very intuitive user interface. Detection results are shown on the main HTML result page. This page contains a list of similar file pairs, as well as their similarity percentage. JPlag offers a side-by-side view of the selected source code file pairs, where similar source code segments from both files are marked with the same color.

MOSS is an automatic system for determining the similarity of Java, C, C++, Scheme and programs written in other languages [26]. It is available in the form of a web service. This web service can be accessed using the 'mossnet' script that can be obtained from the MOSS website[2]. MOSS can automatically eliminate matches to legitimately shared source code. The exact details of the algorithm used by this system are not publicly available, in order to ensure that it will not be circumvented. Still, it is known that MOSS uses the document fingerprinting technique (i.e. it uses a sequence of characters to form a unique document fingerprint) and the Winnowing algorithm [26].

Sim is a system for detection of similarities between programs by evaluating their correctness, style and uniqueness [6, 27]. Sim can analyze programs written in Java, C, Pascal, Modula-2, Lisp or Miranda and can also process plain text files. It is important to mention that this plagiarism-detection system is no longer actively supported, but its source code is still publicly available. It is known that similarity metric used by this system may be greatly plagued due to the incapability of its dynamic programming technique to deal with rearranged source code segments [33, 34].

Plague is one of the first structure-based source code similarity detection tools [28]. Plague creates structure profiles of each source code file under comparison [2]. These structure profiles are compared using the Heckel algorithm [35]. Plague works with code written in the C programming language. This tool has some known portability issues because it uses several UNIX utilities.

YAP, as its name states, was developed based on Plague, with some enhancements [24]. Like Plague, YAP uses a mixture of UNIX utilities, tied together with a UNIX shell script. Sdiff is the basic utility YAP is based on. YAP was improved into YAP 2, and the latest version of this tool is called YAP 3 [25]. While YAP and YAP 2 use the longest common subsequence and Heckel as the comparison algorithms, respectively, YAP 3 [25] uses a much more reliable algorithm—running Karp–Rabin-GST (RKR-GST) [7].

Plaggie is a newer open source content comparison-based similarity detection system for determining the similarity of Java programs [29]. This system is similar to JPlag, but employs the RKR-GST algorithm without any speed optimization algorithm. Unlike JPlag, Plaggie is a standalone tool.

FPDS is a source code similarity detection tool developed with the aim to improve the speed of similarity detection by using an indexed data structure to store files [9, 36]. The tokenenized versions of source code files are compared using an algorithm similar to the RKR-GST algorithm. One of the problems with the FDPS tool is that results cannot be visualized by means of displaying similar segments of code [37].

Marble is a simple, easily maintainable, structure-oriented, plagiarism-detection tool for Java and C# programs [8]. This tool splits the submission of source code files into separate files in such a way that each file contains only one top-level class. After that, details that can be easily changed are removed from these files. In this procedure, keywords (like class) and frequently used class (like String and System) and method names (like toString) are preserved, while comments, excessive white-space, string constants and import declarations are simply removed. Other tokens are abstracted to their token 'type'. Finally, Marble creates two normalized versions of each file: one in which the properties, methods and inner classes are grouped together and sorted, and the other one in which the orders of fields, methods and inner classes are exactly like those of the original file. Sorting is performed in a heuristic fashion. For example, methods are first ordered based on the number of tokens, then by the total length of the token stream and finally alphabetically [8]. The comparison of the normalized files is done by using the Unix/Linux utility *diff*.

A qualitative feature and efficiency comparisons of plagiarism-detection tools are done by Kleiman and Kowaltowski [7], Hage *et al.* [8], Lancaster and Culwin [38] and Cosma [39]. These comparisons indicate that no tool significantly outperforms the others. Besides the fact that they are among the most important state-of-the-art structure-oriented plagiarism-detection systems, the available literature indicates that JPlag and MOSS are a common choice for comparisons and validation of other plagiarism-detection tools. Some of the described systems (Sim, Plague, YAP, Plaggie and FPDS) have obvious disadvantages when compared with JPlag and MOSS. Also, some of the described systems have obvious similarities with JPlag or MOSS (YAP 3, Plaggie and FPDS). All these are the reasons why comparison of features of only JPlag and MOSS is given in Table 1.

Both systems support multiple programming languages, while the other systems introduced in this section support fewer languages than JPlag and MOSS. For example, Plaggie supports

[2]http://theory.stanford.edu/~aiken/moss/.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

TABLE 1. Comparison of features—JPlag and MOSS.

Feature	JPlag	MOSS
Number of supported languages	6	23
Presentation of results[a]	5	4
Usability[a]	5	4
Exclusion of template code	Yes	Yes
Exclusion of small files	Yes	Yes
Historical comparison	No	Yes
Submission/file-based rating	subm.	subm.
Local or remote	Remote	Remote
Open source	No	No
Similarity algorithm	GST	Winnowing
Language extensibility	No	Yes[b]
Algorithm extensibility	No	No
Platform independent	Yes	No

[a]On scale 1–5, where higher grades mean better results.
[b]MOSS language extensibility can be provided by the authors of MOSS [9].

only the Java programming language. Both systems are marked with high grades (on a scale of 1–5) for the presentation of the results and system usability [8]. On the other hand, the results of Sim are saved in a text file, while its command line interface is fairly usable. Similarly, system usability and presentation of the results of the other systems introduced in this section is also inferior when compared with JPlag and MOSS [8].

The template code in JPlag and MOSS, as well as in some other tools (e.g. Plaggie), can be excluded by providing a file containing the template code. In contrast, the template code in Sim and Marble cannot be excluded. JPlag and MOSS allow for excluding submissions from the results below certain similarity value. Plaggie and Sim do not support this. One of the drawbacks of the JPlag system is that the source code files from previous assignment submissions cannot be compared with the files from new assignment submissions. On the other hand, MOSS and Sim do support this. In JPlag and MOSS, all files within a selected directory will be considered as belonging to the same submission, and ratings are by submission. Both systems are available in the form of a Web service, while this is not the case with the other systems. JPlag uses the tokenization technique and the GST algorithm, while MOSS uses the fingerprinting technique and the Winnowing algorithm. JPlag does not support any language extensibility, while language extensibility of MOSS can be provided by the authors of MOSS [8]. These two systems, like other described systems, are not designed to be algorithm expandable. In addition, JPlag is platform independent, while MOSS is not.

Regarding the efficiency of the plagiarism-detection tools, JPlag outperforms the rest of the plagiarism-detection tools by offering a higher robustness against various modifications of plagiarized source code [5, 7, 8, 15]. The presented results

```
public class A {
  public static void main(String[] s) {
      System.out.println("test 1");
      System.out.println("test 2");
      System.out.println("test 3");
  }
}
```
(A)

```
Begin_Class
Var_Def, Begin_Method
Apply
Apply
Apply
End_Method
End_Class
```
(B)

FIGURE 1. JPlag—example Java source code (**A**) and corresponding tokens (**B**).

were one of the main reasons for using JPlag in the process of evaluation of SCSDS. However, as shown in Appendix, some drawbacks that JPlag has are reported on. Most of those drawbacks are consequence of the way in which JPlag converts source code into token strings. JPlags' tokenization process creates too abstract token representation of the compared source code. This is the main reason why modifications like reordering of statements within code blocks, reordering of code blocks, inlining or code insertions cannot be adequately detected using the GST algorithm [13].

On the other hand, in some cases, JPlags' tokenization process generates unnecessary tokens. A simple Java program given in Fig. 1 JPlag is represented with eight corresponding tokens. If 20 or 30 unnecessary import statements are inserted in the plagiarized version of this program, the plagiarized program will be represented with 28 or 38 tokens, respectively. Thus, the average similarity between these, basically identical, programs will drop to only 44.4 or 34.7%, respectively.

Similarly, a simple insertion of a *System.out.println()* statement between every source code line (in a plagiarized program) will dramatically reduce the average similarity between the original and the plagiarized program. Again, the reason for this is the way tokenization is done in JPlag. Every *System.out.println()* statement will be represented with only one token, and every one of these tokens will split the original token set into smaller parts. If the number of tokens in these split parts is less than the minimum match length that the RKR-GST algorithm uses, then the average similarity between the original program and the plagiarized one will be very small.

Although the authors of [5] state that JPlag puts semantic information into tokens where possible, in order to reduce spurious substring matches, there are situations where this is not the case. For example, expressions like the following two: $String\ s = `s'$ and $inti = 1$, will be represented by the same two tokens *Var_Def* and *Assign*, respectively, and thus they will be treated as the same. The reason why those two expressions are represented by the same two tokens is JPlag's too abstract token representation of the compared source code.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

4. SCSDS OBJECTIVES

On the basis of the discussed advantages and disadvantages of existing solutions for source code similarity detection (primarily advantages and disadvantages of the JPlag system), the objectives for SCSDS are created. The most important objective is an efficient source code similarity detection. In order to be highly robust to lexical and structural modifications described in Section 2, as well as to address the challenges discussed in Section 3 (i.e. (C1)–(C9) challenges), the system must be able to efficiently detect and point out similarities between source-code files. To accomplish this, the system must implement a detailed and efficient tokenization algorithm, and support multiple algorithms for similarity detection that will operate on a produced token set. This token set must characterize the essence of a program in a way that is difficult to change by a plagiarist. Additionally, tokens contained in this token set should contain as much semantic information as possible. The importance of a tokenization algorithm can easily be illustrated through the fact that the efficiency of two plagiarism-detection tools, which use the same similarity detection algorithm (e.g. RKR-GST), completely depends on a token set produced by their tokenization algorithms.

In addition, the proposed system must support:

(1) algorithm extensibility—the system must be easy to extend with new algorithms for similarity measurement and
(2) exclusion of template code—the system must ignore legitimately shared source code during the similarity detection process.

5. OUR APPROACH

Similarly, as per the reported designs commonly found in other systems [5, 14, 15, 40], our similarity detection approach comprises the following five phases (Fig. 2):

(1) pre-processing,
(2) tokenization,
(3) exclusion,
(4) similarity measurement and
(5) final similarity calculation.

In phases (1) and (2), we aim to reduce the 'noise' in a given set of source code files. These phases are very important in the process of source code similarity detection. The quality of similarity detection can be significantly improved with more elaborate pre-processing and tokenization techniques [7]. It is also important to mention that *these two phases are programming language dependent*. Legitimately shared source code should not be considered during the similarity measurement phase. This is why the exclusion of such a code is done in phase (3). Phase (4) can be repeated several times. The number of repetitions is defined by the number

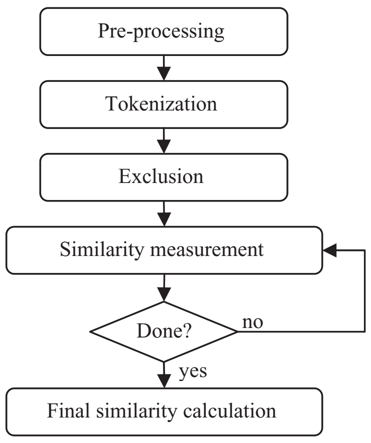


FIGURE 2. SCSDS similarity detection approach.

of implemented similarity measurement algorithms. Currently, there are two similarity measurement algorithms implemented in SCSDS. These algorithms will be discussed later. The final similarity calculation (phase (5)) is done on the results obtained from phase (4). The exclusion, similarity measurement and final similarity calculation phases (i.e. phases (3)–(5)) are *programming language independent*.

Pre-processing (*phase* (1)). The first phase of source code similarity detection makes the detection process robust to the following simple source code transformations: addition, modification or deletion of comments (L2), split or merge of variable declarations (L6), changing the order of variables (S1), as well as addition of some redundant statements (S4).

For this reason, all comments are removed from the original source code. This process includes removing block comments, single-line comments, trailing comments and the end-of-line comments. Combined variable declarations are split into a sequence of individual declarations. The fully qualified names are replaced by simple names, while package declarations and import statements are removed from the original source code. Also, in this phase, variables in statements are grouped by type. An example is given in Fig. 3. As can be seen, after the pre-processing phase, all comments have been removed from the source code; fully qualified class names have been replaced by their corresponding simple names; combined variable declarations have been split into two individual declarations; while variables in statements are grouped by type.

Tokenization (*phase* (2)). Tokenization [11] is the process of converting the source code into tokens. As described in Section 3, this technique is very popular and used by many source code plagiarism-detection systems. The tokens are chosen in such a way that they characterize the essence of a program, which is difficult to change by a plagiarist. For example, white-space should never produce a token. This tokenization technique is commonly used to neutralize various source code modifications: modification of source code formatting

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

```
---- original source code ----
package net.etfbl.test;
import java.lang.*;
import java.util.*;
/*
 *  block comment
 */
public class A {
  /* single-line comment */
  public static void main(String[] args) {
      java.lang.Integer i1 = 1, i2 = 2;
      java.lang.String s = "";
      Long l = 5l; /* trailing comment */
      java.lang.Integer i3 = i1 + i2;
      // end-of-line comment
  }
}

---- source code after pre-processing ----
public class A {
  public static void main(String[] args) {
      Integer i1=1;
      Integer i2=2;
      String s = "";
      Long l = 5l;
      Integer i3 = (i1 + i2);
  }
}
```

FIGURE 3. SCSDS pre-processing.

(L1), language translations (L3), reformatting or modification of program output (L4), renaming of identifiers (L5) and modification of constant values (L8), as well as to prepare an appropriate input for the comparison algorithm in the similarity measurement phase.

A simple tokenization algorithm can substitute all identifiers and values with tokens <IDENTIFIER> and <VALUE>, respectively [37]. SCSDS uses a significantly modified approach. The SCSDS tokenization algorithm substitutes identifiers with the appropriate tokens. These tokens are chosen based on the identifier type. For example, all identifiers of Java numeric types, i.e. all identifiers of byte, short, int, long, float, double along with their corresponding wrapper classes (Byte, Short, Integer, Long, Float and Double, respectively), are substituted with the <NUMERIC_TYPE> token. Also, their values are substituted with the <NUMERIC_VALUE> token. With this tokenization algorithm, four main advantages are gained. First, tokens chosen this way better characterize the essence of a program. Secondly, the possibility of false positives is significantly reduced. For example, by using the SCSDS tokenization algorithm, expressions like the following two: $string3 = string1 + string2$ (all identifiers are of type String) and $int3 = int1 + int2$ (all identifiers are of type *int*) cannot be falsely matched; the same would happen if the simple tokenization algorithm mentioned by Mozgovoy [37] were used. Also, expression $integer3 = integer1 + integer2$ (all identifiers are of type *Integer*) and expression $int3 = int1 + int2$ (all identifiers are of type *int*) will be treated as the same. Additionally, this algorithm is still simple and fast. An example is given in Fig. 4. It can be seen that all identifier types and values are represented by

```
java.lang.Integer i = 1;
java.lang.String s = "";
Long l = 5l

<NUMERIC_TYPE><IDENTIFIER> = <NUMERIC_VALUE>
<STRING_TYPE><IDENTIFIER> = <STRING_VALUE>
<NUMERIC_TYPE><IDENTIFIER> = <NUMERIC_VALUE>
```

FIGURE 4. SCSDS—example Java source code and corresponding tokens.

corresponding tokens. As previously explained, for the same source code, JPlag generates three pairs of *Var_Def* and *Assign* tokens.

Variables and methods in many programming languages can be attributed with access modifiers. In Java and C#, variables and methods can be attributed with access modifiers, like *private*, *protected* or *public*. In C#, there is one additional access modifier—*internal*. If access modifiers are changed to allow for a more general accessibility, the semantics of an existing program is usually not affected. Sometimes, it is even possible to restrict accessibility without affecting the essence of a program. This is why access modifiers should be removed. Similarly, all other modifiers can be removed without affecting the essence of a program. This is why the SCSDS tokenizer ignores them all. Additionally, the SCSDS tokenizer ignores other unnecessary data, so that the semantics of an existing program is not affected. For example, the SCSDS tokenizer ignores semi-colons, as shown in Fig. 4.

Generally, there are drawbacks related to the tokenization process—significant increase in the degree of similarity between any two source code files when compared with their non-tokenized versions. This can lead to false positives.

The impact of SCSDS tokenization process on source code similarity detection will become much clearer later in this article when the results of the experimental evaluation are reported (Section 7).

Exclusion (*phase* (3)). There are situations when the source code can be shared between programmers. For example, students can share some common base source code given by the instructors. Some similarity detection systems, like JPlag, MOSS and Plaggie, allow the user to give such legitimately shared source code (i.e. template code) that will be ignored during the similarity detection phase [8]. The danger of such template code is that the similarity between the template code may obscure the similarity between non-template code. Thus, the ignorance of the template code can help prevent many false positives.

Unlike the similarity detection systems mentioned in the previous paragraph, SCSDS uses a slightly modified approach. SCSDS creates a tokenized version of template code, in the same way as it creates a tokenized version of the source code in the tokenization phase (phase (2)). The result of this process is called 'exclusion token sequence'. In the exclusion phase, all findings of an 'exclusion token sequence' or its parts (token subsequences of an 'exclusion token sequence') are removed

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

from the input token sequence. For this purpose, SCSDS uses the RKR-GST algorithm. It is important to mention that one of the advantages of the RKR-GST algorithm is that token subsequences can be found independently of their position in the token sequence. The RKR-GST algorithm is described in detail later in this section. This way the template code is ignored during the similarity detection phase. It is important to notice that SCSDS can also ignore modified versions of the template code. The reason for this is that SCSDS does not look for exact matches of the template code or its parts in the input source code, but it does look for exact matches of the tokenized version of the template (i.e. 'exclusion token sequence') or its parts in the input token sequence. This way the given source code, which has to be completed by the students, can be ignored during the similarity detection phase.

Additionally, this approach does not have any restrictions regarding contents of documents in a corpus. For example, suppose we have a corpus consisting of a sub-corpus of documents that contain template code and a sub-corpus of documents that does not use this common base. Then, during the exclusion phase, only findings of an 'exclusion token sequence' from input token sequences that represent documents in the first sub-corpus will be removed in the exclusion phase; input token sequences that represent documents in the second sub-corpus will be intact.

Additionally, it is important to note that there is always a chance that a tokenized version of the template code accidentally occurs in some other places. Naturally, the larger the template code is, the smaller is the chance that the tokenized version of the template code accidentally occurs in other places, for some non-template code of the same size.

Some typical source code, which generates many false positives, can be identified and removed in the exclusion phase. A good example of such source code is a Java bean, a class that has a default constructor, and allows for accessing to its properties by using getter and setter methods. Comparing Java beans will always result in a high similarity. This does not indicate a plagiarism, but it is simply a result of the way such classes are implemented [8].

Similarity detection and measurement (*phase* (4)). Currently, SCSDS implements two algorithms for similarity detection: the RKR-GST algorithm [31, 32] and the Winnowing algorithm.

As previously explained, most JPlag drawbacks are a consequence of its tokenization process and not of the RKR-GST algorithm. It is shown by Kleiman and Kowaltowski [7] that RKR-GST gives better results when used on a token set created in such a way that extraneous information is removed and better normalization is achieved. Also, the tilling approach is considered to be quite advanced and reliable, and the most widely used systems implement algorithms that can be treated as tilling variations [10]. As can be seen in the Kleiman and Kowaltowski paper [7], there are few situations in which the Winnowing algorithm gives better results than the RKR-GST algorithm. These are the main reasons why the first two algorithms that SDSDS implements are RKR-GST and Winnowing.

The RKR-GST algorithm was designed by Michael Wise [31]. When comparing two strings Str1 and Str2, the aim of this algorithm is to find a set of substrings that are identical and that satisfy the following requirements [5].

(1) Any token of Str1 may only be matched with exactly one token from Str2.
(2) Substrings are to be found independent of their position in the string.
(3) Because long substring matches are more reliable, they are preferred over short ones. Short matches are more likely to be spurious.

The first rule implies that it is impossible to match parts of the source code completely that have been duplicated in a plagiarized program. The second rule implies that changing the order of statements within code blocks and reordering of code blocks will not have a big impact on the similarity detection process. In other words, changing the order of statements within code blocks and reordering of code blocks is not an effective attack, if the reordered code segments are longer than the minimal match length. If the reordered code segments are shorter than the minimal match length, then mentioned modifications can be an effective attack. This combined with the abstract token representation of JPlags is likely the main reason why JPlag scores much lower than 100% for the programs in which only methods have been moved around, as evidenced by Hage *et al.* [8].

The RKR-GST algorithm consists of two phases. In the first phase of the algorithm, two strings are searched for the longest contiguous matches. These matches are obtained by using the Karp–Rabin procedure [32], and must be longer than the minimum match length threshold value. In the second phase of the algorithm, the matches are marked, so that they cannot be used for further matches in the first phase of a subsequent iteration. This guarantees that every token will only be used in one match. After all the matches are marked, the first phase is started again. The algorithm finishes its work when the match length stays the same as the minimum match length threshold value.

In SCSDS, the RKR-GST similarity measure of token sequences a and b is calculated by using the following formula (formula 1):

$$\text{sim}(a, b) = \frac{2 * \text{coverage}}{\text{length}(a) + \text{length}(b)},$$

where coverage is the length of all matches, and length is the number of tokens in the token sequence.

SCSDS can calculate this similarity measure differently, using the following formula (formula 2):

$$\text{sim}(a, b) = \frac{2 * \text{coverage}}{\text{strength}(a, b)},$$

where coverage is the length of all matches, and slength is the smaller number of tokens from the two token sequences.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

Formula 2 will return the maximum similarity value of 1 (100% similarity) in some cases of plagiarism (e.g. when some additional code or additional methods are added to the original source code). The code added in this way will break the original source code (and the original token sequence) into two or more parts. Still, one of the advantages of the RKR-GST algorithm is that token subsequences are found independently of their position in the token sequence. This is why we use the high output from formula 2 as an indicator of a possible plagiarism caused by adding some redundant code. Despite that, for other cases of plagiarism, we prefer formula 1, which returns a similarity of 1 only when the token sequences are identical. The second reason for this is that formula 2 returns a rather high similarity value when one of the two token sequences is significantly smaller.

The Winnowing algorithm is very simple [26]. At the beginning of the algorithm, all white-spaces are removed from the input string (which can be the original source code or the tokenized code). After that, 'n-grams'[3] from the resulting string are hashed, thus creating a sequence of hashes. This set of hashes is divided into a series of overlapping windows of a certain size. For each window, the hash with the lowest value is selected as its representative. This significantly reduces the number of hash values and allows for more efficient similar substring detection [26]. Also, the hash sets are precomputed for each input string separately and kept in a memory instead of being recalculated for each pair-wise comparison. This speeds up the similarity detection and measurement phase.

Similarly to RKR-GST, the Winnowing similarity measure of fingerprint sets a and b is calculated using the following formula (formula 3):

$$\mathrm{sim}(a,b) = \frac{2 * \mathrm{setSize}(c)}{\mathrm{setSize}(a) + \mathrm{setSize}(b)},$$

where setSize is the size of a set, and set c is intersection of sets a and b.

Final similarity calculation (*phase* (5)). The fifth phase in this similarity detection process is the final similarity calculation. This calculation is based on similarity measure values obtained from the similarity detection algorithms, and their weight factors. The overall similarity measure between two source code files a and b is calculated using the following formula (formula 4):

$$\mathrm{sim}(a,b) = \sum_{i=1}^{n} \frac{w_i * \mathrm{sim}_i(a,b)}{w},$$

where sim_i is the similarity measure value obtained from the similarity detection algorithm i, w_i is the weight factor of the similarity detection algorithm i, w is the sum of weight factors of all used similarity detection algorithms and n is the number of all used similarity detection algorithms. In the current implementation of SCSDS, $n = 2$ is used.

[3]n-gram is a contiguous substring of length n

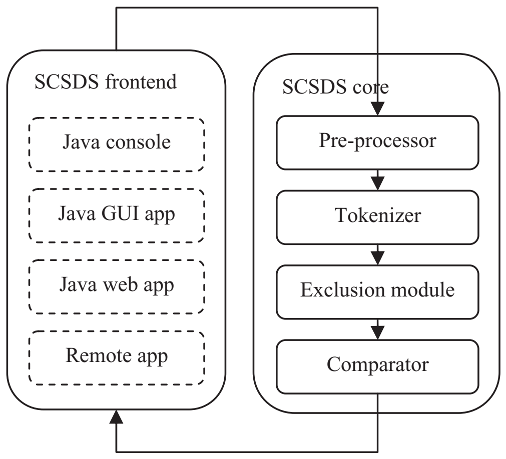


FIGURE 5. SCSDS design.

If the overall similarity measure value is larger than the threshold value specified by the user, then the corresponding pair of source code files is marked as suspected. Together with the overall similarity measure value, SCSDS will show similarity measure values of all algorithms. For the selected pair of source code files SCSDS will show suspected plagiarized source code segments.

When the RKR-GST algorithm is used on a more detailed (and not too abstract) token representation of compared source code files, better similarity results will be achieved. This is the reason why modifications like those described in the end of Section 3 do not affect the results of the SCSDS similarity detection. It is important to mention that RKR-GST is the primary algorithm that SCSDS uses, while the impact of the Winnowing algorithm can be adjusted by the corresponding weight factor.

6. DESIGN AND IMPLEMENTATION

SCSDS consists of two parts: the SCSDS core and SCSDS frontend (Fig. 5). The SCSDS core part is the main part of SCSDS and consists of the following modules: Pre-processor, Tokenizer, Exclusion module and Comparator. The SCSDS core functionalities are exposed to the frontend directly or as a SOAP-based Web service.

The pre-processor module is responsible for the pre-processing phase of the SCSDS similarity detection process. This module removes all comments from the input source code files, including block comments, single-line comments, trailing comments and end-of-line comments. Also, this module splits combined variable declarations into a sequence of individual declarations, replaces the fully qualified names by the simple names, removes import statements and changes the order of

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

variables in statements, according to the alphabetical order. The output of this module is forwarded to the tokenizer module. Currently, there is only one fully implemented pre-processor module in SCSDS—for the Java programming language, while the pre-processor modules for the C# and C++ languages are under development.

The tokenizer module is responsible for the tokenization phase of the SCSDS similarity detection process. The tokenizer module creates a list of token objects. Each token object contains the token string and the number of the line from which the token originates. This allows us to determine and show any similarities between two files with respect to the original source files later. To enable SCSDS to handle another programming language, a new tokenizer must be implemented. Because SCSDS is implemented using object-oriented design principles, this can be done without changing the whole architecture of the system. The tokenizer module forwards the resulting list of tokens to the exclusion module.

The exclusion module is responsible for the exclusion phase of the similarity detection process. This module removes an 'exclusion token sequence' (created from a legitimately shared source code) from the list of tokens obtained from the tokenization module. The resulting list of tokens is forwarded to the comparator module.

The comparator module is responsible for the similarity measurement phase of the similarity detection process. This module executes the following comparators: the RKR-GST comparator, which implements the RKR-GST algorithm, and the Winnowing comparator, which implements the Winnowing algorithm. The results of this process are similarity measures calculated by using formulas described in Section 5. Like the tokenizer module, this module can also be extended without changing the whole architecture of the system. There is only one mandatory requirement for a new comparator—it must implement the SCSDS comparator interface.

The SCSDS core is implemented in the Java programming language. By using Java, we achieved platform independency. For implementation of the pre-processing and exclusion functions, we use regular expressions, while for parsing and tokenization functions, we implemented an extension of Eclipse Java development tools.

The SCSDS frontend is responsible for selection of source code files to be compared and for displaying of similarity measurement results. The SCSDS frontend can be implemented in various ways. For example, the frontend can be implemented as a Java console application (command-line Java application), a Java GUI application, a Java Server Pages or as a Java Server Faces (JSF) application. These frontends can use the SCSDS core functionalities directly. Currently, the implemented SCSDS frontends are Java console and Java GUI frontend, while a JSF frontend is under development.

In addition, SCSDS is designed in such a way that frontends can be implemented in different languages and technologies. These frontends must be implemented as Web service clients for the SCSDS core Web service. In this way, we provided a programming language and technology independency for SCSDS frontends.

SCSDS creates a report identifying the pairs of suspicious source code files. Hence, the user of this tool can further investigate similarities for selected source code file pairs (Fig. 6).

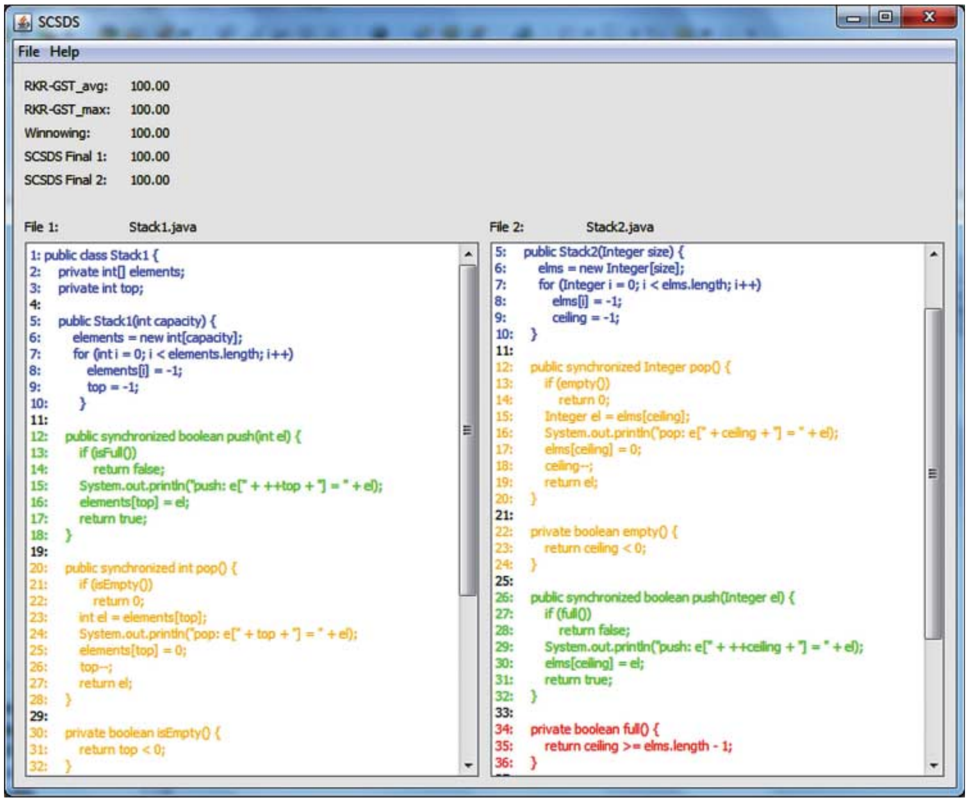
As mentioned before, SCSDS currently supports only the Java programming language. To add a support for a new programming language, some effort must be made. More precisely, a pre-processor module and toknizer should be implemented for every new language that is going to be supported by the SCSDS. To do this, one must know the syntax and semantics of the languages in question.

7. EVALUATION

For the evaluation of the SCSDS system, we performed a well-known conformism test [9, 10]. The conformism test determines how many source code files, considered to be plagiarized by some system, are found in a common 'plagiarized files subset', defined by some other system ('the jury') by processing the same test set. In this test, the JPlag system, described in Section 3, is used as 'the jury' system. The JPlag system is chosen based on the results presented by Prechelt *et al.* [5], Kleiman and Kowaltowski [7], Hage *et al.* [8], Mozgovoy [10] and Burrows *et al.* [15]. To the best of our knowledge, there is no well-known set of programs recommended for plagiarism-detection tool testing. Thus, we decided to conduct our experiments on two test sets of Java programs, either of them created similarly as in [5–7, 15, 36, 40]. Both test sets contained several different solutions of a typical programming assignment from a Java programming language course and additional plagiarized versions of these solutions.

The base of the first test set (test set A) is a collection of three different solutions to the same programming assignment. These solutions are different, non-plagiarized, implementations of a 'matrices calculator' program for multiplication, addition, subtraction, inversion and other operations on matrices. This base test set is extended with 48 additional programs. These programs are created by 11 experienced students and 4 teaching assistants (referred to as 'plagiarizers'), according to the typical lexical and structural modifications described in Section 2. The 'plagiarizers' were told to behave like students who were plagiarizing and trying to deceive a plagiarism-detection system. It is important to mention that the 'plagiarizers' did not have any prior knowledge about the SCSDS system nor had they been previously involved in any research on automated plagiarism detection. Table 2 shows the number of new plagiarized versions for each of the three original programs (denoted as 'p'), as well as the number of plagiarism pairs we obtained in this way (denoted as 'Pp'). Each typical lexical and structural modification, described in Section 2, is applied to at least three different plagiarized programs. Although there

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018



```
RKR-GST_avg:     100.00
RKR-GST_max:     100.00
Winnowing:       100.00
SCSDS Final 1:   100.00
SCSDS Final 2:   100.00
```

File 1: Stack1.java

```
public class Stack1 {
  private int[] elements;
  private int top;

  public Stack1(int capacity) {
    elements = new int[capacity];
    for (int i = 0; i < elements.length; i++)
      elements[i] = -1;
      top = -1;
    }

  public synchronized boolean push(int el) {
    if (isFull())
      return false;
    System.out.println("push: e[" + ++top + "] = " + el);
    elements[top] = el;
    return true;
  }

  public synchronized int pop() {
    if (isEmpty())
      return 0;
    int el = elements[top];
    System.out.println("pop: e[" + top + "] = " + el);
    elements[top] = 0;
    top--;
    return el;
  }

  private boolean isEmpty() {
    return top < 0;
  }
```

File 2: Stack2.java

```
  public Stack2(Integer size) {
    elms = new Integer[size];
    for (Integer i = 0; i < elms.length; i++)
      elms[i] = -1;
      ceiling = -1;
  }

  public synchronized Integer pop() {
    if (empty())
      return 0;
    Integer el = elms[ceiling];
    System.out.println("pop: e[" + ceiling + "] = " + el);
    elms[ceiling] = 0;
    ceiling--;
    return el;
  }

  private boolean empty() {
    return ceiling < 0;
  }

  public synchronized boolean push(Integer el) {
    if (full())
      return false;
    System.out.println("push: e[" + ++ceiling + "] = " + el);
    elms[ceiling] = el;
    return true;
  }

  private boolean full() {
    return ceiling >= elms.length - 1;
  }
```

FIGURE 6. SCSDS—side-by-side view of selected source code file pair.

TABLE 2. Test sets A and B.

Set	Programming assignment	*p*	Pp
A	1	28	406
	2	14	105
	3	6	21
B	1	20	210
	2	10	55
	3	8	36

are different approaches to creating test sets (e.g. 'Call for plagiarisms' described by Prechelt *et al.* [5]), we felt that this was a more appropriate way in which a reliable data set for the intended evaluation can be constructed and which is more similar to an authentic context in which plagiarism attempts might happen. Although SCSDS detected several plagiarism cases from real-world sets of student assignment and exam submissions, we decided not to use these sets. The main reason for this decision was the fact that these plagiarisms consisted of only several (mainly simple) modifications. This way created test sets would not be comprehensive enough for the intended evaluation allowing for drawing general conclusions about the evaluated system.

The second test set (test set B), based on three different non-plagiarized implementations of a 'stack' (the program for the stack structure), is created in the same way as the first test set. Besides three base programs, this test set contains additional 38 programs (Table 2) which were created by the 'plagiarizers'.

The main differences between test sets A and B are the number of plagiarized pairs, the complexity and the length of contained programs. The average length of source code files in test set A is 337 LOC and the standard deviation is 143.54. The average number of methods per class is 15, the average number of attributes per class is 7 and the average number of variables per method is 5. The longest program in test set A has 630 LOC. Most of the methods have a cyclomatic complexity between 5 and 7, while there are several classes with methods having cyclomatic complexity equal to or greater than 10, but no more than 16. The average length of source code files in test set B is 69 LOC and the standard deviation is 25.09. The average number of methods per class is five, the average number of attributes per class is four and the average number of variables per method is three. All methods in the programs contained in test set B have a cyclomatic complexity of four or less.

Overall, test set A consists of 51 programs; that is, there were 1275 (51 × 50/2) unique pairs of programs in test set A, with 532 plagiarism pairs. Test set B consists of 41 programs, that

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

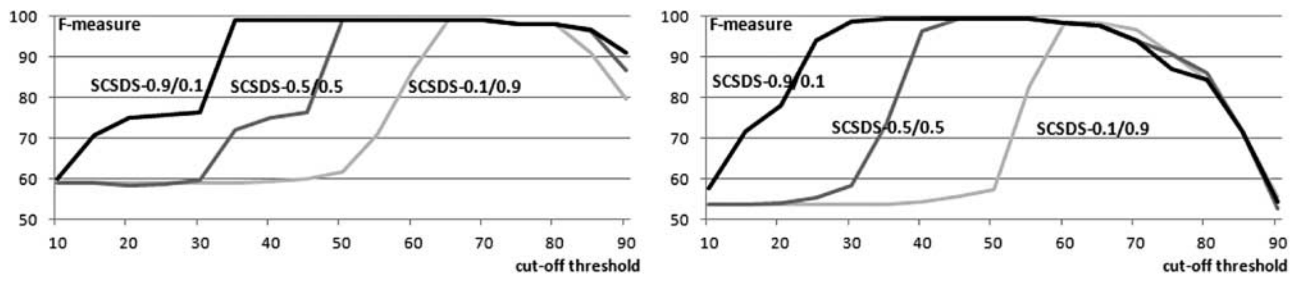


FIGURE 7. SCSDSs' F-measure values for program pairs in test set A (left) and F-measure values for program pairs in test set B (right).

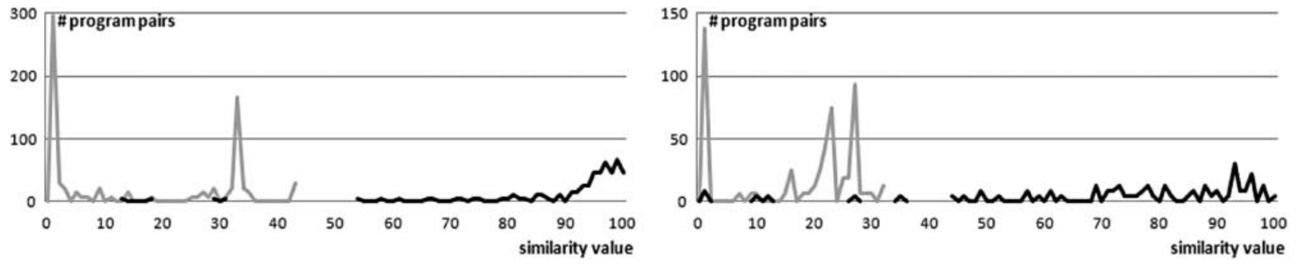


FIGURE 8. JPlags' similarity values among program pairs in test set A (left) and similarity values among program pairs in test set B (right).

is, there were 820 (41 × 40/2) unique pairs of programs, with 301 plagiarism pairs.

The evaluation was conducted in two phases. In the first phase, the aim was to compare the effectiveness of JPlag and SCSDS. In the second phase, the aim was to compare the results of the JPlag and SCSDS similarity detection conducted on both test sets. After the fully automated plagiarism detection within the test sets has been done, we checked the results by a careful manual comparison of each program pair.

To compare the effectiveness of JPlag and SCSDS, we employ the precision and recall measures [5, 15, 40]. The precision (P) measure is the percentage of program pairs identified as plagiarized with respect to the total number of suspected program pairs. The recall (R) measure is the percentage of program pairs identified as plagiarized with respect to the total number of plagiarized program pairs. We also aggregated the precision and recall values, by using the standard F-measure computed as

$$F = 2 * \frac{P * R}{P + R}.$$

During the first phase of the evaluation, we used 18 different minimum match length threshold values (from 3 to 20) for the RKR-GST algorithm implemented in SCSDS. Our experiments revealed that the minimum match length threshold value of 12 gives the best performance. On the basis of results reported by Burrows *et al*. [15], we decided to use 4 g and windows of size 4 for the Winnowing algorithm implemented in SCSDS. The final similarity measure value for SCSDS was calculated by using formula 4 (given in Section 5). We calculated all the SCSDS final similarity values along with precision, recall and F-measure values [for different cut-off threshold (CT) values] for all the weight-algorithm combinations, from 0.9 for RKR-GST and 0.1 for the Winnowing algorithm, to 0.1 for RKR-GST and 0.9 for the Winnowing algorithm. Our experiments revealed that the best results are obtained for the 0.9 and 0.1 weight-algorithm combination the RKR-GST and Winnowing algorithms.

As can be seen in Fig. 7, for the 0.9 and 0.1 weight factors of the RKR-GST and Winnowing algorithms, respectively, SCSDS has the maximum F-measure value (denoted as SCSDS-0.9/0.1) for CT values between 35 and 70% for test set A, and for CT values between 35 and 55% for test set B. For the other weight-algorithm combinations, SCSDS has the maximum F-measure value for a narrower range of the CT values.

Figures 8 and 9 show the distribution of similarity values (X-axis) found among plagiarism pairs (black line) and among non-plagiarism pairs (gray line) of programs in both test sets, returned by JPlag and SCSDS, respectively.

As can be seen, JPlag's behavior is not perfect for both test sets. There are some overlaps of similarity values among program pairs in test set A and similarity values among program pairs in test set B. For 27 of the total 532 plagiarism pairs in test set A, JPlag returned similarity values of 31% or less. The minimum similarity value of the other plagiarism pairs was 54%, while the maximum similarity value of non-plagiarism pairs was 43%. JPlag's behavior for the second set is even worse. For 26 of a total of 301 plagiarism pairs in test set B JPlag, returned similarity values of 35% or less, while the minimum similarity value of the other plagiarism pairs was 44%. The maximum similarity value of non-plagiarism pairs in test set B was 32%.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

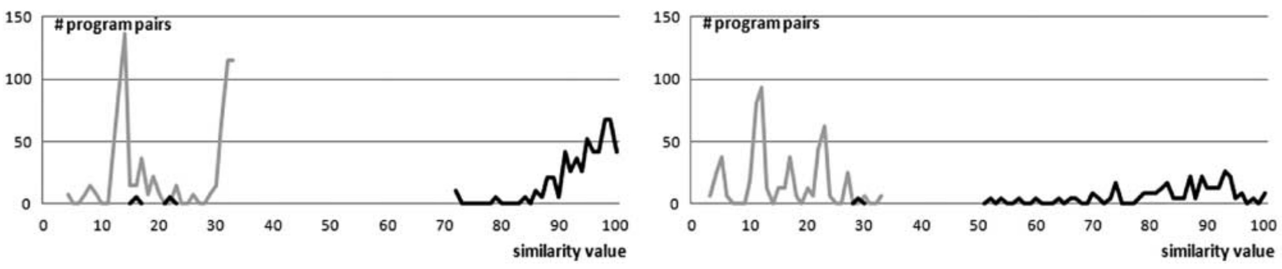


FIGURE 9. SCSDSs' similarity values among program pairs in test set A (left) and similarity values among program pairs in test set B (right).

TABLE 3. JPlags' and SCSDSs' precision and recall values among program pairs in test set A (left) and similarity values among program pairs in test set B (right) for CT values from 10 to 90.

	JPlag—test set A			JPlag—test set B			SCSDS—test set A			SCSDS—test set B		
CT	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*
90	100.00	74.62	85.47	100.00	32.89	49.50	100.00	83.46	90.98	100.00	37.21	54.24
85	100.00	81.58	89.86	100.00	42.86	60.00	100.00	93.98	96.90	100.00	55.81	71.64
80	100.00	86.28	92.63	100.00	48.50	65.32	100.00	95.11	97.49	100.00	73.09	84.45
75	100.00	88.34	93.81	100.00	59.80	74.84	100.00	96.24	98.08	100.00	77.08	87.06
70	100.00	91.16	95.38	100.00	71.43	83.33	100.00	98.12	99.05	100.00	88.70	94.01
65	100.00	93.04	96.39	100.00	75.75	86.20	100.00	98.12	99.05	100.00	91.36	95.48
60	100.00	93.98	96.90	100.00	79.73	88.72	100.00	98.12	99.05	100.00	94.35	97.09
55	100.00	94.92	97.39	100.00	84.05	91.33	100.00	98.12	99.05	100.00	95.68	97.79
50	100.00	95.86	97.89	100.00	85.71	92.31	100.00	98.12	99.05	100.00	98.67	99.33
45	100.00	95.86	97.89	100.00	89.70	94.57	100.00	98.12	99.05	100.00	98.67	99.33
40	94.44	95.86	95.14	100.00	91.36	95.48	100.00	98.12	99.05	100.00	98.67	99.33
35	91.89	95.86	93.83	100.00	92.69	96.21	100.00	98.12	99.05	100.00	98.67	99.33
30	66.45	96.80	78.80	93.62	92.69	93.15	66.74	98.12	79.44	97.41	100.00	98.69
25	61.90	97.74	75.80	63.68	94.35	76.04	62.70	98.12	76.51	89.05	100.00	94.21
20	61.90	97.74	75.80	47.18	94.35	62.90	62.14	98.12	76.09	64.04	100.00	78.08
15	60.80	98.87	75.30	43.96	94.35	59.98	61.73	100.00	76.34	55.84	100.00	71.66
10	60.59	100.00	75.46	43.48	97.67	60.17	42.94	100.00	60.08	40.40	100.00	57.55

The distribution of similarity values found among plagiarism pairs and among non-plagiarism pairs returned by SCSDS is better than those of JPlag. For 10 out of the total 532 plagiarism pairs in test set A, SCSDS returned similarity values of 22% or less, while the similarity value of the other plagiarism pairs was not <72%. This obviously demonstrates a higher probability to recognize similarity between the authentic versions of the programs and their plagiarized versions. The maximum similarity value of non-plagiarism pairs in test set A, returned by SCSDS, was 33%. For only 4 out of total 301 plagiarism pairs in test set B, SCSDS returned similarity values of 29% or less, while the minimum similarity value of the other plagiarism pairs in this test set was 52%. The maximum similarity value of non-plagiarism pairs in test set B, returned by SCSDS, was 33%.

The detailed results are given in Table 3. There are values for precision (denoted as P), recall (denoted as R) and F-measure (denoted as F) for different CT values. The CT value separates plagiarized program pairs from non-plagiarized ones. If the similarity value of two programs is larger than CT value, then the program pair is marked as suspect. We used CT values from 10 to 90%.

It is important to note that, for both test sets, SCSDS has a higher recall and higher or the same precision as for every CT value, except for the CT value of 10%. As can be seen, SCSDS has recall higher than 90 and the maximum precision for CT values between 35 and 85% for test set A, and for CT values between 35 and 65% for test set B. In addition, SCSDS has the maximum recall for CT values lower than or equal to 15%, for test set A, and for CT values lower than or equal to 30%, for test set B. At the same time, JPlag has the maximum recall for CT values lower than or equal to 10%.

After the analysis of the results, we noted that both JPlag and SCSDS were more robust to some lexical and

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

TABLE 4. JPlags' and SCSDSs' precision and recall values among program pairs in modified test set A (left) and similarity values among program pairs in modified test set B (right) for CT values from 10 to 90.

	JPlag—test set A			JPlag—test set B			SCSDS—test set A			SCSDS—test set B		
CT	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*
90	100.00	76.93	86.96	100.00	34.37	51.16	100.00	86.04	92.50	100.00	38.88	55.99
85	100.00	84.10	91.36	100.00	44.79	61.87	100.00	96.89	98.42	100.00	58.33	73.68
80	100.00	87.98	93.61	100.00	50.69	67.28	100.00	98.06	99.02	100.00	76.38	86.61
75	100.00	89.92	94.69	100.00	62.50	76.92	100.00	98.06	99.02	100.00	80.55	89.23
70	100.00	93.02	96.38	100.00	74.65	85.49	100.00	100.00	100.00	100.00	92.70	96.21
65	100.00	94.96	97.41	100.00	79.16	88.37	100.00	100.00	100.00	100.00	94.09	96.96
60	100.00	95.93	97.92	100.00	81.94	90.07	100.00	100.00	100.00	100.00	97.22	98.59
55	100.00	96.89	98.42	100.00	86.45	92.73	100.00	100.00	100.00	100.00	100.00	100.00
50	100.00	97.86	98.92	100.00	86.45	92.73	100.00	100.00	100.00	100.00	100.00	100.00
45	100.00	97.86	98.92	100.00	90.97	95.27	100.00	100.00	100.00	100.00	100.00	100.00
40	94.39	97.86	96.09	100.00	92.70	96.21	100.00	100.00	100.00	100.00	100.00	100.00
35	91.98	97.86	94.83	100.00	94.09	96.96	100.00	100.00	100.00	100.00	100.00	100.00
30	66.14	98.83	79.25	93.44	94.09	93.76	62.46	100.00	76.89	95.68	100.00	97.79
25	61.87	100.00	76.44	62.58	94.09	75.17	61.35	100.00	76.05	88.34	100.00	93.81
20	61.87	100.00	76.44	45.93	94.09	61.73	59.86	100.00	74.89	63.01	100.00	77.31
15	61.35	100.00	76.05	42.74	94.09	58.78	53.97	100.00	70.10	54.75	100.00	70.76
10	59.86	100.00	74.89	43.20	97.22	59.82	42.19	100.00	59.34	39.34	100.00	56.47

structural modifications (namely (L1)–(L8), (S1)–(S3), (S5)–(S9) and (S11)), while they were less robust to (S4) and (S10) modifications. Also, both JPlag and SCSDS were less robust to various combinations of structural modifications. Both tools are fully resistant to all lexical modifications, except to (L4) modifications (reformatting or modification of program output).

It is important to mention that both tools returned low similarity values when evaluated on programs, which contained some of the (S10) modifications (structural redesign of the source code) including moving of statechanging methods into a separate new class and extraction of methods into a parent class. It is important to mention that the (S10) modifications, especially their combinations, can represent a real threat to plagiarism-detection tools, above all because they can be done automatically by using Eclipse (i.e. its available refactoring tools) or similar tools. After having excluded these programs from the evaluation, the resulting precision and recall values improved for both the tools. These new precision and recall values among program pairs in this way modified test sets A and B are given in Table 4.

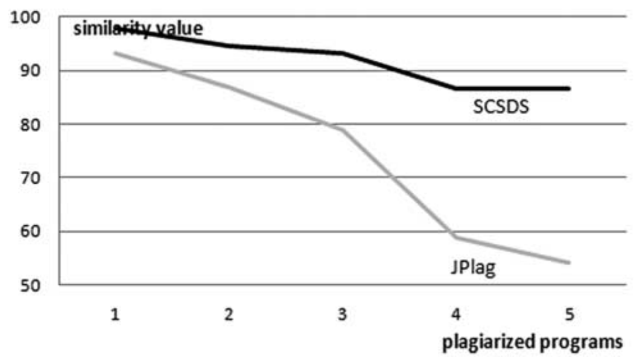
As can be seen, when the CT value is higher than 20%, SCSDS has a higher than or the same recall/precision as JPlag. Also, SCSDS detected all plagiarized programs with no false positive detection, for CT values between 35 and 70% (for test set A) and for CT values between 35 and 55% (for test set B). It is important to note that, for these modified data sets, SCSDS perfectly discriminates the plagiarized cases from the other programs for a fairly wide range of CTs. This is not the case with JPlag. Table 4 shows how recall changes when we increase the CT value: only for rather high CT values, SCSDS will miss some plagiarisms. Also, for these modified data sets, SCSDS always has at least either the perfect precision or perfect recall and for appropriately chosen CT values, SCSDS even gets both precision and recall at the highest values at the same time.

The main reason for these results is the tokenization process that SCSDS uses (described in Section 4) and that is much more detailed than the tokenization process, which JPlag uses. The average number of tokens JPlag generated per one source code file was 355 and 77 for programs in test set A and test set B, respectively, while the standard deviation was 158.68 and 41.06, respectively. On the same test sets, SCSDS generated 1397 and 234 tokens on average, respectively, while the standard deviation was 691.84 and 52.11, respectively.

As can be seen in Tables 3 and 4, for CT values lower than 35%, the precision significantly decreases for both JPlag and SCSDS. Also, it can be seen that, for CT values higher than 70%, the recall significantly decreases for both tools. This is why the F-measure value significantly decreases for these extreme CT values, and we can say that reasonable CT values can be found in the range 35–70%. Independent samples t-test revealed that there was a statistically significant difference of (i.e. SCSDS has significantly higher) average values of the F-measures for both the test sets (A and B) and for both the experiments (Tables 3 and 4) in the range of CT values of 35–70%, as follows:

(1) JPlag ($M = 96.35$, $\text{SD} = 1.46$) and SCSDS ($M = 99.05$, $\text{SD} = 0.00$), $t(7) = -5.24$, $p = 0.000$, for test set A and values reported in Table 3;

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018



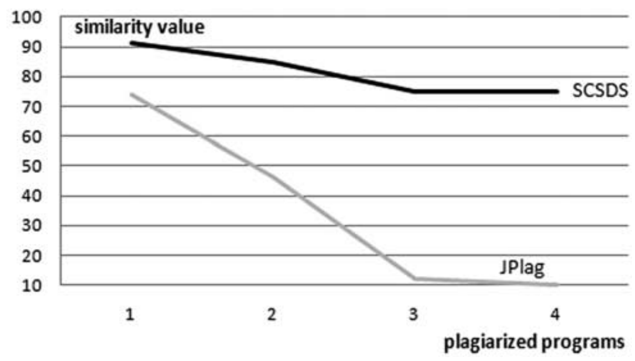


FIGURE 10. The matching percentage of JPlag and SCSDS in a case of addition of redundant statements or variables into the original programs in test set A (left) and test set B (right).

(2) JPlag ($M = 91.02$, SD $= 4.61$) and SCSDS ($M = 97.71$, SD $= 2.06$), $t(9.67) = -3.75$, $p = 0.002$, for test set B and values reported in Table 3;
(3) JPlag ($M = 97.36$, SD $= 1.48$) and SCSDS ($M = 100.00$, SD $= 0.00$), $t(7) = -5.05$, $p = 0.001$, for test set A and values reported in Table 4;
(4) JPlag ($M = 92.23$, SD $= 4.02$) and SCSDS ($M = 98.97$, SD $= 1.56$), $t(9.07) = -4.42$, $p = 0.002$, for test set B and values reported in Table 4.

To have a good plagiarism-detection tool, it is important to discriminate the plagiarisms from the other programs, i.e. it is important to have an appropriate distribution of similarity values found among plagiarism pairs and among non-plagiarism pairs. Ideally, if the plagiarism pairs are separated from non-plagiarism pairs, it can be said that a perfect discrimination of plagiarized programs from the genuine ones is achieved, for some CT values. Such CT values can be seen as a 'gap' between the minimum similarity value of the plagiarism pairs and the maximum similarity value of non-plagiarism pairs. It is important to mention that the more changes made to plagiarized programs, the more the matching percentage of JPlag decreased than that of SCSDS, and dramatically lowered the upper bound of the 'gap'. For example, in a case of addition of redundant statements or variables (S4), the 'plagiarizers' have made five plagiarized programs by inserting 8, 17, 25, 52 and 119 redundant statements and variables into the original program from test set A, respectively. The matching percentage of JPlag decreased from 93.20 to 54.30%, while the matching percentage of SCSDS decreased from 97.93 to 86.70% (Fig. 10). Similarly, the 'plagiarizers' have made four plagiarized programs by inserting 6, 10, 14 and 44 redundant statements and variables into the original program from test set B, respectively. In this case, the matching percentage of JPlag decreased from 73.70 to only 10.00%. At the same time, the matching percentage of SCSDS decreased from 91.93 to 74.94% (Fig. 8).

The number of programs in the test set affects the runtime of SCSDS more significantly than their size. More precisely, the runtime of SCSDS increases quadratically with the number of programs in the test set, whereas slightly superlinearly with the size of programs. The average time for reading, parsing and comparing one program pair from test sets A and B was 1501 and 262 ms, while the standard deviation was 800 and 88 ms, respectively. All tests were done on an Intel Duo CPU 2.00 GHz with 2 GB of RAM.

8. CONCLUSION AND FUTURE WORK

This paper describes a source code similarity detection system called SCSDS. This system showed promising results in detecting source code similarity when various lexical or structural modifications are applied to plagiarized source code. The results were tested on a representative test set of Java source code files. The performance of the SCSDS similarity measurement has shown promising results when compared with the JPlag performance. As a confirmation of these results, an independent samples t-test revealed that there was a statistically significant difference between average values of F-measures for the test sets that we used and for the experiments that we conducted in the practically usable range of CT values of 35–70%. The main reason for these results is that the tokenization process of SCSDS generates a detailed token set where tokens contain as much semantic information as possible.

When compared with the well-known state-of-the-art source code similarity detection systems, SCSDS has one extra feature, which is the combination of similarity detection algorithms. This system offers a possibility of adjustment of similarity algorithm weights, according to user needs. SCSDS can be extended easily in order to support new languages and new similarity detection algorithms. Also, this is one of the promising areas of extension to this work. It would be interesting to include some additional similarity measures, such as those proposed by Jones [12], Sager *et al.* [41] and Jiang *et al.* [42], in further evaluation. To further improve our similarity detection process, we will examine the impact of the position of the statements in a code (the context) on similarity detection. We

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

feel that the context can help significantly in most cases of plagiarism hiding.

Because SCSDS uses several similarity detection algorithms in the similarity detection process, it is obvious that this system is slower when compared with existing systems. By adding new similarity detection algorithms, the speed performance of SCSDS will be even worse. This is why a speed performance improvement of SCSDS is needed and will be investigated in the future work. There are some other limitations to SCSDS that should be addressed in the future work. An efficient, intuitive and easy accessible user interface is critical to wide adoption. This is why we are developing a new user interface in the form of a rich internet application.

REFERENCES

[1] Hannabuss, S. (2001) Contested texts: issues of plagiarism. *Libr. Manage.*, **22**, 311–318.

[2] Kustanto, C. and Liem, I. (2009) Automatic Source Code Plagiarism Detection. *SNPD Proc. 2009 10th ACIS Int. Conf. Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, Daegu, Korea, May 27–29, pp. 481–486. IEEE Computer Society, Washington, DC, USA.

[3] Maurer, H., Kappe, F. and Zaka, B. (2006) Plagiarism—a survey. *J. Univers. Comput. Sci.*, **12**, 1050–1084.

[4] Parker, A. and Hamblen, J. (1989) Computer algorithms for plagiarism detection. *IEEE Trans. Educ.*, **32**, 94–99.

[5] Prechelt, L., Malpohl, G. and Philippsen, M. (2002) Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.*, **8**, 1016–1038.

[6] Arwin, C. and Tahaghoghi, S.M.M. (2006) Plagiarism Detection Across Programming Languages. *Proc. 29th Australasian Computer Science Conf.*, Hobart, Australia, January 16–19, pp. 277–286. Australian Computer Society, Inc., Australia.

[7] Kleiman, A.B. and Kowaltowski, T. (2009) Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs. Technical Report IC-09-08. Instituto de Computação, Universidade Estadual de Campinas.

[8] Hage, J., Rademaker, P. and Van Vugt, N. (2011). Plagiarism Detection for Java: A Tool Comparison. *Proc. 1st Computer Science Education Research Conf., CSERC '11*, Heerlen, The Netherlands, April 7–8, pp. 33–46. ACM, New York, NY, USA.

[9] Mozgovoy, M., Frederiksson, K., White, D.R., Joy, M.S. and Sutinen, E. (2005) *Fast Plagiarism Detection System*. In: String Processing and Information Retrieval: 12th International Conference (SPIRE 2005). Lecture Notes in Computer Science, 3772/2005, pp. 267–270. Springer, London.

[10] Mozgovoy, M. (2006) Desktop tools for offline plagiarism detection in computer programs. *Inf. Educ.*, **5**, 97–112.

[11] Joy, M. and Luck, M. (1999) Plagiarism in programming assignments. *IEEE Trans. Educ.*, **42**, 129–133.

[12] Jones, E.L. (2001) Metrics based plagiarism monitoring. *J. Comput. Small Coll.*, **16**, 253–261.

[13] Liu, C., Chen, C., Han, J. and Yu, P. (2006) GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. *Proc. 12th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Philadelphia, USA, August 20–23, pp. 872–881. ACM, New York, NY, USA.

[14] Roy, C.H. and Cordy, R. (2007) A Survey on Software Clone Detection Research. Technical report No. 2007-541, School of Computing, Queen's University at Kingston, ON, Canada.

[15] Burrows, S., Tahaghoghi, S.M.M. and Zobel, J. (2007) Efficient and effective plagiarism detection for large code repositories. *Softw. Pract. Exp.*, **37**, 151–175.

[16] Hoad, T. and Zobel, J. (2002) Methods for identifying versioned and plagiarised documents. *J. Am. Soc. Inf. Sci. Technol.*, **54**, 203–215.

[17] Heintze, N. (1996) Scalable Document Fingerprinting. *USENIX Workshop on Electronic Commerce*, Oakland, CA, USA, November 18–21, pp. 191–200.

[18] Donaldson, J., Lancaster, A. and Sposat, P. (1981) A Plagiarism Detection System. *Proc. 12th SIGCSE Technical Symp. Computer Science Education*, St. Louis, MO, USA, pp. 21–25. ACM, New York, NY, USA.

[19] Halstead, M.H. (1972) Natural laws controlling algorithm structure? *ACM SIGPLAN Not.*, **7**, 19–26.

[20] Halstead, M.H. (1977) *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA.

[21] Bailey, C.T. and Dingee, W.L. (1981) A software study using halstead metrics. *SIGMETRICS Perform. Eval. Rev.*, **10**(1), 189–197.

[22] Donaldson, J., Lancaster, A. and Sposat, P. (1981) A Plagiarism Detection System. *Proc. 12th SIGCSE Technical Symp. Computer Science Education*, St. Louis, MO, USA, pp. 21–25. ACM, New York, NY, USA.

[23] Verco K. and Wise M. (1996) Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. *Proc. Australian Conf. Computer Science Education*, Sydney, Australia, July 1996, pp. 81–88. ACM, New York, NY, USA.

[24] Wise, M.J. (1992) Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing. *ACM SIGSCE Bull.*, **24**, 268–271.

[25] Wise, M.J. (1996) YAP3: Improved detection of similarities in computer programs and other texts. *ACM SIGCSE Bull.*, **28**, 130–134.

[26] Schleimer, S., Wilkerson, D.S. and Aiken, A. (2003) Winnowing: Local Algorithms for Document Fingerprinting. *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Diego, CA, USA, June 9–12, pp. 76–85. ACM, New York, NY, USA.

[27] Gitchell, D. and Tran, N. (1999) Sim: A Utility for Detecting Similarity in Computer Programs. *Proc. 30th SIGCSE Technical Symp. Computer Science Education*, New Orleans, LA, USA, March 24–28, pp. 266–270. ACM, New York, NY, USA.

[28] Clough, P. (2000) *Plagiarism in Natural and Programming Languages: An Overview of Current Tools and Technologies*. Research Memoranda: CS-00-05. Department of Computer Science, University of Sheffield, UK.

[29] Ahtiainen, A., Surakka, S. and Rahikainen, M. (2006) Plaggie: GNU-Licensed Source Code Plagiarism Detection Engine for Java Exercises. *Proc. 6th Baltic Sea Conf. Computing Education Research*, Uppsala, Sweden, November 9–12, pp. 141–142. ACM, New York, NY, USA.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018

[30] *JPlag—detecting software plagiarism*. https://www.ipd.uni-karlsruhe.de/jplag/ (last visit July 29, 2011).

[31] Wise, M.J. (1993) *String Similarity via Greedy String Tiling and Running Karp–Rabin Matching*. Department of CS, University of Sydney, ftp://ftp.cs.su.oz.au/michaelw/doc/RKR GST.ps.

[32] Karp, R.M. and Rabin, M.O. (1987) Efficient randomized pattern-matching algorithms. *Ibm J. Res. Dev. Math. Comput.*, **31**, 249–260.

[33] Chen, X., Li, M., Mckinnon, B. and Seker, A. (2002) A Theory of Uncheatable Program Plagiarism Detection and its Practical Implementation. Technical Report, UCSB.

[34] Chen, X., Francia, B., Li, M., McKinnon, B. and Seker, A. (2004) Shared information and program plagiarism detection. *IEEE Trans. Inf. Theory*, **50**, 1545–1551.

[35] Heckel, P. (1978) A technique for isolating differences between files. *Commun. ACM*, **21**, 264–268.

[36] Mozgovoy, M., Karakovskiy, S. and Klyuev, V. (2007) Fast and Reliable Plagiarism Detection System. *Frontiers in Education Conf.—Global eEngineering: Knowledge without Borders*, Milwaukee, USA, October 10–13, pp. S4H-11–S4H-14. IEEE Computer Society, Washington, DC, USA.

[37] Mozgovoy, M. (2007) Enhancing computer-aided plagiarism detection. Dissertation, Department of Computer Science, University of Joensuu, Department of Computer Science, University of Joensuu, PO Box 111, FIN-80101 Joensuu, Finland.

[38] Lancaster, T. and Culwin F. (2004) A comparison of source code plagiarism detection engines. *Comput. Sci. Educ.*, **14**, 101–117.

[39] Cosma G. (2008) An approach to source-code plagiarism detection and investigation using latent semantic analysis. Ph.D. Thesis, University of Warwick, Department of Computer Science.

[40] Brixtel, R., Fontaine, M., Lesner, B., Bazin, C. and Robbes, R. (2010) Language-Independent Clone Detection Applied to Plagiarism Detection. *10th IEEE Int. Working Conf. Source Code Analysis and Manipulation*, Timisoara, Romania, September 12–13, pp. 77–86. IEEE Computer Society, Washington, DC, USA.

[41] Sager, T., Bernstein, A., Pinzger, M. and Kiefer, C. (2006) Detecting Similar Java Classes Using Tree Algorithms. *Proc. 2006 Int. Workshop on Mining Software Repositories*, Shanghai, China, May 22–23, pp. 65–71. ACM, New York, NY, USA.

[42] Jiang, L., Misherghi, G., Su, Z. and Glondu, S. (2007) Deckard: Scalable and Accurate Tree-based Detection of Code Clones. *Proc. 29th Int. Conf. Software Engineering*, Minneapolis, USA, May 20–26, pp. 96–105. IEEE Computer Society, Washington, DC, USA.

APPENDIX

As described in [5], JPlag has also some important drawbacks:

(C1) It is not resistant to every form of reordering of statements within code blocks (S2). In six out of eight test cases, JPlag got locally confused when the reordering of statements was done (there were no data dependences that required a certain order of statements), i.e. in 75% of test cases that applied reordering of statements within code blocks (S2), JPlag did not detect this kind of plagiarism and the JPlag similarity value was significantly reduced.

(C2) It is not resistant to every form of reordering of code blocks (S3). In three out of 30 test cases (10%), JPlag got locally confused since a list of very small methods was reordered and intermixed with declarations of some variables.

(C3) It is not resistant to modifications of control structures (S5). In each of 35 test cases, JPlag got locally confused when some new expressions or new statements were inserted into (or removed from) the original source code. When this kind of modification is done, the token list is modified in such a way that almost no matching blocks will be found by JPlag. Examples of such modifications are replacement of a *for* loop by a *while* loop and replacement of a *switch* statement by a sequence of *if* statements.

(C4) It is not resistant to every form of changing of data types and modification of data structures (S6). In five out of six test cases (83%), JPlag got locally confused; for example, when a character array was used instead of a string, or when a two-element array of *int* values was replaced by two *int* variables.

(C5) It is not resistant to every form of inlining and refactoring (S7). In 16 out of 20 test cases (80%), JPlag got locally confused by inlining of small methods or by code refactoring into new methods.

(C6) It is not resistant to redundancy (S8). In each of 15 test cases, JPlag got locally confused by addition or removal of methods that are never called or variables that are never used.

(C7) It is not resistant to addition of temporary variables and changing of subexpressions (S9). In each of 28 test cases, JPlag got locally confused; for example, when subexpressions were moved into the declaration of additional temporary variables, and then in the original expressions, the new temporary variables were used instead of the replaced subexpressions.

(C8) It is not resistant to structural redesign of the source code (S10). In each of three test cases, JPlag got locally confused; for example, when state-changing methods were moved to a newly created helper class.

(C9) It is not resistant to modification of scope (S11). In each of nine test cases, JPlag got locally confused since modification of scope modified the token list in such a way that no matching blocks were found.

Plagiarism attacks where these kind of modifications are combined can reduce JPlag similarity value below the given threshold value.

Downloaded from https://academic.oup.com/comjnl/article-abstract/56/1/70/530993 by University of Texas at Arlington user on 18 October 2018