Plagiarism Detection:
A Tool Survey and Comparison

Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz

Centro de Ciências e Tecnologias da Computação (CCTC)
Departamento de Informática, Universidade do Minho
Gualtar, Portugal
`{vtiagovm,danielamoraisfonte,pedrorangelhenriques,danieladacruz}@gmail.com`

─── **Abstract** ───

We illustrate the state of the art in software plagiarism detection tools by comparing their features and testing them against a wide range of source codes. The source codes were edited according to several types of plagiarism to show the tools accuracy at detecting each type.

The decision to focus our research on plagiarism of programming languages is two fold: on one hand, it is a challenging case-study since programming languages impose a structured writing style; on the other hand, we are looking for the integration of such a tool in an Automatic-Grading System (AGS) developed to support teachers in the context of *Programming courses*.

Besides the systematic characterisation of the underlying problem domain, the tools were surveyed with the objective of identifying the most successful approach in order to design the aimed plugin for our AGS.

1 Introduction

Plagiarism is concerned with the use of work without crediting its author, including the cases where someone uses previous code. It overrides copyrights in all the areas from arts or literature to sciences, and it is from the old days a forensic subject. Plagiarism does not only affect creativity or commercial businesses but also has negative effects in the academic environment. If a student plagiarises, a teacher will be unable to properly grade his ability.

In an academic context, it is a problem that applies not only to text documents but to source-code as well. Very often students answer the teacher assessment questions, submitting plagiarised code. This is why teachers have a strong need to recognise plagiarism, even when students try to dissimulate it. However, with a large number of documents, this becomes a burdensome task that should be computer aided.

This paper is precisely concerned with this subject, discussing approaches and tools aimed at supporting people on the detection of source code files that can be plagiarised.

Due to the complexity of the problem itself, it is often hard to create software that accurately detects plagiarism, since there are many ways a programmer can alter a program without changing its functionality. However, there are many programs for that purpose.

Some of them are off-line tools, like **Sherlock** [13, 11], **YAP** [22, 14, 11], **Plaggie** [1, 11], **SIM** [10, 1, 11], **Marble** [11], and **CPD** [5] which, even though it was only made to detect copies, is still a useful tool. There are also online tools like **JPlag** [16, 14, 7, 1, 11, 15] and **MOSS** [17, 14, 7, 1, 11, 15], and even tool sets like **CodeSuite** [18].

The objective of this paper is to introduce and discuss existing tools in order to compare their performance. As the tools that were analysed use distinct technological approaches, it is important to choose the best candidate as to build the envisaged tool for our Automatic Grading System (namely iQuimera, an AGS under construction in our research group – for details, please see [8]) upon it.

The paper is organised in five sections. The state-of-the-art concerning source code plagiarism detection tools is presented in Section 2. In that section, we start with some basic concepts and we briefly discuss the major methodological approaches supporting the tools. Then we define a criteria composed of eight characteristics that should be kept in mind when studying each tool. This criteria enable us to create and present a comparative table that allows a quick survey. After this, another table, comparing the performance of the tools during the experimental study conducted, is shown and discussed presenting an overview of the problem. Section 3 describes the experimental research done, focusing on the source code files that were carefully prepared to test the tools exhaustively. Section 4 is devoted to the experimentation itself. For that, the tools are introduced in alphabetical order, a short description is provided and the specific results for each test are presented in tables. The conclusion and future work are presented in Section 5.

2   State of the Art

There are several techniques for the detection of plagiarism in source code. Their objective is to stop unwarranted plagiarism of source code in academic and commercial environments.

If a student uses existing code, it must be in conformance with the teacher and the school rules. The student might have to build software from the ground up, instead of using existing source code or tools that could greatly reduce the effort required to produce it but, this will allow a teacher to properly grade the student according to his knowledge and effort.

If a company uses existing source code, it may be breaking copyright laws and be subjected to lawsuits because it does not have its owners consent.

This need led to the development of plagiarism detectors, this is, programs that take text files (natural language documents, programs or program components) as input and output a similarity measure that shows the likelihood of there being copied segments between them. These outputs will usually come as a percentage.

2.1   Background

To understand the method used to create the tools and the offered functionality, it is necessary to understand the basic concepts involved. The following is a list presenting relevant terms:

Token A word or group of characters, sometimes named *n-gram* where n denotes the number of characters per token (as seen in [23]). Since white-spaces and newlines are usually ignored, the input "while(true) {" could produce two 5-grams: "while(" and "true){".

Tokenization The conversion of a natural text into a group of tokens, as defined above.

Hash A number computed from a sequence of characters, usually used to speed up comparisons. For instance, if we use the ASCII[1] values of each character we could turn the token "word" into 444 (119 + 111 + 114 + 100).

Hashing The conversion of a sequence of tokens into their respective hash numbers.

---

[1] American Standard Code for Information Interchange

Fingerprint A group of characteristics that identify a program, much like physical fingerprints are used to identify people. An example would be if we consider a fingerprint to be composed by 3 hashes that are multiples of 4 (as seen on [23]).

Matching Algorithm The algorithm used to compare a pair of hashes, which allows the detection of the similarity degree between them. The comparison is performed by verifying if the fingerprints of each file are close, according to a pre-defined threshold. An example is to use the sum of the differences (between hashes) as the matching algorithm and a value of 10 as the threshold. In which case, taking a pair of files with the fingerprints: [41,582,493] and [40,585,490], the program would match, as the sum of the differences is 9 (|41-40|+|582-585|+|493-490| = 1+3+3).

Structural information This is information from the structure of a programming language. An example would be the concept of comments, conditional controls, among others.

The implementation of these concepts is always specific to each tool; since this process is not usually explained, it can only be learnt by inspecting its source code.

2.2 Approaches

From our research, the following list was built detailing the several methodologies that were used.

- An attribute-based methodology, where metrics are computed from the source code and used for the comparison. A simple example would be: using the size of the source code (number of characters, words and lines) as an attribute to single out the source codes that had a very different size. This methodology was mentioned by [20].
- A token-based methodology, where the source code is tokenized and hashed, using those hashes to create the fingerprints. This methodology was used by [21] and by [17].
- A structure-based methodology, where the source code is abstracted to an Internal intermediate Representation (IR), for example, an AST[2] or a PDG[3]) and then this IR is used for the comparison. This enables an accurate comparison. This methodology was used by [2] and by [14].

These methodologies go from the least accuracy with high efficiency to the highest accuracy with low efficiency.

Some examples of the metrics that could be used in an attribute-based methodology would be: files size, number of variables, number of functions and number of classes, among others. These metrics will usually be insufficient as students will usually solve the same exercise, which would cause a lot of suspicion from the tool.

The token-based methodology came with an attempt to balance the accuracy and the efficiency, often using RKS-GST[4] [21] which is a modern string matching algorithm that uses tokenization and hashing. To improve the results even further some tools mix some structure dependent metrics and modifications such as removing comments (used by JPlag 4.3) or sorting the order of the functions (used by Marble 4.4).

The structure-based approach uses abstractions that maintain the structure of the source code. This makes the technique more dependent on the language but it will also make the detection immune to several types of plagiarism such as, switching identifier names, switching the positions of functions and others.

---

[2]   Abstract Syntax Tree
[3]   Program Dependency Graph
[4]   Running Karp-Rabin matching and Greedy String Tiling
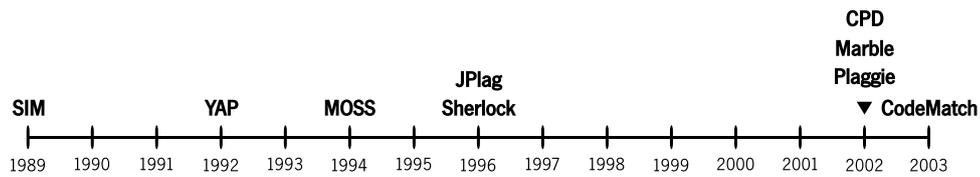
Figure 1 A timeline showing the years in which each tool was developed or referenced.

2.3 Existing Tools

We found several tools for the detection of software plagiarism throughout our research. Some of those tools were downloaded and tested. Other tools, like Plague [19, 22, 13] and GPlag [15, 3] were not taken into account since we could not access them.

A timeline was produced (see Figure 1), indicating the years when the tools were developed or, at least mentioned in an article. More details about the analysed tools are presented in Section 4.

2.3.1 Features Comparison

Inspired on [11], the following criteria were used to compare the tools:

1st) Supported languages: The languages supported by the tool.

2nd) Extendable: Whether an effort was made to make adding languages easier.

3rd) Quality of the results: If the results are descriptive enough to distinguish plagiary from false positives.

4th) Interface: If the tool has a GUI[5] or presents its results in a graphical manner.

5th) Exclusion of code: Whether the tool can ignore base code.

6th) Submission as groups of files: If the tool can consider a group of files as a submission.

7th) Local: If the tool can work without needing access to an external web service.

8th) Open source: If the source code was released under an open source license.

A table (see Table 1) was produced to report the criteria defined above. The values can be ✓(Yes), ✗(No), a ? in the cases where we could not ascertain if the feature is present, or a number in the case of supported languages.

Table 1 Comparison of the plagiarism detection tools.

Name	1	2	3	4	5	6	7	8
CodeMatch	36	✗	✓	✓	✓	✗	✓	✗
CPD	6	✓	✗	✓	?	?	✓	✓
JPlag	6	✗	✓	✓	✓	✓	✗	✗
Marble	5	✓	✓	✗	✓	?	✓	✗
MOSS	25	✗	✓	✓	✓	✓	✗	✗
Plaggie	1	?	?	✓	?	?	✓	✓
Sherlock	1	✗	✗	✗	✗	✗	✓	?
SIM	7	?	✓	✗	✗	✗	✓	?
YAP	5	?	✓	✗	✓	?	✓	✗

---

[5] Graphical User Interface

We can observe that both the CodeMatch and the MOSS tools support several languages, which makes them the best choices when analysing languages that the other tools do not support. However, others like CPD or Marble are easily extendable to cope with more languages. Note that if the tools support *natural language*, they can detect plagiarism between any text document but will not take advantage of any structural information.

Overall, we found that GUIs are unnecessary to give detailed output, so long as the tool can produce descriptive results that indicate the exact lines of code that caused the suspicion. This is observable with the use of Marble, SIM and YAP tools as they do not offer GUIs but still have descriptive results. On the other hand, tools like Sherlock do not present enough output information.

On academic environments, both the 5$^{\text{th}}$ and the 6$^{\text{th}}$ criteria are very important as they allow teachers to filter unwanted source code from being used in the matches. This means that tools like JPlag and MOSS allow for the proper filtering of the input source code.

As said in the introduction, the 7$^{\text{th}}$ criteria reveals that JPlag and MOSS are the only tools dependent on online services as they are web tools.

In what concerns the availability of tools on open licenses, only CPD and Plaggie satisfy that requirement. For those wanting to reuse or adapt the tools, the 7$^{\text{th}}$ and the 8$^{\text{th}}$ criteria are important as the tool would need to have a license allowing its free use, and integration would benefit from having the tool distributed alongside the application.

2.3.2 Results Comparison

In order to test each tool and compare the results, we needed to start from an original and produce several files with different cases of plagiarism. To systematise the work, we felt the need to identify different types of plagiarism. So we built the types list below based on [13, 3, 6].

1$^{\text{st}}$ type of plagiarism This type of plagiarism is an exact copy of the original. It is the simplest type of plagiarism as no modifications were done to hide it.

2$^{\text{nd}}$ type of plagiarism This type of plagiarism is when the comments are changed or removed. It is the easiest modification as it has no risk of affecting the code. Note that most plagiarism detectors either ignore comments or have an option to ignore them and will not be diverted by this type of plagiarism. Of course, to do so the tools need the syntatic information on how comments are defined in the language.

3$^{\text{rd}}$ type of plagiarism This type of plagiarism is done by changing every identifier such as variable or function names.

4$^{\text{th}}$ type of plagiarism This type of plagiarism is when local variables are turned into global ones, and vice versa.

5$^{\text{th}}$ type of plagiarism This type of plagiarism is when the operands in comparisons and mathematical operations are changed (e.g. x < y to y >= x).

6$^{\text{th}}$ type of plagiarism This type of plagiarism is when variable types and control structures are replaced with equivalents. Care has to be taken as to avoid breaking the code functionality since the types will need to be converted to have the same behaviour.

7$^{\text{th}}$ type of plagiarism This type of plagiarism is when the order of statements (read lines) is switched. This is a common type of plagiarism as one only needs to make sure that the source code will keep the behaviour.

8$^{\text{th}}$ type of plagiarism This type of plagiarism is when groups of calls are turned into a function call and vice versa.
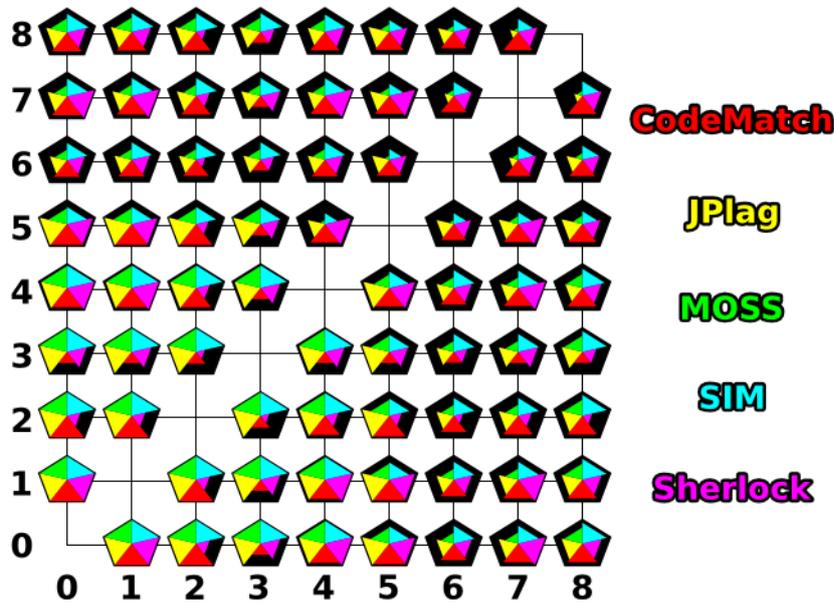
Figure 2 An overview of the results obtained.

The two types of plagiarism listed below were also referred in the cited lists. However, we will not consider them as they depend entirely on human supervision.

- Generating source-code by use of code-generating software.
- Making a program from an existing program in a different language.

Figure 2 shows the results that were obtained from comparing several files (as detailed in Section 3) and gives us an overview. The graphic gives a condensed overview of the results achieved, where each number represents a type of plagiarism that was used on the files that were compared. Each polygon has 5 triangles with each colour representing a tool. The size of the triangles represents the similarity degree between the files in the X and Y axis. While the graphic appears to be symmetric, the results for the MOSS and SIM tools are asymmetrical in a few cases (as detailed in Section 4).

We can see that the 1st type of plagiarism (exact copy) was easily detected by all the tools but the other types of plagiarism always had a big impact on some of the tools. Here are the most noticeable:

- MOSS has the most trouble with a lot of the types of plagiarism as it tries very hard to avoid false-positives, thus discarding a lot of information.
- Sherlock has a lot of trouble with the 2nd type of plagiarism (comments changed) as it compares the entire source codes without ignoring the comments.
- CodeMatch has the most trouble with the 3rd type of plagiarism (identifiers changed) as its algorithms must make some sort of match between identifiers.

We can see that most tools have a hard time when comparing from the 6th to the 8th types of plagiarism as they had a lot of small changes or movements of several blocks of source code. These are the types of plagiarism that would be best detected with the use of a structural methodology as, despite those changes to the structure, the context remains intact.

We can conclude that every tool has a weakness when it comes to certain types of plagiarism but notice that none of the results are 0%, the triangle representing the match is just so small that it is easy to miss. We also confirm that no type of plagiarism can fool all the tools at the same time.

3 Strategy for Testing the Tools

To demonstrate the accuracy of the existing tools detecting the different cases of plagiarism, two sample programs were collected and edited based on the types of plagiarism listed above. As the selected tools are prepared to work with Java or C programs, the exercises were written in both languages.

The following list of actions were performed to create the eight variants.

- to produce the first case, it is a straightforward file copy operation.
- to apply the second strategy, we simply removed the comments or changed their contents.
- to create a third variant, we changed most variable and function identifiers into reasonable replacements.
- to produce a copy of the forth type, we moved a group of local variables into a global scope and removed them from the function arguments. The type declaration was removed but the initialisation was kept.
- to obtain a file for the fifth type, we switched the orders of several comparisons and attributions. For example, we replaced $x+=y-5$ by $x+=-5+y$.
- to get a file for the sixth type, we replaced variable types such as *int* and *char* with *float* and *string*, along with the necessary modifications to have them work the same way.
- to create another copy according to the seventh type, we moved several statements, even some which broke the behaviour (write after read) to consider cases were the plagiarist was not careful.
- to produce a file exhibiting the eighth type, we applied this type of plagiarism in the easy (move entire functions) and the hard (move specific blocks of code in to a function) ways.

3.1 Source Code used in the Tests

In this subsection we give a brief description about the two sets of source files used in the tests. Each set includes both the original source file and 8 variants, where each one had a different type of plagiarism applied to it, according to the list above (see Section 2.3.2).

As all the files are based on the same original, from a theoretical point of view, every test should return a match of 100%.

Source code samples were written in both Java and C languages. As the samples are crucial for the assessment performed, we would like to show them here. Due to space limitations, it is impossible to include the original programs here (or their variants); so, we decided to make them available online, and just include in the paper a description of their objective and size.

3.1.1 Program 1: Calculator

The first set is based on a program which goal is to implement a simple calculator with the basic operations (addition, subtraction, division and multiplication) and a textual interface.

Only the Java version of the file was used to produce the results presented. The following metrics characterise the original file:

- **Number of lines**: 56
- **Number of functions**: 1
- **Number of variables**: 4

The Java source code files are available online at
`https://tinyurl.com/dopisiae/slate/source/calc_java.zip`,
as well as the C source code files, at
`https://tinyurl.com/dopisiae/slate/source/calc_c.zip`.

3.1.2   Program 2: 21 Matches

The objective was to make an interactive implementation of the 21 matches game where, starting with 21 matches, each player takes 1 to 4 matches until there are no more. The one getting the last match will lose the game. The game can be played against another player or against the computer.

Only the C version of the file was used to produce the results presented. The following metrics characterise the original file:

- **Number of lines**: 189
- **Number of functions**: 4
- **Number of variables**: 8

The C source code files are available online at
`https://tinyurl.com/dopisiae/slate/source/21m_c.zip`,
as well as the Java source code files, at
`https://tinyurl.com/dopisiae/slate/source/21m_java.zip`.

4   Tool Details and Test Results

Along Section 2, nine tools were identified (and ordered in a timeline) and they were compared: CodeMatch, CPD, JPlag, Marble, MOSS, Plaggie, Sherlock, SIM, and YAP. Aiming at providing a deeper knowledge about the tools explored, in this section they will be introduced with some more detail and the output they produced when experimented against the two test sets will be shown.

As described in SubSection 2.3.2, 8 types of plagiarism were considered. Two test sets were created containing an original program file and 8 variant files, modified according to the type of plagiarism.

We have produced tables from the results of each tool when matching each of the files with the other 8. This gives us a look at the accuracy of their metrics for each type of plagiarism. As stated earlier, from a theoretical point of view every test should return a match of 100% so, the higher the results the higher the tools accuracy.

Note that, due to the 16 page constraint, we only show the full result tables for the CodeMatch tool and trimmed the remaining result tables to their first line.

To compare the results pertaining to tables of the same tool, we used the following algorithm: Given two tables, we subtract each value in the second table with the value in the first table and add their results. As an example, consider the tables [4,3] and [2,1]. In this case the formula is: (2-4)+(1-3), which is equal to -4. This allows us to see whether the results increased or decreased from the first table to the second. We will refer to this metric as the difference metric (DM). If there are several lines of results, we will use the average which is calculated by dividing the sum of every line DM by the number of lines.

■ **Table 2** The results produced the CodeMatch tool for the Calculator set.

	0	1	2	3	4	5	6	7	8
0	100	100	95	70	95	95	76	90	86
1	100	100	95	70	95	95	76	90	86
2	95	95	100	63	90	90	69	84	80
3	70	70	63	100	70	70	63	70	67
4	95	95	90	70	100	91	75	89	85
5	95	95	90	70	91	100	72	84	86
6	76	76	69	63	75	72	100	76	71
7	90	90	84	70	89	84	76	100	82
8	86	86	60	67	85	86	71	82	100

Note that, the DMs we show are from the full results. You can find all the results as well as the DM calculations on the following link: `https://tinyurl.com/dopisiae/slate/index.html`.

Since most tools only give a metric for each pair of files, most tables are symmetrical. For the asymmetrical tables, the percentage must be read as the percentage of the line file that matches the column file.

4.1 CodeMatch

CodeMatch [11] is a part of CodeSuite [18] and detects plagiarism in source code by using algorithms to match statements, strings, instruction sequences and identifiers. CodeSuite is a commercial tool that was made by SAFE[6], which is housed at `http://www.safe-corp.biz/index.htm`. It features several tools to measure and analyse source or executable code.

This tool is only available as an executable file (binary file) and only runs under Windows.

CodeMatch supports the following languages: ABAP, ASM-6502, ASM-65C02, ASM-65816, ASM-M68k, BASIC, C, C++, C#, COBOL, Delphi, Flash ActionScript, Fortran, FoxPro, Go, Java, JavaScript, LISP, LotusScript, MASM, MATLAB, Pascal, Perl, PHP, PL/M, PowerBuilder, Prolog, Python, RealBasic, Ruby, Scala, SQL, TCL, Verilog, VHDL and Visual Basic.

4.1.1 Results for the Calculator Code

CodeMatch returned good results (see Table 2) for the Calculator, Java source codes (see Section 3.1.1). Note that the files were compared with themselves since CodeMatch compares the files in two folders and the same folder was used. This is a moot point as those cases got 100% matches.

4.1.2 Results for the 21 Matches Code

The results (see Table 3) for the 21 Matches source code were similar to the previous (DM=-15.56), except for the 3rd type of plagiarism (Identifiers changed). This is probably due to the fact that the 21 Matches source codes have more identifiers than the Calculator ones.

---

[6] Software Analysis and Forensic Engineering

Table 3 The results produced by CodeMatch tool for the 21 Matches Code.

	0	1	2	3	4	5	6	7	8
0	100	100	100	54	90	96	78	87	87
1	100	100	100	54	90	96	78	87	87
2	100	100	100	57	90	96	76	87	87
3	54	54	57	100	54	54	52	54	53
4	90	90	90	54	100	86	78	85	86
5	96	96	96	54	86	100	75	84	84
6	78	78	76	52	78	75	100	77	77
7	87	87	87	54	85	84	77	100	85
8	87	87	87	53	86	84	77	85	100

Table 4 The results produced by JPlag tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	100	100	84.7	100	40.4	43.4	39.6

4.2 CPD

CPD [5] is a similarity detector that is part of PMD, a source code analyser that finds inefficient or redundant code, and is housed at `http://pmd.sourceforge.net/`. It uses the RKS-GST algorithm to find similar code.

It supports the following languages: C, C++, C#, Java, Ecmascript, Fortran, Java, JSP, PHP and Ruby.

Since CPD only returns detailed results without a match percentage metric, we could not produce the tables.

4.3 JPlag

JPlag [16, 14, 7, 1, 11, 15] takes the language structure into consideration, as opposed to just comparing the bytes in the files. This makes it good for detecting plagiarism despite the attempts of disguising it.

It supports the following languages: C, C++, C#, Java, Scheme and Natural language.

JPlag gives us results organised by the average and maximum similarities, without repeated values. It produces an HTML file presenting the result and allows the user to view the detailed comparison of what code is suspected to be plagiarism.

4.3.1 Results for the Calculator Code

As we can see (in Table 4), the results were good since there where a lot of exact matches (100%), showing that JPlag was impervious to several types of plagiarism.

4.3.2 Results for the 21 Matches Code

JPlag was better (DM=114.49) at detecting the plagiarism for this set of source codes since the results (see Table 5) are similar in relation to the exact matches and got better matches on the harder cases like the 6th, 7th and 8th types of plagiarism.

■ **Table 5** The results produced by JPlag tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	100	100	100	93.7	100	62.8	70.5	75.6

■ **Table 6** The results produced by Marble tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100 U	100 U	100 U	94 U	100 S	77 S	81 S	66 U

4.4 Marble

Marble [11], which is described in `http://www.cs.uu.nl/research/techreps/aut/jur.html`, facilitates the addition of languages by using code normalisers to make tokens that are independent of the language. A RKS-GST algorithm is then used to detect similarity among those tokens.

It supports the following languages: Java, C#, Perl, PHP and XSLT.

The results are presented through a suspects.nf file which has several lines in the following structure: "echo *M1 S1 S2 M2 U/S* && edit *File 1* && edit *File 2*". The *M1* and *M2* values indicate the match percentages, *S1* and *S2* give us the size of the matches and the *U/S* flag indicates if the largest percentage was found before (U) or after (S) ordering the methods.

Note that only the Calculator results are available since Marble does not support the C language.

4.4.1 Results for the Calculator Code

As expected, a few source codes accuse the movement of methods (have an S flag). This is verified in the 5th, 6th and 7th types of plagiarism that had operations, variables and statements moved, respectively.

4.5 MOSS

MOSS [17, 14, 7, 1, 11, 15] automatically detects similarity between programs with a main focus on detecting plagiarism in several languages that are used in programming classes. It is provided as an Internet service that presents the results in HTML pages, reporting the similarities found, as well as the code responsible. It can ignore base code that was provided to the students and focuses on discarding information while retaining the critical parts, in order to avoid false positives.

It supports the following languages: A8086 Assembly, Ada, C, C++, C#, Fortran, Haskell, HCL2, Java, Javascript, Lisp, Matlab, ML, MIPS Assembly, Modula2, Pascal, Perl, Python, Scheme, Spice, TCL, Verilog, VHDL and Visual Basic.

This tool gives us the number of lines matching between each pair of files and calculates the match percentages by dividing it by the number of lines in the file. The algorithms used were made to avoid false positives at the cost of getting lower percentages.

Its HTML interface was produced by the author of JPlag (Guido Malpohl) and, the results come in both an overview and detailed forms.

Table 7 The results produced by MOSS tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	99	99	99	85	99	30	55	36

Table 8 The results produced by MOSS tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	99	99	99	94	67	44	45	67

4.5.1   Results for the Calculator Code

As we can see in Table 7, the results are reasonable but the strategies used to avoid false positives decreased the matches (ex.: 100% to 99%). We can also notice that some types of plagiarism, namely the 6th, 7th and 8th had low matches due to their high complexity.

4.5.2   Results for the 21 Matches Code

As expected the increase in the size of the source codes translated into the variance of the results (see Table 8) albeit the increases (DM=4.67) were rather small when compared to other tools.

4.6   Plaggie

Plaggie [1, 11], which is housed at `http://www.cs.hut.fi/Software/Plaggie/`, detects plagiarism in Java programming exercises. It was made for an academic environment where there is a need to ignore base code. It only supports the Java language. As we were unable to compile the tool, we can not present its results.

4.7   Sherlock

Sherlock [13, 11], which is housed at `http://sydney.edu.au/engineering/it/~scilect/sherlock/`, detects plagiarism in documents through the comparison of fingerprints which, as stated in the website, are a sequence of *digital signatures*. Those digital signatures are simply a hash of (3, by default) words.

It allows for control over the *threshold* which hides the percentages with lower values, the number of words per *digital signature* and the *granularity* of the comparison by use of the respective arguments: -t, -n and -z. It supports the following language: Natural language.

Sherlock is an interesting case as its results are a list of sentences in a "*File 1* and *File 2*: *Match%*" format. This format does not help the user find the highest matches, it simply makes the results easier to post-process. Results were produced with all the combinations of the settings from -n 1 to 4 and -z 0 to 5.

4.7.1   Results for the Calculator Code

With the default settings (equivalent to -n 3 -z 4), we can see that some of the results (see Table 9) are good, mainly for the 1st (Unaltered copy), 4th (Scope changed) and 7th (Statements order switched) types of plagiarism. Seeing as Sherlock matches natural language, the similarity values will subside due to textual changes and be mostly unaffected by movements.

■ **Table 9** The results produced by Sherlock tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	62	44	100	62	25	100	45

■ **Table 10** The results produced by Sherlock tool for the Calculator code with the -n 2 argument.

	1	2	3	4	5	6	7	8
0	100	55	72	100	100	72	100	80

To demonstrate the effect of tweaking the -n and -z parameters, two more tables, Tables 10 and 11, are presented exhibiting the results of Sherlock when invoked with the arguments "-n 2" and "-z 3" respectively.

Table 10 shows us that using the -n parameter can greatly improve (DM=198.78) the results. This argument changes the number of words per digital signatures, Meaning that with a smaller value, Sherlock will find plagiarism in smaller sections of source code. This translates into better matches in small changes and source code movements, but worse matches on longer modifications (as seen for the 2nd type of plagiarism).

The -z argument changes Sherlocks "granularity," a parameter that serves to discard part of the hash values. As seen on Table 11, Sherlock was more sensitive to changes which resulted in an overall decrease (DM=-36).

Overall, we can see that Sherlock is a very specific tool and that further studies would have to be made in order to ascertain the adequate parameters to use for specific situations.

4.7.2 Results for the 21 Matches Code

For the 21 Matches source codes, the results (see Table 12) seem more accurate (DM=146.44) than the results for the Calculator source codes (see Table 9). This was likely due to the increased size of the source codes.

4.8 SIM

SIM [10, 1, 11], which is housed at `http://dickgrune.com/Programs/similarity_tester/`, is an efficient plagiarism detection tool which has a command line interface, like the Sherlock tool. It uses a custom algorithm to find the longest common sub-string, which is order-insensitive.

It supports the following languages: C, Java, Pascal, Modula-2, Lisp, Miranda and Natural language.

SIM is an interesting tool. Despite not having a GUI, its results are quite detailed. The -P argument can be used to report the results in a "*File 1* consists for *Match%* of *File 2* material" format, giving a quick rundown of the results. It has a few arguments that change its behaviour; however, unlike Sherlock, its default values seem to work well for most cases of plagiary.

4.8.1 Results for the Calculator Code

This tool is quite good with results showing exact matches for most types of plagiarism.

■ **Table 11** The results produced by Sherlock tool for the Calculator source code with the -z 3 argument.

	1	2	3	4	5	6	7	8
0	100	31	41	88	76	42	88	41

■ **Table 12** The results produced by Sherlock tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	97	32	62	90	85	89	93	88

4.8.2 Results for the 21 Matches Code

On Table 14, we can see that it was harder to detect plagiarism in bigger source codes (DM=-120.33), although it was beneficial in a specific case, the 4$^{\text{th}}$ type of plagiarism.

4.9 YAP

YAP[7] [22, 14, 11], which is housed at `http://www.pam1.bcs.uwa.edu.au/~michaelw/YAP.html`, is a tool that currently has 3 implementations, each using a fingerprinting methodology with different algorithms. The implementations have a tokenizing and a similarity checking phase and just change the second phase. The tool itself is an extension of Plague.

YAP1 The initial implementation was done as a Bourne-shell script and uses a lot of shell utilities such as diff, thus being inefficient. It was presented by Wise [20].

YAP2 The second implementation was made as a Perl script and uses an external C implementation of the Heckel [12] algorithm.

YAP3 The third and latest iteration uses the RKS-GST methodology.

The tools themselves are separate from the tokenizers but packaged together. These tools are said to be viable at the detection of plagiarism on natural language, aside from their supported languages.

It supports the following languages: Pascal, C and LISP.

We were unable to produce understandable results.

5 Conclusions

In this paper, plagiarism in software was introduced and characterised. Our purpose is to build a tool that will detect source code plagiarism in academic environments, having in mind that detecting every case is implausible. We hope that it will be able to produce accurate results when faced with complex types of plagiarism, like switching statements and changing variable types, and will be immune to the simpler types of plagiarism, such as modifying comments and switching identifier names. One of the most difficult and challenging problem in this context is the ability to distinguish between plagiarism and coincidence and we are aware of the danger of false positives [4, 9]. However this is a topic that deserves further investigation. To plan and support our working decisions, we devoted some months to the research of the existing methodologies and tools, as related in this paper. The comparative study involving nine tools that were available for download and were ready to run, gave us

---

[7] Yet Another Plague

◼ **Table 13** The results produced by SIM tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	100	100	44	100	91	99	100

◼ **Table 14** The results produced by SIM tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	100	100	100	97	72	75	75	88

the motivation to build a tool that adopts a structure based methodology with the AST as the abstract structure. Although the existing tools do not detect accurately all the types of plagiary identified, we found that most of them were easy to use with their default options but Marble and YAP have special file structure requirements and Plaggie has to be compiled. We found JPlags interface to be intuitive and offering a wide variety of options but, as it is not local, we recommend SIM as a viable alternative. We would like to point out that CodeSuite package offers a good number of tools for the detection of source code theft. The next task will be to develop the prototype tool based on the chosen methodology, and perform its evaluation in real case studies.

References

1 Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of 6th Koli Calling Intern. Conference on Comp. Ed. Research*, pages 141–142, New York, USA, 2006. ACM.

2 I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of IEEE ICSM 1998*, pages 368–377, 1998.

3 Andrés M. Bejarano, Lucy E. García, and Eduardo E. Zurek. Detection of source code similitude in academic environments. *Computer Applic. in Engineering Education*, 2013.

4 Miranda Chong and Lucia Specia. Linguistic and statistical traits characterising plagiarism. In *COLING 2012*, pages 195–204, 2012.

5 Tom Copeland. Detecting duplicate code with PMD's CPD, 2003.

6 G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Trans. on Educ.*, 51(2):195–200, May 2008.

7 Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *3rd IEEE IC-BNMT*, pages 668–673, 2010.

8 Daniela Fonte, Ismael Vilas Boas, Daniela da Cruz, Alda Lopes Gançarski, and Pedro Rangel Henriques. Program Analysis and Evaluation using Quimera. In *ICEIS'2012*, pages 209–219. INSTICC, June 2012.

9 Cristian Grozea and Marius Popescu. Who's the thief? automatic detection of the direction of plagiarism. In *In CICLing*, pages 700–710, 2010.

10 Dick Grune and Matty Huntjens. Het detecteren van kopieën bij informatica-practica. *Informatie*, 31(11):864–867, 1989.

11 Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28, 2010.

12 Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.

13 Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Trans. on Educ.*, 42(2):129–133, May 1999.

14 Xiao Li and Xiao Jing Zhong. The source code plagiarism detection using AST. In *International Symposium IPTC*, pages 406–408, 2010.

15 Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD'06*, pages 872–881. ACM Press, 2006.

16 Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultät für Informatik, Universität Karlsruhe, 2000.

17 Saul Schleimer. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD*, pages 76–85. ACM Press, 2003.

18 Ilana Shay, Nikolaus Baer, and Robert Zeidman. Measuring whitespace patterns as an indication of plagiarism. In *Proceedings of the ADFSL Conference*, pages 63–72, 2010.

19 Geoff Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138, 1990. Special Issue on Using Software Metrics.

20 Michael J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *ACM SIGCSE Bulletin*, volume 24, pages 268–271. ACM, 1992.

21 Michael J. Wise. *Running Karp-Rabin matching and greedy string tiling.* Basser Dept. of Computer Science, University of Sydney, Sydney, 1993.

22 Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *SIGCSEB: SIGCSE Bulletin*, pages 130–134. ACM Press, 1996.

23 Robert M. Zeidman. Software tool for detecting plagiarism in computer source code, 2003.