A comparison of plagiarism detection tools

Jurriaan Hage

Peter Rademaker

Nikè van Vugt

Technical Report UU-CS-2010-015
June 2010

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

A comparison of plagiarism detection tools

Jurriaan Hage
Peter Rademaker
Nikè van Vugt

Abstract

In this paper we compare five tools for detecting plagiarism in source code texts: JPlag, Marble, MOSS, Plaggie, and SIM. The tools are compared with respect to their features and performance. For the performance comparison we carried out two experiments: to compare the sensitivity of the tools for different plagiarism techniques we have applied the tools to a set of intentionally plagiarised programs. To get a picture of the precision of the tools, we have run the tools on several incarnations of a student assignment and compared the top 10's of the results.

1 Introduction

Source code plagiarism can be defined as trying to pass off (parts of) source code written by someone else as one's own (i.e., without indicating which parts are copied from which author).

Plagiarism occurs often in academic environments. Students then intentionally or unintentionally include sources in their work without a proper reference. Manual detection of plagiarism in a collection of hundreds of student submissions is infeasible and uneconomical. Therefore, software tools have emerged that assist lecturers in detecting the presence of plagiarism.

It should be understood from the start that none of these tools can actually prove the presence (or absence, for that matter) of *plagiarism.* Typically, the tools return a *measure of similarity* for each pair of programs (or even for each pair of modules taken from different submissions), and then human intervention is needed to determine whether the similarity is due to plagiarism, or is caused, for example, by the fact that a certain assignment is implemented in a standard way (e.g. stacks and lists are generally implemented in a common way), by students using the same examples from the lectures, by students working together, or by plain coincidence. Even a lecturer who suspects students of plagiarism will want to hear their side of the story before passing a verdict or passing the case on to others for consideration.

Marble [12] is a plagiarism detection tool for Java programs, developed by the first author. It is used at the Department of Information and Computer Sciences at Utrecht University to assist lecturers in the detection of plagiarism in programming assignments. Thus far, Marble has helped in detecting several cases of plagiarism. Even so, the question quickly arises how well a plagiarism detection tool performs as compared to others, in terms of how well they can counter attempts by students to hide the plagiarism and to which extent the top-n submissions of highest similarity for each tool can be shown to be actual plagiarism. The former study is addressed by a sensitivity analysis in Section 5.1, the latter study is performed for $n = 10$ in Section 5.2. The studies are intended to reveal possible weak points of the tools, but may also teach us to which extent different tools may be complementary. In plagiarism detection, more than in most other areas, different tools can reinforce each other.

Before we do the quantative performance comparison, we also provide a qualitative evaluation of the tools. Here we answer such questions as: what kind of interface do the tools provide, what are the languages they can deal with, and so forth. This issue is addressed in Section 4, while the criteria we use to perform this evaluation are described in Section 3.

The main goal of this paper is to assist lecturers and teaching assistants with an overview of the current state of the art in plagiarism detection for source code, to highlight the features of each tool and provide measurements of their performance, in order to make an informed decision which tool to use for plagiarism detection. Moreover, this evaluation provides pointers for the maintainers of the tools under evaluation in what ways their tool may be improved.

These goals motivate our choice of tools for the comparison. Generally speaking, we have included tools that are non-commercial, readily available, and, because of our particular comparison benchmark, suited for plagiarism detection for Java. The tools, JPlag, MOSS, Marble, Plaggie and SIM, are introduced in Section 4. Other tools that we have found during our study, but that are not considered here, are described in Section 2, which also relates our work to plagiarism tool comparisons available in the literature.

2 Related work

First, we give an overview of the different implementation strategies that are used by tools for similarity detection. Next, we discuss all the tools we found in the literature and on the Internet, whether we include them in our comparison, and if not, why not. Finally, we shortly consider comparative studies similar to our own, and explain why we feel the need to do some comparisons of our own.

2.1 Implementation strategies

Since the implementation of the tools is not our main concern – we are interested in comparing the results of the tools – we give only a short overview of the different comparison techniques described in the literature.

Many, but not all, approaches start by transforming the original source code file into a *token string*, where tokens are the lexical entities that are the building blocks of a program (e.g., identifiers, keywords and such). On the way, source code aspects that can be easily changed without changing the semantics of the program (comments, variable names, indentation) are removed. The token representations of the programs are then compared to detect similarities.

Early systems used a purely *attribute-based approach*: several properties of a source code file were counted (see for example [13]). Thus, each file was represented by a sequence of numbers. Two files with representing sequences that were 'similar enough' were considered as possible cases of plagiarism.

Later, structure-based systems were developed, that compare the *structure* of programs rather than transforming them into sequences of numbers. Often, the original programs are first tokenized, and then the resulting token strings are compared using various string comparison algorithms (e.g., Running-Karp-Rabin Greedy-String-Tiling in [21]). Also the program dependency graphs used in [15] can be considered structure-based.

2.2 Tools

The criteria for selecting tools for our comparison largely depend on the answers to the following questions: Can the tool deal with Java programs? Is it readily available? Is it free?

Table 1 provides an overview of the results of our investigation of the literature and the Internet. Note that we have omitted from the present study a few tools that could have been included. Their evaluation is still ongoing, and the results will be made available in a future study.

The tools that were selected are described in detail in Section 4. The citations mentioned in connection with a tool are original sources that describe the tool: a paper by the authors of the tool, or the tool's homepage, or both. When no citations are given, we were unable to find them.

Tool	**Included**	**Reason not included**
Big Brother	no	not found
CodeMatch [1]	not yet	commercial (but free evaluation license possible)
Cogger [17]	no	not found
DetectaCopias	no	in Spanish, not found
GPlag [15]	no	not found
Jones [13]	no	not found
JPlag [18]	yes	
MOSS [2, 20]	yes	
PDetect [16]	no	does not support Java
Plaggie [3, 8]	yes	
PlagioGuard [19]	no	not found
Saxon	no	not found
Sherlock [4]	not yet	
SID [5, 11]	not yet	
SIM [6]	yes	
TEAMHANDIN	no	does not support Java
XPlag [9]	no	meant for inter-lingual plagiarism detection
YAP3 [21]	no	does not support Java

Table 1: Known program plagiarism detection tools

2.3 Comparisons

Comparisons of program plagiarism detection tools can be roughly divided into two categories: feature comparisons and performance comparisons.

Feature comparisons are qualitative comparisons; they describe the properties of a tool, like which programming languages it supports, whether it is a local or a web-based application, which algorithm is used to compare the files, etc (see for instance [14, 10, 19]. By nature, such a comparison is purely descriptive, and based on such a comparison it is difficult to say which of the tools should be considered 'the best'.

Performance comparisons are quantitative comparisons; they typically describe some experiments run on some tools. They compare the results of tools, rather than their properties (see [10, 11]).

Papers presenting one particular tool often provide the results of experiments run to test that tool (see for instance [20, 11]). In the JPlag technical report [18], a short textual comparison of the most important features of JPlag, MOSS and YAP3 is given, followed by an extensive empirical evaluation of JPlag. In the latter, there is also included a long list of so-called "futile attacks": commonly used attempts to fool a plagiarism detector (most of which did not fool JPlag).

In [11] the Software Integrity Diagnosis system (SID) is presented, which is a specific continuation of a general project that concerns itself with determining the measure of similarity of two sequences (whether these sequences represent genomes, documents, music or programs). The measure that SID uses is based on Kolmogorov complexity. The results of SID are compared to the results of MOSS and JPlag on the same programs. The authors report that in many cases the results of the three tools are similar, but that both MOSS and JPlag seem to be sensitive to random insertion of redundant code (e.g., a lot of `System.out.println` statements). The results of our sensitivity analysis (Section 5) indicate that indeed MOSS is rather sensitive to numerous small changes in the source code.

3 Criteria for qualitative comparison

In this section we list the criteria that we use for our qualitative comparison of the selected tools.

1 - Supported languages The minimal requirement for tools to be included in this survey was to support plagiarism detection in Java source code files, but some tools support several other languages.

2 - Extendability Directly related to supported languages is extendability: the ability of a tool to be adapted or configured so that it can be used for other programming languages. This can be useful since it's quite likely that within a computer science faculty more than one programming language is used, and one does not want to use a different tool for every different programming language.

3 - Presentation of results After the running of the tools, a lot of effort has to be done to check if found similarities between files concern actual cases of plagiarism, cooperation between students, or a coincidence. In most cases, this takes a lot more time than running the query itself. Therefore, it is important to present the results in such a way that post processing can be done as efficiently as possible.

A good presentation of the results should at least contain the following elements:

Summary: Here meta data like the total number of submissions, the succesfull parses, the parameters used for running the detection, and a chart showing the distribution of similarities over the result should be shown. Such a histogram can help identifying the range of similarities that clearly represent no plagiarism, and the range of values that should be investigated further.

Matches: The matches should be listed sorted by similarity, in a comprehensive way. This can be done pairwise, or in clusters. It should also be possible to set a certain threshold on the minimum similarity to include in the result overview.

Comparison tool: To be able to easily compare pairs that are marked as 'similar' it is helpful if there is an editor that is able to display both files next to each other, highlighting the similarities.

4 - Usability Another criterion is the ease of use of the tool. It should be possible for a user to use the tool without first having to spend a lot of effort in getting the tool to work. For instance, a graphical user interface instead of a command line interface can be very helpful.

5 - Exclusion of template code It is normal for student programming assignments to share some common base code from which students have to complete an assignment. Also, it often happens that something explained in the lectures can be used in a programming assignment.

In both cases, the results of a search for plagiarism may include many legimate matches, that do not indicate plagiarism, but can be explained by one of the previously mentioned legitimate causes. Some plagiarism detection tools allow the user to place such legitimately shared code in a common base file that will be ignored during the detection phase. This can help prevent a lot of false positives.

6 - Exclusion of small files Related to the exclusion of template code is the exclusion of small files. Very small files – such as so called Java beans, that only consist of attributes and their getter and setter methods – are most likely to return high similarity scores. This is simply a result of the way such classes are implemented and does not indicate plagiarism. A tool can either mention the file size in its result, which can help in detecting false positives caused by small files, or it may provide a way to exclude files up to a certain size.

7 - Historical comparisons With this criterion we denote the ability of a tool to compare a new set of submissions with submissions from older incarnations of the same programming assignment, *without* again mutually comparing the older incarnations. So there must be a way to distinguish older submissions from newer ones. Either by indicating which are the new or old submissions when starting the tool, or by putting different incarnations in different directories.

8 - Submission or file-based rating Whether a tool rates the submissions by every separate file or by submission (a directory containing the files of which the program consists) greatly influences the readability of the output. When a submission consists of multiple files, it is important that the plagiarism detection is performed for each file in the submission, since the detection of plagiarism in only one of the files of the submission is enough to consider the whole submission as being plagiarized and therefore invalid. When a submission based rating is used the comparison might still be file based. Then the question is how the file comparison scores are combined into a score for the whole submission.

9 - Local or web-based Some tools are provided as web services. This requires a lecturer to send the student assignments over the network. Here you take a risk of exposing confidential information to the outside world.

Other tools have to be downloaded and run locally.

10 - Open source An advantage of open source is of course the possibility of extending or improving the program to better suit the situation you intend to use it for.

4 Feature comparison

In this section we introduce the tools we selected for our comparison, and discuss in particular the criteria described in Section 3. The tools are presented in alphabetical order.

JPlag

JPlag [18] was developed by Guido Malpohl at the University of Karlsruhe. In 1996 it started out as a student research project and a few months later it evolved into a first online system. In 2005 JPlag was turned into a web service [7] by Emeric Kwemou and Moritz Kroll.

JPlag converts programs into token strings that represent the structure of the program, and can therefore be considered as using a structure-based approach. For comparing two token strings JPlag uses the "Greedy String Tiling" algorithm as proposed by Michael Wise [21], but with different optimizations for better efficiency.

1 - Supported languages JPlag supports Java, C#, C, C++, Scheme and natural language text.

2 - Extendability The only part of JPlag that depends on the language in which the programs to be checked are written is the front-end, where programs are converted into token strings (by a parser or a scanner, depending on the programming language). No description is found of how to make a front-end for a language that is not currently supported.

3 - Presentation of results JPlag presents its results as a set of HTML pages. The pages are sent back to the client and stored locally. The main page is an overview that includes a table with the configuration used to run the query, a list of failed parses, a chart showing the distribution of the similarity values, and listings of the most similar pairs, sorted by average similarity as well as by maximum similarity.

One distinctive feature of JPlag is the clustering of pairs. This makes it easier to see whether a submission is similar to several other submissions. This can be the case when students of previous years have put their solution to the assignment on the web.

When selecting a pair, all files in both submissions are shown side-by-side in frames. Parts of the files that have been found to be similar on token-level are marked with colors. On the top of the screen a list of all the similar parts in the submissions is displayed. When selecting such a part, the editors jump to the corresponding code.

4 - Usability The very easy to use Java Web Start client, the clear listings of the results and the editor for comparing submissions all contribute to a very easy to use application.

5 - Exclusion of template code It is possible to submit a basecode directory containing files that should be excluded from the comparison.

6 - Exclusion of small files Yes

7 - Historical comparisons No

8 - Submission or file-based rating JPlag expects that every program that has to be examined is situated in its own directory, which is seen as a separate submission of a student. All files within such a directory will be considered as belonging to the same submission. Ratings are by submission.

9 - Local or web-based JPlag is available as a web-based service. There is a Java Web Start client available on the JPlag website [7] to upload the files to the server. They also provide extensive instructions on how to write your own JPlag client.

10 - Open source No

Marble

Since there is no English publication on Marble, we take some more time here to explain the particulars of this tool.

Marble is a tool developed in 2002 at Utrecht University by the first author. The intention was to create a simple, easily maintainable tool that can be used to detect cases of suspicious similarity between Java submissions. By collecting all the submitted programs for the various assignments in the computer science department in Utrecht, and by comparing against these as well, Marble has been instrumental in exposing a few dozen cases of proven plagiarism. For reasons of scalability it is essential that the tool can distinguish between old and new submissions, so that old submissions are not compared to each other anymore.

Marble uses a structure-based approach to compare the submissions. It starts by splitting the submission up into files so that each file contains only one top-level class. The next phase is one of normalization, to remove details from these files that are too easily changed by students: a lexical analysis is performed (implemented in Perl using regular expressions) that preserves keywords (like `class`, `for`) and frequently used class and method names (like `String`, `System`, `toString`). Comments, excessive white-space, string constants and import declarations are simply removed, other tokens are abstracted to their token "type". For example, every hexadecimal number is replaced by H and every literal character by L.

For each file, Marble actually computes two normalized versions: one in which the order of fields, methods and inner classes is exactly like that of the original file, and one in which the fields, methods and inner classes are grouped together, and each group is sorted. Sorting is performed in a heuristic fashion. For example, the methods are first ordered by number of tokens, then by total length of the token stream, and finally alphabetically.

In order to be able to extract inner classes, methods and fields from the Java class file without having to parse, Marble first annotates the braces { and } in the program with their nesting depth, and splits up classes by matching on braces of the right nesting depth (depth 0 corresponds to the braces of the class definition, depth 1 to inner classes and methods). Knowing the location of the opening brace of a method does not directly give the start location of the method, but by scanning backwards to the first encountered semi-colon or closing brace, it is possible to find it. Inner classes are treated similarly.

The person running the tool can choose to compare the sorted or the unsorted normalized versions (or both). Because the sorting is heuristic, small changes to the methods in a class can totally change the sorting of methods. We have observed one case where a student made a number of changes, but did not reorder the methods. Because of the changes, corresponding methods in the original version and the plagiarised version ended up in substantially different positions, negatively influencing the score. This is why it also makes sense to compare the unsorted versions.

The actual comparison of the normalized files is done using the Unix/Linux utility `diff`. The score is then computed from the ratio between the number of lines on which they differ from each other and the total number of lines in the two compared normalized files.

1 - Supported languages Marble supports Java. Support for Perl, PHP and XSLT is experimental.

2 - Extendability The language-dependent part is the normalization phase, which can easily be adapted for similar programming languages.

3 - Presentation of results The results are outputted to a script named either `suspects.nf` (unsorted) or `suspects.nfs` (sorted), which, when run, outputs for each pair that exceeds a given threshold of similarity the similarity score, the size of both files, and then opens both *original* files in a `diff` editor to show the differences. The user may also choose not to run the script, but to open it in a text-editor and manually investigate the suspects.

4 - Usability Marble is available as a Perl script (on demand, from `jur@cs.uu.nl`) and has a command line interface.

5 - Exclusion of template code It is not possible to exclude template code.

6 - Exclusion of small files The Marble script uses a threshold value to exclude files below a certain size.

7 - Historical comparisons If submissions are stored in an appropriately ordered file system (one directory per assignment, divided into subdirectories for the different incarnations, which are divided into subdirectories for the different reviewers, that contain the submissions of that incarnation (each in their own directory)), then Marble is able to compare each file of a new submission to not only the files from submissions inside the same incarnation, but also to those in older incarnations – *without* comparing the older submissions among themselves.

8 - Submission or file-based rating For every pair of files a similarity rating is computed.

9 - Local or web-based Marble is run locally.

10 - Open source No

MOSS

MOSS is an acronym for *Measure Of Software Similarity.* MOSS was developed in 1994 at Stanford University by Aiken et al. It is being provided as a webservice that can be accessed using a script that can be obtained from the MOSS website [2]. A MOSS account (and submission script) can be obtained by e-mail from `moss@moss.stanford.edu`. The MOSS submission script works for Unix/Linux platforms and may work under Windows with Cygwin, but the latter is untested (from personal correspondence with the author).

To measure similarity between documents, MOSS compares the standardised versions of the documents: MOSS uses a document fingerprinting algorithm called *winnowing* [20]. Document fingerprinting is a technique that divides a document into contiguous substrings, called k-grams (with k being picked by the user). Every k-gram is hashed, and a subset of all the k-gram hashes is selected as the document's fingerprint. Winnowing is an efficient algorithm for selecting these subsets.

1 - Supported languages MOSS can currently analyze code written in the following languages: C, C++, Java, C#, Python, Visual Basic, JavaScript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, HCL2.

2 - Extendability Yes (by the authors).

3 - Presentation of results The output of MOSS is an HTML presentation with clickable links and an integrated HTML `diff` editor that allow for easy navigation through the results. It is placed on a web page on the MOSS web server. A link to that web page is returned when MOSS is finished checking the documents. Precautions have been taken to keep the result pages private. The results cannot be crawled by robots or browsed by people surfing the Web. The random number in the URL is made known only to the account that submitted the query, and there is no way to access the results except through that URL. The result pages expire automatically after 14 days; code is not retained indefinitely on the server. Taken together, these measures would seem to make the potential for abuse quite small.

4 - Usability When registering to MOSS a submission script is mailed that can be used to upload the submissions.

5 - Exclusion of template code MOSS allows one to supply a base file of code that should be ignored if it appears in programs; MOSS never considers code that appears in a base file to match any other code. Furthermore, MOSS can automatically eliminate matches to code that one expects to be shared (e.g., libraries or instructor-supplied code), thereby eliminating false positives that arise from legitimate sharing of code.

6 - Exclusion of small files Yes

7 - Historical comparisons Yes, at the command line the user can indicate which are new and which are old submissions.

8 - Submission or file-based rating By default, MOSS compares files based on submissions or directories, however, the submission script exposes an option that allows for file to file comparison.

9 - Local or web-based MOSS is a web-based system.

10 - Open source No

Plaggie

Plaggie [8, 3] is a source code plagiarism detection engine meant for Java programming exercises. In appearance and functionality, it is similar to JPlag, but there are also aspects of Plaggie that makes it very different from JPlag: Plaggie must be installed locally and its source code is open. Plaggie was developed in 2002 by Ahtiainen et al. at Helsinki University of Technology. It is a stand-alone command line Java application.

The basic algorithm used for comparing two source code files is the same as for JPlag: tokenisation followed by Greedy String Tiling as described in [18]. The authors mention that they did not implement the optimisations that were implemented in JPlag.

In the Plaggie readme file, the authors list known unsuccessful attacks (changing comments or indentation, and changing the names of classes, methods or variables) as well as known successful attacks ("moving inline code to separate methods and vice versa, inclusion of redundant program code, changing the order of if-else blocks and case-blocks") and known problems (accuracy of Plaggie when used on GUI code or automatically generated code).

1 - Supported languages Java 1.5

2 - Extendability Apparently not.

3 - Presentation of results By default, Plaggie's results are shown in plain text on the standard output and are stored in a graphical HTML format (using frames). It also offers an option to disable the plain text output. The output includes a table showing statistics such as the distribution of the different similarity values, the number of files in submissions, etc. The HTML report includes a sortable table containing the top results and their various similarity values. For further inspection a submission can be clicked which leads us to a side-by-side comparison of the files, highlighting the similarities.

4 - Usability Configuring Plaggie has to be done via a configuration file that is placed in the directory containing the submissions. Running Plaggie is done using its command line interface.

5 - Exclusion of template code Template code can be excluded by providing the file containing the template code. In addition, Plaggie offers the possibility to exclude code from the comparison based on filename, subdirectory name, or interface.

6 - Exclusion of small files Plaggie does allow excluding submissions from the results below a certain similarity value. It does not, however, allow the exclusion of files based on their size.

7 - Historical comparisons No

8 - Submission or file-based rating Plaggie compares file by file, but accumulates the results per submission.

9 - Local or web-based Plaggie is run locally. Its results are in HTML format, which allows for web-based publication.

10 - Open source Yes, GNU-licensed.

SIM

SIM [6] is a software similarity tester for programs written in C, Java, Pascal, Modula-2, Lisp, Miranda, and for natural language. It was developed in 1989 by Dick Grune at the VU University Amsterdam. The current version, SIM 2.26, is from 2008. Matty Huntjens wrote the shell scripts that take the output of SIM and turn it into a plagiarism report.

The process SIM uses to detect similarities is to tokenize the source code first, then to build a forward reference table that can be used to detect the best matches between newly submitted files, and the text they need to be compared to. Clever programming makes sure that the computational time stays within reasonable bounds.

While SIM is no longer actively maintained and supported, its source code is publicly available, as are some binaries [6]. According to the authors, the plagiarism detection side of it is very much tailored to the situation at the VU University Amsterdam and therefore not very portable.

1 - Supported languages C, Java, Pascal, Modula-2, Lisp, Miranda and natural language texts.

2 - Extendability SIM can be readily extended by providing a description of the lexical items of a new language.

3 - Presentation of results The results of SIM are presented in a flat text file that first outputs some general information about the compared files, such as number of tokens of each of the files, total number of files, names, etc.

4 - Usability Command line interface, fairly usable.

5 - Exclusion of template code No

6 - Exclusion of small files No

7 - Historical comparisons According to our definition of this criterion (that states that earlier incarnations are not compared among themselves each time a new incarnation is added): no. However, according to the SIM website [6], SIM's efficiency is very useful here: "SIM is very efficient and allows us to compare this year's students' work with that collected from many past years".

8 - Submission or file-based rating SIM's output is on a per-file basis, however, files are only mutually compared if they come from different submission directories.

9 - Local or web-based SIM is run locally.

10 - Open source Yes

In Table 2 we summarize the evaluation for easy comparison. Instead of mentioning all supported languages of the tools again, we have simply counted them. For the criteria '3 - Presentation of results' and '4 - Usability' we introduce a scale of 1 to 5, 1 meaning 'poor' and 5 meaning 'very good'.

Feature	**JPlag**	**Marble**	**MOSS**	**Plaggie**	**SIM**
1 - Supported languages (#)	6	1	23	1	5
2 - Extendability	no	no	no	no	yes
3 - Presentation of results (1-5)	5	3	4	4	2
4 - Usability (1-5)	5	2	4	3	2
5 - Exclusion of template code	yes	no	yes	yes	no
6 - Exclusion of small files	yes	yes	yes	no	no
7 - Historical comparisons	no	yes	no	no	yes
8 - Submission or file-based rating	submission	file	submission	submission	file
9 - Local or web-based	web	local	web	local	local
10 - Open source	no	no	no	yes	yes

Table 2: Feature matrix plagiarism detection tools

5 Performance comparison

In this section we describe two experiments we have performed to measure the performance of the tools quantitatively. The first of these is a sensitivity analysis, the second a top-n comparison.

In the sensitivity analysis we take two Java classes, refactor them both in seventeen different ways and consider how similar the refactored versions are to the original versions, according to each of the tools (Section 5.1, first part). In this way we hope to gain some insight in the weak and strong points of each tool.

Since we have seen in earlier experiments that combining refactorings quickly degrades the scores further, we have expanded our sensitivity experiment by also considering the effects of a combination of refactorings (Section 5.1, second part). Both parts of the sensitivity experiments compare files rather than submissions.

In our last experiment, the top-n comparison (Section 5.2), we run the tools on a real-life collection of programs (in which a few cases of plagiarism are known to be hiding), and we consider the top-10 of highest scoring pairs of submissions for each tool in turn. Hence, this experiments compares submissions rather than files.

There are several reasons why we feel the need to do this top-n comparison. First of all, when we want to determine the accuracy of one tool, we could run it on a large collection of real-life submissions in which numerous occurrences of obvious, less-obvious, and highly-refactored cases of plagiarism are hidden. Then we could manually check whether the tool found the right plagiarism candidates. Obviously, for large enough collections this is a very time-consuming task. If, however, we can run several tools on this same set of submissions, we can use the extra information the other tools provide to more easily determine whether 'our' tool is accurate.

A second reason that we included the top-n comparison is the following. In the sensitivity experiment, for instance, each tool returns a score for each pair consisting of one original file and one refactored file. We may normalize the score for maximal similarity to 100, and minimal similarity to 0. If a tool scores 100 for a refactored version in comparison with the original, then we consider the tool to be insensitive to that particular refactoring. However, it is hard to draw any conclusions from the other scores in the range 0-100, because each tool has its own way of computing a score: a score of 90 for one tool does not necessarily mean the same as a score of 90 for another tool. By comparing the top-n's of all the tools, we have eliminated at least this problem.

Finally, note that it is very easy to construct a tool that always scores 100 for any comparison of two files. Such a tool is of course very insensitive and will do well in the sensitivity experiment. However, it is bound to do badly in the top-10 comparison.

5.1 Sensitivity analysis

We start this subsection with the part of the set-up of this experiment that is common for both the 'single refactorings' and the 'combined refactorings' part.

Common set-up
To compare the sensitivity of the tools to (single) counter measures, we have carried out an experiment in which we have created seventeen different versions of the programming assignment "Animated Quicksort" (and a combi version for the combined sensitivity experiment). This assignment was used in a course on Distributed Programming at Utrecht University. The version at hand consists of five Java files: `QSortAlgorithm.java`, `QSortApplet.java`, `QSortView.java`, `QSortObserver.java`, `QSortModel.java`. Out of these files, the latter three either are small or are expected to be very similar by nature. Therefore, for both sensitivity experiments, we take only `QSortAlgorithm.java` and `QSortApplet.java` into account.

We consider seventeen strategies that can be used to try and disguise a plagiarized program (see Table 3). The modifications in this table preserve the semantics of the program but change its appearance. Furthermore, the modifications do not require in-depth knowledge of the program and do not take much time to apply. Students who plagiarize often do not have the knowledge or time to build a program themselves, which makes these modifications ideal candidates to disguise plagiarism.

Modifications 12 through 17 are based on the refactoring functionalities offered by the popular Eclipse IDE[1], which is known to be used by a lot of students.

Sensitivity to a single refactoring

Below we first describe the set-up for this particular experiment, then the way in which we present the results, and finally we interpret these results.

Set-up
We denote the original version of the class at hand by 'version 0' (i.e., there is a version 0 for both `QSortApplet.java` and `QSortAlgorithm.java`). We create seventeen different versions of each of the two classes, using the refactorings described in Table 3. The resulting versions are numbered according to the numbers of the refactorings in Table 3; in each version only a single modification is applied. In this way we aim to be able to assess the sensitivity of the tools to the different modification strategies.

Some tools rate per file and others per submission. Since the sensitivity analysis is a per-file analysis, we sometimes had to explicitly create two submission directories, one containing only a version 0 and the other containing only a version i, for $i = 1..17$, in order to ensure that the tool in question indeed returned a score for each of the pairs '(version 0, version i)'.

Presentation of the results
Figures 1 and 2 show the results of the sensitivity experiment. For each modified version, the

[1] `http://www.eclipse.org`

Version	Description
0	Original Animated Quicksort assignment (two files out of five)
1	Translated comments and minor layout changes
2	Moved 25% of the methods
3	Moved 50% of the methods
4	Moved 100% of the methods
5	Moved 50% of class attributes
6	Moved 100% of class attributes
7	Refactored GUI code
8	Changed imports
9	Changed GUI text and colors
10	Renamed all classes
11	Renamed all variables
12	Eclipse - Clean up function: member access (use 'this' qualifier for field and method access, use declaring class for static access)
13	Eclipse - Clean up function: code style (use modifier final where possible, use blocks for if/while/for/do, use parentheses around conditions)
14	Eclipse - Generate hash code and equals function
15	Eclipse - Externalize strings
16	Eclipse - Member type to top level (extract inner classes)
17	Eclipse - Generate getters and setters (for each attribute generate a getter and setter)

Table 3: Semantics preserving modifications applied to version 0

figures show six similarity values. (In Appendix A we have collected similar charts, but there each chart compares `diff` to one tool only.)

The first value is the similarity value computed using the Unix `diff`[2] utility directly on the source files. This serves as a quantification of the effect of the modification. The value is computed as follows:

$$\text{similarity} = 100 - \frac{100 * \text{nr. of different lines } (\texttt{diff})}{\text{nr. of lines file1} + \text{nr. of lines file2}}$$

Note that it is important to use `diff` with the option `-w`, in order to ignore lines that differ only in the amount and/or position of whitespace.

The next five values are the similarity values as computed by the five plagiarism detection tools we consider. For Marble the selection of the similarity value is straightforward, since for every pair of files Marble produces a single similarity score. Other tools, however, return two similarity scores for every comparison between a pair of files. A score indicating the percentage of lines in file A that are found to be similar to lines in file B, and vice versa, a score that indicates the percentage of lines in file B that are similar to lines in file A. For all tools that use this scoring approach we have selected the maximum score of the two similarity scores.

Interpretation of the results

We do not always know exactly which function a tool uses to compute its scores. Therefore, when tool X scores higher than tool Y for a certain version, this does not necessarily mean that tool Y is more sensitive to this kind of attack than tool X.

Moreover, one way of using a plagiarism detection tool is to run it on a set of submissions, and then check the candidate plagiarism pairs designated by the tool, starting with the pair with the highest score, then the second highest, and so on, until no candidates are found to be serious supected cases anymore. In this way, the exact scores are not important.

Still, we can gather some interesting information from Figures 1 and 2. As mentioned before, we consider a tool score of 100 as indicating insensitivity of the tool to the refactoring. Furthermore, a very low tool score, a tool score lower than the corresponding `diff` score and a tool score that

[2]`http://www.gnu.org/software/diffutils/diffutils.html`

is significantly lower than the corresponding scores of the other tools all are incentives for further consideration.

From the charts in Figures 1 and 2 we can observe several interesting things. We consider each version (or group of related versions) in turn. Moreover, for each version and each tool, we choose between the two figures to get the most meaningful score. For instance, for `QSortAlgorithm.java`, versions 7, 8, 9, 15 and 16 score 100 for both `diff` and each tool, because those refactorings do not affect that particular file at all. However, they do cause some changes in `QSortApplet.java`, hence for these versions we choose the values from Figure 1.

We start with a general remark about the two figures. Four out of the five tools quite often score 100, and even `diff` does that occasionally. MOSS, however, never scores 100. Since this also happens for identical files, this may be an intrinsic feature of MOSS, and we tend to consider scores like 98 and 99 as being 100.

- First of all, in both figures we can see from the scores for version 1 that all tools are insensitive to changes in layout and comments (strictly speaking, MOSS is not *completely* insensitive).
- The bars for versions 2, 3 and 4 show the sensitivity for changes in the location of methods. Only Marble is completely insensitive to changes to the locations of methods. In Figure 2 we observe that JPlag, MOSS, Plaggie and SIM are considerably sensitive to changes to the locations of methods. In the same figure, these tools also show a slight decrease in the similarity score when we change the location of the class attributes.
- None of the tools is completely insensitive to changes in the location of the class attributes (versions 5 and 6).
- Figure 1 shows that all tools but Marble score significantly lower than `diff` when GUI code is refactored.
- The modifications applied in versions 8, 9, 10, and 11 are completely ineffective for all tools in both files (with a small exception for JPlag and MOSS). These modifications concern the renaming of variables and classes and the changing of imports.
- The modifications in versions 12 (in which `this` qualifiers are added to every field and method access) seem to cause some confusion for Marble, and quite a lot for MOSS and SIM. JPlag and Plaggie are insensitive to this type of refactoring. For MOSS this corresponds to the results reported in the comparative study by the authors of SID [11], but for JPlag it does not.
- Almost the same holds for version 13 (Eclipse's code style clean-up function). However, Plaggie's score drops here as well.
- Both JPlag and Marble seem to do poorly on version 14 (generation of hash code and equals function by Eclipse): they score lower than `diff`. However, MOSS, SIM, and surprisingly – since it is very similar to JPlag – Plaggie do fairly well here.
- As for version 15 (externalisation of strings), none of the tools is insensitive, but JPlag and Marble score higher than `diff`, whereas the other three score lower.
- JPlag, Plaggie and SIM are insensitive to the refactoring from version 16 (member type to top level), whereas Marble and MOSS are clearly not. However, Marble scores higer than `diff` in both figures, and MOSS as well in the first (and almost in the second).
- Version 17 (generation of getters and setters): none of the tools are completely insensitive, but Marble and JPlag seem to have some problems (scoring lower than `diff` in Figure 2).

At first glance one might be tempted to conclude that, although there are some differences in rating, all attempts to disguise the plagiarism by the different modifications techniques would have been found by all of the tools. After all, the tools return a fairly high similarity score for

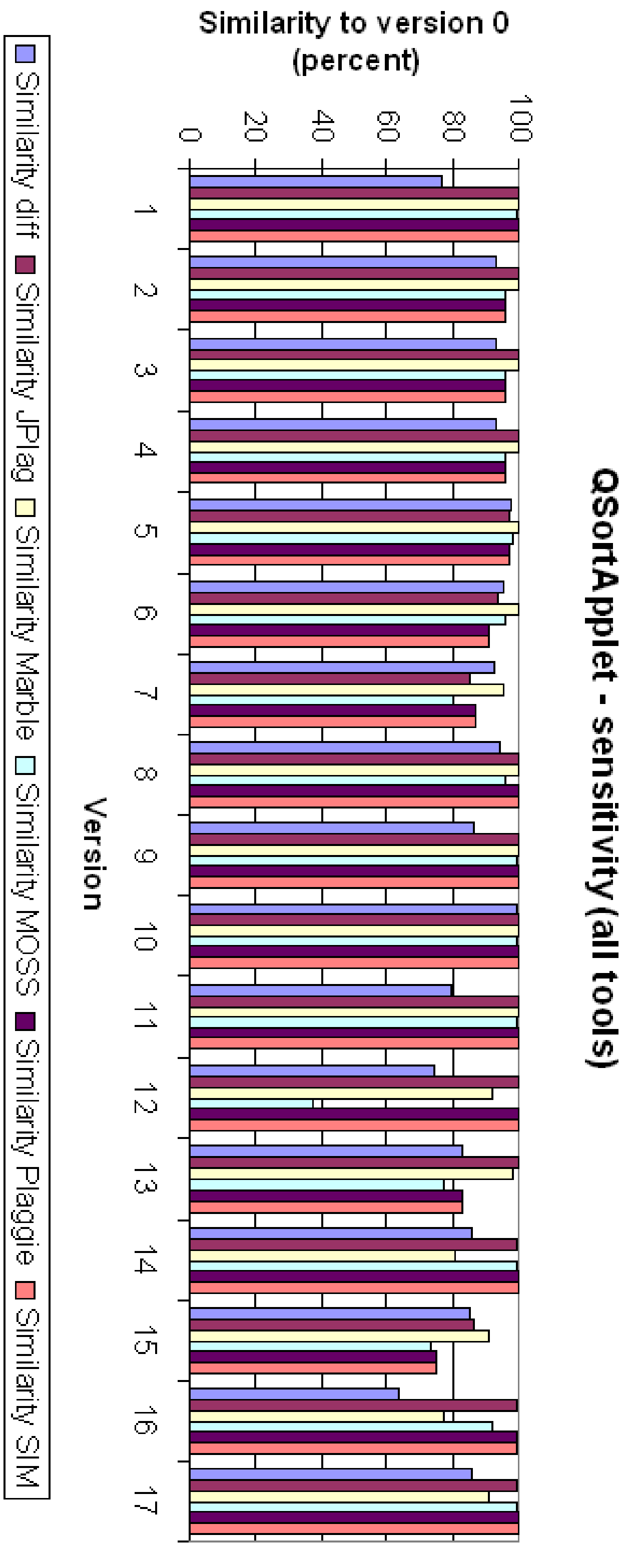

QSortApplet - sensitivity (all tools)
Similarity to version 0 (percent)
0
20
40
60
80
100
Version
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Similarity diff
Similarity JPlag
Similarity Marble
Similarity MOSS
Similarity Plaggie
Similarity SIM

Figure 1:

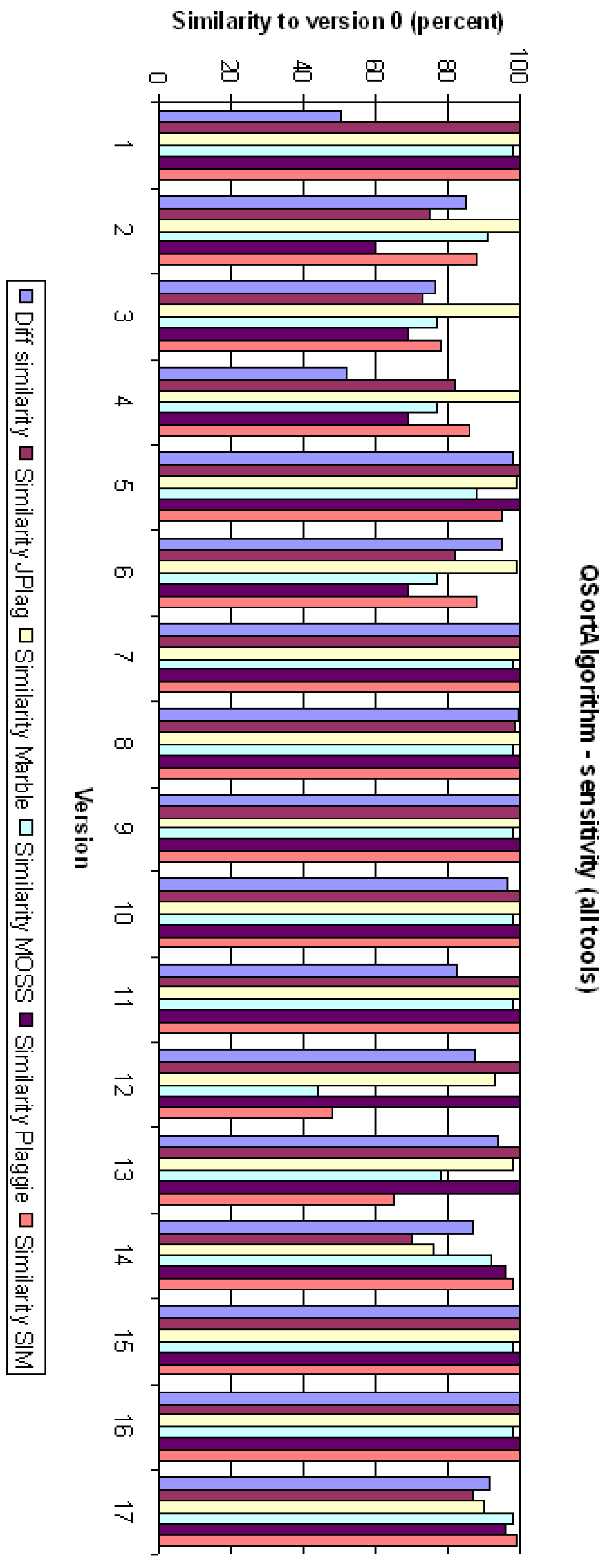

QSortAlgorithm - sensitivity (all tools)
Similarity to version 0 (percent)
0
20
40
60
80
100
Version
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Diff similarity
Similarity JPlag
Similarity Marble
Similarity MOSS
Similarity Plaggie
Similarity SIM

Figure 2:

most versions. Note, however, that the validity of the previous statement depends on how the tools rate non-plagiarism cases. For instance, it could be that the tools rate non-plagiarised pairs with similar values as we have seen in this experiment. This should be verified in an experiment. The top-n comparison in Section 5.2 can also provide some information on this aspect. However, at this moment we can say from our experience with the different tools during our experiments, that most tools do seem to rate non-similar pairs significantly lower than the scores we have seen in this sensitivity experiment.

Sensitivity to combined modifications

Here we present the set-up and results of the second part of the sensitivity experiment.

Set-up and presentation of the results
To take the previous experiment a step further, we have combined several effective modifications (in particular: 4, 6 and 11–17) to see if we can escape detection in this way. These modifications have shown to be most effective for most of the tools. Moreover, modifications 11-17 are very easy to apply since they can be performed automatically by the Eclipse IDE. The results in Figure 3 show the scores for `QSortApplet.java` and `QSortAlgorithm.java`.

Interpretation of the results
From the results in Figure 3 we observe that the similarity scores in both MOSS and SIM drop significantly for all files. Marble, JPlag and Plaggie show a smaller decrease in similarity score. To be able to detect plagiarism in a submission, only one file needs to end up high in the ranking. From experience with Marble we can say however that a score of 64 for `QSortAlgorithm.java` will end up quite high in the listing of the results.

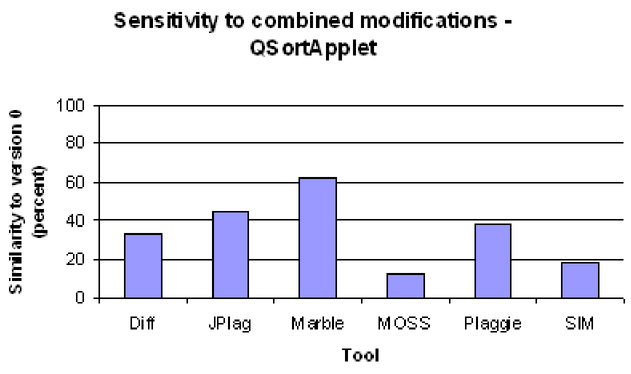
5.2 Top-n comparison

Another idea for comparing the performance of the tools was by means of comparing the top-n output of the tools. In this subsection we describe the outcomes of this experiment for $n = 10$. That is, we consider the top-10 of highest scoring pairs of submissions for each tool in turn. This gives us a minimum of 10 (since $n = 10$, and assuming that there are indeed 10 pairs that are similar enough to be detected by at least one tool) and a maximum of 50 (since we consider five tools) pairs of programs, for which we then investigate (1) whether we think that the programs are indeed highly similar, and (2) whether the similarity can be established as being due to plagiarism.

If we consider all of these highly similar pairs together, then the tools that have the most of these pairs in their top-10, and preferably then the cases that constitute actual plagiarism, are considered to be good. It should be noted though that how well a tool does in the comparison, strongly depends on the ways students have tried to hide the plagiarism. For example, if refactoring technique X is never used to hide plagiarism within our set of submissions, then all things equal, we shall consider two tools Y and Z to be equally precise, although it might be that Y can well detect the presence of masking plagiarism with X, while Z is hopeless.

When a tool lists a pair highly, and the elements of this pair turn out not to be similar, then this is a clear sign that something can be improved, particularly if other tools do not list the combination.

Set-up
For the top-10 comparison we use a different corpus than the one we used for the sensitivity experiment. At Utrecht University, we have a rather large database of submissions for a Mandelbrot assignment that have been handed in over the years (starting with course year 2007–2008, and going all the way back to 2002–2003). It has been confirmed that a few cases of plagiarism are hiding in this collection. We simply provided the tools with all these submissions, appropriately ordered in directories.



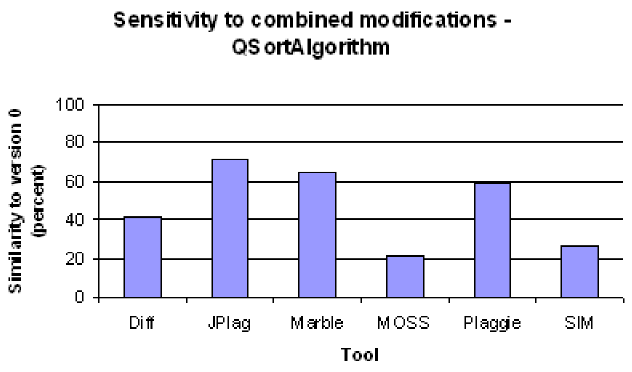


Figure 3: The diff and tool similarity with respect to version 0 for all tools, after applying modifications 4, 6, and 11–17 to QSortApplet.java and QSortAlgorithm.java, respectively

Presentation of the results
Table 4 presents the results of the top-10 comparison. For each of the five tools, we looked up the 10 pairs of submissions with highest similarity. It should be noted that this comparison was made by first converting the output files of the various tools to a comparable format. It is essential to have some tool support here, particularly if we want to redo the experiment later for other sets of submissions. The conversion tools also provide evidence that the scores from the various tools are treated uniformly, and document how the changes were made. Doing the conversion manually does not easily provide such a proof.

This amounted to a total of 28 different pairs, that are all presented in Table 4, in no particular order. The row numbers 1–28 are only added for easy reference.

Moreover, we have chosen to provide our qualification of the pair in question, instead of showing anonymizations of the submissions of which it consists. Six different qualifications appear in the table:
– the meaning of 'plagiarism' is that this pair was confirmed as a case of plagiarism
– the qualifications 'false alarm' and 'similar' have only a subtle mutual difference: a pair dubbed 'similar' contains some files that are indeed similar, but not plagiarized, whereas 'false alarm' means that the files were not even similar (and also not plagiarized)
– the qualification 'resubmission' stands for a legitimate resubmission of the same program (by the same student, in different course years, so no plagiarism)
– there are three pairs (rows 21, 25 and 28) qualified as 'same submission': indeed these consist of identical submissions (no plagiarism). Only SIM found these pairs, which is not very surprising, since SIM originally is a clone detection tool, which warrants self comparison.
– finally, 'small file' indicates that the similarity stems from the fact that the files in question just are very small (no plagiarism).

The table contains 13 relevant pairs (qualifications 'plagiarism', 'resubmission' or 'similar') that all contain pairs that are indeed highly similar and should be investigated. There are 4 irrelevant pairs (qualifications 'same submission' or 'small file') that do not belong in any top-10. The other 11 pairs ('false alarm') we would rather not see in a top-10, but we cannot just say that they should not be there.

Rows 1, 2 and 17 are linked: the pair in row 1 consists of the pair (A, C) and row 17 of the pair (B, C), where A and B are new submissions, whereas C was submitted by a different student one year previously. Row 2 represents the pair (A, B). The parse errors are caused by B being parse-incorrect.

Each column shows, for the tool mentioned in the header of that column, the scores of its top-10, plus the scores of the remaining 18 pairs – if we could find those in the results of the tool, that is.

The reason that Marble shows a score for each row (except for those marked as 'same submission', since those are only compared by SIM), while the other tools do not, is that in this particular run Marble generates a score for each pair that is compared, whereas the other tools generate scores up to a certain threshold.

Interpretation of the results
From these results we can observe that the top-10's of JPlag, Marble and MOSS are fairly similar, whereas the top-10's of Plaggie and SIM differ quite a lot from the other three. Below, we describe the top-10's in more detail.

Performance JPlag For a lot of rows, there is no JPlag score, the cause of which should be investigated. Out of the 5 cases of plagiarism, JPlag misses 2 because of parse errors (see also below) and 1 because there is just no score. The other two have fairly high scores.

Performance Marble Marble does quite well, by computing for almost all cases of 'plagiarism', 'resubmission' and 'similar' scores of 82 or higher (the two exceptions are rows 26 and 27, but still the scores there are higher than all scores for the irrelevant categories 'false alarm', 'same submission' and 'small file').

	JPlag	**Marble**	MOSS	**Plaggie**	SIM
1 plagiarism		86	46	60	30
2 plagiarism	parse error	82	49	parse error	
3 plagiarism	94	94	72	89	70
4 false alarm		43		94	4
5 false alarm		44		94	96
6 false alarm		44		94	
7 false alarm		44		94	
8 resubmission	95	92	60	92	61
9 similar	100	84	87	100	100
10 resubmission	100	100	84	100	
11 false alarm		44		94	
12 false alarm	84	56		65	
13 false alarm	84	52		69	
14 false alarm	85	57		69	
15 resubmission	100	100	73	100	100
16 false alarm		47		94	
17 plagiarism	parse error	92	71	parse error	73
18 false alarm		28			83
19 plagiarism	84	89	49	92	56
20 false alarm	86	54			
21 same submission					100
22 similar	79	94		90	43
23 small file		31			97
24 resubmission	96	89	80	97	75
25 same submission					100
26 resubmission	83	61	63	79	63
27 similar	85	75		82	49
28 same submission					100

Table 4: Top 10 comparison for Mandelbrot submissions from course year 2007/2008

Performance MOSS MOSS finds all but two (see also below) pairs that are worthy of further investigation. Sometimes the scores of these pairs are rather low, but as discussed before, this may just be a property of MOSS. In other words, MOSS's threshold score may just be lower than those of the other tools.

Performance Plaggie Plaggie returns quite a lot of high scores for cases that turn out to be 'false alarm'. Moreover (or: consequently), several cases that should be investigated end up lower in the top-10 than these 'false alarms' (especially the plagiarisms in rows 1, 3 and 19, which are on positions 414, 44 and 41, respectively). Plaggie also misses two cases of plagiarism because of parse errors (see also below).

Performance SIM SIM's scores seem rather unpredictable, see for instance the range of scores for the cases 'plagiarism', 'resubmission' and 'similar', which is 30 (plagiarism) to 100 ('similar' and 'resubmission'). Also, there is a 'false alarm' in row 5 with the high score of 96 (note, however, that this is the only 'false alarm' that was deemed important enough by SIM to get a score. This case should be investigated further.)
Positions 1, 2 and 3 from SIM's top-10 contain 'same submissions' (rows 28, 25 and 21, respectively). Row 21 actually appears also on position 9 of this top-10. Moreover, row 9 appears on positions 5 (score 100) and 6 (score 99, not mentioned in Table 4). Finally, row 23 ('small file') is on position 7. Summarizing, SIM tends to compute high similarity scores

for cases that clearly do not belong in a top-10. Mainly, this is caused by the fact that SIM sometimes compares a submission to itself. This is not so surprising, since SIM itself is a clone detection tool, which warrants self comparison. Another reason that the top-10 contains irrelevant pairs is the fact that SIM does not provide an option to exclude files below a certain size.

Missing comparison by SIM and MOSS The pair in row 10 is missing from the output of SIM, and row 22 and 27 are missing from the output of MOSS. However, both rows clearly need a closer look (although both turned out not to be plagiarism). Maybe this is a bug in the tools, or an error in the use of the tools for the experiment. This requires a closer look at both SIM and MOSS, as well as at our operation of those tools in this experiment, to see what exactly is going on.

Parse errors Parse errors in one of the submissions from rows 2 and 17 prevent these two plagiarism cases from being detected by both JPlag and Plaggie. This parse error is not caused by the tools, but by one of the submissions itself.
Fortunately both tools keep a log file in which these two cases are mentioned. However, it is possible to intentionally submit a file that is syntactically incorrect. Small errors – for example (intentionally) forgetting a closing parenthesis – are usually corrected by the assistants that inspect the submission. To ensure that this kind of trick cannot help in disguising plagiarism, one could decide to only accept compiling programs for grading.

5.3 Runtime efficiency

Efficiency can be an important issue for plagiarism detection tools. The tools are often run on a large number of submissions, and these submissions may contain multiple files. Although we have not carried out precise timing experiments we can at least report that for the experiments we did, (including an experiment covering 6 incarnations of first year programming assignment) we have encountered no severe issues with the efficiency of any of the tools. The running time range from several minutes to about 30 minutes. We do think that a running time of 30 minutes is still acceptable for the purpose of detecting plagiarism in an academic environment. However, such tools may not scale up for use in a environment with tighter time constraints. Note also that the tools offered as a service may be very fast, but if you happen to check for plagiarism when many others are as well, it may take some time before you get the results.

6 Conclusion

We have compared five plagiarism detection tools. We have compared these tools with respect to ten tool features, and we have compared the performance by a sensitivity analysis on a collection of intentionally plagiarised programs and on a set of real life submissions, comparing the performance by examining the top 10 results for each tool to the results of the others. The results of the comparison give good insight into the strong and weak points of the different tools.

Our findings from the two comparisons can be summarized as follows:

- Many tools are sensitive to numerous small changes. Refactoring number 12 is an example of such a transformation.
- All tools do well for the majority of single refactorings, but many tools score rather badly when refactorings are combined, worse than what may be obtained from simply using `diff`.
- A striking result of the top-10 comparison is that the top-10's for JPlag, Marble and MOSS are fairly similar, whereas the top-10's of Plaggie and SIM differ quite a lot from the other three.
- Along the way we have discovered a few cases where a more detailed investigation of the behaviour of the tools in question is in order.

7 Future work

To further support our findings, we may want to extend our comparison to larger and more varied sets of (real-life) data. We are fortunate to have a large database of Java submissions around. If, as part of the study, we have to make the submissions we have experimented upon available, then privacy will become an issue.

On reflection, much of the work in the performance evaluation is due to differences in the format the results are provided by the tools. We have employed various Perl scripts in order to reduce this effort, but such script are not infallible. Establishing how the output of each tool can be transformed into a format that allows easy comparison between tools, is, with the view on more tools to be considered, a useful undertaking. Furthermore, automatic collection and transformation of tool output will also allow us to more easily extend our comparison to more data.

On this note, we have left the comparison of the five tools from this paper to the tools `CodeMatch`, SID, `Sherlock` as work for the future.

A final direction for further research is to consider how the situation is for other languages. The first author has recently been involved in an investigation how to perform plagiarism detection for Haskell (still unpublished), but more importantly maybe, the Department of Computer Science is going to replace Java in its curriculum with C#.

Acknowledgements

We are very grateful to Steven Burrows for giving many helpful directions on how to improve an earlier verion of this survey. We thank Alexander Aiken for answering questions about MOSS.

References

[1] http://www.safe-corp.biz/products_codesuite.htm.

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

[3] http://www.cs.hut.fi/Software/Plaggie/.

[4] http://www.cs.su.oz.au/~scilect/sherlock/.

[5] http://genome.uwaterloo.ca/SID/.

[6] http://www.cs.vu.nl/~dick/sim.html.

[7] http://wwwipd.ira.uka.de/jplag/.

[8] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 141–142, New York, NY, USA, 2006. ACM.

[9] Christian Arwin and S. M. M. Tahaghoghi. Plagiarism detection across programming languages. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 277–286, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[10] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper.*, 37(2):151–175, 2007.

[11] Xin Chen, Brent Francia, Ming Li, Brian McKinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.

[12] Jurriaan Hage. Programmeerplagiaatdetectie met marble. Technical Report UU-CS-2006-062, Department of Information and Computing Sciences, Utrecht University, 2006.

[13] Edward L. Jones. Metrics based plagarism monitoring. In *CCSC '01: Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 253–261, , USA, 2001. Consortium for Computing Sciences in Colleges.

[14] Thomas Lancaster and Fintan Culwin. A comparison of source code plagiarism detection engines. *Computer Science Education*, 14:2:101 – 117, 2004.

[15] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD06*, pages 872–881. ACM Press, 2006.

[16] Lefteris Moussiades and Athena Vakali. Pdetect: A clustering approach for detecting plagiarism in source code datasets. *The computer journal*, 48(6), 2005.

[17] Cunningham P. and Mikoyan A.N. Using cbr techniques to detect plagiarism in computing assignments. In *working papers of 1st. European Workshop on Case-Based Reasoning*, pages 178–183, November 1-5 1993.

[18] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics, 2000.

[19] Deepak Rao et al Sanjay Goel. Plagiarism and its detection in programming languages. Technical report, Department of Computer Science and Information Technology, JIITU, 2008.

[20] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003.

[21] Michael Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

A Sensitivity charts

This appendix contains, for each tool, a chart showing the result of the tool compared to the result of `diff`, for both `QSortApplet.java` and `QSortAlgorithm.java`.

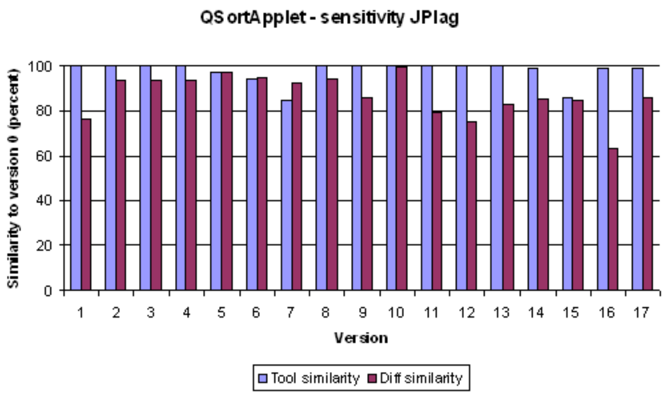


Figure 4:

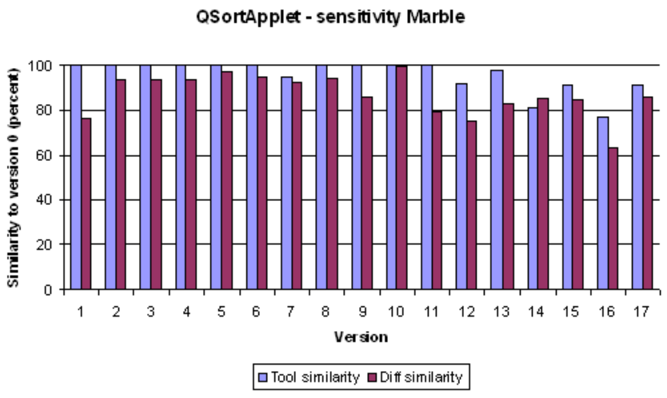


Figure 5:

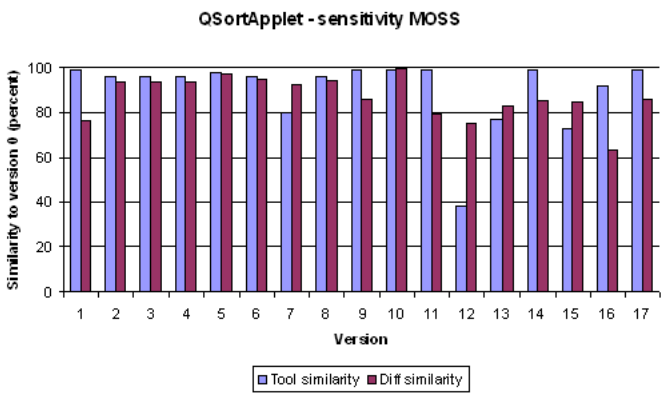


Figure 6:

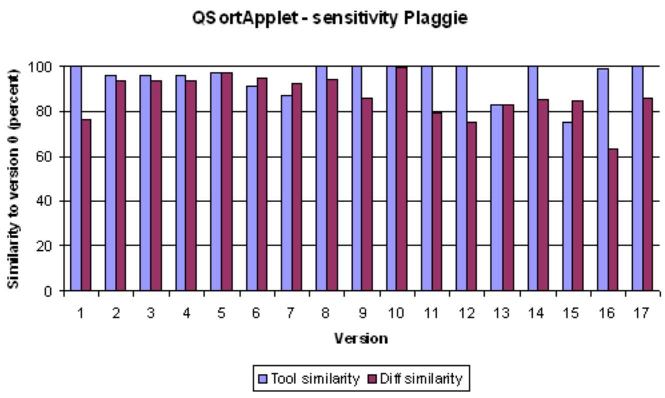


Figure 7:

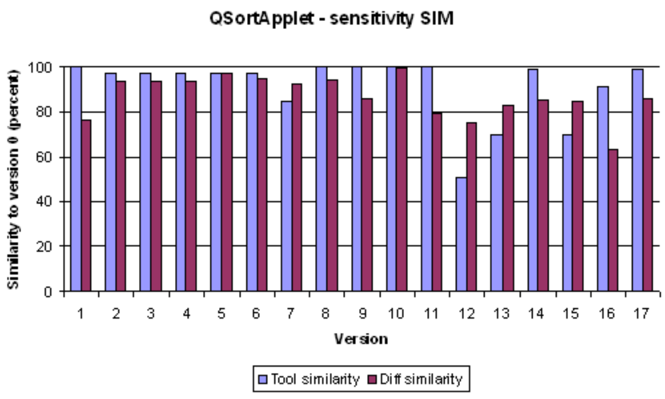


Figure 8:

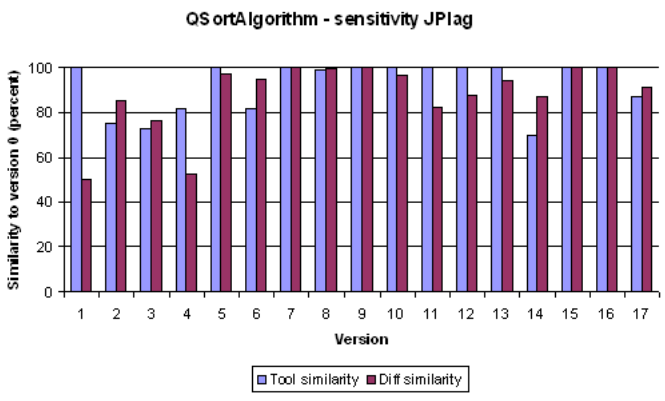


Figure 9:

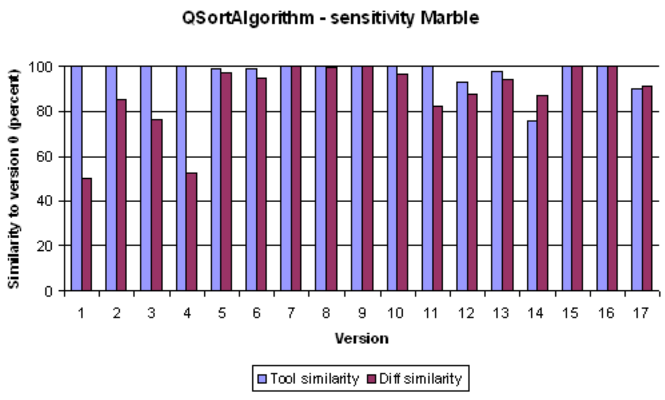


Figure 10:

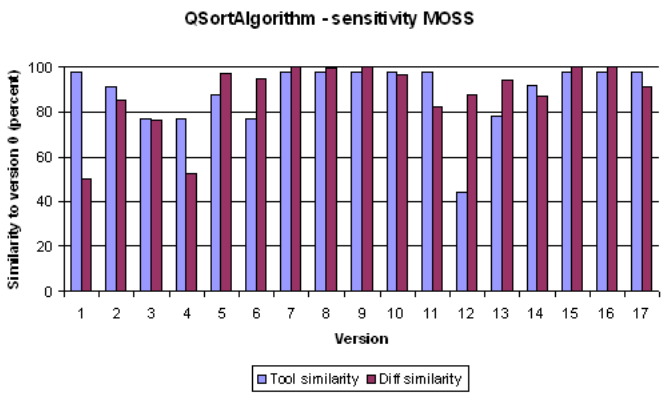


Figure 11:

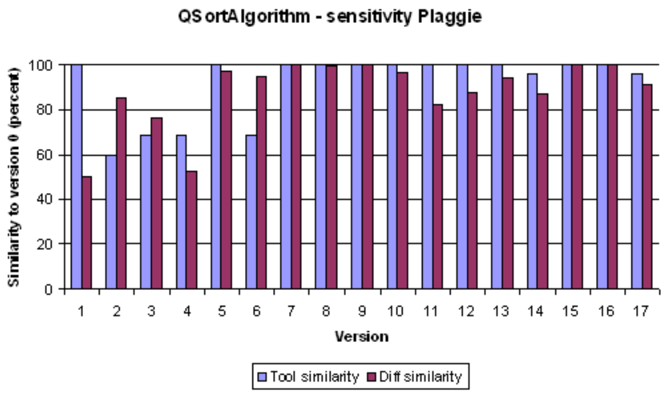


Figure 12:

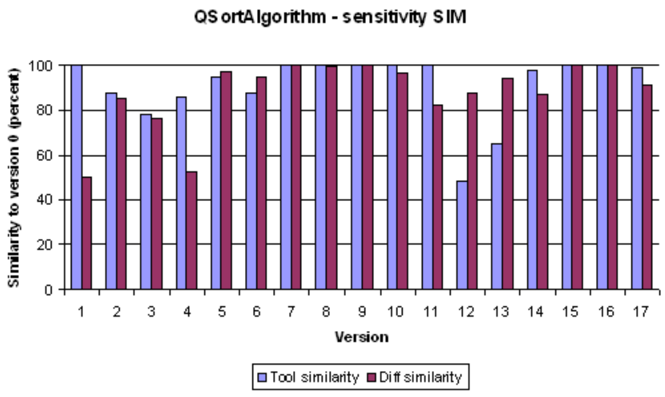


Figure 13: