Lessons learnt in applying automated code plagiarism detection in an introductory programming module[1]

Bertram Haskins, Nelson Mandela Metropolitan University, South Africa
Vreda Pieterse, University of Pretoria, South Africa

ABSTRACT

This paper investigates automated code plagiarism detection in the context of an introductory programming module. Three methods for detecting plagiarism are compared to determine whether these systems yield differing results. These methods are the use of MD5 hashes and the application of two plagiarism detection systems, namely MOSS and NED. The same set of solutions to the same problem was evaluated, using each of the three methods. This set was selected as a representative sample as it was characteristic of most other data sets submitted by students in the introductory programming module over the course of four years. The discrepancies in the results obtained by these detection techniques were used to devise guidelines for effectively detecting code plagiarism.

Keywords: programming, plagiarism, detection methods

1. INTRODUCTION

An academic course requires interaction between the instructor and the student. We subscribe to the learning theory of Gibbs (1988) that doing is an effective way of learning. We believe that students learn best by writing their own code. It increases their understanding and if assessment is timely, it allows the instructor to gauge their level of understanding. Unfortunately, students often submit work done by other students because they are more interested in getting marks than in learning and gaining programming experience. These submissions constitute acts of plagiarism, which is the bane of many lecturers.

This paper critically reflects on the lessons learnt in the application of automated techniques for the detection of plagiarised code. This study applies plagiarism detection as a tool for identifying the students who plagiarise. Knowledge about whether homework was plagiarised could be used to ensure that the marks awarded to students are a true reflection of their understanding, and also to identify individuals who may benefit from an intervention.

This paper shares the experience and reflects on the lessons learnt when applying different techniques for detecting plagiarism. The aim is to describe the insight gained in a way which other academics may find useful. These insights might assist others to extend their current teaching practices to include strategies for the identification of plagiarised code.

1 Date of submission 30 August 2015
Date of acceptance 29 January 2016

The next section (2) sets the scene for our discussion. In Section 3.1 we discuss the possible reasons for acts of plagiarism and give an overview of how plagiarism can be addressed in academic institutions. This discussion highlights the need for an automated process for the identification of plagiarised assignments. Section 3.2 explains some of the methods and systems that could be employed to detect code similarity.

The main objective of this paper, namely to provide guidelines that may improve the accuracy of the detection of code plagiarism, is based on our observations of the results obtained from applying three existing plagiarism detection techniques to a selected data set. This data was generated in our introductory programming module. Section 4 gives an overview of the classroom situation which inspired this study and describes the data set used in our investigation. Section 5 discusses the observations made when applying the different checks for code similarity to the selected data. The lessons learnt from this investigation are condensed in a series of guidelines in Section 6. Section 7 summarises the observations and concludes by positioning the application of techniques for detecting code plagiarism as part of a broader educational responsibility.

2. PROBLEM STATEMENT

Assessment should enhance the students' learning experience and measure their competence accurately. Recent technological advances which enable students to cheat, threaten the purpose of assessment. To counter this threat, plagiarism detection has become commonplace in academic institutions and has sparked the development of sophisticated tools to detect cheating.

Various factors may prevent institutions from using the proprietary tools or other tools that are publically available to detect copied work. One such factor is the varying results of the detection techniques contrasted in this study. Another factor is that software development houses tend to employ a closed development model which does not permit external access or modification to their proprietary code. A third factor might be the intellectual property constraints on the code that students have written and on student information. These may restrict permission to submit the student's homework solutions to a plagiarism detection service. These and other factors may make the local development of an automated plagiarism detection technique the most viable option for most information technology and computer science departments.

The purpose of this study is to propose guidelines which could be used when developing a plagiarism strategy for the identification of copied code in an introductory programming module. Educators should be able to combine the insights gained from these guidelines with their local subject knowledge so that they can devise a plagiarism detection technique which is adapted specifically to their needs.

3. BACKGROUND

This section is an overview of contemporary views on plagiarism and the techniques used for automatically identifying possible cases of plagiarism. It gives insight into the underlying reasons why students copy assignments and discusses several of the techniques used for identifying copies.

3.1 Views on Plagiarism

Different communities have varying views on plagiarism. Many practices such as working together and reusing code may be acceptable in a professional context but are considered unacceptable in an academic context (Simon & Sheard, 2015). On the one hand, professors and administrators regard plagiarism as a serious academic crime, an ethical transgression or even a sin against an ethos of individualism and originality. Students, on the other hand, revel in sharing, in multiplicity and in accomplishment at any cost (Blum, 2011). It is likely that ignorance of which behaviours constitute plagiarism is widespread among students and staff alike (Gullifer & Tyson, 2014).

Students may justify official punishable behaviour on a moral basis which focuses on values such as friendship, interpersonal trust and good learning. In situations where students are aware that the likelihood of being caught is minimal, they may feel that they are justified in copying since everyone else is doing it (Selwyn, 2008). Louw and Pieterse (2015) identified several reasons why students copy, such as a lack of ability to write their own version, a lack of time to finish a task on time, a desire for better marks, a lack of interest and a distrust of the automatic assessment of their code. They maintain that lack of ability is most often the reason why a student would copy someone else's work. Kyrilov and Noelle (2015) also find that students are likely to cheat and disengage if their work is automatically assessed.

Various ways of dealing with plagiarism have been proposed. What is clear, however, is that it is an academic's duty to educate students about cheating, plagiarism and intellectual property rights (Granzer, Praus & Balog, 2013). This simple act of education may prevent plagiarism by fostering an attitude that condemns plagiarism (Briggs, 2003; Gibson, 2009). Other prevention strategies include emphasising low stakes formative assessment (Macdonald & Carroll, 2006), punishing students who cheat and expecting students to work in conditions where it is difficult to cheat (Schoeman & Pieterse, 2004).

When a student has copied code, the authors of this paper interpret it as an indicator that the student needs intervention. Such an intervention is a response to the reason for copying. Depending on the reason, intervention may provide training to enhance the student's understanding of the work. Furthermore, an intervention may provide guidelines for better time management or motivate the student to pursue goals aimed at acquiring knowledge, not aimed solely at achieving better academic results.

3.2 Detecting Plagiarism

There are various automated systems for the detection of plagiarism in prose, such as Turnitin (2015) and PaperRater (2015). These systems allow an input text to be scrutinised and compared to existing sources to determine how much of the document has been plagiarised. These plagiarism detection tools tend to be inaccurate, however, if used for the detection of plagiarised documents written in a programming language. As programming languages are more rigid than natural languages; they also contain many acceptable cases of repetition. This may make the system mistakenly identify standardised, repetitive programming structures, shared among a large number of files, as an indicator of plagiarism (Acampora & Cosma, 2015). For this reason, such systems are not feasible solutions for detecting copied code.

A rudimentary way of identifying copied code, applied by Pieterse (2014), is the comparison of the *m*essage-*d*igest 5 (MD5) hashes of documents. The MD5 algorithm is a widely used cryptographic hash function which produces a 16-byte value for any input. When used for observing code similarity, the input to the function is the entire student program. MD5 is believed to produce a unique value for any given file (Rivest, 1992) and is commonly used to verify data integrity. If two documents produce the same MD5 hash, it can be assumed that the contents of the documents are identical in every respect. This technique is only useful for identifying exact copies. More advanced techniques are required to identify sections of code which are similar, but not exact matches.

Software for detecting code similarity may be broadly classified into two categories, namely attribute-counting and structure-metric systems (Chen, Francia, Li, McKinnon & Seker, 2004). Attribute-counting systems track certain attributes, such as repetitive tokens, in an input text, in order to devise a profile of the input text. Structure-metric systems extract and compare representations of the overall program structure. These two categories encompass a wide variety of systems for detecting similarity in code. These systems include: Measure of Software Similarity (MOSS) (Aiken, 1994), JPlag (Prechelt, Malpohl & Philippsen, 2002), Plaggie (Ahtiainen, Surakka & Rahikainen, 2006), the Software Integrity Diagnosis system (SID) (Chen et al., 2004) and the n-gram and Edit Distance plagiarism detector (NED) (Haskins, 2014). Many

of these systems hide the details of their underlying algorithms to avoid a situation where students learn how to circumvent the plagiarism detection process.

MOSS is a widely used free Internet service hosted by Stanford University for automatically detecting similarity between programs, in languages such as C, C++, Java, C#. It has been publicly available since 1997. As part of its implementation, it uses an algorithm for document fingerprinting (Schleimer, Wilkerson & Aiken, 2003). Bowyer and Hall (1999) discuss an implementation of the MOSS system which evaluates documents at various levels. The plagiarism indicators used in the evaluation include the number of tokens matched, the number of lines matched and the percentage of overall overlap between two programs. Their approach is not unique, since Joy and Luck (1999) present a study involving an incremental approach for comparing the code of two programs at three different levels of pre-processing. The purpose is to create feature sets which serve as input into a neural network and clustering algorithms.

JPlag is a publicly available web service, which compares input programs as pairs to compute a similarity value and highlight regions of similarity. It outputs plagiarism reports as a set of HTML pages. As part of its plagiarism detection process, it converts each program into tokens (substrings) and then attempts to match these tokens to substrings in another program. It can apply this technique to program a source code written in Java, Scheme, C or C++.

Plaggie is a stand-alone Java-based system used for detecting code plagiarism in Java code. The use of Plaggie is governed by a GNU licence and, as such, it is freely available for download. It was created to address an initial shortfall in JPlag in which it was unable to distinguish or exclude the pre-written code given to students as part of an exercise. JPlag has since addressed this shortcoming.

SID makes use of an information-based metric to measure the amount of information shared between two sequences. Input programs are broken down into tokens and an algorithm is applied to calculate the shared information among the tokens. Then program pairs are ranked according to the amount of shared information, referred to as their similarity distances. SID currently accepts programs written in Java or C++, but may be expanded to include other languages if a parser for the language is written.

NED was developed, tested and validated on first-year C# programming assignments in an introductory programming course (Haskins, 2014). The software employs three means of performing comparisons, namely exact match, n-grams and inverse edit distance. The n-grams are a means of dividing a word or sentence into smaller, overlapping sections or tokens. The n signifies the length of an individual section. These individual sections may then be compared with the sections of another word to determine similarity. This may include a process as simple as calculating a frequency measure of the n-grams in an input statement and then comparing these frequencies to a reference (Cavnar & Trenkle, 1994).

Other studies measure the number of changes necessary for converting one statement into another, referred to as the edit distance (Masek & Paterson, 1980; Wagner & Fischer, 1974). Haskins and Botha (2014) apply a combination of n-grams and edit distance as a means of gauging the similarity between words, for the sake of automatically normalising text to resemble English more closely. They apply the normalisation to text found on a mobile mathematics tutoring platform.

NED was created to compare the content of files at a character level with no direct regard for underlying language structure by applying n-gram comparisons and edit distance measurements. Therefore it is applicable not only to files containing C# code, but also to those containing content in languages such as C++ and Java. A decision was made to investigate the applicability of three of the techniques discussed in this section to the selected introductory programming module. The techniques were chosen because they could process C++ code and their use was free of charge. Moreover two kinds of plagiarism, namely direct matches and code similarity, had to be addressed. MD5 hashes were chosen for comparisons as

the hashes are a simple method of identifying direct copies of assignments. MOSS was selected as it is one of the best-known systems for detecting plagiarism (code similarity) and is freely available. Lastly, NED was selected as it is a fairly new system which is being developed at one of the universities involved in this study. This allowed the insights gained during this study to be applied to the improvement of the plagiarism detection capabilities of NED. The next section discusses the approach taken in this study when contrasting the three selected automated plagiarism detection techniques.

4. SITUATIONAL OVERVIEW

This section describes the data used in our investigation. The first two subsections describe the classroom situation where the data was created and how the specific data set for the investigation was selected. The concluding subsection describes the characteristics of the data, based on a manual scrutiny of a sample drawn from the data. This was needed to support the observations described in Section 3.

4.1 Scenario

The investigation was into the plagiarism practices of the students in an introductory programming module, with large student enrolment numbers. The module introduces imperative programming using the C++ programming language. Table 1 shows the student numbers in this module for 2012 to 2015.

Table 1:
Enrolment Figures

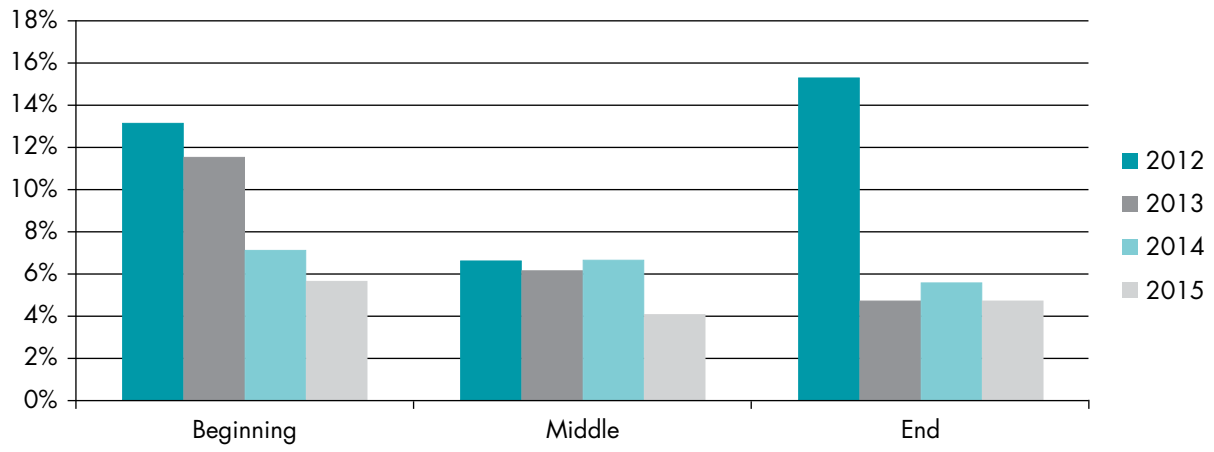
Year	2012	2013	2014	2015
Number of students	554	593	679	750

The MD5 hashes of the student programs were used to identify exact copies. This test was repeated three times a year; at the beginning, around the middle and at the end of the semester. Figure 1 shows the results of these comparison tests, which were conducted from 2012 to 2015.

During 2012 no effort was made to follow up the culprits, whereas in 2013 the students who were identified as having participated in this practice were called in for an intervention.

The graph in Figure 1 shows that the practice of uploading exact copies of code stabilised after the lecturing staff started to intervene when students were identified as being involved in copying the work of others.

Figure 1:
Percentage of students who uploaded exact copies



4.2 Selecting a data set

The variation in coping behaviour is lowest in the middle of the semester. Accordingly, a decision was made to use the submissions of a task in the middle of the semester in 2014. This set serves as a representative sample of student submissions in the representative introductory programming module. The basis for this selection was that the number of submissions in this task was closest to the average number of submissions for the different tasks. This data set contains 452 C++ solutions to the same problem.

4.3 Characteristics of the data set

The characteristics of the data set should be known if the investigation is to be meaningful. To this end, the characteristics of the data set were determined by manually securitising a random sample of 50 programs drawn from the set.

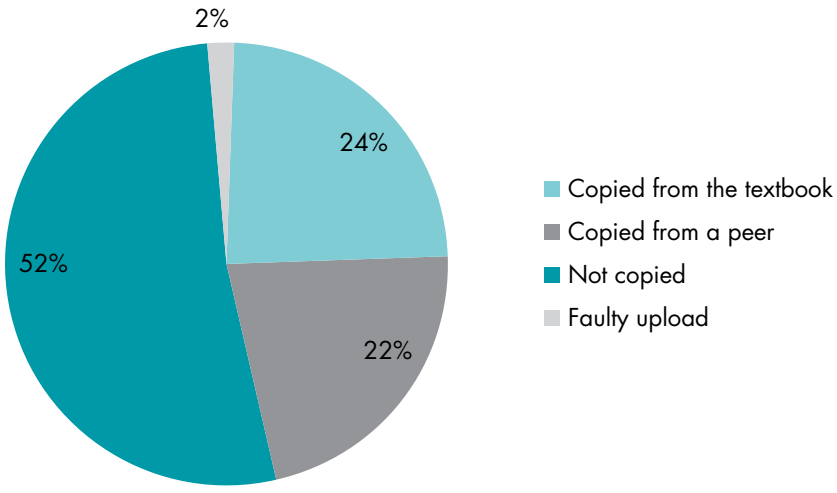
Two researchers independently scrutinised and analysed the submissions in the sample. Their findings were compared and a collaborative final decision was made in each case where there were differences in individual judgements.

Figure 2 shows the characteristics of the data in the sample. One of the 50 submissions was invalid, because it was the solution to a different problem. The student might have inadvertently submitted this solution in the wrong place. Most of the solutions (52%) were classified as not copied as no evidence was found in these solutions to justify the notion that they were not the student's own work.

Eleven (22%) of the submissions in the sample were identified as being copied by or from peers. Four pairs of students and one group of three students submitted almost identical solutions. The textbook contains a complete solution to the problem but uses advanced techniques which had not yet been taught at the time when the problem was given to the students. Nonetheless 24% of the submissions in the sample had obviously been copied from the textbook.

Based on the random nature of the sample, it can be assumed that the data used in the comparison of plagiarism detection techniques, as described in the next section, would demonstrate characteristics similar to those of the sample described here.

Figure 2:
Characteristics of the data in the sample



5. COMPARISON OF RESULTS

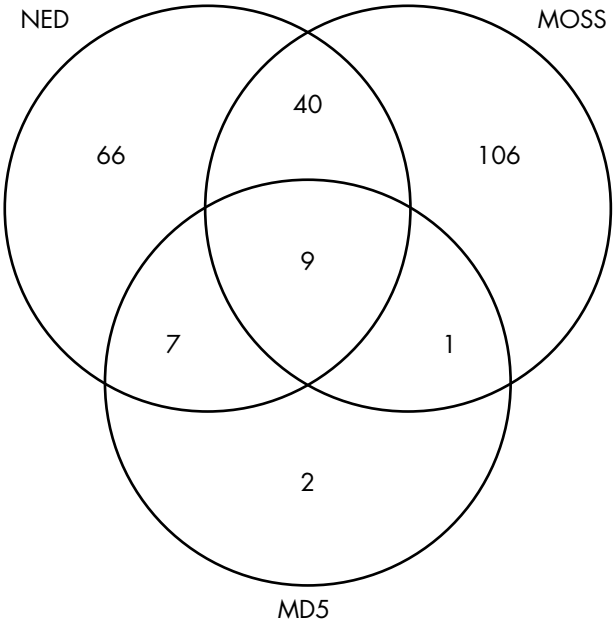
The three plagiarism detection techniques selected as described in Section 2 were used to process the data set discussed in Section 3. Their results are compared, not to prove the viability of these systems, but to highlight the differences in their detection capabilities. The examples highlighted in this fashion are used in Section 5 to propose a set of guidelines for the detection of code plagiarism.

Students were allowed to submit their solutions multiple times, therefore we were able to compare each new submission with all their other submissions, including the previous versions of their submissions. The older versions were only compared, however, when using the MD5 technique. The other two techniques were used for comparing the final submissions only.

When comparing the MD5 hashes of programs, a total of 18 student pairs involving 19 individuals were identified as possible offenders. The MOSS system reported that a total of 250 pairs, involving 156 individuals, were potential cases of plagiarism. NED identified the assignments of 122 individual students as possible cases of plagiarism.

The assignments identified by the three techniques were not the same because of differences in their approaches. The Venn diagram in Figure 3 shows the number of assignments that each of the techniques identified as possible copies. The areas of overlap signify the assignments that were identified by multiple approaches.

Figure 3:
Overlap between individual assignments



The following section provides some insight into the causes of the discrepancies in the various detection techniques.

6. LESSONS LEARNT

This section probes and reflects on the reasons for the variances in the results of the three techniques. The insights gained from this reflection are condensed into a series of guidelines for the adoption of an automated code plagiarism detection technique in an introductory programming module.

6.1 Insights

The various approaches to plagiarism detection have varying strengths and weaknesses. Each can detect copies which the other approaches cannot detect. When combining the assignments that all three approaches identified as possible copies, 231 individual assignments were identified as possible copies. This constitutes 51% of the 452 submitted assignments.

The initial analysis of the 50 assignments in the sample, shown in Figure 2, shows that approximately 46% of the submitted assignments are likely to be either copied from the textbook or from a peer. The results obtained from the automated plagiarism detection show only a 5% deviation from the analysed sample. These results are promising in that an automated plagiarism detection approach which combines the strengths of all three investigated approaches may achieve a level of plagiarism detection similar to a manual approach.

In some cases, the plagiarism is fairly clear-cut, as the assignments are exact copies of one another. Unaltered copies tend to be the least frequent, especially in the later stages of a module. Most cases of plagiarism involve making subtle changes to assignments, possibly in an attempt to fool the program or the person performing the comparison.

The findings obtained from scrutinising the possible cases of plagiarism, identified by the automated detection methods, indicate that the students used four main approaches in an attempt to hide their acts of plagiarism. These approaches are the renaming of identifiers, the addition or removal of variables, a change in the order of statements or the use of alternate forms of statements.

In total, 106 students submitted tasks that MOSS identified as being possible copies but had a similarity lower than 90,0% according to NED. In all the cases where copies were identified as possible breaches by MOSS but not by NED, the students had changed the variable names but made limited changes to the rest of the code. A single submission had a similarity of 98,5% with the entry in one of the pairs. On closer investigation, this less-than-perfect similarity arose because the student had added 'y' to a list of vowels in the code.

To avoid the detection of duplicate assignments, the students also frequently changed the order of statements. This is a simple but effective change, as many direct text comparison techniques would be completely fooled by this reordering. In 66 cases, NED was the only process that identified the cases of plagiarism. On closer scrutiny, it seems that almost all of these assignments were in fact copies of one another but with slight modifications intended to ensure that they were not exact copies of one another. These modifications included making changes to the case or adding or removing white spaces, such as tabs.

Based on the lessons learnt in our experimentation with automated code plagiarism detection techniques, a set of guidelines was devised to aid in the adoption of such techniques in any module similar to our introductory programming module.

6.2 Proposed Guidelines

Martins, Fonte, Henriques and da Cruz (2014) define various types of plagiarism, such as exact copies and changes in comments. We realised that the forms of plagiarism found in this study correspond with the types set out by Martins et al. (2014). Accordingly, we combined our observations from the previous section with the plagiarism types of Martins et al. (2014) to compile a set of guidelines for the adoption of automated plagiarism detection techniques in an introductory programming module. These guidelines are listed in Table 2. These guidelines have a twofold purpose: firstly, to advise the lecturers who are in a position to develop their own code plagiarism detection software; and secondly, to guide other lecturers to

determine which features to look for in automated code plagiarism detection software. Lecturers could also use these guidelines to pre-process assignments before handing them to a human marker or submitting them to an automated code plagiarism detection tool. This may reverse some of the actions students are likely to take to avoid detection.

Table 2:
Guidelines for the automated detection of plagiarism

Number	Guideline
1	Examine metadata
2	Check for direct matches
3	Remove unnecessary white space
4	Ignore non-contributing content
5	Address comments
6	Generalise the input files
7	Prepare for out-of-sequence statements
8	Be aware of the effect of assignment length

The first guideline, 'Examine metadata', refers to the fact that many students simply submit exactly the same file. These submissions require no further examination of the file content if a match can be made when comparing their metadata, such as the creation date. Although this seems a fairly simple comparison; and one which may quite easily be modified, the results have shown that a fair number of the copied assignments can be identified by such a simple comparison, and consequently this avoids using the more expensive comparison techniques.

The second guideline, 'Check for direct matches', addresses assignments which have been copied among a group of students, opened and then saved again before submission. Such assignments can be identified by performing a simple direct string comparison or by comparing MD5 hashes.

Adding extra white space, such as tabs, spaces and new lines, is a simple way to fool direct string comparisons. To address Guideline 3, we suggest removing all unnecessary white space characters and condensing multiple spaces to single spaces before performing comparison checks.

In most programming assignments, in any language, certain portions of the assignments will occur in every submitted assignment. These may be forward library declarations or even specific method names such as Main. Guideline 4 attempts to address this content by allowing for the creation of a mechanism which ignores these types of frequently occurring non-contributing content. To this end, the lecturer can compile a list of statements which may safely be ignored in the comparison process.

The inclusion of comments as a documentation feature in the program code is a common feature in programming languages. These statements may influence the similarity of files. Therefore, Guideline 5 suggests that comments should be addressed before comparing assignments. Comments may include valuable information which ought to be compared. We suggest an approach in which the comments in a document are separated from the code. This would allow only the code to be compared and then the comments could be analysed separately.

Many of the assignments identified by MOSS, but not by NED, were instances where the assignments were copies in which only the variable names had been modified. Guideline 6 therefore suggests that a generalisation step may be useful in which all variable declarations and usages are converted to simple instances of key words such as VARIABLEDEC or VARIABLEUSE. A similar process could be applied to method names. Removing this level of variance would facilitate a much better comparison of text, as changing the variable names is one of the simplest ways for students to modify their assignments.

An easy way for students to attempt to avoid the detection of plagiarised code is to change the sequence of the content in the file, i.e. by swapping the sequence of methods. This change does not affect the function of the program, but could have a dramatic effect on the metrics of the detection software. To overcome this, Guideline 7 suggests that whichever technique(s) is(are) chosen for performing the comparison should take into account that the content of the files may be the same, but jumbled. Lecturers may counteract this by asking students to submit assignments which follow a specific sequence in method and structure. This may also help lecturers to enforce specific programming practices.

The final guideline, 'Be aware of the effect of assignment length', emphasises that the shorter an assignment, the greater the overlap between the content of individual assignments. This is especially problematic in an introductory programming module because the assignments are frequently fewer than 100 lines in length. Addressing this issue would require user intervention or a form of supervised learning to determine the optimal threshold of overlap among assignments.

7. CONCLUSION AND FUTURE WORK

The main objective of this study was to propose a set of guidelines to improve the accuracy of the detection of code plagiarism. To this end, three different automated techniques to identify plagiarised assignments were described. These techniques are the use of an MD5 hashing algorithm, a well-known plagiarism detection system called MOSS and a recently developed system called NED.

The comparison of MD5 hashes of programs can identify the lowest number of copied assignments and is only useful for identifying assignments in which the content is an exact match. Both MOSS and NED are very useful for identifying possible cases of plagiarism. MOSS can identify copied assignments even when variable names have been modified. NED can identify assignments in which the structure was modified by the addition of white space, changes in comments and the reordering of statements. As MOSS allows boilerplate code to be ignored, better results may be obtained from MOSS by tweaking its settings. The results obtained from NED may also be optimised by determining the optimal threshold at which the similar programs are likely to be copies on a per-data-set basis.

The results of both MOSS and NED may have been influenced by the short length of the submitted assignments and by the fact that many students seemed to have worked in groups. Group work often means that many of the submitted assignments contain only slight cosmetic changes.

We acknowledge that the detection of possible copies is only one of the activities needed to fulfil our responsibility to educate students about cheating, plagiarism and intellectual property rights. The technical guidelines we have compiled have the potential to increase the accuracy of automatically detecting the plagiarism of code, which in turn is likely to enhance the effectiveness and quality of our teaching regarding these matters.

Furthermore, the lessons learnt from comparing the results of the various automated detection techniques have highlighted the discrepancies in the results obtained from these techniques. The accuracy of plagiarism detection could be increased by applying the guidelines either in a technical manner, i.e.

analysing or modifying the submitted assignments, or by using the guidelines to restructure parts of the teaching process, such as adding extra requirements for, or restrictions on assignment specifications. This restructuring might also result in students focusing on their own assignments, which may decrease the number of plagiarised assignments submitted.

Future work in this domain will focus on the optimisation of our existing plagiarism detection techniques and classroom practices in order to develop a reflective view of the plagiarism rate before and after the implementation of the guidelines devised in this study.

REFERENCES

Acampora, G. & Cosma, G. (2015) 'A Fuzzy-based approach to programming language independent source-code plagiarism detection' *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)* pp.1-8.

Ahtiainen, A., Surakka, S. & Rahikainen, M. (2006) 'Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises' *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006* pp.141-142. New York: ACM.

Aiken, A. (1994) *MOSS: a system for detecting software plagiarism*. http://theory.stanford.edu/ aiken/ moss/ (Accessed 12 January 2013).

Ashworth, P., Bannister, P. & Thorne, P. Students on the Qualitative Research Methods Course Unit. (1997) 'Guilty in whose eyes? University students' perceptions of cheating and plagiarism in academic work and assessment' *Studies in Higher Education* 22(2) pp.187-203.

Blum, S.D. (2011) *My word!: Plagiarism and college culture*. New York: Cornell University Press.

Bowyer, K.W. & Hall, L.O. (1999) 'Experience using "MOSS" to detect cheating on programming assignments' *Frontiers in Education Conference, 1999. FIE'99*. San Juan, Puerto Rico. *29th Annual IEEE* 3 pp.13B3-18.

Briggs, R. (2003) 'Shameless! Reconceiving the problem of plagiarism' *The Australian Universities*' Review 46(1) p.19.

Cavnar, W.B. & Trenkle, J.M. (1994) 'N-gram-based text categorization' *Proceedings of the 3rd Annual Symposium on Document Analysis and Information Retrieval* pp.161-175. Las Vegas, US.

Chen, X., Francia, B., Li, M., Mckinnon, B. & Seker, A. (2004) 'Shared information and program plagiarism detection' *IEEE Transactions on Information* Theory pp.1545-1551.

Gibbs, G. (1988) *Learning by Doing: A Guide to Teaching and Learning Methods*. Far Eastern University Publications: Manila.

Gibson, J.P. (2009) 'Software reuse and plagiarism: a code of practice' *ACM SIGCSE Bulletin* 41(3) pp.55-59.

Granzer, W., Praus, F. & Balog, P. (2013) 'Source code plagiarism in computer engineering courses' *Journal on Systemics, Cybernetics and Informatics* 11(6) pp.22-26.

Gullifer, J.M. & Tyson, G.A. (2014) 'Who has read the policy on plagiarism? Unpacking students' understanding of plagiarism' *Studies in Higher Education* 39(7) pp.1202-1218.

Haskins, B. (2014) 'Utilising n-grams and edit distance as a means of identifying copied programming assignments' *Proceedings of the 43rd annual conference of the Southern African Computer Lecturers' Association* (SACLA) pp.30-35. Port Elizabeth, South Africa

Haskins, B. & Botha, R. (2014) 'A mixed-method approach to normalising Dr Math microtext' *Proceedings of the International Conference on Computational Science and Technology (ICCST)* Malaysia pp.1-6.

Joy, M. & Luck, M. (1999) 'Plagiarism in programming assignments' *IEEE Transactions on Education* 42(2) pp.129-133.

Kyrilov, A. & Noelle, D.C. (2015) 'Binary instant feedback on programming exercises can reduce student engagement and promote cheating' *Proceedings of the 15th Koli Calling Conference on Computing Education Research* pp.122-126. New York: ACM.

Louw, D. & Pieterse, V. (2015) 'Dealing with Plagiarism in Introductory Programming' *International Conference on Computer Science Education Innovation & Technology (CSEIT). Proceedings* p.4. Global Science and Technology Forum.

Macdonald, R. & Carroll, J. (2006) 'Plagiarism – a complex issue requiring a holistic institutional approach' *Assessment & Evaluation in Higher Education* 31(2) pp.233-245.

Martins, V.T., Fonte, D., Henriques, P.R. & da Cruz, D. (2014) 'Plagiarism detection: A tool survey and comparison' In A.S. o. Maria João Varanda Pereira José Paulo Leal (Ed.) *3rd Symposium on Languages, Applications and Technologies (SLATE'14)* pp.143-158. OASICS Schloss Dagstuhl.

Masek, W.J. & Paterson, M.S. (1980) 'A faster algorithm computing string edit distances' *Journal of Computer and System Sciences* 20(1) pp.18-31.

PaperRater. (2015) *PaperRater – Home*. http://www.paperrater.com/ (Accessed 24 February 2016).

Pieterse, V. (2014) 'Decoding code plagiarism' *Proceedings of the 43rd annual conference of the Southern African Computer Lecturers' Association (SACLA)*, Port Elizabeth, South Africa pp.36-42.

Prechelt, L., Malpohl, G. & Philippsen, M. (2002) 'Finding plagiarisms among a set of programs with JPlag.' *Journal of Universal Computer Science* 8(11) p.1016.

Rivest, R. (1992) 'The MD5 message-digest algorithm' *Internet Request for Comments: 1321*. https//:www.irt.org/rfc/rfc1321.htm (Accessed 31 August 2016).

Schleimer, S., Wilkerson, D.S. & Aiken, A. (2003) 'Winnowing: Local algorithms for document fingerprinting' *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* pp.76-85. New York, NY, USA: ACM.

Schoeman, I.L. & Pieterse, V. (2004) 'Managing programming assignments in the computer science classroom' *Proceedings of the 34th annual conference of the Southern African Computer Lecturers' Association (SACLA)* pp.50–59.

Selwyn, N. (2008) 'Not necessarily a bad thing…: a study of online plagiarism amongst undergraduate students' *Assessment & Evaluation in Higher Education* 33(5) pp.465-479.

Simon & Sheard, J. (2015) 'Academic Integrity and Professional Integrity in Computing Education' *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* pp.237-241. New York: ACM.

Turnitin. (2015) *Turnitin – Features – Overview*. http://turnitin.com/enus/features/overview (Accessed 5 January 2015).

Wagner, R.A. & Fischer, M.J. (1974, January) 'The string-to-string correction problem' *Journal of the Association of Computer Machinery* 21(1) pp.168-173.