Does automated anti-plagiarism have to be complex? Evaluating more appropriate software metrics for finding collusion

Thomas Lancaster and Mark Tetlow
Department of Computing
University of Central England
Birmingham UK
B42 2SU

Abstract

Much research in the plagiarism detection literature relates to attempting to discover which students have copied student source code submissions from one another — a process commonly known as collusion. The majority of the collusion literature suggests that structure metrics (metrics that look beyond semantics to attempt to find disguise) are the most appropriate comparators for finding such collusion. This paper contrasts two paired structure metrics with a paired superficial metric, the metrics having been identified from the plagiarism detection literature. The metrics are compared on a corpus of Visual Basic source code, a programming language that has not been considered previously in the detection literature. The results find that the superficial metric considered, Lancaster word pairs, which calculates the proportion of consecutive words in common between two documents more accurately differentiates between collusion and coincidence and can be argued to be the most effective of the metrics. This suggests that the premise that structure metrics are the most appropriate methods for automated detection might need to be reconsidered.

Keywords

plagiarism detection, student cheating, software metrics

Motivation for the source code plagiarism detection study

An important component of an academic award in computing is that students should demonstrate an ability to program, that is to write a piece of source code that meets a given specification. Programming is a skill often compared to riding a bicycle; it is not something that can be picked up by merely reading about it, instead it requires practice. It is perhaps inevitable that some students may resort to unfair means to complete such source code submissions. One example of this includes working with other students and submitted derived versions of each other's work, a process known as collusion. A second example involves taking another student's work and submitting that or an alternative version of it, a process named by Culwin and Lancaster as intra-corpal plagiarism (Culwin & Lancaster, 2001). It has also been noted that some students are resorting to using external agencies, such as RentACoder.com or Internet cheat sites, like FreeEssays.com, to complete assignments on their behalf. The detection of this type of plagiarism is beyond the scope of this paper (Lancaster & Culwin, 2005b).

In order to maintain the momentum on their side tutors have been relying on plagiarism detection engines that can take a corpus of student submissions and look for intra-corpal plagiarism. This process can be complicated by the fact that a student may not submit an identical submission to one of their peers; instead they may incorporate elements of disguise into the submission.

For instance a solution modelling a light switch may contain methods to switch the light on and to switch it off, but the order in which these two methods are listed should not affect the correct working of the program. A student with little technical knowledge could reorder these two methods as an attempt at disguise. Another possible disguise strategy, without going into technical details, would be to rename the methods, for instance a method to turn a light off could be renamed from switchOff() to turnLightOff() and there are many more complicated alternativesavailable. The method through which a corpus of source code submissions can be compared and similarity identified is known as a metric. If a metric is to be considered appropriate it needs to be able to see through any student attempts at disguise.

Culwin, MacLeod and Lancaster's study of the usage of source code detection on the largest scale, from 2001, revealed that only 26% of responding UK higher education computing departments used automated plagiarism detection methods (Culwin et al., 2001). This figure suggests that there is still a need to educate tutors about the availability of and necessity for such tools. Two main systems were deployed, the Measure of Software Similarity (MOSS) (Boywer & Hall, 1999) system and the JPlag system (Prechelt & Phlippsen, 2002). The study quotes personal correspondence that suggests that around 10% of any corpus of source code submissions sent for processing to MOSS will contain intra-corpal plagiarism. Although this evidence is largely anecdotal it is a relatively large proportion of any student cohort. This would suggest that there is an urgent need for automated plagiarism detection to be used for any programming assignments. The study also noted that the similar pairs of submissions found across the two engines can differ, suggesting that neither submission will find all similar pairs of submissions and that the real figure may be higher.

Lancaster and Culwin categorised several types of systems for plagiarism detection based around the metrics that each deployed (Lancaster & Culwin, 2005a). These classifications were designed to remove inconsistencies in the existing literature. Of particular note are those types of metrics that Lancaster and Culwin identified that applicable to this source code and hence will be most relevant to the remainder of the study reported in this paper. All of the metrics discussed are known as paired metrics, this means that they process two submissions at a time and look for a level of similarity within them. Those pairs of submissions containing the higher levels of similarity are hence the ones that will need to be investigated by a tutor looking for possible plagiarism. The metrics can also be classified as superficial metrics, where any kind of simple count of some measure common to two documents can be made and structure metrics, which depend on some level of parsing the documents prior to processing.

In a further study Lancaster and Culwin looked into the historical source code detection literature (Lancaster & Culwin, 2004a). One main trend amongst recent source code detection engines was evident. Both MOSS and JPlag used structure metrics. Although the engines do not publish details of the full workings of their systems they are believed to use a two-stage process. The first stage consists of tokenisation, taking a piece of source code and replacing any features that could be individualised with a standard token. For instance this could involve replacing variable names or programming keywords that could be easily changed. The second involves searching for common substrings, although the exact comparison algorithm used here is not known. Other recent engines that are believed to use this type of metric include YAP3 (Verco & Wise, 1996a; 1996b) and Big Brother (Irving et al., 2002).

There is little recent evidence to suggest that this style of metric is inappropriate for source code detection. There has only been one identified post 1990 investigation where a superficial metric has been considered (Jones, 2001) where Jones reconsidered some of the earliest source code plagiarism detection ideas attributed to Halstead (Halstead, 1977). Jones study showed that a count of the number of tokens common to two submissions and the number unique between two submissions could be made and the values compared using a closeness calculation. However Jones also found that this technique was not foolproof. Lancaster and Culwin have suggested an alternative, known as the 'Lancaster word pairs' metric, originally developed to find similar in student submissions in free text, such as essays (Lancaster & Culwin, 2004b). The metric works simply by calculating the proportion of pairs of consecutive words in common across two documents. The authors suggest that this metric should be largely language independent and could be applicable to source code.

One largely unrelated metric has also been considered in the recent relevant literature. This is an example of a structure metric that does not use a pre-processing stage of tokenisation and is known as the compression metric. The metric was originally tested for this purpose by Saxon (Saxon, 2000) and tested further by Campbell and Culwin (Campbell & Culwin, 2003). The metric relies on the premise that compression using the zip algorithm looks for areas of replicated content. Where such duplication occurs both areas can be considered together for compression, replacing both areas by a representative token and hence dramatically reducing the size of the compressed file. Hence, where two documents are largely similar if compressed together their size should be close to that of the size of these documents compressed individually. The initial study by Saxon suggested that this metric could give better results than a structure metric (Lancaster & Culwin, 2005a); however, Campbell could not replicate such good results when compared with the MOSS engine (Prechelt et al., 2002).

The source code detection literature has revealed one major omission, the lack of any detection solutions for detection of source code submissions written in the Visual Basic language. In their study Culwin, MacLeod and Lancaster found that 25% of UK higher education computing schools would require a national service to provide detection for this language, but that none of the existing detection engines supported it (Culwin et al., 2001). This omission can be considered surprising since Visual Basic is a commonly used language in industry.

This study aims to investigate the main classes of metrics that could be used to detect plagiarism in source code submissions and due to the absence of current studies it is produced with particular reference to the Visual Basic language. The study will consider a paired structure metric that uses tokenisation, a paired structure metric using compression and a paired superficial metric using word pairs. If the majority of literature and existing tools can be considered indicative the first of these will be the most successful.

Considerations for detection in Visual Basic

The Visual Basic language has been chosen as the basis for this study since it is a language for which widely available detection engines do not appear to have been previously implemented. Although the results of the study could be argued to be dependent on the programming language being considered they can also be considered to be applicable to other languages. This would be an item for future study. As with most programming languages Visual Basic has its own idiosyncrasies that must be addressed to ensure that the methods tested are appropriate for their use.

Most of the existing plagiarism detection engines operate on a corpus of single files; for example these might include class files in Java, or essays on the same theme in free text. A Visual Basic project, the base unit in Visual Basic operates, consists of a minimum of two files, the project file and the form file. The project might also contain addition files in the form of class files and module files. In order for the tests to be complete it is necessary for all of the files making up a project to be compared, differentiating the most applicable techniques for Visual Basic from those for detection in other languages.

To counteract this the metrics being tested will operate on the Visual Basic project concatenated into a single file in a standard order based on the component files contained within it. If the metrics used are successful this choice of ordering should not affect the quality of the results obtained.

Disguise strategies

In order to assess the suitability of the three metrics it is necessary to construct a suitable indicative corpus of documents that represent the different ways in which a student may choose to plagiarise. A metric could then be considered to be effective if it can identify these different methods that students could use to cheat. There is no standard data set for this purpose, although a number of academics have attempted to identify different program modification methods.

Figure 1 shows Hamblen and Parker's traditional spectrum of modifications that can be made to source code programs without changing the results of executing the code (Hamblen & Parker, 1989). These can be considered to represent the different disguise strategies that a student might use to submit source code written by another student as if it were the result of their own effort. These can be considered to represent the different modifications that a successful plagiarism detection engine will need to successfully identify.

The possible modifications listed by Hamblen and Parker range in sophistication. At the lowest level, their L1 categorisation, changes are only made to comments, requiring little technical knowledge. The other end of the spectrum contains their L6 categorisation, where changes are made to the software's main locus of control. Since it could be argued that a student exhibiting this level of technical prowess has demonstrated successful programming ability it is more important to identify the changes at the lower end of the spectrum.

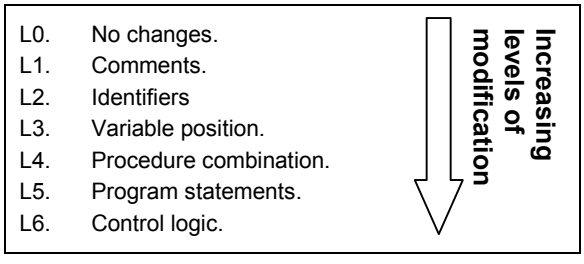

Figure 1: Hamblen and Parker's program plagiarism spectrum

Other academics have identified alternative techniques. For instance, Jones identifies the ten plagiarism transformations shows in Figure 2 (Jones, 2001). Joy and Luck consider two main classifications of techniques: lexical changes not dependent on program structure and structural changes that require more language specific skills (Joy & Luck, 1999). There is some level of overlap between these different classifications. Joy and Luck's lexical changes largely correspond with Jones' transformations 1 to 7 and Hamblen and Parker's levels L1 to L5, although it must be stressed that this is not a direct mapping. Joy and Luck's lexical changes cover the higher levels, namely Jones' transformations 8 to 10 and Hamblen and Parker's L6.

1. Verbatim copying.
2. Changing comments.
3. Changing white space and formatting.
4. Renaming identifiers.
5. Reordering code blocks.
6. Reordering statements within code blocks.
7. Changing the order of operands/operators in expressions.
8. Changing data types
9. Adding redundant statements or variables.
10. Replacing control structures with equivalent structures.

Figure 2: Jones' plagiarism transformations

The synchronicity between the different sets of disguise strategies has led to the development of a set considered suitable for testing for collusion on Visual Basic projects, designed to cover the ways in which students are believed to plagiarise and covering a representative sample of the different techniques. These will be tested by manually inserting plagiarism into a Visual Basic corpus. The techniques chosen for testing are summarised in Table 1.

Table 1: Possible plagiarism techniques to be tested

Name of technique	**Summary of technique**
Original	A direct copy with no attempt at disguise.
White space	The white space in the document has been reorganised, extra spaces or tabs may have been added, or they may have been removed.
Comments	Comments have been added, altered or removed.
Identifiers	Names of the identifiers in the program have been changed.
Blocks	Sections of code for which the ordering has only a superficial effect on the program, such as methods, have been reordered.
Inner blocks	Sections of code within a method have been reordered where the result has no effect on the implementation, such reordering variable declarations.
Control structures	More structural dependent attempts at disguise have been implementing without altering the results, such as replacing a loop or conditional structure.

Implementation of metrics

Three metrics have been selected to be implemented, as a comparative sample of those used both traditionally and most recently within the plagiarism detection literature. The exact implementations developed have been chosen to be appropriate for the concatenated Visual Basic project files.

The metrics used are all paired in origin; this means that they require two projects to operate on, computing a similarity score from each. This score is nominally scaled between 0 and 100, with 100 intended to illustrate the higher levels of similarity, although these cannot be perceived as simple percentages. Instead, operationally, a tutor might choose to start with those pairs of projects designated to have the highest levels of similarity, check these for potential plagiarism and then work iteratively down through the list until such a point where no further checks have been deemed by them to be appropriate.

Outline algorithms to describe how each of the three metrics being considered can be computed are shown in Table 2.

Table 2: Possible metrics for finding collusion in Visual Basic source code

Name of metric	**Representative algorithm**
Lancaster word pairs	Consider two concatenated Visual Basic projects, denoted A and B. For each document compute a sorted list of all pairs of consecutive words, along with a count of the number of times this pair appears. Denote these lists A and B respectively. Compute values c1, c2, c, u1, u2 and u, representing commonality and uniqueness between the projects, where: c1 is the number of times a pair of words occurs in a, so long as it occurs at least once in b. c2 is the number of times a pair of words occurs in b, so long as it occurs at least once in a. c = c1 + c2. u1 is the number of times a pair of words occurs in a, so long as does not occurs in b. u2 is the number of times a pair of words occurs in b, so long as does not occurs in a. u = u1 + u2 Then similarity score for A and B = 100c / (c + u).
Tokenised longest common substrings	Consider two concatenated Visual Basic projects, denoted A and B. Produce two further documents, denoted a and b respectively, which are tokenised versions of A and B. Denote the length of these documents, in words and tokens, as la and lb respectively. Compute the longest substring common to both A and B, with a minimum length of five words or tokens. A common substring is defined as a series of words and tokens common to both documents and in the same order, but not necessarily consecutive to one another. Iteratively repeat this process on the words and tokens in A and B that have not yet been allocated to a substring, until no further allocations are possible. Denote the total length of these substrings as lc. Then similarity score for A and B = 200 lc / (la + lb).
Compression	Consider two concatenated Visual Basic projects, denoted A and B. Produce two further documents C and D, where C is A with B appended to it and D is B with A appended to it. For each of A, B, C and D, compress it using the zip algorithm and compute its size, giving a, b, c and d respectively. Then similarity score for A and B = 100 ((2(a + b) / (c + d)) – 1)).

The tokenisation strategy chosen works by iterating around a Visual Basic project, looking for certain features that can be related to known keywords. For example, consider the structure:

Else If (conditional_statement) Or / And (Another Condition) Then

…

End If

Here the conditional statement could vary between student documents without changing the overall structure of the code. The Or / And (Another Condition) is optional. Only the Else If, Then and End If components are both required in their exact form. The tokenisation method used would lines of these forms to two common and represented tokens to ensure that anything of this form is flagged as possible plagiarism. The implementation used looks for any of 16 representative Visual Basic statements, which are processed in a defined order. This tokenisation stage is implemented using regular expressions.

For the purposes of these tests the detection engine has been implemented using the Java programming language, although this choice of language should have no influence on the validity of the results. The zip compression algorithm has been implemented using the provided java.util.zip package of classes for data compression. The formula for compression relies on the informal observation that the size of two concatenated compressed files is never more than the sum of the sizes of the two individually compressed files and is always more than the size of either of these individual compressed files. This observation has successfully held in practice.

Comparative tests

In order to assess the suitability of the three metrics a corpus of five Visual Basic projects has been collected. Each of these has been disguised using the seven possible plagiarism techniques listed in Table 1 and the similarity score detailing how much commonality there is between the original document and its plagiarised version has been calculated.

Tables 3 to 5 summarise the results of these computations under the three different metrics. In each case the mean similarity score for the five pairs is shown, along with the minimum, maximum and range of the scores received. As a comparative value, the mean similarity score for every possible set of pairs from the ten documents created, giving a total of 55 pairs, is shown. This includes the 50 pairs that could be considered to have little commonality, which is comparable to what could be expected from a corpus submitted directly from students. The final row of each table, denoted 'factor', shows the ratio between the mean similarity score of the plagiarised pairs and the mean similarity score from the overall corpus. This gives an indication of how well the metric differentiates the plagiarised pairs from the remainder of the corpus. The higher values of this ratio represent greater levels of differentiation.

Table 3: Plagiarism detected under the Lancaster word pairs metric

	Original	White space	Comments	Identifiers	Blocks	Inner blocks	Control structures
Mean	100.0	99.8	91.8	91.9	99.7	97.8	98.1
Min	100.0	99.1	88.9	83.8	99.0	96.9	97.2
Max	100.0	100.0	93.1	96.1	100.0	98.6	99.0
Range	0.0	0.9	4.2	12.4	1.0	1.6	1.8
Unrelated	63.0	39.2	38.6	39.2	39.2	39.2	39.2
Factor	**1.6**	**2.5**	**2.4**	**2.3**	**2.5**	**2.5**	**2.5**

Looking purely at the similarity scores obtained by the Lancaster word pairs metric, as shown in Table 3, demonstrates that this metric is very successful at finding superficial changes, such as comments, identifiers and white space, giving scores ranging anywhere from 83.8 to 100. These could be considered to be in the range that a tutor would check. Perhaps surprisingly, as there is no attempt at tokenisation involved, the metric gives even better scores for the more structural changes, such as blocking and control structures. The range of values obtained from the plagiarised pairs varies by no more than 1.8. This shows that attempts at disguise leave enough pairs of code words unchanged that the metric can see through this.

The factors differentiating between commonality purely due to the constraints of programming in the same language and the copied pairs are all 2.3 or greater. This would suggest a clear cut off point for any tutors looking down a list of ordered results to find similarity.

Table 4: Plagiarism detected under tokenised longest common substrings metric

	Original	White space	Comments	Identifiers	Blocks	Inner blocks	Control structures
Mean	99.7	99.5	99.5	98.0	99.7	98.3	99.4
Min	99.4	98.2	98.2	97.5	98.8	96.8	98.5
Max	100.0	100.0	100.0	99.2	100.0	99.6	100.0
Range	0.6	1.8	1.8	1.7	1.2	2.7	1.5
Unrelated	63.9	62.2	64.2	64.2	64.2	64.2	64.3
Factor	**1.6**	**1.6**	**1.5**	**1.5**	**1.6**	**1.5**	**1.5**

The results for the tokenised longest common substrings metric, as shown in Table 4, are also largely conclusive. There is a slight curiosity with the implementation of the metric used in that it returns slightly less than a score of 100 for identical submissions. This is largely due to the manner in which multiple Visual Basic programs are combined to make a single project and has no bearing on the validity of the results.

The results generally give clear and consistent differentials between the plagiarised pairs and the overall corpus statistics. The plagiarised pairs have a similarity score close to 100 and an unrelated score around 64. Informal tests have shown that this type of metric is particularly suited to identifying changes in looping structures and, to a lesser extent, changes in conditionals. The clear cut off point between plagiarism and coincidence shows that this metric is useful for tutors, supporting the literature research.

Table 5: Plagiarism detected under compression metric

	Original	White space	Comments	Identifiers	Blocks	Inner blocks	Control structures
Mean	70.4	59.9	57.5	62.8	63.2	67.0	63.1
Min	3.0	2.8	2.8	3.0	3.0	3.0	3.0
Max	89.5	83.3	80.6	84.1	88.2	87.6	86.4
Range	86.4	80.4	77.9	81.1	85.2	84.6	83.4
Unrelated	36.1	37.5	35.7	36.1	36.1	36.1	36.1
Factor	**1.9**	**1.6**	**1.6**	**1.7**	**1.7**	**1.9**	**1.7**

Table 5 shows the results obtained under the compression metric. Although there appears to be a cut off point between the plagiarised and non-plagiarised it is not as clear-cut as that from the Lancaster word pairs or the tokenised longest common substrings metric. One reason for this is that when the submission is compressed with the exact copy of itself the result returned is lower than 100. There is a range of 86.4, showing a discrepancy almost matching the entire possible range of scores from 0 to 100.

In this case the raw scores obtained from each pair make the reasons for the discrepancy more obvious. One of the five Visual Basic projects from which the plagiarised versions were derived compressed to give an initial similarity score of 3.0, showing that the choice of zip algorithm used struggled to process this particular document. This might be a sign that the success of this approach is very dependent on the exact choice of implementation by the developer, and hence that to use this metric would give very capricious results. When this particular pair of documents is removed from the corpus the remaining results give discrepancy factors along the same lines as the ones received the Lancaster word pairs metric, which might be an indication of how the results have been received favourably in other studies. In this case the metric cannot be recommended.

Table 6 summarises the results obtained across this corpus under the different metrics. The table demonstrates how clearly the results give a clear cut off point between likely plagiarism and random similarity.

Table 6: Comparison of metrics tested on different plagiarism methods

	Original	White space	Comments	Identifiers	Blocks	Inner blocks	Control structures
Lancaster word pairs	1.6	2.5	2.4	2.3	2.5	2.5	2.5
Tokenised longest common substrings	1.6	1.6	1.5	1.5	1.6	1.5	1.5
Compression	1.9	1.6	1.6	1.7	1.7	1.9	1.7

Taken at face value the results support the Lancaster word pairs metric as being the most suitable. The results are interesting as this is largely a superficial attribute counting metric that has not previously been applied to source code. Whilst both compression and tokenised longest common substrings give reasonable results there is not sufficient evidence to suggest that these are more appropriate for this corpus.

Conclusions

This paper has compared three metrics from the literature, all of which have been argued to be suitable for plagiarism detection, on a new version of the problem, to detect plagiarism in Visual Basic code. The tests have been carried out on a corpus representative of student submissions with five of them manually changed to introduce different types of plagiarism.

The tests found that the Lancaster word pairs metric, which assesses the proportion of consecutive words or tokens that the two submissions have in common, was the most successful of the metrics. This metric had previously only been applied to free text. The metric most reported to be appropriate for detection within free text, the tokenised longest common substrings metric, was also found to be successful, but not at the same level as the Lancaster word pairs metric.

The most disappointing results were reserved for the compression metric, which proved to only partially successful, largely dependent on the original source document from which the plagiarised copies were derived. This might be because of the choice of the implementation of the zip algorithm, or this could be due to the decision to implement the results on Visual Basic code. It does suggest that this metric is not the 'miracle cure' for plagiarism that the literature might have you to believe.

One shortcoming identified with the Lancaster word pair metric is that although it has proved to be very suitable for discovering which pairs of submissions are plagiarised, it is not likely to form the basis for a complete plagiarism detection process. This is because the metric, on its own, does not provide any location metric stating where two documents have been judged to be similar to one another. One advantage of the tokenised longest common substrings metric is that this has to compute sequences of similarity in order to work out the extent of similarity. This information can easily be provided to a tutor to provide a more usable detection process. This functionality has been implemented into a prototype tool, which is shown in Figure 3.

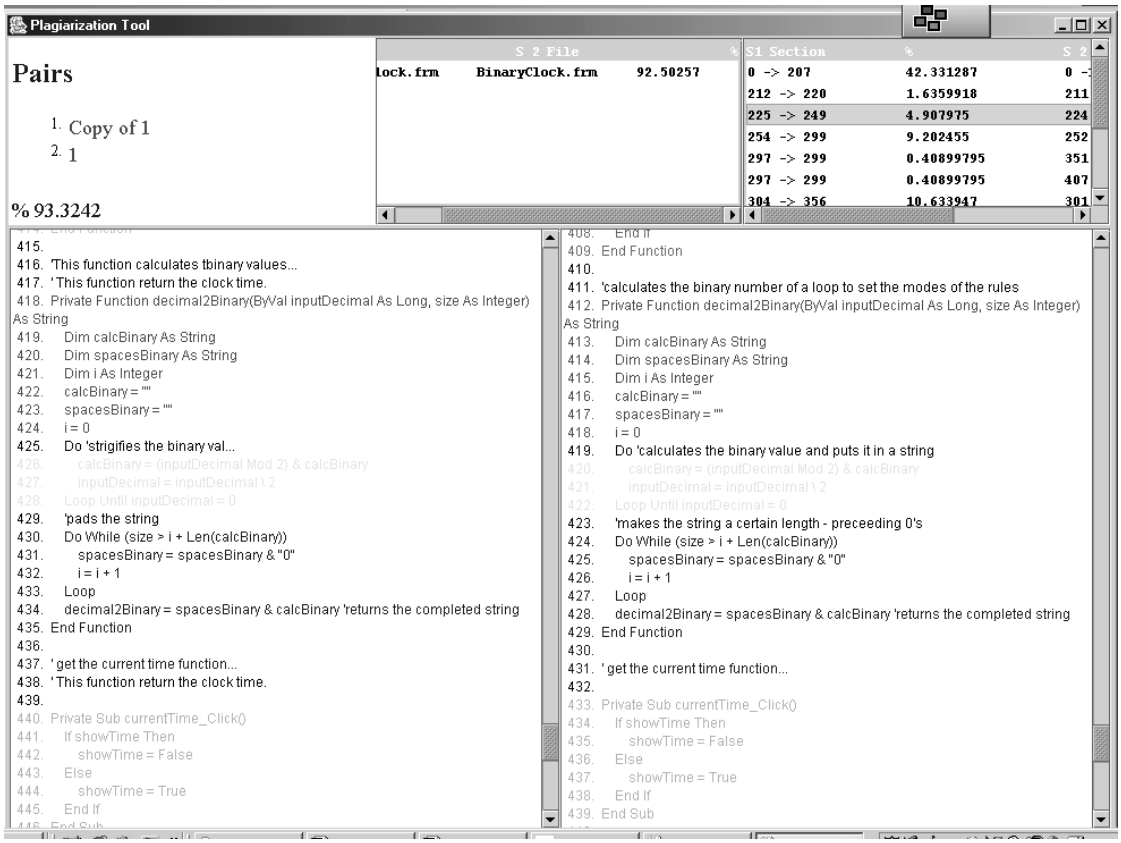

Figure 3: Identifying areas of similarity in Visual Basic code

The results have demonstrated that a metric for free text submissions can successfully be applied to source code submissions. However, testing solely on two languages is not sufficient to suggest that the Lancaster word pairs algorithm would be suitable for a wider range of cases. There are a number of idiosyncrasies across different languages that might suggest that a wider ranging study is needed. The production of a common corpus of documents, or a common range of plagiarism techniques to test against, could prove to be a basis for further work to give a clear framework on which to base effective detection tools upon. Although the work presented in this paper might form the basis for such a study it is not considered to be ideal as there is no clear mapping given about how this could be translated across different languages and into free text.

Visual Basic is not a programming language that has been widely considered in the detection literature before. Despite this, this study has shown that a range of suitable algorithms exists for detection of source code plagiarism. It is likely that this is also true for other programming languages that have not yet been considered.

There is now no reason why student source code should be submitted without having been checked for collusion, if for no reason other than the peace of mind of the tutors concerned. This can help to ensure that academic integrity has been preserved and students have successful worked for the qualifications that they are subsequently awarded. Computing departments without a clear anti-plagiarism strategy need to consider that such tools are easily available or can be produced.

References

Boywer, K. W., & Hall, L. O. (1999). *Experience using "MOSS" to detect cheating on programming assignment.* Paper presented at the 29th ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico (pp. 18–22).

Culwin, F., & Campbell. (2000). *Using compression to identify plagiarised programs.* Paper presented at the 4th Annual Conference of the LTSN Centre for Information and Computer Sciences, Galway, August 2003.

Culwin F., & Lancaster T. (2001). *Visualising intra-corpal plagiarism*. Paper presented at the International Conference on Information Visualisation, London, UK.

Culwin, F. MacLeod, A., & Lancaster, T. (2001). *Source code plagiarism in UK HE computing schools, issues, attitudes & tools*. South Bank University Technical Report, SBU-CISM-01-02.

Halstead, M. H. (1977). *Elements of software science.* (n.p.): Elseveir.

Hamblen, J. O., & Parker, A. (1989). Computer algorithms for plagiarism detection. *IEEE Transactions on Education, 32*(2), 94–99.

Irving, R., MacDonald, G., McGookin, D., & Prentice, J. (2002). *Big Brother (Glasgow University Computer Science Department's collusion detector system, version 2.0) user manual.* (Available from Glasgow University)

Jones, E. L, (2001). *Metrics based plagiarism monitoring.* Paper presented at the 6th Annual CSSC Northeastern Conference, Middlebury, VT.

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

Lancaster T., & Culwin F. (2004b). *A visual argument for plagiarism detection using word pairs.* Paper presented at Plagiarism: Prevention, Practice and Policy Conference 2004. Organised by JISC Plagiarism Advisory Service, Newcastle, UK.

Lancaster, T., & Culwin F. (2005b). *Preserving academic integrity — Fighting against non-originality agencies*. (To appear in *British Journal of Educational Technology* in 2005)

Lancaster, T., & Culwin, F. (2004a). A comparison of source code plagiarism detection engines. *Journal of Computer Science Education, 4*(2), 101–118.

Lancaster, T., & Culwin, F. (2005a). Classifications of plagiarism detection engines. *Italics, 4*(2). Retrieved June 24, 2005, from http://www.ics.ltsn.ac.uk/pub/italics

Prechelt, L., Guido, M., & Phlippsen, M. (2002). JPlag: Finding plagiarisms among a set of programs with Jplag. *Journal of Universal Computer Science, 8*(11), 1016–1038.

Saxon, S. (2000). *Comparison of plagiarism detection techniques applied to student code*: *Compute*r *science project* (Pt. II). Cambridge: Trinity College.

Verco, K. L., & Wise, M. J. (1996b). Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *Proceedings of first Australian conference on computer science education*, Sydney, Australia.

Verco, K. L., & Wise, M. J. (1996a). Plagiarism a la mode: A comparison of automated systems for detecting suspected plagiarism. *The Computer Journal, 39*(9), 741–750.