Same Difference: Detecting Collusion by Finding Unusual Shared Elements

Pam Green[1], Peter C.R. Lane[1], Austen Rainer[1], Steve Bennett[1], and Sven-Bodo Scholz[2]

[1]School of Computer Science, University of Hertfordshire, Hatfield, AL10 9AB, UK.
[2]School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK.

Abstract

Many academic staff will recognise that unusual shared elements in student submissions trigger suspicion of inappropriate collusion. These elements may be odd phrases, strange constructs, peculiar layout, or spelling mistakes. In this paper we review twenty-nine approaches to source-code plagiarism detection, showing that the majority focus on overall file similarity, and not on unusual shared elements, and that none directly measure these elements. We describe an approach to detecting similarity between files which focuses on these unusual similarities. The approach is token-based and therefore largely language independent, and is tested on a set of student assignments, each one consisting of a mix of programming languages. We also present a novel technique for visualising one document in relation to another in the context of the group. This visualisation separates code which is unique to the document, that shared by just the two files, code shared by small groups, and uninteresting areas of the file.

Keywords: source-code, plagiarism detection, unusual similarities, survey, visualisation, n-grams

[1]email: [p.d.1.green, p.c.lane, a.w.rainer, s.j.bennett]@herts.ac.uk; [2]s.scholz@hw.ac.uk

1 Introduction

Joy and Luck (1999) highlight two reasons for one student to copy another's work: they are either unable to understand how to do the work, or are unwilling to take the time to do so. In both cases, it might be assumed that the ability or time needed to disguise the copied work is lacking. Indeed, many academic staff will recognise that unusual elements in student submissions trigger suspicion of inappropriate collusion. These elements may be odd phrases, strange constructs, peculiar layout, or spelling mistakes (Cosma, 2008, p.191).

From a review of twenty-nine approaches to source-code plagiarism detection, we have found that the majority focus on overall file similarity, and not on unusual shared elements. In particular, none of these approaches directly measure the unusual similarities between files. The work reported here computes these unusual similarities, accounting not only for those which appear in a pair of files, but also those which appear in few files. This method is based on trigram analysis, finding pairs, or small groups, of files which share trigrams not found in the rest of the files in the group.

We continue with background information, in particular, with the analysis of approaches to source-code plagiarism detection. In Section 3, we describe two problems in measuring similarity between files of source code produced for assignments, and explain our method for overcoming these problems by measuring the unusual similarity between files, based on trigram analysis. In Section 4, we report on our study using the proposed measure on a set of student projects. The visualisation, described in Section 5, is also based on trigram analysis, and highlights the interesting similarities between two files. We conclude after a discussion in Section 6.

2 Background

2.1 Similarity in program code

The aim in source-code collusion detection is to find meaningful similarity between files submitted by a group of students. Our approach computes this similarity based on elements in the text. However, in any group of documents there are elements which naturally occur frequently. For example, articles and prepositions appear in most English language texts. If the documents have the same subject, topic-related words may also be repeated.

In program code, keywords, idioms and common constructs will appear in many files. For example, "for (i = 0; i < n; i++)" or "# include < stdio.h >" in C code. There are other reasons for similarity between code-based assignments:

- template code may be provided as part of the assignment, meaning that it will be used by every student,
- code used in exercises and examples during the course is likely to be reused,
- code is automatically produced by a development tool, such as Visual Studio, or
- the common aim of the task will constrain students to produce similar code.

This additional incidental similarity means that program code comparison can be more challenging than text comparison (Cosma, 2008; Freire et al., 2007, revised 2011; Hage et al., 2010; Mozgovoy, 2007).

2.2 Approaches to source-code plagiarism detection

The approaches to plagiarism detection in code that we surveyed are listed in date order, from 1996 to 2011, in Table 3 (see pages 19–22). Previous surveys have focused on a smaller number of tools and include Cosma (2008), Hage et al. (2010), and Lancaster (2003). In the table, the authors and, where relevant, the names of the tools (in bold text), are listed in the first column, with dates in the next column.

In broad terms, the steps taken when comparing files are:

- pre-processing the code,
- transforming the processed code,
- breaking the new representation into elements suitable for matching,
- matching these elements,
- post-processing the results, and
- displaying the results.

Key	Meaning	Key	Meaning	Key	Meaning
C	Comments	I	Identifiers	P	Punctuation
C-1	Comments replaced by single token	K	Macros	S	Single character identifiers
		L	Literals	U	Unique terms
G	Globals	M	Imports	W	White-space
H	Headers	N	Numbers	?	Possibly other, unknown

Table 1: Key to the elements removed from code during preprocessing by the plagiarism detection tools analysed in Table 3, column 3.

Pre-processing: During preprocessing, different parts of the code are excluded by different tools, either explicitly, or as a consequence of the transformation process, such as white-space (W) in tokenising, or comments (C) in graph construction. Keys to what is removed by each tool are in the column marked 'Pre', with the meanings of the keys in Table 1.

Transformation: Code is generally transformed into an intermediate representation. The column headed "TI" notes the way that the code is transformed initially. These new forms are tokens (Tk), intermediate language (IL), trees (Tr), graphs (Gr) or metrics (Mt). Following initial transformation, further changes may be made. For example, by filtering out some of the information, by parameterising or flattening trees to form sequences or sets of tokens. Further details about the transformations are in column 5.

Elements for matching: The new representations of the code are usually broken into smaller elements for matching. For example, text and tokens may be divided into lines, statements, or n-grams, which comprise n adjacent words, characters or tokens. The units may then be compared directly, or transformed before comparison. For example, by hashing, fingerprinting, or by forming sets, bags or frequency vectors. These elements are shown in column 6.

Matching: Elements are either expected to match exactly, or some measure of their similarity is computed. Similarity measures are noted in column 7, these vary, depending to some extent on what is suitable for the elements to be matched. In some cases researchers compare several measures, and the one found to be most effective is listed here.

Post-processing: The need for post-processing depends on the application, the elements matched, and the method by which they are matched. Some methods provide an immediate measure of similarity. For example, the Jaccard[1] or Dice[2] coefficients of similarity between two sets of token n-grams, or the cosine distance between two attribute vectors. Otherwise, after matching sections of code within files, such as in string tiling, alignment methods, or clone-based methods, the proportion of shared code to total file size provides a measure of similarity between two files.

Displaying results: Results can be provided as a ranked list of pairwise similarity scores, as a similarity matrix, or file clusters based on their similarity. Other graphical displays include Cosma's (2008) box-and-whisker plots,

[1] Jaccard coefficient $= \frac{|\text{Set intersection}|}{|\text{Set union}|}$ [2] Dice coefficient $= 2 \times \frac{|\text{Set intersection}|}{|\text{Set 1}|+|\text{Set 2}|}$

Freire's (2008) individual histograms, or the Categorical Patterngrams of Ribler and Abrams (2000). Many of the tools provide a method for displaying the text of the files side-by-side, with detected similarities highlighted, so that the tutor can determine the nature of the similarity. Column 8 shows how results are reported and also, to fit the page, any remarks, which are in square brackets.

Excluding template code: The column headed "Exc." indicates whether the tool is reported to have the option of excluding template code from the comparison between files, (✓) or whether common code is inversely weighted in the calculation of similarity (*), making template code, which will appear in all or most submissions, less important in the resulting similarity score.

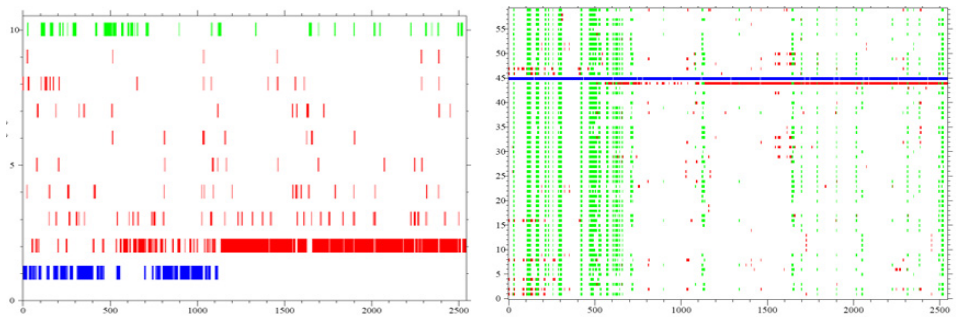
Unusual similarity: Hoad and Zobel (2003, p.2) state that *"In plagiarized assignments, it is common to find that some errors or atypical usages have been copied verbatim."* The last column shows whether files which share this type of unusual similarity are identified (✓, directly; *, indirectly; O, optionally). This last feature will not highlight collusion where disguises are employed, however, it is of interest where the student has too little time or skill to make sufficient alterations to their submission.

Four approaches, by Arwin and Tahaghoghi (2006), Burrows et al. (2007), Ji et al. (2007), and Cosma (2008), accentuate similarities which occur infrequently within the group. The first two because they use a similarity measure which inversely weights terms, depending on how often they occur in the set of files. The second two because they weight the terms in the input vectors. Ji et al. apply weights to the scores assigned during the local alignment of keyword sequences, and Cosma uses latent semantic analysis on vectors of filtered terms.

Two other approaches target unusual similarity between files more directly. Moss (Aiken, 1997), provides a parameter, m, which allows exclusion from its comparison elements which occur in more than m documents. However, as it compares a subset of the hashed n-grams in each file, some unusual similarities between files may be missed. Full details of Moss's implementation are not available, making further discussion difficult. Ribler and Abrams (2000), is the only approach to specifically target unusual similarity between files based on the full set of n-grams in each file.

2.3 Ribler and Abrams' categorical patterngrams

Ribler and Abrams' approach relies on the visual inspection of graphical displays of the character n-grams in a file. The categorical patterngram, in Figure 1a, places the sequence of n-grams in one file along the x-axis. The



(a) Categorical patterngram (b) Composite categorical patterngram

Figure 1: Ribler and Abrams's patterngrams, taken from (Ribler and Abrams, 2000)

number of other files in the group which contain each n-gram is plotted on the y-axis, for the values 1–9. If the n-gram is unique to the file being analysed, it is plotted at y = 1 and shown in blue. If in 10 or more files, it is considered uninteresting and plotted at y = 10, in green. Figure 1a shows a file which shares a suspicious sequence of code with one other file, indicated by the dense sequence of red lines at n = 2.

Another graph, the composite categorical patterngram, is constructed to show which files contain the n-grams, see Figure 1b. This graph is based on one file, with the n-grams it contains plotted for each of the files in the comparison group. This time, the y-axis is labelled with file numbers and the presence of the n-gram in a file marked by a coloured point. The base file n-grams are coloured blue, otherwise the point is green if the n-gram is in 10 or more files and red otherwise. This highlights those files with which the base file shares significant sections. Figure 1b is based on the same file, 45, as Figure 1a, and shows that most of the code shared by few is in file 46, where the large amount of common code indicates collusion.

2.4 Ferret

The copy detection tool Ferret was originally developed by the University of Hertfordshire Plagiarism Detection Group to measure the similarity between text files (Lane et al., 2006; Lyon et al., 2001), and has since been adapted for use with program code (Rainer et al., 2008). Efficiency in processing is achieved by constructing a trigram to file (trigram-file) index on a single pass through each file in the group. The index is used to compute the Jaccard coefficient of similarity between pairs of files, based on trigrams (3-grams) of words or tokens.

Table 2 shows an excerpt from an example trigram-file index. Twenty-one files are compared, one of the code provided for the course, file [0], and

Label	Trigram	Files in which the trigram occurs
	.....	.....
A.	unique_identifier = -1	FILES:[17]
	} unique_identifier =	FILES:[17]
	else } unique_identifier	FILES:[17]
B.	char* id123 ,	FILES:[4 12]
	void func234 (	FILES:[4 12]
	; shiftleft (	FILES:[4 12]
C.	] ; struct	FILES:[4 12 15]
	void dosomething (	FILES:[2 4 12 19]
	result == -4	FILES:[1 4 12 14]
	if (somecondition	FILES:[2 4 12 19]
	(somecondition)	FILES:[4 12 17 19]
D.	abc = 3	FILES:[4 13 19 20]
	int abc =	FILES:[3 4 7 8 12 16 17 19 20]
	int xyz =	FILES:[4 6 7 9 13 16 17 19 20]
E.	= 1 ;	FILES:[0 1 3 4 5 6 7 8 10 12 14 15 17 18 19 20]
	printf ("	FILES:[0 1 3 4 6 7 9 10 12 14 15 17 19 20]
	= ("	FILES:[0 1 3 4 5 9 12 13 17 19]
	(i =	FILES:[0 1 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19]
	.....	.....

Table 2: Extract from an example of a trigram-file index showing trigrams and the files where they occur. For example, the trigram "`char* id123 ,` " occurs in just two files, numbered 4 and 12, while the trigram "`( i =` " is in the majority of the 21 files. The letters A–E are not part of the Ferret output, but label areas which are discussed further in Section 3.

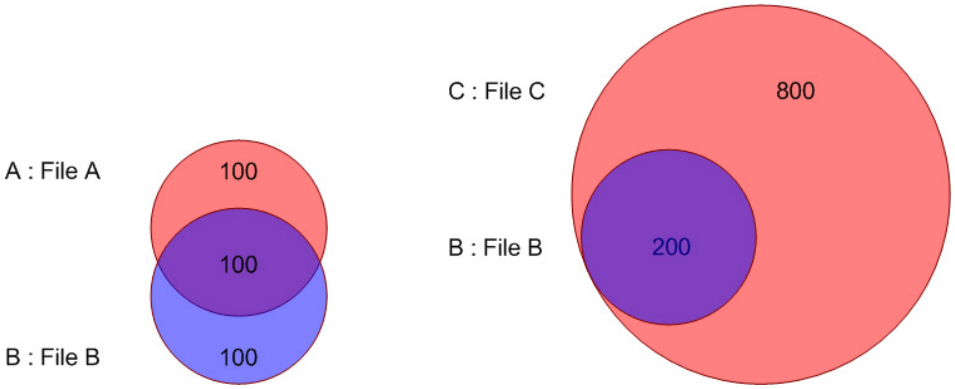
twenty student assignments, numbered [1–20]. In this table, the trigrams are on the left, with the files in which they occur on the right.

3 Measuring similarity in program code assignments

Most of the methods (26 of 29) listed in Table 3 use proportional similarity measures. For example, the number of tokens in matched sections to the total in the file (Mozgovoy et al., 2005), and those using the Jaccard coefficient (Chilowicz et al., 2009; Jadalla and Elnagar, 2008; Lane et al., 2006; Moussiades and Vakali, 2005; Xiong et al., 2009) or Dice coefficient (Huang et al., 2010; Ji et al., 2007).

These measures are used to rank pairs of files, so that the most similar can be further investigated. Proportional measures work best when the files are of roughly the same size. For example, when essays are required to be a specific length, or for programming tasks which consist of a defined set of exercises. However, when file sizes differ, such as in a more open software development task, proportional measures can be misleading.

When a portion of a large document is copied to another document, the proportion of copied trigrams to the total may be small. This may mean that, although significant, copying is missed. In Figure 2, file B is compared with



(a) $|A|=|B| = 200,\ |A \bigcap B| = 100$ (b) $|C| = 1000,\ |B| = 200,\ |C \bigcap B| = 200$

Figure 2: The Venn diagrams represent the trigrams in 3 files, A, B and C. In each diagram, file B has 200 trigrams. File A also has 200 trigrams and the 2 files share 100 trigrams. File C has 1000 trigrams and contains file B.

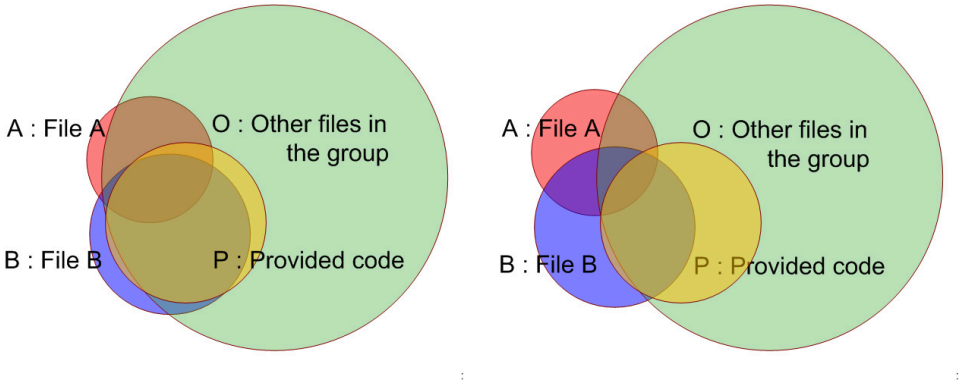
two other files, A and C. The Jaccard coefficient, or resemblance between files A and B is $\frac{100}{300} = 0.33$, whereas between files B and C the resemblance is $\frac{200}{1000} = 0.2$. File B is more likely to be derived from file C than file A, but the resemblance between files A and B is higher than that between files B and C.

One option which overcomes the potential drawbacks of proportional measures for files of unequal sizes is to count the elements shared by each pair of files, to find the amount, rather than the proportion of shared elements.

3.1 Trigrams shared by two files

There are also problems with using a count of shared trigrams to measure similarity. As discussed in Section 2.1, even when there has been no collusion between the authors, the incidental similarity between files of code submitted for an assignment can be high. Factors include the constraints introduced by using the same programming language(s), the code provided to the students for the course, the common aim of the task, and, possibly, auto-generated code. We aim to avoid these problems by using a measure which does not include this incidental similarity.

The Venn diagrams in Figure 3 show four circles representing the trigrams in a group of files. The red and blue circles are two student files, A and B. The yellow circle, file P, contains code provided for the course, either in the form of examples given in teaching, or template code for the assignment. The green circle, labelled O, represents the other files in the group. Files A and B share the same number of trigrams. In the left-hand



(a) Little unusual similarity

(b) Much unusual similarity

Figure 3: The Venn diagrams represent the trigrams in two student files, A and B, in code provided for the course, P, and in other students' work, O. In each diagram, the files A and B share the same number of trigrams. On the left, the majority appear in other files in the group, while on the right, there are a large number of "uniquely shared trigrams", $(A \bigcap B) \backslash (P \bigcup O)$.

diagram, 3a, the majority of the code shared by A and B also appears in other files in the group, while in the right-hand diagram, 3b, about half of the trigrams are "uniquely shared" by the two files. This set of trigrams, $(A \bigcap B) \backslash (P \bigcup O)$, is a strong indication that the files should be further investigated.

3.2 Extending shared counts to groups

Collusion is not always between just two people, but may also be among a group. For example, in the excerpt in Table 2, students 4 and 12 not only uniquely share 3 trigrams, but also share 5 other trigrams with one or two others (section C). There is also evidence, from the trigrams in sections C, D and E, that student 19 is working with students 4 and 12. Of the 10 trigrams shown for student 19, 4 are in the provided code, and therefore used by many students. The remaining 6 are all shared with student 4, and 4 of these are also shared with student 12.

The count of trigrams shared by two students can be extended to include the trigrams shared by the two documents and by a small group of others. In Figure 4, the area shared by the two files A and B, and by other files except for the provided code, $((A \bigcap B \bigcap O) \backslash P)$, is shown split into sections. Some of these trigrams will be shared by files A, B, and just one of the other files. The other file can be any one of the rest of the group. In the top left part of the diagram are three such groups, formed by A, B, and each of the files labelled i (green), ii (pink) and iii (cyan). In the lower left section, three

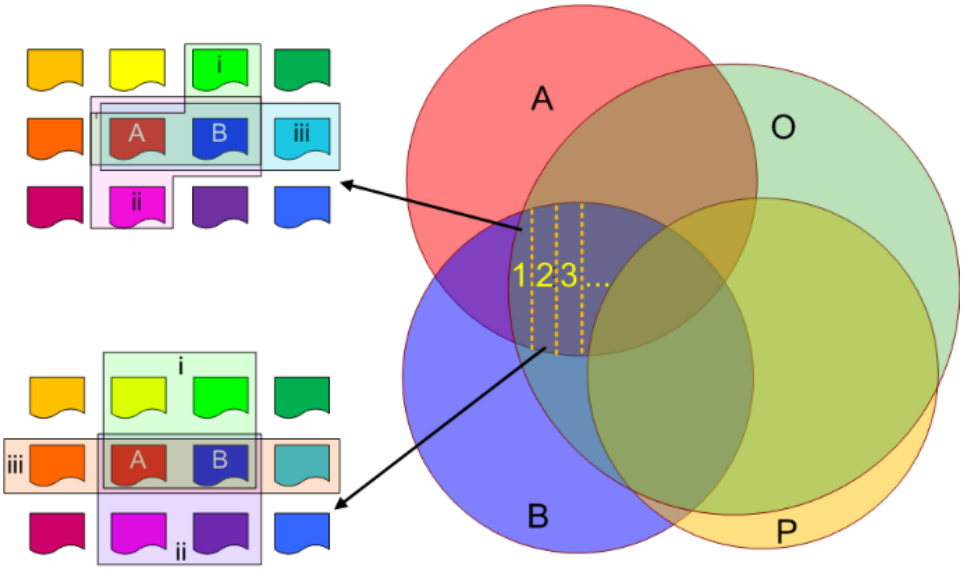


Figure 4: Two files may share trigrams uniquely, or may share trigrams which are shared with few other files. This diagram illustrates the files A and B and examples of the way they may share trigrams with one other file at the top, or two other files at the bottom.

groups of four files which include files A and B are shown in the same way.

One way to compute this extended count of shared trigrams is to weight the trigrams shared by A, B, and n others according to the number of files containing the trigrams. Illustrated in Figure 5, the two files, A and B, uniquely share 100 trigrams, share 60 with any one of the other files, and 240 with any two others. In the example, we weight the shared trigrams by $\frac{1}{n+1}$. The weighted similarity count is therefore:

$$\left(100 * \frac{1}{1}\right) + \left(60 * \frac{1}{2}\right) + \left(240 * \frac{1}{3}\right) = 100 + 30 + 80 = 210.$$

This weighted count can be used as a measure of similarity between files. In this example, groups of up to four files are taken into account, however, the method allows for calculation of any maximum group size.

3.3 Individual counts

Counting trigrams is useful in other respects: for measuring individual effort, and for measuring engagement with tasks set during a course.

Trigrams unique to one document, $(A \setminus (P \bigcup B \bigcup O))$, such as the trigrams in section A of Table 2, may be seen as a measure of individual effort. Alternatively, if there is suspicion that code has been sourced from the internet, the unique trigrams can be used as search terms.

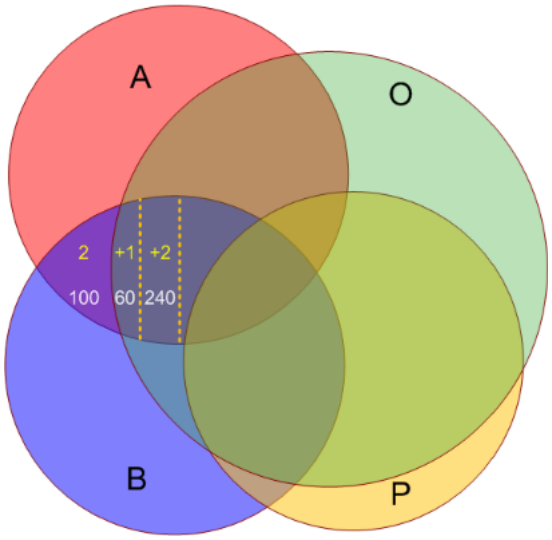


Figure 5: Two files may uniquely share trigrams, may share trigrams with each other and one other, two others, ... Here the files A and B uniquely share 100 trigrams, share 60 which are also shared with another member of the group, and 240 with two other members.

If the students' work is collected regularly, then a measure of each student's engagement with the exercises set during the course can be found by counting trigrams shared with the provided code. For example, $A \bigcap P$ in Figures 3–5. It may be difficult to determine the number of trigrams expected to be shared, as the provided code will probably need to be edited to complete the exercises. However, as long as some of the students are doing the exercises, a baseline level should be apparent. Students falling below this level can then be identified.

4 Study

The group in our study were taking a course during which they were asked to develop a community website using ASP.NET and VB.NET. The students were expected to commit their code to an online repository on a regular basis, and staff had access to the code for analysis purposes. Motivations for using a repository included encouraging steady development and thus discouraging contract cheating, and enabling feedback, both to staff for monitoring, and to students to show their progress in relation to others in the group. Trigram analysis was used both in feedback, to measure originality, and in monitoring, to measure similarity.

There were 64 students initially registered on the course. Their projects were written using Visual Studio, which produces a large amount of auto-generated code. Although not necessary, except to exclude images, the

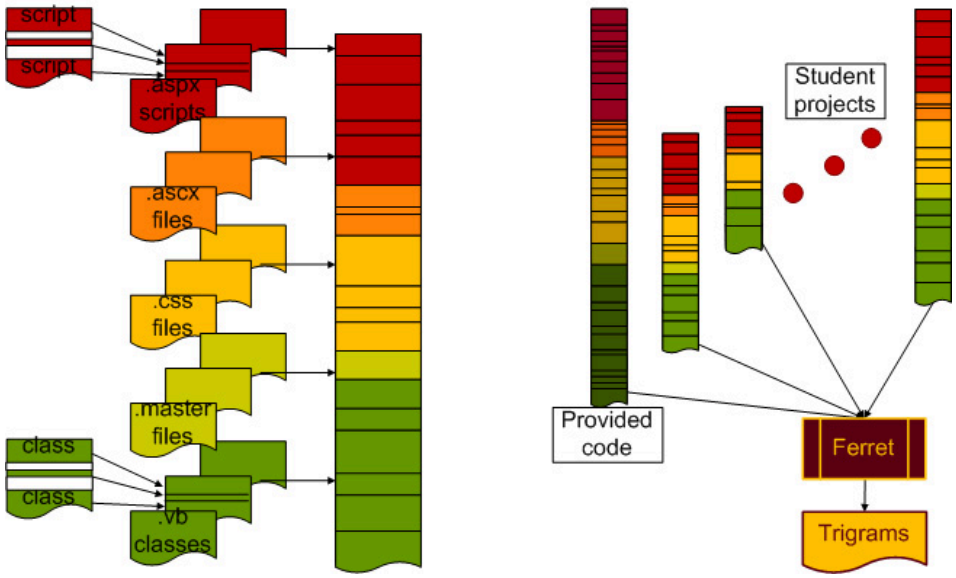


Figure 6: The selected files are concatenated to form one large file for each student (shown on the left). Provided code is also placed in one file. This file and those of student code are presented to Ferret to obtain information about the distribution of trigrams in the files.

files were filtered prior to comparison. Only files expected to contain the student's own code were selected for analysis. These were .ascx, .css and .master files, with scripts from .aspx files, and classes from .vb files.

Each student's code was concatenated into one large file to simplify comparisons between their work, see the left-hand side of Figure 6. This technique has been used before in plagiarism detection, for example by Lancaster and Tetlow (2005). The concatenation introduces new trigrams at the file margins and these vary depending on file order. However, this is a small price for making the rest of the process more straightforward.

Ferret currently has tokenisers for C-type languages, but appears to work effectively with the code in these projects. A C-type tokeniser will give a slightly different token set to a language-specific tokeniser. For example, "not equals" in Visual Basic, "<>", which is treated as two tokens, "<" and ">", instead of one.

The code provided for the course, in the form of examples or exercises, was also concatenated into one large file. This file and all of the student files were compared by Ferret and the trigram-file index and similarity scores saved for analysis.

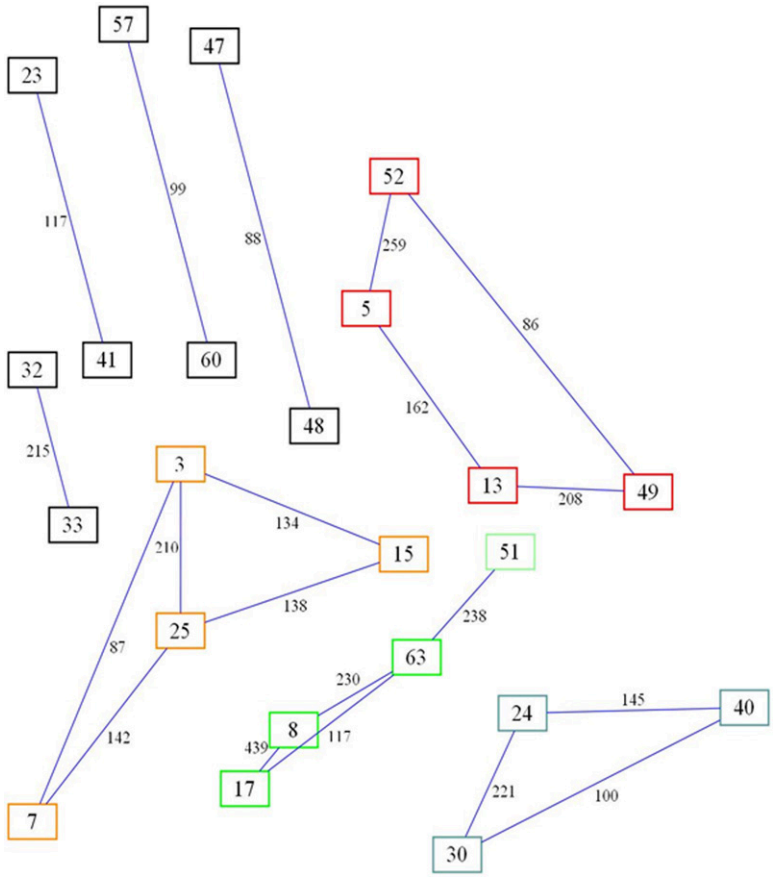


Figure 7: Connections between files with a weighted trigram count of at least 85, calculated for a maximum of 4 files. The connections are inversely proportional to the weighted trigram count.

4.1 Results

Weighted trigram counts were calculated for each pair of documents, taking into account any trigrams shared by up to two others. The results are presented in graphical form in Figure 7. Connections of less than an arbitrary weighted count of 85 are removed, leaving four pairs of files and four groups. The group formed by 24, 30 and 40 is fully connected, the others partially so.

As the course was closely monitored, we did not expect to find inappropriate collusion, and generally this was the case. However, a number of features make this set of projects a good test for the measures. The projects vary in size, ranging from 400 to 7,000 lines of filtered code. There is high incidental similarity because many of the ideas introduced in class were incorporated into the projects as they were developed. There is also some unusual similarity, where students have found similar ways of dealing with

more advanced technical problems. Third party code, such as that for dealing with differences in browser display, was acceptable provided it was properly attributed.

As with all collusion detection techniques, the similarity measure used here provides a filter which points to pairs or groups of files which merit further investigation. Comparing this measure to the normal proportional Ferret similarity measure, $\frac{|(A \bigcap B)|}{|(A \bigcup B)|}$, we found that of the top 20 pairs detected by the weighted count based measure, only 6 were in the top 20 of those detected by Ferret. The majority of pairs with the higher Ferret similarity scores consisted of projects which had not been well-developed, and therefore had a larger than usual proportion of provided code.

We also measured the trigrams uniquely shared by two files proportionally, using the Jaccard coefficient, $\frac{|(A \bigcap B) \backslash (P \bigcup O)|}{|(A \bigcup B) \backslash (P \bigcup O)|}$. Twelve of the top 20 similar pairs of files based on this measure were the same as those in the top 20 of the weighted counts. The 8 selected by proportional measure and not by weighted measure did not show significant areas of similarity, but were selected because one or both projects were small.

5 Enhanced display of a two file comparison

The outputs of many copy-detection tools show the two files under scrutiny next to each other, highlighting the shared parts of the files. In this section, we show how trigram analysis can be used to display more information about the interaction between two files in the context of a group.

In Figure 9, an excerpt from a comparison between two projects is depicted. The comparison is output as XML, which is minimised here to fit the page. The left-hand column shows the code coloured by Ferret in normal use. Trigrams shared by the two files are coloured blue, and are otherwise black. This does not provide information about the nature of the similarity.

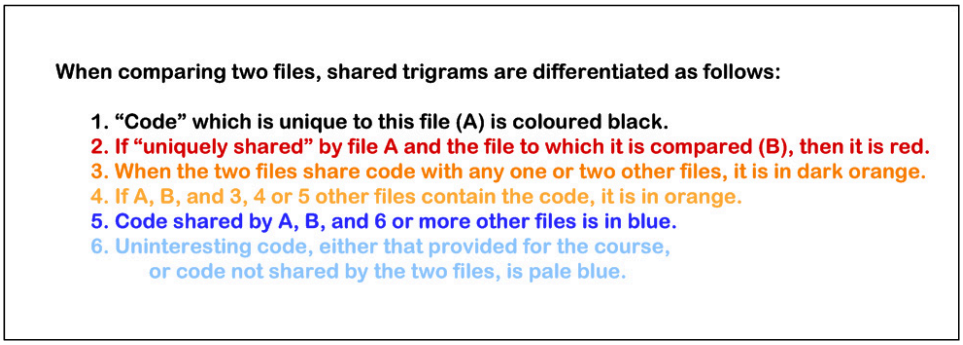


Figure 8: One way to colour the code in a file (A) compared to another (B). The colours and gradations can be altered to suit the user's needs.

By analysing the trigram-file index, more information can be shown. In the middle column, the trigrams which are uniquely shared by the two projects are coloured red. On the right, the idea is developed to provide additional information, which is detailed in Figure 8.

The trigrams are coloured black if they are unique within the group. Red code is, again, that uniquely shared with the other project. Shades of orange show trigrams are shared by the two, but also shared by few others. Here, dark orange means one or two others sharing, and pale orange, three, four or five others sharing. Bright blue means that code is shared by the two projects and at least six others, and pale blue means that the code is provided for the course, and is therefore uninteresting whether shared by A and B or not.

The group thresholds are arbitrary, and it is possible that a larger class warrants larger group sizes. However, Burrows et al. (2007, p.14), in their work on detecting plagiarism, note that *"Our test data and anecdotal evidence indicates that students generally do not work in very large groups"*, and this is taken into account.

6 Summary and Discussion

6.1 Survey

We have surveyed twenty-nine approaches to source-code plagiarism detection. Although it is recognised that unusual similarity between files is a good indicator of possible collusion, few of the approaches surveyed take this into account, with only two specifically targeting the elements shared by few files, and none directly measuring them all.

Another observation from the survey is that twenty-one of the approaches initially tokenise the code. Tokenising requires a suitable lexical analyser (lexer), but means that the method is otherwise language-independent. A lexer for one language can be used to tokenise other languages in a reasonable manner, as we have shown by use of a C-type lexer. However, eighteen of these approaches parameterise the tokenised code, and for this a language-specific lexer is necessary to identify keywords. Tools which change the code to graph, tree or metric representations are tied to one language. Those which convert the code into an intermediate language aim for language independence, but are restricted to the languages supported by the compilation suite. Only those approaches which are token-based and do not parameterise will be able to analyse a mix of languages, that is Ferret (Lane et al., 2006) and PlaGate (Cosma, 2008) among those listed in Table 3.

6.2 Measures

Almost all of the surveyed similarity measures are proportional to the size of one or both of the files being compared. Although proportional measures are likely to perform well with files of similar sizes, there can be a problem when file sizes differ. Count-based measures can find similarity which may be missed by proportional measures, especially where files are large. However, care must be taken in what is counted, otherwise pairs of large files will have more in common than pairs of small files, because of the inherent similarity in program code. This problem is resolved by computing similarity based on less common trigrams. In our study, we found the count-based measures more effective than the two alternative proportional measures tested.

Jones (2001, p.2) reports on methods used to disguise plagiarism: changes to comments, white-space, identifiers and data-types; reordering code within statements, moving blocks of code, adding redundant statements, and exchanging one type of control structure with another. Many of the approaches are concerned with detecting similarity in the face of these disguises. These approaches use strategies such as parameterising , which lead to a loss of textual information. However, there is a place for information preserving approaches, such as Ferret (Lane et al., 2006) and GPlag (Liu et al., 2006).

Although our method is not robust to systematic identifier renaming, it will find unusual common code which has not been well disguised. This similarity would probably be found naturally by someone marking a small group of moderately sized assignments. However, as the size and number of assignments increases, and particularly when marking is distributed among several tutors, unusual similarity is unlikely to be found without automatic analysis.

6.3 Visualisation

Our graduated visualisation pinpoints the areas of similar files which warrant investigation, especially helpful in a large file with high levels of incidental similarity. Parts of the file which are unique to one student are also highlighted, useful in understanding the novelty of the work, or perhaps where outside sources have been used. As far as we know, no other approach offers the level of detail displayed by our source code visualisation. Although our display is based on trigram analysis, other units of comparison, such as lines, clones, or procedures, could be used in a similar way. Colour-blindness is a consideration (Jefferies et al., 2011), however, the scheme can be easily altered to suit the user; for example, by the use of underlining or shading in place of one or more of the colours.

7 Conclusion

Analysis of twenty-nine approaches to source-code plagiarism shows that unusual similarity between student submissions is not directly measured by existing tools. Our method for computing similarity overcomes the problems which can arise from proportional measures and from the inherent similarity in program code, and performed well on our dataset. The visualisation adds group context to the comparison between two files. The more interesting sections of the file are highlighted, which helps in assessing the nature of the similarity between files. Future work will include testing the measure on other groups of assignments and further developing the tool.

Figure 9: An excerpt from project A, compared to project B. On the left, is the normal Ferret colouring, where trigrams shared by A and B are in blue, and black means not shared. The middle is the same, except that the trigrams uniquely shared by the two projects are in red. On the right, the excerpt is coloured according to the scheme shown in Figure 8.

Author	Date	Pre.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Wise **YAP3**	1996	C, I L	Tk	Synonym unification, functions rearranged by call sequence	Hashed n-grams where n reduces iteratively	Running Karp-Rabin Greedy String Tiling	% Coverage by matched tiles		
Aiken **Moss**	1997 2003	W, ?	Tk	Token n-grams, hashed	Hash values reduced by winnowing to fingerprint the file	Matching fingerprints trigger more detailed text matching	Similarity as %, matched lines to total	✓	O
Gitchell and Tran **SIM**	1999	C-1	Tk	Parameterised, e.g. TKN-ID-I, TKN-FOR	Token strings One program against modules of the other	Normalised alignment ids =, 2; ≠, 0; gap, -2; other tokens =, 1; ≠, -2	High scoring pairs displayed		
Joy and Luck **Sherlock**	1999	- C, W C, W	- - Tk	1. Original code 2. Excluding C, W 3. Tokenised	Each set of lines	Runs of equal lines, may be gapped, as proportion of file size	Similar files clustered by Kohonen SOFM [Since updated]		
Ribler&Abrams **Categorical patterngram**	2000	W	-	Character n-grams (optionally previously parameterised)	Hashed n-grams	Matching hash-values	Graphical display for results	✓	✓
Jones	2001	C, W	Tk	Counts of lines words, chars, length, vocabulary & volume	1. Code attribute vectors 2. Error log att. vectors 3. Execution log att.vecs.	1. Euclidean distance 2. Compilation log dist. 3. Execution log dist.	Pairwise similarities		
Prechelt et al. **JPlag**	2002	C, I L	Tk	as YAP3	– – –	– – –	[as YAP3 with hashing optimised]	✓	
Belkhouche et al. **Brass**	2004	C, W	Tr	Partitioned structure chart and data dictionary	Subgraph type sequ's. Nodes Type frequency Identifiers	L.C. Subsequence Subtrees & token types % matched nodes by type & by type-id pairs	Pairwise % Matches, near-matches, & nodes [Tiered: 1. subgraphs, 2. nodes, 3. symbols]		

Table 3. Approaches to plagiarism detection Continued on next page

Author	Date	Pre.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Chen et al. **SID**	2004		Tk	Token sequences	Compressed files (Cm)	Compression distance $1 - \frac{Cm(a)-Cm(a\|b)}{Cm(ab)}$	File pairs ranked by similarity		
Lancaster and Tetlow	2005		Tk	1. Words 2. Parameterised 3. Unprocessed	'Word' bigrams L.C. Substrings Compressed files	$100 \times$ Jaccard coeff't $200 \times \frac{Tokens\ in\ matched\ sections}{Total\ tokens\ in\ file}$ $100 \times \frac{Cm(a)+Cm(b)}{Cm(ab)+Cm(ba)} - 1$	[Test framework to compare 3 measures - finds bigrams best]		
Moussiades &Vakali **PDetect**	2005	C	Tk	Indexed keywords e.g. {void1, int1, int2, for1, ...}	Keyword set	Jaccard coefficient	Clustered on weighted undirected graph		
Mozgovoy et al. **FDPS**	2005 2007	W	Tk	Parameterised (p-match)	Token sequences using suffix array	$\frac{Tokens\ in\ matched\ sections}{Total\ tokens\ in\ file}$	Similarity matrix		
Arwin & Tahaghoghi **Xplag**	2006	C, W	IL	Converted to RTL, optimised, tokens filtered	N-grams of selected tokens	Okapi BM25	Relative % score	*	*
Lane et al. **Ferret**	2006	W	Tk		Token trigrams	Jaccard coefficient	Pairwise similarities		
Liu et al. **Gplag**	2006	C, I W	Gr	Procedural program dependence graphs (PDGs)	Graph pairs, filtered to exclude small units and unlikely matches	Subgraph isomorphism	Counts of approx. matched procedures		
Merlo **CLan**	2006	C, W	Mt	Function metrics (branch, parameter, etc. counts)	Metric vectors	Clone clusters based on thresholds	Proportion of files covered by clones		
Noh et al. **EXPDec**	2006	C, W	Tr	1.XML tree to 6 sets of feature frequencies 2. Control sequences	Feature matrices Control matrix	Comparison depends on the features Levenshtein (weighted)	Pairwise similarities		

Table 3. Approaches to plagiarism detection Continued on next page

Author	Date	Pre.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Burrows et al.	2007	C, W	Tk	Parameterised to 1 of 55 token types, e.g. int S, return g, (A, = K	Token type 4-grams	1. Okapi BM25 2. String alignment on best matches match 1, indel -2, mis -3	Pairwise similarities	*	*
Ahtiainen et al. **Plaggie**	2007	C, I L	Tk	as JPlag	– – –	– – –	[as JPlag but open source]	✓	
Freire et al. **AC**	2007 2008	U	Tk	1.Token type counts 2.Token sequence	1. Token frequ'y vectors 2. Compression (Cm)	Cosine distance $\frac{Cm(ab)-min(Cm(a),Cm(b))}{max(Cm(a),Cm(b))}$ variance analysis	Pairwise similarities File cluster graph Relative sim'y histogram		
Ji et al.	2007	C, W	Tk	Static trace "keywords" e.g.IF LE BLOCKSTART RETURN FUNCCALL	Local alignment with scores weighted on on keyword frequencies	Dice coefficient aligned section to file sizes	Similarity matrix	*	*
Cosma **PlaGate**	2008	C, N P, S, U	Tk	Term frequencies	Procedure level term vectors resulting from latent semantic analysis	Cosine similarity	Similarity matrix/plot	*	*
Jadalla &Elnagar **PDE4Java**	2008	C, W	Tk	Parameterised, unifying identifiers, separators, keywords .. as digits	Bag of 4-grams of the digits	Jaccard coefficient	Pairwise similarities Clusters similar files using DBSCAN		
Chilowicz et al.	2009	C, I W	Tk	Parameterised, unifying some token types e.g. IF LPAR identifier	Token suffix array Global call graph of subfunc'ns inc. shared	Jaccard coefficient & containment based on subfunctions	Pairwise file similarities, containment(1in2,2in1)		
Xiong et al. **BUAA_ AntiPlagiarism**	2009	C, W, H, G	IL	CIL to AST to linear representation, which is tokenised	Token 4-grams	Jaccard coefficient	Clusters similar files		

Table 3. Approaches to plagiarism detection Continued on next page

Author	Date	Pre.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Brixtel et al.	2010		Tk	Alphanumerics unified e.g. t t = t + t;	Choice e.g. lines, functions	Choice, any suitable measure for sequences	Similarity matrix [1-to-1 section matching by Munkres algorithm]		
Hage et al. **Marble**	2010	C, M S	Tk	Tokens unified by type Optional function sort	Transformed lines	L.C. Substring (diff) Similarity = 100 - $\frac{no.different lines(f1,f2)\times 100}{length f1+length f2}$	Pairwise similarities above threshold score		
Huang et al.	2010	C, K W, ?	Tk Tr	1. Identifiers unified, concatenated into string 2. 4 AST node type sets	1. Winnowed hashed n-grams = fingerprint, f 2. Nodes paired on sim.	1. Dice. $\frac{(2\times LCSeq(f1,f2)}{leng(f1+f2)}$ 2. $\Sigma_{i=1}^{4} w_i.(meansim)_i$ weighted combination of similarities 1. & 2.	Pairwise similarities		
Juričić	2011	C, W	IL	Converted to CIL Filtered to exclude metadata & locations	Machine instruction strings e.g. ldc, stloc, add	$1 - \frac{Levenshteindistance}{Maximum filesize}$	Similarity matrix		

Table 3: Approaches to source code plagiarism detection. Author and **tool** names are listed in column 1. Parts of the code removed during pre-processing are in column 3, comments (C) and white-space (W) are most common, as these can be a by-product of transformation, other keys are in Table 1. The initial transformation of the code is shown in the next column, (TI), where Tk means that the code is tokenised, other keys are: Gr-graph, Tr-tree, Mt-metrics, and IL-intermediate language. Brief descriptions of further transformation, elements matched, and similarity measures are in columns 5–7. Column 8 has report formats and [selected remarks]. The last 2 columns show whether the tool allows **exc**lusion of template code, and if **dis**similarity to the rest of the group is considered. [✓- explicit, * - implicit, O - optional]

References

Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GPL-licensed source code plagiarism detection engine for Java exercises. In M. Wiggberg A. Berglund, editor, *6th Baltic Sea Conference on Computing Education Research*, pages 141–142. Uppsala University, Sweden, 2007.

Alex Aiken. Moss system for detecting software plagiarism. http://theory.stanford.edu/aiken/moss/, 1997.

Christian Arwin and Seyed M. M. Tahaghoghi. Plagiarism detection across programming languages. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *ACSC*, volume 48 of *CRPIT*, pages 277–286. Australian Computer Society, 2006. ISBN 1-920682-30-9.

Boumediene Belkhouche, Anastasia Nix, and Johnette Hassell. Plagiarism detection in software designs. In Seong-Moo Yoo and Letha H. Etzkorn, editors, *ACM Southeast Regional Conference*, pages 207–211. ACM, 2004. ISBN 1-58113-870-9.

Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-independent clone detection applied to plagiarism detection. In *SCAM '10:Proceedings of the 10th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 77–86. IEEE Computer Society, 2010. ISBN 978-0-7695-4178-5.

Steven Burrows, Seyed M. M. Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. *Software Practice and Experience*, 37(2):151–175, 2007.

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.

Michel Chilowicz, Étienne Duris, and Gilles Roussel. Finding similarities in source code through factorization. *Electronic Notes in Theoretical Computer Science*, 238 (5):47–62, 2009.

Georgina Cosma. *An Approach To Source-code Plagiarism Detection And Investigation Using Latent Semantic Analysis*. PhD thesis, University of Warwick, 2008.

Manuel Freire. Visualizing program similarity in the AC plagiarism detection system. In *Proceedings of Advanced Visual Interfaces (AVI)*, pages 404–407, New York, USA, May 2008. ACM Press. ISBN 1-978-60558-141-5.

Manuel Freire, Manuel Cebrián, and Emilio del Rosal. AC: An integrated source code plagiarism detection environment. *CoRR*, abs/cs/0703136, 2007.

Manuel Freire, Manuel Cebrian, and Emilio del Rosal. Uncovering plagiarism networks. arXiv:cs/0703136v7 [cs.IT], 2007, revised 2011.

David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In Jane Prey and Robert E. Noonan, editors, *SIGCSE*, pages 266–270. ACM, 1999. ISBN 1-58113-085-6.

Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Utrecht University, June 2010. Further information from http://www.cs.uu.nl/docs/vakken/apa/10plagiarismdetection.pdf.

Timothy C. Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003. ISSN 1532-2890.

Liuliu Huang, Shumin Shi, and Heyan Huang. A new method for code similarity detection. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*, volume 2, pages 1015–1018, dec. 2010.

Ameera Jadalla and Ashraf Elnagar. Pde4java: Plagiarism detection engine for Java source code: a clustering approach. *IJBIDM*, 3(2):121–135, 2008.

Amanda Jefferies, Colin Egan, Edmund Dipple, and Dave Smith. Do you see what I see? Understanding the challenges of colour-blindness in online learning. In S Greener and A Rospiglio, editors, *Proceedings of 10th European Conference for E-Learning*, pages 210–217, Reading, U.K., 2011. Academic Conferences Ltd.

Jeong-Hoon Ji, Soo-Hyun Park, Gyun Woo, and Hwan-Gue Cho. Source code similarity detection using adaptive local alignment of keywords. In David S. Munro, Hong Shen, Quan Z. Sheng, Henry Detmold, Katrina E. Falkner, Cruz Izu, Paul D. Coddington, Bradley Alexander, and Si-Qing Zheng, editors, *PDCAT*, pages 179–180. IEEE Computer Society, 2007. ISBN 0-7695-3049-4.

Edward L. Jones. Metrics based plagiarism monitoring. *Journal of Computing Sciences in Colleges*, 16(4):253–261, 2001.

Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(1):129–133, 1999. URL `http://eprints.ecs.soton.ac.uk/3872/`.

Vedran Juricic. Detecting source code similarity using low-level languages. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*, pages 597 –602, June 2011.

Thomas Lancaster. *Effective and Efficient Plagiarism Detection*. PhD thesis, Birmingham City University, Birmingham, UK, 2003.

Thomas Lancaster and Mark Tetlow. Does automated anti-plagiarism have to be complex? evaluating more appropriate software metrics for finding collusion. In *Ascilite 2005*, pages 361–370, Brisbane, Australia, 2005. ascilite 2005.

Peter C. R. Lane, Caroline M. Lyon, and James A. Malcolm. Demonstration of the Ferret plagiarism detector. In *2nd International Plagiarism Conference*, 2006.

Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *KDD*, pages 872–881. ACM, 2006. ISBN 1-59593-339-5.

C. M. Lyon, J. A. Malcolm, and R. G. Dickerson. Detecting short passages of similar text in large document collections. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*. SIGDAT, Special Interest Group of the ACL, 2001.

Ettore Merlo. Detection of plagiarism in university projects using metrics-based spectral similarity. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

Lefteris Moussiades and Athena Vakali. Pdetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.

Maxim Mozgovoy. *Enhancing Computer-Aided Plagiarism Detection*. PhD thesis, Department of Computer Science, University of Joensuu, University of Joensuu, P.O.Box 111, FIN-80101 Joensuu, Finland, November 2007.

Maxim Mozgovoy, Kimmo Fredriksson, Daniel R. White, Mike Joy, and Erkki Sutinen. Fast plagiarism detection system. In Mariano P. Consens and Gonzalo Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 267–270. Springer, 2005. ISBN 3-540-29740-5.

Seo-Young Noh, Sangwoo Kim, and Cheonyoung Jung. A lightweight program similarity detection model using xml and levenshtein distance. In Hamid R. Arabnia, editor, *FECS*, pages 3–9. CSREA Press, 2006. ISBN 1-60132-008-6.

Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *The Journal of Universal Computer Science*, 8 (11):1016–, 2002.

Austen W. Rainer, Peter C. R. Lane, James A. Malcolm, and Sven-Bodo Scholz. Using n-grams to rapidly characterise the evolution of software code. In *The 4th International ERCIM Workshop on Software Evolution and Evolvability*, 2008.

Randy L. Ribler and Marc Abrams. Using visualization to detect plagiarism in computer science classes. In *INFOVIS*, pages 173–178, 2000.

Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X.

Michael J. Wise. Yap3: improved detection of similarities in computer program and other texts. In John Impagliazzo, Elizabeth S. Adams, and Karl J. Klee, editors, *SIGCSE*, pages 130–134. ACM, 1996. ISBN 0-89791-757-X.

Hao Xiong, Haihua Yan, Zhoujun Li, and Hu Li. BUAA AntiPlagiarism: A system to detect plagiarism for C source code. In *International Conference on Computational Intelligence and Software Engineering, CISE 2009.*, pages 1–5, 2009.