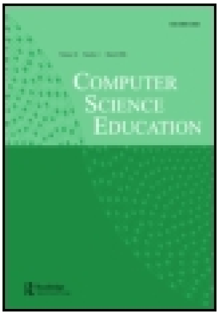

Computer Science Education
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ncse20

A Comparison of Source Code Plagiarism Detection Engines

Thomas Lancaster [a] & Fintan Culwin [b]

[a] School of Computing, University of Central England Birmingham U.K.

[b] Centre for Interactive Systems Engineering School of Computing, London South Bank University London U.K.

Published online: 09 Aug 2010.

To cite this article: Thomas Lancaster & Fintan Culwin (2004) A Comparison of Source Code Plagiarism Detection Engines, Computer Science Education, 14:2, 101-112, DOI: 10.1080/08993400412331363843

To link to this article: http://dx.doi.org/10.1080/08993400412331363843

Computer Science Education
2004, Vol. 14, No. 2, pp. 101–117

A Comparison of Source Code Plagiarism Detection Engines

Thomas Lancaster[1] and Fintan Culwin[2]
[1]School of Computing, University of Central England, Birmingham, U.K.,
and [2]Centre for Interactive Systems Engineering, School of Computing,
London South Bank University, London, U.K.

ABSTRACT

Automated techniques for finding plagiarism in student source code submissions have been in use for over 20 years and there are many available engines and services. This paper reviews the literature on the major modern detection engines, providing a comparison of them based upon the metrics and techniques they deploy. Generally the most common and effective techniques are seen to involve tokenising student submissions then searching pairs of submissions for long common substrings, an example of what is defined to be a paired structural metric. Computing academics are recommended to use one of the two Web-based detection engines, MOSS and JPlag. It is shown that whilst detection is well established there are still places where further research would be useful, particularly where visual support of the investigation process is possible.

INTRODUCTION

Computer programming is considered an essential part of an undergraduate computing course, but in these days of ever-increasing class sizes and ever-decreasing student commitment it is very difficult to preserve *academic integrity.* That is to ensure that students have written the programs they submit for *academic credit*. Reports indicate that source code plagiarism may be reaching record proportions (Clare, 2000; Irving, 2000). Automated tool support to ensure that students have not plagiarised is an essential wherever

Address correspondence to: Thomas Lancaster, School of Computing, University of Central England, Franchise Street, Perry Barr, Birmingham B42 2SU, U.K. E-mail: thomas.lancaster@uce.ac.uk

10.1080/08993400412331363843$16.00 © Taylor & Francis Ltd.

programming is taught although indications are that this is only used sparsely.

A survey by Culwin, MacLeod and Lancaster revealed that only 14 of 54 responding UK Higher Education Computing departments used automated detection for source code submissions (Culwin et al., 2001). The remaining institutions primarily relied on manual inspection of code, a process through which similarity can easily be missed. Only four institutions surveyed thought that plagiarism was rare. Eleven departments said that they had considered automated detection but not got round to using an engine. The survey cites personal communication with Alex Aitkin, provider of the MOSS service, that states that around 10% of any source code corpus submitted to MOSS will be plagiarised. This is from students who are aware that their work will be checked, hence the proportion plagiarising unchecked work could be expected to be higher. This suggests a need for computing academics to be informed of what solutions are available for plagiarism detection in source code and for them to use such a system.

This paper presents a comparison of existing modern source code plagiarism detection engines and literature based around the techniques they deploy. The comparison is motivated by material presented by Lancaster in his PhD thesis (Lancaster, 2003). The language used to discuss plagiarism was defined by Culwin and Lancaster in their plagiarism taxonomy (Culwin & Lancaster, 2000). Important terms introduced or defined for the first time in this paper are presented in italics for ease of readability. Most notably *source code plagiarism* is usually an example of *intra-corpal plagiarism*, plagiarism within a corpus of documents. *Source code* is an example of a constrained text, since there are a limited number of possible words and tokens that can be used within correctly formed programs.

Traditionally *source code detection engines*, those tools that compare student submissions for similarity to be checked by a human to see if it represents plagiarism, have been classified into two types, attribute counting systems and structure metric systems. Lancaster and Culwin argued that these classifications have been inconsistently applied and are limited (Lancaster & Culwin, 2003). In particular engines would be placed in different classifications depending on which author's definitions were used and some engines could not be classified in this way. Instead Lancaster and Culwin proposed a new set of classifications based general properties of the engines and the metrics that they deploy to find similarity. These will be used to compare the engines in this paper.

Table 1. Modern Detection Engines Identified.

Name of engine	Literature
Big Brother	Irving (2000), Irving et al. (2002)
Cogger	Cunningham and Mikoyan (1993)
DetectaCopias	DetectaCopias (2002)
Jones	Jones (2001a, 2001b)
JPlag	Prechelt, Guido, and Phlippsen (2000), Prechelt, Malpohl, and Phlippsen (2002)
MOSS	Bowyer and Hall (1999)
Saxon	Saxon (2000)
SHERLOCK	Joy and Luck (1999)
SIM	SIM (1989)
TEAMHANDIN	Culwin and Naylor (1995)
YAP3	Verco and Wise (1996a, 1996b), Wise (1996)

IDENTIFIED DETECTION ENGINES

Table 1 shows the main modern detection engines that have been identified and the key literature or Web sites where the engine is described. Other literature that gives a more general comparison of selected engines will be discussed later. The engines are introduced now to allow them to be classified. Two tools are not named in the literature, these are referred to here throughout by the names of their authors, Jones and Saxon respectively (Jones, 2001a, 2001b; Saxon, 2000). Saxon's engine is the only one that exists solely for research purposes but is included here because the original detection method used is of interest.

GENERAL CLASSIFICATIONS OF SOURCE CODE DETECTION ENGINES

Lancaster and Culwin identified a number of general ways in which plagiarism detection engines can be classified once it has been decided that they operate on source code submissions (Lancaster & Culwin, 2003). The engines can be classified based on the location of the documents they process, either *local* where the tool is downloaded or processing done on the home machine, or *Web-based* where processing is done over the Internet. They can also be classified based on their availability. *Private* engines are only available to their host institution, *public* engines are available for widespread use. Alternatively engines may be made available by *special arrangement*; this category should

Table 2. General Classifications of Source Code Engines.

Name of engine	Location	Available	Current
Big Brother	Local	Special arrangement	Yes
Cogger	Local	Private	No
DetectaCopias	Local	Public	Yes
Jones	Local	Private	Yes
Jplag	Web-based	Public	Yes
MOSS	Web-based	Public	Yes
Saxon	Local	Private	No
SHERLOCK	Local	Public	Yes
SIM	Local	Public	Yes
TEAMHANDIN	Local	Private	No
YAP3	Local	Public	No

include those engines that are not free of charge although none are known. Many engines described in the literature are no longer available. These are classified as *past* engines. Engines that are still available are classified as *current*.

Table 2 classifies the engines identified in Table 1 based on these general classifications.

It is also necessary to know what *programming languages* the engines operate on, that is the languages that can be parsed by the detection engine. Some engines will only process submissions that can be compiled and will reject those that are not *parsable*. Table 3 shows this information. Some engines are also stated to work on *free text* such as essays and this information is also included in the table, where known.

CLASSIFICATIONS BASED ON TECHNIQUES USED TO FIND SIMILARITY

Of more academic interest and in many ways better for comparative purposes are the metrics that the engines use to find similarity. Lancaster and Culwin identified a number of sets of metrics that are useful classifiers, where each engine can only be classified under one metric in a set (Lancaster & Culwin, 2003).

The first set of metrics relates to the number of submissions that have to be analysed consecutively to generate metrics, two types of which are used in the identified engines. *Singular metrics* are ones where every submission is extracted down to a vector of representative numeric, known as its fingerprint.

Table 3. Classifications of Source Code Engines by Programming Languages Processed.

Engine name	Java	Ada	COBOL	Scheme	C	C++	Pascal	ML	Modula-2	Miranda	Lisp	Free text	Must be parsable?
Big Brother	√	√										√	No
Cogger					√								Unknown
DetectaCopias	√												Unknown
Jones			√										No
JPlag	√			√	√	√						√	Yes
MOSS	√	√		√	√	√	√	√			√		No
Saxon	√												No
SHERLOCK							√		√	√			No
SIM	√				√		√		√	√	√	√	Unknown
TEAMHANDIN		√	√		√	√	√						No
YAP3					√						√	√	Unknown

Table 4. Classifications of Source Code Engines by Metrics Used.

Name of engine	Singular	Paired	Superficial	Structural	Tokenisation
Big Brother		√		√	√
Cogger		√		√	
DetectaCopias		√		√	
Jones	√		√		√
JPlag		√		√	√
MOSS		√		√	√
Saxon		√		√	
SHERLOCK		√		√	√
SIM		√		√	√
TEAMHANDIN	√		√		
YAP3		√		√	√

These fingerprints are then combined for each possible pair in a corpus to give a numeric similarity score, of which higher similarity scores show more similarity. The alternative *paired metrics* take two submissions and generate a similarity score directly from them and are intended to require more comparative information about the two submissions than could be generated by combining two singular metrics.

The second set relates to metric complexity. *Superficial metrics* are those where the value can be gauged simply by looking at one or more programs with no semantic knowledge, *Structural metrics* require semantic knowledge. The border between these metrics can be fuzzy.

The final metric consideration is whether the engines apply *tokenisation* to submissions or not. That is whether they reduce the source code to a representative form, for instance by replacing variable names with common tokens throughout a corpus, before processing. Such a technique is intended to reduce the effect of some student disguise strategies.

Table 4 shows the metric techniques used by the engines identified in Table 1.

SINGULAR METRIC ENGINES

There are few recent engines that use singular metrics for anything other than comparative research purposes. The most recent attempt is in work by Jones (2001a, 2001b). Jones creates three different fingerprints for each source code submission. A similarity score is produced for each pair of submissions under each fingerprint with some normalisation to within a known range. The first fingerprint is a vector of physical aspects of the file, namely the number of characters, words and lines. This is said to be easily fooled. The second fingerprint contains three Halstead metrics, namely the total number of tokens, the number of unique tokens and a combination of the two. These are metrics used by Halstead in one of the earliest engines (Halstead, 1977). Jones shows these to be more effective, but can be fooled by adding new unused procedures or functions. The third fingerprint contains both the physical and Halstead fingerprints and is said to give fairly random results. Generally Jones research is limited showing little knowledge of dominant paired metric engines and few acknowledgements. It does suggest that the Halstead metrics may still be suitable for some situations.

One further recent singular metric engine forms part of Culwin and Naylor's TEAMHANDIN engine (Culwin & Naylor, 1995). The fingerprint used two superficial metrics, these are the number of reserved words and the number of identifiers that exist on a pre-declared list. Tokens are weighted pragmatically according to their frequency of occurrence within a corpus. Submissions with close values for both metrics are flagged as potential plagiarism. TEAMHANDIN has one additional benefit over older engines and that is an ability to cluster source code submissions. Hence the user can be presented with groups of similar submission as opposed to pairs of just two submissions when investigating similarity. The approach is shown to be successful by validating it with known plagiarism cases.

Generally comparative literature studies do not find singular metric engines to be particularly suitable for detection. Verco and Wise have shown that what they called attribute counting systems, which primarily use singular metric techniques are inefficient compared to other metrics available (Verco & Wise, 1996a, 1996b). Granville found a single paired metric to be more suitable than a single singular metric (Granville, 2002). Saxon's more thorough testing also found that the best metric was a paired metric (Saxon, 2000). Hence although some singular metric engines might still give reasonable results on certain source code corpora they have been largely superseded by paired metric engines. All dominant recent engines used the paired metric approach.

LOCAL PAIRED METRIC ENGINES

Nearly all modern local paired metric engines use a tokenisation pre-processing stage. Only Joy and Luck's SHERLOCK engine has been identified as an exception (Joy & Luck, 1999). The stated benefit of SHERLOCK is that it is language independent; this means that there is no need to reprogram SHERLOCK to accommodate new programming languages. Five metrics are used to compare each pair of submissions. These involve finding the length of the longest common substring in the four combinations of source code considered: with and without white space and comments, as well as in a completely tokenised version of the code. Joy and Luck state that the language independent approach finds much of the plagiarism, but does not always find as much as pure tokenisation engines suggesting that these may be more appropriate. Luck has stated that the intention is to make SHERLOCK available for download as open source software (Luck, 2002, private communication).

One of the more advanced engines using tokenisation is Michael Wise's YAP3 (Wise, 1996). This is the third version of a engine developed from the likes of Plague (Whale, 1990) using structure matching algorithms to detect shuffled areas of code. Wise states that there are two main improvements to the algorithm: sorting functions into the order they were first called and mapping synonyms into a common form. This is combined with a fast comparison algorithm and partial parsing of the source code files, which merely extracts words from a lexicon, giving operational time efficiency benefits. This means that programs that will not compile can still be checked for plagiarism.

Irving's Big Brother engine is an example of an engine developed for use by a single institution, in this case Glasgow, although Irving will make it available on a limited basis for external tutors who require assistance (Irving, 2000). Big Brother proved to be particularly effective in finding plagiarism in an outbreak during the 1998–1999 session, where nearly 15% of a 400 student cohort were found to have submitted work with notable similarity. Big Brother is a standard tokenisation engine that works on programs with comments and formatting removed. It can find both the longest common substring and longest common subsequence as proportions of the pair's length (Irving et al., 2002). Of the two, Irving states that the latter is the most successful plagiarism differential. Additionally the program can find the combined length of common substrings over a given size, which is said to be more accurate when the copying is fragmented, but is operationally much slower. Generally the methodology seems fairly standard and hence the results should be accurate but the engine does not contain a mechanism for reporting why two submissions have been judged similar. This is a shortcoming since it is often difficult to see why a machine has judged two submissions similar.

The Cogger engine is described as being novel since it does not require exhaustive analysis of all pairs of submissions in a corpus (Cunningham & Mikoyan, 1993). Instead the engine computes a function call tree, one type of tokenisation, for each piece of source code and clusters these using a neural network and Case Based Reasoning (CBR) techniques. The paper concludes by stating that the results given are variable, since applying CBR to the plagiarism problem successfully needs more work, particularly in generalised cases, which are often missed. Hence this approach can be considered more of a curiosity than one likely to be developed into a real engine, since there doesn't seem to have been any further work done on Cogger since 1993.

The Software and Text Similarity Tester (SIM) is mentioned briefly here since it is available for free use by a Web download (SIM, 1989). SIM is reported to be used with more than adequate results at Vrije University, Amsterdam. It is said to work by finding the longest common strings in tokenised student submissions. Details of SIM's effectiveness in comparison with more modern engines are limited since it is an older engine that seems largely ignored until the launch of the Web.

One final underdeveloped paired metric technique on the borderline between superficial and structural, has created a lot of interest amongst the detection community and so is included here in more detail as a pointer towards further work. Saxon suggests finding the entropy of a piece of source code, a numerical measure of the compressibility of the submission (Saxon, 2000). This is done using standard gzip functions from the Java libraries with the size of the compressed file measured. Defining com(A) as the compressibility of a file A and X++Y as the concatenation of files X and Y (or Y appended after X), a similarity score is taken as $[(\mathrm{com}(A{+}{+}A) + \mathrm{com}(B{+}{+}B))/(\mathrm{com}(A{+}{+}B) * 2)]$. The idea is that if there is redundant information in B that already occurs in A the pair of files will compress greatly. A large benefit of the technique is that it is language independent, since no properties of programming languages are needed, although the compression calculations are computationally intensive. Saxon's initial tests finds the technique to be highly effective in comparisons against standard paired metric systems.

Limited initial results from Campbell have been less successful (Campbell, 2002, private communication). Campbell compares the ranking list generated by using the metric on submitted student Java source code pairs with the list ordering returned by the MOSS detection service. Campbell finds that few documents occur in the upper portion of both lists, the area that would likely be checked by tutors for potential plagiarism. His best results use tokenised then compressed versions of the files. The results however are stated to be only initial results and depend on the reliability of the MOSS ordering, so Saxon's technique may still be suitable with modifications. This does suggest that there is a need for further research into the technique and comparisons against other paired metric approaches.

WEB-BASED PAIRED METRIC ENGINES

Two final engines, MOSS and JPlag, operate remotely and allow legitimate tutors open access. Both come highly regarded as detection solutions so will be discussed here in greater depth.

Perhaps the better known of two to computing academics is Aiken's Measure of Software Similarity (MOSS) (MOSS, 1994). No technical details of MOSS are available, only an independent critique describing one approach for using the engine and integrating it into a Computing course with an appropriate policy (Bowyer & Hall, 1999). More detail about the engine is given in the local university press, which stated that 10% of students on one unit had been suspended after their work was checked by MOSS (Sanders, 1998). The article stated that MOSS was particularly useful for classes on compilers.

Culwin, MacLeod and Lancaster describe MOSS from an operational perspective (Culwin et al., 2001). Tutors submit source code using a submission script, usually on Unix platforms. Results of the plagiarism analysis are available on a Web site on the MOSS server for 2 weeks. A list of pairs ordered by the number of tokens matched is presented to the user, who can then select a pair for further investigation. Two pieces of source code are shown side by side with areas that MOSS believes similar highlighted in the same colour. A tutor can make a decision as to whether this constitutes plagiarism. The main additional feature noted is that the MOSS results are self-adjusting and sections of code common to a majority of submissions are assumed to be supplied code and hence excluded from the calculations. This is reported to be counter-intuitive, since two identical submissions may be reported as less than 100% similar. Tutors can also supply base code to exclude from the calculations, a feature not supported by many other engines.

The alternative Web-based tool, JPlag, offers similar functionality to MOSS (JPlag, 2002). It is available for public use over the Web or by command line where additional parameterisation is possible. Prechelt, Malpohl and Phlippsen describe JPlag from a technical perspective (Prechelt et al., 2000, 2002). The engine computes tokenisation metrics for code using an optimised version of the YAP3 algorithms. After processing, results are made available on an overview Web page, containing a histogram of submissions at ten levels of similarity. In practical usage only those submissions at the highest levels of similarity will be checked further. Users can view pairs on a side by side comparison, a technique used by both JPlag and MOSS.

Prechelt, Malpohl and Phlippsen evaluated JPlag by submitting four corpora of Java source code to it (Prechelt et al., 2000, 2002). Each corpus contained a maximum of 59 submissions, some corpora had plagiarism manually added. The authors define precision (P) and recall (R), where precision is a measure of false hits and recall is a measure of missed pairs. The performance of the engine is rated by calculating P+3R and trying to minimise this value, rating missed pairs as far more crucial than false hits. The papers show that parameters exist for each corpus that rank all the plagiarism highly, although these are not the same for every corpus. The parameters include such things as the minimum tokenised string match length and the choice of tokens to be used. This does suggest that there is not one best solution for finding plagiarism and that knowledge of the content of submissions is necessary to choose appropriate parameters.

The tests carried out on JPlag involve a number of different types of plagiarism being manually added to a corpus (Prechelt et al., 2000, 2002). The tests show that pairs containing these relatively simple types of plagiarism can be identified. More complex source code plagiarism techniques, such as exception handling, modifying data structures and adding redundant methods are shown to be likely to fool the engine. It is open to debate how far a student needs to modify a program before they have demonstrated sufficient skill to meet the learning outcomes for a programming assignment, so this may not be a cause for concern.

Culwin, MacLeod and Lancaster provided a review of both MOSS and JPlag from a technical perspective and tested them by submitting an identical corpus of source code submissions to each (Culwin et al., 2001). The tests were restricted to source code submissions that could be parsed by both engines. They are unable to rate one detection engine above the other, finding beneficial features to each. They do recommend that institutions seriously consider using such a service.

The authors find that there is little correlation between the ordered similarity results supplied by JPlag and those supplied by MOSS. Only the results of the MOSS percentage ranked metric compared with JPlag's only metric, also representing percentage, showed any significant correlation. There are cases of pairs that rank highly under one engine but do not rank anywhere in the limited list of top matches returned by the other. This means that if one engine is used exclusively there may be missed pairs in the rankings obtained if students can figure out how to plagiarise and avoid the automated detection.

The main advantages of MOSS found over JPlag are that MOSS will process files that JPlag cannot parse, finding plagiarism that JPlag might otherwise miss. MOSS uses three metrics to compute its results, this gives more information to a user than JPlag's one. The main advantages of JPlag found over MOSS are that JPlag will accept submissions over a regular Web browser (Netscape only) or command line, whereas MOSS requires command line submission only. A quirk of the MOSS implementation means that identical submissions may sometimes be reported at less than 100% similarity; this is not a problem with JPlag.

One recent addition to MOSS that is worthy of mentioning is the MOSSCliques post processing software (Popyack et al., 2003). At present MOSSCliques is not integrated with MOSS and there are no published details about how it operates but it clearly addresses the need for identifying clusters where more than two students have collaborated, a problem identified by Culwin and Naylor (1995).

OTHER TECHNICAL DETECTION APPROACHES

The modern literature also includes three notable approaches that are not strictly detection engines but are worth mentioning for completeness and to suggest potential future work. Harris advocates that plagiarism should be found by examining source code history files and quizzing students on their knowledge of their programs (Harris, 1994). There is no comparative literature on the benefits of assessment of printed source code, compared with that of assessing students verbally. Indications would suggest that such interviews would be capricious, since even structured interviews have a degree of subjectivity and may be labour intensive to schedule and carry out for already busy tutors.

Jones suggests using a version of the Glatt plagiarism detection process (Jones, 2001a, 2001b). The Glatt process for free text student submissions involves removing words and asking students to replace them. The argument is that if a student wrote the submission they should be able to place a majority of these words. Jones suggests, but does not evaluate, removing random lines from the code, or introducing known errors into the program that students would then have to correct. This is intended to ensure that students wrote the source code submission themselves and has some programming ability.

An approach suggested by Jones, but not investigated, involves examination of ‘second-order products’ (Jones, 2001a, 2001b). Jones first suggests generating a log of error messages at compilation time. A number of tokens could be determined based on the error messages and singular metrics generated for each log. He also suggests generating a log when running programs with known inputs. Those metrics with high similarity scores could then be investigated for plagiarism. The success of the first approach rests on Jones’ assumption that plagiarising student submissions would generate the same error messages, which is suspect. The second approach depends on students not changing the lines of coding that print values to the screen, which does not need a lot of technical knowledge. The approach also seems like a lot of work as a fresh set of tokenisation strategies needs generating for every new assignment specification for results that are generated in English. Jones needs to be more convincing that his second-order products are feasible indicators of source code plagiarism.

VISUAL TECHNIQUES FOR DETECTION

Generally an approach where student submissions are compared and a list of similar pairs generated can be considered non-visual. This means that there is still a large amount of work for a tutor to work out why the pair of submissions have been considered similar, a particular problem for longer code. Some engines use further visual techniques to show why a pair or cluster of submissions have been flagged. This is an approach advocated by Culwin and Lancaster to reduce tutor workload (Culwin & Lancaster, 2001a, 2001b, 2001c). Existing approaches are described here in overview. A more detailed review is given by Lancaster, including how the techniques might be applicable to free text submission (Lancaster, 2003).

Several tools present a pair of submissions with hyperlinks or colour coding to show where they are similar. The most notable proponents for source code are MOSS and JPlag, as described in the previous section. SHERLOCK also uses a similar technique, presenting two files twice over, with and without comment stripping (Joy & Luck, 1999). Instead of colour coding sections of code are flagged with comments that a given section is suspicious, which seems a less intuitively obvious way of showing similarity. SHERLOCK also includes a simple clustering view.

A pair of linked techniques that plot submissions on a grid are DotPlot (Helfman, 1994) and a related tool DUPLOC (Ducasse, Rieger, & Demeyer, 1999; Rieger & Ducasse, 1998). This is a simplified version of an approach used by the VAST tool for free text submission (Culwin & Lancaster, 2001c). In its most basic form, DotPlot sees lines of code in two tokenised files plotted along the *x* and *y*-axes of a grid. Where the lines match a dot is plotted at the corresponding co-ordinate, where they differ the co-ordinate is left blank. DUPLOC and VAST make this approach interactive, with the VAST approach showing overlapping fragments at different pixel intensities depending on the degree of localised similarity within them and is thought to be more useful than DotPlot.

Two graphics for presenting an entire corpus are also known. These are called patterngrams and are used within the Same tool (Ribler, 1997; Ribler & Abrams, 2000). The two graphics differ slightly, but both work along the idea of a base file, which is plotted alongside the bottom of a graphic, with similarity in all other files shown vertically. Generally the view is not intuitive.

CONCLUSIONS

Source code plagiarism is still an ongoing concern for academic institutions who need to ensure the integrity of their academic awards. There are a number of tried and tested tools and techniques for detection, most of which use paired metrics with tokenisation, generally searching for long common substrings.

It is difficult to recommend one tool above all others. Generally both MOSS and JPlag come highly regarded and are both well used within the academic computing community. Institutions wishing to use an entirely localised detection solution, for instance to avoid issues when submitting student work to an external source, should consider downloading and installing YAP3, although there are few details about how this compares technically with more recent engines. Generally MOSS seems to be the more robust and available solution, although the benefits over JPlag are arguably marginal.

One notable absence in the current detection tools is support for Visual Basic and Prolog, two languages found by Culwin, Lancaster and MacLeod to be highly used but not widely supported (Culwin et al., 2001). There are still techniques that could be evaluated such as Saxon's compressibility (Saxon, 2000). There also appears to be a need for more visual techniques to support the detection process. This also suggests that while source code detection is a well understood area of research there are still plenty of areas for further work that do require attention.

REFERENCES

Bowyer, K.W., & Hall, L.O. (1999). Experience using 'MOSS' to detect cheating on programming assignments. In: *29th ASEE/IEEE Frontiers in Education Conference*, San Juan, Puerto Rico, pp. 18–22.

Campbell, P. (2002). Private communication from South Bank University, London, UK.

Clare, J. (2000). Computer plagiarism 'threatens the value of degrees'. *Daily Telegraph 3/7/2000.* Available online at http://www.telegraph.co.uk/et?ac=004228431730477&rtmo=VDw3wDqK&atmo=rrrrrrrq&pg=/et/00/7/3/ncopy03.html

Culwin, F., & Lancaster, T. (2000). *A descriptive taxonomy of student plagiarism.* Unpublished. Available from South Bank University, London, UK.

Culwin, F., & Lancaster, T. (2001a). *Plagiarism issues for higher education, Vine 123.* Available from LITC, South Bank University, London.

Culwin, F., & Lancaster, T. (2001b). *Plagiarism prevention, deterrence & detection.* Available online at http://www.ilt.ac.uk/resources/Culwin-Lancaster.htm

Culwin, F., & Lancaster, T. (2001c). Visualising intra-corpal plagiarism. In: *Information Visualisation 2001*, London, UK.

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

Culwin, F., & Naylor, J. (1995). Pragmatic anti-plagiarism. In: *Proceedings of the 3rd All Ireland Conference on the Teaching of Computing*, Trinity College, Dublin.

Cunningham, P., & Mikoyan, A.N. (1993). *Using CBR techniques to detect plagiarism in programming assignments.* Available from Department of Computer Science, Trinity College, Dublin.

DetectaCopias. (2002). Available online at http://www.dcc.uchile.cl/∼rmeza/proyectos/detectaCopias/index.html

Ducasse, S., Rieger, R., & Demeyer, S. (1999). A language independent approach for detecting duplicated code. In: *Proceedings of International Conference on Software Maintenance.*

Granville, A. (2002). *Detecting plagiarism in Java code.* Student project. Available from University of Sheffield, UK.

Halstead, M.H. (1977). *Elements of software science.* Amsterdam: Elsevier.

Harris, J.K. (1994). Plagiarism in computer science courses. In: *Proceedings of the 1994 Ethics in Computer Age*, pp. 133–135.

Helfman, J.I. (1994). Similarity patterns in language. In: *Proceedings of IEEE Symposium on Visual Languages*, pp. 173–175.

Irving, R. (2000). Plagiarism detection: Experiences and issues. In: *JISC Fifth Information Strategies Conference: Focus on Access and Security, British Library, London*.

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. (2001a). Plagiarism monitoring and detection – towards an open discussion. In: *7th Annual CSSC Central Plains Conference*, Branson, Missouri, 6–7 April.

Jones, E.L. (2001b). Metrics based plagiarism monitoring. In: *6th Annual CSSC Northeastern Conference*, Middlebury, Vermont, 20–21 April.

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

JPlag. (2002). JPlag. Available online at http://www.jplag.de

Lancaster, T. (2003). *Effective and efficient plagiarism detection.* PhD thesis. Available from South Bank University, London, UK.

Lancaster, T., & Culwin, F. (2003). *Classifications of plagiarism detection engines*. Unpublished. Available from South Bank University, London, UK.

MOSS. (1994). Measure of software similarity. Available online at http://www.cs.berkeley.edu/~aiken/moss.html

Popyack, J., Herrmann, N., Zoski, P., Char, B., Cera, C., & Lass, R. (2003). *Academic dishonesty in a high-tech environment*. Special session, presented at The Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education, Reno, Nevada.

Prechelt, L., Guido, M., & Phlippsen, M. (2000). *JPlag: Finding plagiarisms among a set of programs* (Technical Report 2000–1). Germany: Facultat fur Informatik, Universitat Karlsruhe.

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

Ribler, R.L. (1997). *Visualizing categorical time series data with application to computer and communications network traces*. PhD dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

Ribler, R.L., & Abrams, M. (2000). Using visualisation to detect plagiarism in computer science classes. In: *Information Visualisation 2000*, pp. 173–177.

Rieger, M., & Ducasse, S. (1998). Visual detection of duplicated code. In: *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*.

Sanders, R. (1998). *Online plagiarism detector helps CS professors bust cheating programmers*. Available online at http://www.coe.berkeley.edu/EPA/EngNews/98S/aiken.html

Saxon, S. (2000). *Comparison of plagiarism detection techniques applied to student code, Part II*. Computer Science project, Trinity College, Cambridge.

SIM. (1989). *The software and text similarity engine*. Available online at http://www.cs.vu.nl/~dick/sim.html

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.

Verco, K.L., & Wise, M.J. (1996b). Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In: *Proceedings of First Australian Conference on Computer Science Education*, 3–5 July 1996, Sydney, Australia.

Whale, G. (1990). Identification of program similarity in large populations. *The Computer Journal*, *33*(2), 140–146.

Wise, M.J. (1996). YAP3: Improved detection of similarities in computer program and other texts. In: *1996 SiGCSE Technical Symposium*, Philadelphia, USA, pp. 130–134.