Extending computer-assisted text analysis techniques to the detection of source code plagiarism and collusion: assisting manual inspection

Maeve Paris
School of Computing and Intelligent Systems
University of Ulster

1.0 Introduction

While there have always been opportunities for collusion among students, the Internet explosion has facilitated the ease with which material can be copied and pasted. Fortunately, the academic sleuth has access to a wide range of publicly-available detection software and systems. Unfortunately, these products vary greatly in terms of functionality, usability and effectiveness.

Traditionally, a distinction has been drawn between software and services to detect text-based plagiarism or collusion, and products to detect such practices in computer programs. The Joint Information Services Committee (JISC) even commissioned two separate studies, the *Technical Review of Plagiarism Detection Report* (Bull et al, 2001) which focused on text-based assignments, and *Source Code Plagiarism in UK HE Computing Schools, Issues, Attitudes and Tools* (Culwin et al, 2001), which concentrated on source code plagiarism. The text-based survey evaluated the performance of five products with mixed results, while the source code survey evaluated two products, with different strengths and weaknesses. This separation of concerns is not spelled out explicitly in either report: both seem to be based on an assumption that the practice of plagiarism and collusion differs between text-based documents and examples of computer programs written in a particular programming language.

This paper argues for a change of focus: a computer programming language should be treated in similar fashion to any natural language. While the syntax and semantics of programming languages are more formal and restricted than those of natural languages, it is still possible to distinguish a programming (and hence a programmer's) style, much in the same way as one can attempt to distinguish a writing style in a written text or corpus. It could follow that source code samples might be suitable for the application of computer-assisted text analysis techniques. Concordances, the use of KWIC indexes, and other statistical methods from the domain of computational linguistics can all be employed to assist the academic in a computing school in identifying instances of plagiarism or collusion among samples of students' source code.

This paper will define source code with examples from different programmers, with a view to illustrating the existence of programming styles, and identify shortcomings in existing detection techniques. It will indicate how computer-assisted text analysis might assist the academic in manual inspection of students' source code, by augmenting human expertise. An analysis based on authentic examples from the Java programming language will test the suitability of a concordance programme in assisting an academic. The study will attempt to ascertain whether the use of a concordance programme will accelerate the process of plagiarism and collusion detection, and possibly even make it more accurate.

2. 0 Plagiarism and collusion

For teaching and assessment purposes, a distinction can be drawn between the offences of plagiarism and collusion. Plagiarism is where a student submits work for assessment which has been borrowed from another writer but fails to indicate (whether by the use of quotation marks or explicit referencing) which sections or phrases have been borrowed. Collusion is usually the manifestation of a joint effort between students where the intention is to mislead the assessor in identifying the person responsible for writing the material. Whether it is a case of reproducing someone's work or producing joint work under different names, the omission of any acknowledgement is of most concern.

While the aim of this study is to consider methods for detecting plagiarism and collusion, this does not extend to the domain of authorship attribution (determining the author of a piece of work). This is distinct from plagiarism and collusion detection: 'plagiarism detection attempts to detect the similarity between two substantially different pieces of work but is unable to determine if they were produced by the same author' (deVel et al, 2001). However, these fields overlap, since both authorship analysis and plagiarism detection are concerned with similarity detection, which 'calculates the degree of similarity between two or more pieces of work without necessarily identifying the authors' (deVel et al, 2001).

3.0 Source code

Source code refers to the set of programming statements written by a programmer in a particular programming language. This code is then compiled into object code: 'source code and object code refer to the "before" and "after" versions of a computer program that is compiled before it is ready to run in a computer' (SearchWebServices.com, 2001).

A Java programmer, for example, types a set of statements in the Java programming language into a visual development environment (or a simple text editor). These statements are then saved into a file, the source file. In the case of Java, the source file is then compiled into instructions that the Java Virtual Machine (Java VM) can understand, in the shape of a bytecode file. It is this program which can be run in the computer.

Figure 1 is a sample of Java source code which most learners encounter. This program outputs the phrase 'Hello World!' to the screen.

```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Figure 1: Hello World!

Even this trivial example highlights some characteristics of source code which are of interest to this study: different ways to comment on code (using //… or /*…*/), indentation (through the placing of { and }, for example), naming schemes (the class is named HelloWorldApp), and system output (seen through the print instruction).

3.1 Source code as a means of expression

Programming languages can be categorised according to generation (from first generation machine language to fourth generation SQL), or type (such as procedural or declarative), but all languages typically consist of a vocabulary and syntax. While these may be restricted in nature, they can be treated as a form of language from a linguistic perspective.

To illustrate this claim, consider the examples in Figures 2, 3 and 4 below. All are suggestions as to how to write a function that prints 'Hello World!' 100 times:

```
for(int j=0; j<100; j++)
{
system.out.println(printFunction());
}
//this function print Hello World 100 times without using for loop and recursion...
public string void printFunction()
{
return("Hello World");
}
```

Figure 2: Java program

```
public class foo
{
  int i = 0;

  public void bar()
  {
     System.out.println("Hello World");
     if(++i < 100)baz();
  }

  public void baz()
  {
     System.out.println("Hello World");
     if(++i < 100)bar();
  }
}
```

Figure 3: Java program

```
String[] strArray = new String[100];
  java.util.Arrays.fill(strArray, "Hello World");
  String str = java.util.Arrays.asList(strArray).toString();
  str = str.substring(1, str.length()-1);
  str = str.replace(',', '\n');
  System.out.println(str);
```

Figure 4: Java program

All of these programs have the same functionality; they all have the same outputs, but all solve the problem differently. Different algorithms are used, and different styles are used. The first example uses a *for* loop, the second uses an *if* loop, the third tries to avoid loops and recursion altogether. Only the first uses comments, while only the second uses indentation to enhance readability. The second also makes greater use of white space to separate off lines of code. There are different variables used and different method/ class names. From this small example, it is possible to see how the same functionality can be implemented differently by different programmers in the same language.

These statements provide instructions for the computer to carry out a particular function, broken down into different sections and sequences, some of which are conditional. While the programming language provides a limited vocabulary and syntax, the individual programmer has a lot of flexibility in determining how the functionality will be achieved, from choice of algorithm (how the function is achieved), to naming of variables and classes, to layout on the screen (use of spacing, indentation, whitespace, and so on). There is considerable flexibility in the choice of variables, expressions, statements and blocks, and control flow statements.

Sallis et al (1996) identified this flexibility in the context of authorship attribution: 'the stylistic influence of an individual on algorithm implementation within the constraints of a given programming language is limited but can be identified to some extent as traits or tendencies in the expression of logic constructs, data structure definition, variable and constant names and calls to fixed and temporary data sets.' (Sallis et al, 1996). More recently, de Vel et al (2001) asserted that 'it is possible to identify the author of a section of program code in a similar way that linguistic evidence can be used for categorising the authors of free text' (deVel et al, 2001).

The issue of source code as a means of expression has also come to the fore in the United States through the Digital Millennium Copyright Act (DCMA), which was intended to protect publishers from electronic piracy. One major court case has raised the question whether code can be defined as speech. In January 2000, the Motion Picture Association of America filed a lawsuit against the magazine 2600 based on the charge that 2600 had violated the DCMA by publishing the DeCSS decryption routine, a code for decrypting DVDs. A computer scientist at Carnegie Mellon, David Touretsky, gave expert evidence arguing that computer code has expressive content which can convey ideas, like other forms of speech. Judge Kaplan agreed that 'both source and object code have expressive content, and thus deserve First Amendment protection. Code really *is* speech' (Touretsky, 2001).

4.0 Existing detection systems

Source code plagiarism detection systems are generally based on metrics and aim to produce a measurement which quantifies the closeness of two programs. Metrics (both software metrics and linguistic metrics) can be gathered on a range of items, including the number of each type of data structure, the cyclomatic complexity of the control flow of the program, the quantity and quality of comments, the types of variable names chosen, and the use of layout conventions. All of these can assist in building a profile of a particular authoring style (Gray et al, 1997). The JISC report (Culwin et al, 2001) distinguished between early systems based on attribute counting, and those based on structure metrics, on which most modern systems are based and which produce more effective results.

Jones (2000) characterised plagiarism detection as a pattern analysis problem, where plagiarising transformations have been applied to a source file. He identified the following transformations:

- Verbatim copying
- Changing comments
- Changing white space and formatting
- Renaming identifiers
- Reordering code blocks
- Reordering statements within code blocks
- Changing the order of operands/operators in expressions.
- Changing data types.
- Adding redundant statements or variables.
- Replacing control structures with equivalent structures.

(Jones, 2000)

Most contemporary detection systems adopt a lexical-structural approach to identify these transformations: source programs are tokenised, and profiles are created and compared. While some academic institutions have developed their own in-house detections systems, such as Big Brother (Irving, 2002), there are also services available through a Web interface. The main players in this field are sim (Software Similarity Tester), YAP (Yet Another Plague), MOSS (Measure of Software Similarity) and JPLAG. The sim system (Gitchell and Tran, 1999) tokenises source programs and compares strings using pattern-matching algorithms based on work from the human genome project.

The YAP (Wise, 1996) approach also tokenises source programs but only retains those tokens which are concerned with the structure of the program. This is based on a lexicon which is created specifically for each programming language. The output is a numeric profile which computes the closeness between two programs. This closeness between programs is partly a function of the programming language chosen and the type of task undertaken (for instance, the COBOL programming language is by nature a highly structured and verbose language, and can lead to very similar programs, likewise with Visual Basic).

MOSS (MOSS, 2002) can be applied to a range of programming languages. Registered instructors can submit batches of programs to the MOSS server, and results are returned to a website. Little information is available on how the tool works (presumably because if this were known, it would be possible to evade detection), but it is based on the syntax or structure of a program, rather than the algorithms which drive the program (Stutz, 1998). The MOSS database stores an internal representation of programs, and then looks for similarities between them.

JPLAG, on the other hand, compares submitted programs in pairs, and is based on the assumption that plagiarists may vary the names of variables or classes, but they are least likely to change the control structure of a program. The algorithm employed is available as a technical report (Prechelt et al, 2000).

The performance of both MOSS and JPLAG were evaluated in the JISC report (Culwin et al, 2001). The survey concluded that there was not much consensus between both engines in terms of identifying instances of plagiarism, that JPLAG was easier to use but supported fewer languages than MOSS and could not deal with programs which do not parse; ‘the results returned from each system are apparently widely different’ (Culwin et al, 2001). As students often submit files which do not parse, such a limitation would mean that many files would be not be under consideration. A performance comparison of JPLAG, MOSS, and Sherlock (Warwick University’s detection system) produced by the LTSN (LTSN, 2002) also revealed patchy performance and inconsistencies between the tools on items such as changing comments and replacing expressions by equivalents. In addition, neither JPLAG nor MOSS is easy to use: JPLAG requires that the user have Netscape 4 or use an applet, while MOSS requires configuration of perl files on a UNIX account.

However, the JISC report also noted that ‘the tools are not intended as tests for plagiarism. They supply an ordered list of apparent similarities that allow a tutor to more efficiently locate the parts that should be examined to determine if they require further investigation’ (Culwin et al, 2001).

Another report (Stutz, 1998) noted that tools such as JPLAG which rely on control structure metrics to detect pairwise similarities are problematic: ‘these primitive constructs – the IF, THEN and ELSE statements- are used in about the same ratio in just about every program. The end result is that plagiarism detection software that uses this scheme is prone to generate false positives’ (Stutz, 1998).

5.0 Computer-assisted text analysis techniques

Given the patchy performance of existing tools (Culwin et al 2001, LTSN 2002), perhaps the separation of concerns into source code and text is of limited use to the practising academic, and it may be that techniques from CATA are of greater use in detecting instances of plagiarism. Some commentators have speculated on potential of this area, although sometimes from the opposite perspective: Clough (2000) identified similarities between methods used for source code and text plagiarism detection, including ‘replacement of synonyms, re-ordering of sentences, insertion and deletion of text, change of author style, etc.’ He concluded that ‘methods used for software plagiarism detection may well work for text also’ It should also be argued that if there are similarities between detection of plagiarism in both areas, the methods used for text plagiarism detection may well work for software plagiarism detection also.

Sallis et al (1996) observed that work in the domain of computational linguistics relating to the issue of authorship attribution based on text corpora has parallels for source code, and they advocated ‘a combination of techniques from conventional software metrics and computational linguistics’ (Sallis et al, 1996).

Spelling and grammar measurements may be useful in identifying similarity between programs, and tools developed for computer-assisted text analysis would be able to assist in their identification: ‘many programmers have difficulty writing correct prose. Misspelled variable names (e.g. TransactoingReciept) and words inside comments may be quite telling if the misspelling is consistent. Likewise, small grammatical mistakes inside comments or print statements, such as misuse or overuse of em-dashes and semicolons might provide a small additional point of similarity between two programs’ (Spafford & Weaver, 1993).

6.0 Analysis

The intention of this analysis was to investigate the extent to which a concordance program such as Concordance (Watt, 2002) could assist in the detection of similarities in source code programs. The role of the tool is to assist the experienced academic, and was also compared with a manual inspection of the same files. The Concordance program offers a user interface to the analysis and also offers a web-based version of the concordance produced.

In order to accomplish the analysis, a corpus of Java source code programs was obtained from first year BSc Computer Science students. The corpus was assembled from a set of files which were created in response to the following section of an assignment:

> (a) Develop a static method which takes a percentage value as a parameter and returns a letter grade according to the following criteria:
>
> - 80% or greater = ‘A’
> - 60% - 79% ‘B’
> - 40% - 59% = ‘C’
> - less than 40% = ‘D’
>
> (b) Write a program which:
>
> - prompts for and reads a student’s name and percentage gained
> - uses the method in (a) to determine the grade obtained
> - prints to the screen the student’s name and the grade achieved
>
> (Sayers, 2002)

A corpus of eleven files were used for this analysis, and three approaches were taken: manual inspection of the files, use of JPLAG, and use of the Concordance program. Two of the files, StudentGrade.java and GradeWork.java were similar, apart from a difference in class name and substitution of ‘got’ for ‘achieved’ in the output. Prac7.java and Percentage.java had an identical static method to convert percentages to grades, which was also unusually back-to-front in its approach so would be expected to attract the attention of an assessor. The other files had trivial similarities, and some would not compile successfully in order to run the program.

6.1 Manual Inspection

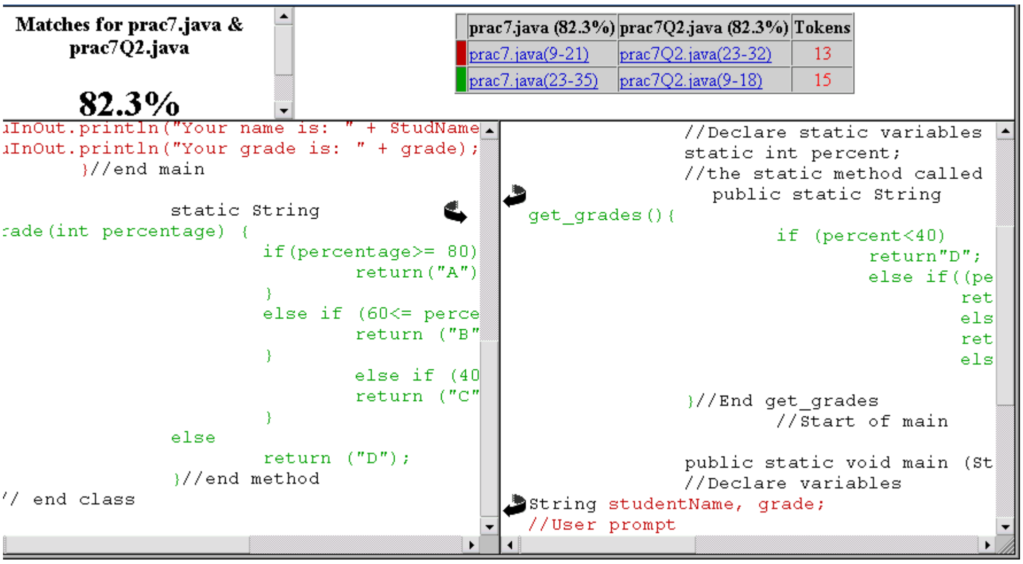
The files were presented as they are submitted in printed form to an assessor who identified the two very similar files in five minutes and took a further ten minutes to establish that there were similarities in Prac7.java and Percentage.java. The assessor felt that this method would be too time-consuming for a larger group of files, especially as papers had to be shuffled around, and the variations in indentation meant that it was not easy to keep track of algorithms. The assessor noted that she was particularly looking for similar variable or class names, similar loops, programs with comments removed, or unusual data types or naming schemes.

6.2 JPLAG

The eleven files were submitted to the JPLAG server through a downloadable Java applet. This was relatively easy to set up, and results were returned to a webpage. JPLAG unfortunately rejected all but three of the files on the grounds that they did not parse. This meant that JPLAG failed to consider the majority of files submitted. It generated the following results:

prac7Q2.java->prac7.java (82.3%) question4.java (61.5%)
question4.java->prac7.java(46.1%)

This had not been revealed by the manual inspection. A consideration of the 82.3% similarity between the two files revealed the following screen:



Matches for prac7.java & prac7Q2.java

82.3%

	prac7.java (82.3%)	prac7Q2.java (82.3%)	Tokens
🟥	prac7.java(9-21)	prac7Q2.java(23-32)	13
🟩	prac7.java(23-35)	prac7Q2.java(9-18)	15

```
ıInOut.println("Your name is:  " + StudName
ıInOut.println("Your grade is:  " + grade);
       }//end main

               static String
:ade(int percentage) {
                        if(percentage>= 80)
                                return("A")
                        }
                        else if  (60<= perce
                                return  ("B"
                        }
                                else if  (40
                                return  ("C"
                        }
               else
                        return  ("D");
               }//end method
'/  end class
```

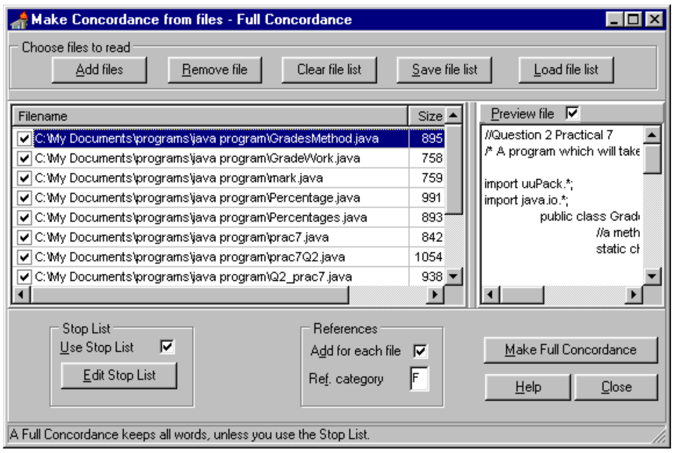
```
                    //Declare static variables
                    static int percent;
                    //the static method called
                      public static String
 get_grades(){
                            if  (percent<40)
                                    return"D";
                                    else if((pe
                                            ret
                                            els
                                            ret
                                            els

                    }//End get_grades
                            //Start of main

                    public static void main (St
                    //Declare variables
 String studentName, grade;
 //User prompt
```

Figure 5: JPLAG results window

Closer examination by the assessor revealed that all that these sections shared was the control structure, but the way the structure was used was different in each case. The assessor felt that these could not be considered as similar for the purposes of plagiarism or collusion detection. JPLAG failed to identify the other clearer example of similarity, as the program would not parse. These are two drawbacks of the JPLAG approach.

6.3 Concordance

The eleven programs were input to the Concordance (Watt, 2002) tool, which can be easily adapted to use for computer programs. Some changes need to made before requesting a concordance: the existing stop list was modified to take into account the most frequent words which are to be found in Java programs (public, static void, main, and so on), and changes were made to definitions of what constitutes a word. Unlike other concordance applications, the Concordance tool had no problem dealing with java files and batch converted them to text files while retaining file and line information. The interface was simple and compact (Figure 6).



Filename	Size
☑ C:\My Documents\programs\java program\GradesMethod.java	895
☑ C:\My Documents\programs\java program\GradeWork.java	758
☑ C:\My Documents\programs\java program\mark.java	759
☑ C:\My Documents\programs\java program\Percentage.java	991
☑ C:\My Documents\programs\java program\Percentages.java	893
☑ C:\My Documents\programs\java program\prac7.java	842
☑ C:\My Documents\programs\java program\prac7Q2.java	1054
☑ C:\My Documents\programs\java program\Q2_prac7.java	938

Preview file ☑

```
//Question 2 Practical 7
/* A program which will take

import uuPack.*;
import java.io.*;
            public class Grad
                        //a meth
                        static cl
```

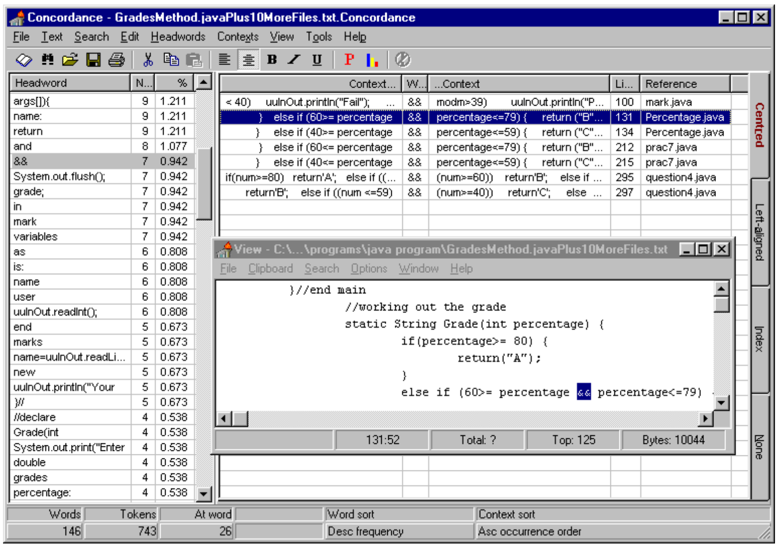
Stop List: Use Stop List ☑ Edit Stop List

References: Add for each file ☑ Ref. category F

A Full Concordance keeps all words, unless you use the Stop List.

Figure 6: selecting files

To speed the analysis it is possible to select only those words which occur more than once, as those appearing once only have no relevance to this study. The results are displayed, sorted by descending frequency, as in Figure 7 below.



Headword	N...	%
args[]){	9	1.211
name:	9	1.211
return	9	1.211
and	8	1.077
&&	7	0.942
System.out.flush();	7	0.942
grade;	7	0.942
in	7	0.942
mark	7	0.942
variables	7	0.942
as	6	0.808
is:	6	0.808
name	6	0.808
user	6	0.808
uulnOut.readInt();	6	0.808
end	5	0.673
marks	5	0.673
name=uulnOut.readLi...	5	0.673
new	5	0.673
uulnOut.println("Your	5	0.673
}//	5	0.673
//declare	4	0.538
Grade(int	4	0.538
System.out.print("Enter	4	0.538
double	4	0.538
grades	4	0.538
percentage:	4	0.538

Context...	W...	...Context	Li...	Reference
< 40) uulnOut.println("Fail"); ...	&&	modm>39) uulnOut.println("P...	100	mark.java
} else if (60>= percentage	&&	percentage<=79) { return ("B"...	131	Percentage.java
} else if (40>= percentage	&&	percentage<=59) { return ("C"...	134	Percentage.java
} else if (60<= percentage	&&	percentage<=79) { return ("B"...	212	prac7.java
} else if (40<= percentage	&&	percentage<=59) { return ("C"...	215	prac7.java
if(num>=80) return'A'; else if ((...	&&	(num>=60)) return'B'; else if ...	295	question4.java
return'B'; else if ((num <=59)	&&	(num>=40)) return'C'; else ...	297	question4.java

View - C:\...\programs\java program\GradesMethod.javaPlus10MoreFiles.txt

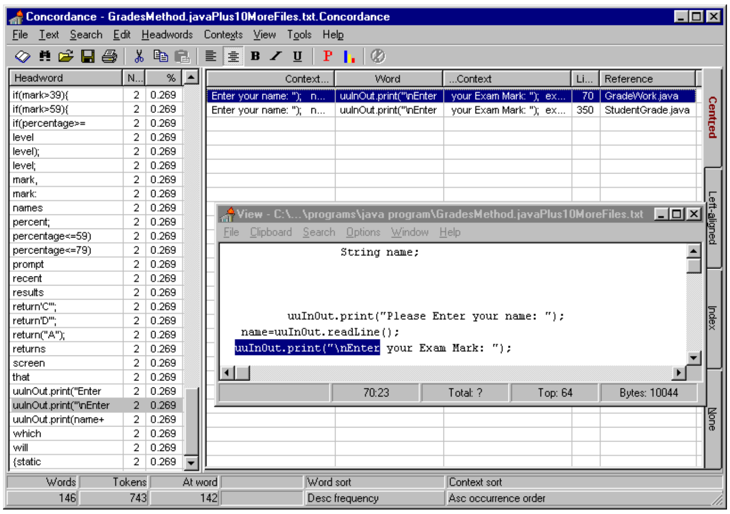
```
        }//end main
            //working out the grade
            static String Grade(int percentage) {
                if(percentage>= 80) {
                    return("A");
                }
                else if (60>= percentage && percentage<=79)
```

131:52 Total: ? Top: 125 Bytes: 10044

Words	Tokens	At word		Word sort	Context sort
146	743	26		Desc frequency	Asc occurrence order

Figure 7: Concordance window

The assessor could select suspect headwords on the left, and the keywords appeared in context on the right. To investigate the program further, a simple click on the line brought up the program concerned in a window. As the files are referenced it is easy to see where problems arise. In the above example, similarities are revealed between Prac7.java and Percentage.java, and in the example below, Figure 8, there are clear points of comparison between StudentGrade.java and GradeWork.java. These are the results which the manual inspection also revealed.


Concordance - GradesMethod.javaPlus10MoreFiles.txt.Concordance
File Text Search Edit Headwords Contexts View Tools Help
B I U P
Centred
Left-aligned
Index
None

Headword	N...	%
if(mark>39){	2	0.269
if(mark>59){	2	0.269
if(percentage>=	2	0.269
level	2	0.269
level);	2	0.269
level;	2	0.269
mark,	2	0.269
mark:	2	0.269
names	2	0.269
percent;	2	0.269
percentage<=59)	2	0.269
percentage<=79)	2	0.269
prompt	2	0.269
recent	2	0.269
results	2	0.269
return'C'";	2	0.269
return'D'";	2	0.269
return("A");	2	0.269
returns	2	0.269
screen	2	0.269
that	2	0.269
uulnOut.print("Enter	2	0.269
uulnOut.print("\nEnter	2	0.269
uulnOut.print(name+	2	0.269
which	2	0.269
will	2	0.269
{static	2	0.269

Context...	Word	...Context	Li...	Reference
Enter your name: "); n...	uulnOut.print("\nEnter	your Exam Mark: "); ex...	70	GradeWork.java
Enter your name: "); n...	uulnOut.print("\nEnter	your Exam Mark: "); ex...	350	StudentGrade.java

View - C:\...\programs\java program\GradesMethod.javaPlus10MoreFiles.txt
File Clipboard Search Options Window Help

```
                    String name;



           uuInOut.print("Please Enter your name: ");
  name=uuInOut.readLine();
 uuInOut.print("\nEnter your Exam Mark: ");
```

	70:23	Total: ?	Top: 64	Bytes: 10044

Words	Tokens	At word		Word sort	Context sort
146	743	142		Desc frequency	Asc occurrence order

Figure 8: Detecting similarities

The assessor had no experience of a concordance application and took some time to learn how to use it, but was pleased with the result, especially as it was quick and easy to identify real instances of similarity. The Concordance program was judged to have assisted the assessor in locating instances of similarity quickly and accurately, and enabled her to identify the files where there was collusion or plagiarism. One drawback was having to scroll through the list of headwords in order to make judgements concerning the selection of words to check; in addition, the program needed to be configured to handle computer files, and this took some time.

7.0 Conclusions

Traditionally there has been a separation of concerns into text and source code plagiarism detection systems, but perhaps it would be more effective to merge both concerns, by using software metrics to inform computational linguistics (and vice versa). If one accepts that source code is a language (albeit one with a restricted lexicon and syntax), this preliminary study indicates that the detection of similarity in programs might benefit from techniques from the world of computer-assisted text analysis. It may be that our student plagiarists tend to make amendments to the very items which source code detection systems ignore: they tend to leave control structures largely intact, but make changes to variables, or class names. Many student assignments are relatively short, and one would expect a preponderance of certain types of control structures, so any judgements made on these alone would not be accurate as the JPLAG example showed. This analysis will lead to a larger study with more files under consideration and longer programs, with a longer-term view to the development of a tool which would combine features of a concordance with metric-based profiles.

Existing tools for detecting similarities in source code are useful but do not necessarily yield the expected results. Some of this is due to the fact that the practising academic has to deal with many examples of programs, some of which may not be capable of being compiled, but are just as susceptible to being examples of collusion or plagiarism. Some of this is also due to the algorithms which support these tools and are often based on control structures. In the end, of course, it is not the role of tools such as JPLAG or Concordance to judge plagiarism: they merely indicate its possibility, and so they are tools for the user to assist in its detection. ‘Although attempts have been made at automatically detecting plagiarism, it will still require the intervention of a human to prove the plagiarism and provide the final verdict’ (Clough 2000). It is up to the user to decide if plagiarism or collusion has taken place, but some combination of software metrics and computer-assisted text analysis might prove more effective.

References:

Bull J, Collins C, Coughlin E, Sharp D 2001*Technical Review of Plagiarism Detection Report* JISC.

Clough P 2000 *Plagiarism in natural and programming languages: an overview of current tools and technologies*. [online] University of Sheffield. Available at: http://www.dcs.shef.ac.uk/~cloughie/plagiarism/HTML_Version/ [Accessed 18 December 2003].

Culwin F, MacLeod A, Lancaster T 2001 *Source Code Plagiarism in UK HE Computing Schools, Issues, Attitudes and Tools* JISC.

DeVel O, Anderson A, Corney M, Mohay GM 2001 Mining Email Content for Author Identification Forensics. *SIGMOD RECORD* 30(4): 55-64.

Gitchell D, Tran N 1999 Sim: a utility for detecting similarity in computer programs**.** *Proceedings of the thirtieth SIGCSE technical symposium on computer science education* New Orleans, Louisiana, pp 266-270.

Gray A, Sallis P, MacDonell S 1997 Software Forensics: Extending Authorship Analysis Techniques to Computer Programs. *Proceedings of the 3rd Biannual Conference of the International Association of Forensic Linguists* pp 1 – 8.

Irving R 2000 Plagiarism Detection: Experiences and Issues. *JISC Fifth Information Strategies Conference, Focus on Access and Security*, London.

Jones E 2000 Plagiarism monitoring and detection – towards an open discussion. *Proceedings of 7th annual CCSC Central Plains Conference,* Branson, Missouri*,* April 6-7.

LTSN 2002 *Tools to assist detecting plagiarism* [online]. Available at: http://www.dcs.warwick.ac.uk/ltsn-ics/resources/plagiarism/tools/comparison_performance.html [Accessed 1 February 2003].

MOSS 2002 Available at: http://www.cs.berkeley.edu/~aiken/moss.html

Prechelt L, Malpohl G, Philippsen M 2000 *JPlag: Finding plagiarisms among a set of programs*, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation.

Sallis, P, Aakjaer A, MacDonell 1996 Software Forensics: old methods for a new science. *Proceedings of Software Engineering: Education and Practice* IEEE Computer Society Press, pp 481-485.

Sayers H 2002 *Coursework for Com 158M1* University of Ulster.

SearchWebServices.com 2001 *Source code* [online] Available at: http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci213030,00.html [Accessed 15 January 2003].

Stutz M 1998 Catching Computer Science Cheaters. *Wired News* [online] Available at: http://www.wired.com/news/technology/0,1282,10464,00.html [Accessed 18 January 2003].

Touretsky DS 2001 Free Speech Rights for Programmers. *Communications of the ACM*, 44(8).

Watt RJC 2002 *Concordance*. [online] Available at: http://www.rjcw.freeserve.co.uk/ .

Weeber SA, Spafford EH 1993 Software forensics: can we track code to its authors? *Computers & Security* 12(6): 585-595.

Wise M 1996 YAP3: improved detection of similarities in computer programs and other texts. *SIGCSE'96*, Philadelphia, Feb 16 – 17.