Research Report 345

Plagiarism in Programming Assignments

M Joy and M Luck

RR345

The number of students following programming courses is steadily increasing at the same time as access to computers and networks is readily available. There is a significant minority of students who -- for a variety of reasons -- take advantage of the available technology and illicitly copy other students' programming assignments and attempt to disguise their deception. Software that can help tutors to detect plagiarism is therefore of immense assistance in detecting -- and so helping to prevent -- such abuse.

We introduce a novel approach to designing such software which performs well in comparison to sophisticated software available elsewhere, yet is simple both in concept and in implementation. Our approach reduces substantially the effort needed to upgrade to new programming languages, and has been tested on a variety of classes and several different programming languages.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

June 1998

Plagiarism in Programming Assignments

Mike Joy and Michael Luck,
Department of Computer Science,
University of Warwick,
COVENTRY,
CV4 7AL,
UK
email: {M.S.Joy,Michael.Luck}@dcs.warwick.ac.uk

June 19, 1998

Abstract

The number of students following programming courses is steadily increasing at the same time as access to computers and networks is readily available. There is a significant minority of students who – for a variety of reasons – take advantage of the available technology and illicitly copy other students' programming assignments and attempt to disguise their deception. Software that can help tutors to detect plagiarism is therefore of immense assistance in detecting – and so helping to prevent – such abuse.

We introduce a novel approach to designing such software which performs well in comparison to sophisticated software available elsewhere, yet is simple both in concept and in implementation. Our approach reduces substantially the effort needed to upgrade to new programming languages, and has been tested on a variety of classes and several different programming languages.

1 Introduction

The numbers of students following computer programming courses — either within a computing degree or as part of another course — are increasing. One consequence of this increase in numbers is a corresponding increase in difficulty in detecting isolated instances of students engaging in unacknowledged collaboration or even copying of coursework.

Assessment of programming courses typically involves students writing programs, either individually or in teams, which are then marked against criteria such as correctness and style. Unfortunately, it is very easy for students to exchange copies of code they have written. A student who has produced working code may be tempted to allow a colleague to copy and edit their program. This is discouraged, and is likely to be regarded as a serious disciplinary offence.

It is not sufficient to remind students of regulations forbidding plagiarism; they must understand that it *will* be detected, and that it will not be condoned. However, it is easy for a lecturer to fail to detect plagiarism, especially when class sizes are measured in hundreds of students.

Automation provides a means with which to address these concerns [10]. Much of the program submission, testing and marking process has the potential to be automated, since programs are, by definition, stored in a machine-readable form. We have been developing software which will allow students to submit programming assignments on-line. An integral part of our software consists of a module to assist in the detection of instances of possible plagiarism, using a simple but novel technique. In this paper, we discuss the software and its implications for the management of large courses.

2 What is Plagiarism?

Plagiarism — unauthorised copying of documents or programs — occurs in many contexts. In *industry*, a company may seek competitive advantage; *academics* may seek to publish results of their research in advance of colleagues. In these instances, the issue is treated very seriously by both parties, and the person performing the unauthorised copying may be backed by significant technical and/or financial resources. Detection becomes correspondingly difficult.

Alternatively, *students* may attempt to improve marks in assessments. They are, however, unlikely to enjoy financial or technical support for such an activity,

and the amount of time available to them is short. The methods used to conceal copied work are therefore, in general, unsubtle. Only moderately sophisticated software tools are required to isolate potential instances of plagiarism.

There are many reasons for students copying material from each other, or colluding in producing a specific piece of work. These include the following:

- A weak student produces work in close collaboration with a colleague, in the belief that it is acceptable.
- A weak student copies, and then edits, a colleague's program, with or without the colleague's permission, hoping that this will go unnoticed.
- A poorly motivated (but not necessarily weak) student copies, and then edits, a colleague's program, with the intention of minimising the work needed.

In the first case, the students concerned are treading on a potentially grey area of acceptability — we expect, and desire, that students should share knowledge, thereby reinforcing the learning process. The boundary between plagiarism and legitimate cooperation is poorly defined, and some students may be used to different customs and norms. It is, nevertheless, still necessary to discover collaboration, so that any misunderstandings are resolved, and so that extra resources can be targetted at the students involved, if though it appropriate.

In the second case, the weak student is likely to have a poor understanding of the program they have edited, and the similarities between the two programs are likely to be strong. Not only may disciplinary action be required, but also — and perhaps more importantly — the tutor has been alerted that remedial tuition may be required to assist the weak student.

In the final case, it may be that the student has very good knowledge of the subject, and is able to make sophisticated modifications to the original program. Such a student will be more difficult to identify; it might be argued that if a student can edit a program so much that it is undetected, then that very act is itself a substantial software development task. However, a genuinely lazy student is unlikely to fall into that category.

It must be realised that it is *always* possible for undetectable plagiarism to occur, no matter how sophisticated the tools available. There is a tradeoff on the part of the lecturer between the resources invested in detecting plagiarism, and the diminishing returns of finding the few (if any) cases which are difficult to

detect. The dishonest student must also balance the work needed to conceal their plagiarism against the effort to create a piece of coursework on their own.

2.1 Techniques for Plagiarism

It is not feasible to classify *all* possible methods by which a program can be transformed into another of identical (or similar) functionality — such a task would be open-ended, as the number of languages available is steadily growing. However, two common transformation strategies can be identified.

2.1.1 Lexical Changes

Lexical changes are those which could, in principle, be performed by a sophisticated text editor. They do not require knowledge of the language sufficient to parse a program. For instance, all of the following come under this banner.

- Comments can be reworded, added and omitted.
- Formatting can be changed.
- Identifier names can be modified.
- Line numbers can be changed (in languages such as FORTRAN).

2.1.2 Structural Changes

A *structural change* requires the sort of knowledge of a program that would be necessary to parse it. It is highly language-dependent. Some examples (appropriate to the language Pascal) are given below.

- Loops can be replaced (e.g. a `while...do` loop in Pascal can be substituted for a `repeat...until` loop).
- Nested `if` statements can be replaced by `case` statements, and vice-versa.
- Statement order can be changed, provided this does not affect the meaning of the program.
- Procedure calls may be replaced by function calls, and vice-versa.

- Calls to a procedure may be replaced by a copy of the body of the procedure.
- Ordering of operands may be changed (e.g. `x < y` may become `y > x`).

2.2 The Burden of Proof

Not only do we need to detect instances of plagiarism, we must also be able to demonstrate *beyond reasonable doubt* that those instances are not chance similarities.

In our experience, most students who plagiarise do so because they *do not understand* fully how to program. The modifications they make — once spotted — are usually sufficiently obvious that they will readily admit their guilt.

If modifications to a program have been made which are so large as to radically alter the structure of the program, then it is difficult, if not impossible, to prove a charge of plagiarism to a disciplinary officer. However, there is small incentive for a student to engage in such a significant modification, since the time and effort required would be of a simliar magnitude to that involved in writing the program afresh.

3 Techniques for Detection

The ability to detect instances of similar programs can be distilled into being able to decide whether or not a *pair* of programs are sufficiently similar to be of interest. Management of a larger collection of programs is a topological exercise [11]. That is, consider a graph whose nodes represent programs, and an arc denotes that the nodes it joins have been detected as "similar". A connected sub-graph represents a collection of programs all of which may be related. Note, however, that the similarity relationship is not transitive. if programs A and B are similar, and also B and C are similar, it may not be the case that our detection mechanism would positively identify the similary when given as input the pair of programs consisting of just A and C. Detection and analysis of clusters of programs represented by connected sub-graphs enables us to discover groups of more than two similar programs.

There are two principal comparison techniques.

- Calculate and compare *attribute counts*[7, 1, 2]. This involves assigning to each program a single number representing capturing a simple quanti-

tative analysis of some program feature. Programs with similar attribute counts are potentially similar programs. The size of a program, for example, would be a very simple attribute count. These metrics can be combined so that each program is assigned a tuple of numbers, and programs are considered similar if most or all of the corresponding tuple elements are similar. Such measures as counts of operators and operands are typically used to construct attribute counts, and more sophisticated but related metrics such as cyclomatic complexity[6] and scope number[3] have been examined.

- Compare programs according to their *structure*[8, 5]. This is a potentially more complex procedure than comparing attribute counts, and depends fundamentally on the language in which the programs are written.

Whale [11, 12] and Verco [10] have carried out a detailed comparison of various attribute count and structure comparison algorithms. They conclude that attribute count methods alone provide a poor detection mechanism, outperformed by structure comparison, while the structure comparison software developed by Whale (*Plague*) [11] and Wise (*Yap*) [13]. report a high measure of success, a recurring feature of structure comparison software is its complexity, and a detailed understanding is required of the target language. Wise reports 2.5 days to adapt Yap to handle the Pascal language rather than C [13], for instance.

A system which incorporates sophisticated comparison algorithms is, by its nature, complex to implement, potentially requiring the programs it examines to be fully parsed. The investment in resources to produce such a system is heavy, but this may be justifiable in the commercial context, if it is necessary to prove copyright violation. In an educational context, the effort expended by students to hide their plagiarism is likely to be much less. Furthermore, students will not necessarily use a single programming language throughout their degree course, and any detection software must be readily upgradeable to handle new languages and packages. There is, therefore, a need for a relatively simple method of program comparison which can be updated for a new programming language with minimal effort, and yet which is sufficiently reliable to detect plagiarism with a high probability of success.

4 The Warwick Approach

Several criteria were isolated which we felt essential to a robust and practical package.

- The program comparison algorithm must be reliable — ideally a structure comparison method.
- The program comparison algorithm must be simple to change for a new language.
- The lecturer using the package must have an easy-to-use interface (preferably with graphical output) to enable them to isolate potential instances of plagiarism rapidly.
- The output from the package must be in a form which is clear to someone unfamiliar with the programs it is examining. If two students are suspected of being involved in plagiarism, clear evidence needs to be presented both to them *and* and to a third party (such as a disciplinary officer) who might need to become involved.

In order to preserve the correct functioning of a copied program, only limited editing can be performed, unless the person copying the program already understands well how it works. It is thus reasonable to assume that some lexical changes, as described above, would probably be implemented, together with a limited number of structural changes.

We might expect, then, that by filtering out all this information, and reducing a program to mere *tokens* or primitive language components, similarities would become apparent. Even with substantial structural changes, we would expect there to be significantly large sections of the programs which are tokenwise the same. In practice, this filtering process removes much data. For simpler programs typical of introductory programming courses, students have a limited choice of algorithms to use, and tokenised representations of their programs yield many spurious matches. Indeed, similarity of tokenised representations alone is insufficient to demonstrate plagiarism, unless a program is complex or of an unusual structure.

4.1 Incremental Comparison

We adopted the following approach, which we call *incremental comparison*. A pair of programs is compared three times,

- in their original form,
- with the whitespace and all comments removed, and
- translated to a file of *tokens*.

A *token* is a value, such as *name*, *operator*, *begin*, *loop-statement*, which is appropriate to the language in use. The tokens necessary to detect plagiarism may not be the same as those used in the parser for a real implementation of the language — we do not need to parse a program as accurately as a compiler. Our scheme will work even with a very simple choice of tokens, and a rudimentary parser. Thus it is easy to update for a new language. Each line in the file of tokens will usually correspond to a single statement in the original program.

If a pair contains similarities, then it is likely that one or more of these comparisons will indicate as much. By examining the similarities and the corresponding sections of code in the original program, it should be possible to arrive at a preliminary decision as to whether the similarities are accidental, or are worthy of further investigation.

4.2 Implementation

We have implemented this scheme in a program, called SHERLOCK, which allows a lecturer to examine a collection of submitted programs for similarities. It assumes that each program is stored as a single file, and is written using a specific predefined language. Each pair of programs in the collection is compared three times, as described above.

4.2.1 Runs and Anomalies

A *run* is a sequence of lines common to two files, where the sequence might not be quite contiguous. That is, there may be a (possibly small) number of extra or deleted lines interrupting the sequence. The allowable size of interruptions (which we call *anomalies*), and density within the sequence, are configurable. For

instance, (using a default configuration) in Table 1, Sequence 1 and Sequence 2 form a run with two anomalies comprising one extra and one deleted line. By contrast, Sequence 1 and Sequence 3 do not form a run since there are six anomalies in nine lines.

When comparing two programs, SHERLOCK traverses the two programs looking for runs of maximum length. An entry is appended to a *record file* for each run, indicating which two programs are being compared, where the runs are located in the files, the number of anomalies in each run, and the size of the run as a percentage of the length of each program.

4.2.2 Presentation of Data

When all pairs of programs have been compared, a neural network program (a Kohonen *self-organizing feature map* [4]) is invoked which reads the record file and creates an image which illustrates the similarities between the programs listed in the record file. The main purpose of the neural network program is to arrange the components within the image so that it is clear and uncluttered (insofar at that is possible with a given set of data).

This image is a graph, whose nodes represent the files being compared, and whose arcs indicate that significant similarities have been found between the files whose nodes are their end-points. The shorter the line, the stronger the similarities. The function of the neural network is to design the layout for the image, a procedure which would be difficult by other means.

In Figure 1, which is typical of the sort of output produced by the neural network, the named files are grouped into 3 clusters. Files in separate clusters have essentially no similarities; those in the A–C cluster and the E–I cluster have similarities, but these are relatively weak. Cluster J–K is very tight, and large parts of files J and K are almost identical.

The image may be viewed or printed. The lecturer is then presented with a copy of the record file, and invited to select an entry from the file. Typically, an entry representing a long run for two programs close together in the image would be selected initially. The line sequences forming the run would then be displayed in separate windows, so they can easily be compared.

By repeatedly selecting entries from the record file, the lecturer is quickly be able to arrive at a preliminary judgement as to which programs are worth a detailed examination.

Sequence 1	*Sequence 2*	*Sequence 3*
`begin`	`begin`	`begin`
`line 2`	`line 2`	`extra line`
`line 3`	`extra line`	`line 3`
`line 4`	`line 3`	`extra line`
`line 5`	`line 4`	`line 4`
`line 6`	`line 5`	`extra line`
`line 7`	`line 7`	`another line`
`line 8`	`line 8`	`line 7`
`end`	`end`	`end`

Table 1: Illustration of Runs

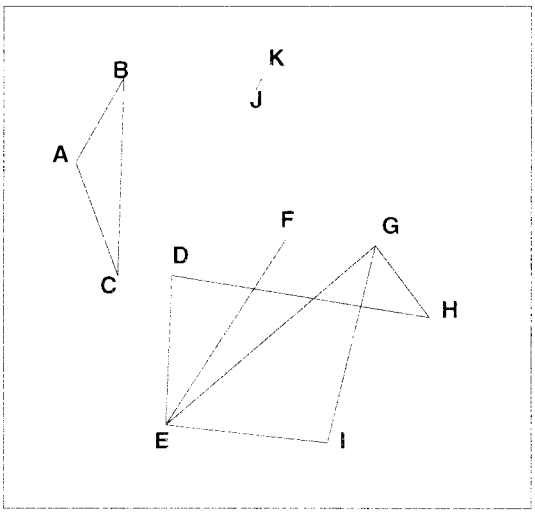

Figure 1: Neural Net Output

4.3 Testing

Our software has been used for several courses:

- an introductory programming course (in Pascal) for Computing students;
- an introductory programming course (in Pascal) run as a service course for the Mathematics Departments,
- a data structures course attended by computing students (again, using Pascal as the programming language);
- a course in functional programming (in Miranda) for first year Computing students;
- a course on UNIX shell and utilities, and
- a second year software engineering course using C++.

Each course is attended by over 100 students. We thus had a useful environment in which we could implement and test software which might assist us in detecting unauthorised collusion. Adapting the software to handle a different language has been done comfortably in a single afternoon. A number of instances of copied work have been detected in all of these courses.

It is not possible to demonstrate exactly what proportion of plagiarised programs such software will detect, for the reasons outlined at the start of this paper. However, we are confident that SHERLOCK has enabled that proportion to be high, and to demonstrate this we performed two tests.

4.3.1 First Test: Attempted Deception

We selected a program of medium length and good quality (2:1 standard) submitted for a later assignment in the Pascal course for Computing students. This we felt was a typical program which might lend itself to being copied. We then passed this to two postgraduate students who are skilled in Pascal, and requested them to attempt to edit the program with the intention of fooling SHERLOCK. Neither was able to do so without making substantial changes requiring a good understanding of Pascal and of the solution – a student with such knowledge would be unlikely to be motivated to plagiarise.

4.3.2 Second Test: Comparison with Plague

The software was tested on a suite of 154 programs written in Modula-2 [9], and on which Plague had been run. Of 22 instances of plagiarism initially detected by SHERLOCK, Plague found 21, and detected 2 others missed by SHERLOCK. "Fine-tuning" the parameters to SHERLOCK improved its performance and it then detected all 24 cases. We claim that SHERLOCK is capable of achieving a similar level of performance as Whale's Plague.

4.4 Some Statistics

Another measure of the effectiveness of our approach is to ask the question: "has it decreased the volume of plagiarism?" Of course, this cannot be unequivocally answered, but the number of detected instances has decreased substantially since we began to use the software, and our students became aware of it. In Table 2, we present the numbers of students suspected of plagiarism and detected by our software for whom the suspicion was well-founded. This is tabulated against the total number of students submitting assignments in which the software was used, and the ratio as a perentage.

There were, of course, "false hits", but the numbers of them are similar to the numbers of students actually detected copying work, and would be filtered manually.

Year	*Suspects*	*Total*	*%*
1996/7	2	275	0.73
1995/6	15	484	3.10
1994/5	34	564	6.03

Table 2: Statistics

It is clear that the volume of *detected* plagiarism has decreased substantially. This is due either to a reduced level of plagiarism, or to a greater proportion of students being able to hide the changes they have made. The latter, as we have already remarked, is a difficult exercise, and we therefore claim that the incidence of plagiarism has decreased.

5 Conclusions

We have designed a simple method which assists us with the detection of instances of plagiarism in computer programs. Our scheme is easy to adapt for the large variety of programming languages in use, and is sufficiently robust to be highly effective in an educational environment. Whilst having a detection rate as good as other more complex software, it presents its report as a simple graph, enabling large numbers of programs to be checked quickly and effficiently. By using "runs", SHERLOCK provides straightforward documentation which can be used as clear and convincing evidence should a suspected instance of plagiarism be disputed.

6 Acknowledgements

The authors wish to thank Geoff Whale for providing the test data and William Smith for the initial software development.

References

[1] Faidhi, J.A.W and Robinson, S.K., "An Empirical Approach for Detecting Program Similarity within a University Programming Environment", *Computer Education* **11** pp. 11-19 (1987).

[2] Grier, S., "A Tool that detects Plagiarism in Pascal Programs", in *12th SIGCSE Technical Symposium*, St. Louis, Missouri, pp. 15–20 (1981).

[3] Harrison, W.A. and Magel, K.I., "A Complexity Measure Based on Nesting Level", *ACM SIGPLAN Notices* **16**(3), pp. 63-74 (1981).

[4] Kohonen, T., *Self-Organization in Associative Memory*, Springer-Verlag, Berlin (1988).

[5] Magel, K., "Regular Expressions in a Program Complexity Metric", *ACM SIGPLAN Notices* **16**(7), pp. 61-65 (1981).

[6] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering* **SE-2**(4), pp. 308-320 (1976).

[7] Rambally, G.K. and Le Sage, Mauricio, "An Inductive Inference Approach to Plagiarism Detection in Computer programs", *Proceedings of the National Educational Computing Conference, Nashville, TN*, ISTE, Eugene, OR, pp. 23-29 (1990).

[8] Robinson, S.S. and Soffa, M.L., "An Instructional Aid for Student Programs", *ACM SIGCSE Bulletin* **12**(1), pp. 118-129 (1980).

[9] Smith, W.O., "A Suspicious Program Checker", BSc Dissertation, Department of Computer Science, University of Warwick (1994).

[10] Verco, Kristina L. and Wise, Michael J., "Plagiarism à la Mode: A Comparison of Automated Systems for Detecting Suspected Plagiarism", *The Computer Journal* **39**(9), pp. 741-750 (1997).

[11] Whale, G., "Identification of Program Similarity in Large Populations", *Computer Journal* **33**(2), pp. 140-146 (1990).

[12] Whale, G., "Software Metrics and Plagiarism Detection", *Journal of Systems and Software* **13**, pp. 131-138 (1990).

[13] Wise, M.J., "Detection of Similarities in Student Programs: YAP'ing may be preferable to Plague'ing", *ACM SIGCSE Bulletin* **24**(1), pp.268-271 (1992).