Plagiarism in Programming Assignments

Mike Joy and Michael Luck

***Abstract*—Assessment of programming courses is usually carried out by means of programming assignments. Since it is simple to copy and edit computer programs, however, there will always be a temptation among some students following such courses to copy and modify the work of others. As the number of students in these courses is often high, it can be very difficult to detect this plagiarism. We have developed a package which will allow programming assignments to be submitted on-line, and which includes software to assist in detecting possible instances of plagiarism. In this paper, we discuss the concerns that motivated this work, we describe the developed software, tailoring the software to different requirements, and finally we consider its implications for large group teaching.**

***Index Terms*—Plagiarism, programming.**

I. Introduction

THE numbers of students following computer programming courses in the U.K.—either within a computing degree or as part of another course—are increasing [1], [2]. At the same time, university staff are under considerable pressure to deliver such courses using available resources with maximum efficiency. 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 blatant copying of coursework.

Assessment of programming courses usually involves students writing programs, 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. Unless students have been organized into "teams" to produce group coursework, this is discouraged, and is likely to be regarded as a serious disciplinary offense.

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 an instructor 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. 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. At Warwick, 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 a course leader to detect 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.

Manuscript received June 30, 1995; revised January 21, 1999.

The authors are with the Department of Computer Science, University of Warwick, Coventry, CV4 7AL, U.K.

Publisher Item Identifier S 0018-9359(99)03953-9.

II. What is Plagiarism?

Plagiarism—unacknowledged 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 unauthorized 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 colleage, in the belief that it is acceptable.
- A weak student copies, and then edits, a colleage's program, with or without the colleage'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 minimizing 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 may be poorly defined, and some students may be familiar with different customs and norms. It is, nevertheless, still necessary to discover situations where there is a high probability that collaboration has taken place, so that any misunderstandings are resolved.

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 important—the tutor has been alerted that remedial intervention may be appropriate 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.

It must be realized 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 instructor 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.

A. 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, some common transformation strategies can be identified.

1) Lexical Changes: By *lexical changes*, we mean changes which could, in principle, be performed by a text editor. They do not require knowledge of the language sufficient to parse a program.

- 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) 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.

- Loops can be replaced (e.g. a `while` loop in Pascal can be substituted for an `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`).

B. 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 plagiarize 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 to their actions.

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.

III. 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]. There are two principal comparison techniques.

- Calculate and compare *attribute counts* [8]. This involves assigning to each program a single number representing capturing a simple quantitative 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 [7] and scope number [3] have been examined.
- Compare programs according to their *structure* [9], [6]. This is a potentially more complex procedure than calculating and comparing attribute counts, and depends fundamentally on the language in which the programs are written.

Whale [11], [12] has carried out a detailed comparison of various attribute count and structure comparison algorithms. He concludes that attribute count methods alone provide a poor detection mechanism, outperformed by structure comparison. His structure comparison software, *Plague*, is claimed to yield excellent results.

Both Plague [11] and Yap [13] (a system based on Plague, but simplified and optimized for speed) report a high measure of success. However, a recurring feature of structure comparison software is its complexity, and a detailed understanding is required of the target language. It is thus difficult to convert the software for new languages. For example, Wise reports 2.5 days to adapt Yap to handle the language Pascal rather than C [13], for which Yap was originally written.

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. 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.

IV. The Warwick Approach: Sherlock

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 instructor using the package must have an efficient 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* 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, unless the person copying the program already understands well how it works, only limited editing can be performed. The *lexical* changes 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 tokenized representations of their programs yield many spurious matches. Similarity of tokenized representations alone is insufficient to demonstrate plagiarism, unless a program is complex or of an unusual structure.

A. Incremental Comparison

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

- in their original form;
- with the maximum amount of whitespace removed;
- with all comments removed;
- with all comments *and* maximum amount of whitespace removed;
- 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.

TABLE I
Illustration of Runs

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`

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 not.

B. Implementation

We have implemented this scheme in a program, called SHERLOCK, which allows an instructor 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 five times, as described above.

1) Runs and Anomalies: We define a *run* to be 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 I, 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 (three deletions and three insertions).

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, which indicates which two programs were 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.

2) Presentation of Data: When all pairs of files have been compared, a neural-network program (a Kohonen *self-organizing feature map* [5]) is invoked which reads the record file and creates a PostScript image which illustrates the similarities between the programs listed in the record file.

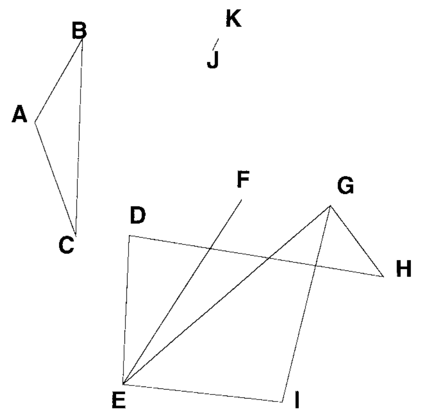

Fig. 1. Neural-net output.

The image consists of a number of points (representing the original files), connected by lines. A line joining two points indicates that significant similarities have been found between the corresponding files, and 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 Fig. 1, which is typical of the sort of output produced by the neural network, the named files are grouped into three 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 instructor 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, an instructor will quickly be able to arrive at a preliminary judgement as to which programs are worth a detailed examination.

C. Testing

At Warwick, we have implemented a software package called BOSS [4] ("The BOSS Online Submission System") which allows students to submit online programming assignments. It contains a collection of programs which run under the UNIX operating system, and is designed specifically for courses which have a large number of students attending, and which are assessed by means of programming exercises. Introductory programming courses in high-level computer languages are the typical target for BOSS.

BOSS is in use for four courses at present:

- 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 course on UNIX shell and utilities;
- a course which covers LEX and YACC

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 unauthorized collusion. Furthermore, SHERLOCK has also been used on a functional programming course (using the language Miranda) and a unit covering data structures (using Pascal). Adapting the software to handle a different language can be 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 plagiarized programs such software will detect, for the reasons outlined at the start of this paper. However, we have used the software on many courses, and have successfully managed to reduce the incidence of plagiarism through identification of particular cases. Specifically, over a period of three years, the proportion of students involved in such incidents has decreased from a relatively high 6% to a more acceptable 1%. We believe this is due to an awareness among students of the existence and efficacy of the plagiarism detection software. To demonstrate that our confidence is well-founded we performed two tests to gain a quantitative measure of performance.

1) First Test: Attempted Deception: We selected a program of medium length and good quality 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 plagiarize.

2) Second Test: Comparison with Plague: The software was tested on a suite of 154 programs written in Modula-2 [10] on which Plague had been run. Of 22 instances of plagiarism initially detected by SHERLOCK, Plague found 21, and detected two 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.

V. Conclusion

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. While 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 efficiently. By using "runs," SHERLOCK provides straightforward documentation which can be used as clear and convincing evidence should a suspected instance of plagiarism be disputed.

Acknowledgment

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

References

[1] Higher Education Statistics Agency, *Higher Education Statist. U.K. 1994/95*. Cheltenham, U.K.: HESA, 1995.

[2] Higher Education Statistics Agency, *Higher Education Statist. U.K. 1995/96*. Cheltenham, U.K.: HESA, 1996.

[3] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," *ACM SIGPLAN Notices*, vol. 16, no. 3, pp. 63–74, 1981.

[4] M. S. Joy and M. Luck, "Software standards in undergraduate computing courses," *J. Comput.-Assisted Learning*, vol. 12, no. 2, pp. 103–113, 1996.

[5] T. Kohonen, *Self-Organization in Associative Memory*. Berlin, Germany: Springer-Verlag, 1988.

[6] K. Magel, "Regular expressions in a program complexity metric," *ACM SIGPLAN Notices*, vol. 16, no. 7, pp. 61–65, 1981.

[7] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, 1976.

[8] G. K. Rambally and M. Le Sage, "An inductive inference approach to plagiarism detection in computer programs," in *Proc. Nat. Educational Comput. Conf.*, Nashville, TN. Eugene, OR: ISTE, 1990, pp. 23–29.

[9] S. S. Robinson and M. L. Soffa, "An instructional aid for student programs," *ACM SIGCSE Bull.*, vol. 12, no. 1, pp. 118–129, 1980.

[10] W. O. Smith, "A suspicious program checker," B.Sc. dissertation, Dept. Comput. Sci., Univ. Warwick, U.K., 1994.

[11] G. Whale, "Identification of program similarity in large populations," *Comput. J.*, vol. 33, no. 2, pp. 140–146, 1990.

[12] ——, "Software metrics and plagiarism detection," *J. Syst. Software*, vol. 13, pp. 131–138, 1990.

[13] M. J. Wise, "Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing," *ACM SIGCSE Bull.*, vol. 24, no. 1, pp. 268–271, 1992.

Mike Joy received the B.A. degree in mathematics and the M.A. degree from Cambridge University, and the Ph.D. degree in computer science from the University of East Anglia.

He is currently a Lecturer in Computer Science at the University of Warwick. His research interests include computer-based learning and teaching, agent-based systems, functional programming, and software engineering.

Dr. Joy is a member of the British Computer Society and of the Association for Computing Machinery, and is a Chartered Engineer.

Michael Luck received the degree of B.Sc. in computer science from University College London in 1985. After spending a year at the University of Illinois at Urbana-Champaign, he returned to University College London and received the Ph.D. in computer science for work on motivated discovery in 1993.

Since then, he has been a Lecturer in Computer Science at the University of Warwick. His research includes artificial intelligence, concerned with single-agent and multiagent systems, computer-assisted learning, tutoring systems, and educational technology.