*6th Annual CCSC Northeastern Conference, Middlebury, Vermont, April 20-21, 2001.*

# METRICS BASED PLAGIARISM MONITORING

*Edward L. Jones*
*Department of Computer and Information Sciences*
*Florida A&M University*
*Tallahassee, FL 32307*
*850-412-7362*
*ejones@cis.famu.edu*

ABSTRACT

Plagiarism in programming courses is a pervasive and frustrating problem that undermines the educational process. Defining plagiarism is difficult because of the fuzzy boundary between allowable peer-peer collaboration and plagiarism. Pursuing suspected plagiarism has attendant emotional and legal risks to the student and teacher, with the teacher bearing the burden of proof. In this paper we present a metrics-based system for monitoring similarities between programs and for gathering the "preponderance" of evidence needed to pursue suspected plagiarism. Anonymous results from monitoring are posted to create a climate in which the issue of plagiarism is discussed openly.

## 1 INTRODUCTION

Plagiarism is a common problem in programming courses. Although course syllabi warn students that plagiarism is punishable, the effort the instructor must invest to build and pursue a case of plagiarism effectively guarantees prosecution will seldom occur. It is important that educators address the issue of plagiarism, despite the additional time and emotional burden required to confront offenders [1]. Educators are obligated not only to talk about issues of ethics and social responsibility [2], but also to view ethical conduct with the same weight as technical competence. Certainly, plagiarism will be a constant temptation for computer science students. What ethical dilemma besides plagiarism is better suited to satisfy the curriculum mandate to treat ethical problems? Plagiarism is a concrete ethical problem that will recur throughout the student's academic career.

This paper illustrates how simple metrics can provide initial levels of screening for excessive collaboration and plagiarism. The application of metrics is a part of an overall process of accumulating the evidence necessary to confront violators. Our approach is extensible, permitting the use of additional measures and documentation. The system provides a sanitized report of the results from the monitoring, which is distributed to students to sensitize them to fact that “big brother” is watching.

In section 2, we discuss other approaches to plagiarism prevention, detection, and confrontation. In section 3, we describe the metrics used by our approach, and the environment in which evidence is collected on suspicious programs. Section 4 contains sample results from using our approach. In section 5 we illustrate how to extend this approach by adding other measures. The final section presents conclusions and suggestions for future work.

## 2 BACKGROUND AND RELATED WORK

The temptation to plagiarize in the academic setting is not limited to students. One alleged case of academic plagiarism by a faculty member illustrates the sticky issues of burden of proof, libel, due process, and professional disciplining [3]. The effort to pursue plagiarism has an attendant risk of counter-suit, with few guarantees of resolution. Nonetheless, we have a professional responsibility to get our students to think critically about ethical issues [2], and to confront plagiarism when it does occur [1].

Plagiarism detection is a pattern analysis problem. A plagiarized program is either an exact copy of the original, or a variant obtained by applying various textual transformations such as those shown in Table 1. A plagiarism detection method must produce a measure that quantifies how close two programs are. Except for the case of a verbatim copy, detection approaches that use direct comparison of text files are weak, since there is no obvious closeness measure. Most approaches take a lexical approach, where the program tokens are classified as language keywords and user symbols [4,5,6]. The sim plagiarism detection system [4] converts the source programs into token strings, then compares the strings using dynamic programming string alignment techniques like those used in DNA string matching. This powerful technique is capable of detecting the program modifications shown in Table 1.

**Table 1.** Plagiarism Transformations

1. Verbatim copying.
2. Changing comments.
3. Changing white space and formatting.
4. Renaming identifiers.
5. Reordering code blocks.
6. Reordering statements within code blocks.
7. Changing the order of operands/operators in expressions.
8. Changing data types.
9. Adding redundant statements or variables.
10. Replacing control structures with equivalent structures.

The YAP family of approaches [5,6] also tokenizes the source program, but retains only the tokens that indicate the program structure -- control structures, blocks, subprograms, and uses of library functions. A language-specific lexicon defines the significant language elements, including library functions, to be tokenized. The bodies of subprograms are expanded once, in the order in which they are called. Each subsequent call to an expanded subprogram is replaced by a unique token. The resultant token string is the program profile that is used in comparisons. The closeness of two programs is computed using a longest common subsequence algorithm applied to the pair of program profiles. The YAP approach detects most of the plagiarism transformations listed in Table 1.

# 3 OUR APPROACH

Our approach creates program-size independent, numeric program profiles. Closeness is computed as the normalized Euclidean distance between profiles. We investigate the three profiles shown in Table 2: one focuses on the physical attributes of the source file, another on language usage, and the third combines the other two. These profiles are selected for economy of construction and robustness of detecting the plagiarism transformations given in Table 1.

## 3.1 Programming Environment

We use a Unix-based program submission and grading environment. Each student is provided a private repository for submitting assignments and receiving feedback from the instructor. The student environment contains a set of Unix commands for accessing the repository. The student is permitted unlimited submissions, subject to deadline controls. The environment maintains a time-stamped log of all submissions. The submission environment facilitates the capture of artifacts used to document potential violations, and permits the accumulation of a historical archive of student work that provides insight into historical patterns of student behavior. This archive also serves as the test bed of students programs used in this work.

## 3.2 Profiles

The *physical profile* P describes a program in terms of its physical attributes: number of lines(*l*), words(*w*), and characters (*c*). The physical profile is obtained from the UNIX word count (wc) command. The *Halstead profile* H consists of Halstead metrics length (*N*), vocabulary (*n*) and volume (*V*). Profile H characterizes the program in terms of program token types and frequencies. Comments are discarded. Operator tokens correspond to source language keywords, operator symbols, and standard library module names. Operands are programmer-defined words.

**Table 2.** Physical and Halstead Profiles

| | |
|---|---|
| Physical | l = line count<br>w = word count<br>c = character count |
| Halstead | N = number of token occurrences (Halstead length)<br>n = number of unique tokens (Halstead vocabulary)<br>V = N $\log_2$ n ( Halstead volume) |
| Composite | Physical + Halstead |

A third profile is obtained by combining the physical and Halstead profiles. The *composite* profile is the 6-vector *(l,w,c,N,n,V)*. Composition combines the similarity detection strengths of two or more profiles. Table 3 shows the types of transformations (from Table 1) against which the physical, Halstead and composite profiles are robust. For the transformation where a check mark appears, the profile will give the same value. In other instances, the value may differ only slightly. Note that in the distance computation, small differences in one component are generally weighted downward

by the other components. Superficial plagiarism transformations usually have only small effects on the computed distance

**Table 3.** Plagiarism Detection Robustness of Various Profiles

| Transformation | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| profile P | √ | | | | | | | | | |
| profile H | √ | √ | √ | √ | √ | √ | √ | √ | | |
| Composite profile P+ H | √ | √ | √ | √ | √ | √ | √ | √ | | |

### 3.3 Closeness Computation

Closeness of two programs is the Euclidean distance between their profiles. The closeness of identical programs is zero. In order to establish a common yardstick by which to compare closeness among programs, each profile is normalized before computing the Euclidean distance. Normalization results in a uniform scale for comparison, the range zero to the square root of two [0,1.414]. More sophisticated schemes for normalizing profiles and weighting profile components exist, but were not investigated. Program pairs are ranked by closeness values. The range and distribution of closeness values for programs vary from assignment to assignment. There is no a priori closeness threshold that indicates plagiarism. Instead, the location of the threshold is determined from the data at hand, using statistical percentiles.

The tools to construct and analyze these profiles are implemented as Unix shell scripts and C++ programs. The Halstead profiling is the most complicated, since the source program must be tokenized, and each token classified as an operator or operand. A lexicon defining the operators in the source language must be constructed. The Halstead profiler is implemented as a C++ program.

### 3.4 Countermeasures Against Detection

Each profile has its blind spots. Students who understand the measures embodied in the profiles can make benign changes to their source code to mislead the plagiarism monitor. One can mislead the physical profile by adding extra lines of comments, by splitting or combining physical lines, and by renaming variables. The Halstead profile, which is based on vocabulary usage, is immune to these countermeasures. It is susceptible, however, to the addition of extra code that is never called. It is important to know the blind spots of a profile, and to overcome these by using a composite profile that makes it harder for students to mask plagiarism.

The distribution of closeness values depends upon the programming language and the nature of the assignment. A verbose and structured programming language like COBOL leads to programs that look very much alike. The closeness values for such programs tend to cluster at the lower end of the spectrum. A similar clustering is also likely when the assignment requires the reuse of modules provided by the instructor. The percentile approach is the preferred way to identify program pairs whose closeness appears exceptional.

## 4 SAMPLE RESULTS AND OBSERVATIONS

In this section, we show results from a programming assignment in the COBOL language. The data gathered for the class are very reflective of the behavior of the students taking the class. Collaboration was widespread: students tended to work

together in-groups of two or more. The pattern of collaboration (permitted and excessive) reflected in the closeness data reflects the observations of the instructor and other students. Based on these results and other experiences not reported here, we make some observations about how to interpret closeness results, and how to build that "better mousetrap."

Table 4 shows the pair-wise closeness measures that fall below the 5th percentile for the three profiles. Student logins have been replaced by fictitious names. The first observation is that, as expected, the closeness rank order of a program pair is different for each profile, since each profile measures different characteristics. Each profile may suggest a different set of "suspects."

**Table 4.** Closeness Results -- 5th Percentile

| Physical Profile | Halstead Profile | Composite Profile |
|---|---|---|
| 0.00000000 alpha alpha | 0.00000000 alpha alpha | 0.00000000 alpha alpha |
| 0.00000652 alpha beta | 0.00000000 alpha beta | 0.00002785 alpha beta |
| 0.00026963 beta gamma | 0.00000000 omega delta | 0.00135498 sigma delta |
| 0.00026981 alpha gamma | 0.00002150 alpha stallion | 0.00181437 omega delta |
| 0.00031262 gamma epsilon | 0.00002150 beta stallion | 0.00306925 omega sigma |
| 0.00048815 sigma delta | 0.00004025 omega sigma | 0.00513103 gamma badger |
| 0.00049825 alpha epsilon | 0.00004025 sigma delta | 0.00644130 octopus owl |
| 0.00050169 beta epsilon | 0.00056763 kappa rabbit | 0.00647050 alpha theta |
| 0.00066481 gamma theta | 0.00113476 omega rabbit | 0.00647057 beta theta |
| 0.00073158 beta theta | 0.00113476 delta rabbit | 0.00758068 delta owl |

A second observation is that the Halstead profile is more of an essential measure than the physical profile. The physical profile accommodates a *weight test*, focusing on size attributes of the source code. The Halstead profile accommodates a *vocabulary test*, focusing on the usage of significant words. Non-content text such as comments and white space, is ignored. Because the essence of plagiarism is program content, Halstead closeness should be used as a primary indicator, and physical closeness as a secondary indicator. When essential content is close and the incidental content is also close, the likelihood is higher that the programs are the same.

A third observation is that the composite profile does not combine the effects of the separate profiles in a predictable way. Of the ten smallest composite-profile closeness pairs, only three were in among the ten smallest pairs for the physical or Halstead profiles. Of the ten largest composite-closeness pairs (see Table 5), only one was among the ten largest physical and Halstead closeness pairs. However, five of the ten largest physical-closeness pairs were among the largest ten composite-closeness pairs. This phenomenon is explained by the way each measure in weighted in the closeness calculation. The profile component with the greatest magnitude difference has the highest weight. This favors the physical profile, since the values of its components (e.g., number of characters and words) are higher than values of the Halstead profile components.

The strongest case for plagiarism is when closeness is 0.000 using multiple profiles. However, even when a clear case of plagiarism presents itself, it is not obvious who the perpetrator is, and whether the alleged plagiarism is the result of theft or collusion. Clearly, this is the point at which intervention is required.

**Table 5.** Closeness Results - Upper Bounds

| Physical Profile | Halstead Profile | Composite Profile |
|---|---|---|
| 0.02753655 omega queen | 0.05031283 omega dove | 0.14348834 omega fox |
| 0.02777388 badger fox | 0.05031283 delta dove | 0.14387480 owl fox |
| 0.02813216 octopus dove | 0.05144719 dove rabbit | 0.14745269 queen epsilon |
| 0.02867729 owl fox | 0.05163772 dove kappa | 0.14801536 octopus fox |
| 0.02917513 queen elephant | 0.05325362 dove badger | 0.15247692 queen dove |
| 0.02985955 queen rabbit | 0.05432432 queen dove | 0.15793745 queen stallion |
| 0.03211501 octopus fox | 0.05510299 stallion dove | 0.16328829 queen rabbit |
| 0.03572043 queen kappa | 0.05511161 alpha dove | 0.17267538 queen elephant |
| 0.04426516 queen dove | 0.05511161 beta dove | 0.17831873 queen kappa |
| 0.04842636 queen fox | 0.05544899 dove epsilon | 0.24876065 queen fox |

## 5 EXTENDING THE BASIC APPROACH

The basic approach is to construct physical, Halstead, and composite profiles for each source code file, to perform a pair-wise closeness analysis, then publish the closeness lists at some percentile. Prime suspects are those who appear near the top of the closeness lists. Being near the top of multiple closeness lists is tantamount to having corroborating evidence.

In this section, we extend the basic approach by considering two second-order products generated in the normal course of grading the program. The first product is the compilation error log. A program that compiles produces an empty log; otherwise, a log of error messages is produced. The other second-order product is the execution log produced when the program is run against a standard set of tests. The metrics-based approach to comparing source files is applied to these second-order products, as summarized in Table 6.

**Table 6.** Plagiarism Detection on Program Grading Artifacts

| | |
|---|---|
| Source Code | • Construct profiles for each submitted program.<br>• Compute source closeness measures.<br>• Identify *source suspects* |
| Compilation Log | • Compile each program, saving compilation log.<br>• Construct profiles for each failed compilation.<br>• Compute compilation closeness measures.<br>• Identify *compilation suspects*.<br>• Compare compilation closeness for source suspects.<br>• Refine/Revise compilation suspects list. |
| Execution Log | • Execute each program capturing execution log.<br>• Construct profiles for each execution log.<br>• Visually inspect execution logs for source suspects. |

Plagiarism detection for compilation logs focuses on anomalies -- typically plagiarism is error preserving. A physical profile is constructed for each log. A lexicon of compiler error keywords, such as `"type mismatch"` or `"duplicate definition"` must be constructed in order to construct the Halstead profile. Closeness lists are generated for the compilation profiles, and compilation suspects identified. Compilation suspicion strengthens the case made by the source suspicion.

An execution log is generated when each program is executed. The physical profile is constructed. Programs with identical behavior will have identical profiles. Because the vocabulary of program output changes from assignment to assignment, an assignment-specific lexicon must be created when the Halstead profile is used. The lexicon entries reflect program output requirements (e.g., output captions, and menu and message wording). Execution log closeness lists are generated for programs that have been identified as source suspects. The execution suspicion strengthens the case for the source suspicion.

The final step is to visually examine suspects with corroborating evidence. Based on the closeness lists (metrics) and the visual inspection, the teacher can make appropriate intervention, as described by Harris [1]. Having captured evidence throughout the course about collaboration patterns of closeness enables the teacher to evaluate the severity and to take action appropriate to the offense. Clearly, a repeat offender should be treated more severely than a first-time offender.

## 6 CONCLUSIONS AND FUTURE WORK

The metrics-based physical and Halstead profiles are simple to construct, and they lead to efficient computation of pair-wise closeness measures. The use of these metrics in the context of program submission and grading enable the capture of historical records of patterns of collaboration among students. When intervention is warranted, this body of evidence provides a factual basis for taking appropriate action.

To date, this approach has been used to observe patterns of collaboration, not to pursue violators. We will apply this technique to other programming courses to observe the effects of assignment type and programming language on closeness patterns. These studies will provide additional data needed to understand how to establish the closeness threshold below which teacher intervention is warranted. We will also pilot the practice of posting anonymous closeness results, to sensitize students to plagiarism and to stimulate classroom discussion about this and other ethical issues. We will investigate an alternative closeness computation in which profile components are weighted to reflect their relative importance. Additionally, we will investigate normalization formulae that prevent domination by the profile component with the largest magnitude (e.g., character count of the physical profile). Finally, we will formalize this approach into a portable system supporting multiple programming languages.

Our final observation, though troubling, is that a worse offense than plagiarism is having someone outside the class write the program the student submits. Presently, this situation can be discerned only by written or programming examinations. A possible solution to this dilemma is to require the student, at the time of submission, or later, in a controlled environment, to identify and remove program errors injected by an automated error-seeding program. This approach is currently being investigated in conjunction with work on automating the grading of student programs [7].

## REFERENCES

[1] Harris, J.K. Plagiarism in Computer Science Courses. *Ethics in the Computer Age*, Gatlinburg, TN USA, 1994, 133-134.

[2] Schulze, K.G., and Grodzinsky, F.S. Teaching Ethical Issues in Computer Science: What Worked and What Didn't. 27th SIGCSE Technical Symposium, Philadelphia, PA USA (1996), 98-101.

[3] Kock, N. A Case of Academic Plagiarism. *Communications of the ACM 42*,7 (July 1999), 96-104.

[4] Gitchell, D. and Tran, N. Sim: A Utility for Detecting Similarity in Computer Programs. *Proceedings 30th SIGCSE Technical Symposium*, New Orleans, LA, USA (March 1999), 266-270.

[5] Wise, M.J. Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. *Proceedings, 23rd SCGCSE Technical Symposium*, Kansas City, USA. (March 5-6, 1992), 268-271.

[6] Wise, M.J. YAP3: Improved Detection of Similarities in Computer Program and Other Texts. *Proceedings 27th SIGCSE Technical Symposium*, Philadelphia, PA, USA (February 15-17, 1996), 130-134.

[7] Jones, E.L. Grading Student Programs – A Software Testing Approach. *Journal of Computing in Small Colleges 16*, 2 (January 2001), 185-192.