USING METRICS TO DETECT PLAGIARISM

STUDENT PAPER

Shauna D. Stephens
Department of Computer and Information Sciences
Florida A&M University
Tallahassee, FL 32307
sstephen@cis.famu.edu

ABSTRACT

Plagiarism in programming courses calls for a need for professors to be able to correctly identify occurrences of plagiarism in order to safeguard the educational process. However, detecting plagiarism is a long and tedious ordeal because of the fine line between allowable peer-peer collaboration and plagiarism. In this paper we present a metrics-based tool which aids in the process of detecting plagiarizers by monitoring the similarities between programs.

1 INTRODUCTION

The common problem of plagiarism in programming courses continues to be a prevailing issue for course instructors. Plagiarism not only undermines the educational process, it places a heavy responsibility on professors to correctly identify occurrences of plagiarism. Detecting plagiarism is a long and tedious process often requiring extra time examining students' source code. Therefore, a tool is needed which aids the instructor in narrowing down the search for violators and decreasing the time spent analyzing source code. This paper illustrates how a simple metrics tool can aid in this process by providing initial levels of screening so that professors can concentrate their time on the most likely suspects.

A plagiarized program is either an exact copy of the original, or a variant obtained by applying various textual transformations. A plagiarism detection method must produce a measure that quantifies how close two programs are. Except for the case of a verbatim copy, detection methods 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 [1,2,3]. The SIM plagiarism detection system [1] converts the source programs into token strings, then compares the strings using dynamic programming string alignment techniques like those used in DNA string matching. The YAP family of approaches [2,3] also tokenizes the source program, but retains only the tokens that indicate the program structure – control structures, blocks, subprograms, and uses of library functions. The closeness of two programs is computed using a longest common subsequence algorithm applied to the pair of program profiles. In contrast, the approach used in our work uses simple metrics. Instead of complex algorithms, we create program-size independent, numeric program profiles. These profiles are selected for economy of construction and robustness in detecting common plagiarism transformations.

2 APPROACH

Plagiarism detection is performed in a UNIX-based program submission and grading environment. Each student is given 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 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.

The student programs for a specific assignment are stored in a specific directory unique to that assignment. Plagiarism detection is accomplished through a set of Unix shell scripts and C++ programs. Entry to the system is through the UNIX shell script plagiarism_detector, which performs a complete plagiarism check on all pairs of programs submitted for a given assignment. A sample session is shown in Figure 1, showing the steps in the process: build profiles, compute closeness, and select the closest program pairs based on percentiles. Each of these steps is described in more detail in the sections 3-5.

Figure 1. Plagiarism_Detector Session

```
============================================================
Plagiarism Detection Started Fri Jan 26 14:13:36 EST 2001

Building physical profile: Fri Jan 26 14:13:36 EST 2001
Calculating physical closeness: Fri Jan 26 14:13:37 EST 2001
Building halstead profile: Fri Jan 26 14:16:28 EST 2001
Calculating halstead closeness: Fri Jan 26 15:00:49 EST 2001
Merging physical/halstead profiles: Fri Jan 26 15:03:37 EST 2001
Calculating composite closeness: Fri Jan 26 15:03:49 EST 2001
Calculating halstead percentiles: Fri Jan 26 15:03:50 EST 2001
Calculating physical percentiles: Fri Jan 26 15:03:51 EST 2001
Calculating composite percentiles: Fri Jan 26 15:03:51 EST 2001

Number of Programs =        31
Total Lines of Code =     12548 total
Elapsed Time = 50 MINUTES

Plagiarism Detection Completed Fri Jan 26 15:03:53 EST 2001
============================================================
```

3 METRICS-BASED PROFILES

A profile is a quantitative representation of the source program that measures one or more aspects of the program, such as size or complexity. When more than one measure is used, the profile becomes an array (vector) of numbers, where each number may have a different meaning. Having a numeric description is beneficial because it allows the indirect comparison of two programs using mathematical techniques such as Euclidean distance, without having to compare the programs directly.

3.1 Physical Profile

The physical profile is a representation of a program's physical attributes: number of lines (*l*), words (*w*), and characters (*c).* Two programs are identical if these counts all agree. This profile aids in detecting changes in the cosmetics of a program such as deleting comments, removing blank lines and adding lines of code which don't do anything or never get executed. In a Unix system, issuing the word count (wc)

command can generate physical profiles; the results from the command appear as shown in Figure 2.

Figure 2. Sample Output For Physical Profile

```
348    1116   12561 mickey.cbl
454    1581   21897 goofy.cbl
547    1735   22077 donald.cbl
201     556    6974 pluto.cbl
206     550    7143 minnie.cbl
```

3.2 Halstead Profile

The Halstead profile describes a program based on program token types and the frequency of their occurrence. We count two types of tokens, operators and operands, ignoring comments. Operator tokens correspond to source language keywords, operator symbols and standard library names. Operand tokens are all other programmer-defined words. These counts are used to produce the Halstead length (N), vocabulary (n), and volume (V). Figure 3 shows an example of the Halstead profile produced by the plagiarism_detector.

Figure 3. Sample Output For Halstead Profile

```
259   995   5529.044 mickey.cbl
250  1145   6322.073 goofy.cbl
285  1366   7721.300 donald.cbl
161   489   2484.807 pluto.cbl
165   483   2466.172 minnie.cbl
```

Computing the Halstead profile requires breaking the source code into tokens, classifying the tokens, and counting the unique tokens in each category. A file of COBOL keywords is used to classify tokens. In the current implementation of plagiarism_detector, a shell script computes the Halstead profile. This calculation accounts for roughly 90% of the time required to check for plagiarism (see Figure 1).

3.3 Composite Profile

The third profile is obtained by combining the physical and Halstead profiles to obtain the composite profile. The composite profile combines the strengths of the physical and Halstead profiles. A composite profile is a simple way to combine different characterizations which leads to a stronger plagiarism detection tool. Figure 4 shows sample output for the composite profile.

Figure 4. Sample Output For Composite Profile

```
547 1735 22077 285 1366  7721.300 donald.cbl
201  556  6974 161  489  2484.807 pluto.cbl
206  550  7143 165  483  2466.172 minnie.cbl
770 2824 34910 323 2097 12115.737 cinderella.cbl
444 1271 18018 249  987  5445.726 tarzan.cbl
```

4 CALCULATING CLOSENESS

The next step after calculating all three profiles is calculating the closeness. The closeness of two programs is computed as the Euclidean distance between two profiles p1=(a1,b1,c1) and p2=(a2,b2,c2):

$$D = sqrt\ [\ (a1 - a2)^2 + (b1 - b2)^2 + (c1 - c2)^2]$$

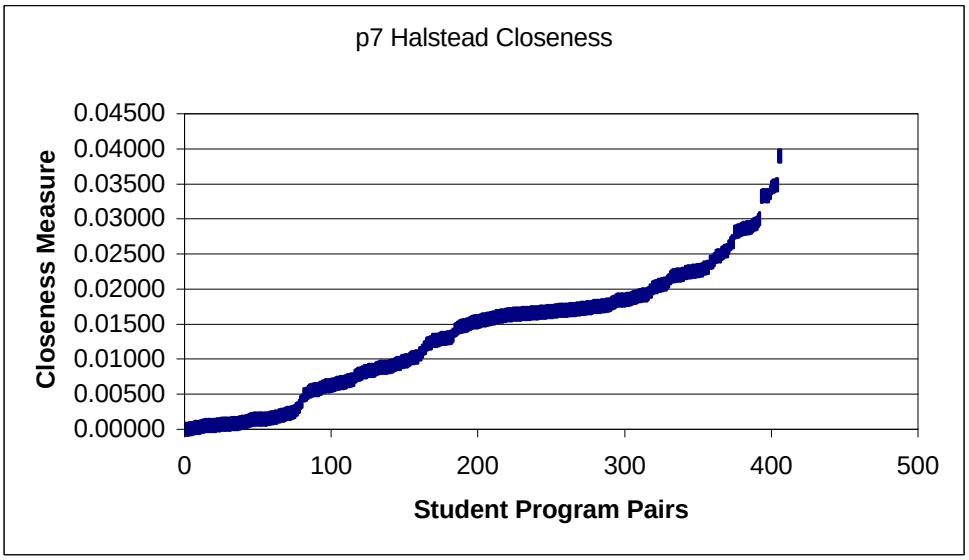
Profiles are normalized before calculating the distance. Normalization establishes a standard yardstick by which to compare the programs. Normalization provides the scale with the range 0.000 to 1.414 (2). Each assignment may produce a set of pair-wise distances with a distinctive signature, with respect to the range of closeness values and the clustering of values. The plagiarism_detector computes all the pair-wise closeness measures, and produces a list sorted by closeness value (as shown in Figure 5).

Figure 5. Pair-wise Closeness Values

```
0.00000000  cinderella.cbl  cinderella.cbl
0.00001929  mulan.cbl       beauty.cbl
0.00015572  beast.cbl       scrooge.cbl
0.00022046  happy.cbl       sneezy.cbl
0.00055286  bashful.cbl     grumpy.cbl
```

5 IDENTIFYING SUSPECTS USING PERCENTILES

The only time plagiarism is clear-cut is when the closeness value of two programs is 0.000, which means that the two programs are exact copies of one another, or one has been transformed into the other by a process that does not alter the token counts. Figure 5 shows an example (cinderella.cbl) of a case where programs are judged "identical" by the plagiarism detector. Except for that special case, it is not clear how to establish a closeness threshold below which two programs are *too close*.



The range and distribution of closeness values vary from assignment to assignment, so there is no predetermined threshold below which a closeness value is "unusually" close or suspicious. Programming languages which are highly structured such as COBOL lead to programs that look very similar. The closeness values for such programs tend to cluster at the lower end of the spectrum. A similar clustering is

also apparent when the assignment involves the reuse of modules provided by the instructor.

The approach used in this study is to sort program pairs in increasing closeness order, and to focus attention on a small percentage of the pairs. The location of the threshold is determined by the data at hand, using statistical percentiles. The lower the percentile, the greater the likelihood of plagiarism. With such a screening tool, the instructor can easily identify programs that may need to be looked at more closely by simply looking at the programs in the lowest percentile. Figure 6 shows the suspicious programs at the 5th percentile level, i.e., the closest 5% of all the program pairs.

Figure 6. Suspicious Programs (5th Percentile)

```
SUSPICIOUS PROGRAMS (composite.close - 5 percentile)

CLOSENESS          Program Name1       Program Name2
--------------     ----------------    -------------
1   0.00000000     cinderella.cbl      cinderella.cbl
2   0.00007933     donald.cbl          happy.cbl
3   0.00120359     beast.cbl           scrooge.cbl
4   0.00247064     pluto.cbl           sleepy.cbl
5   0.00363620     beast.cbl           grumpy.cbl
```

6 SUMMARY AND FUTURE WORK

There is no foolproof method in detecting plagiarism. An automated tool can only serve as an aid to identifying plagiarizers. A determined student can find ways around any system once they know what kind of method or tool is being used to detect questionable activities. The approach described in this paper can be expanded to use additional tactics to make plagiarism more and more difficult for students.

Future work will include the investigation of other statistical methods to determine a cutoff value for suspicious programs. On-going work is focused on shortening the running time of the *plagiarism_detector* tool. A C++ version of the shell script that computes the Halstead profiles is being written. Finally, efforts are underway to generalize this approach to support multiple programming languages.

ACKNOWLEDGEMENTS

The author, an undergraduate researcher funded under NSF grant EIA-9906590, wishes to thank her mentor, Dr. Edward L. Jones, for time spent on this project during the past semester.

REFERENCES

[1] 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.

[2] 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.

[3] 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.