Investigating Student Plagiarism Patterns and Correlations to Grades

Jonathan Pierce
University of Illinois
jmpierc2@illinois.edu

Craig Zilles
University of Illinois
zilles@illinois.edu

ABSTRACT

We analyzed 6 semesters of data from a large enrollment data structures course to identify instances of plagiarism in 4 assignments. We find that the majority of the identified plagiarism instances involve cross-semester cheating and are performed by students for whom the plagiarism is an isolated event (in the studied assignments). Second, we find that providing students an opportunity to work with a partner doesn't decrease the incidence of plagiarism. Third, while plagiarism on a given assignment is correlated with better than average scores on that assignment, plagiarism is negatively correlated with final grades in both the course that the plagiarism occurred and in a subsequent related course. Finally, we briefly describe the Algae open-source suite of plagiarism detectors and characterize the kinds of obfuscation that students apply to their plagiarized submissions and observe that no single algorithm appears to be sufficient to detect all of the cases.

1. INTRODUCTION

There has long been concern about widespread plagiarism[1] in undergraduate Computer Science courses in higher education [16]. Previous work has used surveys to determine that up to 90% of students admit to at least one type of academic integrity violation [16]. Here, we will choose to focus on code plagiarism. Code plagiarism occurs when students source their code from another party instead of writing it themselves, often modifying the code somewhat from the original source in order to attempt to evade detection. Previous work suggests that plagiarism rates for programming assignments in U.S. universities to be upwards of 10% [8].

Coding assignments are particularly ripe for plagiarism. Commonly, all students are given the same skeleton code and functional specification. With the advent of increasingly large enrollments, *auto-graded* assignments with a piece of software against a set of unit tests has become common. Even if course staff does look at assignments by hand, large enrollments mean that it is intractable to compare every pair of assignments by hand.

Furthermore, plagiarism is greatly facilitated by the reuse of assignments. Reusing programming assignments from semester to semester is a common practice in large courses, because doing so allows the assignments to be refined and bugs to be fixed. This reuse, however, means that students have many more sources from which to get a copy of the assignment, including students from previous semesters or students that upload their code to public code sharing web sites (*e.g.*, GitHub, Pastebin). Increased ease may make plagiarism, for some students, a significant temptation.

Beyond moral arguments, plagiarism is a problem for several reasons. First, many faculty respond to plagiarism by making homework a relatively small component of the overall course grade, which gives exams a disproportionately large weight compared to assignments [11]. Doing so makes the course grade very sensitive to a small number of samples of student performance and can fuel test anxiety [18]. Second, plagiarism can contribute to grade inflation, if students are illegitimately getting credit for assignments that they couldn't complete. Finally, if undetected, plagiarism can facilitate students progressing through courses without achieving the desired learning goals, which affects both their ability to complete subsequent courses and program objectives.

In this work, we seek to explore the behavior of students that plagiarize in a number of dimensions. While there has been significant previous work on plagiarism detection, we found that most of it focuses on the development of effective plagiarism detectors [4, 6, 7, 8, 9, 10, 15, 17]. In contrast, we found little prior work analyzing the behavior and outcomes of students who plagiarize. In this work, we make three main contributions:

1. We characterize the ways that students attempt to evade detection (by changing or obfuscating their submissions relative to its source).
2. We characterize the plagiarism instances in a number of dimensions, including whether the cheating is intra-semester or inter-semester, whether partners were allowed, and the occurrence of repeat offenses.
3. We analyze the correlations between a student's plagiarism and their grades in the course as well as their grades in a subsequent programming-intensive course.

[1]Plagiarism is defined as "the practice of taking someone else's work or ideas and passing them off as one's own" and is generally considered a violation of academic integrity at most U.S. universities.

SIGCSE '17, March 08-11, 2017, Seattle, WA, USA

DOI: http://dx.doi.org/10.1145/3017680.3017797

Section 2 describes our data collection, our development of an open-source plagiarism detector (Algae), and manual inspection/coding of the identified plagiarism instances. The resulting labeled-for-plagiarism corpus is analyzed both in isolation and in conjunction with grade information. Our general findings are described in Section 3. In Section 4, we describe the plagiarism detectors that make up the Algae suite that were used in this work and report of their relative effectiveness in detecting plagiarism.

2. EXPERIMENTAL METHOD

The research was performed at a large, public mid-western university, primarily in the context of sophomore-level data structures course. Features of this course include:

- It is a core course, early in the curriculum required for all CS and CE majors.
- Hundreds of students took the course in each semester that we investigated, providing a large data set.
- The assignments studied have remained essentially unchanged for the semesters investigated.
- All assignments are in the same language (C++), permitting the use of a single tool suite.
- Some course assignments must be done alone, while others can be done with a partner.

We analyzed six semesters (Fall 2011 through Spring 2014) of data from this course, including code repositories, per-assignment grades, and final letter grades. In addition, we were provided access to eight semesters (Fall 2011 through Spring 2015) of final letter grades for a systems-oriented programming course that is required for CS majors, for which the data structures course is an immediate prerequisite. This data encompasses 2,409 students in total.

All of the data was anonymized prior to delivery to us by a third party. This anonymization deterministically mapped all instances of each student's university-issued identifier to a specific random number. In this way, we could not know the identity of the student responsible for any piece of data, but we could still associate the elements of the different data sets with the same individual.

This paper presents our findings from four of the programming assignments in the data structures course. The other assignments in the course were either found to be too simplistic or had a solution space that was too narrow for us to reliably infer plagiarism with confidence. The four assignments studied include:

- `list` - An implementation of a list abstract data type using a doubly-linked list, along with associated other functions (such as a merge sort).
- `quadtree` - An implementation of a quadtree (where each child corresponds to a quadrant of an image) and associated functions (such as removing unnecessary nodes in order to compress an image).
- `kdtree` - An implementation of a k-dimensional binary search tree, used to efficiently find the nearest matching color in a set to a given color using nearest neighbor search.
- `maze` - Students generate, solve, and render random mazes and their solutions.

Instances of plagiarism were inferred[2] from the data set as follows:

1. An open-source plagiarism detector called Algae [13] was developed (Pierce) to identify *plagiarism candidate*s (*i.e.*, suspicious pairs of assignments). Algae consists of a suite of six different plagiarism detectors; in Section 4, we briefly describe the detectors and report on their relative effectiveness.
2. For each assignment, all six semesters of data was provided to Algae simultaneously; for each of the six detectors, Algae returned up to 750 plagiarism candidates, sorted by its perceived likelihood of plagiarism.
3. Each plagiarism candidate was inspected manually by a member of our research team (Pierce) and classified as *cheating* or *not cheating*. The human inspector would classify as cheating instances where they believed that there was at least a 75% chance that one assignment was derived from the other or if there was at least a 75% chance that an assignment had been created or processed by a computer program rather than than written by hand (*e.g.*, automatically obfuscated).

In the process of this manual inspection, we identified two optimizations that significantly improved the efficiency of classifying the candidates. First, we observed that plagiarism often occurred in clusters of many similar (or identical) submissions, where presumably many people had access to the same source (e.g., code that was uploaded to a public GitHub repository). In such cases, many of the candidates identified by Algae would be all of the combinatorial pairings of such a cluster. We avoided manually evaluating all of these pairings through *auto-implication*. After a submission s_1 was manually implicated for cheating, the system would auto-implicate any other submission s_2 where Algae ranked the candidate pair (s_1, s_2) among its top 750 candidate pairs. Note, auto-implication was not transitive.

Second, because clusters were inspected in sorted order of similarity from highest to lowest, we could stop inspecting when we stopped finding cases that we characterized as cheating. We found Algae's relative ranking of the candidates to be reliable, permitting us to mark all lower ranking candidates as not cheating after it was deemed unlikely to find additional cheating cases.

As a result of these assumptions, in this text we will use the terms "cheating" or "plagiarism" for brevity when in fact we mean "presumed cheating" and "presumed plagiarism".

3. RESULTS

Our analysis of the data structures assignments revealed the presence of plagiarism at rates consistent with those found in previous studies. While the cheating rate varied from semester to semester (by as much as a factor of two), there was no discernible pattern in the cheating rate over time. Notably, the rate of cheating does not appear to be increasing as the assignments continue to be reused.

[2]We did not have access to data with respect to plagiarism discovered and/or prosecuted during the offering of the course.

Table 1: Distribution of cheating repeat offenders.

Number of Assignments Student Plagiarized	Fraction of Cheaters
1	64.77%
2	19.57%
3	10.18%
4	5.48%

In this section, we focus our analysis on: (1) the degree to which students cheat on multiple assignments, (2) whether they plagiarize from students concurrently taking the course with them or from those in previous semesters, (3) what effect allowing students to work on assignments with a partner has on cheating rate, and (4) the relationship between the grades earned by cheaters and non cheaters.

Repeat Offenders: As shown in Table 1, over half (64.77%) of all students found to be plagiarizing did so only on one assignment, with decreasing percentages cheating on two or more assignments (as shown in Figure 1). Only 5.48% of cheaters cheated on all four assignments we looked at. This suggests that very few students are failing to do any assignments on their own.

Cross semester: We found that a majority (57%) of the clusters implicated for plagiarism contained students from multiple semesters, while the remaining 43% of clusters contained students from a single semester. While this data suggests that one way to curb plagiarism is to not reuse assignments between semesters, it could be that producing new assignments doesn't actually change the rate of plagiarism, but merely shifts it to students having to cheat from students taking the course concurrently. An interesting follow-up experiment would compare the rate of cheating on newly produced assignments with those that have been reused.

Partners: One of the assignments that we studied (kdtree) allowed the students to work with another currently enrolled student as a partner by documenting this partnership as part of the submission. If this partners listing is mutual, both partners are from the same semester, and only those two students have similar code, Algae ignores any matching between those two students for identifying plagiarism candidates. If, however, two students are partners and one of them is found to have cheated, the second student is also assumed to have cheated.

While many faculty might hope that by providing a legal opportunity to collaborate with another student might reduce the incidence of cheating, this is not born out by our results. The cheating rate on the kdtree assignment was the second highest and almost tied for the highest rate of the four assignments. Given such a small sample size, it is hard for us to draw any real conclusions about the effects of allowing or disallowing partners on plagiarism, but we find this first data point to be surprising and disappointing.

Assignment Grades: Students identified as cheating on a given assignment tended to do better (out of 100 points) than non-cheating students on that assignment (as shown in Table 2). However, the scores for students who plagiarized tended to be much further from perfect than one might expect. This suggests that many times when students cheat, they do not know if their source received a perfect score or not.

Table 2: Average labs scores (by cheating found).

Assignment	Non-cheating Ave.	Cheating Ave.
list	83.42	84.57
quadtree	72.51	78.21
kdtree	82.04	89.97
maze	77.61	80.09

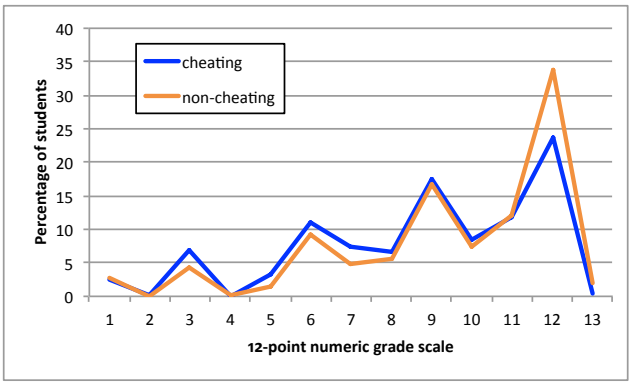

Figure 1: Final grade distribution of cheaters and non-cheaters in the data structures course. Cheaters tend to have fewer high grades and more low grades.

Course Grades: For overall course grades, we received the letter grade and not the percentage used to compute that letter grade. To permit us to compute average grades, we mapped the letter grades we received to a 12-point scale, where F, D-, D, D+, ..., A, A+ corresponded to 0, 1, 2, 3, ..., 11, 12 on the numeric scale. The distribution of final grades for cheaters and non-cheaters is shown in Figure 1.

In contrast to the assignment grades, cheaters performed worse in the class overall. The average grade for all students was 8.29/12. For the cheaters that had final grades (those that did not drop the course), the average grade was 7.73/12. For the remaining honest students, their average grade is 8.44/12. Plagiarizing students do 0.71 points worse on the 12-point scale. This corresponds to roughly the difference between an B- and C+. This difference while modest is significant. A standard two-tailed paired T-test of the grade distributions of cheating and honest students revealed them to be statistically significantly different with $p = 0.0186$.

Furthermore, students tend to do worse the more assignments that they cheat on (as shown in Table 3). Students who cheated on all four assignments that we studied had an average final grade of 6.39. At 1.59 points lower than the non-cheating average, this difference is over twice as large as the spread between cheaters and non-cheaters.

Of course, correlation is not causation. It could be either that plagiarism causes lower grades (because students have poor understanding of the material and do worse on exams as a result), or it could be the case that poorer students are more tempted to plagiarize in the first place.

Subsequent Course Grade: We also see cheaters having lower performance in subsequent programming courses. Figure 2 shows the grade distributions (again on the 12-point) scale, for the systems-oriented programming course, for each of the cheater and non-cheater populations that continue on to take that course. On average there is a grade depres-

Table 3: Average final grade (by cheating)

Number of Assignments Student Plagiarized	Average Final Grade (on 12-point scale)
0	8.44
1	7.98
2	7.15
3	6.98
4	6.39

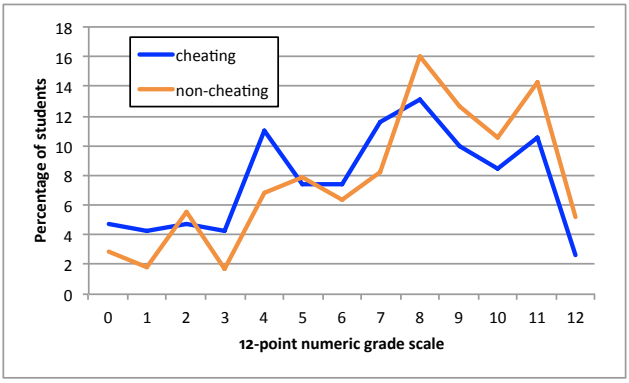

Figure 2: Final grade distribution in follow-on system-oriented programming course of cheaters and non-cheaters. Cheaters tend to have fewer high grades and more low grades.

sion of 0.89 points on a 0-12 scale. Again, while plagiarizers did worse, they did not do hugely worse. However, this data does seem to suggest that students who cheat tend to fare worse in future classes (whether they cheated in them or not). Another T-test over the cheating and non-cheater grade distributions for the systems course revealed them to be statistically significantly different with $p = 0.00019$.

4. ALGAE DETECTORS

The Algae[3] platform was developed to include a broad variety of plagiarism detectors; the diversity of these detectors allows us to characterize the degree of obfuscation that the cheaters use to make their plagiarized submissions appear different than the source text. In this section, we'll describe the salient features of each of the detectors and then present results about the portion of plagiarized submissions each detector captures.

4.1 Detector Descriptions

As we describe each of Algae's detectors it is important to note that most of the described detectors are either re-implementations of pre-existing detection algorithms from the research literature or greatly simplified detectors for catching specific instances of plagiarism. We are describing the detectors, so that the reader can understand the characterization of the types of obfuscation observed, which is the contribution of this section of the paper.

Near-Identical (Lazy) The Lazy detector attempts to find plagiarism where students have made little to no effort to differentiate their submitted code from their source. It takes each submission, converts all text to uppercase removes all comments, whitespace, and some extraneous punctuation — ( ) ; , { } _ ' " — and then hashes the result using SHA-256. This generates a single number for each file, and the algorithm considers any pairs of files with the same hash as plagiarism.

Identical Token Stream (IdentToken) The IdentToken detector uses the Clang C/C++ compiler [2] front end to convert a source file into a token stream, and some tokens are replaced with generic substitutes to make the detector resilient to simple transformations (*e.g.*, renaming variables). Specifically, literal tokens are converted to a string specifying the type (*e.g.*, NUM, STRING, CHAR, BOOL), identifiers are encoded by their type (*e.g.*, variable declarations → DEC, variable references → REF, variable uses → USE), and, like the Lazy detector, it removes comments, whitespace, and extraneous punctuation. Once transformed, the file is again hashed and identical hashes are considered plagiarism. This detector is akin to an early detector that required identical token stream [6].

Modified Token Edit Distance (MTED) The Modified Token Edit Distance pre-processes the token stream the same way as the IdentToken detector, with two additions:

- MTED only cares about code that exists within a function.
- MTED deterministically reorders functions by sorting them in ascending order of contained token count (tie-broken alphabetically by the function names). This should negate the threat of a student re-ordering functions in code in order to increase their edit distance from their source.

MTED then computes the edit distance between every pair of submissions using standard Levenshtein distance. In this way, the detector is resilient to some non-trivial code reorderings and other attempts at obfuscation. Algae stores the the edit distances, sorts them, and returns the pairs with the N smallest.

The MTED algorithm is similar to previous detectors that use edit distance to compare tokenized code [4, 8, 9]. SIM [9] differs from MTED in the effort that it uses to match functions, where MTED merely sorts them by size. Other work [4, 15] explicitly avoids pairwise edit distance due to its computational complexity (*e.g.*, filtering using an inverted index to find candidates for computing edit distance [4]), but we found that even on a corpus of 2,409 students, it only took around 12 hours to run the MTED detector. JPlag [14], a spiritual cousin to MTED, computes edit distance by the fractional coverage of shared substrings of tokens.

Inverted Token Index (InvToken) The InvToken detector attempts to detect partial copying and/or extreme reordering by using the same tokenization as IdentToken, but looking for matching substrings instead of identical token streams. Specifically, each 12-gram of a token stream is recorded. 12-grams that show up in 75% or more of the submissions are discarded. The similarity of two submissions is equal to the fraction of the 12-grams that were found in both submissions, with the weight of each 12-gram inversely proportional to the fraction of submissions containing that 12-gram across all submissions. Again, the N pairs with the highest similarity are output by the detector. The InvToken

[3]Algae's name was chosen in homage to the popular MOSS system [15].

Table 4: Fraction of plagiarism cases (by detector)

Detector	Percent of Plagiarism Cases Detected
Lazy	15.69%
IdentToken	18.49%
MTED	49.71%
InvToken	72.77%
InvIdents	67.35%
Obfuscation	0.38%

detector uses an inverted index to quickly find shared substrings, and so shares similarities to the popular [3] MOSS program. MOSS differs from InvToken in that it (1) uses a different pre-processor to generate the token stream, (2) uses a winnowing algorithm to remove code that comes from the assignment, and (3) treats every n-gram as having the same weight [15].

Inverted Identifier Index (InvIdents) The InvIdents detector attempts to find plagiarism through identifying variables and functions that are named in relatively unique ways. Using Clang, all of the identifiers in the source file are extracted, converted to lower case with any underscores removed, and written to a file. Then the similarity of two files can be computed based on the number of shared identifiers, with the weight scaled based on the infrequency of the identifier, as was done in InvToken. As with all collectors, the N highest scoring results are chosen for reporting to the user.

Statistical Individual Dissimilarity (Obfuscation) This detector looks for features indicative of the code having been obfuscated by a piece of software (*e.g.*, [1, 5]), by collecting a number of metrics from each submission and looking for outliers. Specifically, Clang is used to collect the length of the longest line, average identifier length, and the number of lines, whitespace characters, comments, functions, #defines, mathematical operations, and returns. For each of these nine statistics, the mean and and standard deviation are computed, and then a submission is flagged as likely obfuscated if its sum of the absolute value of the z-scores for each statistic exceeds a specified threshold.

4.2 Detector Performance

In this section, we report on the relative effectiveness of the different collectors. A summary table is provided as Table 4. Note that the percentages do not sum to 100% because some cases were identified by multiple collectors.

Surprisingly, 15% of detected cheating cases were effectively identical copies of other submitted works (Lazy). The IdentToken detector performed only 3% better than Lazy, which suggest that if students go to the effort to rename variables, they are likely to make other obfuscations as well.

The obfuscation detector only found three students with obfuscated code across all assignments. We should note that none of these cases appear to have been professionally obfuscated by software. Instead, students appear to have applied by hand many of the same transformations that a professional software code obfuscater would have. Overall, this suggests that professional software code obfuscation is not currently in practice by students in order to evade plagiarism detection.

The MTED detector proved far more effective than the trivial IdentToken and Lazy detectors (capturing nearly 50% of all detected cheating), but less effective than the inverted index based detectors (InvToken, InvIdents). When evaluating MTED clusters, we tended to notice that it would find matchings between people who "gave up" and only made trivial changes to the handed-out boilerplate, because these small code files would have small edit distances. This leads to a lot of false positives, as the number of such clusters grows quadratically with the number of students who have near-boilerplate code.

The InvToken detector outperformed all other detectors, catching 72.77% of all detected cheating. In several cases, it was able to successfully catch students who had plagiarized only part of their code (typically the harder to implement functions), which is something that the MTED detector struggled with. Runtime performance of InvToken (and InvIdents) was far better than that of MTED. Both detectors took around an hour to run on our corpus, which is over an order of magnitude faster than the MTED detector.

InvIdents also performed well, catching 67.35% of all detected cheating. In general, it outperformed the MTED detector, showing that shared identifier names are indeed a viable way to detect cheating.

With all of MTED, InvIdents, and InvToken, we noticed that students did indeed often reorder functions. Therefore, it is safe to assume that any effective cheating detector needs to be able to handle this code transformation.

In Table 5, we characterize the overlap between each pair of detectors. As expected, the Lazy and IdentToken detectors have a high degree of overlap (as InvIdents should technically find a superset of the Lazy detector's results). Between MTED, InvToken, and InvIdents, the highest degree of similarity occurred between MTED and InvToken at 63% overlap, and the lowest occurred between MTED and InvIdents at around 40%. These results suggest (because no two of these more advanced detectors have anywhere close to 100% overlap) that in order to fully detect plagiarism, multiple methods must be deployed.

5. CONCLUSION AND FUTURE WORK

Our key finding is that plagiarism appears to have a modest, but statistically significant, negative correlation with learning outcomes (as measured in terms of final course grades). We find this to be true in both the data structures course in which the cheating instances occurred, as well as a systems-oriented programming course taken later in the CS curriculum. A key piece of future research would explore this relationship to identify whether it is causal, and in which direction. That is, are weaker students more prone to cheating or does cheating weaken students.

In addition, we were surprised that our data suggests that providing legal means for students to collaborate led to no noticeable reduction in the rate of illegal forms of student collaboration (*i.e.*, plagiarism). We feel this non-intuitive finding deserves more study.

In addition, our work suggests that cross-semester sharing is a significant mechanism for plagiarism. While it is tempting to suggest that frequent development of new assignments might reduce the plagiarism rate, our data can neither support nor refute this hypothesis. We believe a study that compares the frequency and sharing patterns of plagiarism of re-used assignments with fresh assignments would be interesting future work.

Table 5: The percentage overlap between each pair of detectors, computed as the number of instances in common divided by the minimum of the number of instances caught by each detector.

	IdentToken	MTED	InvToken	InvIdents	Obfuscation
Lazy	62.83	30.77	23.41	19.61	0.00
IdentToken		40.66	29.15	21.15	0.00
MTED			63.03	39.48	0.83
InvToken				50.91	0.58
InvIdents					0.49

Finally, our research revealed that a significant portion of cheaters put very little effort into avoiding detection, but that more effective detectors are necessary to catch more advanced cheating. No single detector was able to catch more than 3/4 of total detected cheating (given the cluster counts we used), suggesting that multiple methods need to be employed for full effectiveness. While we believe that Inverted-index systems like InvToken and MOSS [15] offer the best combination of runtime performance and effectiveness, the InvIdents (which find the preservation of unique identifier names) detector provides an important complement to those techniques and, in particular, proved useful for distinguishing true positives from false positives while evaluating clusters. We believe that extending that detector to also consider comments and declared literals could further improve its effectiveness.

5.1 Acknowledgments

The authors would like to thank A. Mattox Beckman, Jr. and Cinda Heeren for their contributions related to this research.

6. REFERENCES

[1] C/C++ Obfuscator. http://stunnix.com/prod/cxxo/.

[2] Clang: A C language family frontend for LLVM. http://clang.llvm.org/index.html.

[3] K. W. Bowyer and L. O. Hall. Experience using "MOSS" to detect cheating on programming assignments. In *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, volume 3, pages 13B3–18. IEEE, 1999.

[4] S. Burrows, S. M. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, 2007.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[6] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A plagiarism detection system. *SIGCSE Bull.*, 13(1):21–25, Feb. 1981.

[7] S. Engels, V. Lakshmanan, and M. Craig. Plagiarism detection using feature-based neural networks. *SIGCSE Bull.*, 39(1):34–38, Mar. 2007.

[8] M. Freire, M. Cebrián, and E. Del Rosal. AC: An integrated source code plagiarism detection environment. *arXiv preprint cs.IT/0703136*, 2007.

[9] D. Gitchell and N. Tran. Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266–270, Mar. 1999.

[10] S. Grier. A tool that detects plagiarism in pascal programs. *SIGCSE Bull.*, 13(1):15–20, Feb. 1981.

[11] C. J. Hwang and D. E. Gibson. Using an effective grading method for preventing plagiarism of programming assignments. *SIGCSE Bull.*, 14(1):50–59, Feb. 1982.

[12] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 756–765, New York, NY, USA, 2011. ACM.

[13] J. Pierce. Algae, 2015. http://www.github.com/JonathanPierce/Algae.

[14] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[15] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[16] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh. Cheating and plagiarism: Perceptions and practices of first year it students. *SIGCSE Bull.*, 34(3):183–187, June 2002.

[17] G. Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138, 1990.

[18] M. Zeidner. *Test Anxiety The State of the Art.* Plenum Press, 1998.