ASEE 2022 ANNUAL CONFERENCE
Excellence Through Diversity

Paper ID #36462

Detecting Possible Cheating In Programming Courses Using Drastic Code Change

Nabeel Alzahrani

Nabeel Alzahrani is a Computer Science Ph.D. student in the Department of Computer Science and Engineering at the University of California, Riverside. Nabeel's research interests include code integrity, causes of student struggle, and debugging methodologies, in introductory computer programming courses.

Frank Vahid (Professor)

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, since 1994. He is co-founder and Chief Learning Officer of zyBooks, which creates web-native interactive learning content to replace college textbooks and homework serving 500,000 students anually. His research interests include learning methods to improve college student success especially for CS and STEM freshmen and sophomores, and also embedded systems software and hardware. He is also founder of the non-profit CollegeStudentAdvocates.org.

© American Society for Engineering Education, 2022
Powered by www.slayte.com

Detecting Possible Cheating In Programming Courses Using Drastic Code Change

Abstract

Automated code similarity detection tools are widely used in programming classes to detect students cheating who copied from classmates or from a common online source. But today, other ways of cheating are becoming common that do not yield code similar to classmates, such as copying from an online source that other classmates didn't find, or copying code developed by a hired contractor. Using a cloud-based programming environment that records every code compile/run, we detected code clearly not written by students in our class and noticed running of such code was sometimes preceded by a "drastic change" in their code history -- a run whose code is so dramatically different from the previous run as to be unlikely to have been derived normally from the previous run. Some students submitted such code right away, what we call an "initial leap". Other students tried writing code themselves, gave up, and then copied from an online source or hired a contractor, what we call "gave up". Among either group, we further noticed that some would sequentially try a variety of copied solutions attempting to find one that works, what we call "solution hopping," causing even more instances of drastic changes in the student's code history. Thus, we developed a tool, based on a simple "text diff" algorithm, to detect drastic code changes in student code progressions, and to point instructors to possible cheating cases. We conducted two experiments. The first experiment measured the accuracy of our tool. The tool averaged 100% sensitivity and 100% specificity using real data, and synthetic data. The second experiment studied the prevalence of drastic changes in real student programs in our course. The study showed about 32% of students in the initial leaps group, and 5% in the gave up group, which we manually confirmed as actual cheating. Furthermore, 24% of initial leap students and 47% of gave up students subsequently solution hopped. We plan to make our drastic change detection tool available to the CS community as a free web tool.

Introduction

A common form of cheating on programming assignments involves a student initially trying, then struggling, and eventually giving up and copying a solution from elsewhere. Regarding such cheating, Malan [1], who teaches Harvard's CS50, notes "All too often were students' acts the result of late-night panic". Fig. 1 provides an example from real code developed by a student at our university, recorded by the cloud platform we require students to use, with changes between successive code pairs highlighted. The student was required to write code that finds the maximum of three input values. The first four code snippets show the student's code being developed along a normal progression. But, the student eventually gets stuck, having failed to consider cases where input values are equal. Giving up, the last snippet shows code the student copied from an online source (which we found online).

Another common form of cheating involves students not even trying to write the program themselves, but going straight to online solutions or contractors.

Many programming courses, especially classes with large numbers of students, use automated code similarity detection tools such as MOSS [2] to help detect potential cheating. Such tools focus on detecting similar submissions from multiple students. However, for the above situation, similarity

detection fails to detect cases where a student submits a copied solution that is unique, such as code obtained from today's popular online tutors who provide solutions for just a few dollars and often within just tens of minutes [3].

```
if (x > y) {
  cout << x;
}
```

```
if ((x > y) & (x > z)) {
  cout << x;
}
```

```
if ((x > y) && (x > z)) {
  cout << x;
}
```

```
if ((x > y) && (x > z)) {
  cout << x;
}
if ((y > x) && (y > z)) {
  cout << y;
}
if ((z > x) && (z > y)) {
  cout << z;
}
```

```
(Drastically changed)
if ((num1 >= num2) && (num1 >= num3)) cout << num1;
else if ((num2 >= num1) && (num2 >= num3)) cout <<
num2;
else cout << num3;
```

Figure 1: The first four code snippets follow a normal progression, but the fifth code snippet labeled *Drastically changed* shows a drastic change, involving the arbitrary change in program identifiers, the switch from brace use to no braces, and a different algorithm.

Fortunately, today a different type of automation is possible due to the trend of instructors having access to the history of a student's code. For example, some instructors require students to frequently commit code to GitHub [4]. Thousands of courses and hundreds of thousands of students now use auto-graders [5], and many such auto-graders record the code every time the code is submitted for auto-grading [6]. Some instructors even have the students develop code in IDEs that auto-save the code at regular intervals, or that save the code every time the student runs the code [7].

Thus, to detect some forms of cheating, a new approach complementary to similarity checking is possible today. The approach looks across the progression of a student's saved code, trying to detect the case where a student was submitting a normal code progression, but then (perhaps in "a late night panic") suddenly submits very different code -- what we call a "drastic change". As with similarity checking tools, a drastic change detector does not label anything as cheating, but rather simply points

instructors to cases that the instructor (or TAs) may wish to investigate. Thus, a key challenge is to build a highly-accurate drastic change detector, to avoid wasting an instructor's time with false positives, and to avoid missing drastic changes as well (false negatives).

This paper presents a tool we have developed, and used in our courses, for examining code progressions to detect a "drastic" change suggestive of cheating. Our goal is to make this tool freely available to all programming instructors, akin to MOSS' availability. A key challenge is to eliminate false positives, because students often make substantial (but not drastic) changes to code during a normal development process. The paper provides experiments on both synthetic and real code progressions, showing the algorithm yields good accuracy. The paper also presents an experiment on the prevalence of drastic change in real student code.

Problem Statement

The input to our tool is a source code progression for each student on a programming assignment, as in Fig. 2. Student A has m code runs and Student B has n code runs. Code A1 is a link to the entire source code for Student A's first code run.

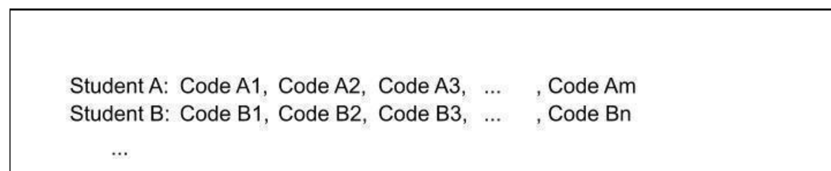

Figure 2: Code progressions for two students on a programming assignment. For simplicity, we sometimes refer to each code instance as a "run".

We get the code progressions in from zyLabs offered by zyBooks [8], which records source code for every program submitted for auto-grading, and if the instructor requires the use of the built-in IDE, also records the source code every time a student runs their program during development. zyLabs allows an instructor to click a button to download a log file (a lab file) of all student code progressions for any programming assignment. Our approach also applies to any other tool or scenario where an instructor can access student code progressions, such as when using other commercial or custom auto-graders and/or IDEs that make available history of student code, or when requiring students to commit code regularly to GitHub.

In our problem definition, the approach examines every pair of sequential code instances in the progression, comparing the pair's first code (Code1) with the pair's second code (Code2). In Fig. 2, the approach compares Code A1 and Code A2, then Code A2 and Code A3, etc. (In the future, we may consider a broader definition that at once considers a larger region of the code progression).

For any pair (Code1, Code2), we assign a "drastic change" value between 0 and 1, where 0 means almost no change, 1 means a very drastic change, and a value of 0.5 or above means an instructor may wish to examine the code for possible cheating. The approach never concludes cheating itself, but rather, like MOSS, directs instructors to suspicious cases. A "drastic change" is informally defined as a suspiciously large change that an instructor may want to inspect for possible cheating. Not all large

changes are suspicious, so the key here is to try to match what instructors would consider possible cheating. We also classify drastic changes into two categories: initial leaps and gave ups. An initial leap occurs when the first code run size (lines of code) is bigger than 50% of the first highest-score code run (the first code run with a full score) size in a student code progression. We skip the initial leap detection if the first highest-score code run size is less than 15 lines of code or if a student does not have a code run with a full score. A gave up is a drastic change anywhere in the code progression except in the first code run. We also have solution hopping as a sub-category for each drastic change category. Solution hopping is a drastic change in addition to an initial leap or a gave up. Our tool detects all these categories as a “drastic change”.

Text Comparison Based Solution

Experienced programmers can readily notice drastic changes in code. One indicator might be a sudden and arbitrary change in program identifiers, such as changing x, y, z to num1, num2, num3 in Fig. 1. Another is changing code style, such as changing from consistent brace use to no braces in Fig. 1. Another is using a different algorithm, as in switching from a multiple-if structure to an if-else structure in Fig 1. Any one such change may not be suspicious, but multiple such changes between two runs increase suspicion.

We wish to detect drastic changes automatically. Perhaps the most obvious approach to automatically detecting drastic change (for gave ups and solution hoppings) is to just use a "diff" tool on each code pair in a student's progression. Diff is a text comparison tool that finds added, changed, and detected characters between files, and is a built-in utility in most operating systems [9]. We used the diff library that is part of the Python language [10]. We noticed (from synthetic data analysis, explained later in the experiments section) that drastic change mostly occurs when 1 - diff’s returned value (the similarity ratio or simRatio for short) is less than or equal to a threshold of about 0.8. Therefore, we considered any code progression with a simRatio less than or equal to the threshold as a drastic change, and return 1 - simRatio as that drastic change value. For simRatio above 0.8, we set the drastic change value to 0, meaning the code pair is quite similar. Our tool is orthogonal to the used programming language in programming assignments, and supports any programming language.

Experiments

We ran two experiments to answer two research questions: (1) What is the accuracy (in terms of sensitivity and specificity) of our tool in detecting drastic changes?, and (2) What is the prevalence of drastic changes (in terms of initial leaps, gave ups, and solution hoppings percentages)?. We used a quantitative approach, probability sampling, and statistical analysis (sensitivity/specificity for the first experiment, and percentage for the second experiment) to conduct both experiments, but used experimental design for the first experiment, and descriptive design for the second experiment.

In the first experiment to test the efficacy of our tool in detecting drastic changes (Experiment 1, the tool’s efficacy experiment), we carried out the experiment on synthetic progressions, where we injected drastic changes into real progressions, and on unmodified real progressions where we manually discovered actual drastic changes in student code.

In the second experiment to find the prevalence of drastic changes (Experiment 2, the prevalence of drastic changes experiment), we carried out the experiment on real data.

In each experiment, we used 2 zyBook lab files, one lab file for a 240-student C++ CS1 section (176-line instructor solution) and and the other lab file for a 430-student Python CS1 section (121-line instructor solution) in Fall 2019 at two large research universities.

In the tool's efficacy experiment, for a chosen lab, we sorted the student roster in descending order by their student ID and picked 20 students who received a full score (10 points) and had no drastic change in their code progressions after examining their progressions manually.

We randomly divided the 20 students into two groups of 10 students, Group 1 and Group 2. For each student in Group 1, we replaced the first full-score-code-run of one student with the first full-score-code-run from the corresponding student in Group 2. The reason for doing so is to introduce drastic change for each student in Group 1.

For each student in Group 1, we manually examined each code pair in their progression for drastic changes, and manually recorded the results. The results were 10 students with drastic change. Then, we ran our drastic change detection tool on both groups to detect drastic changes with a drastic change value of 0.5 and above. The tool detected 10 students with drastic change in the Group 1 and 10 students with no drastic change in the control Group 2. This concludes the experiment using synthetic progressions.

Again, in the tool's efficacy experiment, for a chosen lab, we sorted the student roster in descending order by their student ID and picked 30 students who received a full score (10 points) where 10 students had drastic change, 10 students had no drastic change, and 10 students had big change (we define big change as having more than 20 lines of code difference between 2 consecutive code runs) after examining their progressions manually.

The reason for doing so is to have a balanced sample of students where 10 students had possibly drastic change, 10 students had possibly no drastic change, and 10 students with **big change** to see how accurately our tool can discern drastic change from big change.

For each student, we manually examined each code pair in their progression for drastic changes, and manually recorded the results. The results were 20 students with drastic change and 10 students with no drastic change. Then, we ran our drastic change detection tool to detect drastic changes with a drastic change value of 0.5 and above. The tool detected 20 students with drastic change and 10 students with no drastic change. This concludes the experiment using real progressions.

We conducted the tool's efficacy experiment twice (Experiment 1 and Experiment 2). For Experiment 1, we used 30 students from the C++ lab file. For Experiment 2, we used 30 students from the Python lab file. For Experiment 2, the results for synthetic data manual and tool analysis were 10 students with drastic change and 10 students with no drastic change. For the real data, the manual and tool analysis results were 15 students with drastic change and 15 students with no drastic change.

Table 1 presents the drastic changes (gave ups) accuracy using sensitivity and specificity for each of the 2 experiments. For both experiments, the tool averaged 100% sensitivity, and 100% specificity.

In the prevalence of drastic changes experiments, we used the same 2 lab files as they are without changing any data, so the experiments used real code progressions. However, in this experiment, we did not limit the experiments to only 30 students per lab, but included all students for a total of 670 students for the two labs. Table 2 summarizes the prevalence of drastic change experiments in real student data for Experiment 1 (using the C++ lab file) and Experiment 2 (using the Python lab file).

We believe the high average ratio of initial leaps of 32% might not be all related to possible cheating, and some might be rather related to students developing their code in their favorite IDEs then copying their code to the zyBook platform, hence triggering a high average ratio of initial leaps. Instructors can award students points for developing their code completely inside the cloud platform IDE to reduce the chance of such inaccuracy.

Table 1: The tool's efficacy experiments.

Experiment	Synthetic data set (#students = 20)		Real data set (#students = 30)	
	Sensitivity	Specificity	Sensitivity	Specificity
1	100%	100%	100%	100%
2	100%	100%	100%	100%
Average	100%	100%	100%	100%

Table 2: The prevalence of drastic changes (initial leaps, gave ups, solution hoppings) experiments.

Experiment	(N = #students)	Students with initial leaps	Students who gave up
1	(N = 240)	30% (19% of those then solution hopped)	5% (18% of those then solution hopped)
2	(N = 430)	34% (28% of those then solution hopped)	5% (76% of those then solution hopped)
	Average	32% (24% of those then solution hopped)	5% (47% of those then solution hopped)

Discussion

Recently, we considered whether the diff that comes with the version control system Git might be more appropriate for drastic change detection than the Python diff. According to Nugroho [12], Git uses Myer's algorithm to produce its diff. We tested Google's implementation of Myer's algorithm [13]. We noticed similar results when using Myer's algorithm for most cases (except for code improvement and code relocating). Switching to Myer's algorithm may be considered in future work. Also, code templates could alter our results. A code template is prewritten code (predefined variables, function layout, etc.) perhaps provided by the instructor, which students might add to their code, which could increase the code size if added yielding a high drastic change value. We used a simple method to detect templates by automatically noting the minimum common code across the top 10% of students who received full scores and had the smallest first code runs. However, other methods might be more accurate. Future work may include allowing template code to be provided as input to the tool. Moreover, our tool spends on average 15 minutes for each lab in the experiment, over 95% of which is spent just downloading code runs from the zyBooks server. In the future, we plan to investigate ways of detecting drastic change without having to download all code runs, by downloading a select subset. Also, the tool only examines two adjacent code runs at a time for simplicity, rather than examining the entire sequence of code runs, which could provide more insight into drastic change.

Acknowledgments
This material is based upon work supported by the National Science Foundation under Grant No. 2111323.

Conclusion

Commonly-used automated code similarity detection tools are not effective in detecting possible cheating in unique solutions. We introduced a tool to supplement similarity detectors to detect possible cheating using drastic changes in student code progression. The experiments showed that using diff is accurate for drastic change detection. On average, the tool achieved 100% sensitivity and 100% specificity. We also ran an experiment to detect drastic changes (initial leaps, gave ups, and solution hoppings) in real student code. The experiment revealed that 32% of drastic changes are initial leaps (24% of those solution hopped) and 5% are gave ups (47% of those solution hopped). We plan to make the tool available for the CS community as a free web tool.

References

[1] D. J. Malan, B. Yu, and D. Lloyd, "Teaching academic honesty in CS50," in Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 2020, pp. 282–288.
[2] A. Aiken, "MOSS: A system for detecting software similarity," Retrieved August, vol. 29, p. 2017, 1994.
[3] S. Manoharan and U. Speidel, "Contract Cheating in Computer Science: A Case Study," 2020, pp. 91–98.
[4] "GitHub: Where the world builds software," GitHub. https://github.com/ (accessed Jan. 12, 2022).

[5] "The rise of the zyLab program auto-grader in introductory CS courses" https://docs.google.com/document/d/e/2PACX-1vQYwxlY738_9zFFwOer1kKTNGuJx1Qe3IDW8XHf_OOYbaq9Drf_a9ljCqjcHY9Vv4ryPK423W7FmHwZ/pub (accessed Jan. 12, 2022).
[6] "IDE | CodeHS." https://codehs.com/ide (accessed Jan. 12, 2022).
[7] "zyLab Autograder, with Free Sample Labs in Java," zyBooks. https://www.zybooks.com/catalog/zylab-autograder-with-free-sample-labs-in-java/ (accessed Jan. 12, 2022).
[8] "zyBooks - Build Confidence and Save Time With Interactive Textbooks," zyBooks. http://www.zybooks.com/home/ (accessed Jan. 12, 2022).
[9] "diff(1) - Linux manual page." https://man7.org/linux/man-pages/man1/diff.1.html (accessed Jan. 12, 2022).
[10] "difflib — Helpers for computing deltas — Python 3.9.6 documentation." https://docs.python.org/3/library/difflib.html (accessed Jan. 12, 2022).
[11] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in Git?," Empirical Software Engineering, vol. 25, no. 1, pp. 790–823, 2020.
[12] google/diff-match-patch. Google, 2021. Accessed: Jan. 12, 2022. [Online]. Available: https://github.com/google/diff-match-patch