Grass: Automatically Generating Reviewer-friendly Evidence Reports from Plagiarism Detection Tools

Jennifer Alexandra Thompson, Virginia Polytechnic Institute and State University
Prof. Stephen H Edwards, Virginia Polytechnic Institute and State University

Stephen H. Edwards is a Professor and the Associate Department Head for Undergraduate Studies in the Department of Computer Science at Virginia Tech, where he has been teaching since 1996. He received his B.S. in electrical engineering from Caltech, and M.S. and Ph.D. in computer and information science from The Ohio State University. His research interests include computer science education, software testing, software engineering, and programming languages. He is the project lead for Web-CAT, the most widely used open-source automated grading system in the world. Web-CAT is known for allowing instructors to grade students based on how well they test their own code. In addition, his research group has produced a number of other open-source tools used in classrooms at many other institutions. Currently, he is researching innovative methods for giving feedback to students as they work on assignments to provide a more welcoming experience for students, recognizing the effort they put in and the accomplishments they make as they work on solutions. The goals of his research are to strengthen growth mindset beliefs while encouraging deliberate practice, self-checking, and skill improvement as students work.

Dr. Bob Edmison, Virginia Polytechnic Institute and State University

Dr. Edmison is a Collegiate Associate Professor in the Department of Computer Science at Virginia Tech. His research focuses on alternative grading practices, accessibility, and the tools to support them.

Grass: Automatically Generating Reviewer-friendly Evidence Reports from Plagiarism Detection Tools

Abstract

Addressing student programming plagiarism presents a dual challenge: initial code similarity detection and subsequent formal case reporting. While tools like MOSS are effective at the first step, instructors face a significant, time-consuming manual effort in verifying cases, identifying collusion groups, and translating complex technical evidence into a defensible report for a non-computing academic integrity board. This manual preparation often requires undoing "disguising" techniques used by students and summarizing results for a lay audience. This paper describes experiences with Grass, a tool designed to automate this critical reporting workflow. Grass works by feeding submissions into a similarity detection tool like MOSS and then analyzing the raw similarity results to identify multi-student collusion groups, processing the source code to remove obfuscatory differences, and generating case reports designed for non-programmer reviewers. These reports include a written summary of the findings, multiple descriptive visualizations, and side-by-side code comparisons that highlight essential structural similarities. The primary objective of Grass is to minimize the manual work required after a potential violation is detected. By providing automatically generated, high-quality, and easily digestible evidence reports, Grass significantly reduces the workload on course staff, encourages consistent adherence to academic integrity policies, and streamlines the path from initial detection to disciplinary review. This paper describes the design of Grass, how it operates, and our experiences using it over multiple programming courses.

1 Introduction

Research into automated plagiarism detection spans decades, and the ability of programs to automatically compare multiple texts continues to improve and evolve. However, detecting possible plagiarism is only the first step in a (potentially lengthy) process. At schools where there is a formally defined academic integrity process, instructors typically have to report the incident and provide the student work involved. However, when students attempt to conceal plagiarism by rewriting comments, renaming variables, and rearranging code, the situation may be less apparent to those without a programming background. Preparing summaries of incidents for a non-programming audience takes additional effort that is not addressed by focusing only on detection. As noted by Albluwi [1], a key area for future work is creating tools that support the entire workflow of documenting and filing plagiarism within an institution.

We have identified the following key concerns that can increase reporting overhead:

- Collecting supporting evidence that indicates plagiarism is occurring, including source code, timestamps, and histories of submissions when students make multiple submission attempts to automated tools as they work.
- Clustering the results of the plagiarism detection tool to identify groups of students who have colluded, rather than focusing solely on pairs of similar assignments.
- Processing the code to aid in readability by non-programmers, and to address the effects of disguising techniques that students often use to try to make plagiarism less visible. It is important for non-technical readers to understand what portions of the code indicate plagiarism to the reporter.
- Compiling a final report intended for a non-programming audience, summarizing the gathered data along with the source code comparison into one formatted document for submission to the institution's academic integrity process.

Currently, no widely-used methods exist to streamline the process after detection. The time spent working on plagiarism reporting should be minimized as much as possible. Instructors stand to benefit from automation throughout the entire plagiarism detection and reporting process, rather than just the front-loaded task of similarity detection.

To this end, we created Grass, a tool designed to both automate the use of MOSS and generate comprehensive reports for instructors to review and submit to academic integrity departments. Built for the Spring 2021 semester, this system automates the resolution of these points as much as possible. The automated steps are executed with minimal manual intervention to connect their input and output, and the results are then presented to teaching staff for review and submission. The implementation is open-source and available for use as a docker image (`https://github.com/web-cat/grass`).

This paper describes the approach used, along with our experiences using this early implementation throughout a semester-long CS 2 course. This tool has significantly reduced the amount of work needed to report academic integrity violations. Instead of spending hours examining evidence to determine the possibility of plagiarism and gathering evidence to support the case, a system like this can significantly reduce or eliminate the manual effort of compiling case reports. Through our early experiences, this has clearly simplified the reporting process. A single part-time graduate assistant can manage report generation and verification for a class of over 500 students. The instructor then only has to review the compiled reports before submitting to the academic integrity process, and the reports are immediately ready for a non-technical, non-programming audience to review.

2 Background

One well-used definition of source plagiarism comes from Parker and Hamblen: "a program which has been produced from another program with a small number of routine transformations" [2]. A detailed review of the academic state of source code plagiarism detection by Novak, Joy, and Kermek examined multiple sources and distilled sixteen unique styles of transformation students may undertake when plagiarizing [3]. These transformations broadly fall into the categories of logical transformation and cosmetic textual transformation. According to

the authors, the majority of existing literature focuses on detecting transformations that do not manipulate the underlying logic of the program at all. This may be based on the intuition that students more often cosmetically alter source code in an attempt to hide plagiarism, while logical transformations may occur less often in practice. Some plagiarism detection methods do not directly examine source code, like Tahei and Noelle's investigation of resubmission patterns by students [4].

At the center of the Grass report generator, we needed to select an existing tool to perform comparisons of student work. Novak, Joy, and Kermek discovered 120 publications proposing new plagiarism detection tools [3]. Here, we summarize the most popular plagiarism detection tools, drawn from multiple sources [3, 5, 3, 6, 7]. In our search, we identified these characteristics as most important in a plagiarism detection tool:

- Easy execution: Requiring complex setup hampers distribution.
- Understandability: Non-experts need to understand the report.
- Sensitivity adjustment: In our experience, evidence corroborated from multiple sensitivity settings is more difficult to contest.
- Low false positive rate.
- Support for many programming languages.

2.1 Examination of Available Tools

Plagiarism detection has been intensely studied, and the ability of programs to automatically compare multiple texts continues to improve and evolve. In the field of computer science, most source code plagiarism detection tools are based on a similarity measure, where the contents of all code is transformed or abstracted and then compared to each other. The specifics of the method and measure vary for each plagiarism detection tool.

JPlag allows users to visualize similarities between any two students' code, either through a Web service, or using an offline version that can be run locally [8]. It is open-source, actively developed, and supports Java, C#, C, C++, Python 3, and Scheme. JPlag uses a greedy string tiling algorithm that allows for sensitivity tuning by adjusting the minimum token string length required to be marked important. However, this algorithm is solely concerned with matching tokens between two texts, and JPlag cannot easily account for skeleton code that may be provided to all students at the start of an assignment. While many of these details are promising, the strictly pairwise comparisons and lack of ability to compensate for instructor-provided or commonly-occurring are important limitations.

Plaggie is an open-source plagiarism detection tool written in Java [9]. The exact details of its algorithm and operation are not discussed in the paper, but the source code's Readme file uses a greedy string tiling algorithm, referencing JPlag. The introduction paper notes that the source code is unlikely to be updated, and indeed it appears to be unchanged since its introduction in 2006. While it does offer sensitivity tuning and a generated HTML report, it lacks proper support and updates.

SIM is another open-source plagiarism detection tool that supports C, C++, Java, Pascal, Modula-2, Miranda, Lisp, and 8086 assembler code [10]. It tokenizes input code, creating a parse tree of both programs, and then finds the shortest alignment of a section of code into the other program. While it seems effective against the easiest and most common forms of cosmetic code altering, it has similar limitations to JPlag: It can only compare students pairwise, and cannot account for intentionally provided code. As a preliminary tool it may be useful, but it lacks extended features.

Sherlock is an open-source, but abandoned, tool from the University of Warwick [11]. It uses a language-agnostic algorithm that looks for similarities across five different transformations of the original source text, and runs an n^2 comparison across all student-pair combinations in an attempt to cluster submissions together. While it does offer sensitivity tuning, its ability to cluster students is solely based on all students having passed an arbitrary similarity threshold. In addition, its algorithm is lossy, deliberately discarding portions of input code to speed up its computation.

MOSS is a closed-source web service that uses a winnowing algorithm to detect similarities in provided source code [12]. It supports a wide variety of languages, and provides information from a slightly different angle than the previous tools. The winnowing algorithm is similar in many ways to the greedy string tiling algorithm used by other tools, but one of the key differences is that MOSS's similarity score tells the user "the portions of code marked similar appear in no more than m submissions." This significantly changes the approach to detecting plagiarism, as it allows the user to think about students operating in groups larger than two. By adjusting this sensitivity using different values for m, a fine-grained sliding scale can be created to observe how similarity changes as more submissions are allowed to share a piece of code in common. Additionally, this allows one to automatically exclude code that is repeated by a large number of students—such as code provided by the instructor, appearing in the textbook, or forming a common idiom taught in the class. For these reasons, along with the simplicity of uploading source files to a web service, we ultimately selected MOSS as the similarity detector for our report generation system.

It is important to note that while we ultimately decided to use MOSS as the plagiarism detection tool for our report generator, the additional steps after initial detection are still concerns regardless of the similarity detector chosen. For any similarity detection tool, it should be possible to modify our approach to automate the remaining tasks, so any tool could be used in principle. However, the m parameter that MOSS provides for determining when code occurs frequently enough to be considered "common code" is one powerful feature that we employ during evidence collection, and that is not yet readily available in other plagiarism detection tools.

3 Design

Grass is broken down into four main components, as shown in Figure 1. Each component can be thought of as a separate module. The plagiarism detection tool (here, MOSS) identifies the students to examine. After all modules are run, the report generator then compiles all output from the previous modules into a single report.

In this section, we will discuss our initial implementations of each component. This implementation is publicly posted on GitHub [13]. Before these modules can do their work, the

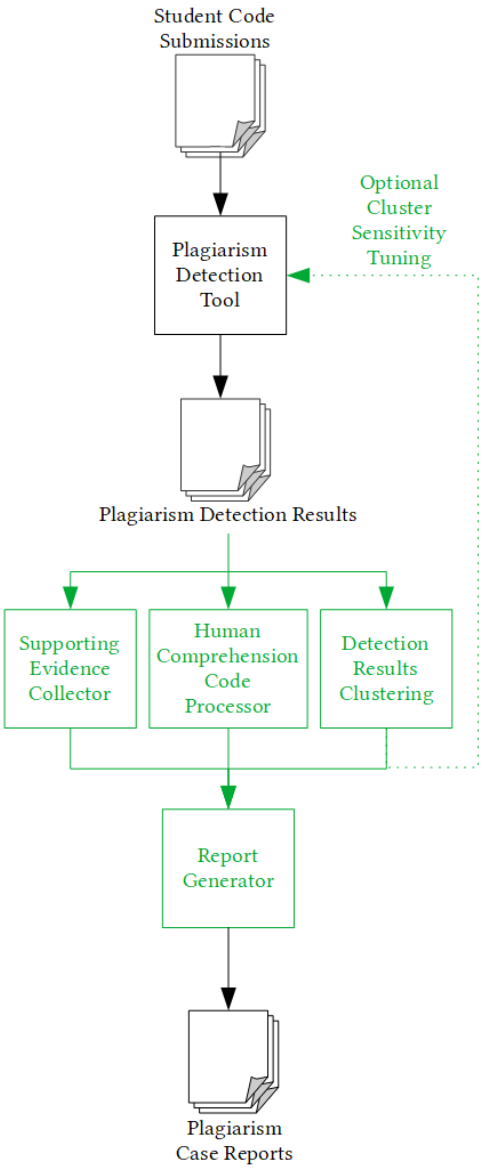

Figure 1: Workflow of the case report generation system, with new automation steps in green.

main plagiarism detection tool needs to be run. For this, we have also set up some automation to make the process straightforward.

3.1 Initial Plagiarism Detection

Before any components are run, we gather students' final code submissions. Additionally, any unauthorized assignments found online (e.g., GitHub or Chegg) can be added as input to be compared against student submissions.. Using the Mosspy library [14], Grass submits these submissions to MOSS with different m-values to gain a broader perspective of student code similarity. From there, the HTML page MOSS generates is downloaded and examined

programmatically. In this implementation, the similarity threshold is determined by the instructor. Students whose similarity score is higher than this value are identified as suspicious, and their results saved for further review.

As mentioned previously, the use of MOSS in particular is not integral to the tool's operation. Any plagiarism detection tool that produces parsable output could be used. In its current form, the system does not rely on the specific similarity measure returned by MOSS, other than to determine which students are suspicious.

3.2 Clustering Suspicious Students

Once the suspected students are returned, they can be clustered into groups of collaborators. Even though MOSS (and many tools) report similarity in student pairs, it is not uncommon for plagiarized work to be reused by multiple students, forming larger groups. The goal of clustering is to identify these groups so that collected evidence can be calibrated for the group size. Further, this process results in one report for each entire cluster of students, instead of having a report for every combination of students in the cluster. Due to the nature of the partitioning, it is possible that not all combinations of student pairs will be obtained from the MOSS report. This issue is solved with basic graph theory. If students are nodes, then the existence of a MOSS result is an edge between two students. A cluster of students is a group of nodes that can all reach each other.

In Grass, further MOSS reports are requested once the students are clustered. These reports use a sensitivity value specific to the size of each cluster. MOSS's sensitivity parameter allows one to define how "rare" (or infrequent) a code segment must be before it is considered significant in measuring similarity. By adjusting this parameter based on the cluster size, it is possible for MOSS to report how much code is uniquely repeated only within that cluster, and not by any other students. This data on *uniquely repeated code* that is not used by other students is useful to support the argument that the marked similarity us unlikely to result from chance.

The amount of code that is unique to a cluster can be a strong indicator of plagiarism. Students may claim that "there is only one way to solve this problem, so solutions naturally look similar," Pointing out code that is unique to only the students within a cluster makes this position difficult to defend, especially in classes with hundreds of students. This measure of unique duplicated code is saved for use in final report generation. Additionally, the clustering of students is saved, so that all students deemed plagiarizing together can be combined in a single report.

3.3 Collecting Supporting Evidence

With a list of students suspected of plagiarism, more information about their submissions can be gathered. This evidence may not be enough to make a case on its own, but can help provide context to make a better informed decision.

Students in our courses use Web-CAT, an automated grading system, to submit their work. Web-CAT allows instructors to download information about every submission students have made for an assignment. From this data, we extract:

- Time of each student's first and last submission
- Total number of submissions by each student
- Automatic grading score for every submission
- Average number of submissions across the class

When organized well, this data can allow a reviewer to reasonably judge whether there are correlations between student submission information. A few examples based on real cases that were discovered and reported may illustrate this utility.

Consider a case where one student does not begin submitting until after the second student has made their final submission. Student 1 may rapidly make a few submissions in a short period of time. This may be corroborative evidence when Student 1 has copied Student 2's code, and rapidly resubmits obfuscated code or other minor changes.

Students with an extremely small number of submissions may also be suspect. Receiving a perfect score within one or two submissions is atypical. If a student meets these criteria while also having code similar to another student, the reviewer may view this as a further indication of plagiarism.

Finally, the score assigned by the auto-grader may provide supporting evidence. When two students share code, they may receive nearly identical scores. This can run alongside the first example case, where a student only begins submitting after another has finished. If Student 1 copied all or a majority of Student 2's code, then it is likely that the score for Student 1's first submission would be identical to Student 2's final submission.

3.4 Code Comprehension Processing

In an effort to conceal their plagiarism, students may use a variety of techniques to evade detection [3]. These obfuscation attempts are most effective against people who do not regularly read source code, including those who may participate in academic integrity case processing. To help these non-technical reviewers better understand the similarities between source code, we propose normalizing submission source code to aid in reader comprehension. For the purpose of submission to an academic integrity process, the original code should always be included. However, a modified version of the code may better highlight the essential similarities for non-technical readers.

Renaming variables, rewriting comments, and reorganizing or reordering code are common methods of disguise used by students. While detection tools typically ignore these differences when measuring similarities despite these changes, a person reading source code to manually verify plagiarism must be able to visualize code similarities in spite of any obfuscation attempts. Grass makes this process easier by taking steps to counteract some superficial modifications that could mislead reviewers.

Once the list of suspect students are taken from the MOSS report, we normalize their source code to create a side-by-side comparison. This process counteracts many basic attempts at plagiarism obfuscation. We normalize code by:

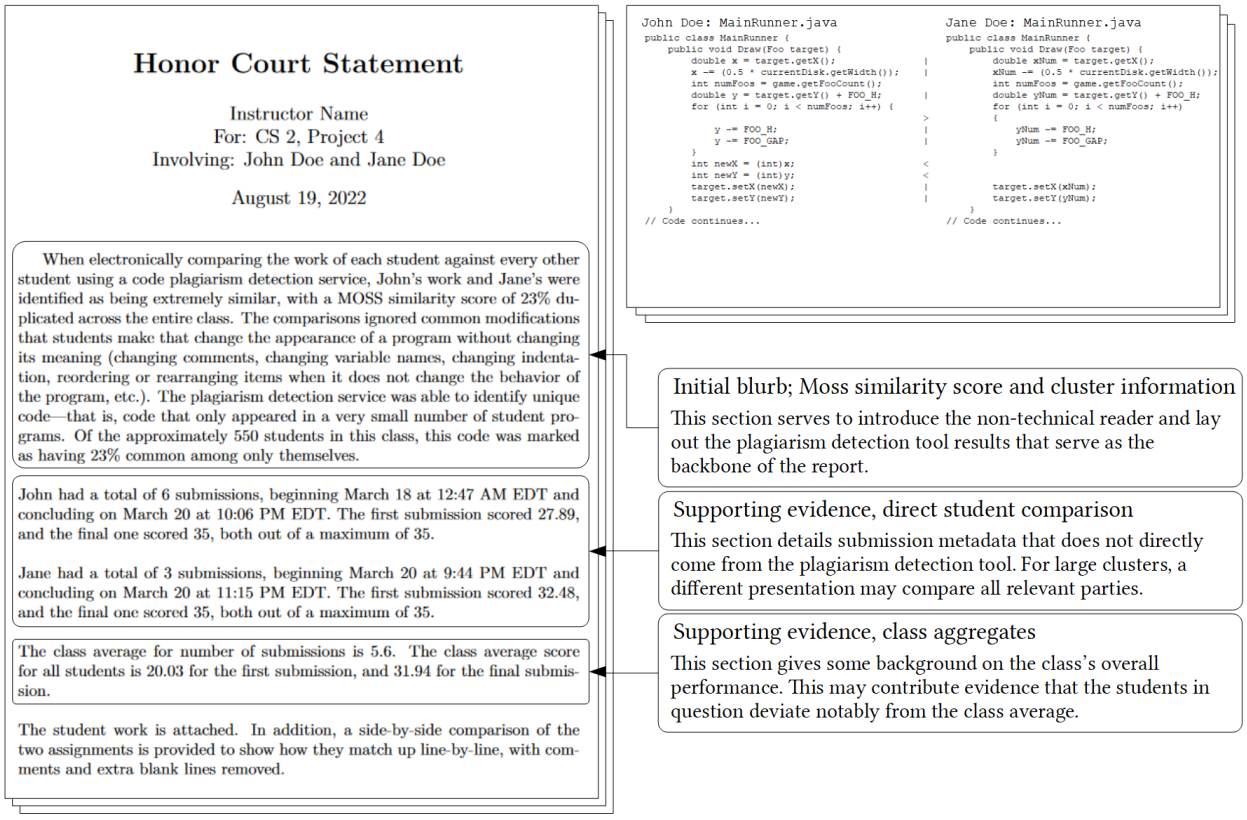

Figure 2: Example of an automatically generated report. The original source code and modified versions follow the first page.

- Removing comments and blank lines.
- Sorting declarations in source files by function length. All student code was written in Java, so methods within each class are sorted by length.

We can then use `diff` to generate a side-by-side comparison of the source files, ignoring whitespace differences. In multi-file submissions, files were compared alphabetically, as project specifications required nearly all files to have specific names. A compilation of all side-by-side comparisons is included in the final report. Figure 2 illustrates how the side-by-side `diff` comparison can guide non-technical readers to quickly understand the similarities and differences between two students' code.

3.5 Final Report Generation

After completing the previous processes, the final report can be generated. Grass can generate reports as LaTeX or DOCX files; we use DOCX for its easy-to-modify capabilities, allowing instructors to add details that strengthen a case. Reports also include an informational handout on reading code to support a non-computing audience's understanding. The reports were organized in the following manner:

- Incident students' names, and assignment

- Results from plagiarism detection tool
- Supporting evidence gathered
- Human comprehensible edited source code comparison
- Original submission source code

Figure 2 contains an example of the cover page of such a report, showing all information except source code, which effectively acts as an appendix to the report. One important point to note is that as group clusters increase in size, the number of side-by-side comparisons becomes unmanageable. To resolve this, large clusters only include code comparisons against one student designated as the cluster exemplar. The cluster exemplar is the student whose code is the most similar to all other students in the cluster (the *medoid* of the cluster, selected to approximate the cluster's *centroid*). Similarly, the presentation of MOSS results includes a chart detailing similarity scores across all pairs of cluster members.

4 Experiences

The first version of Grass was piloted on a single CS2 course for the Spring 2021 semester. The course had approximately 550 students enrolled, and we ran this system on 4 programming projects over the course of the semester. For the specific purpose of managing the plagiarism reporting process, a single graduate teaching assistant was added as a part-time worker to generate reports from each programming project and to verify which generated reports should be further submitted to the academic integrity process.

Overall, the process of submitting plagiarism reports was very straightforward. Running the various programs that make up each module required only a few minutes of manual input. That time could be further reduced by binding the various modules together in a single program. Once the reports were generated, they were handed off to the graduate assistant for review.

The graduate assistant reported spending approximately 5 hours per assignment reviewing the generated case reports. After verifying the results, a few more hours were spent looking for unique patterns within the MOSS summaries. While the MOSS results already highlight that information, some portions of code are more suspicious or interesting than others. The assistant also commented that in some cases, the first submission for a student was useful to look at rather than only the final submission, and requested that to be included in the generated reports.

Notably, after one project had already been analyzed by our system, the instructor for the class discovered a small number of GitHub repositories containing code for all of the assignments in the class posted by prior students. Regardless of the intent behind creating such a repository, these repositories could be freely accessed and plagiarized by any students. Upon discovering this, we downloaded the source code from the repositories and included them as submissions to MOSS when analyzing the corresponding assignments. From there, the code was treated as if it were a regular submission. This identified numerous students who had code very similar to the GitHub repositories. While these students would have been detected regardless of the inclusion of the GitHub code, this discovery allowed for extra detail to be added to these reports, pointing towards the repository as a possible source used in plagiarism.

Group Size	% Reports	% Reported Students	% Class
2	77.8%	66.1%	6.9%
3	11.1%	15.2%	1.6%
4	3.7%	3.4%	0.4%
5	3.7%	5.1%	0.5%
6	3.7%	10.2%	1.1%

Table 1: Report generator results over the semester.

Table 1 contains information regarding different sizes of clusters reported across the entire semester. A cluster only needs a single report, and Column 2 is calculated based on the total number of reports generated. Column 3, however, compares the number of students in a cluster size against all students reported. Finally, Column 4 reports the number of students reported compared against the entire class. Two-thirds of the students involved occurred in student pairs, but one third of the students reported occurred in clusters of size 3 or more. The larger cluster sizes were primarily plagiarizing from the previously mentioned GitHub repositories, and may not have been the result of direct student collaboration.

In addition, the authors of this paper have prior experience reporting plagiarism for this course. Without a more rigorous study, we cannot provide a comparison between the number of plagiarism cases reported with this automated system and with manual methods. However, the overall experience is significantly improved for the course staff involved when using this system. In particular, the amount of work required to report students was dramatically reduced. Being able to cluster students automatically reduced concerns that only a part of a plagiarizing group was being reported. While students were clustered by hand in past semesters, the process required manual review of MOSS results and was a time consuming and error prone process.

Confidence about the validity of the reports was also significantly increased using this new approach, largely because multiple features of the system specifically address concerns commonly voiced by participants in our institution's academic integrity process. Placing student code in a simplified side-by-side comparison is one example, producing a useful visual aid for case reviewers. Automatically generating such comparisons saved many hours of manual labor. The increase in productivity allowed staff to investigate edge cases that might otherwise go unreported due to lack of time. Overall, the process was largely reduced to one that could be completed by a part-time graduate assistant, requiring significantly less input and direction from the primary instructor than in the past.

5 Conclusion

This paper describes our implementation and experience automating the process of producing plagiarism case reports. Unlike plagiarism detection tools, which focus primarily on software similarity measures, this work instead focuses on the additional steps needed as part of a complete workflow in reporting plagiarism incidents. This includes gathering supporting evidence, clustering students into groups rather than simply pairs, preparing side-by-side code comparisons intended for readability by a non-programming audience, and generating a summary report combining all these pieces together. The resulting case reports are suitable for submission to a

university's academic integrity process after an instructor reviews them to ensure the cases are valid and worth pursuing.

We chose MOSS as the plagiarism detection tool that feeds the first step of this process due to instructor familiarity, language support, and flexibility. However, as each module of our system is a self-contained unit, it would be straightforward to incorporate a different similarity detection tool if desired. Once suspicious student submissions are identified, a series of programs are run that cluster students, gather supporting evidence, process code for comprehension, and generate a final report. This significantly reduces the workload required, such that a single part-time graduate assistant can produce the reports, review them, and send them to the course instructor for final review. While we anticipate Grass will eventually eliminate nearly all human input except for final review, a major current limitation is the lack of ability to detect AI-generated code. Future work involves incorporating AI-flagging into the Grass workflow. The amount of evidence and presentation of information at that point should allow instructors to easily decide whether a case should be reported, instead of requiring prohibitive time manually gathering evidence and compiling a report.

Based on our experiences, this approach significantly decreased the amount of time required to examine and report plagiarism cases. For a class of more than 550 students, the generated reports allowed a single part-time graduate assistant managed the majority of the work regarding validation of reports. Compared to previous semesters, confidence regarding cases reported was higher. The side-by-side comparison is also noted as highly beneficial for its ability to more easily demonstrate to case reviewers why the students are suspected of plagiarism. Overall, we have found that this automated system to be of significant value, but it can still be improved to add more information and further reduce effort. Our implementation is open-source and available on GitHub [13].

5.1 Future Work

While this first implementation was a success, it also pointed to possible improvements. We seek to more accurately select students for consideration to help ensure that all clusters are found. Using a fixed cutoff is not responsive to overall class performance. Employing a dynamic partitioning method can more accurately determine which students should be suspected of plagiarism, based on their similarity performance against the class overall.

The supporting evidence module can gather many more different types of data. Of particular interest is the discussion by Tahaei and Noelle about examining patterns of resubmission [4]. Because Web-Cat stores all previous submissions, examining student work for this information is a straightforward process. Indeed, the graduate student tasked with managing plagiarism reports has commented that often a student's first submission is useful to examine. The first submission may have strong evidence like an incorrect `@author` tag that the final submission has corrected. There may also be benefits to examining how code similarity between students has changed across submissions, or by examining the contents of comments in all submissions. If code is managed using version control software, useful information may exist inside students' version history.

Finally, the module that processes student code for enhanced comprehension can be better

designed. While sorting code by function length is effective, it is not unusual for functions that serve the same purpose to have different lengths. In addition to length, code can be sorted by signature, name, or other factors. It may be useful to retain comments and present them separately, or analyze them for more potential evidence. While using `diff` has proven useful for our case, the GumTree program [15, 16] acts as a more powerful `diff`. Some superficial differences may be caught by GumTree, but it may also be necessary to further normalize source code to make comprehension more accessible.

On a more general level, the overall usefulness of this system needs more stringent evaluation. The time benefits of each module can be measured, and surveys of their perceived usefulness taken. By precisely determining the additional features which instructors find useful, we can streamline the report generation to include information instructors want the most. We look forward to making the source code plagiarism reporting process as straightforward, accurate, and easy as possible for instructors.

References

[1] I. Albluwi, "Plagiarism in programming assessments: A systematic review," *ACM Trans. Comput. Educ.*, vol. 20, no. 1, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371156

[2] A. Parker and J. Hamblen, "Computer algorithms for plagiarism detection," *IEEE Transactions on Education*, vol. 32, no. 2, pp. 94–99, 1989.

[3] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review," *ACM Trans. Comput. Educ.*, vol. 19, no. 3, May 2019. [Online]. Available: https://doi.org/10.1145/3313290

[4] N. Tahaei and D. C. Noelle, "Automated plagiarism detection for computer programming exercises based on patterns of resubmission," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ser. ICER '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 178–186. [Online]. Available: https://doi.org/10.1145/3230977.3231006

[5] A. Ahadi and L. Mathieson, "A comparison of three popular source code similarity tools for detecting student plagiarism," in *Proceedings of the Twenty-First Australasian Computing Education Conference*, ser. ACE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 112–117. [Online]. Available: https://doi.org/10.1145/3286960.3286974

[6] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism detection for java: A tool comparison," in *Computer Science Education Research Conference*, ser. CSERC '11. Heerlen, NLD: Open Universiteit, Heerlen, 2011, p. 33–46.

[7] O. Karnalim, Simon, and W. Chivers, "Similarity detection techniques for academic source code plagiarism and collusion: A review," in *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, 2019, pp. 1–8.

[8] L. Prechelt, G. Malpohl, and M. Philippsen, *JPlag: Finding plagiarisms among a set of programs*. Citeseer, 2000.

[9] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises," in *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, ser. Baltic Sea '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 141–142. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/1315803.1315831

[10] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," in *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 266–270. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/299649.299783

[11] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, 1999.

[12] 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*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 76–85. [Online]. Available: https://doi.org/10.1145/872757.872770

[13] J. A. Thompson, "Grass: Automatically generating reviewer-friendly plagiarism case reports," apr 2026, accessed: 2026-04-29. [Online]. Available: https://github.com/web-cat/grass

[14] S. Chishti, "moss.py," jul 2024, accessed: 2026-04-29. [Online]. Available: https://github.com/soachishti/moss.py

[15] A. Fujimoto, Y. Higo, J. Matsumoto, and S. Kusumoto, "Staged tree matching for detecting code move across files," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 396–400. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/3387904.3389289

[16] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982