Application of Source Code Plagiarism Detection and Grouping Techniques for Short Programs

Dylan Ryman
Department of Engineering Education
University of Cincinnati
Cincinnati, OH, USA
rymandn@mail.uc.edu

P.K. Imbrie
Department of Engineering Education
University of Cincinnati
Cincinnati, OH, USA
imbriepk@ucmail.uc.edu

Jeff Kastner
Department of Engineering Education
University of Cincinnati
Cincinnati, OH, USA
kastnejy@ucmail.uc.edu

***Abstract*—Academic misconduct in programming assignments, such as excessive collaboration with peers or use of online resources, is of growing concern to the integrity of engineering education. Software development skills have become increasingly relevant to many fields, and consequentially the number of students submitting software assignments has also increased dramatically. The online learning environment caused by current global conditions broadens opportunities for students to engage in unethical academic behavior. Current techniques for the identification of academic misconduct in software submissions suffer from multiple issues such as the production of an excessive amount of difficult to interpret data, and the use of algorithms with limited effectiveness on short program submissions, such as those containing fewer than fifty lines. This paper presents how several existing techniques for identifying software similarity are made more effective when combined and fine-tuned in a way that increases sensitivity and decreases noise for short source code submissions. This paper shows how similarity results can be applied in a robust framework for determining which submissions are similar enough to warrant investigation. Finally, this paper introduces a new technique for grouping similar submissions, which helps identify collections of student submissions that all contain matching features. This increases efficiency by reducing the number of submission pairs that require human analysis. The application of these techniques enables comprehensive analysis of similarity in short software submissions, reduction of noise through the use of robust methods for determining which submissions were likely involved in academic misconduct, and improvements in human review efficiency by grouping sets of similar submissions and reducing the total amount of data for review.**

***Index Terms*—plagiarism detection, source code plagiarism**

I. Introduction

Academic misconduct is of growing concern to the integrity of engineering education. The recent popularity of remote learning at academic institutions has reduced the effort required and increased the temptation for students to engage in academic dishonesty, however, the dire consequences of this behavior remain the same.

The use of programming assignments in engineering education is generally limited in comparison to computer science education. Exams given in introductory engineering courses often contain programming problems in combination with a variety of other problems that cover non-computational components of the curriculum. This results in the submission of many short programs for exams or other assignments that cover computational tools in the broader context of engineering education. These short submissions present additional challenges in the area of automatic plagiarism detection, as the chance that shorter and uncomplicated programs will be similar by chance is substantially higher than typical long program submissions in computer science coursework. Most existing work in software plagiarism detection has a focus on computer science education, so one aim of this paper is to alter these methods to be more appropriate for accurately detecting academic misconduct in the shorter and less complex submissions found in introductory engineering coursework.

The source code plagiarism detection algorithms in most common use at academic institutions today were, for the most part, developed over a decade ago and have undergone limited modification. Available computational power has increased substantially over this timeframe, yet most existing tools are unable to take full advantage of these technological advances. Another goal of this paper is to show how existing plagiarism detection algorithms can be modified to take greater advantage of modern computational power to help increase test sensitivity and decrease noise in results, especially when used on shorter program submissions where sensitive testing is essential to capture all possible data that could suggest misconduct.

Existing source code plagiarism detection systems typically produce a similarity score and require the user to determine a cutoff value for scores that constitute plagiarism. This can be a difficult and time-consuming process when dealing with a large data set. This paper introduces the use of a multivariate outlier detection framework that is robust to the true positives in the sample data set for automatically selecting the set of submissions that warrant human review, avoiding the problem of manually selecting a similarity threshold.

Finally, the wide sharing of resources used to engage in academic misconduct is especially prevalent in an online environment with large cohort size. Widespread resource sharing often occurs through a centralized online source such as the online tutoring website “Chegg” [1], but peer-to-peer sharing has also been observed to occasionally cause the same result. Wide sharing of source code used to engage in academic dishonesty results in a batch of highly similar submissions, creating a number of high file-to-file similarity detections that grow quadratically with the group size. Reviewing the

similarity of isolated submission pairs is time-consuming and ineffective in this case, so this paper introduces the use of a modern unsupervised graph clustering framework designed to detect communities of misconduct and present the data in an easy to consume format for the human reviewer.

II. Background

A. Source Code Similarity Measures

Several web services and software applications currently exist for evaluating similarity in source code files. MOSS (measure of software similarity) is the most broadly applicable and widely used service for determining similarity scores [2]. MOSS processes source code files by discarding easily modified information such as variable names and comments, then tokenizing the source code using a non-public language-specific method. Hashes of *k-grams*, or substrings of the tokenized data of length k, are computed at all indices in the document, then specific *k-grams* are selected in windows for use in comparison, providing a theoretical lower bound for sensitivity [3].

Even though MOSS is over a decade old, it is still in common use at academic institutions due to its broad programming language support and proven effectiveness [4]. Despite its wide use, MOSS is vulnerable to simple attacks, such as *Mossad*, and has limited support for configuring algorithmic parameters that are essential to detection performance [5]. MOSS is only available as a web service which raises student privacy concerns and is vulnerable to being inaccessible during peak usage times due to server load [6]. The MOSS source code is unavailable for analysis or local execution, and a request to the author for sources or binaries was denied.

Another popular plagiarism detection tool is JPlag, which uses a different approach for similarity detection [7]. Rather than comparing counts of similar *k-grams*, JPlag uses a string similarity algorithm known as greedy string tiling. This specific tool only supports a few programming languages, namely Java, Scheme, C, and C++, limiting its effectiveness in engineering coursework often focused on Python or MATLAB. Similar to MOSS, it is only accessible as a web service and has no available reference implementation. Plaggie is an open-source alternative to JPlag based on the same technology, eliminating privacy concerns. However, it is only functional on Java source code, highly limiting its applicability [8].

B. Determining Suspicious Similarity

Under the assumption that most students do not engage in academic misconduct, the vast majority of source code submission pairs constitute the expected similarity for programs that are not influenced by plagiarism. The problem of determining which submission pairs are similar enough to warrant investigation can be rephrased as an outlier detection problem, where the anomalies for detection are the source code files that come from the population of plagiarised submissions rather than non-plagiarised submissions. Minimal research into this technique has been conducted to the point. A previous paper suggested the use of interquartile range-based outlier detection to detect anomalous similarity, but this univariate method is imprecise and not robust to true positive cases of academic misconduct contaminating the sample [9].

C. Grouping Similar Submissions

Similarity grouping is a useful concept for dramatically reducing the time and effort required during the human review of detected anomalous similarities. For example, if five students all shared the same source code or used the same resource, every file would be similar to each of the other four files, creating twenty highly similar pairs for manual review. Detecting and presenting these communities to the reviewer can help alleviate this challenge.

Existing work has shown that an effective method to detect these communities is graph clustering, where each source code submission is represented as a node and weighted links are added between nodes with high similarity. A clustering algorithm is then applied to determine which submissions are members of the same community [10]. The utility of this approach is determined by the selected clustering algorithm and the effectiveness of several algorithms has been evaluated [11]. However, commonly used algorithms for this purpose, such as MajorClust, often produce suboptimal results for community detection [12]. Little research has been conducted on effective methods to review clustering output data.

III. Methods

This section will present methods developed for gauging similarity between student files, determining which students engage in academic misconduct, and grouping sets of similar submissions to facilitate efficient manual review.

A. Similarity Engine

MOSS's approach of *k-gram* matching was determined to be the most appropriate primary method for similarity detection. Additionally, all reviewed similarity detection systems simply discarded some information in the source code such as variable names and string literals. While it is trivial for students to change these components of the source code, not all students do this. When determining similarity for very short code submissions such as those often found on exams, it is critical to use all available information including variable names. While the incorporation of these easily changed components is not appropriate for the *k-gram* matching technique, they are preserved and analyzed separately to form a hybrid similarity model. No implementations of this technology are publicly available, so a new source code similarity engine was designed and implemented from the ground up. The following is a description of the processing pipeline of the new similarity engine.

1) Tokenization: The first step is to preprocess each source code to introduce whitespace and identifier insensitivity, remove easily modified source code components such as variable names and string literals, and reduce the hash search space by removing comments and other extraneous information. Initially, all commented lines are removed from the source

code, then all whitespace and newline characters are removed. Newline characters are used to identify the end of comments, so this step must be performed second. Next, each string literal is replaced with an identical placeholder token, causing the search to be insensitive to changes in string literals. In this phase, identifier names and string literals are stored along with the sanitized source code, as they are used in a different component of the similarity engine. Similarly, each numeric digit is replaced with the same digit placeholder token. Finally, a list of keywords specific to the language of the source code is referenced, and each keyword is replaced with a token unique to that keyword. This reduces the search space by substantially shortening the program length.

2) File Preprocessing: Computations such as hash calculations on source code are expensive and invariant to the comparison source code. Performance optimization is employed by computing this invariant data only one time and storing it with the source code to be later used by the comparison algorithms without recomputation. Data produced by the preprocessor includes sets of hashes of the preserved variable names and string literals as well as hashes of all *k-grams* in the source code. The *k-gram* generation is improved over MOSS in many ways. MOSS discards most hashes through the use of the Winnowing algorithm, and this was an appropriate decision for the computational power available at the time and MOSS's focus on detecting similarity in larger programming projects. It was determined that, for the short programming assignments target by the project, storing and comparing all *k-gram* hashes for all source code files is completely reasonable when appropriate design considerations are made. Additionally, instead of a single k-value being used for hash computations, three different programming language-defined k-values are used to compute three independent sets of hashes for each source code file. This effectively allows the *k-gram* similarity test to be run at multiple sensitivity levels for each submission pair, which is beneficial for weighting matches of longer *k-grams* more heavily while ensuring that matches of shorter *k-grams* that could still constitute plagiarism are accounted for. The combination of these two enhancements makes this *k-gram* generation method more resilient to the *Mossad* attack covered previously.

3) Sample Preprocessing: After the initial preprocessing step for individual files, a final preprocessing step is executed for the entire sample of files. This step computes holistic data about the sample using the data computed in the file preprocessing steps. The primary data computed in the reference implementation is a frequency table for each set of *k-grams* and the sets of identifiers and string literal hashes. These frequency tables are used to assign weights to individual *k-gram* or literal matches during the execution phase. This is an effective noise reduction technique because it lowers the importance of shared hashes between files proportionally to the number of total files that they are shared between. In addition to making the similarity detection more sensitive to particularly unique code snippets, it also removes the need for incorporating special handling for a template file or "starter code" into the detection engine. This is because all files will share the same starter code, and consequentially the frequency table will assign these shared hashes an importance level of zero, preventing them from affecting the similarity results.

4) Execution: Finally, after precomputed data for each source code file and holistic data about the sample has been prepared, each source code file must be compared with each other source code file. Particular care was taken when designing the implementation for this step as it has the potential for both memory and computing time performance problems. This is primarily because the complexity of this task grows quadratically, as every file must be individually compared to every other file, while the complexity of both of the preprocessing tasks grows linearly with the number of files. For each pair of student submissions, the intersection of each set of hashes is computed. Additionally, a modified Jaccard similarity coefficient is computed for each set of hashes where the cardinality of the intersection of the two sets is replaced by the sum of the precomputed importance factor linearly scaled to the interval $[0, 1)$ for each hash in the intersection. Given the first hash set, A, the second hash set, B, and the function F where $F(x)$ returns the scaled importance factor for some hash x, the similarity score s is computed using (1).

$$s = \frac{\sum_{x \in A \cap B} F(x)}{|A \cup B|} \tag{1}$$

Considering the high computational complexity of this large number of comparisons, the preprocessing steps were designed specifically to allow each pair to be processed completely independently in the execution step. Due to this independence, distributed computing techniques can be used to maintain high performance even with a large sample size. A generator component contained within the main process uses the data set to create a stream of pairs for analysis. Worker components, known as processing rails, request pairs from the generator for analysis. The reference implementation creates a number of workers within the context of the main process equal to the number of CPU cores on the host machine, however, it would be possible to implement workers over network requests with minimal effort, allowing the execution phase to scale using a compute cluster. To use minimal memory, the pairs are not precomputed, but rather computed in real-time by the generator as the processing rails request new jobs. Finally, each processing rail sends the results of the similarity test to a reducer component which resides in the main process. The reducer tracks the most interesting similarity results by discarding the lowest similarity pairs as new higher similarity pairs are received from the processing rails. In the reference implementation, the number of maintained results is equal to the initial sample size, or approximately the square root of the total number of pairs received. This step is necessary for maintaining memory efficiency, as the similarity results contain the complete intersection sets for each set of hashes in the pair. These full analysis results must be preserved for the files most likely to constitute cheating as these results are used

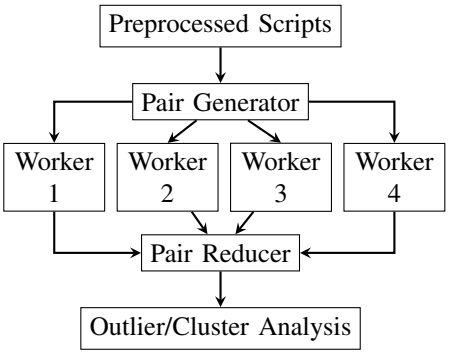

Fig. 1. Parallel processing pipeline.

for data presentation. The complete data processing pipeline is illustrated in Fig. 1. The similarity coefficients for all pairs are always maintained by the reducer, for reasons that will be later clarified in this paper, but the memory impact of this is trivial.

B. Similarity Outlier Detection

Appropriate determination of which pairs of submissions warrant human review is critically important for including all true positive cases while not including unnecessary data that slows down the review process. Existing similarity detection systems such as MOSS provide a similarity score or a similarity percentage. These measures are not particularly useful for determining which pairs should be reviewed because they are not relative to typical similarities in the population. This problem is made more severe when using these systems on short files, where the limited number of approaches to solving the problem causes more similarities between files by chance. An ideal criterion for selecting pairs for review is not a static similarity threshold, but rather similarity scores that are anomalous in the population of scores.

The similarity engine presented in this paper produces a multivariate similarity score for each pair of submissions. One numeric score is generated for each set of hashes. The current implementation produces five-dimensional data, however, further improvements that add new hash generation mechanisms could increase this dimension. Detecting anomalies in this data set, or the submission pairs that should be manually reviewed, is well suited to multivariate outlier detection.

An outlier detection framework based on robust Mahalanobis distance was selected. First, a covariance matrix is calculated. Ideally, multivariate outlier detection would be used on the entire dataset, however, calculating the covariance matrix for such a large dataset is impractically slow. To reduce the size of the dataset, the maximum similarity pair for each student is selected. The resulting outlier detection will determine students worthy of investigation, and the pairs to review can trivially be determined by selecting the pairs with the highest similarity for each student. The FAST-MCD algorithm is used to compute an approximate minimum covariance determinant for the data [13]. The inner fifty percent quantile of the data is used for MCD calculation to reduce MCD contamination caused by incorporating anomalous data into the calculation. The fifty percent quantile assumes that no more than twenty-five percent of students engage in academic misconduct. This assumption is reasonable for exams, and the quantile can be adjusted appropriately given the estimated percentage of students who cheat. Mahalanobis distances are computed, then MCD location and scatter estimates are used to perform an adaptive reweighting step used to determine an outlier rejection threshold [14]. Finally, outliers are determined by selecting adjusted data points greater than the 0.975 χ^2 quantile [15].

C. Similarity Grouping

In the previous step, suspicious files were identified using multivariate outlier detection. Now, communities of students who all plagiarised using the same shared resource will be identified. This is useful not only for analyzing student behavior but also for reducing the number of submission pairs that need to be reviewed. Rather than reviewing every file that a given file is highly similar to, which may be many if the file was shared extensively, the file will be identified as highly similar to all other files in the community. The number of high similarity pairs grows quadratically with community size, so this method has the potential to substantially reduce the review space.

Graph clustering, or the division of a graph into partitions, was determined to be the appropriate approach for community detection. Initially, a graph data structure is created and each student selected as an outlier is added to the graph as a node. Edges are then added between nodes representing a similarity detection. Edges are selected by filtering all of the similarity pairs to only those where both students in the pair were identified in the outlier detection phase. The remaining pairs are sorted from most similar to least similar, and the first m similarity pairs are added to the graph as edges connecting the two students in the pair, where m is three times the number of nodes in the graph. Edges are assigned a weight equal to the similarity score of the pair that they represent. Preliminary testing suggests that this number of edges produces graphs with a useful density, however, further research is needed to determine an optimal value for this constant, or to determine an improved method for edge selection.

Many graph clustering algorithms for community detection are available. Some, such as modularity clustering, are too slow to optimally solve on a graph of this size in a reasonable amount of time. Others, such as k-means clustering, require the number of clusters to be predetermined. This is impossible because the number of groups of students who engage in plagiarism by sharing resources is undetermined. Others still, such as correlation clustering, are only functional on unweighted graphs.

The selected graph clustering algorithm for this use case is Markov clustering [16]. It is an unsupervised clustering algorithm that is both fast and easy to implement. This algorithm determines clusters by simulating stochastic flow using random walks through the graph. This simulation is

TABLE I
MOSSAD ATTACK RESISTANCE

Original Code

```
int NchooseK(int n, int k){
  if(k==0)return 1;
  if(n==k)return 1;
  else
      return NchooseK(n-1, k-1) +
          NchooseK(n-1, k);
}
```

Code after *Mossad* attack

```
int NchooseK(int n, int k){
 if(k==0) return 1;
  int NchooseK(int n, int k);
   if(n==k)return 1;
    else
      return NchooseK(n-1, k-1) +
          NchooseK(n-1, k);
    int i = 0;
}
```

Short *K-Gram* Match: 1.8x expected (Score: 0.633, Typical: 0.351)
Medium *K-Gram* Match: 2.4x expected (Score: 0.806, Typical: 0.339)

efficiently carried out by iterative fast matrix operations until convergence. The Markov clustering algorithm has repeatedly proven to be a highly effective utility for community detection [17]. Many other clustering algorithms are available and future work could determine if MCL is optimal or if better options exist.

Finally, an interactive force-directed layout of the clustered graph is generated for manual review. This enables a human reviewer to easily visually identify communities of plagiarism, and the interactive layout enables the reviewer to rapidly view and compare files directly on the graph view. In addition to the visual layout, a standalone similarity report is generated for each cluster in the graph.

IV. RESULTS

A. Similarity Scores

The similarity engine is remarkably effective at assigning high similarity scores to files that are suspiciously similar while mitigating false positives. The novel frequency-based approach of hash weighting has been observed to significantly reduce the similarity scores of submissions similar by chance while simultaneously increasing the similarity scores of students who cheated. The methods described place greater importance on similar files with particularly unique characteristics, which are often the key indicators that cheating did occur.

In addition to being less sensitive to noise, the described approach of generating several sets of *k-grams* with different lengths increases the sensitivity of detection and introduces resilience to the *Mossad* attack. The authors of the *Mossad* attack shared a code snippet and an associated version modified using the *Mossad* procedure [4]. The modified version is not detected by MOSS as similar to the initial version. This paper shows resilience to the *Mossad* attack by testing the same code snippet and modification. First, several implementations of the binomial coefficient algorithm were produced independently by individuals with no knowledge of the *Mossad* authors' implementation. These additional data points represent the majority of students who do not cheat and are necessary to take full advantage of this paper's frequency-based approach. Table 1 shows the output of this paper's methods and the relative similarity scores to the cleanroom implementations. The files were easily selected as suspicious by the outlier detection component.

B. Identification of Cheating

The multivariate outlier detection approach used to identify students who likely engaged in academic misconduct appears to be highly effective in selecting true positive cases. Three programming homework assignments were selected from an introductory engineering course for use in evaluating the specificity of this paper's approach to the selection of students who likely engaged in academic misconduct. The typical solution length for all three of these assignments was under 30 lines of code. Both MOSS and this paper's method selected all pairs of submissions that were ultimately identified as sure true positives through manual review, however, MOSS typically returned two to three times as many positive results. Due to the short length of the submissions and the use of an absolute similarity measure, most of the unfiltered results returned by MOSS were false positives. Most of the results returned by this paper's method were true positives. In one specific instance, MOSS identified 254 pairs of similar files and this paper's method identified 28 suspicious students through 30 pairs of similar files. The exact number of students identified by MOSS in this example is unclear because MOSS does not present this information. Ultimately, 24 students were accused of cheating. All students accused of cheating were identified by this paper's method and no student identified by MOSS alone was accused. Each of the similarity pairs that MOSS produces must be reviewed manually as reviewers are provided with only a simple similarity percentage for each pair. An arbitrary value can be selected as a cutoff threshold, however, the many factors that affect average similarity make an appropriate selection challenging. This paper's approach avoids the issue entirely through the use of outlier detection rather than similarity percentages where a cutoff threshold is required. The outlier detection approach was successful at identifying all students ultimately accused of cheating while filtering out significantly more noise than MOSS in the tests performed, showing that this paper's method is significantly

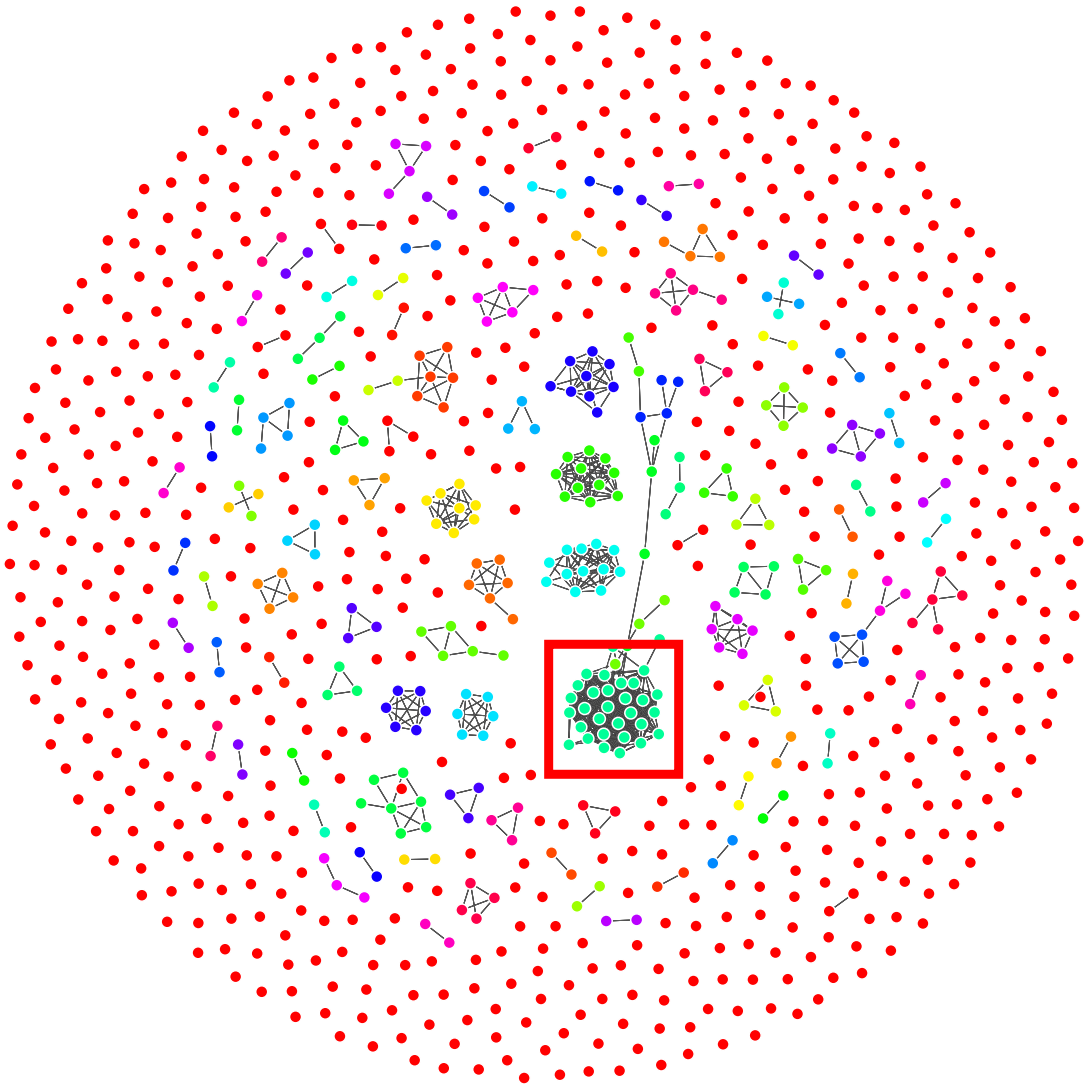

Fig. 2. Similarity clustering with highlighted region.

more accurate in identifying cases where enough evidence exists to establish cheating.

C. Clustering and Visualization

The similarity clustering method presented in this paper has proven to be highly effective in identifying communities of academic misconduct, and the related data visualization tools significantly accelerate the manual review process. Similarity results are divided into clusters using the methods discussed, then each cluster is assigned a color. The entire graph is presented in a web-based graphical user interface to the reviewer in a force-directed layout, where nodes are given their associated cluster color. A clustering was generated for a programming homework assignment taken from an introductory engineering course and the results are displayed in Fig. 2. The dense and highly connected clusters representing communities of academic misconduct are immediately evident in this view. Note that clusters in this visualization should be identified by color, as in rare cases edges from distinct clusters can intersect due to the nature of the force-directed layout. A particularly dense region of the graph is highlighted with a red box for the purpose of this paper's presentation. All of the submissions in this box were nearly identical and quickly identified as a solution from the online tutoring website "Chegg". All of these cases could then be immediately handled as a single group. Compared to the MOSS presentation of a single list of similar pairs, this method is significantly more efficient. In a list of pairs, each pair would need to be manually identified as belonging to this cluster, a different cluster, or potentially handled individually.

Additional enhancements were made to the graphical user interface to facilitate effective manual review. Hovering over any individual node displays the code submission associated with that node directly over the graph. Clicking on any edge downloads a Microsoft Word file containing the similarities between the two files to the user's computer. This portable file format is easily sharable and usable as evidence as it

does not require access to the visualization tool to present similarity. These visualization and data export features enable a streamlined review process and have proven to significantly increase review efficiency.

V. Conclusion

This paper has presented a novel framework for detecting source code similarity, selecting students who engage in academic misconduct, and identifying communities of academic misconduct with associated data visualizations. This framework has advantages over existing techniques and is more sensitive and specific than current approaches when used for short source code submissions. Future work will involve fine-tuning parameters used in similarity scoring, and evaluating additional different methods of outlier detection and graph clustering for use in this work.

References

[1] S. Manoharan and U. Speidel, "Contract Cheating in Computer Science: A Case Study," *2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pp. 91–98, Dec. 2020.

[2] A. Ahadi and L. Mathieson, "A Comparison of Three Popular Source code Similarity Tools for Detecting Student Plagiarism," *Proceedings of the Twenty-First Australasian Computing Education Conference*, pp. 112–117, Jan. 2019.

[3] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pp. 76–85, Jun. 2003.

[4] D. Sheahen and D. Joyner, "TAPS: A MOSS Extension for Detecting Software Plagiarism at Scale," *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*, pp. 285–288, Apr. 2016.

[5] B. Devore-McDonald and E. D. Berger, "Mossad: defeating software plagiarism detection," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, Nov. 2020.

[6] A. Aiken, *Plagiarism Detection*, 26-Apr-2021. [Online]. Available: http://theory.stanford.edu/ aiken/moss/.

[7] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *Journal of Universal Computer Science*, pp. 1016–1038, Mar. 2003.

[8] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises," *Proceedings of the 6th Baltic Sea conference on Computing education research Koli Calling 2006*, pp. 141–142, Feb. 2006.

[9] G. Cosma, "An approach to source-code plagiarism detection investigation using latent semantic analysis", unpublished.

[10] A. Ohmann, "Efficient Clustering-based Plagiarism Detection using IPPDC", unpublished.

[11] L. Moussiades, "Discovering Clusters of Plagiarism in Students' Source Codes," *Journal of Engineering Science and Technology Review*, vol. 9, no. 1, pp. 8–12, Feb. 2016.

[12] L. Moussiades and A. Vakali, "PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets," *The Computer Journal*, vol. 48, no. 6, pp. 651–661, Nov. 2005.

[13] P. J. Rousseeuw and K. V. Driessen, "A Fast Algorithm for the Minimum Covariance Determinant Estimator," *Technometrics*, vol. 41, no. 3, pp. 212–223, Aug. 1999.

[14] P. Filzmoser, "A multivariate outlier detection method", unpublished.

[15] P. Filzmoser, R. G. Garrett, and C. Reimann, "Multivariate outlier detection in exploration geochemistry," *Computers & Geosciences*, vol. 31, no. 5, pp. 579–587, Jun. 2005.

[16] S. M. Dongen, "Graph clustering by flow simulation", unpublished.

[17] K. Choi, J.-Y. Lee, Y. Kim, and D. Lee, "Comparison of graph clustering methods for analyzing the mathematical subject classification codes," *Communications for Statistical Applications and Methods*, vol. 27, no. 5, pp. 569–578, Sep. 2020.