Computer Science

Capstone Report - Fall 2022

Visualize Code Similarity using Graphs and Clustering Algorithms

Junyan Feng,
Momoe Nomoto,
Yuqian Sun

supervised by
Ratan Dey

Preface

This report is part of our coursework for the CS Senior Capstone at NYU Shanghai. We have been researching the topic of source code plagiarism detection and visualization for the fall semester of 2022. The purpose of this report is to summarize our work on the visualization of plagiarism detection results of two approaches: a traditional token-based algorithm and a novel machine learning based clustering algorithm. We created custom datasets to verify the correctness and reliability of our algorithms and visualizations. We target this paper towards academic instructors teaching computer science courses to assist them in detecting plagiarism in student assignments, projects, and/or exams and recognizing strains of plagiarism upon a cursory glance of the visualizations.

Acknowledgements

We thank Professor Ratan Dey and Professor Xianbin Gu for helping and supporting us during this capstone project by providing us with valuable feedback on our work.

Abstract

Source code plagiarism is a critical issue in the field of computer science, and various plagiarism detection techniques have been studied in an attempt to maximize accuracy and reliability. However, the presentation of how source code files are related to each other within a group setting and the differences in detection approaches have not been fully investigated. This project aims to portray the distributions and relationships of plagiarism behaviors within a dataset of source code files, as well as compare and contrast the visualizations of a traditional and a novel approach to code similarity detection2. We visualized the plagiarism detection results of the traditional model Moss and our custom model built using an Abstract Syntax Tree (AST) representation and DBSCAN clustering algorithm. While the clustering algorithm detects intended plagiarism, it suffers from false positives, i.e. incorrectly clustering nodes that did not contain traces of plagiarism. Moss, on the other hand, is able to produce a strong contrast between those that had plagiarism and those that did not.

Keywords

Source code plagiarism; Plagiarism detection; Machine learning; Data visualization; Computer science education

Contents

1 Introduction

The phenomenon of plagiarism is a widespread dilemma plaguing the academic field, especially in the computer science domain where plagiarism of source code is frequent due to the abundance of open source software available for use by any developer or user. This applies distinctly to a classroom setting where all students are assigned the same programming assignment and are required to write code in the same programming language. Within these submissions, there will inevitably be similarities among the source code, and as a result, plagiarism detection is challenging in order to avoid flagging coincidental similarities. To address this dilemma, an effective and sophisticated code similarity detection algorithm is of great importance.

There exists a myriad of tools with various algorithmic approaches introduced to measure code similarity, and in recent years, code similarity detection technology has gained increasing attention from researchers. These approaches can be classified into the following categories: metrics-based, text-based, token-based, tree-based, and graph-based [1]. Metrics-based was one of the earliest approaches deployed to detect plagiarism but is ineffective compared to newer approaches, whereas text-based approaches compare two string sequences which are effective in locating exact copies of code but not similar code with minor modifications [1]. A step up from text-based approaches is token-based where tokens are used to represent a program through normalizing textual differences and capturing the essential abstracted sequence [1]. Moss (Measure of Software Similarity) [2] is one such software that employs a token-based approach in a winnowing algorithm which is a local document fingerprinting algorithm that accurately identifies small partial copies within a large set of data [3]. Developed by Stanford University, Moss is a free software and web tool that has been around for over a decade, utilizing an advanced code plagiarism detection engine to effectively calculate similarity among programs of all languages uploaded to the system. Tree-based code similarity focuses on identifying similarities in structure between two programs, and specifically, Abstract Syntax Trees (ASTs) is a data structure that represents merely the structural details of a source code while discarding irrelevant details such as differences in semantics and lexicons [1]. Finally, graph-based algorithms not only capture structural details but also semantics of a source code, but most graph-based algorithms are NP-complete, victim to high computational complexity [1].

Novel methods have been introduced in the field of code similarity measurement research since then including informational theory, data mining, and approaches based on machine learning

methods [4]. In particular, unsupervised clustering and supervised neural networks are popular techniques, and researchers have also been combining various methods to tackle difficult problems in code detection. Xie et al. presented a method based on structural representation, specifically the AST structure, and unsupervised clustering to detect code similarity [4]. While researchers have been coming up with new approaches, the difference between these novel methods compared to a long-standing token-based approach by Moss has yet to be examined.

In this study, we will use a machine learning based clustering algorithm, and compare it to Moss, an existing plagiarism detection software. We create multiple dummy datasets with documented traces of plagiarism to varying degrees to simulate the phenomenon of source code plagiarism in a classroom setting. For each set of code files, we will first collect data on similarity measures amongst the submitted code from Moss and visualize the results in the form of a network graph using graph theory where each node is an instance of a source code file. Visualization of results can reveal insight into patterns and relationships by curating data into a form that allows one to easily discern information and draw conclusions. Next, we will conduct data cleaning and preprocessing on each of the dummy code submission datasets in order to apply a machine learning based unsupervised clustering algorithm to visualize code similarity. Clustering complements code similarity detection as an unsupervised machine learning technique, because objects are grouped into clusters such that objects in the same clusters are as similar as possible, and objects in different clusters are as dissimilar as possible. Lastly, we will compare the performance of the aforementioned approaches: visualization using graphs and clustering algorithms.

2 Related Work

2.1 Token-based Method

Moss is a software that employs a token-based approach in a winnowing algorithm which is a local document fingerprinting algorithm that accurately identifies small partial copies within a large set of data [3]. It is used extensively in programming classes for over a decade to effectively detect plagiarism among students [2]. This tool will serve as a benchmark of a simple but robust token-based code detection system in our research.

A token-based technique to represent source code is widely used in the field of plagiarism detection. To detect copies, many use the idea of k-grams. Ohmann et al. treat documents as data points and the distance between the data points is found by comparison of vectors of k-gram

frequencies [5]. Comparison is conducted using distance metrics, and Ohmann's approach (PIY) supports Euclidean, Manhattan, and Cosine, but the Cosine distance metric is primarily used [5], same as Xie et al.'s work. For each submission, PIY converts source code files into token strings, which are broken into k-grams and stored along with their occurrences in the document [5]. In comparison to Ohmann, Moss employs a winnowing approach as previously mentioned, which uses a pairwise comparison algorithm based on a k-gram fingerprinting string matching technique where hashing is used to select a subset of k-grams as the fingerprint for a document [5]. Even though Ohmann et al. compares their approach to Moss', analysis is conducted only based on accuracy measured by F1 scores, and histograms are provided to visualize the accuracy of different types of plagiarism. No analysis is carried out for the performance of these approaches for groups of source code submissions.

In Zhang et al.'s paper centering on information distance and clustering, source codes are tokenized as well, and similarities are calculated by solving Kolmogorov complexity. A clustering algorithm called Shared Near Neighbors is implemented [6]. PDE4Java is another token-based approach based on k-gram representation and DBSCAN, a density-based clustering algorithm[7]. PDetect also uses k-grams representation, but its clustering process is different and based on the idea of using clustering on a weighted graph where connectivity between two vertices in the same cluster is maximized [8]. As seen in the previous studies, k-grams representation is a popular technique while the approach to clustering algorithms varies in characteristics and application.

2.2 AST-based Method

Whereas previous studies use a token-based method to represent data, syntax structure-based trees like ASTs are popular as well. In addition to Xie et al's approach based on AST clustering which primarily utilizes AST conversion and extraction, cosine similarity calculation, K-Means algorithm, and the Elbow Method [4], there exists various other clustering-based approaches to plagiarism detection. An upgrade to Xie et al's approach is proposed by Hrkut et al. [9] who optimized the K-Means algorithm through parallelization, vector clustering speed up, and data size reduction. Hrkut et al. also created a model in the form of a syntax tree where the tree is transformed into characteristic vectors for normalization, and parallelizations in distance calculation, new cluster center calculation, and vector to cluster assignment are implemented in the K-means algorithm to cluster similar vectors more efficiently [9]. Several earlier research papers [10, 11] also document steps in converting code fragments into ASTs and detecting similar

code sequences by comparing matching subtrees or fingerprints. However, Jiang et al. has a similar approach to Hrkut et al's and introduces the computation of characteristic vectors in their algorithm, defined as a point $< c_1, ..., c_n >$ in the Euclidean space and each c_i represents the count of occurrences of a node in the subtree [11].

2.3 Other Methods

Acampora et al. pioneered an approach different from structure based and token based called the Fuzzy approach, which classifies data (i.e. matrix containing selected features of source code) into clusters and generates a structure containing a rule for each cluster. These rules are used in an ANFIS algorithm [12]. This method differs in complexity and efficiency but is still rooted in the idea of clustering. Various representation methods, similarity comparison, and clustering methods all have their own benefits and detriments. String/token-based approaches are more scalable and efficient, while tree and graph-based approaches are slow but distinguished at taking syntax and semantics into account during comparison.

2.4 Cleaning and Preprocessing

Aside from various algorithms applied to the topic in question, some researchers also take precautions of improving their results in another way. Oscar with his colleagues defines nineteen preprocessing steps that research studies could use to improve the performance of models. These precautions can be divided into the following two aspects: delete noisy elements and make replacements to reduce noises that could be wrongly recognized as representative elements [13]. In his study, these preprocessing steps convey evidence in improving performance for detection, and applying those combined steps in our study may lead to better and more accurate results.

2.5 Visualization of Similarity Graphs

Researchers also put effort into analyzing the results by developing a visualization tool for code similarity detection. Freire introduces two ways to present the relationship between instances of source codes: a graph using a minimum spanning tree (MST) and a histogram illustrating the distribution of distances from one submission to another based on similarity scores [14]. Freire's graph visualization is composed of two types: general graph and individual graph [14]. Whereas the general graph renders a subset of all edges determined by an adjustable threshold, the individual graph centers on a specific submission and displays only the edges and vertices

that are most similar to the center submission [14].

Research papers on similarity visualization for source code are scarce, but papers on the visualization of similarity graphs in other fields are more frequent. Rostami et al. document a new web-based visualization system called SIMG-VIZ for entity resolution and clustering [15]. Actions in SIMG-VIZ include drawing the graph, computing properties of vertices and edges, removing nodes, computing degree distribution, and determining graph statistics [15]. In Rostami et al.'s graphs, vertices of a cluster share the same color to indicate cluster membership and clusters indicate that similar nodes are at a closer distance to each other [15]. On the other hand, Freire's graphs use a few more measures to indicate similarity: edges are color-coded and width-coded to indicate distance, in addition to the submission ID labels given to graph vertices [14]. In the field of visualizing graph dynamics for network security, Liao et al. [16] introduce inter-graph clusters based on graph distance (Euclidean distance), and intra-graph clusters based on similarity levels pushed into edge weights.

2.6 Positioning Our Approach

Based on the readings, methods based on strings and tokens suggest difficulties in catching syntactic and semantic information. Therefore, we will use the tree-based method, AST, to tackle and transform the source code dataset. We expect better performance for certain plagiarism behaviors like adding redundant statements when representing source code files using an AST. On the other hand, Moss or other string/token-based algorithms may not be able to extract the most accurate footprint for analysis from those kinds of plagiarism behaviors. Since AST focuses on the structure of the source code, these noises can be handled more thoroughly. Apart from that, unsupervised clustering algorithms have also shown their strengths in classifying high vectors. Many researchers apply K-means clustering algorithms based on the k-gram method or AST method, but there is no current research on combining the AST representation method and the DBSCAN algorithm. K-means clustering is a centroid-based algorithm that focuses on dividing clear boundaries for every cluster, while DBSCAN is a density-based clustering algorithm, good at excluding outliers and finding clusters in those densest parts. We believe the performance of DBSCAN is more suitable in displaying possible plagiarizing behavior in code and disregarding the outliers. Therefore, in our research, we will choose a combination of the AST method with the DBSCAN clustering algorithm to address the problem.

Moreover, numerous papers introducing combined algorithms either didn't specify what kind of

data preprocessing process has been done or only revealed minimal basic measures. As discussed before, appropriate data preprocessing steps will improve performance. Thus, we plan to take a more thorough approach to data preprocessing in our research to minimize noises that may interfere with the process of generating the ASTs. As our sphere of study is concerned with specifically course work plagiarism, normalizing the data to reveal relative differences is essential.

A large number of existing papers analyze source code similarity and compare approaches using solely a numerical score that is the percentage of similarity. However, plagiarism is essentially a moral problem, and it is possible for pieces of code to be similar yet no violations of academic integrity, such as copying and cheating, have taken place. We will refer to this as unintentional plagiarism. Moss outputs a ranked list based on similarity scores when several programs are uploaded and compared. Solely looking at the similarity scores may not provide enough information to infer actual plagiarism amongst all students of a class. A single similarity score can only inform the user of the similarity of a pair of source code submissions, and numbers and lists cannot display closely-related groups of source code submissions. Visualizing the results of the algorithm through graphs can alleviate such problem by helping users identify groups of similar code and relieving the user of the need to manually traverse through the numbers or list to check for source codes that are not tainted by plagiarism.

Freire's graph visualizations highly resemble our idea of source code similarity visualization, but we aim to add interactivity to our graph by revealing detailed information when the user hovers over nodes and edges. Users will be able to tailor the graph by adjusting parameters to construct a more appealing visualization. In our research, we are looking to visualize the result of Moss' output using graph theory. To quantify the relationships in the similarity graph and detect anomalies, metrics of similarity and properties of the graph have to be defined clearly. Similarity between two nodes could be represented by distance, edge weights, vertex degree, or a combination of these metrics. Distance-based metrics can be further categorized into the type of distance calculation which includes Euclidean distance, cosine similarity, Manhattan distance, Jaccard distance, etc. However, distance-based similarity graphs may be futile as they mimic cluster graphs which will be explored in this project. We will also need to determine the vector representation of nodes, as randomly placing nodes in space will not be sufficient. Regardless, our visualization research will further the step of similarity detection in classrooms by not only comparing similarity scores but also exhibiting a clear holistic view of the group relationship through bridges, nodes, and connectivity concepts of graphs.

3 Solution

3.1 Construction of Test Datasets

We fabricated three datasets where source code files in each dataset contain the solutions to the same set of problems. Each set of data varies in length and the amount of code shared between all files. Dataset 1 is a collection of files related to singly linked lists where a large amount of code is identical in each file due to having the same node class and linked list methods. Each file in dataset 2 contains three functions related to dynamic programming and the amount of overlapping code is minimized. Dataset 3 contains files that are the shortest in length and each has three functions: 2sum, 3sum, and 4sum.

The modifications made to each of the datasets is documented to emulate the phenomenon of plagiarism in a classroom assignment. In each of the three datasets, there are 20 code files with strains of plagiarized code varying from groups of 2 to 5. Each group of code files is altered either minimally or significantly to simulate the varying degrees of plagiarism. The minor alterations include changes in variable names, changes in loop structures, and deletion or addition of comments. The significant changes include expansions of one-line codes to multiple lines, truncation of lines, change in the ordering of lines, and partial copying of functions to differing degrees. These dummy datasets contain variations of data that our program should expect and handle appropriately.

3.2 Visualization of Moss Output

Once we submit each set of data to the server, Moss returns the result as a URL which directs the user to a web page with similarity scores, lines matched, and copies of the submitted code files. We built a web application where the input is the Moss URL and the output is the network graph. In the Moss visualization web application, users can feed in their Moss URL to generate a graph illustrating the holistic view of the traces of plagiarism within the dataset. The web application also allows users to modify node or edge colors, node distance, spring length, and scale of edge width to provide a better view of the composition that can be tailored to datasets of all kinds. Figure 1 displays the user interface of the described web application.

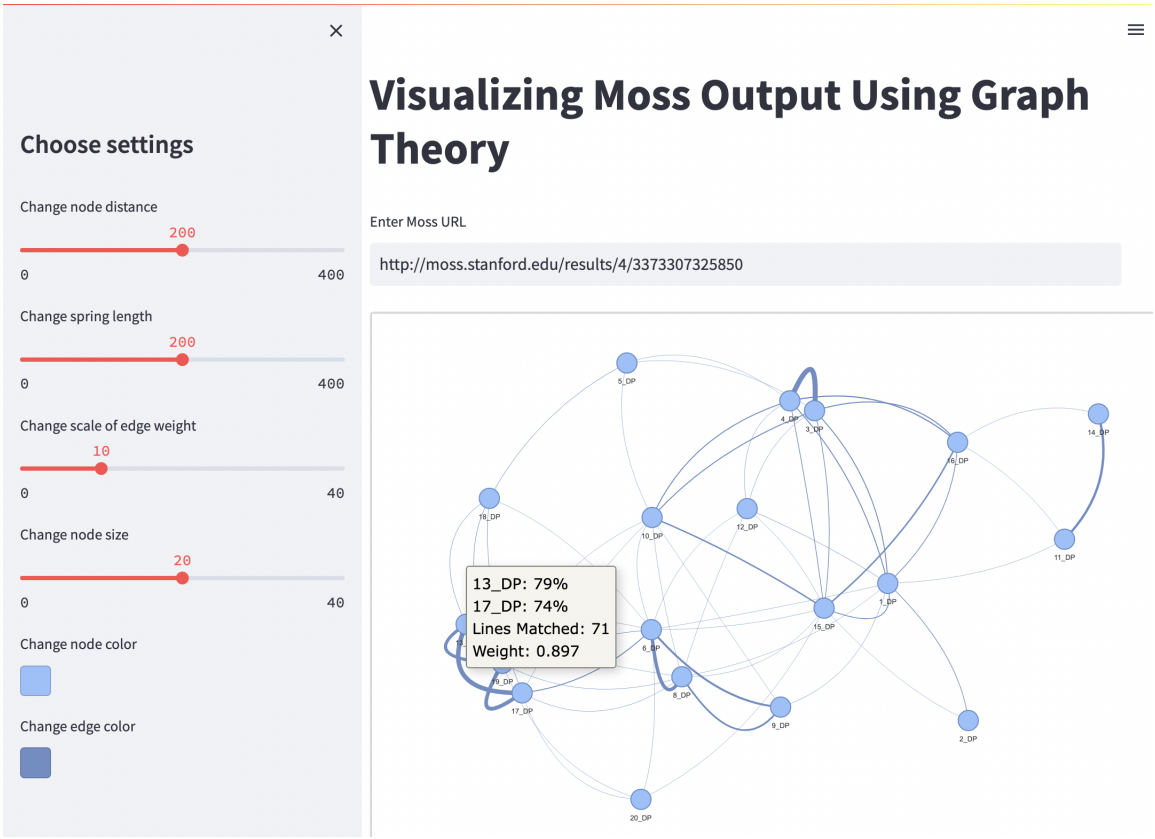


Figure 1: User interface of the Moss output visualization web application

Figure 2 outlines the processes that occur to generate a graph. Once the user enters a URL, data is extracted from the website using an HTML parser where appropriate elements in the document object model (DOM) are searched through web scraping and used as input to the parameters of the graph. To visualize Moss' output, Networkx [17], a Python package for the study of complex networks, is used to generate the graph with each node representing an instance of a code submission and each edge weighted to indicate similarity. The spring layout is chosen as it is based on a Fruchterman-Reingold force-directed algorithm where edges are treated as springs holding nodes close and nodes are treated as repelling objects [17]. With this layout, nodes that are connected with thicker edges are pulled closer together, resulting in an easier and quicker view of connections within a group. Edges that are bridges (i.e. deletion of such edges results in an increased count of connected components) are also clearly defined by extending the edge and secluding the nodes in question. The name of each node is extracted from the filename and the weight of an edge connecting two nodes is determined by the number of lines matched and normalized such that the weight ranges from 0 to 1 mapped from the least number of matched lines to the most number of matched lines in the current corpus. This measure is taken

to enlarge differences for a better interpretation of results. The weight of an edge is conveyed through its thickness. Thicker edges represent higher similarity between two nodes or instances of code submissions. In addition, pyvis [18], a Python library for visualizing networks, is applied to the Networkx graph to add interactivity. Users can drag nodes around without breaking the connections between nodes, and they can hover over nodes to see the name of the file and hover over edges to see the similarity score of the connected nodes, lines matched, and edge weight.

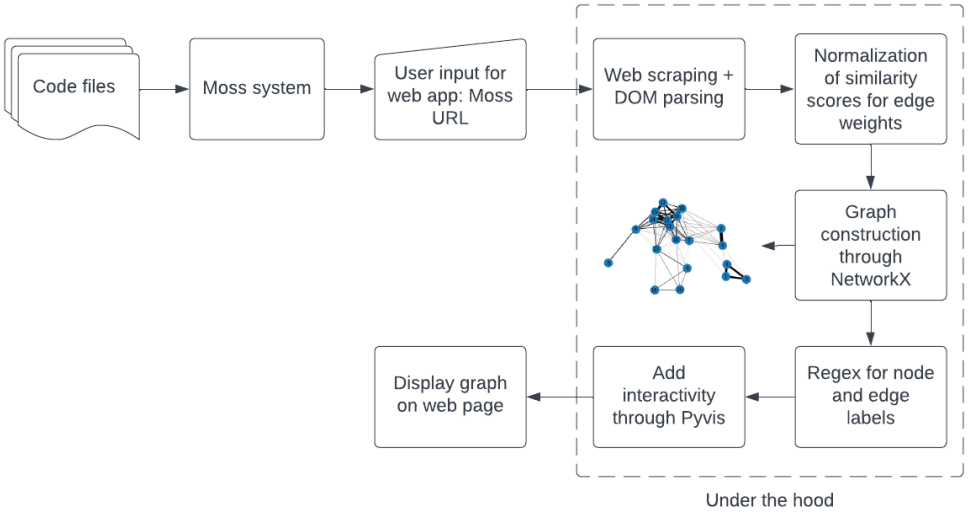


Figure 2: Flow chart of the web application

3.3 Data Preprocessing and Building the AST Model

This study uses ASTs to represent source code instead of a token-based method because ASTs can generate better program structure and lexical structure of source code. In the second phase of our project, we explored and devised our own model that combines the AST method with the DBSCAN clustering algorithm. However, prior to executing the data to our algorithm, data preprocessing is of vital importance to clean noisy data and other inconsistencies in exploratory data analysis. Figure 3 denotes the steps in our data preprocessing phase. We combined the data cleaning and AST generation processes, because in the course of AST generation, data cleaning is handled automatically. In this project, we are specifically dealing with code written in the Python language because many introductory programming courses are taught in Python. We used the Python package ast [19], which is a package used by the Python compiler itself to produce binary code from the AST built from tokens of the source code, and the interpreter runs the byte code to execute the program. Therefore, it is accurate in capturing Python's abstract

syntax grammar as the same module is used during code compilation. Then, we made some modifications to clarify the structure based on Karnalim's "Nineteen ways to deal with data preprocessing for code similarity problems" [13]. We left the basic operations including unifying letter cases, removing imports, comments, and identifies as is. Since the purpose is to examine relative differences in student submissions, we also removed the repetitive parts in the code to enlarge the differences. Figure 4 illustrates an example of how a simple piece of source code is fragmented to generate an AST. Each level of the tree represents code at a different indentation, and ASTs represent only semantically meaningful aspects while disregarding textual information such as punctuation.

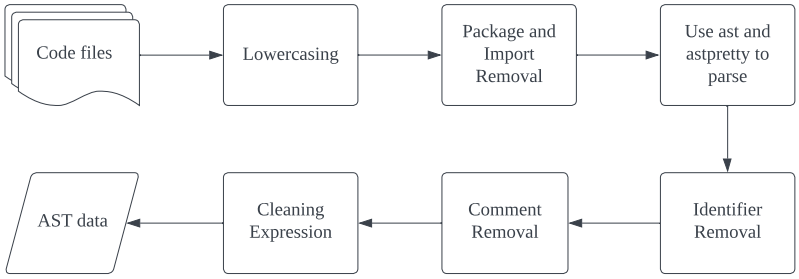


Figure 3: Data preprocessing steps

```
def t():
    a = 1
    b = 1
    if (a > b):
        print("Hello World")
```

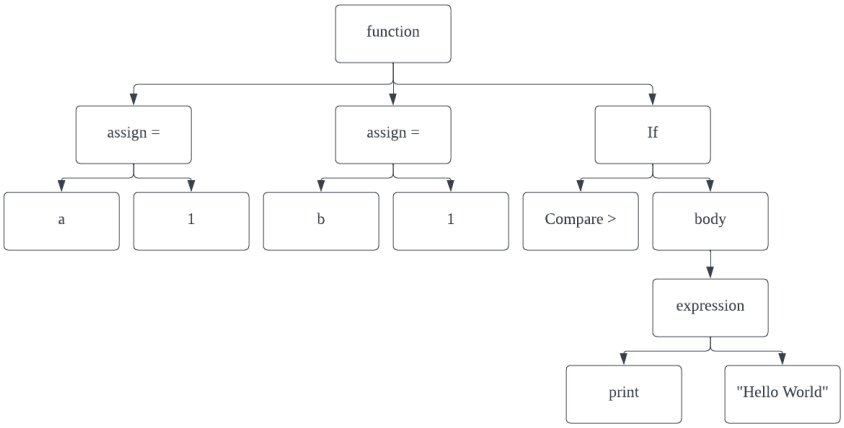


Figure 4: Example of AST generation from source code

3.4 AST Output Vectorization

Once the generation of AST was completed, we transformed the AST into a collection of parent and child node pairs as shown in Figure 5, which represent the semantics and structure of each code file. Then, to prepare the collection in a convenient form for processing and ease of comparison, we used a hash function to transform the data into a list of integers introduced in Zhao's paper [20]. Each integer represents the value of one dimension of a data point where the data points are in the form of a vector. We chose a FNV-1a hash function for the purpose of minimizing collisions and hashing time.

```
[(func, =), (func, =), (func, if), (=, a), (=, 1), (=, b), (=, 1), (If, Compare>),
(if, body), (body, express), (express, print), (express, "Hello World)]
```

Figure 5: Collection of parent child node pairs transformed from AST

3.5 Dimension Reduction and Clustering

PCA requires all the data points to have the same dimension. Since few papers mention how to unify the dimension of data, we attempted two simple methods: one is to find the minimum dimension among all the data points and delete the last few dimensions of data points of higher dimension, and the other is find the maximum dimension and pad the data points of lower dimension with trailing zeros.

Subsequently, we transformed the list of processed data points into a list of two-dimensional

data points using PCA and introduced the result of PCA into the DBSCAN algorithm. The output of DBSCAN demonstrates the clusters of closely related data points.

3.6 Characteristic Vectors

Inspired by Jiang et al.'s study [11], we also investigated another method instead of a hash function for node feature vectorization in order to reduce the dimension and improve performance. According to Jiang et al., characteristic vectors are defined as lists of predefined feature nodes critical for code structure representation. For every subtree of the AST, we have a count of the characteristic vector, which is the number of appearances of a node in the tree structure. Figure 6 illustrates an example of how the previous AST can be converted into characteristic vectors. Figure 7 displays the result of merging the parent-child node pairs with characteristic vectors. This measure is proven to be effective at improving performance. In our case, the characteristic vector is not a single characteristic vector, but a combination of two, which reserves some part of the order connection but at the same time extracts important vectors.

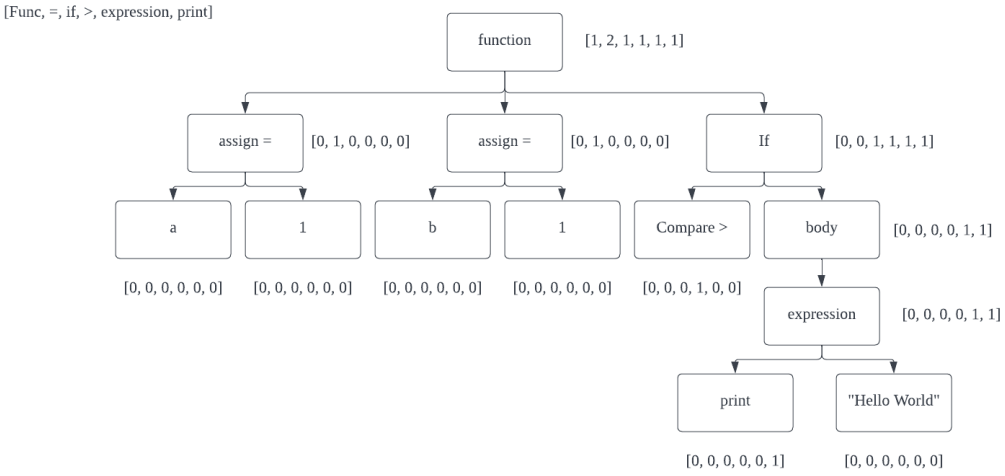


Figure 6: Example of characteristic vectors on AST

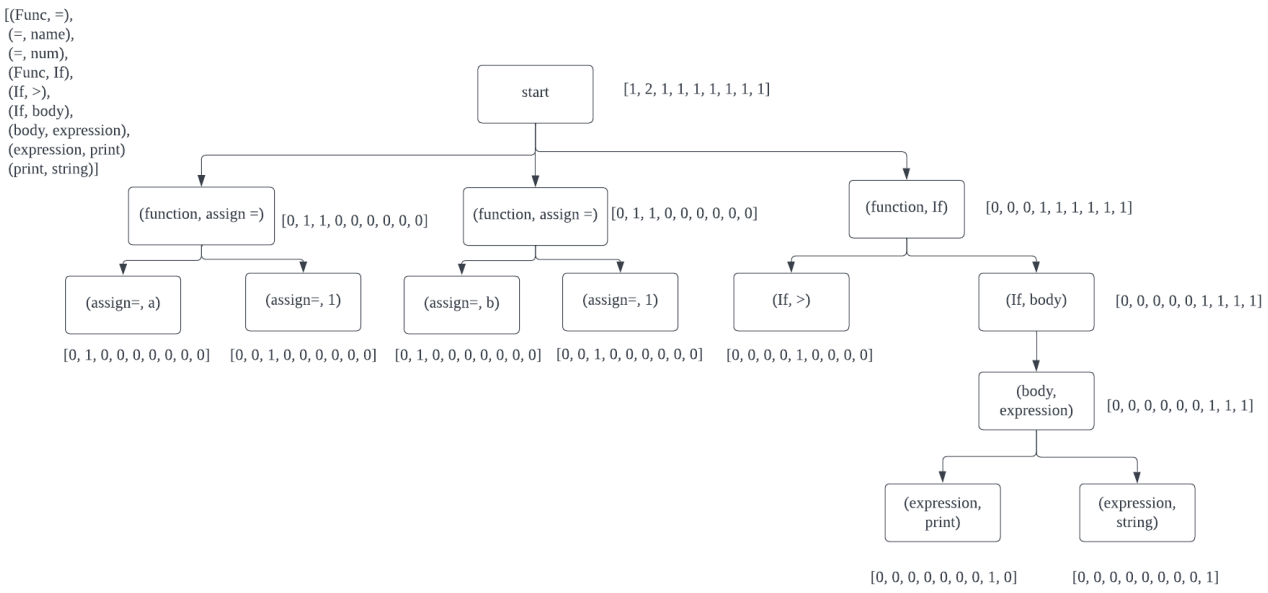


Figure 7: Example of characteristic vectors on combined structure

4 Results

To assess the correctness and validity of our methods, we created dummy datasets to check whether traces of intentional plagiarism during the stage of creation materialize in the Moss and DBSCAN visualizations. Table 1 presents the strains of plagiarizing behavior within each dataset, and these strains should match the nodes with the edges of the highest weights in the Moss visualization and nodes that are closest in distance in the DBSCAN visualization for maximum accuracy. Table 2 details the specific plagiarizing behaviors within the plagiarizing groups in each dataset. These two tables will serve as a basis for confirming the correctness of our work.

Intended Plagiarizing Groups in Each Dataset		
Dataset 1	Dataset 2	Dataset 3
1, 3, 5	3, 4	1, 4, 5
2, 4	6, 8, 9	7, 14, 16, 19
8, 9	11, 14	9, 11
11, 12, 14, 15, 18	13, 17, 19	12, 13
17, 19		

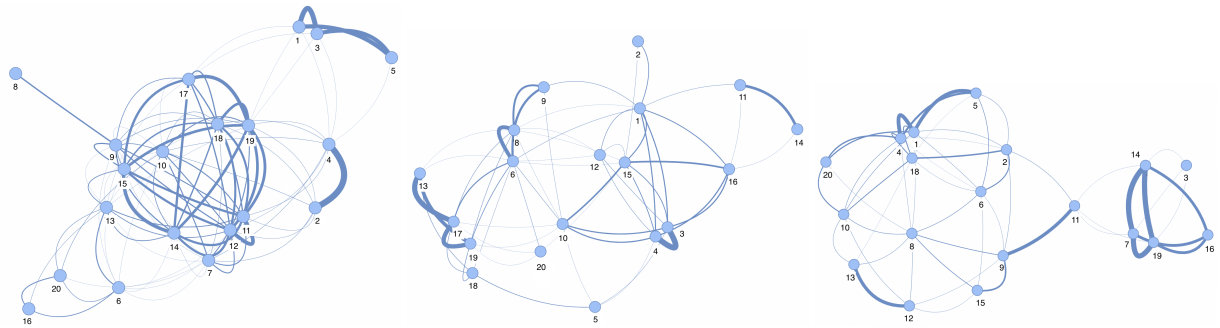
Table 1: Groups with intended plagiarizing behavior of different degrees in each dataset

Types of Plagiarizing Behavior in Each Dataset		
Dataset No.	Groups	Plagiarizing Behavior
1	1, 3, 5	Rename variables for 1, 3, 5 and change loop structure for 5
	2, 4	Rename variables and add comments
	8, 9	Expand one-liners and copy two functions
	11, 12, 14, 15, 18	Copy one or two functions entirely, rename variables, and change loop structure
	17, 19	Rename variables, add comments, and modify ordering
2	3, 4	Rename variables
	6, 8, 9	Copy two functions completely for 6 and 9, rename variables, and change loop structure for 8
	11, 14	Copy three functions completely, rename variables for others
	13, 17, 19	Copy four functions completely between 13 and 17, and copy two functions between 17 and 19
3	1, 4, 5	Rename variables and change loop structure
	7, 14, 16, 19	Rename variables for 14 and 19, change loop structure for 19, and change line ordering/length for 16
	9, 11	Copy two functions and rename variables
	12, 13	Copy one function entirely, rename variables, and change line ordering in another function

Table 2: Details of plagiarizing behavior within the groups in each dataset

4.1 Correctness of Graph for Moss Output

Figure 8 displays the corresponding network graphs for each dataset, where edge weights and node labels are adjusted for easier comparison. The graphs maintain correctness because the files at the top of the Moss web page with the most lines matched are the same files with most traces of intended plagiarism during dataset creation. Meanwhile, in the resulting graph, the edges with the highest weight are between those in the same said groups in Table 1. For instance, the group containing nodes 2 and 4 has heavy traces of plagiarism, and in Figure 8(a), the edge between nodes 2 and 4 is one of the thickest at 93 lines matched.



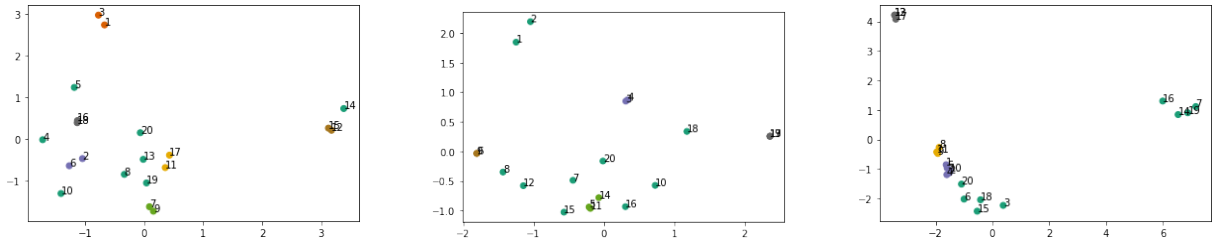
(a) (b) (c)

Figure 8: Network graphs for (a) Dataset 1 (b) Dataset 2 (c) Dataset 3

4.2 Correctness of Graph for PCA + DBSCAN Output

We implemented two methods to transform data into vectors that PCA can be applied towards: hash function and characteristic vectors.

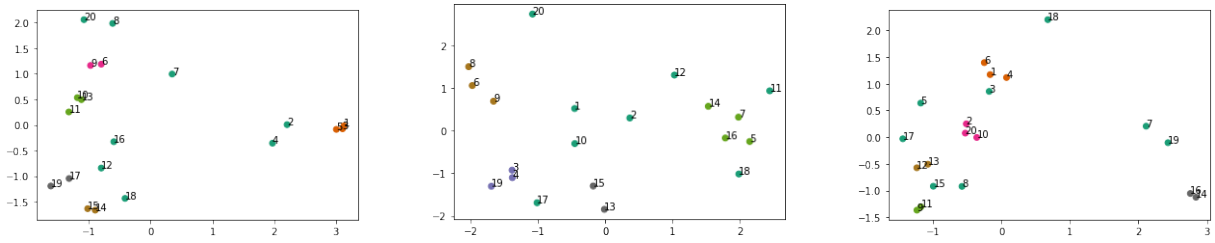
The first method we explored is the usage of a hash function. Figure 9 shows the result of PCA + DBSCAN with data transformed into vectors using hashed function. The dimensions of all the vectors are unified to the lowest dimension among all the dimensions. In the experiment using our own data set, the dimensions of ASTs are significantly distinguished from each other, which leads to high bias.



(a) (b) (c)

Figure 9: AST+hash+PCA+DBSCAN for (a) Dataset 1 (b) Dataset 2 (c) Dataset 3

Figure 10 shows the result of PCA + DBSCAN with data transformed into characteristic vectors. This result is significantly closer to the result of Moss than the result of the previous method. By transforming data to characteristic vectors, we gain a list of data (in the form of a vector) with the same dimension. Since there is no need to unify the dimension, no information of data is lost when we use PCA.



(a) (b) (c)

Figure 10: AST+hash+PCA+DBSCAN for (a) Dataset 1 (b) Dataset 2 (c) Dataset 3

5 Discussion

5.1 Methods of Data Transformation

For the first method, the DBSCAN visualization graph's correctness largely depends on how different the dimensions of the data points are. PCA requires all the data points (vectors) to have the same dimension. Thus, dimension unification is of utmost importance. The two methods we used either add incorrect information (zeros) to data points or delete necessary information from the data points, both of which influence the result of DBSCAN negatively.

Meanwhile, one related research paper mention using cosine similarity formula to calculate the similarity between codes (also in the form of a vector) [4]. However, this paper didn't mention the dimensionality problem. To use the cosine similarity, the data points still have to be unified to the same dimension. Thus, the dimensionality problem still remains to be dealt with.

For the second method, we were able to minimize the loss of data and made great progress in controlling information.

5.2 Detection of Plagiarism Behaviors in Datasets

Upon a cursory glance at visualizations of Moss output, hashing output, and characteristic vector (CV) output, along with Table 2 on how code files were modified, variable renaming was easily detectable in both the hashing output and CV output. An example is nodes 1 and 3 in dataset 1 which are extremely close together in all three methods. However, when loop structures are changed (such as changing a for loop to a while loop), hashing output did not capture the similarity entirely. An example is node 5 where loop structure is changed and variables are renamed in comparison to 1 and 3. Another example is between nodes 14 and 15 in the same dataset where only loop structures are changed without variable renaming. On the other hand, the CV method flawlessly captured the similarity between nodes 1, 3, and 5, evidenced in Figure 10. In addition, the CV output is less sensitive to modifications in comments compared to hashing output, that is, the addition or removal of comments did not influence the distance of nodes (e.g. nodes 2 and 4). There are groups where only some functions were copied, and both CV and hashing output retained some distance between these nodes, which is correct, but this is a method some students use to fly under the radar of code plagiarism detection systems. Both the CV output and hashing output incorrectly detected groups of files that had no intended plagiarism behaviors such as nodes 16 and 18 in hashing output and nodes 10 and 13 in CV output, which is false positives.

Meanwhile, in Moss output, there were no connections between nodes 16 and 18, and the edge between nodes 10 and 13 had a minimal weight at 8 matching lines with a maximum at 93 lines matched. Therefore, Moss has the upper hand in the metric of accuracy.

5.3 Comparing the Approaches

While Moss was able to detect all the plagiarism behaviors in the dummy dataset, the visualization could be convoluted when there are bigger groups of nodes or when datasets increase in size. Understanding the holistic view of the relationships between nodes could take more time than looking at a distance-based similarity graph. The edge weight based approach to visualizing Moss output may not be optimal since as the number of nodes increases, the graph becomes exponentially complex. Even though Moss provides very accurate results as expected, post processing in Moss is tedious. Moss does not provide results in a visual form, while clustering models take code files as input and output the results directly as visualizations. When Moss does not detect any lines matched between a code file and any other files, the node will not show up in the visualization, while in the clustering outputs, all nodes are present.

Meanwhile, the clustering method (Characteristic Vector + PCA + DBSCAN) can generate a two dimensional graph that visualizes the clusters of data points. With different colors representing different clusters, the graph is easier on the eyes and more direct than Moss visualization, especially when the data set is large. Therefore, a distance based similarity visualization system is preferred for bigger datasets.

However, the current problem with the CV output which we have determined to be more accurate than the hashing output is its inaccuracy with nodes that had no intentional plagiarism behavior, resulting in some false positives. It is due to the loss of semantic structure in CV so that the number of vector matches but the code is not the same.

Thus, CV method is not optimal as well. We need to hold better character vector selector, but maintain the structural and the key components at the same time.

6 Conclusion

While token based, hash based, and characteristic vector (CV) based source code plagiarism detection methods can all detect minor changes in data that do not influence the overall structure (such as variable renaming), the token based method Moss excels in detecting similarities even

when structure is changed. However, between hash based and CV based methods, CV has an upper hand in accurately clustering nodes with intentional plagiarism since there is no loss of information and dimension unification is not needed. Yet, both suffer from false positives which are inaccuracies in dispersing nodes that had no intentional plagiarism. Moss itself lacks a visualization tool and requires post processing to view the results holistically. An edge weight based visualization approach will also suffer from increased complexity when size of dataset increases. On the other hand, a clustering approach which is distance metric based is easier to discern, and the input is code files with the output being the visualizations directly. Further exploration into normalizing, reducing dimension, or a more sophisticated characteristic vector selector could be useful to allow for a more precise distance based graph. A more thorough data cleaning process such as the technique of compile, decompile, and recompile can also be explored for more precise results.

References

[1] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, p. 2464–2519, 2017.

[2] Moss, "A system for detecting software similarity." [Online]. Available: https://theory.stanford.edu/~aiken/moss/

[3] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting." [Online]. Available: http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

[4] Y. Xie, W. Zhou, H. Hu, Z. Lu, and M. Wu, "Code similarity detection technique based on ast unsupervised clustering method," in *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, 2020, pp. 1590–1595.

[5] T. Ohmann and I. Rahal, "Efficient clustering-based source code plagiarism detection using piy," *Knowledge and Information Systems*, vol. 43, no. 2, p. 445–472, 2014.

[6] L. Zhang, Y.-t. Zhuang, and Z.-m. Yuan, "A program plagiarism detection model based on information distance and clustering," in *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, 2007, pp. 431–436.

[7] A. Jadalla and A. Elnagar, "Pde4java: Plagiarism detection engine for java source code: a clustering approach," *IJBIDM*, vol. 3, pp. 121–135, 01 2008.

[8] 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, 06 2005. [Online]. Available: https://doi.org/10.1093/comjnl/bxh119

[9] P. Hrkút, M. Ďuračík, M. Mikušová, M. Callejas-Cuervo, and J. Zukowska, "Increasing k-means clustering algorithm effectivity for using in source code plagiarism detection," in *Smart Technologies, Systems and Applications*, F. R. Narváez, D. F. Vallejo, P. A. Morillo, and J. R. Proaño, Eds. Cham: Springer International Publishing, 2020, pp. 120–131.

[10] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 368–377.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[12] G. Acampora and G. Cosma, "A fuzzy-based approach to programming language independent source-code plagiarism detection," in *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2015, pp. 1–8.

[13] O. Karnalim, Simon, and W. Chivers, "Preprocessing for source code similarity detection in introductory programming," 11 2020.

[14] M. Freire, "Visualizing program similarity in the ac plagiarism detection system," in *Proceedings of the Working Conference on Advanced Visual Interfaces*, ser. AVI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 404–407. [Online]. Available: https://doi.org/10.1145/1385569.1385644

[15] M. Rostami, A. Saeedi, E. Peukert, and E. Rahm, "Interactive visualization of large similarity graphs and entity resolution clusters," in *EDBT*, 2018.

[16] Q. Liao, A. Striegel, and N. Chawla, "Visualizing graph dynamics and similarity for enterprise network security and management," in *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, ser. VizSec '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 34–45. [Online]. Available: https://doi.org/10.1145/1850795.1850799

[17] Networkx, "Network analysis in python." [Online]. Available: https://networkx.org/

[18] W. H. Institute, "A python library for visualizing networks." [Online]. Available: https://pyvis.readthedocs.io/en/latest/

[19] P. S. Foundation, "Abstract syntax trees." [Online]. Available: https://docs.python.org/3/library/ast.html

[20] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An ast-based code plagiarism detection algorithm," in *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, 2015, pp. 178–182.