Instructor-Centric Source Code Plagiarism Detection and Plagiarism Corpus

Jonathan Y. H. Poon
National University of Singapore
Computing 1,
13 Computing Drive
Singapore 117417
poonyanh@comp.nus.edu.sg

Kazunari Sugiyama
National University of Singapore
Computing 1,
13 Computing Drive
Singapore 117417
sugiyama@comp.nus.edu.sg

Yee Fan Tan [*]
KAI Square
33 Ubi Avenue 3,
#07-58 Vertex Tower A
Singapore 408868
yeefan@kaisquare.com

Min-Yen Kan
National University of Singapore
Computing 1,
13 Computing Drive
Singapore 117417
kanmy@comp.nus.edu.sg

ABSTRACT

Existing source code plagiarism systems focus on the problem of identifying plagiarism between pairs of submissions. The task of detection, while essential, is only a small part of managing plagiarism in an instructional setting. Holistic plagiarism detection and management requires coordination and sharing of assignment similarity – elevating plagiarism detection from pairwise similarity to cluster-based similarity; from a single assignment to a sequence of assignments in the same course, and even among instructors of different courses.

To address these shortcomings, we have developed Student Submissions Integrity Diagnosis (SSID), an open-source system that provides holistic plagiarism detection in an instructor-centric way. SSID's visuals show overviews of plagiarism clusters throughout all assignments in a course as well as highlighting most-similar submissions on any specific student. SSID supports plagiarism detection workflows; *e.g.*, allowing student assistants to flag suspicious assignments for later review and confirmation by an instructor with proper authority. Evidence is automatically entered into SSID's logs and shared among instructors.

We have additionally collected a source code plagiarism corpus, which we employ to identify and correct shortcomings of previous plagiarism detection engines and to optimize parameter tuning for SSID deployment. Since its deployment, SSID's workflow enhancements have made plagiarism detection in our faculty less tedious and more successful.

Categories and Subject Descriptors

K.3.2 [**Computer Science and Information Science Education**]: Computer science education; I.2.7 [**Natural Language Processing**]: Text analysis; H.5.2 [**User Interfaces**]: User-centered design

[*]This work was done while he was a doctoral student at the National University of Singapore.

General Terms

Algorithms, Experimentation, Languages, Performance

Keywords

Plagiarism assessment, Plagiarism detection, Programming, Similarity, User interface, Corpus studies

1. INTRODUCTION

Plagiarism is a serious issue in undergraduate courses involving programming assignments [20]. In 2004, 181 students at School of Computing at the National University of Singapore admitted to committing source code plagiarism [16]. In 2005, the Centre for Academic Integrity (CAI) reported that almost 40% of 50,000 students at more than 60 universities admitted in plagiarism [12].

Plagiarism detection and subsequent disciplinary actions are of paramount importance in education. If students involved in plagiarism are not identified and proper action is not taken, it unfair to their peers whom produced and submitted original works. Furthermore, the students involved learn less than their hardworking counterparts, tarnish the reputation of their own institutions, and reduce the value of their own degrees.

Detecting plagiarism, however, is a time-consuming and tedious process when performed manually. Inspecting n assignment submissions for plagiarism could require instructors to access all $n \times (n-1)/2$ pairs of submissions. Furthermore, the process becomes more difficult when students actively attempt to obfuscate the traces of plagiarism by modifying the plagiarized copies. Thus, according to [15], research on automated source code plagiarism detection was started in the mid-1970s [17, 4, 9, 2, 6].

The basic task in (source code) plagiarism detection, to decide whether one input is plagiarized from another input, has been studied extensively. However, a holistic plagiarism management system requires expanding the basic task to a larger scope: allowing instructors to zoom in on the suspicious code segments (rather than at the assignment level), support detection on a set of inputs (rather than single files; as code submissions often contain several files), support the input of the instructor's skeleton code (when the instructor provides some template code for students to start from). Furthermore, an instructor-centric system needs to provide visualization to see these suspected cases at a cluster level, across assignments and help fellow instructors pin down plagiarism across courses. All of these issues are addressed in our implemented Student Submissions Integrity Diagnosis (SSID), an open-source system that provides holistic plagiarism detection reported here.

```
import java.util.*;

public class A{
    public static void main(String[] args){
        for(int i=0; i<100; i++){
            System.out.println("Here");
        }
    }
}
```

(a) Original submission

```
import java.util.*;

public class A{
    public static void main(String[] args){
        for(int i=0+0+0; i<100; i++){
            System.out.println("Here");
        }
    }
}
```

(b) Plagiarised copy

Figure 1: Original and plagiarized code pair that goes undetected by MOSS, where the bolded text indicates modification. MOSS reports a similarity value of 97% when the original is submitted against itself (*i.e.*, (a) and (a) as the input pair), but 0% similarity when the (a) original submission and (b) plagiarized copy are input.

A key contribution of our work is our source code plagiarism corpus. To gain insight on how source code plagiarism happens, we asked volunteers to plagiarize sample input code. The subsequent analysis of the plagiarized examples allows us to classify attacks into two classes and identifies attack modes that current plagiarism systems do not handle. This corpus is also publicly available, which we hope will spur development in future plagiarism benchmarking and studies.

2. RELATED WORK

Plagiarism detection systems typically perform pairwise comparison of submissions to detect plagiarism and do so in two ways, using: (1) attribute-counting metrics, or (2) structure metrics.

In attribute-counting metric systems, code similarity is based on counts of particular entities. In Ottenstein's system [17], Halstead's software science metrics [10] are used with the number of unique operators and operands. To improve upon this, subsequent attribute-counting systems have added metrics for counting the number of loops [4], control statements [9], keywords [2], as well as measuring the average length of the procedure or function [6].

Structure metrics compute similarities based on code structure. An essential property in structure matching is that many spurious matches can involve small code fragments, thus the *Minimum Match Length* (MML) parameter is important to set correctly. As it has been shown that structure metrics produces better plagiarism detection results [21], many of the widely used plagiarism detection systems, such as MOSS (Measure Of Software Similarity) [1], the YAP (Yet Another Plague) family [22, 24], sim [8], and JPlag [18] are based on structure metrics. From our understanding, MOSS and JPlag are the current benchmark systems for code plagiarism detection [3]. However, even these systems can be easily confused; plagiarists can insert non-functional code to ensure that no plagiarised code segments are larger than the MML. An example of this problem is shown in Figure 1, in which a plagiarized copy goes undetected by MOSS.

The aforementioned systems deal with plagiarism at the pairwise level. In recent work, Freire [7] and Meyer [14] also developed a plagiarism detection system for programming assignments that allows pairwise comparison. However, as stated earlier, a real-world plagiarism detection system needs to perform n-way plagiarism detection. Two systems that are not publicly available – PDetect [15], an attribute-counting metric system, and PDE4Java [11], a structure metric system – provide the capability to detect plagiarism clusters.

In summary, currently available systems focus on the core problem of detecting plagiarism in a pairwise or clusterwise manner. But crucially, the current state-of-the-art misses in-depth knowledge about how plagiarism is actually carried out (from the plagiarists' view), as well as the knowledge of how the core problem affects an instructor's workflow (from the instructor's view).

3. PLAGIARISM DETECTION METHOD

In implementing SSID, we take the structure metric method but make changes to improve detection accuracy and efficiency. For clarity, we review the entire working of our core algorithm, which is largely in common with other structure metric systems, but highlights salient differences between SSID and other structure metric systems. SSID follows a three step pipeline to judge the plagiarism status of a set of submissions:

Tokenization is the first step. We note that tokenization is programming language specific, as the tokenizer needs to know the syntax and keywords associated with the inputs. Correct identification of keywords and syntax has been shown to be an important factor in any programming language specific support tool [19]. SSID is implemented for Java and C input. We use Java in our examples.

Our tokenizer currently ignores differences in whitespaces and comments, as format alteration is commonly employed in plagiarism [13]. We define four token types: constants, keywords, symbols, and variables. Our Java tokenization scheme merges both numbers and characters into a single *constant* category, as Java allows easy conversion of character comparisons to numeric ones and vice versa. We also remove string constants, as suggested by Wise *et al.* [24]. We also assign an integer hash value to each unique string instead of storing each keyword and symbol as a string, to facilitate fast string comparison. Also, as structure metrics can generate spurious matches, we differentiate between tokens used at the end or beginning of statements. As a result of the tokenization phase, the token N-grams that represent each submission are indexed into a hash table.

Pairwise, asymmetric similarity is computed per input submission pair. We adopt the *Greedy-String-Tiling* algorithm [23], but modify the *Minimum Match Length* (MML) termination criterion. We deem two statements identical if and only if their contiguous tokens are identical. We also make a small efficiency improvement by using hash value comparison, as suggested by Prechelt *et al.* [18] in their JPlag system. This step creates a hash table representation for each submission. Then, retrieving the indices of a N-gram is done in $O(1)$ time instead of $O(n)$ time, reducing the practical average submission comparison complexity to $O(c^2)$ from $O(c^3)$, where c is the number of tokens in a submission.

A crucial extension in SSID is the support of skeleton code. Instructors often provide students with skeleton code as a framework to start structuring their programming assignments. Such code, if not properly considered by the plagiarism system, will indicate plagiarism when in fact the suspicious code was initially provided by the instructor. We thus extend the basic algorithm to exclude skeleton code from comparison. SSID does this by adding a *mark* property to differentiate between the ordinary *match marks* that are determined by the *Greedy-String-Tiling* algorithm [23], and the *skeleton marks* that match both submissions and the instructor provided skeleton code. We consider a match as valid (to be flagged as part of plagiarism) if and only if the number of *match-marked* statements within a marked region satisfies the MML.

To further improve the efficiency of the algorithm, we leverage our tokenization scheme, which distinguishes tokens that form the beginning of statements. Instead of considering all N-grams, SSID only considers the ones where the first token is marked as the beginning of a statement. This significantly reduces the practical runtime by about 4 times and reduces spurious matching.

The algorithm finds the matched regions in the input submissions A and B in decreasing sizes of contiguous identical statements, until no more matches satisfying MML can be found. To provide the user with a clear notion of the similarity of two submissions, we define a similarity measure to reflect the percentage of tokens that are covered by matches. For example, if the code for submission A is entirely embedded within another submission B, the similarity from A to B is 100%, but is smaller in the opposing direction. This results in the asymmetric similarity measure $sim(A \rightarrow B) = \frac{\text{\# of match marked tokens in } A}{\text{\# of tokens in } A}$. We then define the symmetric similarity score $sim(A, B)$ as the maximum of the two asymmetric values $sim(A \rightarrow B)$ and $sim(B \rightarrow A)$.

Determining plagiarism clusters is the final step. SSID groups submissions that are all highly similar to each other. The clustering employed uses a similarity threshold to assign each submission into at most one group (plagiarism cluster) or as a singleton (original submission), in a deterministic, reproducible manner. These clustering requirements are fulfilled by *DBScan* [5].

Finally, we note that **the real-time performance** of SSID is also critical. Instructors need immediate feedback for incorporating plagiarism detection into their already busy workflows. On an ordinary modern generation 2.8 GHz Linux laptop, SSID evaluates datasets consisting of 80+ introductory programming assignment submissions, amounting to over 3600 submission pairs, in less than 4 seconds on average, tested over three runs. This is an adequately short delay for most instructors.

4. PLAGIARISM CORPUS

Prechelt *et al.* [18] noted that the success of plagiarists is proportional to their effort in attempting plagiarism. However, we hypothesize that plagiarists might find certain types of plagiarism easier to perform than others.

To better understand how plagiarism works in practice, we asked student volunteers to actively plagiarize submissions from two individual in-lab introductory level programming assignments. As the assignments are carefully monitored, we have a large set of true negative examples. The attempted modifications then constitute positive known examples of plagiarism, while the original submissions for the assignments constitute negative examples. To begin with, we picked four of the known original submissions from each assignment as samples for participants to perform plagiarism. The four samples were picked as they constituted different means to solve the problem and our asymmetric similarity metric indicated that these were substantially different.

28 student volunteers participated in our corpus collection work. They generated a plagiarized version of their given source code sample, under time pressure to simulate the limited time and effort that plagiarists commonly have in practice. In total, $28 \times 2 = 56$ positive and about 180 negative examples are given in our corpus. The corpus is also distributed along with the SSID system at the SSID website[1]. Using our symmetric similarity metric $sim(A, B)$, when the size of the N-grams used is set to 2, and MML is set to 2 statements, our system successfully differentiates between the plagiarized positives and original negatives with 100% accuracy. The parameter settings were set by using the standard method of cross validation, where different subsets of the corpus are rotated for determining optimal parameter settings and accuracy testing. Figure 2 shows the similarity distribution for the first assignment.

4.1 Attack Types

We analyzed the different types of plagiarism attempts (termed *attacks*) that participants attempted. For example, a detection failure (termed *confusion*) can occur when lines of code are inserted into the copied code segment, splitting the original code segment in two. As SSID is structure metric based, insertions can prevent SSID from identifying the copied code segments. In our description of each attack, we include a ratio in parenthesis that indicates the attack's effectiveness, as measured by the number of times the attack confused SSID (left number), over the total number of times the attack was observed (right number).

[1]http://wing.comp.nus.edu.sg/downloads/SSID/PlagiarismCorpus.html

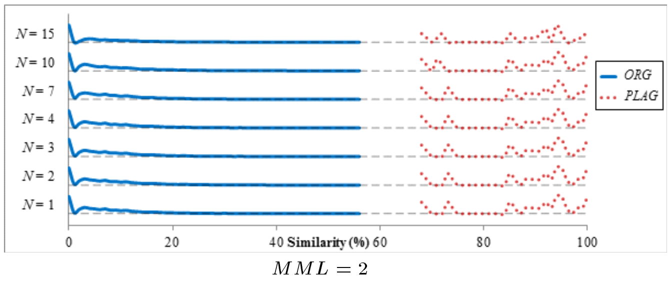


Figure 2: Similarity distributions for the first assignment in the corpus for various sized N-gram lengths where MML is set to 2. Note the perfect separation between original non-plagiarized submissions (*ORG*) and plagiarized ones (*PLAG*).

Note while individual attacks may confuse SSID, a single plagiarized assignment consists of many attacks, and perfect separation of positive and negative cases does not require perfect per-attack detection; indeed, it is a collective decision over all attacks present in a plagiarism attempt. We classify the observed attacks into three broad categories, as follows:

1. *Immutable attacks* do not modify the token sequences and are not effective against SSID. We first list them and then collectively discuss salient points about individual attacks. This was the largest category of observed attacks, amounting to 122 observed attacks, out of 246 in total or about 40%.

1) Insertion, modification or deletion of comments (0/35)
2) Indention, spacing or line breaks modifications (0/38)
3) Identifier renaming (0/41)
4) Constant modification (0/2)
5) Insertion, modification, or deletion of modifiers (0/6)
6) No change (0/0)

Structure metric systems are all immune to Attacks 1, 2, 3 and 4 as the conversion of the program into token types canonicalizes any renaming. For Attack 5, the Java tokenizer in SSID specifically ignores changes in access modifiers (*e.g.*, `public`, `protected`).

2. *Size dependent attacks* may confuse SSID, dependending on the size of the modified code segment. 64 instances were observed in our corpus. They are most effective in short programs, where the modification is proportionally large compared to the original source. But in large programs, they may be inefficient due to the amount of effort needed to perform the modifications.

7) Reordering of independent statements (6/10)
8) Reordering of methods (6/16)
9) Insertion or removal of parentheses (0/20)
10) Inlining or refactoring of code (13/18)

For Attacks 7 and 8, SSID correctly detected a portion of the attacks. When the original code segment were split into chunks smaller than MML = 2, SSID failed. For example, one attack reordered all of the three contiguous and independent statements within a block. Since the resultant code segment was a single state-

```
left = tree.getLeft();
right = tree.getRight();
```
(a) Original submission

```
right = tree.getRight();
left = tree.getLeft();
```
(b) Plagiarised copy

Figure 3: A reordering statement attack (Attack type 7) that is successfully detected by SSID.

```
if(myString.equals("right")) return 1;
if(myString.equals("left")) return 1;
return 0;
```
(a) Original submission

```
if(myString.equals("right")) ||
    myString.equals("left")) return 1;
return 0;
```
(b) Plagiarised copy

Figure 4: A refactoring attack (Attack type 10) that confused SSID.

ment in length, the code segments did not exceed the MML threshold and went undetected. Attacks were detected by SSID when the reordered statements consisted of two large blocks or were structurally identical. In Figure 3, the resulting token sequence is the same even after the statement reordering, and can be detected as a match.

Attack 9 modifies the syntactic scheme of the code, by adding or removing parentheses that are optional in Java. SSID's tokenization step automatically canonicalizes the Java syntax used in most cases, inserting missing parentheses to single-lined conditional statements and loops; in those cases the attacks failed.

Inlining and refactoring (#10) requires more effort. Six of the inlining attacks confused SSID as the inlined method length caused the original code segments to be less than MML in length, or were combined with other successful attacks like statement reordering. Seven refactoring attacks, by combining conditional statements together (as in Figure 4), succeeded.

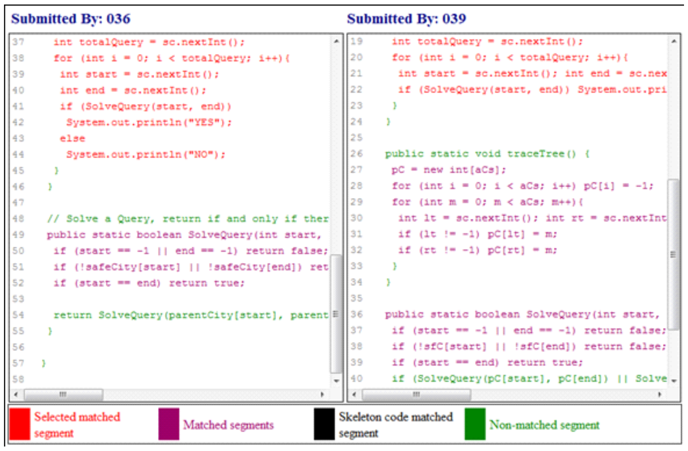
3. *Successful Attacks* confused SSID in all instances. They were observed in our corpus 60 times.

11) Redundancy (8/8)
12) Scope modification (7/7)
13) Modification of control structures (14/14)
14) Declaration of variables (10/10)
15) Modification of method parameters (1/1)
16) Modification of import statements (2/2)
17) Introduction of bug (1/1)
18) Modification of temporary variables in expressions (10/10)
19) Modification of mathematical operations and formulae (2/2)
20) Structural redesign of code (5/5)

We comment briefly on the more common attacks. Redundancy attacks (#11) remove or insert non-functional code. Scope modifications (#12) involve moving variable declarations to larger or smaller scope, which make it similar to size-dependent attacks. Control constructs were modified (#13) by changing looping variables, initialization and termination conditions, running loops in reverse. In conditional statements, the logic is changed by swapping the `if` and `else` blocks, or refactoring `if-else` compounds into separate `if` blocks. For variable declaration (#14), plagiarists combined several variable declarations into a single compound statement, or separated a compound statement into components. Finally, plagiarists expanded or compressed expressions, using or dropping temporary variables to store the return values from expressions, accessing them later when needed.

5. USER INTERFACE WORK FLOW

In addition to comparing pairwise similarity between submissions and detecting plagiarism clusters, SSID has a Web-based log system that records plagiarism activities for each student, as well as visuals to illustrate plagiarism clusters and other plagiarism-related information for gathering evidence on plagiarism.


Submitted By: 036

```
    int totalQuery = sc.nextInt();
    for (int i = 0; i < totalQuery; i++){
     int start = sc.nextInt();
     int end = sc.nextInt();
     if (SolveQuery(start, end))
      System.out.println("YES");
     else
      System.out.println("NO");
    }
   }

   // Solve a Query, return if and only if ther
   public static boolean SolveQuery(int start,
    if (start == -1 || end == -1) return false;
    if (!safeCity[start] || !safeCity[end]) ret
    if (start == end) return true;

    return SolveQuery(parentCity[start], parent
   }

  }
```

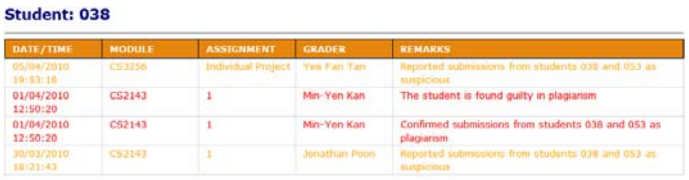
Submitted By: 039

```
    int totalQuery = sc.nextInt();
    for (int i = 0; i < totalQuery; i++){
     int start = sc.nextInt(); int end = sc.nex
     if (SolveQuery(start, end)) System.out.pri
    }
   }

   public static void traceTree() {
    pC = new int[aCs];
    for (int i = 0; i < aCs; i++) pC[i] = -1;
    for (int m = 0; m < aCs; m++){
     int lt = sc.nextInt(); int rt = sc.nextInt
     if (lt != -1) pC[lt] = m;
     if (rt != -1) pC[rt] = m;
    }
   }

   public static boolean SolveQuery(int start,
    if (start == -1 || end == -1) return false;
    if (!sfC[start] || !sfC[end]) return false;
    if (start == end) return true;
    if (SolveQuery(pC[start], pC[end]) || Solve
```

Figure 5: Snapshot of *Pairwise Comparison* interface. Codes in red and dark red denote the matched regions the instructor selected and matched regions in the displyed submissions, respectively. Code in green denotes no identical region is found in the other submission.


Student: 038

DATE/TIME	MODULE	ASSIGNMENT	GRADER	REMARKS
05/04/2010 19:53:18	CS3256	Individual Project	Yee Fan Tan	Reported submissions from students 038 and 053 as suspicious
01/04/2010 12:50:20	CS2143	1	Min-Yen Kan	The student is found guilty in plagiarism
01/04/2010 12:50:20	CS2143	1	Min-Yen Kan	Confirmed submissions from students 038 and 053 as plagiarism
30/03/2010 18:21:43	CS2143	1	Jonathan Poon	Reported submissions from students 038 and 053 as suspicious

Figure 6: Snapshot of log system interface.

(1) Pairwise Comparison Interface. As described in Section 3, we employ the structure metric method in our SSID. An advantage of the structure metric method is that the comparison results can be easily displayed. While user interface of MOSS uses the small number of colors for matched code regions, our interface uses several colors to display the comparison between assignments. Figure 5 shows a partial snapshot of the *Pairwise Comparison* interface. The interface allows the user to have a quick overview on the code segments found identical in both submissions.

(2) Log System. One of the major differences between our SSID and other similar systems is that SSID has a log system. Every report of suspicious pairs, confirmation of plagiarism cases and results of investigation (if the student is found guilty or innocent) is recorded in the system.

In Figure 6, student '038' is under investigation by an instructor from the course CS3256 for plagiarizing the submission from student '053.' '038' has denied committing plagiarism and reasoned that both their submissions were similar because they solved the assignment together. Through the records on '038' in the log system, the instructor can see that '038' has been found guilty of plagiarizing another submission from '053' in another course CS2143. Therefore, the instructor can conclude that student '038' has probably committed plagiarism in CS3256 as well.

(3) Detecting Plagiarism Clusters. Once a group that conducts plagiarisms is defined, the group can be displayed in the *Cluster Summary* interface. For example, Figure 7 shows the (partial) *Clus-*

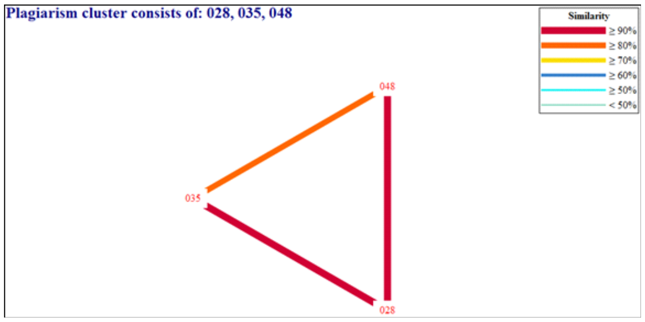
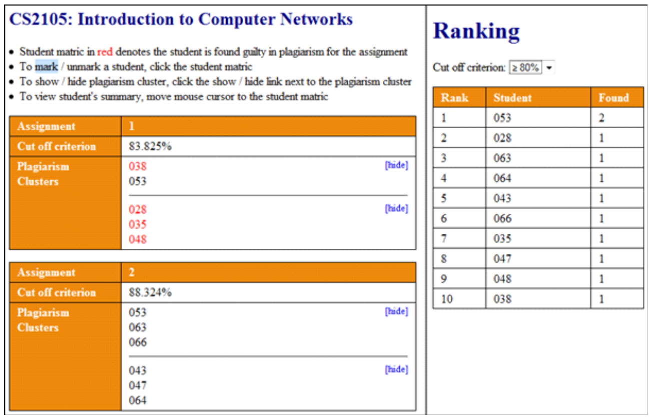


Figure 7: Snapshot of *Cluster Summary* interface (partial). Vertices and edges denote students' ids and the maximum similarity between two students' submission, respectively.



CS2105: Introduction to Computer Networks

- Student matric in red denotes the student is found guilty in plagiarism for the assignment
- To mark / unmark a student, click the student matric
- To show / hide plagiarism cluster, click the show / hide link next to the plagiarism cluster
- To view student's summary, move mouse cursor to the student matric

Assignment	1
Cut off criterion	83.825%
Plagiarism Clusters	038 053 [hide]
	028 035 048 [hide]

Assignment	2
Cut off criterion	88.324%
Plagiarism Clusters	053 063 066 [hide]
	043 047 064 [hide]

Ranking

Cut off criterion: ≥ 80%

Rank	Student	Found
1	053	2
2	028	1
3	063	1
4	064	1
5	043	1
6	066	1
7	035	1
8	047	1
9	048	1
10	038	1

Figure 8: Snapshot of *Assignments* interface. The left frame shows the overview of each plagiarism clusters in assignments. The right frame show the statistics of the top 10 students being detected in a plagiarism cluster.

ter Summary interface which consists of students '028,' '035,' and '048.' Each vertex and each edge in the interface represent a student id and the maximum similarity value of the submission pair between two students, respectively. In addition, edges are represented with different colors and thickness. Based on the evidence given in this interface, the instructor may consider forming a learning group consisting of the students '028,' '035,' and '048' to enhance their learning experience. Since the edges connected to '028' are thicker than the one connecting '035' and '048,' it is highly possible that '035' and '048' have both copied their assignment from that of '028.'

(4) Finding Plagiarism Activities. After checking a number of assignments through SSID, an instructor wants to know the statistics on the students who often conduct plagiarism. This can be done through *Assignments* interface as shown in Figure 8. This interface provides a summary of plagiarism clusters for each assignment (left in Figure 8). Furthermore, the interface also provides a list of the top 10 students in plagiarism clusters, which helps the instructor in monitoring plagiarism activity (right in Figure 8).

(5) Similarity between Students. According to Figure 8, student '038' and '053' belong to the same group. An instructor also wants to verify whether '038' belong to another plagiarism cluster where student '053,' '063,' and '066' belong. To address this issue, our SSID provides the *Between Individual* interface (see Figure 9). After selecting the target group of students, this interface displays the similarity between their submission pairs.

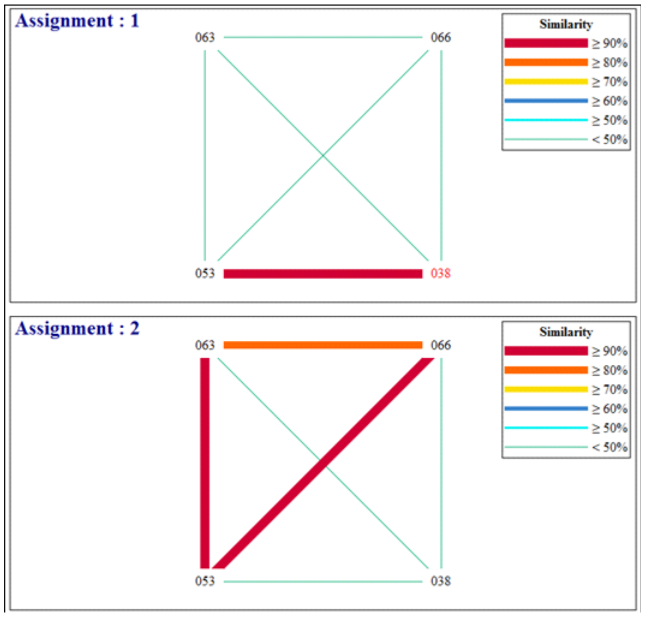
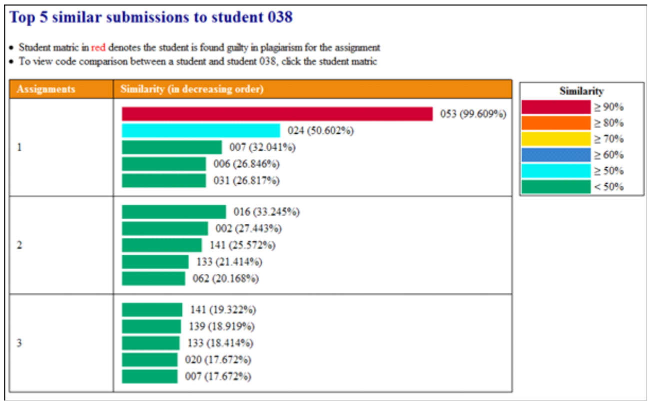


Figure 9: Example of *Between Individual* interface.



Top 5 similar submissions to student 038

- Student matric in red denotes the student is found guilty in plagiarism for the assignment
- To view code comparison between a student and student 038, click the student matric

Assignments	Similarity (in decreasing order)
1	053 (99.609%) 024 (50.602%) 007 (32.041%) 006 (26.846%) 031 (26.817%)
2	016 (33.245%) 002 (27.443%) 141 (25.572%) 133 (21.414%) 062 (20.168%)
3	141 (19.322%) 139 (18.919%) 133 (18.414%) 020 (17.672%) 007 (17.672%)

Similarity	
🟥	≥ 90%
🟧	≥ 80%
🟨	≥ 70%
🟦	≥ 60%
🟦	≥ 50%
🟩	< 50%

Figure 10: Snapshot of *Top Similarities* interface. This interface displays the top 5 similar submissions in each assignment for student '038.'

According to Figure 9, the similarities between the submission from '038,' and the submission from '063' and '066' are below 50% for both Assignments 1 and 2 (upper and lower in Figure 9, respectively). While '038' has similarity of 90% with '053' in Assignment 1, the similarity between the submissions from these students dropped to less than 50% in Assignment 2. This serves as a strong evidence to support the fact that '038' has probably stopped plagiarizing the work of others.

(6) Finding the Submissions Most Similar to a Student's Submission. SSID provides the *Top Similarities* interface (see Figure 10) to inform the instructor whether a suspicious student who may perform plagiarism has stopped plagiarizing other students' assignments.

Suppose that our target student is '038' as shown in Figure 10. After an instructor selects the target student and entering the number of candidates (K) to display, the K students are paired up with the target student '038' based on similarities between submissions of '038' and those of others. SSID sets the default value of K to 5. In Figure 10, an instructor has chosen to display the top 5 similar submissions in each assignment for student '038.' Since there are no similarities greater than 50% in Assignments 2 and 3, the instructor can conclude that '038' has been working on his own since his submission of Assignment 2.

6. CONCLUSION

We have presented a plagiarism detection system that assists instructors with the tedious task of detecting and reporting source code plagiarism cases. We extended the core task of detecting plagiarism between a pair of students, to a clustering framework, in order to support detection of plagiarism done by groups. To support instructors, SSID extends the visualization of assignment similarity to a group level and offers database views of code similarities across assignments and courses. Currently, our SSID only accepts Java programs as inputs. However, the system can be easily adapted by comparing other program languages if a tokenizer for the target language is introduced.

To better understand code plagiarism, we asked student volunteers to plagiarize input code samples in the Java programming language to build a corpus. On our corpus, our SSID is able to distinguish between original and plagiarized submissions with 100% accuracy. Our analysis of the resultant corpus identified two major categories of attacks: immutable attacks and size dependent attacks. However, we also identified 10 other modes of attacks that are not properly identified by current detection systems (including SSID), which serve as an agenda for future work in source code plagiarism detection.

7. REFERENCES

[1] A. Aiken. Moss: A System for Detecting Software Plagiarism. 1994. http://theory.stanford.edu/˜aiken/moss/.

[2] H. L. Berghel and D. L. Sallach. Measurements of Program Similarity in Identical Task Environments. *ACM SIGPLAN Notices*, 19(8):65–76, 1984.

[3] V. Ciesielski, N. Wu, and S. Tahaghoghi. Evolving Similarity Functions for Code Plagiarism Detection. In *Proc. of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO'08)*, pages 1453–1460, 2008.

[4] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A Plagiarism Detection System. In *Proc. of the 12th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '81)*, pages 21–25, 1981.

[5] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231, 1996.

[6] J. A. Faidhi and S. K. Robinson. An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment. *Computers & Education*, 11(1):11–19, 1987.

[7] M. Freire. Visualizing Program Similarity in the AC Plagiarism Detection System. In *Proc. of the Working Conference on Advanced Visual Interfaces (AVI'08)*, pages 404–407, 2008.

[8] D. Gitchell and N.Tran. Sim: A Utility for Detecting Similarity in Computer Programs. In *Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99)*, pages 266–270, 1999.

[9] S. Grier. A Tool That Detects Plagiarism in Pascal Programs. In *Proc. of the 12th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '81)*, pages 15–20, 1981.

[10] M. H. Halstead. *Elements of Software Science*. Elsevier Science Ltd., 1977.

[11] A. Jadalla and A. Elnagar. PDE4Java: Plagiarism Detection Engine for Java Source Code: a Clustering Approach. *International Journal of Business Intelligence and Data Mining*, 3(2):121–135, 2008.

[12] C. L. Jocoy and D. DiBiase. Plagiarism by Adult Learners Online: A case study in detection and remediation. *The International Review of Research in Open and Distance Learning*, 7(1), 2006.

[13] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pages 872–881, 2006.

[14] C. Meyer, C. Heeren, E. Shaffer, and J. Tedesco. CoMoTo – the Collaboration Modeling Toolkit. In *Proc. of the 16th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'11)*, pages 143–147, 2011.

[15] L. Moussiades and A. Vakali. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. *The Computer Journal*, 48(6):651–661, 2005.

[16] W. T. Ooi and T. C. Tan. A Survey on Awareness and Attitudes towards Plagiarism among Computer Science Freshmen. *CDTLink*, 9(3), 2005.

[17] K. J. Ottenstein. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1976.

[18] L. Prechelt, G. Malphol, and M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[19] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 76–85, 2003.

[20] P. Vamplew and J. Dermoudy. An Anti-Plagiarism Editor for Software Development Courses. In *Proc. of the 7th Australasian Conference on Computing Education (ACE'05)*, pages 83–90, 2005.

[21] K. L. Verco and M. J. Wise. Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-counting Systems. In *Proc. of the 1st Australasian Conference on Computer Science Education (ACSE '96)*, pages 81–88, 1996.

[22] M. J. Wise. Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. In *Proc. of the 23rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '92)*, pages 268–271, 1992.

[23] M. J. Wise. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. 1993. ftp://ftp.cs.su.oz.au/michaelw/doc/RKR_GST.ps.

[24] M. J. Wise. YAP3: Improved Detection of Similarities in Computer Program and Other Texts. In *Proc. of the 27th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '96)*, pages 130–134, 1996.