B. Comp. Dissertation

SSID: A User-Centric Plagiarism Checking System

By

Poon Yan Horn Jonathan

Department of Computer Science

School of Computing

National University of Singapore

2009/2010

B. Comp. Dissertation

SSID: A User-Centric Plagiarism Checking System

By

Poon Yan Horn Jonathan

Department of Computer Science

School of Computing

National University of Singapore

2009/2010

Project No: H079590
Advisor: Associate Professor Min-Yen Kan

Deliverables:
Report: 1 Volume

Abstract

Despite years of efforts, detecting plagiarism in programming assignments is still a difficult task for course instructors. Although many detection systems can compute the similarities between pairs of submissions and detect plagiarism clusters for an assignment, they do not provide sufficient information on how code is exchanged among students. More importantly, they do not provide enough information on how plagiarism takes place within a group of students over several assignments. In this project, I have designed and implemented a plagiarism detection system - *Student Submissions Integrity Diagnosis* (SSID). SSID provides visuals that can show information that ranges from a general overview of plagiarism clusters throughout all assignments in a course, to the top similar submissions on any specific student. Compared to existing systems like MOSS and JPlag, SSID prevents mismatches in code segments due to inappropriate selection of the first token in a match. More importantly, SSID allows instructors to indicate suspicious submission pairs and confirm plagiarized pairs. These will be automatically entered into SSID's logs and shared among all instructors. Evidence of plagiarism can now be gathered easily.

Subject Descriptors:

I.5.2	Design Methodology – *Pattern analysis*
I.5.3	Clustering – *Similarity measures*
I.5.5	Implementations – *Interactive systems*
K.3.2	Computer and Information Science Education – *Computer science education*

Keywords:

plagiarism assessment, clustering, programming, similarity, visualization

Implementation Software and Hardware:

Windows 7 Professional (64-bit), CentOS release 5.4 (Final), Java 1.6 SE, Ruby 1.8.6, Ruby on Rails 2.1.1, NetBeans IDE 6.8, MySQL Server 5.0.87

Acknowledgement

I would like to thank my mentors: Associate Professor Kan Min-Yen, Dr. Kazunari Sugiyama, and Mr. Tan Yee Fan, for their guidance and time spent on this project during the past one year.

Table of Contents

1. Introduction

Plagiarism is a serious issue in undergraduate courses involving programming assignments (Vamplew & Dermoudy, 2005). In 2005, the Centre for Academic Integrity (CAI) reported that almost 40% of 50,000 students at more than 60 universities admitted in plagiarism (Jocoy & DiBiase, 2006). Furthermore, Cheang, Kurnia, Lim, & Oon (2003) reported that in 2000, there were 98 plagiarism cases detected for the first assignment in the module CS1102 in the School of Computing, National University of Singapore. Moreover, in 2004, 181 students in the same school admitted to committing plagiarism (Ooi & Tan, 2005).

If students involved in plagiarism are not identified and proper action is not taken, it is unfair to those who produced and submitted original works. Furthermore, students involved in plagiarism learn less than their hardworking counterparts, tarnish the reputation of their own institutions, and reduce the value of their own degrees. All these reasons make detecting and preventing plagiarism the two most important tasks for instructors.

Detecting plagiarism, however, is a time-consuming and tedious process when performed manually. Inspecting n submissions for plagiarism requires one to access the $n \times (n-1)/2$ pairs of submissions. Furthermore, the process becomes more difficult when students hide the traces of plagiarism by modifying the plagiarized copies. Thus, research on automated plagiarism detection in programming assignment was started in the mid-1970s (Moussiades & Vakali, 2005).

Besides detecting plagiarism, research was also done to assist in preventing plagiarism. It was discovered that factors such as the fear of failure and the difficulty of the work influenced students to plagiarize (Sheard, Carbone, & Dick, 2003). Steps that can be taken to prevent plagiarism included setting up learning groups led by instructors to build students' confidence and provide extra learning opportunities (Moussiades *et al.*, 2005). Each group could consist of the students and their plagiarism accomplices. These students can be selected based on the plagiarism clusters detected through manually measuring the similarity between each suspected or each confirmed plagiarism case. However, this is yet again tedious and time-consuming task. A more informative plagiarism detection system is required.

This system should not only provide instructors with the similarities between pairs of submissions, but also the segments that match one another. The system should also give instructors access to the students' plagiarism history. Lastly, the system should provide instructors with visuals on plagiarism clusters to further assist them in planning the learning groups to help students achieve higher academic performance.

1.1. Student Submissions Integrity Diagnosis

I have built a system called *Student Submissions Integrity Diagnosis* (SSID). SSID analyzes and detects plagiarism in Java source code. With appropriate parsers and tokenizers, SSID can be extended to analyze other programming languages as well.

In a nutshell, SSID works as follows:

- First, SSID takes a set of student submissions from a single assignment as the input. Each submission may consist of more than one source file.
- Then, SSID compares the submissions pairwise by searching for matched code segments and computing similarity values.
- After that, SSID generates the results of comparison along with the plagiarism history of each student. The teaching team can now view this information along with the students' submissions before marking a student off as a plagiarism case.
- Subsequently, SSID provides clustering functionality for instructors to detect plagiarism clusters.
- Finally yet importantly, SSID provides 6 visuals to report the detected plagiarism clusters and the individual student diagnosis on plagiarism among the assignments throughout the module.

For evaluators, SSID can be accessed via:

http://aye.comp.nus.edu.sg/myror/poonyanh/SSID/

User account: eva

Password: eva

Sample data is available via:

http://aye.comp.nus.edu.sg/~poonyanh/codes.zip

For enquiries, feel free to contact:

Poon Yan Horn Jonathan (email: u0605243@nus.edu.sg, mobile: 96367357)

1.2. Contributions

The contributions of this project are as follows:

(1) Improving the existing *Greedy-String-Tiling* algorithm (Wise, 1993) to provide exclusion of skeleton code in student submissions and to prevent mismatches in code segments due to inappropriate selection of token as the beginning of a match.

(2) Implementing a plagiarism detection system that allows instructors and teaching assistances to report or record plagiarism cases.

(3) Implementing a plagiarism detection system that provides sharing of plagiarism activities among all course instructors.

(4) Providing visuals to show information that ranges from a general overview of plagiarism clusters throughout all assignments in a course, to the top similar submissions on any specific student.

1.3. Structure of this report

The remainder of this report is structured as follows. In Section 2, I discuss the related works that detect and prevent plagiarism in programming assignments, together with their strengths and weaknesses. In Section 3, I describe how plagiarism detection is performed in SSID. First, I present the algorithms used in pairwise comparison and detecting plagiarism clusters. Second, I show how SSID reacts to attacks adopted by students in their effort to prevent SSID from detecting plagiarism in their work. Third, I explain how I determine the optimal parameter values of SSID, with respect to maximizing detection accuracy while minimizing execution time. In Section 4, I present the external appearance of SSID. First, I describe the module management web interface. Second, I describe the interface for displaying the results from pairwise comparisons. Third, I present the log system interface that displays the plagiarism history of a student. Fourth, I describe the six visuals for displaying information on plagiarism clusters, individual student diagnosis on plagiarism, and, the possible usage scenarios in which each visual can be used. Fifth, I include the development process and the suggestions from reviewers on how the user interface of SSID can be improved. In Section 5, I conclude this report with limitations and future work.

2. Related Work

Plagiarism detection systems perform pairwise comparison of submissions to detect plagiarism. Existing systems employ two types of metrics to perform pairwise comparison for programming assignments: (1) attribute-counting metric, and (2) structure metric (Verco & Wise, 1996).

In attribute-counting metric systems, a pair of student submissions is flagged as plagiarized work if the occurrences of particular attributes in another program are similar. The first plagiarism system, which is based on Halstead's software science metrics (Halstead, 1977), is an attribute-counting metric system proposed by Ottenstein (Ottenstein, 1977). The system identifies a program in terms of four metrics: (1) number of unique operators, (2) number of unique operands, (3) number of operator occurrences, and (4) number of operand occurrences. In order to identify the similarity between pairs of programs more accurately, subsequent attribute-counting metric systems have added metrics such as the number of loops (Donaldson, Lancaster, & Sposato, 1981), the number of control statements (Grier, 1981), the number of keywords (Berghel & Sallach, 1984), and the average length of the procedure or function (Faidhi & Robinson, 1987).

However, the ability of attribute-counting metric systems to detect plagiarism has been proven in Verco *et al.* (1996) to be worse than structure metric systems. Structure metric systems compute similarities for the pairs of submissions based on the similarity between code structures. In order to avoid spurious matches, an essential parameter called *Minimum Match Length* is used. Widely used systems like SIM (Gitchell & Tran, 1999), MOSS (Aiken, 1994), the YAP family (Wise, 1996), and JPlag (Prechelt, Malpohl, & Phlippsen, 2000) are structure metric systems, of which JPlag is the current benchmark system for code plagiarism detection (Ciesielski, Wu, & Tahaghoghi, 2008).

Structure metric systems have the advantage of visual support for code comparison (Moussiades *et al.*, 2005). For example, MOSS and JPlag provide a web interface that displays a side-by-side comparison of source code for each pair of submissions, with each region of matched code shown in a different colour (Prechelt, Malpohl, & Philippsen, 2002).

On the other hand, the state-of-the-art structure metric systems like MOSS and JPlag can be easily confused and the similarity value between two submissions can result in as low as 0%. These systems usually consist of two phases: (1) parsing of code into token sequences, and (2) comparing submissions based on the token sequence generated. By inserting useless code into the plagiarized program, a breaking of the original token sequence is achieved. In addition, because structure metric systems depend on the parameter *Minimum Match Length* to determine a match, a plagiarist can confuse the system by inserting useless code and make sure that there exists no segment from the original source is larger than the *Minimum Match Length*.

For example, from Figure 1, a plagiarist modified the original submission in the initialization clause the *for* loop. Although the output from the program does not change due to the modification, MOSS reported that the two submissions are not similar at all.

```
import java.util.*;

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

```
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");
        }
    }
}
```

Figure 1: Example of confusing MOSS. Code on the left denotes the original submission; code on the right denotes the plagiarized copy. When I submit the original submission for plagiarism detection against itself, MOSS reported a similarity value of 97%. However, when the pair of original and plagiarized submissions is submitted, MOSS reported that no match was found.

However, in Prechelt *et al.* (2000), the authors argued that the successful rate of the plagiarized copy is being undetected is equal to how hardworking the plagiarist is. For a large program, the plagiarist may find difficulty in applying the modifications all over the program.

Other than detecting plagiarism, the systems mentioned above do not provide information on plagiarism clusters. As mentioned in Section 1, detecting plagiarism clusters assists instructors in forming learning groups to provide extra coaching to students. This allows these students to build up confidence in their work, and hence they are less likely to plagiarize for the remainder of the course.

Plagiarism detection systems with the capability to detect plagiarism clusters do exist. Two non-publicly-available systems, PDetect (Moussiades *et al.*, 2005), an attribute-counting metric system, and PDE4Java (Jadalla & Elnagar, 2008), a structure metric system, provide the capability to detect plagiarism clusters. However, the user interfaces of these systems are not presented in their respective papers and thus, unknown.

One important point to note is that none of the systems discussed above provide a management interface where instructors can report suspicious submission pairs and confirm plagiarism cases. Furthermore, the systems do not support viewing of students' plagiarism history within the course or throughout all courses in the institution. Instructors have to depend on the comparison of submission pairs to justify if submissions are considered plagiarized; existing systems do not provide any additional evidence to support decision making by instructors.

3. Plagiarism detection

This section begins with the implementation of SSID. Then I discuss the plagiarism experiment conducted. The discussion includes the modifications students adopted to plagiarize programming submissions in the experiment, and how SSID reacted to these modifications. Finally, I propose the optimal values for the parameters used in the pairwise submissions comparison algorithm based on the results obtained from the experiment.

3.1. System implementation

SSID works as shown in Figure 2:

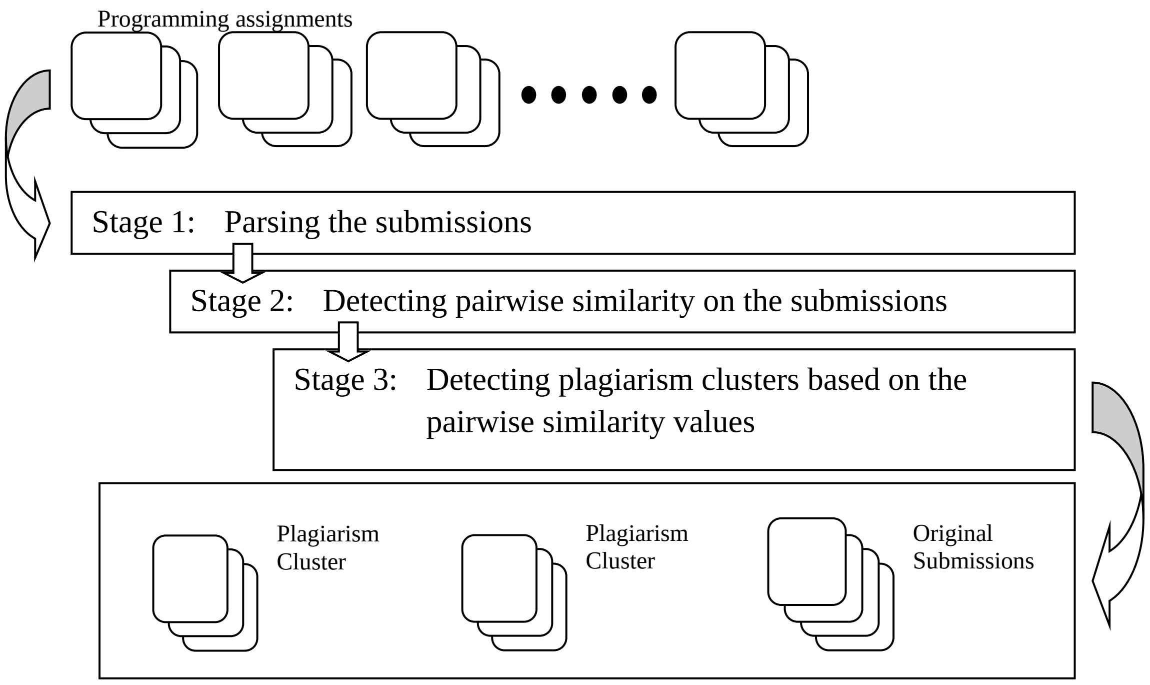

Programming assignments
Stage 1: Parsing the submissions
Stage 2: Detecting pairwise similarity on the submissions
Stage 3: Detecting plagiarism clusters based on the pairwise similarity values
Plagiarism Cluster
Plagiarism Cluster
Original Submissions

Figure 2: SSID workflow

In Stage 1 the submitted submissions from an assignment are first tokenized. From the resulting sequence of tokens, SSID generates *N*-grams which are indexed. In Stage 2, SSID performs pairwise similarity computations and outputs asymmetric similarity values between every pair of submissions. Finally in Stage 3, SSID determines the plagiarism clusters based on the paired similarity values and the cut-off criterion value.

SSID requires three inputs:

N: *Size of N-gram* (*N*) used for indexing in Stage 1 and 2. An *N*-gram is a contiguous subsequence of *N* tokens of a given sequence, *L*: *Minimum Match Length*. The least number of contiguous identical statements required to flag a match in Stage 2,

C: *Cut-off criterion* used in Stage 3. Students with submissions having the pairwise similarity values higher than or equal to this criterion will be grouped together.

I discuss the details of each Stage in the following sections.

3.1.1. Parsing the Submissions

I consider that tokenizers should be language-specific. Each tokenizer should serve only the programming language specified and produce a common output in the format required by the pairwise similarity detection phase. This has been proven to play an important role in maintaining support for a wide variety of programming languages (Schleimer, Wilkerson, & Aiken, 2003).

The main function of tokenizers is to parse the submission code into a sequence of tokens. In addition, tokenizers should omit whitespace and comments in code as format alteration is commonly employed in plagiarism (Liu, Chen, Han, & Yu, 2006). I define four types of tokens: i) constant tokens, ii) keyword tokens, iii) symbol tokens, and iv) variable tokens. Table 1 shows the representation forms obtained from each type of tokens.

Table 1: Token types

Token	Type	Hash Function	Equivalent to
Numbers and characters	Constant	All constant tokens are hashed to the same value.	All constant tokens
Language-specific keywords and reserved words	Keyword	All keyword tokens having the same value are hashed to the same value.	All keyword tokens with the same value
Operators and punctuations (e.g. ; + {)	Symbol	All symbol tokens having the same value are hashed to the same value.	All symbol tokens with the same value
Identifiers	Variable	All variable tokens are hashed to the same value.	All variable tokens

The current version of SSID only accepts Java programs as inputs. However, the system can be easily adapted to compare programs for additional languages when a tokenizer for the target language is introduced. Note that numbers and characters are categorized into a single token type instead of two due to the fact that in Java programming language, it is trivial to modify numeric comparisons into character comparisons. Furthermore, string constants are removed as suggested in Wise *et al.* (1996).

The tokenizer, for instance, transforms the Java source code into tokens as shown by the example in Figure 3. Words and symbols shown in the brackets (under the Generated Token section of Figure 3) denote the value of the token. The numbers following the symbol '#' in the brackets denote the line number of the generated token in the source code.

Java source code

```
1    public class MyClass {
2       public static void main(String[] args) {
3          int value = 1;
4          for (;value<10;value++)
5                System.out.println(value + "");
6       }
7    }
```

Generated Tokens

```
Keyword(class, #1), Variable(#1), Symbol({, #1), Keyword(void, #2),
Variable(#2), Symbol((, #2), Variable(#2), Symbol([, #2), Symbol(], #2),
Variable(#2), Symbol(),#2), Symbol({, #2), Keyword(int, #3), Variable(#3),
Symbol(=, #3), Constant(#3), Symbol(;, #3), Keyword(for, #4), Symbol((, #4),
Symbol(;, #4), Variable(#4), Symbol(<, #4), Constant(#4), Symbol(;, #4),
Variable(#4), Symbol(+, #4), Symbol(+, #4), Symbol(), #4), Variable(#5),
Symbol(., #5), Variable(#5), Symbol(., #5), Variable(#5), Symbol((, #5)),
Variable(#5), Symbol(+, #5), Symbol(), #5), Symbol(;, #5), Symbol(}, #6),
Symbol(}, #7)
```

Figure 3: Example Java Source code and corresponding tokens generated. Words and symbols shown in the brackets (under the Generated Tokens section) denote the value of the token. The numbers following the symbol '#' in the brackets denote the line number of the generated token in the source code.

To facilitate the pairwise comparison process, I implement special rules in the Java tokenizer to recognize the token types shown in Table 1. First, I transform keywords and reserved words like 'true' and 'false' that serve as a constant value into constant tokens. Second, the tokenizer ignores all access modifiers like 'private', 'public'. This is because the insertion or modification to a more general modifier does not affect a program during run-time; the deletion of the modifier or alteration to a stricter modifier results in no effect on some programs as well. Third, curly brackets "{" and "}" are inserted into the bodies of conditional statements and loops (*if*, *while*, *do*, *for*) if they are not explicitly coded in a block.

To reduce the time taken for comparing a token, I assign a unique integer hash value (refer to "Equivalent to" column in Table 1) to each unique string occurrence instead of storing each keyword and symbol as a string.

In order to further reduce the number of comparisons, I enforce two additional attributes for all tokens: i) Beginning of a statement, and ii) Termination of a statement. If a token is the beginning of a statement and/or the termination of a statement, I use these attributes to indicate the location of the token in the statement. In the current implementation, I also mark the first variable token in a variable declaration statement as the beginning of a statement. Furthermore, with the exception of the “;” used in the *for* loop to separate the initialization, termination, and increment clauses, I consider all other “;” indicate the end of a statement. For example, from Figure 3, tokens generated from “class”, “void”, “int”, “value”, “for” and “System” are considered the beginning of a statement. Also, the two tokens generated from “;” in line 3 and 5 are considered the termination of a statement.

Once tokens are formed and *N*-grams are generated, the indexing phase creates a hash table for each submission to map the value of selected *N*-grams to the indices. An *N*-gram is selected for mapping if and only if its first token is marked as the beginning of a statement. I further define that two *N*-grams are equal if and only if their contiguous tokens are also identical (refer to “Equivalent to” column in Table 1).

3.1.2. Detecting Pairwise Similarity on the Submissions

In pairwise similarity detection, the system iterates through the pairs of submissions and computes a similarity value for each pair, one by one. I adopt the *Greedy-String-Tiling* algorithm proposed by Wise (Wise *et al.*, 1993). However, I modify the termination criterion *Minimum Match Length* in the algorithm, that is, the length obtained by counting from the number of contiguous identical tokens to the number of contiguous identical statements. This is because counting the number of statements can provide a better image on the similarity of the pair of submissions from a user’s point of view. I further define two statements are identical if and only if their contiguous tokens are also identical.

In addition, I improve the efficiency by hash value comparison, as suggested in JPlag (Prechelt *et al.*, 2000). I create a hash table for each submission to map the values of *N*-grams to the respective indices as mentioned in Section 3.1.1. Retrieving the indices of an *N*-gram is done in $O(1)$ instead of $O(n)$ originally. This improvement brings down the average complexity of the algorithm for practical cases to about $O(c^2)$, where c is the number of tokens in a submission.

Furthermore, I extend the algorithm to support the exclusion of skeleton code given by instructors from the comparison. To do this, I classify the *mark* property in the original algorithm into two types: i) *match mark* and ii) *skeleton mark*. A *match mark* is equivalent to the original *mark* as defined in the *Greedy-String-Tiling* algorithm; a *skeleton mark*, on the other hand, represents a matched *mark* for both submissions and the skeleton code. Initially, all tokens are unmarked. I consider a match valid (to be flagged as part of plagiarism) if and only if all tokens as specified by the match are *match marked* and the number of statements represented by the tokens satisfies L; I consider a match is obtained directly from the skeleton code if all tokens as specified by the match are *skeleton marked* and the number of statements represented by the tokens satisfies L.

Last but not least, to further improve the efficiency of the algorithm, instead of considering all *N*-grams one by one, my algorithm considers only *N*-grams with the first token being marked as the beginning of a statement. For example, consider the case where the average number of tokens in a statement is 4, an *N*-gram size of 3, and 100 statements in total. With the original implementation, 398 comparisons exist[1] between *N*-grams in a single iteration from the beginning of code to the end. My algorithm, on the other hand, only makes 100 comparisons. This provides a speedup[2] of 3.98 times.

My algorithm takes in three inputs: i) submission *A*; ii) submission *B*; and, iii) skeleton code *S*. The algorithm finds the matched regions in the *A* and *B* in decreasing number of contiguous identical statements, until no more matches satisfying L can be found. The matched regions returned by the algorithm then undergo further processing before being displayed in the user interface. The pseudo-code of my algorithm, together with its explanation, is given in Appendix A.

To provide the user with a clear notion of the similarity of two submissions, I define a similarity measure to reflect the percentage of tokens that are covered by matches. If one submission *A* is identical to another submission *B*, *A* is entirely copied from *B* and the similarity value from *A* to *B* is 100%. This results in the asymmetric similarity measure $sim(A \rightarrow B)$ and $sim(B \rightarrow A)$:

$$sim(A \rightarrow B) = \frac{\text{\# of match marked token in } A}{\text{\# token in } A}, \; sim(B \rightarrow A) = \frac{\text{\# of match marked token in } B}{\text{\# token in } B}$$

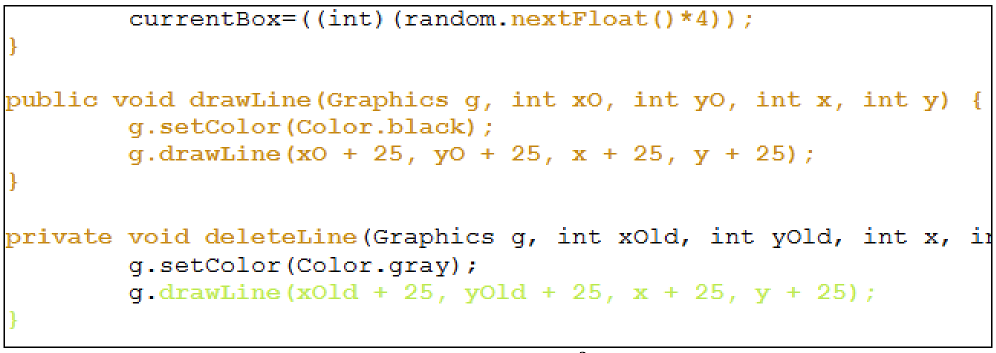
In addition, I define the term $sim(A, B)$ to denote the higher similarity value between $sim(A \rightarrow B)$ and $sim(B \rightarrow A)$:

$$sim(A, B) = \max\left(sim(A \rightarrow B), sim(B \rightarrow A)\right)$$

[1] $Number\ of\ comparisons = 4 \times 100 - 3 + 1 = 398$

[2] $Speedup = \frac{Number\ of\ comparisions\ in\ original\ algorithm}{Number\ of\ comparisions\ in\ my\ algorithm}$

Judging from the matched region found, my algorithm provides a more accurate marking of tokens over the one implemented in JPlag. I only consider JPlag as it is an improvement over the original algorithm (Moussiades *et al.*, 2005). For instance, matches in JPlag may originate or terminate at unsuitable places. According to the algorithm mentioned in Prechelt *et al.* (2000), it is possible to have a match origin and / or terminate in the middle of a statement. This increases unnecessary matching and prevents the detection of other potential matches. For example, from Figure 4, under manual checking, the blown regions in both Figure 4(a) and Figure 4(b) should consist of the methods “drawLine” and “deleteLine”. However, the result flagged in Figure 4(a) was in the middle of an unrelated statements and separated the region into two segments. If JPlag raises only matches with tokens that are the beginning of statements, the mismatch shown in Figure 4 can be avoided. On the other hand, my algorithm, with the assistance from the tokenizer, manages to avoid such mismatches.

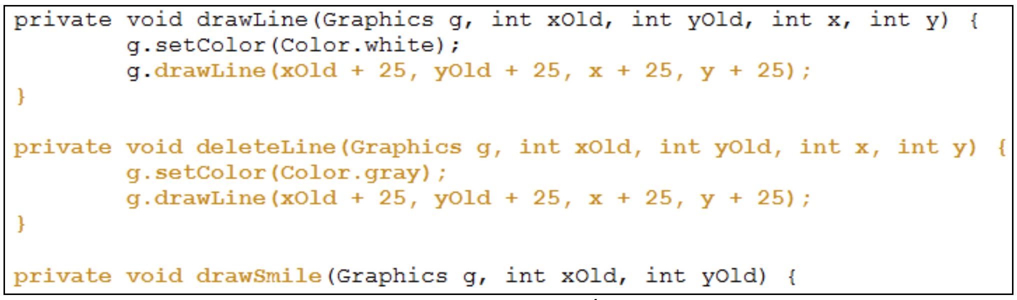


```
        currentBox=((int)(random.nextFloat()*4));
}

public void drawLine(Graphics g, int xO, int yO, int x, int y) {
        g.setColor(Color.black);
        g.drawLine(xO + 25, yO + 25, x + 25, y + 25);
}

private void deleteLine(Graphics g, int xOld, int yOld, int x, i
        g.setColor(Color.gray);
        g.drawLine(xOld + 25, yOld + 25, x + 25, y + 25);
}
```

(a) Partial code of submission 943151[3] from JPlag example



```
private void drawLine(Graphics g, int xOld, int yOld, int x, int y) {
        g.setColor(Color.white);
        g.drawLine(xOld + 25, yOld + 25, x + 25, y + 25);
}

private void deleteLine(Graphics g, int xOld, int yOld, int x, int y) {
        g.setColor(Color.gray);
        g.drawLine(xOld + 25, yOld + 25, x + 25, y + 25);
}

private void drawSmile(Graphics g, int xOld, int yOld) {
```

(b) Partial code of submission 942261[4] from JPlag example

Figure 4: Example of mismatching report by JPlag. The mismatch region is coloured in brown. Under manual checking, the correct identical region should consist of the “drawLine” and “deleteLine” methods.

[3] Retrieved on March 29, 2010 from https://www.ipd.uni-karlsruhe.de/jplag/example/match1.html, lines 181 to 189.
[4] Retrieved on March 29, 2010 from https://www.ipd.uni-karlsruhe.de/jplag/example/match1.html, lines 181 to 189.

3.1.3. Detecting Plagiarism Clusters Based on the Pairwise Similarity Values

As mentioned in Section 2, instructors can set up learning groups by detecting plagiarism clusters. In order to make the clustering task as sample as possible, the clustering algorithm must satisfy the following requirements:

(1) The number of groups must be determined automatically.

(2) A cut-off criterion value (*C*) is used to determine the formation of clusters (groups). Submission pairs higher than or equal to *C* should be grouped together.

(3) The algorithm must assign a student to at most one group. If a student plagiarizes from two or more sources, it is easier for the grader to access the group than to access individual pairs one by one. Therefore, no overlapping of groups is allowed.

(4) The algorithm must be deterministic, meaning the same result must be produced with the same inputs for all runs.

Based on the requirements listed above, I choose to apply DBScan (Ester, Kriegel, Sander, & Xu, 1996). I construct the set of submissions as a weighted complete graph $G = (V, E)$. Each vertex in V represents a student submission and each edge in E represents a pair of student submissions. Each edge (A, B) is weighted by its maximum similarity value $sim(A, B)$. I classify non-plagiarizing students as *noise* and group suspicious students according to the similarity value between their submissions.

The algorithm requires two additional parameters in addition to the points in a graph:

Eps - The maximum radius of the neighbourhood,

MinPts - The minimum number of points in an *Eps*-neighbourhood of a point.

In the setting of plagiarism detection, *Eps* represents the least maximum similarity value between two submissions and *MinPts* represents the minimum number of student submissions required to form a cluster with a target submission such that the maximum similarity values between the submissions and the target submission are higher than or equal to *Eps*. I define the values of the parameters as follows:

Eps = 1 - *C*, i.e. the least maximum similarity value between two submissions,

MinPts = 1, i.e. to form a cluster, a submission needs at least another submission in its *Eps*-neighbourhood; this implies that at least two submissions are needed to form a group.

In addition, the distance measure between two submissions in *DBScan* is $1 - sim(A, B)$.

3.2. Experiment Survey

In order to examine the accuracy on detecting plagiarism of SSID, I have conducted an experiment to gather the information on how a Computing student would plagiarize in programming assignments.

The gathered information includes:

- The possible types of *attacks* a student may implement to *confuse* SSID,
- The *attacks* that SSID is able or unable to detect. In other words, the limitations of SSID.

Here, I define the term *attack* as the act of modifying the existing source code in order to hide the traces of plagiarism, and define the term *confusion* to denote the failure of identifying the similarity by SSID between the original code segment and the result of the *attack*.

3.2.1. Experiment Setup

The experiment consists of three assignments written in Java. Explicit details of the assignments are provided in Appendix B. I have selected four original, non-plagiarized submissions for each assignment as samples for participants to perform plagiarism. The assignments and submissions are selected based on the following criteria:

- The assignments were originally sit-in, i.e., the students are being carefully monitored and communication (physically and/or electronically) between two computer hosts is not possible; thus, I assume the rate of plagiarism is zero,
- The submissions show different approaches to solve each problem; the asymmetric similarity values between each pair of them are low,
- Selected submissions must contain common coding patterns for the target language of Java (conditional statements, loops, accessors, mutators, variable declaration, function calls and arrays etc). These are commonly used in actual assignments.

System Configuration

In this experiment, the two input parameters required for detecting pairwise similarity are predefined as follows:

- The size of *N*-gram (*N*) is set at **2**.
- The *Minimum Match Length* (*L*) is set at **2** statements.

Participants

Participants of the experiment were seniors and graduating students from a Bachelor of Computing course. There were 28 participants in total. Each participant was required to produce a plagiarized copy of each assignment from the selected original samples.

3.2.2. Similarities Between the Original and Plagiarized Submissions

The similarities between the original and the plagiarized submissions, grouped by participant ID, are shown in **Appendix C**. Means and standard deviations of the similarities are shown in Appendix D.

3.2.3. Types of *attacks*

In this section, I analyze the different types of *attacks* that the participants attempted.

For example, *confusion* can occur when a single line of code has been inserted into a plagiarized code segment that separates the original code segment into two. As SSID uses token sequences in matching, the insertion may result in a mismatch between the plagiarized version and the original version, depending on the number of contiguous statements in the two separated code segments.

Each of the following headers prefixed with numbering describes a category of *attack*. Two numbers are displayed in parentheses after each heading: the left denotes the number of times the *attack* successfully *confuses* SSID; the right denotes the number of plagiarized copies found using that *attack* as part of its strategy.

Immutable attacks

The attacks discussed here do not result in any modification to the token sequences that were generated; SSID can detect these *attacks*.

(1) ***Insertion, modification or deletion of comments* (0/54)**

As comments are ignored during the parsing phase, SSID is immune to this type of *attack*.

(2) ***Indention, spacing or line breaks modifications* (0/54)**

As whitespaces are ignored during the parsing phase, SSID is secure against this type of *attack*.

(3) ***Renaming of identifiers* (0/60)**

Although many participants may believe that renaming of identifiers used in methods, variables and classes may *confuse* SSID, this effort is futile since identifiers are considered identical in the comparison phase.

(4) ***Constant modification* (0/2)**

Attacks were performed by changing the value of an unused constant. SSID, however, is immune to this kind of *attack* due to the fact that constant values are identical in the token representation.

(5) ***Insertion, modification, or deletion of modifiers* (0/9)**

In most programming languages, accessing variable, a method or a class can be limited by the defined access modifier. The insertion or modification to a more general modifier does not affect the program during run-time; the deletion of the modifier or alteration to a stricter modifier results in no effect on some programs as well. In addition, with the help of language compilers, the participants can easily check for errors through code compilation. Six participants attempted to modify the given sample with this type of *attack* but failed as SSID ignores modifiers in token generation during the parsing phase.

(6) ***No change* (0/1)**

A submission is found to be the same as the original sample. The *attack*, of course, is detected by SSID.

Size dependent attacks

This section discusses *attacks* that depend on the size of the code segment to successfully *confuse* SSID. In a small program, these *attacks* may come in handy. However, in a large program, they may be inefficient for one to perform such *attacks* to *confuse* SSID. In addition, *attacks* discussed in this section require more time and effort on code modification than the *attacks* discussed in the previous section.

(1) ***Reordering of independent statements* (11/16)**

Out of the 16 *attacks* detected, SSID was *confused* by 11 of them. This is due to the fact that the sub-segments involved are too small to satisfy *L*. For example, one *attack* reordered all of the three contiguous and independent statements within a block. Since the *attacked* sub-segments were small (one statement per sub-segment), each sub-segment did not fulfil *L* and thus SSID was *confused*.

However, such *attacks* were detected by SSID when the reordered statements consisted of two large blocks or they were structurally identical. As an example, consider the code segments as shown in Figure 5.

Original	Reordered

left = tree.getLeft();
right = tree.getRight();
	right = tree.getRight();
	left = tree.getLeft();

Figure 5: An example of reordering statements

The resulting token sequence was the same even after the statements were reordered. SSID raised this as a match since the token representations for both versions were identical.

(2) ***Reordering of methods* (7/23)**

Similar to the reordering of independent statements, methods can also be reordered by the participants. The success of such *attack* depends on the number of detected statements within the method block.

Among all the *attacks*, the most popular one was reordering the accessors and mutators. Although these methods usually contained only one statement, SSID could detect most of the *attacks* as the reordering was done by the rearranging several of these methods in blocks.

(3) ***Insertion or removal of parentheses* (0/21)**

Some participants believed that changing the coding style through adding/removing parentheses for single-lined *if* statements could *confuse* the investigators. The attempts, however, resulted in failure as the parser automatically inserts missing parentheses to single-lined conditional statements and loops.

Although none of the participants succeed, it is possible to *confuse* SSID by inserting useless parentheses. By inserting useless parentheses to create blocks or prioritize a sub-expression, the generated token sequence will be affected.

(4) ***Inlining or refactoring of code* (15/22)**

Six plagiarized submissions have *confused* SSID by inlining methods. The inlined methods were either less than *L* or combined with other successful *attacks* like statement reordering.

In addition, SSID was *confused* by nine *attacks* from refactoring. Seven of which were by combining the conditional statements in an *if-else* statement (see Figure 6). These were completed by grouping the identical operation in the respective *then* clause and joining the conditional statements with the logical 'OR' operators.

Original	Plagiarized

if (myString.equals("right")) return 1;
if (myString.equals("left")) return 1;
return 0;
	if (myString.equals("right") ||
	myString.equals("left"))
	return 1;
	return 0;

Figure 6: An example in combing the conditional statements in an *if-else* statement

Another *attack* put the in-loop break condition into the looping condition. The last *attack* added a new method named "init" to execute the codes that were originally in the constructor. The participant called the "init" method for the instance immediately after the instance construction has completed.

Confused attacks

In this section, I discuss the types of *attacks* that can successfully *confuse* SSID in all situations.

The *attacks* have been used 92 times; all of them have successfully *confused* SSID. Although it sounds successful in the first place, none of these *attacks* were commonly adopted. No participant adopted these *attacks* in all three plagiarized submissions.

(1) ***Redundancy* (10/10)**

Redundancy *attacks* include removing and inserting dead code. Extra lines of codes that are not executed at all are removed from the original submission and / or inserted to the plagiarized copy. In either action, SSID is *confused*.

For example, one participant copied a method entirely and modified the copied method without ever calling it.

(2) *Scope modification* (9/9)

Three *attacks* were done by shifting the declaration of variables from an inner block to an outer block (extending the life of the variables). For example, in Figure 7, instead of having integer *k* to be declared as shown in the column "Original", the declaration is brought outward and results in the code segment as shown in the column "Plagiarized".

Original	Plagiarized

for (int i = 0; i < 10; i++) {
    int k;
    ...
}
	int k;
	for (int i = 0; i < 10; i++) {
	    ...
	}

Figure 7: An example on extending the life of variables

Two *attacks* brought the declaration of variables from an outer block into the inner block (reverse of the example above).

The last type of *attack* in this category is the usage of the keyword "this". In Java, one can access the instance variable, method within the instance by simply calling upon them. In addition, one can also access them by prefixing with the keyword "this". Such modification results in a different token sequence generation.

(3) *Modification of control structures* (25/25)

In this type of *attack*, participants modified the control structures, inserted and/or removed new statements or expressions into or around the modified structures. As a result, the generated token sequence was affected and SSID was successfully *confused*. The detection of this *attack* solely depends on the detection of the inner block from the modified structures.

The following types of *attacks* in loop statements do *confuse* SSID successfully:

- Six *attacks* modified a *for* loop into a while-loop, and two *attacks* changed a *while* loop into a *if-do-while* loop. The modifications also include declaring variables and looping conditions in different coding orders.
- Four *attacks* were done by looping in a reversed direction. Instead of having the loop executes from a lower boundary, increments the control variable after each loop and terminates upon reaching the upper boundary, the participant

modifies the loop to execute from the upper boundary and terminate upon reaching the lower boundary.

- In three *attacks*, the participants increased the initial value of the control variable by 1 and made appropriate modification to the looping condition. When accessing the value of the control variable within the loop, the read value is decreased by 1 to accommodate the change to the initial value.
- There is one case where the participant shifted the statements in the increment segment of the *for* loop to the end of the loop body.

The following *attacks* have been done to the *if* statements:

- Nine *attacks* were found to be negating the condition of *if* statement and switching the bodies of the *then* and *else* clauses.
- One *attack* had successfully *confused* SSID by moving the statements following a *if...return* clause into a newly inserted *else* clause.

In another case, the participant modified the *if-else* statements into contiguous *if* statements.

(4) ***Declaration of variables*** **(15/15)**

In Java, variables of the same type can be declared in a single statement. A possible type of *attack* is to declare variables of the same type from different statements in one statement (see Figure 8) or the other way round.

Original	Plagiarized

int a, b, c, d;
String e, f, g;
	int a; int b; int c; int d;
	String e; String f; String g;

Figure 8: An example on declaration of variable

Since the modification of declaration directly affects the generation of token sequence, this type of *attacks* can successfully *confuse* SSID.

(5) ***Modification of method parameters*** **(3/3)**

For methods with multiple parameters, one can easily change the sequence of the parameters without any effect on the accuracy and performance of the program. The modifications, however, may result in a change of token sequence in both method declarations and the calling statements.

(6) ***Modification of import statements*** **(2/2)**

In two submissions, the participants modified the import statements by specifying the packages required for the program rather than using the wildcard character to specify that all classes where that package is available.

This type of modification results into different set of tokens generated.

(7) ***Introduction of bug*** **(1/1)**

An interesting *attack* involved the introduction of a bug into the plagiarized program to avoid detection. The participant made the *attack* successful by removing the check for null pointer on a variable.

(8) ***Declaration or deletion of temporary variables for sub-expressions*** **(16/16)**

This type of *attacks* targets sub-expressions. As tokens are generated based on the code structure, modifications to sub-expressions will change the token sequence.

From 16 submissions, I observe two different usages of this type of *attack*:

- Participants declare temporary variables to store the return values from sub-expressions and access the variable when needed.
- Instead of declaring one-time-usage temporary variables to store the return values from sub-expressions, participants insert the sub-expressions to the location where the results are needed.

(9) ***Modification of mathematical operations and formulae*** **(5/5)**

5 submissions modified their mathematical operations:

- In 2 submissions, the participants changed the value of *true* to *!false*.
- In 1 submission, the participant replaced the increment operation (*i++*) by the explicit assignment (*i=i+1*).
- In 1 submission, the participant factorized the logical condition clause of the *if* statement (see Figure 9**).**

Original	Plagiarized

if (!myString.equals(“left”)
    || !myString.equals(“right”)) {
        return;
}
	if (!(myString.equals(“left”) &&
	    myString.equals(“right”))) {
	        return;
	}

Figure 9: An example on factorizing logical condition clause of the if statement

- In another submission, two modifications were done. First, instead of assigning a value to the array "*tree*" with the index "*free++*", the participant modified the code into the following:

  tree[free] = ...;

  free = free + 1;

- In addition, the participant simplified a mathematic formula used in the original submission.

(10) *Structural redesign of code* (6/6)

Six submissions were found to have partially structurally redesigned. All the modifications in this category successfully *confused* SSID.

- One participant changed the access modifiers of the variables in the main data structure with public and modified the statements that access the variables through mutator and accessor methods into a direct access (see Figure 10).

Original	Plagiarized

**public class A {
    private int b;
    public void setB(int b) {
        this.b = b;
    }
}
 	
public class C {
    private A a;
    public void run() {
        a.setB(3);
    }
    ...
}**
	**public class A {
	    public int b;
	}
	
	public class C {
	    private A a;
	    public void run() {
	        a.b = 3;
	    }
	    ...
	}**

Figure 10: An example on modification to directly access a variable

- One participant made changes to the code from a method which returns a *boolean* value to a method which returns an *integer* value with mathematical operations to achieve the same result.
- One participant reconstructed the computation method from scratch.
- One participant removed the class variables that were used to share values between methods and extended method parameters to accommodate the changes.

- One participant stored and accessed the value of *true* and *false* as an *integer* value instead of *boolean* value.
- One participant refractored a small segment of a method and rephrased the refractored segment in a different implementation.

3.3. Determining the Optimal Values for Input Parameters

In this section, I determine the optimal values for the input parameters required by SSID based on the similarity between the original and plagiarized submissions. I wish to maximize accuracy, and where possible, minimize time consumption by tuning the following input parameters:

N: Size of *N*-gram (*N*) used for indexing.

L: *Minimum Match Length*, the least number of contiguous identical statements required to flag a match;

I define three sets of submissions for each assignment in this part:

(1) ***ORG***: This contains only the original submissions, except the selected sample submissions, from the sit-in lab environment.

(2) ***PLAG***: This contains the plagiarized submissions from the experiment and the selected sample submissions. However, when the set is referred, only the pairs of submissions formed by the plagiarized submissions and its respective source submission are considered. I leave the detection of plagiarism from more than two participants to the clustering phase.

(3) ***ALL***: This represents the union of (1) and (2). A pair of submissions formed by a submission from ***ORG*** and a submission from ***PLAG*** is considered as an original submission pair, since the plagiarized submission should not be similar to any original submission other than the one it plagiarized. Similar to above, I consider only the pairs of submissions formed by the plagiarized submissions and its respective source submission. I do not include the pairs formed by two submissions from ***PLAG*** if either of them is not the direct source of another.

Table 2 shows the sets of submissions.

Table 2: Overview of all submission sets: assignment and submission set, number of original submissions, original submission pairs, plagiarized submissions, and plagiarized submission pairs

Assignment-Set	**Original submissions (*o*)**	**Original submission pairs (*op*)**[5]	**Plagiarized submissions (*p*)**	**Plagiarized submission pairs (*pp*)**[6]
1-*ORG*	87	3741	0	0
2-*ORG*	86	3655	0	0
1-*PLAG*	0	0	32	28
2-*PLAG*	0	0	32	28
1-*ALL*	87	3741	32	28
2-*ALL*	86	3655	32	28

I have left out assignment 3 – Binary tree traversal because most submissions in assignment 3 were programmed only with the common data structures and were not functional. This resulted in a very high similarity value for most pairs (among the total 14878 pairs, 14517 pairs are found to be more than 80% similar and only 361 pairs are found to be less than 50% similar).

Methods of Determination

I determine the optimal values as follows:

(1) First, I focus on SSID's accuracy in determining plagiarism submissions and original submissions. The distribution of similarity values for the submission pairs in ***ORG*** and ***PLAG*** is discussed with different values of *N* and *L*. I vary the values of *N* with 1, 2, 3, 4, 7, 10, and 15, and the values of *L* with 1, 2, 3, 5, and 10. The results are used to determine a candidate set of values for *N* and *L*.

(2) Then, I verify the accuracy of SSID with the submission pairs in ***ALL*** with the values in the candidate set obtained in (1). This ensures the correctness in detecting plagiarism.

(3) After (2), I look into system performance. Given the candidate set obtained in (1), the execution time required for each combination of values in the set are compared.

(4) Finally, I determine the optimal values for *N* and *L* based on the results from (1) to (3).

Evaluation Measure

I evaluate my approach using *precision* (*P*), *recall* (*R*) and *F-measure* (*F*). *P* is the ratio of detected pairs that are actual plagiarism pairs; *R* is the ratio of all plagiarism pairs that are

[5] Calculated with the formula $\binom{p}{2} = \frac{p\times(p-1)}{2}$

[6] I only consider the pairs of submissions formed by the plagiarized submissions and its respective source submission.

actually flagged; and *F* represents the harmonic mean of *P* and *R*. *R*, *P* and *F* range from 0 to 1. The higher these values, the better evaluation results are.

For a given threshold τ, the number of pairs of submissions with similarity values higher than or equal to τ (*d*), the number of true plagiarism pairs in *d* (*t*), and the number of true plagiarism pairs in total (*pp*), I calculate *P*, *R* and *F* as follows:

$$P = {}^{t}/_{d} \qquad R = {}^{t}/_{pp} \qquad F = 2\frac{P \times R}{P + R}$$

3.3.1. Distribution of Similarity Values

I demonstrate the different distributions of similarity values under the different combinations of *N*-gram indexing size and the threshold of contiguous matching statements required to flag a match.

Distribution graphs are used to show the relationship between the similarity value and number of submission pairs found. Each graph consists of two series: one denotes the distribution of similarity values for the submission pairs in ***ORG***; the other denotes the distribution of similarity values for the submission pairs in ***PLAG***. Appendix E shows these graphs.

Assignment 1: Perilous Cities

The *perilous cities* assignment consists of 3741 original submission pairs and 28 plagiarised submission pairs. From Appendix E, Figure 32 to Figure 36, I observe the following:

- When *L* = 1, the similarity distribution varies with different values of *N* (see Figure 32). The distribution becomes more stable when the value of *L* increases (see Figure 33 to Figure 36). This is be due to the false-positive matching of statements in pairwise comparison (see Figure 11).

Submission 1	**Submission 2**
isSafe = false;	size = 0;

Figure 11: Example of false-positive matching of statements in *Perilous Cities* assignment when *L* = 1. The statement "isSafe = false;" is mismatched with "size = 0;" by SSID.

- When *L* ≤ 3 as shown in Figure 32 to Figure 34, the two series can be separated perfectly for all values of *N*. When *L* = 1, the longest range of separation is from 63% to 73% with *N* = 7. When *L* = 2 and 3, the range of separation is equal for all values of *N* from 57% to 68% and 45% to 57%, respectively.

- However, when $L \geq 5$ (see Figure 35 and Figure 36), the series overlap each other for all values of *N*. For all *N* when L = 5, a maximum F of 0.966 can be obtained from the $sim(A, B)$ ranging from 35% to 36%; for all *N* when L = 10, a maximum F of 0.906 can be obtained for $sim(A, B)$ = 28%. This implies that the accuracy of SSID decreases when *L* increases.
- When the value of *N* is increased, the sensitivity of detection decreases. For example, according to Figure 32, the distribution curve shifts towards the left significantly in the case of *N* > 4. Compared with the actual data, I found that similarity values decrease in 83% and 18% of the submissions pairs in ***ORG*** and ***PLAG*** respectively when *N* is increased from 4 to 7.

Assignment 2: Binary Tree Genealogy

The *binary tree genealogy* assignment consists of 3655 original submission pairs and 28 plagiarised submission pairs. From Appendix E, Figure 37 to Figure 41, I observed the following:

- When L = 1, the similarity distribution varies with different values of *N* (see Figure 37). The distribution becomes more stable when the value of *L* increases. This may be due to the false-positive matching of statements in pairwise comparison.
- When $L \leq 2$, the two series can be separated perfectly for all values of *N* (see Figure 37 and Figure 38). When *L* = 1, the longest range of separation is from 50% to 65% with *N* = 10. When *L* = 2, the range of separation is equal for all values of *N* from 47% to 57%.
- However, when $L \geq 3$, as shown in Figure 39 to Figure 41, the series overlap for all values of *N*. For all *N* when L = 3, a maximum *F* of 0.983 can be obtained from $sim(A, B)$ ranging from 40% to 46%; for all *N* when L = 5, a maximum *F* of 0.963 can be obtained from the $sim(A, B)$ ranging from 38% to 46%; for all *N* when L = 10, a maximum *F* of 0.873 can be obtained for $sim(A, B)$ = 25%. This implies that the accuracy of SSID decreases when *L* increases.
- With the increase of the *N*, the sensitivity of detection decreases. For example, from Figure 37, the distribution curve shifts towards the left significantly when *N* > 4. Compared with the actual data, I observed that similarity values decrease in 81% and 11% of the submission pairs in ***ORG*** and ***PLAG*** respectively when *N* is increased from 4 to 7.

Conclusion on Distribution of Similarity Value

I can conclude that SSID has excellent performance in identifying the plagiarized pairs and the original pairs.

With $L \leq 2$, there exists perfect ranges of cut-off thresholds in both assignments for all N. However, I have also shown that the similarity distribution is unstable when $L = 1$, which may be due to false-positive matching. Furthermore, the accuracy drops when the value of L increases. When $L = 3$, even though there exists a perfect range of separation for the series in the *Perilous Cities* assignment from 46% to 56%, the series overlap in the *Binary Tree Genealogy* assignment and a maximum F of 0.983 is obtained. This implies that the result obtained from having $L = 3$ is not as stable as $L \leq 2$. In addition, when $L = 5$, I can only obtain a maximum F of 0.966 and 0.963 for the *Perilous Cities* assignment and the *Binary Tree Genealogy* assignment, respectively. The situation becomes worse when $L = 10$ as the *maximum F* value drops to 0.906 for the *Perilous Cities* assignment and 0.873 for the *Binary Tree Genealogy* assignment. Therefore, I have decided to use advantage of $L = 2$ in my candidate set of values.

With regard to N, by comparing the figures for both assignments with the values of L as mentioned above, I have identified that using the values of $N = 1$, 2, 3, and 4 gives the best values in separating the series. I do not consider values of N greater than 4 because when $N > 4$, the sensitivity of detection decreases significantly.

3.3.2. Accuracy Verification

SSID is capable of separating plagiarized submission pairs from original submission pairs accurately.

I have computed the F values in ***ALL*** as shown in Table 3. Since ***ALL*** contains the pairs formed by an original submission and a plagiarized submission, it is necessary to verify that SSID can separate these pairs from the plagiarism pairs and treat them as the original pairs.

Table 3 shows that SSID can produce a clear separation of the plagiarism pairs from the original ones in ***ALL***. I found that the values of N and L in my candidate set are suitable to serve as the optimal values.

Table 3: Maximum F-Measure with the different combinations in the candidate set

Set	*N*	*L*	Highest *F*	Range of Similarity values
1-all	1	2	1	57% to 68%
1-all	2	2	1	57% to 68%
1-all	3	2	1	57% to 68%
1-all	4	2	1	57% to 68%
2-all	1	2	1	47% to 57%
2-all	2	2	1	47% to 57%
2-all	3	2	1	47% to 57%
2-all	4	2	1	47% to 57%

3.3.3. Performance

In order to provide fast response of similarity values to a large set of submissions, performance is important. Appendix F shows the average time required to compute the similarity values for the pairs of submissions in $\boldsymbol{ALL}$ for both assignments based on the values of N and L in my candidate set. The computation is done on an Intel(R) Core(TM) 2 Duo CPU P9700 (2.80 GHz) laptop using JDK 1.6.0 Update 14 (32-bit) on Windows 7 Professional (64-bit). The JVM is set to execute only on the first processor (CPU 0). The recorded time is the average time taken from three executions. Each execution begins with reading and parsing the codes and terminates by performing all of the pair-wise comparisons and ignoring the patterns found in the skeleton files.

From Appendix F, although the complexity of the algorithm remains the same, the algorithm achieves an average computation speedup[7] of 1.85 times with $N = 2$ when compared to $N = 1$. The execution time required decreases further, but less significantly, for $N = 3$ and $N = 4$. The increase in execution duration as N becomes smaller may be due to the fact that when the number of unique N-grams decreases, more collision between indices having the same N-gram value occurs.

3.3.4. Optimal values determination

I have filtered non-potential choices and decided on a candidate set of parameter values. I have also discussed how SSID performs in terms of accuracy in the experimental situation under the different parameter values. Furthermore, I have talked about how the choices affect the performance of SSID in terms of execution time. Finally, I determine the optimal values for N and L based on the observations from the previous sections.

[7] $Speedup = (Execution\ Time\ when\ N = 1)\ /\ (Execution\ Time\ when\ N = 2)$

In terms of detection accuracy, using the value $N = 1$ to 4 shows the most promising results in all combinations; $L = 2$ also shows perfect separations for all values of N. However, it is possible that using lower value of N results in long execution time and thus is not preferable.

Therefore, I take $N = \mathbf{4}$ and $L = \mathbf{2}$ to get optimal performance in SSID.

4. User Interface

In addition to comparing pairwise similarity between submissions and detecting plagiarism clusters, SSID comes with a web interface to display the matched code segments between each pair of submissions, a log system to record plagiarism activities for each student, and visuals to illustrate plagiarism clusters and other plagiarism-related information for gathering evidence on plagiarism.

SSID is designed to accommodate two roles according to the types of academic staff involved in a module:

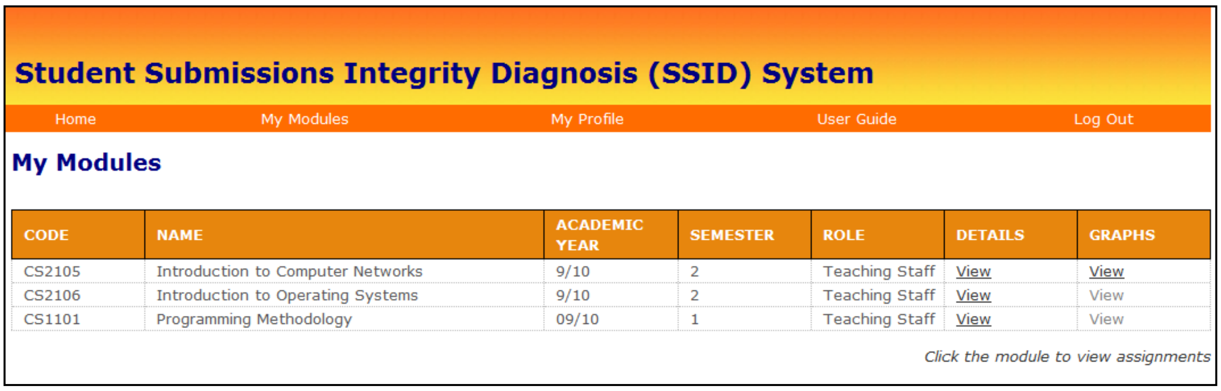
(1) *Teaching assistant* (*TA*), and
(2) *Teaching staff*

*TA*s can only access the module within its term of validity. They can perform all actions within the module except confirming plagiarism cases and declaring investigation results. In addition, *TA*s can only view logs within the module itself.

On the other hand, *teaching staff* can access the module even after its expiry. They can perform all actions within the module, including those not allowed in the *TA* role. Furthermore, they can view all the logs (regardless of module) of any student in the module. This provides them with flexibility in evidence gathering.

4.1. Module Management Interface

The *Module Management interface* (Figure 12) allows users to manage the modules that they can access. Through the *Module Management Interface*, the teaching team can, for example, upload assignment submissions to the system for plagiarism checking, and perform plagiarism clusters detection.



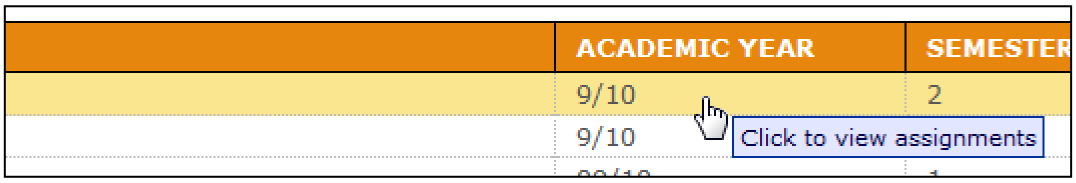
My Modules

CODE	NAME	ACADEMIC YEAR	SEMESTER	ROLE	DETAILS	GRAPHS
CS2105	Introduction to Computer Networks	9/10	2	Teaching Staff	View	View
CS2106	Introduction to Operating Systems	9/10	2	Teaching Staff	View	View
CS1101	Programming Methodology	09/10	1	Teaching Staff	View	View

Click the module to view assignments

Figure 12: SSID *Module Management interface*.

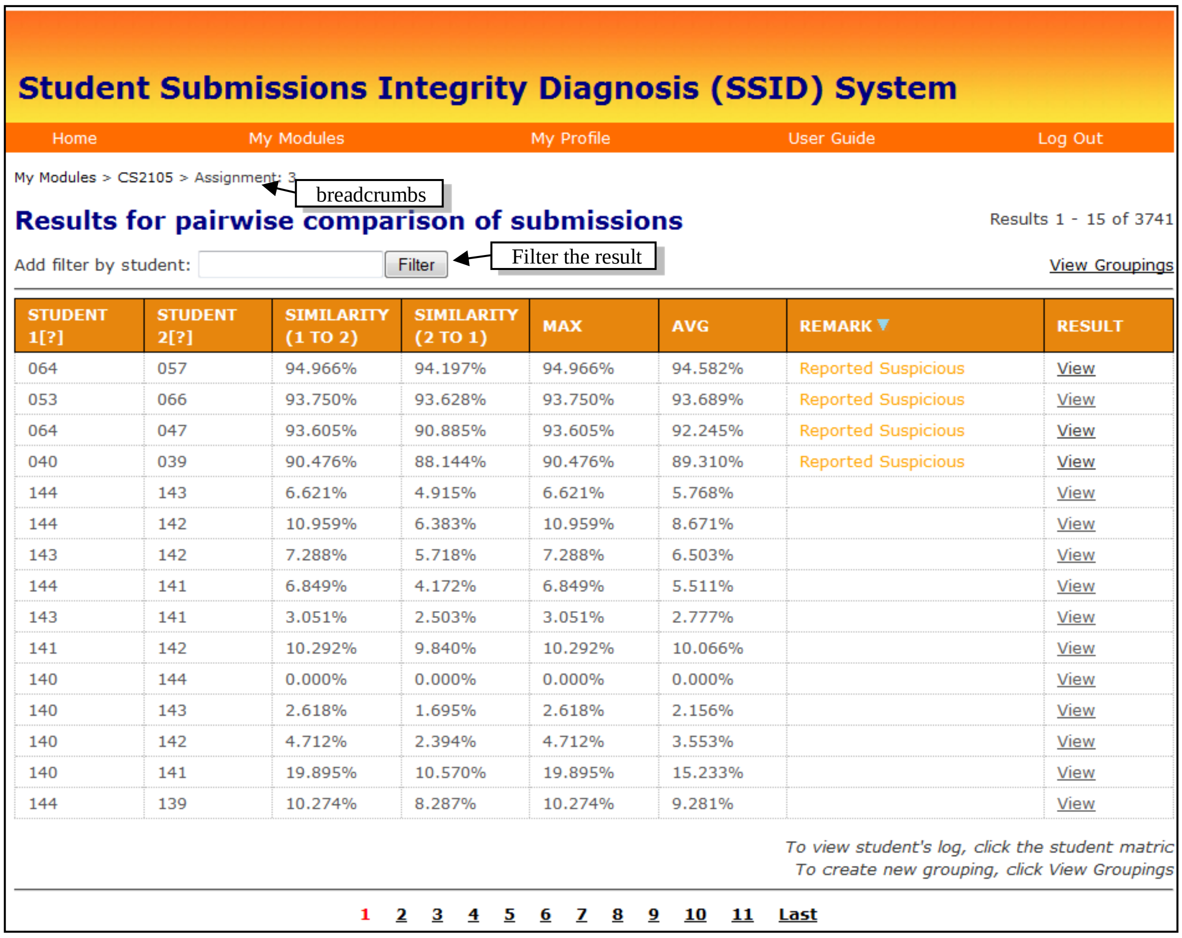
Tables are used in most of the *Module Management interface* to list entries. To provide easy access in most displays throughout the application, clicking the row will direct the user to the default subpage (with the exception where a link is available in the cell). For example, from Figure 12, the default subpage for each module in the list is the assignment-listing page. In addition, the row that the cursor is placed over will be highlighted to provide a visual contrast to aid the user in locating a specific row (see Figure 13).



	ACADEMIC YEAR	SEMESTER
	9/10	2
	9/10	

Figure 13: Highlighting of current row in the *Module Management interface*.

In addition, navigation breadcrumbs are used in subpages to ease navigation to parent pages and to help the user localize themselves within the system (see Figure 14).



Student Submissions Integrity Diagnosis (SSID) System

Home | My Modules | My Profile | User Guide | Log Out

My Modules > CS2105 > Assignment: 3

Results for pairwise comparison of submissions

Results 1 - 15 of 3741

Add filter by student: Filter

View Groupings

STUDENT 1[?]	STUDENT 2[?]	SIMILARITY (1 TO 2)	SIMILARITY (2 TO 1)	MAX	AVG	REMARK	RESULT
064	057	94.966%	94.197%	94.966%	94.582%	Reported Suspicious	View
053	066	93.750%	93.628%	93.750%	93.689%	Reported Suspicious	View
064	047	93.605%	90.885%	93.605%	92.245%	Reported Suspicious	View
040	039	90.476%	88.144%	90.476%	89.310%	Reported Suspicious	View
144	143	6.621%	4.915%	6.621%	5.768%		View
144	142	10.959%	6.383%	10.959%	8.671%		View
143	142	7.288%	5.718%	7.288%	6.503%		View
144	141	6.849%	4.172%	6.849%	5.511%		View
143	141	3.051%	2.503%	3.051%	2.777%		View
141	142	10.292%	9.840%	10.292%	10.066%		View
140	144	0.000%	0.000%	0.000%	0.000%		View
140	143	2.618%	1.695%	2.618%	2.156%		View
140	142	4.712%	2.394%	4.712%	3.553%		View
140	141	19.895%	10.570%	19.895%	15.233%		View
144	139	10.274%	8.287%	10.274%	9.281%		View

To view student's log, click the student matric
To create new grouping, click View Groupings

1 2 3 4 5 6 7 8 9 10 11 Last

Figure 14: The pairwise-comparison-results-listing page.

The pairwise-comparison-results-listing page usually contains more than a hundred entries. Instead of showing the result in a long list like in MOSS and JPlag, the results are displayed fifteen per page (see Figure 14).

Furthermore, by default, results are ordered according to the maximum similarity value (MAX column in Figure 14). They may also be sorted according to other fields by clicking on the column headings. A tooltip prompting the user that the data can be sorted by that column will appear when the cursor is over the column heading. A triangle icon after the column heading shows the results are currently sorted according to that column.

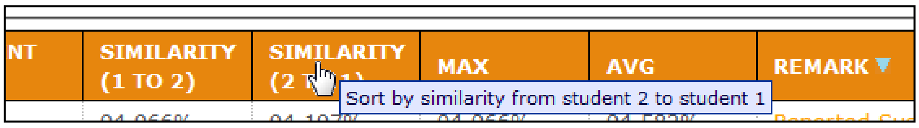


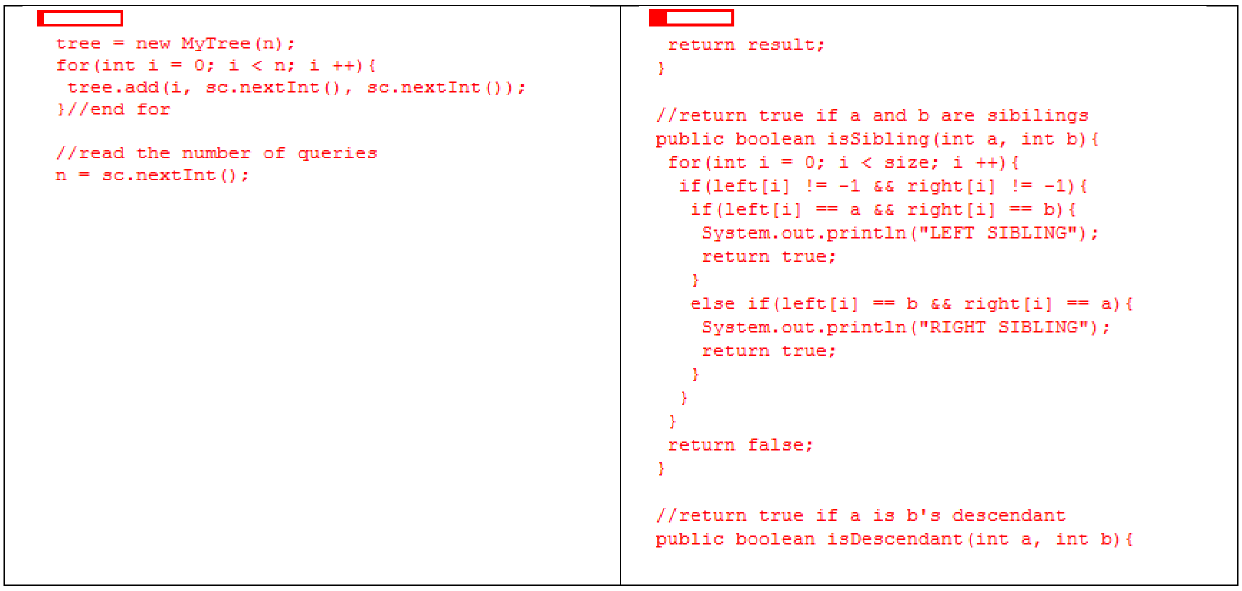
Figure 15: The mouse cursor changes into pointer when over the column heading "SIMILARITY (2 TO 1)" and a tooltip prompts the user that the data can be sorted by the column.

To top it up, a filtering function that allows the users to filter the results by a student's matriculation number is available on the top of the page. This provides fast reference to specific results. This is an essential improvement over the interfaces provided by MOSS and JPlag. Those interfaces require the user to search for the desired pairs of submissions by scrolling through the page or by calling up the "Find" function that is provided by most web browsers.

4.2. Pairwise Comparison Result Interface

As mentioned in Section 2, an advantage of the structure metric systems is that the comparison results can be displayed using different font colours. However, this display may confuse users when the number of matches is larger than the number of predefined colours in the system. For example, Figure 16 shows two different matched regions reported by MOSS that are displayed in the same font colour. There are six reported matched regions and in this case, the font colours were reused for every five matched regions in MOSS.

I take another approach to display regions of matching code. First, the interface should give an overview of all matched regions. Second, the user should be able to instruct the interface to display a specific matched region by clicking a link. This specific region should be easily identifiable. Third, the interface should display the regions that are part of a skeleton code. This feature, however, is not included in systems like MOSS and JPlag.



```
tree = new MyTree(n);
for(int i = 0; i < n; i ++){
 tree.add(i, sc.nextInt(), sc.nextInt());
}//end for

//read the number of queries
n = sc.nextInt();
```

```
 return result;
}

//return true if a and b are sibilings
public boolean isSibling(int a, int b){
 for(int i = 0; i < size; i ++){
  if(left[i] != -1 && right[i] != -1){
   if(left[i] == a && right[i] == b){
    System.out.println("LEFT SIBLING");
    return true;
   }
   else if(left[i] == b && right[i] == a){
    System.out.println("RIGHT SIBLING");
    return true;
   }
  }
 }
 return false;
}

//return true if a is b's descendant
public boolean isDescendant(int a, int b){
```

Figure 16: Different matched code regions reported by MOSS displayed in the same font colour.

Based on the points above, I defined four colours to display the different regions:

(1) Green – No identical region is found in the other submission of the pair.

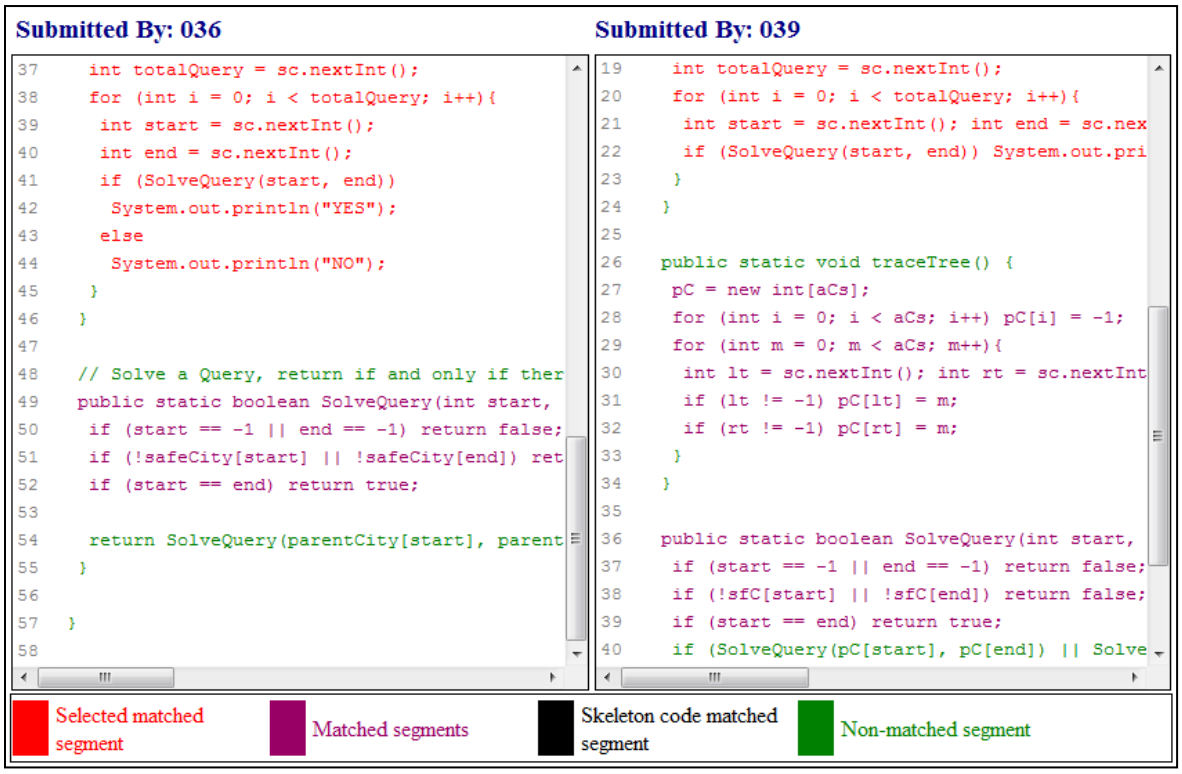
(2) Black – Regions are found to be parts of a skeleton code. These regions do not contribute to the computed similarity value.

(3) Dark Red – Regions are found to be identical in the pair of submissions. These regions contribute to the computed similarity value.

(4) Red – Dark Red region that is currently selected on the display.

Figure 17 shows a partial screen capture of the *Pairwise Comparison Result interface*. The interface allows the user to have a quick overview on the code segments found identical in both submissions (they are shown in dark red initially). To identify the exact identical regions, the user can click on the links provided in the Mapping List window (see Figure 18) to synchronize the displays to the desired matched region. That region will be coloured in red now to aid the user in identifying it easily. Click the line “19-22” (third row, last column) will result in the display of the identical region (in red) as shown in Figure 17.

SSID also provides links to report suspicious pairs, confirm plagiarism cases, and confirm guilty parties on the *Pairwise Comparison Result interface*. This allows the teaching team to mark out suspicious and confirmed plagiarism cases easily.



Submitted By: 036

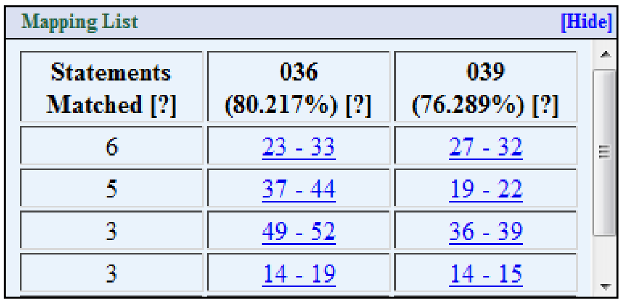
```
37     int totalQuery = sc.nextInt();
38     for (int i = 0; i < totalQuery; i++){
39      int start = sc.nextInt();
40      int end = sc.nextInt();
41      if (SolveQuery(start, end))
42       System.out.println("YES");
43      else
44       System.out.println("NO");
45     }
46    }
47
48    // Solve a Query, return if and only if ther
49    public static boolean SolveQuery(int start,
50     if (start == -1 || end == -1) return false;
51     if (!safeCity[start] || !safeCity[end]) ret
52     if (start == end) return true;
53
54     return SolveQuery(parentCity[start], parent
55    }
56
57   }
58
```

Submitted By: 039

```
19     int totalQuery = sc.nextInt();
20     for (int i = 0; i < totalQuery; i++){
21      int start = sc.nextInt(); int end = sc.nex
22      if (SolveQuery(start, end)) System.out.pri
23     }
24    }
25
26    public static void traceTree() {
27     pC = new int[aCs];
28     for (int i = 0; i < aCs; i++) pC[i] = -1;
29     for (int m = 0; m < aCs; m++){
30      int lt = sc.nextInt(); int rt = sc.nextInt
31      if (lt != -1) pC[lt] = m;
32      if (rt != -1) pC[rt] = m;
33     }
34    }
35
36    public static boolean SolveQuery(int start,
37     if (start == -1 || end == -1) return false;
38     if (!sfC[start] || !sfC[end]) return false;
39     if (start == end) return true;
40     if (SolveQuery(pC[start], pC[end]) || Solve
```

🟥 Selected matched segment 🟪 Matched segments ⬛ Skeleton code matched segment 🟩 Non-matched segment

Figure 17: Partial screen capture of *Pairwise Comparison Result interface*. Submission displayed on the left is from student 036 and submission displayed on the right is from student 039. Code in red denotes the matched regions currently focused on; code in dark red denotes matched regions; code in green denotes no identical region is found in the other submission.



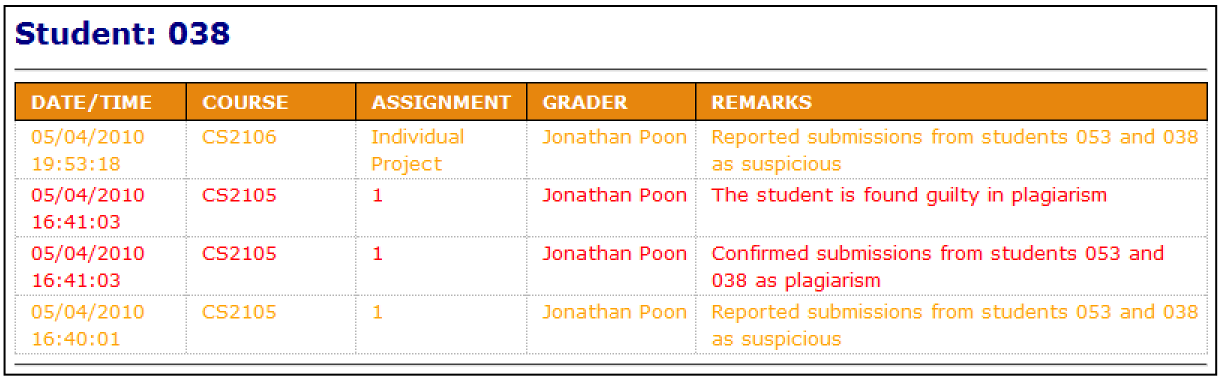
Mapping List [Hide]

Statements Matched [?]	036 (80.217%) [?]	039 (76.289%) [?]
6	23 - 33	27 - 32
5	37 - 44	19 - 22
3	49 - 52	36 - 39
3	14 - 19	14 - 15

Figure 18: Screen capture of list of matched regions. The first column shows the number of statements matched in each region. The second and third columns denote the lines of code in each region in the submissions of student 036 and student 039 respectively.

4.3. Log System Interface

One of the major differences between SSID and other similar systems is that it 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.



Student: 038

DATE/TIME	COURSE	ASSIGNMENT	GRADER	REMARKS
05/04/2010 19:53:18	CS2106	Individual Project	Jonathan Poon	Reported submissions from students 053 and 038 as suspicious
05/04/2010 16:41:03	CS2105	1	Jonathan Poon	The student is found guilty in plagiarism
05/04/2010 16:41:03	CS2105	1	Jonathan Poon	Confirmed submissions from students 053 and 038 as plagiarism
05/04/2010 16:40:01	CS2105	1	Jonathan Poon	Reported submissions from students 053 and 038 as suspicious

Figure 19: Screen capture of *Log System interface*.

In a sample scenario, student ‘038’ is under investigation by *teaching staff* from the module CS2106 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* (Figure 19), the *teaching staff* can see that ‘038’ has been found guilty of plagiarizing another submission from ‘053’ in another module CS2105, in the same semester. Therefore, the *teaching staff* can conclude that student ‘038’ has probably committed plagiarism in CS2106 as well.

4.4. Visual Interfaces

Another major improvement available in SSID is visual support for the accumulation of plagiarism evidence and for aiding in the improvement of education strategies.

In this section, the six visual interfaces from SSID are presented through scenarios that show how a user can interpret each one of them to obtain the desired information.

Scenario 1 – Detecting and Displaying Plagiarism Clusters

> *After submitting the first assignment for pairwise comparison and investigating the plagiarism cases, the teaching team wants to know the plagiarism clusters.*

This can be done by defining a plagiarism grouping. A plagiarism grouping consists of plagiarism clusters based on a cut off criterion value in percentage. SSID provides three options for defining the cut off criterion value based on: (1) the similarity values among confirmed plagiarism pairs in submissions; (2) the similarity values among suspicious or confirmed plagiarism pairs in submissions; (3) a user-defined cut off criterion.

For (1), the cut off criterion for the clustering algorithm as described in Section 3.1.3 will be based on the smallest maximum similarity value amongst the confirmed plagiarism pairs in the assignment. In other words, clusters will contain only students with pairs of submissions that have a maximum similarity value higher than or equal to the smallest maximum similarity value in the confirmed plagiarism pairs. (2) is similar to (1), where the cut off criterion is instead based on the smallest maximum similarity value amongst the suspicious or confirmed plagiarism pairs. For (3), the cut off criterion is based on a user-defined value; the pair of submissions that have a maximum similarity value higher than the cut off criterion will be grouped in a plagiarism cluster.

Once a plagiarism grouping is defined, plagiarism clusters can be displayed in the *Clusters Summary interface*. For example, Figure 20 shows the (partial) *Clusters Summary interface* which consists of students ‘028’, ‘035’, and ‘048’. Each vertex in the interface represents a student and each edge represents the maximum similarity value of the submission pair between two students. In addition, Edges are represented with different colours and thickness, SSID has six default similarity boundaries that ranging from less than 50% (green colour) to higher than or equal to 90% (red colour). From the interface, one can conclude that a learning group can be set for the students ‘028’, ‘035’, and ‘048’. Since the edges connected to ‘028’ are thicker than the one connecting ‘035’ and ‘048’, it is highly possible that ‘035’ and ‘048’ have copied the from ‘028’.

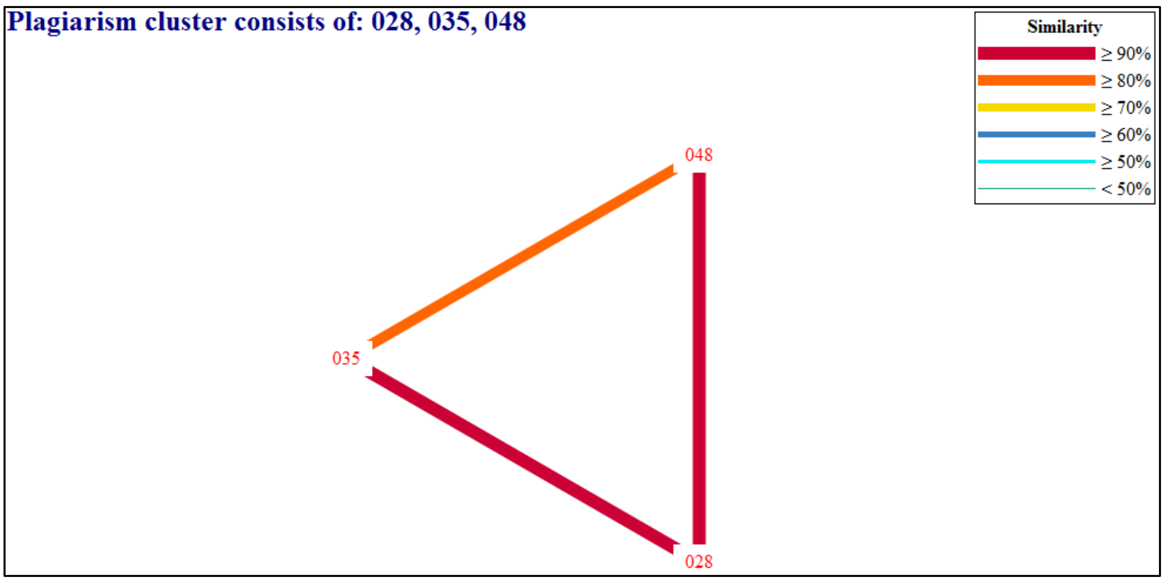
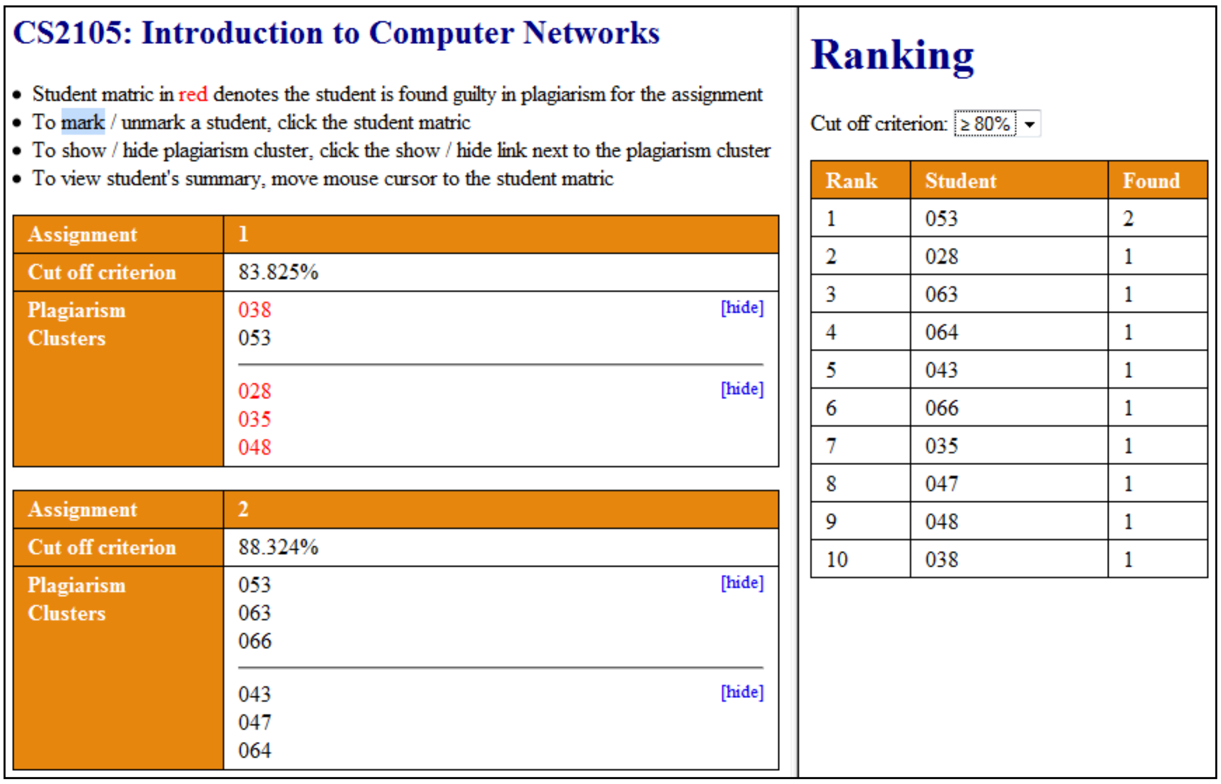


Figure 20: Example of *Clusters Summary interface* (partial). Vertices denote students’ matrics; edges denote the maximum similarity value between two students’ submission. Both colouring and line weighting are used to differentiate similarity values in range.

Scenario 2 – Obtaining an Overview on Plagiarism Activities in Assignments

After checking a number of assignments through SSID, the teaching team wants to know the statistics on the students whom have been plagiarizing most of the time.

This can be done through the *Assignments interface*. The *Assignments interface* (Figure 21) provides a summary of plagiarism clusters within assignments. Furthermore, the interface also provides a list of the top 10 students in plagiarism clusters with a cut off criterion that is higher than or equal to the five values (50%, 60%, 70%, 80%, and 90%). Moreover, a summary of the statistics for each student displayed in the overview is available by moving the mouse curse over the student's matriculation number.


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%

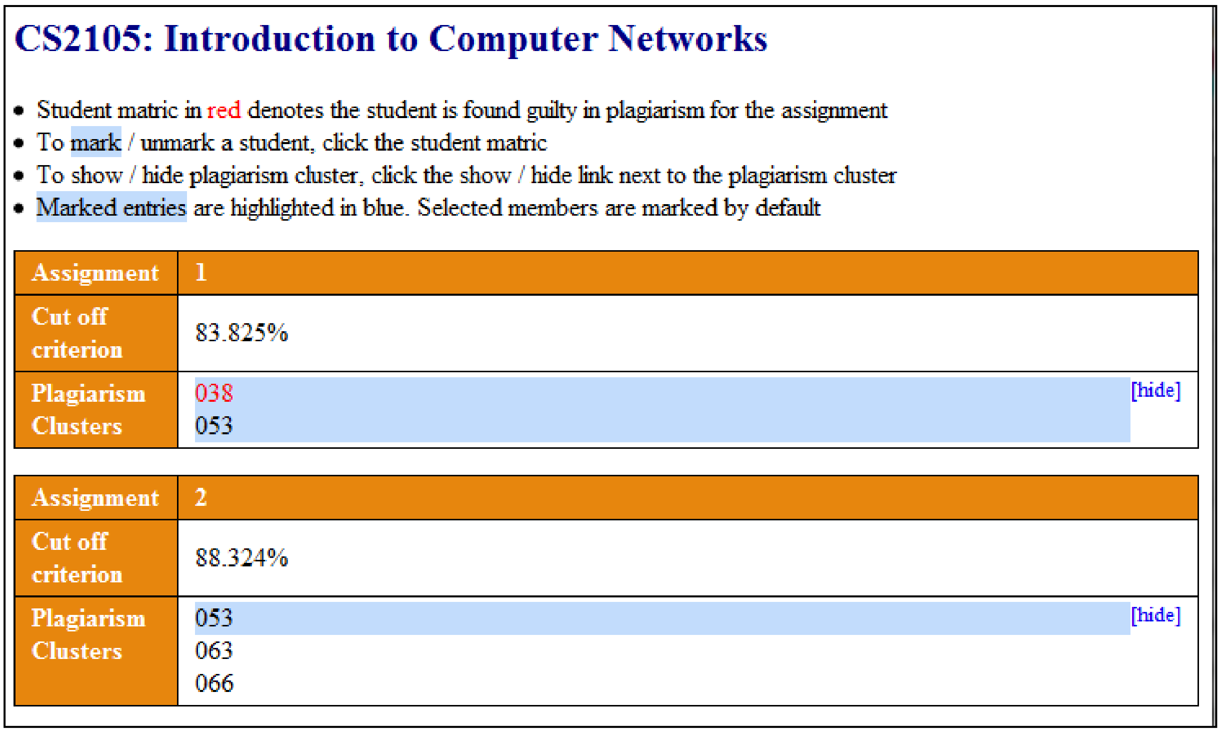
Rank	Student	Found
1	053	2
2	028	1
3	063	1
4	064	1
5	043	1
6	066	1
7	035	1
8	047	1
9	048	1
10	038	1

Figure 21: Example of *Assignments interface*. The left frame shows the overview of each plagiarism clusters in assignments. The right frames show the statistics of top 10 students being detected in a plagiarism cluster with the cut off criterion higher than or equal to 80%.

Scenario 3 – Checking Accomplices over Assignments

To improve the learning experience in a learning group, the teaching team wants to know if the learning group helps to prevent students in further plagiarism activities.

SSID provides a visual interface call *Member Movement*. After selecting a plagiarism cluster as the initial cluster, SSID will use the initial cluster to show other plagiarism clusters that also contain the students from the initial cluster.


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
- Marked entries are highlighted in blue. Selected members are marked by default

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

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

Figure 22: Example of *Member Movement interface* after selecting the plagiarism cluster for students 038 and 053 in assignment 1 as the initial cluster.

Figure 22 shows an example on the *Member Movement interface* with student ‘038’ and ‘053’ as the initial cluster. From the information in this interface, the teaching team can easily tell that the learning group is quite efficient, as ‘038’ has not been found plagiarizing another’s work in assignment 2. On the other hand, ‘053’ has formed another plagiarism cluster with ‘063’ and ‘066’. The teaching team can conclude that they have to pay more attention to student ‘053’.

Scenario 4 – Viewing the Similarity Values between Individuals

Suppose that in scenario 3, the teaching team also wants to verify if by any chance student ‘038’ could be in a plagiarism cluster with the group of students ‘053’, ‘063’, and ‘066’.

To address this issue, SSID provides the *Between Individual interface* (see Figure 23). After selecting the desired group of students, the similarity values of their submission pairs are displayed in a visual format.

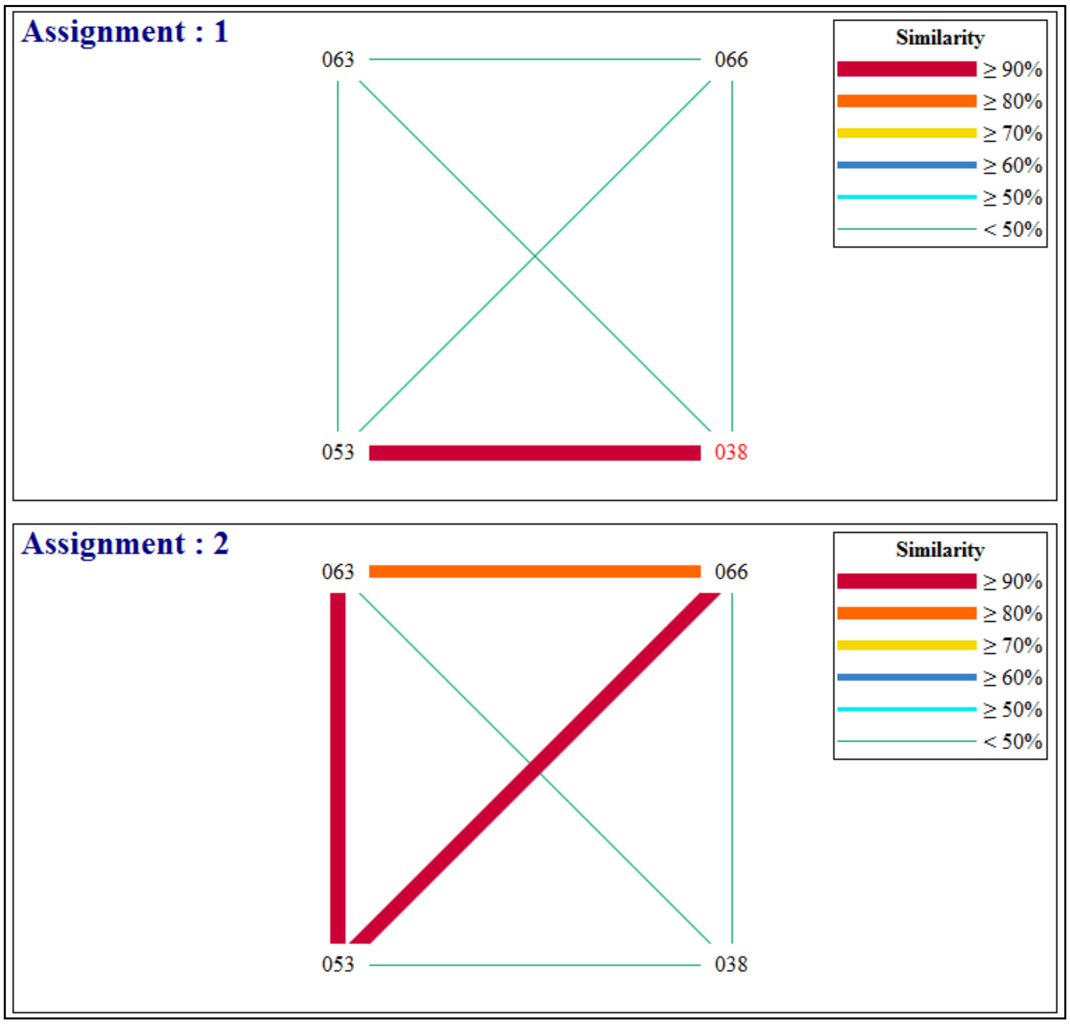


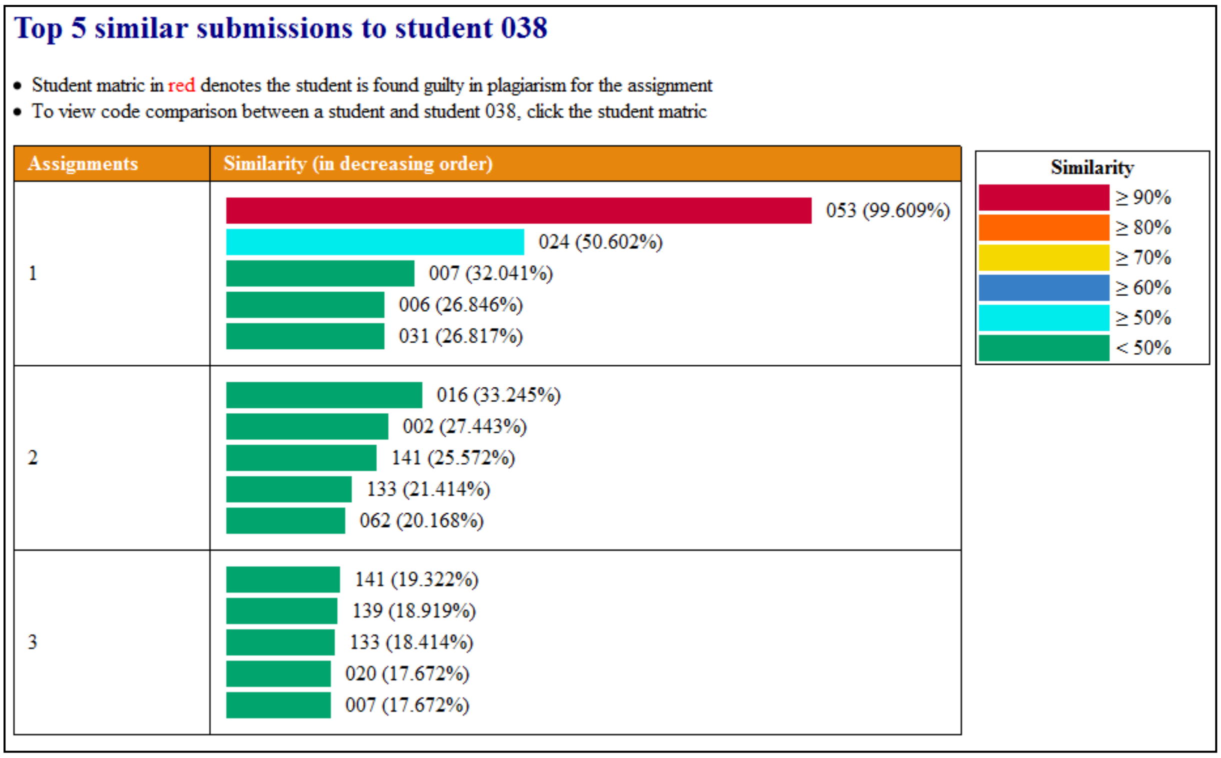
Figure 23: Example of partial *Between Individual interface* (partial).

Figure 23 shows a partial screen capture of the *Between Individual interface*. From the figure, the similarity values between the submissions from ‘038’ and the submissions from ‘063’ and ‘066’ are below 50% for both assignments. Also, ‘038’ has a similarity value of 90% with ‘053’ in assignment 1, but the similarity value dropped to less than 50% in assignment 2. This serves as a strong evidence to back up the fact that ‘038’ has probably stopped plagiarizing the work of others.

Scenario 5 – Obtaining the Most Similar Submissions to an Individual’s

> *From scenario 4, the teaching team has been monitoring student ‘038’ for weeks and they want to know if ‘038’ has been able to keep away from plagiarizing the work of others.*

SSID provides the *Top Similarities interface* for this purpose. After selecting the desired student and entering the number of candidates (*K*) to display, the *K* students are paired up with ‘038’ for similarity checks. These are displayed in a descending order of similarity values for each assignment. By default, the value of *K* is set as **5**.


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 24: An example of the *Top Similarities interface*. The top 5 similar submissions in each assignment for student 038 are displayed.

Figure 24 shows an example of the *Top Similarities interface*. The teaching team has chosen to display the top 5 similar submissions in each assignment for ‘038’. From the example, since there are no similarity values greater than 50% in assignment 2 and 3, the teaching team can conclude that after assignment 1, ‘038’ has been working on his own.

Scenario 6 – Obtaining the Plagiarism Clusters an Individual Belongs

As a continuation from scenario 3, the teaching team has been paying more attention to ‘053’ as the student seems to have been disclosing his or her submissions to other students. After generating the plagiarism clusters for assignment 3, the teaching team wants to know all clusters that ‘053’ belongs to.

By using the *Clusters for a Student interface*, the teaching team can view all the plagiarism clusters that a particular student is in. For example, after selecting ‘053’, the *Clusters for a Student interface* (partial screen capture) is as shown in Figure 25.

From Figure 25, the teaching team can identify that student ‘053’ has been involved in plagiarism clusters for all assignments. It shows that student ‘053’ is consistently involved in

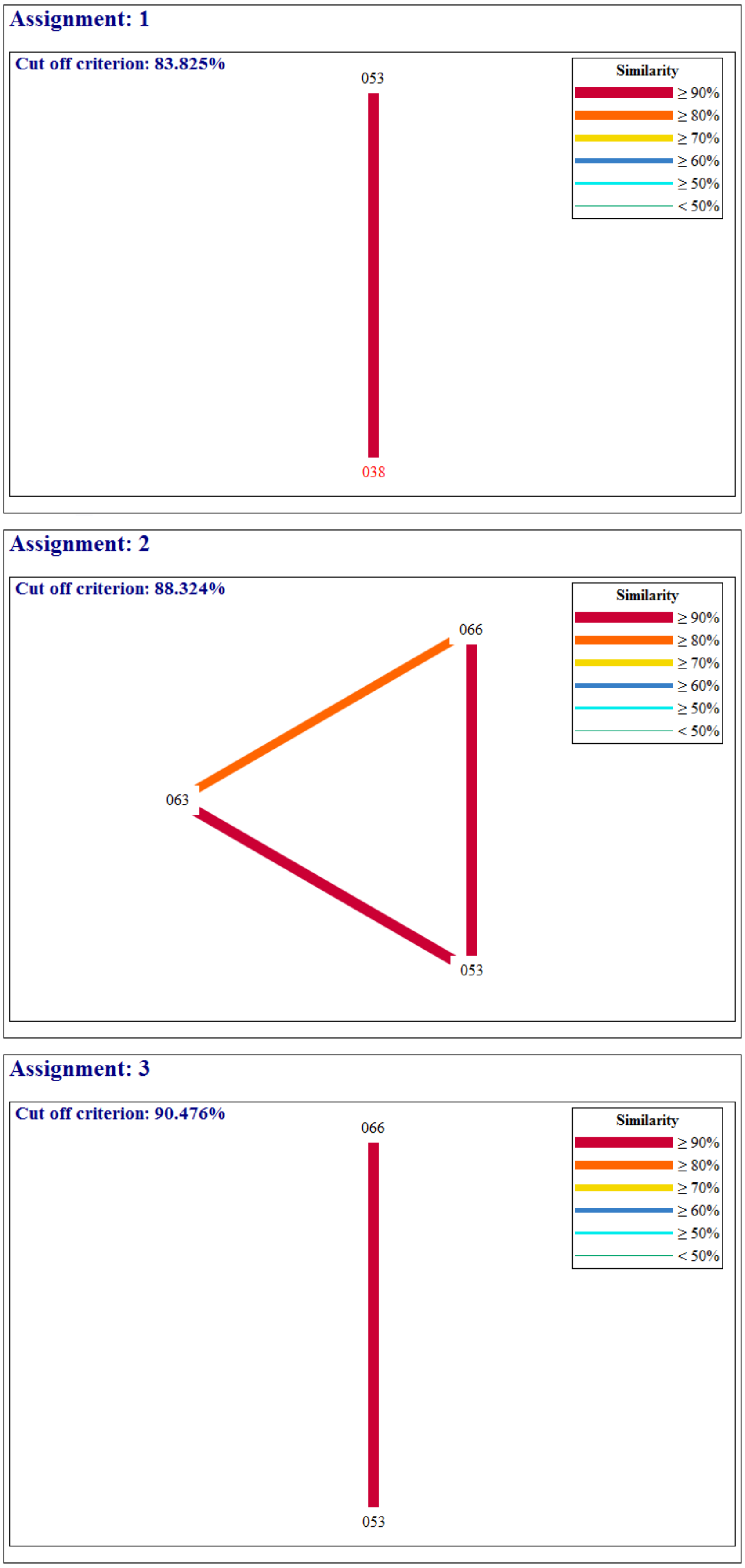


Figure 25: An example of the *Clusters for a Student interface*.

plagiarism, and disciplinary action should be taken against this student. Moreover, submissions from student ‘066’ submissions are highly similar (at least 90% in the similarity measure) to submissions from student ‘053’ in assignments 2 and 3. This shows that the team may need to monitor student ‘066’ as well.

4.5. Development Process and Evaluation

In the development process of the SSID user interface, several designs were implemented and one user review session was executed. The first prototype was completed six months from the start of the project; the second prototype was completed four months later; the user review session was held right after the second prototype; the final product was completed two weeks after.

The first prototype had a messy layout, allowed no access to crucial pages via shortcuts and no breadcrumbs for navigation. Furthermore, the pairwise-comparison-results-listing page was not implemented with sorting and filtering functions and the results are listed in a single page. The user had a hard time navigating through the system. Moreover, the visual interfaces were not clear as no legend was available. This makes the users difficult to interpret the visuals.

The second prototype was improved with implementations of legends in the visual interfaces, highlighting the current row in the tables in the *Module Management interface,* and adding in the sorting and filtering functions to the pairwise-comparison-results-listing page.

After the second prototype was completed, a user review session was held. Five participants from the School of Computing, National University of Singapore were invited for the session: an Associate Professor who has been teaching for more than seven years, a research fellow, a postgraduate student, and two undergraduate students. They were assigned different roles for three modules and were provided with ten set of sample submissions. Each set which contained more than 130 submissions, included a number of plagiarized submissions. Some suggestions from the participants to improve user accessibility are as follows:

(1) For visual interfaces like *Clusters Summary* where edges are used to denote the maximum similarity value between the submissions from two students, it would be convenient if the *Pairwise Comparison Result interface* is displayed when clicking the edge.

(2) For visual interfaces like *Clusters Summary*, usage of unidirectional edges and the asymmetric similarity values in the visual may provide more information on the plagiarism activities among the students in a plagiarism cluster.

(3) For students who have no history of plagiarism, it is recommended to have the links to their logs marked with special icons that would indicate the absence of plagiarism history of these students.

(4) The layout for the visual interfaces should follow the layout provided in the *Module Management interface* for coherence.

(5) When scrolling through the submissions in the *Pairwise Comparison Result interface*, it would be convenient if both submissions could be scrolled together while holding a special key.

These suggestions would greatly improve the user accessibility. The full list of suggestions is available in Figure 43.

Based on the suggestions from the user review session, the development to the final product is concentrated on the user accessibility and webpage design. Improvements from the second prototype include the implementations of breadcrumbs and shortcut access to subpages.

5. Conclusion

I have presented a plagiarism detection system that assists instructors not only in detecting plagiarism between a pair of students, but also in discovering the plagiarism clusters in an assignment.

For user interfaces, SSID provides essential features such as a log system and visual interfaces that are useful in preventing plagiarism but are not available in the existing systems. Furthermore, the system provides links for users to report suspicious pairs, confirm plagiarism cases and declare the guilty parties. These are not available in state-of-the-art systems like JPlag and MOSS.

Regarding plagiarism detection, SSID shows a prefect separation between the original and plagiarism pairs in my experimental data. The *Greedy-String-Tiling* algorithm is extended to support the exclusion of skeleton code given by instructors from the pairwise comparison of submissions. Moreover, efficiency of the algorithm is improved by comparing selective *N*-grams rather than all *N*-grams as described in JPlag.

Limitations

While this final year project is coming to an end, I have not deployed the system in a real life situation. This results in the lack of feedback on how functionality and interfaces of SSID can be improved to assist academic staff during course time. Moreover, although SSID shows a perfect separation between the plagiarism and original pairs among experimental data, there are *attacks* that can always *confuse* the system successfully. In order to achieve better defense against plagiarism, improvement on pairwise comparison of submissions is essential.

Future Work

The future direction for SSID would focus on two parts: (1) improvement on pairwise comparison, and (2) enhancement on user accessibility for the user interface.

To improve on pairwise comparison, I am planning to introduce approximate matching instead of exact matching to the system. For enhancing user accessibility, in order to receive user review in real life situation, SSID is scheduled to detect the plagiarism activities against the take-home practical examination for the module CG1102. The examination will be conducted a week after the submission of this report. If I were to continue this project, I would refine the system until the processes of detecting plagiarism activities, providing summaries and providing evidence on plagiarism activities and clusters are fully automated. Perhaps they could all be done with just a single click.

Bibliography

Aiken, A. (1994). *A System for Detecting Software Plagiarism*. Retrieved April 1, 2010, from Standford Theory: http://theory.stanford.edu/~aiken/moss/

Berghel, H. L., & Sallach, D. L. (1984). Measurements of program similarity in identical task environments. *SIGPLAN Not. , 19* (8), 65-76.

Ciesielski, V., Wu, N., & Tahaghoghi, S. (2008). Evolving similarity functions for code plagiarism detection. *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation* (pp. 1453-1460). Atlanta, GA, USA: ACM.

Donaldson, J. L., Lancaster, A.-M., & Sposato, P. H. (1981). A Plagiarism Detection System. *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education* (pp. 21-25). St. Louis, Missouri, United States: ACM.

Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96).* AAAI Press.

Faidhi, J. A., & Robinson, S. K. (1987). An empircal approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ. , 11* (1), 11-19.

Gitchell, D., & Tran, N. (1999). Sim: a utility for detecting similarity in computer programs. *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education* (pp. 266-270). New Orleans, Louisiana, United States: ACM.

Grier, S. (1981). A tool that detects plagiarism in Pascal programs. *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education* (pp. 15-20). St. Louis, Missouri, United States: ACM.

Halstead, M. H. (1977). *Elements of Software Science.* Elsevier Science Ltd.

Jadalla, A., & Elnagar, A. (2008). PDE4Java: Plagiarism Detection Engine for Java source code: A clustering appoarch. *Int. J. Bus. Intell. Data Min. , 3* (2), 121-135.

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

Liu, C., Chen, C., Han, J., & Yu, P. S. (2006). GPLAG: Detection of Software Plagiarism by Program. *Proceedings of the Twelfth ACM SIGKDD International Conference* (pp. 872-881). Philadelphia, PA, USA: ACM.

Moussiades, L., & Vakali, A. (2005). PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. *The Computer Journal , 48*, 651-661.

Ooi, W. T., & Tan, T. C. (2005). A Survey on Awareness and Attitudes towards Plagiarism among Computer Science Freshmen. *CDTLink , 9* (3), 3, 11.

Ottenstein, K. J. (1977). An Algorithmic Approach to the Detection and Prevention of Plagiairsm. *SIGCSE Bull. , 8* (4), 30-41.

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding Plagiairsms among a Set of Programs with JPlag. *Journal of Universal Computer Science , 8* (11), 1016-1038.

Prechelt, L., Malpohl, G., & Phlippsen, M. (2000). *JPlag: Finding plagiarisms among a set of programs.* Technical Report 2000-1, University of Karlsruhe, D-76131 Karlsruhe, Germany.

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. *Proceedings of the 2003 ACM SIGMOD international Conference on Management of Data.* San Diego, California, June 09 - 12, 2003: ACM, New York, NY, 76-85.

Sheard, J., Carbone, A., & Dick, M. (2003). Determination of Factors which Impact on IT Students' Propensity to Cheat. *Proceedings of the fifth Australasian conference on Computing education. 20*, pp. 119-126. Adelaide, Australia: Australian Computer Society, Inc.

Vamplew, P., & Dermoudy, J. (2005). An Anti-Plagiarism Editor for Software Development Courses. *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42. A. Young and D. Tolhurst, Eds. ACM International Conference Proceeding Series, vol. 106*, pp. 83-90. Newcastle, New South Wales, Australia: Australian Computer Society, Inc.

Verco, K. L., & Wise, M. J. (1996). Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. *Proc. of 1st Australian Conference on Computer Science Education* (pp. 86-95). ACM.

Wise, M. J. (1993, December). *String similarity via greedy string tiling and running Karp-Rabin matching.* Retrieved March 1, 2010, from Dept. of CS, University of Syndney: ftp://ftp.cs.su.oz.au/michaelw/doc/RKR_GST.ps

Wise, M. J. (1996). YAP3: Improved Detection of Similairties in Computer Program and Other Texts. *SIGCSE Bull. , 28* (1), 130-134.

Appendix A

```
Improved-Greedy-String-Tiling(Submission A, Submission B, Skeleton S) {
   tiles = {};
   do {
      matches = {};

      //Phase 1
      maxMatch = Find-Match-Statements(A, B, matches);

      //Phase 2
      Verify-And-Mark-Matches(A, B, S, matches);

   } while (maxMatch > L);
   return tiles;
}
```

Figure 26: Overview of *Improved-Greedy-String-Tiling* algorithm

Figure 26 shows the overview of my algorithm. The algorithm consists two phases:

1) Finding matches with the maximum number of statements and return the number to the algorithm,
2) Verifying and marking matches.

The algorithm repeatedly executes the two phases to find the matched regions in the submissions in decreasing number of contiguous exactly matched statements, until no more matches satisfying the *Minimum Match Length* (*L*) can be found. Then, the algorithm terminates by returning *tiles*, the set of matches found.

```
Find-Match-Statement(Submission A, Submission B, Matches matches) {
   maxMatch = L;
   Forall unmarked and beginning-of-a-statement tokens $A_a$ in A {
      $N_a$ = the N-gram starting with token $A_a$;
      if (Hash table $B_{index}$ in B contains $N_a$) {
         $B_{list}$ = List of indices represented by $B_{index}[N_a]$;
         Forall unmarked tokens $B_b$ in B with the indices in $B_{list}$ {
            j = 0; curMatch = 0; e = 0;
            while ($A_{a+j}$ == $B_{b+j}$ && not_marked($A_{a+j}$) && not_marked($B_{b+j}$)) {
               if ($A_{a+j}$ is an end of statement token) {
                  curMatch++;
                  e = a+j;
               }
               j++;
            }
            if (curMatch == maxMatch)
               matches = matches ∪ (a, b, e, curMatch);
            else if (curMatch > maxMatch) {
               matches = {(a, b, e, curMatch)};
               maxMatch = curMatch;
            }
         }
      }
   }
   return
}
```

Figure 27: Phase 1 – Finding matches with the maximum number of statements and return the number to the algorithm

Figure 27 shows the details of phase 1. The quadruple (*a*, *b*, *e*, *k*) denotes a match between the substrings of submission *A* and submission *B* beginning from the *a*th token in *A* and *b*th token in *B* respectively, with length of *e* and consists of *k* statements. Phase 1 begins with iterating through every unmarked token A_a in *A* which is also at the beginning of a statement, and finding an identical token in *B* through indexing by *N*-grams (lines 2 to 5). Then, the algorithm extends the match by comparing the tokens followed (lines 6 to 14). Tokens compared must not be marked previously (neither match marked nor skeleton marked). If a token is marked as the termination of a statement, the algorithm increments the number of matches (*curMatch*) and records the length of the match (*e*) in number of tokens. As soon as two tokens are found to be different or either one of them is marked, a maximum of *curMatch* is found. The algorithm checks if *curMatch* contains the same number of statements as the maximum match found (*maxMatch*). If *curMatch* is equal to *maxMatch*, the match (*a*, *b*, *e*, *curMatch*) is added to the set *matches* (line 16). However, if *curMatch* contains more statements than *maxMatch*, *matches* is emptied and inserted with the match (*a*, *b*, *e*, *curMatch*) (line 19), and *maxMatch* is set to *curMatch* (line 20). Finally, the phase terminates by returning *maxMatch* (line 21).

Once phase 1 has terminated, *matches* contains all the maximal matches. However, the set will be empty if no match contains at least the number of statements as specified by *L*.

```
1)  Verify-And-Mark-Matches(Submission A, Submission B, Skeleton S,
2)     Matches matches) {
3)     Forall (a, b, e, k) ∈ matches {
4)        if (Any token in {$A_a$ ... $A_{a+e}$} & {$B_b$ ... $B_{b+e}$} is marked)
5)           continue;
6)        if (S != NULL) {
7)           Submission A′ = [$A_a$ ... $A_{a+e}$];
8)           skeletonTiles = Improved-Greedy-String-Tiling(A′, S, NULL);
9)
10)          skeletonStatements = 0;
11)          Forall (a, b, e, k) ∈ skeletonTiles
12)             skeletonStatements += k;
13)
14)          if (skeletonStatements >= k - L)
15)             mark($A_a$ ... $A_{a+e}$ & $B_b$ ... $B_{b+e}$) as skeleton marks;
16)          else {
17)             mark($A_a$ ... $A_{a+e}$ & $B_b$ ... $B_{b+e}$) as match marks;
18)             tiles = tiles ∪ (a, b, e);
19)          }
20)       }
21)       else {
22)          mark($A_a$ ... $A_{a+e}$ & $B_b$ ... $B_{b+e}$) as match marks;
23)          tiles = tiles ∪ (a, b, e);
24)       }
25)    }
26) }
```

Figure 28: Phase 2 – Verifying and marking matches

The detail of phase 2 is shown in Figure 28. The phase begins with checking each *match* in *matches* if the ranges of tokens specified in *match* for *A* and *B* is marked. If a token within the specified range is marked, the algorithm will omit the *match*; otherwise, the algorithm checks if the match can be found in the skeleton code *S* supplied. The check is done by constructed the sequence of tokens for *A* specified in *match* as a new submission *A'* and make a recursive call to the *Improved-Greedy-String-Tiling* algorithm with *A'*, *S* and *NULL* as parameters (lines 7 to 8). Since these are codes comparing the matched tokens and the skeleton code, the algorithm provides no skeleton code for the comparisons. Once the comparison has finished, the algorithm computed the value of *skeletonStatements*, which denotes the sum of statements matched in all returned matches (lines 10 to 11). If $\textit{skeletonStatements} \geq k - L$, the match is considered obtained from the skeleton and the ranges of tokens in *A* and *B* are marked as skeleton marks (lines 14 to 15). If not, it means that the match is original from the skeleton and the match is valid; tokens specified in the match are marked as match marks (lines 16 to 18) and the match is added into the set *tiles*. Furthermore, in cases where there exists no skeleton code, the algorithm marks the ranges of tokens specified in the match as match marks (lines 21 to 23) and the match is added into the set *tiles*. When each of the matches specified in *matches* is considered, the phase terminates.

Appendix B

Assignment 1: Perilous cities **Number of Submissions**: 91			
Description A country with *N* cities connected in the manner of a binary tree with its root being the capital city (numbered as 0). As terrorism spreads over the country, some cities are considered unsafe for travelers, both to go to or to go through. Given this information, you have to answer queries from the travelers: Can I travel safely from city A to city B?			
Sample 1: 184 lines	**Sample 2:** 183 lines	**Sample 3:** 182 lines	**Sample 4:** 58 lines

Figure 29: Perilous cities assignment

Assignment 2: Binary tree genealogy **Number of Submissions**: 90			
Description The relationship between two nodes in a binary tree can be either parent-child, child-parent, left sibling-right sibling, right sibling-left sibling, or none of these cases. While implementing an a program that manipulates a binary tree data structure, you note that it would be helpful to have methods that determine the relationship between pairs of nodes. In this exercise, you are to write a program that takes in a binary tree and several queries. Each query contains the labels of two nodes in the tree. Your program should to return the relationship of the pair of nodes indicated in the query.			
Sample 1: 120 lines	**Sample 2:** 161 lines	**Sample 3:** 154 lines	**Sample 4:** 106 lines

Figure 30: Binary tree genealogy assignment

Assignment 1: Binary tree traversal **Number of Submissions**: 178			
Description You have learned that a binary tree can be reconstructed given its pre-order and in-order sequence. In this exercise, you are given the pre-order and post-order sequences, and asked to reconstruct the tree.			
Sample 1: 103 lines	**Sample 2:** 52 lines	**Sample 3:** 90 lines	**Sample 4:** 137 lines

Figure 31: Binary tree traversal assignment

Appendix C

Table 4: Similarities between the original and plagiarized submissions

Participant id	Assignment 1	Assignment 2	Assignment 3
1-1	93.532%	86.413%	81.932%
1-2	92.238%	99.638%	99.463%
1-3	97.801%	99.638%	99.463%
1-4	92.109%	95.969%	81.475%
1-5	68.474%	57.026%	50.503%
1-6	69.261%	73.913%	91.273%
1-7	99.741%	93.841%	94.280%
1-8	95.213%	80.072%	67.979%
1-9	68.325%	66.667%	37.873%
2-1	99.603%	99.191%	99.286%
2-2	94.974%	91.909%	56.190%
2-3	93.651%	99.191%	99.286%
2-4	85.981%	99.024%	67.308%
2-5	99.603%	99.191%	99.286%
2-6	95.899%	99.191%	99.286%
3-1	84.538%	83.137%	99.639%
3-2	93.481%	91.382%	87.004%
3-3	88.095%	93.716%	98.192%
3-4	94.868%	86.176%	88.989%
3-5	94.313%	99.461%	84.657%
3-6	91.262%	99.461%	99.639%
4-1	89.256%	99.288%	99.727%
4-2	90.323%	67.794%	82.645%
4-3	99.462%	99.288%	99.727%
4-4	85.484%	77.224%	78.962%
4-5	72.554%	96.846%	98.634%
4-6	72.849%	82.500%	64.787%
4-7	73.961%	68.073%	95.342%

The participant id is in the format: {Sample}-{Index}.

Appendix D

Table 5: Mean and standard deviation for the similarities between the original and plagiarized submissions

Assignment	Mean	Standard Deviation
Assignment 1	88.459%	10.240%
Assignment 2	88.758%	12.601%
Assignment 3	85.815%	17.221%
All	87.677%	13.565%

Appendix E

Each graph is a combination of 7 subgraphs: each describes the similarity distribution for an assignment over one *N* and one *L*. Solid blue lines represent the submission pairs in ***ORG***; dotted red lines represent the submission pairs in ***PLAG***.

The *x-axis* denotes the **similarity values** of a pair (0 to 100%). The similarity values used are based on the higher value between the two asymmetric similarity values. Furthermore, the resulting value is rounded to the type of integer.

The *y-axis* denotes the **ratio** of the number of pairs found in respective similarity value to the maximum number of pairs found in all similarity values of a series.

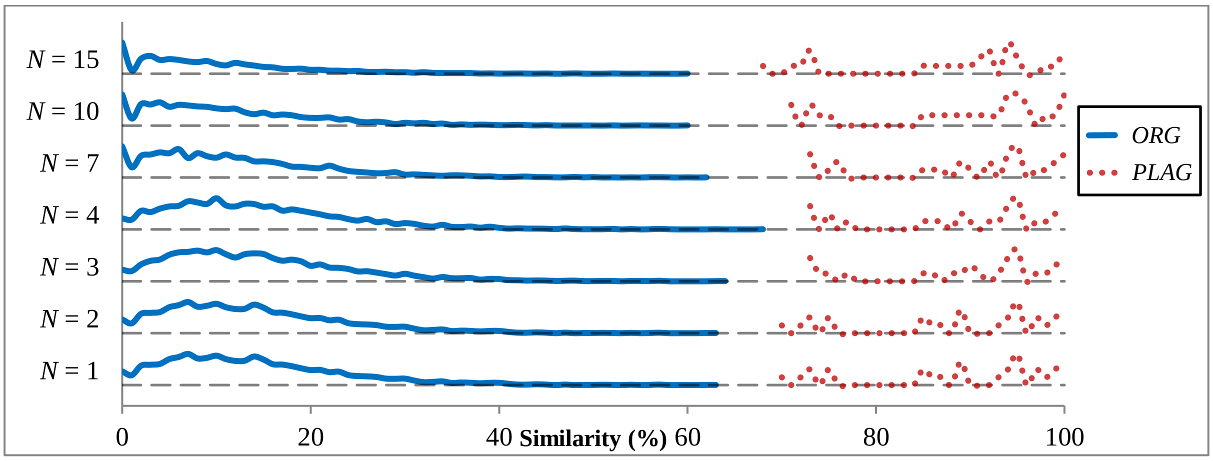


Figure 32: Similarity distribution graph for *Perilous Cities* assignment over different values of *N* with *L* = 1

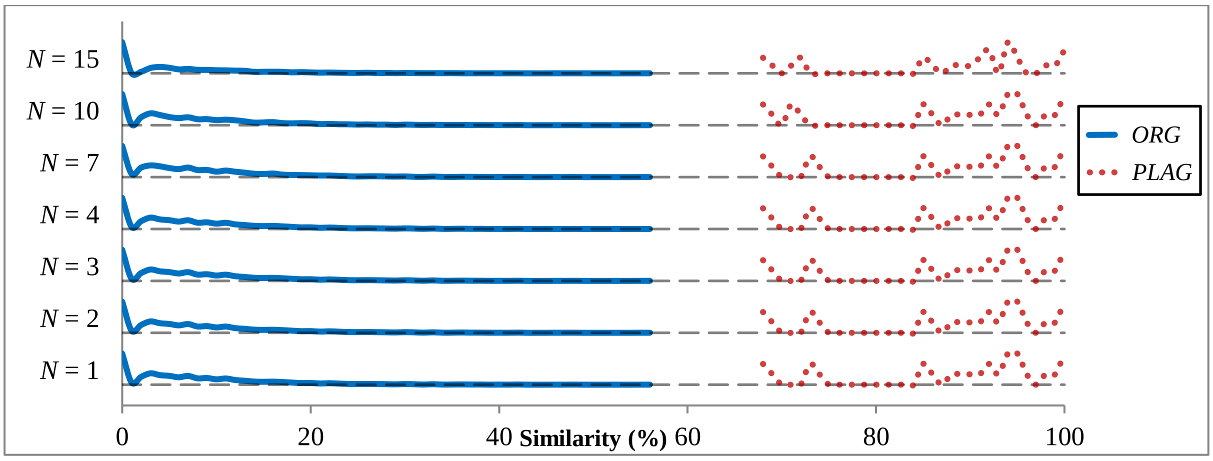


Figure 33: Similarity distribution graph for *Perilous Cities* assignment over different values of *N* with *L* = 2

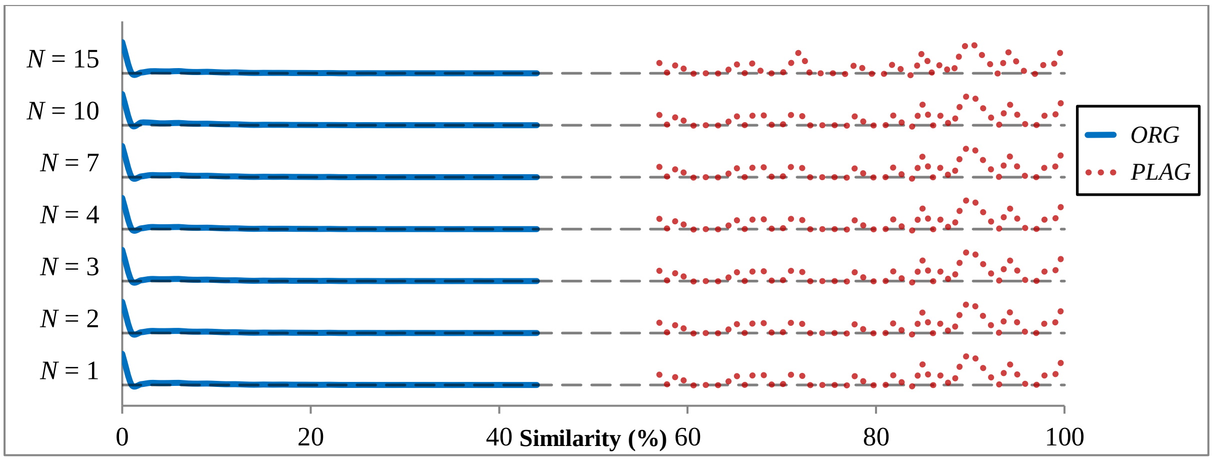


Figure 34: Similarity distribution graph for *Perilous Cities* assignment over different values of *N* with *L* = 3

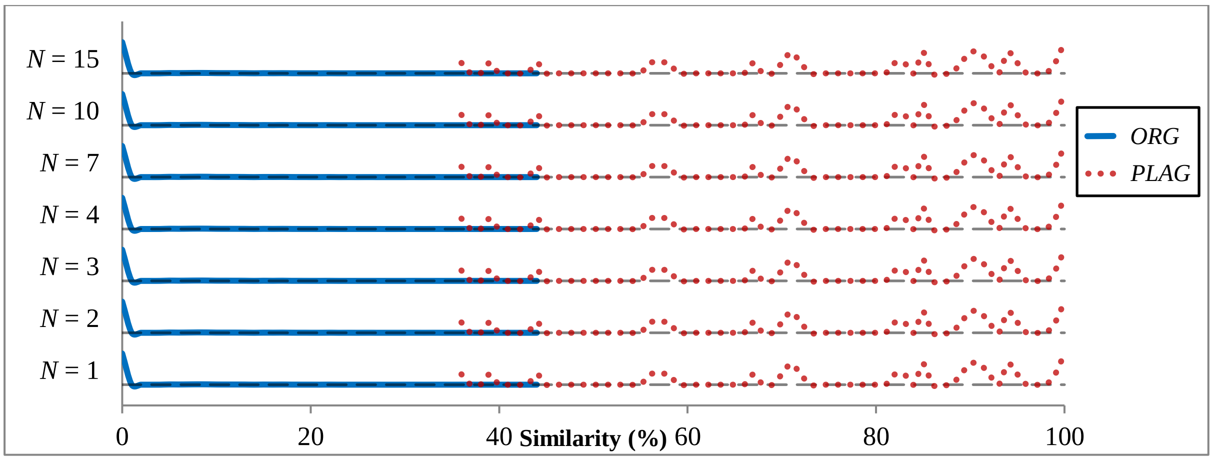


Figure 35: Similarity distribution graph for *Perilous Cities* assignment over different values of *N* with *L* = 5

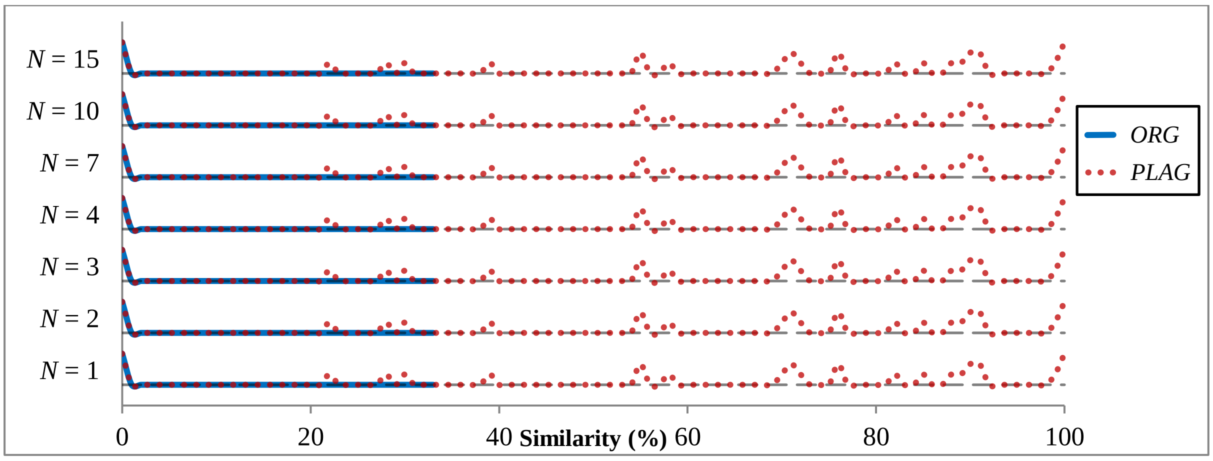


Figure 36: Similarity distribution graph for *Perilous Cities* assignment over different values of *N* with *L* = 10

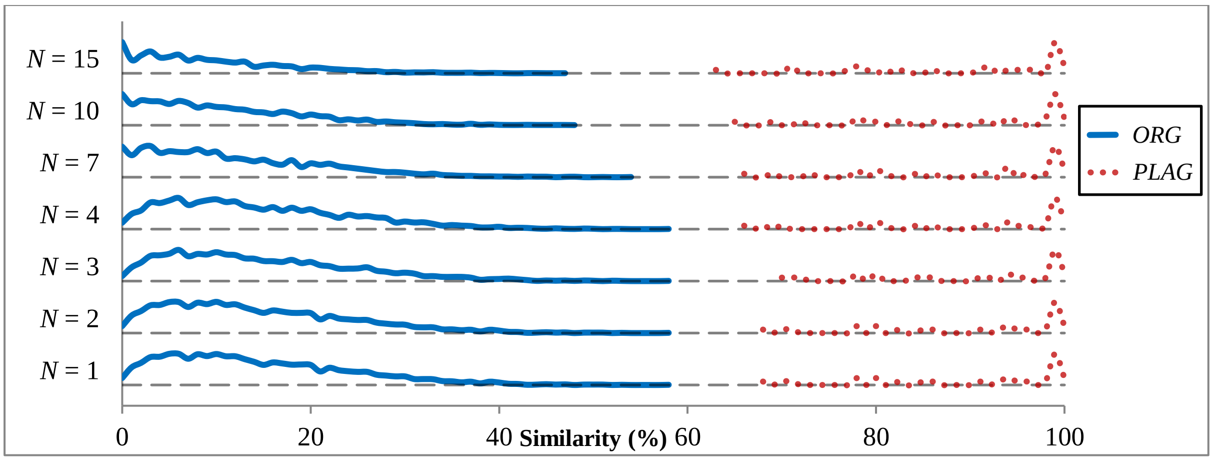


Figure 37: Similarity distribution graph for *Binary Tree Genealogy* assignment over different values of *N* with *L* = 1

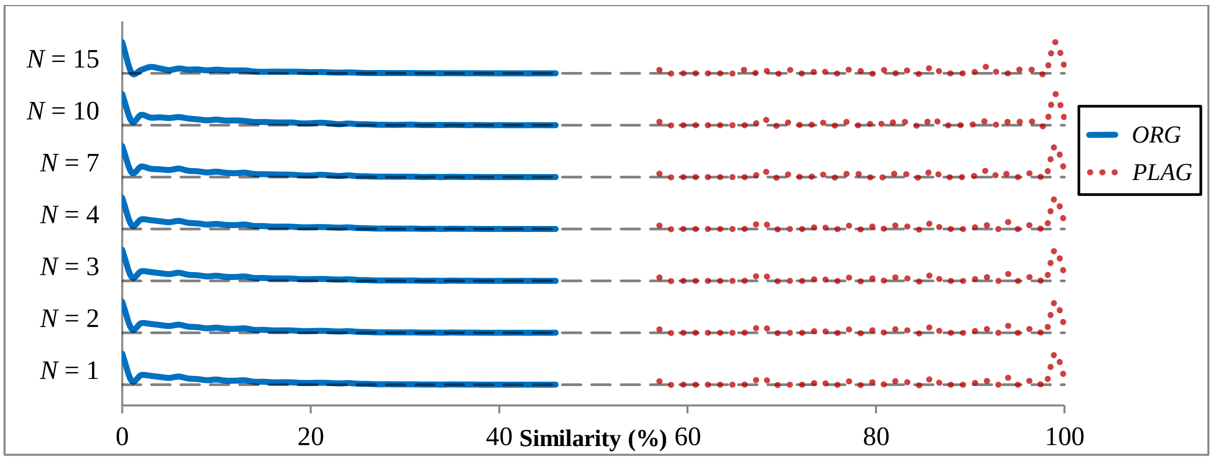


Figure 38: Similarity distribution graph for *Binary Tree Genealogy* assignment over different values of *N* with *L* = 2

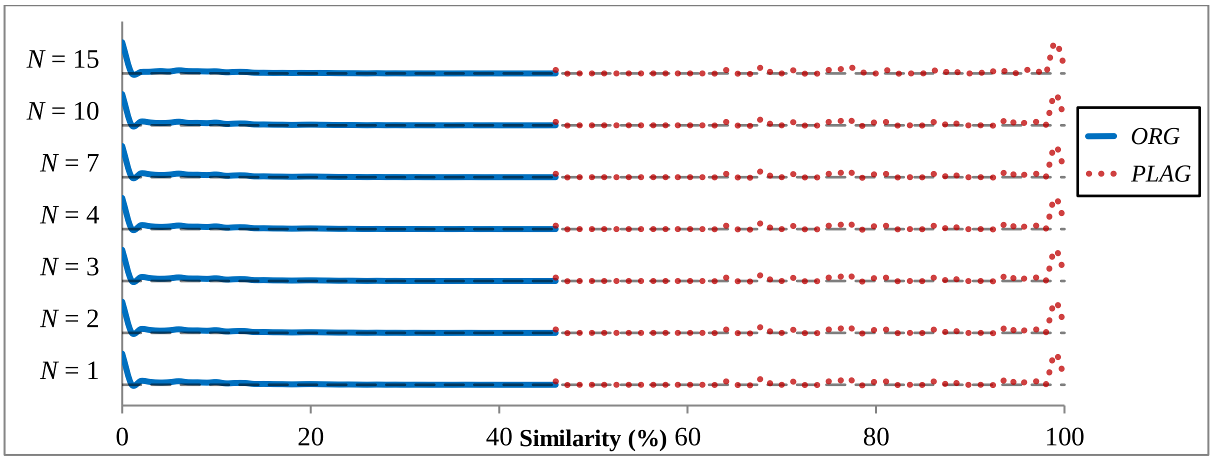


Figure 39: Similarity distribution graph for *Binary Tree Genealogy* assignment over different values of *N* with *L* = 3

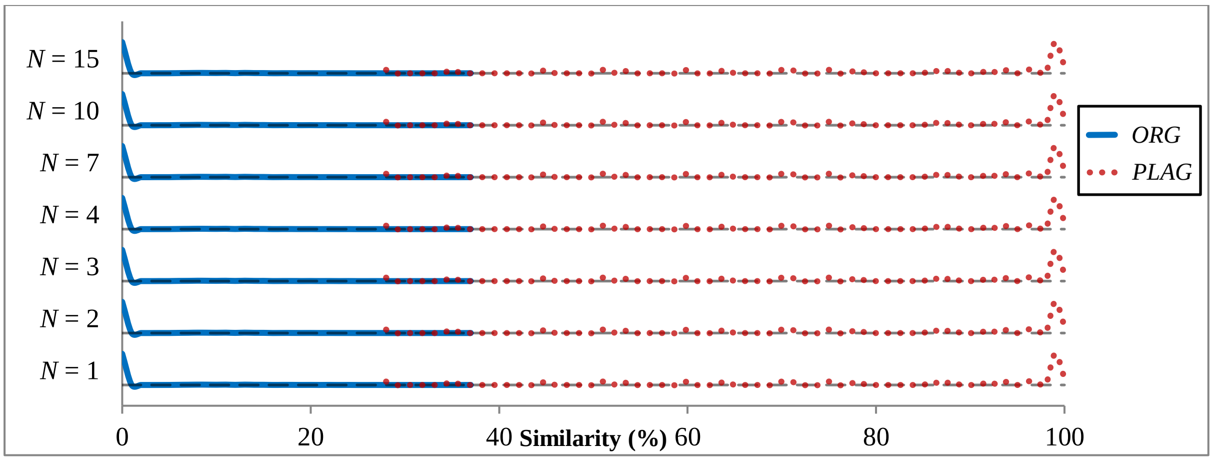


Figure 40: Similarity distribution graph for *Binary Tree Genealogy* assignment over different values of *N* with *L* = 5

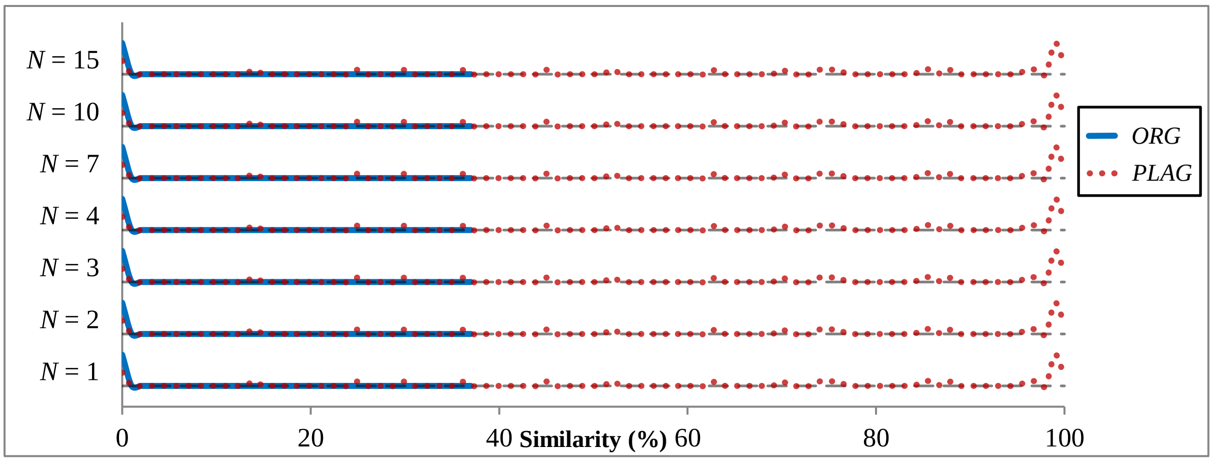


Figure 41: Similarity distribution graph for *Binary Tree Genealogy* assignment over different values of *N* with *L* = 10

Appendix F

The x-axis denotes *N*, the size of *N*-gram (*N*); the y-axis denotes the average time required for the computation to complete. The unit of time is in second.

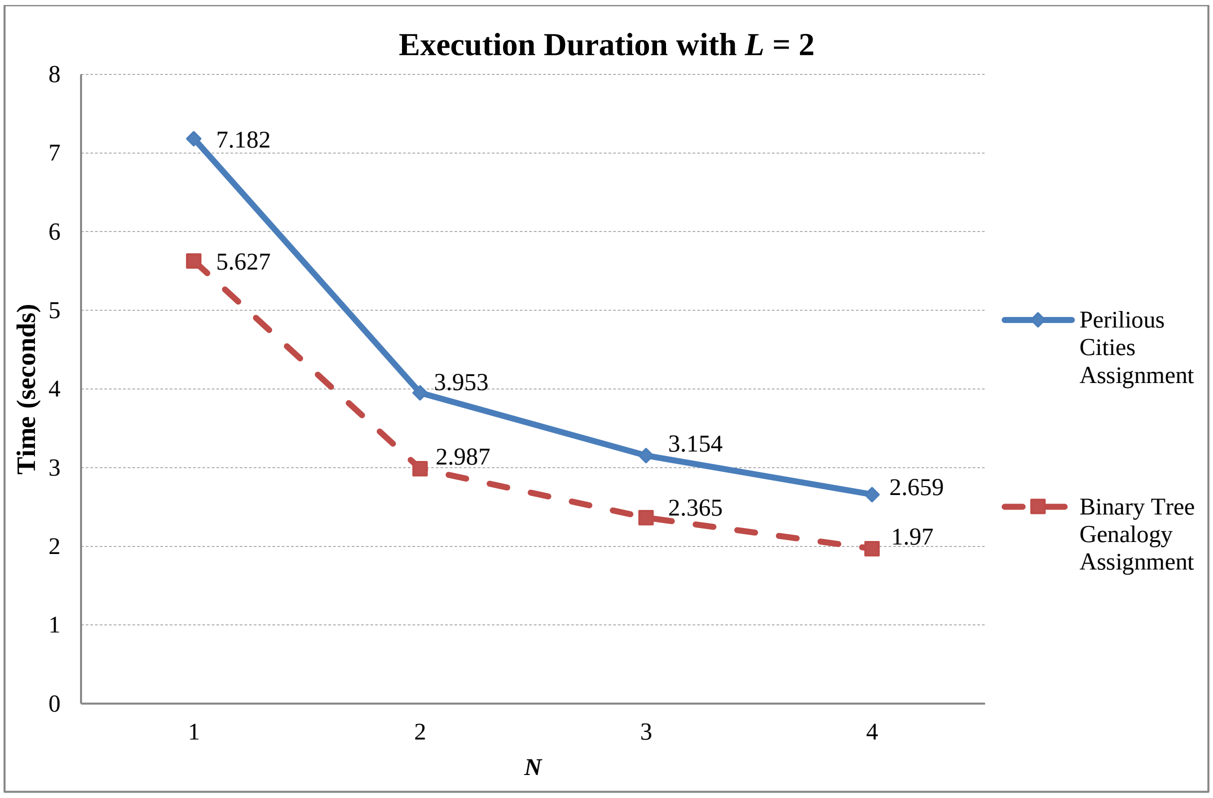


Figure 42: Execution duration of pairwise similarity detection with *L* = 2

Appendix G

The following list contains the suggestions obtained and errors reported in the user review session. A click in first column indicates if the suggestion is implemented and the error is fixed in the final product.

Implemented	Description
	When submitting assignment, the system may take a long time to respond. This may result in a connection timed out. A queue system is recommended to handle the case.
	A automated way of determining plagiarism clusters would be appreciate.
✓	If for an assignment, no plagiarism cluster is generated, the system should not allow the user to view visuals
✓	Spelling errors and language mistakes
✓	Clicked on the cluster to "mark" a student but nothing happened.
	For students who have no history of plagiarism, it is recommended to have the links to their logs marked with special icons that would indicate the absence of plagiarism history of these students.
	For visual interfaces like *Clusters Summary* where edges are used to denote the maximum similarity value between the submissions from two students. It would be convenient if the *Pairwise Comparison Result interface* is displayed when clicking the edge.
	The layout for the visual interfaces should follow the layout provided in the *Module Management interface* for coherence
✓	[?] tooltips don't need to be subscripts
	User guide is too wide. Rescale graphics to be less wide. Table of content of the user guide could be in the left hand column and should stay on the page for the whole time, or have a way to get back to the top so that section navigation is possible.
✓	Navigation links on left hand menu should not keep history (same colour regardless of clicked before or not).
✓	Lighten the gray background a bit. Perhaps use a hint of colour. It looks very monotone.
✓	Breadcrumb navigation could be useful
✓	I don't know what Associated Accounts are. Can we have the [?] in the column headers.

✓	Table layout is incorrect when viewing in the recommended resolution.

Figure 43: List of suggestions and error reports obtained from user review. A tick at the first column indicates the respective suggestion or error is implemented or fixed.

✓	View graphs opens in new tab but other links don't. Be consistent or mark using an icon things that open external windows/tabs.
	Modules view should have some indication of whether assignments were previously submitted or not.
	Some global options are repeated. It's not clear whether these get overridden by defaults from the user or not. The profile part should be clear about what it does. Is it an override? Default?
	You should put a link to a sample.zip file so that a user can get an idea of the submission file's format.
	It would be great to have a progress meter for this upload step or at least a note in red font that says the expected time for calculation.
	How can we show or hide student identifier information?
✓	Default click on row should go to a default URI.
	Perhaps a quick navigator next to show mapping in *Pairwise Comparison Result interface* could be done also.
✓	Hiding and showing of the Mapping List in *Pairwise Comparison Result interface* seems a bit awkward.
	Should be able to lock both scrollers to scroll together. Perhaps by pushing "shift" while scrolling.
✓	Other detected regions should also be shown in some greyed colour or dark red colour.
✓	Default action is to report plagiarism, but any way to mark record as decided specifically as not plagiarized or keep in view?
	Not clear why after reporting we have to decline or declare plagiarism again?
✓	Possible (in FF3) to move Mapping List off the screen such that it's impossible to move again. Either prevent this or ensure show button brings it back up to a moveable point.
✓	Mapping List should show up on bottom right of left panel as the mapping hyperlinks align the red zones to the top of each panel.
✓	Any way to know whether some code is common to the provided skeleton code?
✓	Have a default value for the K graph (could propagate from user profile).
	Allow viewer to traverse between students or groups in graph views

Figure 43: List of suggestions and error reports obtained from user review (cont.). A tick at the first column indicates the respective suggestion or error is implemented or fixed.