Structural Analysis of Source Code Plagiarism using Graphs

George R. Obaido

School of Computer Science and Applied Mathematics,
University of the Witwatersrand, Johannesburg.

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg in fulfilment of the requirements for the degree of Master of Science.

Supervised by
Mr. Pravesh Ranchod and Mr. Richard Klein

May 2017

Declaration

I, Obaido R. George, hereby declare the content of this dissertation to be my own work unless otherwise explicitly referenced. This dissertation is submitted for the degree of Master of Science at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, nor for any other degree.

Signed: 

Date: 31/05/2017

"Education is the most powerful weapon which you can use to change the world."

Nelson Mandela

"If people did not do silly things, nothing intelligent would ever get done."

Ludwig Wittgenstein

Abstract

Plagiarism is a serious problem in academia. It is prevalent in the computing discipline where students are expected to submit source code assignments as part of their assessment; hence, there is every likelihood of copying. Ideally, students can collaborate with each other to perform a programming task, but it is expected that each student submit his/her own solution for the programming task. More so, one might conclude that the interaction would make them learn programming. Unfortunately, that may not always be the case. In undergraduate courses, especially in the computer sciences, if a given class is large, it would be unfeasible for an instructor to manually check each and every assignment for probable plagiarism. Even if the class size were smaller, it is still impractical to inspect every assignment for likely plagiarism because some potentially plagiarised content could still be missed by humans. Therefore, automatically checking the source code programs for likely plagiarism is essential.

There have been many proposed methods that attempt to detect source code plagiarism in undergraduate source code assignments but, an ideal system should be able to differentiate actual cases of plagiarism from coincidental similarities that usually occur in source code plagiarism. Some of the existing source code plagiarism detection systems are either not scalable, or performed better when programs are modified with a number of insertions and deletions to obfuscate plagiarism. To address this issue, a graph-based model which considers *structural similarities* of programs is introduced to address cases of plagiarism in programming assignments.

This research study proposes an approach to measuring cases of similarities in programming assignments using an existing plagiarism detection system to find similarities in programs, and a graph-based model to annotate the programs. We describe experiments with data sets of undergraduate Java programs to inspect the programs for plagiarism and evaluate the graph-model with good precision. An evaluation of the graph-based model reveals a high rate of plagiarism in the programs and resilience to many obfuscation techniques, while false detection (coincident similarity) rarely occurred. If this detection method is adopted into use, it will aid an instructor to carry out the detection process conscientiously.

Keywords: Plagiarism, Source code plagiarism, Programming assignments, Plagiarism graph, Graph model.

Acknowledgements

I wish to express my deepest gratitude to God, my Creator, the Lord, God almighty for the gift of life; who has made today possible. A special gratitude to my supervisors, Mr. Pravesh and Mr. Richard Klein. They have been truly accessible throughout the process and completion of my research work. My sincere gratitude to you both.

To Prof. Turgay Celik, Dr. Abejide Ade-Ibijola and Mr. Abiodun Modupe, I appreciate your words of encouragement during the period of my Masters studies. Again, I would like to thank Liana Meadon and everyone at the WITS Centre of Learning and Teaching Development for the given opportunity to serve as a Digital Literacy Ambassador.

To my late father, Mr. Joseph Obaido; your penchant for academic excellence would forever resonate as drive in my heart. My profound gratitude is to my ever-loving mother, Mrs. Margaret Obaido. To my siblings: Christiana, Victoria, Anthony, Sarah and Faith; I say, "A very big thanks for your continuous support".

To my fiancée, Ms. Patience Thobekile Mhlophe, "I love you very much", is all I need to say. To my friends, Jude-kelvin Oyasor, Keith Benwalson, Elvis Obiora Umejiaku, Rose Gumbanjera, Preven Singh and Dennis Uzizi, thanks to you all for your time. This accomplishment would not have been possible without your likes.

Finally, I would like to acknowledge the WITS Financial Aid and Scholarship Office, and the Council for Scientific and Industrial Research through the Department of Science and Technology Inter-Programme Postgraduate Bursary Support for awarding me the scholarship to undergo this research. I am humbly grateful.

Contents

List of Figures

List of Tables

Publications

Portions of this research have been presented at the following events and conferences;

- G. Obaido, P. Ranchod, R. Klein (2016) Catching Plagiarists: Detecting Plagiarism in Student Source Code Assignments in a Virtual Learning Environment, In Proceedings of the 10th International Technology, Education and Development Conference, Valencia, Spain, pp. 7369-7376.
- The Seventh Cross-Faculty Postgraduate Symposium (2016) at the University of the Witwatersrand, Johannesburg.

Chapter 1

Introduction

1.1 Introduction

In the twentieth century, the advent of the Internet, computers and search engines have made it easier for students to cheat in their assignments. The retrieval of solutions for assignments on the internet requires little or no effort. The traditional ways of copying solutions for assignments from books are becoming obsolete and less likely than the retrieval of solutions from electronic sources [Selwyn, 2008].

The issue of students submitting assignments that they haven't produced has generated many controversies, especially in academia. A recent survey conducted by McCabe [2005] reported that 70% of students admitted to plagiarism, out of which 40% of the students admitted to using the *"cut-and-paste"* approach and about half were found guilty, and liable to a punishable offence. These indications are that plagiarism is commonplace, even though students are aware that the practice is impermissible.

The term *"Plagiarism"* as defined by Encyclopaedia Britannica [Britannica, 2005] is *"the motive of using someone else's work and passing them off as one's work"*. Plagiarism implies taking someone's ideas by copying (stealing) and using it without appropriate credit to the original source. The word *"Plagiarism"* was also defined by Barnhart [1988], in the Dictionary of Etymology, meaning "literary theft", derived from the English word *"plagiary"*('an individual who unjustly copies another person's ideas or words'), originated from the Latin word *"plagarius"* ('kidnapper, seducer') and from the noun "plaga", meaning "net". In the early years, Ben Johnson, the popular English playwright, was the first individual to use the word "plagiary" to connote "literary theft" in the 17th century [Mallon, 1988].

Recent studies on plagiarism [Atkins and Nelson, 2001; Young, 2001; DeVoss and Rosati, 2002; Marshall and Garry, 2005; Howard, 2007] affirmed that the Internet is the primary source of plagiarism. The ubiquitous nature of the Internet has intensified plagiarism in two ways. First, the usage of the Internet has made documents easily accessible for plagiarism. Student assignments can be easily retrieved from the pool of information available on the Internet and these have contributed immensely to the spread of plagiarism. Secondly, it makes the copying process easier because, contents retrieved on the Internet could be processed on word-processing software and easily edited. The rise of online discussion forums has also contributed to the spread of plagiarism [Olt, 2009]. The questions for an assignment could be easily posted in these forums and feedback for them can be generated in good time. Whilst this process serves as a learning aid, it could provide a leeway for plagiarism. Goffe and Sosin [2005] attributed electronic cheating as *"Cyber-cheating"*, since the ease of cheating promotes a lazy approach of using the *"cut and paste"* method of copying electronic sources of information without the need to typeset an assignment. The students who participated in the work of Introna et al. [2003] reckoned that the information on the Internet was in the *"public domain"*. Thus, they argued that the contents gathered from the Internet is for the public consumption. Therefore, they believed it was not an offence to download, edit and print contents from a website, and further claimed the information retrieved was their own content.

Owunwanne et al. [2010] argued that if preventative measures were lax in an academic institution, plagiarism would be on a meteoric rise. If a student plagiarises and gets away with it once, there is likelihood for them to plagiarise even more, since no disciplinary proceedings were taken in the first place. In most universities, it is an offence to plagiarise. An excerpt from the WITS University Plagiarism Policy[1] considers *"allegations of plagiarism brought to its attention by academics, while serious or repeated plagiarism will be handled as a disciplinary offence"*. If an academic institution does not accept they have a problem with plagiarism and deal with perpetrators, they might be denying themselves quality academic standards and independent learning among students. However, the unfairness will be towards those students who choose not to plagiarise. Unfortunately, the problem of plagiarism in academic institutions is increasing rapidly with the rate of students admitted each year.

[1] http://libguides.wits.ac.za

1.2 Motivation

Informal assertions show that in the computing sciences, especially in programming, students plagiarise their programming assignments [Ahmadzadeh et al., 2011]. Students are able to modify a source code that they perhaps obtained from their peers, submit it as their own, and hope their instructor will not discover whether or not the code was copied. If the plagiarised code is not discovered by the instructor, it may be unnoticed until after exams have been marked. An instructor might realise that there are a few students who have not received enough grades in their exams to pass a course, but achieved a higher score in their assignment exercises. In some circumstances, the differences between these two marks are relatively high and can be attributed to anxiety during the examination.

Students who are fond of plagiarising, often find it very difficult to learn programming when they copy each other's assignments, so detecting and discouraging plagiarism is a very serious issue [Chunhui et al., 2013].

However, checking for plagiarism is labour intensive work. Instructors are required to scrutinise all source code submitted by students to verify their authenticity in the early stages of the course. In practice, this is impossible. The number of enrolled students in a university is often very high and this does not enable one to be observant for checking each and every assignment for plagiarism. Automatic detectors are, therefore, of huge importance. Several tools exist to detect similarities between pairs of programs, but an instructor should bear in mind that, the selected tool must be able to explore actual plagiarism, not coincidental similarities which are common in programming courses. In exploring the idea of actual plagiarism, we introduce the concept of graphs and graph theory to source code plagiarism. Figure 1.1 shows a graph structure used to describe student programs and similarity relationships between the programs in a programming class.

In Figure 1.1, student labels represent each submitted program and the lines between each program represent the degree of similarity. For example, the degree of similarity between the programs submitted by "Student L" and "Student Y" is 99. A high degree of similarity between pairs of programs is indicative of plagiarism. By choosing some threshold above which these similarities are displayed, a graph structure (such as that in Figure 1.1) can aid an instructor in identifying groups of plagiarists. If an instructor decides to use a higher threshold value (for example, a threshold value of 80) as an indication that pairs of programs were plagiarised, the graph structure will assist the instructor in carrying out the detection easily. Imagine a programming class of 100 students and with 250 pairwise similarities above the selected threshold, a graph

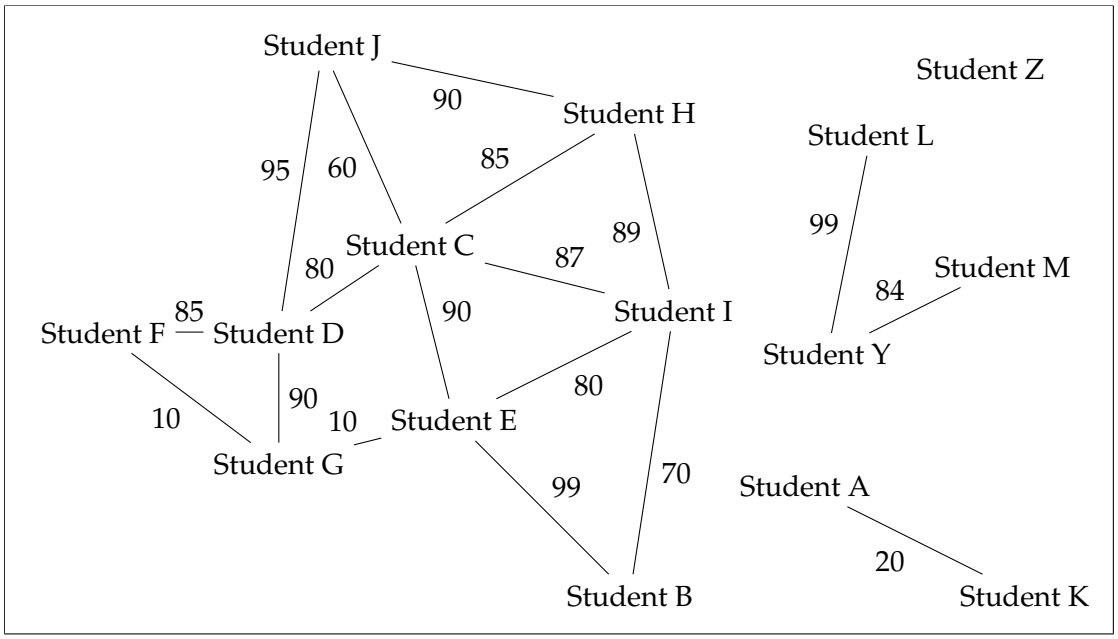


FIGURE 1.1: Application of graphs to source code plagiarism

visualisation aids the instructor in carrying out the detection effortlessly. This is possible because graphs make the relationships between submissions easier to visualise and understand in a way that is intuitive to humans and accessible to mathematical algorithms.

Graphs are abstract objects and have been applied successfully in research related to similarity and matching problems [Conte et al., 2003]. In this research, students' source codes and the strength of similarities between them are represented in the form of graphs. This is useful to show relations of the copied work which will aid us to investigate actual plagiarism from coincidental similarities.

1.3 Problem Description

In programming courses, students work on the same problem in the same environment, while being taught to code by the same instructor using a specific style. This all increases the possibility of finding coincidental similarities even when students are not actually copying. On the other hand, when plagiarising, it is a common practice by students to make certain cosmetic modifications without affecting the output of a program. A plagiarised source code can also be a combination of various portions of different sources: the Internet, colleagues (others) or oneself (self-plagiarism). These modifications reduce the chances of identifying probable plagiarism. For an instructor to carry out the classification of source code similarities effectively, it would require the

instructor to understand what constitutes plagiarism (discussed in the literature of this work), use an automatic detection tool, apply a benchmark (threshold) to adjudicate plagiarism, and to check the history of a plagiarist across a number of programming assignments.

A number of problems are considered and addressed in this work under the categories:

1.3.1 Acceptable threshold to indicate plagiarism

A number of source code plagiarism detection tools exist to assist instructors in investigating plagiarism in programming assignments. However, instructors struggle with an acceptable benchmark that indicates programs that constitute plagiarism in programming assignments. This leaves one with the question: *At what threshold is it sufficient to indicate that a program has been plagiarised?* Prechelt et al. [2002] proposed an automatic plagiarism detection system, JPlag, and considered a fixed cut-off threshold as a sufficient criterion to distinguish plagiarised programs from non-plagiarised programs with near-optimal recall and good precision. Pawelczak [2013] describes a threshold as the level of alternation which represents the maximum allowed similarities between two programs. Hence, one way of considering a threshold that indicates plagiarism in this work, is to perform a *ground truth analysis* (presented in Chapter 7) to manually identify true instances of plagiarism and try to find a suitable threshold. That is, *we perform a thorough evaluation at different thresholds of similarities by manually investigating the source code fragments*.

1.3.2 Constructing and analysing student program plagiarism

A plagiarism detection method is a key element in uncovering plagiarism in programming assignments. While one can imagine a number of modifications students use to obfuscate plagiarism, an instructor is faced with an arduous task of finding an appropriate method of detecting plagiarism in programming assignments. Noynaert [2006] conducted a critical examination of the manual and automatic plagiarism detection methods. The author stressed that both manual and automatic approaches complement each other, but that most instructors would have to use manual methods in verifying the plagiarism report after the automated methods have been used. Furthermore, automated methods are rarely definitive, hence, the need to verify the authenticity of the plagiarism report. Given the complexity of this problem, we are faced with these questions:

1. *does the detection tool assist instructors in correctly identifying plagiarism?*
2. *does the graph-based approach highlight useful structural information within the class to help instructors understand detected program similarity?*

Given the problems above, *we propose an automatic plagiarism detection and analysis tool to detect and represent student relationships as graphs*. That is, we build software prototypes and carry out manual investigation on the source code programs in a bid to uncover plagiarism.

1.3.3 Plagiarism analysis across multiple assignments

Programming assignments are an essential element of computer science education and can help students become familiar with, and understand how the principles can be applied in a real-world scenario. In spite of the effort by instructors to ensure students understand the concept of a programming language by providing them with multiple assignments, there are still students who still engage in plagiarism across their multiple assignments. Given this problem, we are faced with the challenging question: *Can we identify culprits/plagiarists across multiple assignments of the same course?*

In a bid to counteract plagiarism, Wagner [2004] suggested that students should do more than one assignment as this ensures plagiarism to be more conspicuous. Whilst this approach seems plausible, identifying students who plagiarise across multiple assignments should provide instructors with conclusive evidence that students are really engaging in plagiarism. This problem was addressed in this research.

1.4 Research Context

1.4.1 Aims and Objectives

The main aim of this research is to make it easier to determine which cases of measured similarity might be plagiarism. Cases of plagiarism, either *coincidental or actual* are common in programming assignments. In this research, the primary focus is to ensure these issues are addressed. The objectives that make up the aim are to:

1. find better ways of investigating plagiarism in programming assignments using a *graph-based approach*.

2. combine the use of structure similarity detection and graph-based techniques and design a combined approach, which can be used to detect plagiarism in source codes.

3. derive an efficient method for detecting plagiarism in programs.

4. resolve the anomalies of cases in programs. Resolution is done by the manual evaluation of the plagiarism report.

1.4.2 Questions

Specifically, the following questions have been answered during the course of undertaking this research.

1. *At what threshold is it sufficient to indicate that a program has been plagiarised?* This question is addressed in this research. During the evaluation of the plagiarism report, Chapter 7, we realised that even at a lower threshold, programs can still be investigated for plagiarism.

2. *Does the graph-based approach highlight useful structural information to help instructors understand detected program similarity?* Yes, it does. We noticed that the structure of graphs comprises useful indicators to uncovering plagiarism. We were able to distinguish between different components in the structure of graphs presented. More details are presented in Chapter 6.

3. *Can we identify culprits/plagiarists across multiple assignments of the same course?* Yes, this was achieved. We noticed that, inspecting plagiarism across multiple assignments reduced the occurrence of coincidental plagiarism and a number of students were found to have plagiarised each other in multiple assignments. Details of this finding are presented in Chapter 7.

1.5 Methodology

In order to accomplish the objectives of this research, a plagiarism detection tool was built and integrated into an LMS[2]. The proposed tool aims at detecting and analysing plagiarism in programming assignments. The methodology of this research was conducted in three phases:

[2]Learning Management System

1. *pairwise program similarity check*: A plugin already existed for the Moodle LMS that sends student submissions for plagiarism checking by MOSS [Aiken, 1994].

2. *graph visualisation of programs*: Graph visualisation is vital in identifying pairs of the student programs. The visualisation provides additional information when investigating plagiarism. Hence, a popular choice was the Graph visualisation package.

3. *manual classification of the result*: In this phase, the submissions flagged by MOSS were manually assessed for coincidental or actual plagiarism. This involved grouping the responses as *false positive*, *false negative*, *true positive* and *true negative*.

1.6 Contribution of the Study

This study describes our effort to make it easier to determine which cases of similarities in programs may be plagiarism, using a graph-based plagiarism detection approach and addressing the alarming issues of source code plagiarism. This study offers a number of contributions:

1. *easy detection*: The detection process was straightforward. With the click of a button, an instructor can scan a number of programs in an assignment for plagiarism. A more detailed discussion is presented in Chapter 5.

2. *reporting features*: The plagiarism detection tool provided a number of features for presenting the plagiarism report. The tool presented a number of options (histogram, graphs and side-by-side comparison) for investigating plagiarism in a collection of programs (Chapter 5).

3. *long-term storage*: MOSS maintains a database that stores the similarity report, which automatically deletes after 14 days from the MOSS server. Our plagiarism detection system retains reports of previous scans, which is readily available for an instructor at any time (Chapter 5).

4. *enhanced functionality*: The graph-based approach considered in this work possesses features which make it less laborious to investigate program similarities. The graph metrics and parameters (threshold, colours, line count, thickness and percentage) ensured that it was easier to investigate program similarities, rather than on MOSS only, which just gives pairwise similarities in a collection of programs (Chapter 6).

5. *robust visualisation*: Good visualisation can help an instructor understand and make informed decisions based on the data being presented. Ideally, the graph visualisation makes it easy for an instructor to identify patterns in the programming class, especially if the given class size was large and pairwise comparison does not provide enough information to carry out sound judgement (Chapter 6).

6. *decision making*: Effective decision-making is key to an instructor following up on source code plagiarism. The plagiarism detection tool has provided a leeway for making informed decision (Chapter 7).

1.7 Document Outline

The rest of this research is organised as follows:

- **Chapter 2** *Background and related work*: This chapter presents an introduction to source code plagiarism, and a number of reasons why students plagiarise in their assignments are discussed. Furthermore, previous plagiarism detection systems used for measuring similarities in programs are elaborated upon. An overview of the MOSS engine is discussed and the algorithms used by plagiarism detection systems are presented. A number of graph-based methods used for detecting plagiarism in programs are discussed. These methods are compared to the detection method used in this work. Lastly, the classification scheme used in the classification of plagiarism cases is presented. This is useful in the evaluation of the plagiarism detection method applied in this work.

- **Chapter 3** *Graph theory concepts*: In this chapter, basic definitions on the concepts of graphs are discussed. The different graph metrics are examined in relation to source code plagiarism detection, and a number of examples to support this concept are presented.

- **Chapter 4** *Research questions and methods*: This chapter introduces the motivation behind this research. A number of clear focused questions are posed, and the methodology to answer them is outlined.

- **Chapter 5** *Design and implementation*: This chapter presents the methods used in this work. A number of methods are undertaken to create an efficient plagiarism detection system. The plagiarism detection system presents a number of options for detecting plagiarism in the student programs.

- **Chapter 6** *Graph structures*: In this chapter, a number of structures from the implemented plagiarism detection system are analysed. The analysis is useful in evaluating the plagiarism detection system.

- **Chapter 7** *Evaluation*: This chapter provides details on the results of the experiment carried out in this research. A number of experiments are performed on a repository of source code programs. The result from the experiment presented a number of indications which clearly show that the students are really copying each other.

- **Chapter 8** *Conclusion, Limitations, Contribution and Future Work*: This chapter concludes the research, provides the limitations of the study, presents future work, and outlines contribution of the research study.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter discusses work related to the background of source code plagiarism detection and presents the approach for carrying out this research. In particular, Section 2.2 introduces the idea of source code plagiarism; its conceptual definition and the different modifications students make to obfuscate plagiarism. Section 2.3 highlights the reasons why students engage in plagiarism. Section 2.4 introduces the different plagiarism detection systems. Section 2.5 introduces MOSS, the plagiarism detection engine used for this work. Section 2.6 introduces the algorithms used in plagiarism detection. Section 2.7 discusses other graph-based plagiarism detection systems. Section 2.8 introduces the classification scheme. Section 2.9 concludes the chapter.

2.2 Source Code Plagiarism

Source code plagiarism as defined by Parker and Hamblen [1989] *"is a program which has been reproduced with a small number of routine transformations."* It is a significant issue in academia. It is literally the reusing of programs rather than copying of essays. Source code can be reused in different forms from copying and pasting small chunks of a program source code to copying large chunks of a source code and masking these copies with creative techniques to conceal the original copied program [Grier, 1981; Sraka and Kaučič, 2009].

In source code plagiarism, a student who reuses the code of another student, modifies and submits the code as if he/she solely produced it. During the examinations, this can impinge on their performance because they have not engaged with the work

personally. Informal evidence presented by Ahmadzadeh et al. [2011] showed that students who often plagiarise, perform badly during the examination since they cheated during an assignment exercise just to attain good marks. Inspecting each and every program manually is often time-consuming particularly if the class size for a programming course is large. An instructor determines that students passed during an assignment exercise with higher grades compared to their examination scores which are considerably lower, and this is attributed to the reuse of source codes during assignments. These create an imbalance between the two different scores. Hence, an imbalance between the assignment and exam grades is often attributed to plagiarism. Given the complexity of the issues above, the need to automatically check source code is imperative.

While most plagiarism detection tools for essays search for consecutive words in a large pool of text to identify plagiarism, source code plagiarism detectors consider modifications exhibited by students to deceive their instructors. Several modifications exhibited by students in source code plagiarism as enumerated by Faidhi and Robinson [1987] and Seo-Young et al. [2006] are:

Level 1 - Changes in indentation and comments of a program: At this level, sufficient knowledge to actually produce a program is not demonstrated. This involves essentially changing the comments of a program. Two students may reproduce each other's programs but may use entirely different comments to describe their source code. Two student sample Java code is presented in Listing 2.1 and Listing 2.2:

```
1 // A Java code: "I love wits"
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    System.out.println("I love wits");
6   }
7 }
```

LISTING 2.1: Learner A

```
1 /* Sample program to show "I love wits" */
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    System.out.println("I love wits");
6   }
7 }
```

LISTING 2.2: Learner B

The programs produce the same output but the comments used by each students are entirely different. In the above programs, Learner A (Listing 2.1) used a distinct Java description and comment style *"// A Java code :"I love wits""* to illustrate the simple Java program, Learner B (Listing 2.2) used a separate description and comment style "/* *Sample program to show "I love wits"* */ ". At **Level 1**, the comments in the two programs are different but the programs are practically identical - it is extremely likely that the two programs have been plagiarised. We may conclusively determine at **Level 1** that the programs may have been plagiarised.

Level 2 - Changes in Level 1 and changes in program identifiers: At **Level 2**, slight changes in the comment of a program as presented in **Level 1**, and changes in program identifiers may be used to hide plagiarism. If pairs of programs yielded the same results, different identifiers may have been used in the programs to achieve them. Sample Java programs are illustrated in Listing 2.3 and Listing 2.4:

```
1 // A Java code: "I love wits"
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    string A;
6    A = "I love wits";
7    System.out.println(A);
8   }
9 }
```

LISTING 2.3: Learner A

```
1 /* A sample program to show "I love wits" */
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    string Z;
6    Z = "I love wits";
7    System.out.println(Z);
8   }
9 }
```

LISTING 2.4: Learner B

We can deduce that the above pair of source codes are identical, but the two students used different program identifiers. Learner A used the *identifier* (A) as shown in Listing 2.3 while, Learner B used a separate *identifier* (Z) in Listing 2.4. If we investigate the pair of programs as copied programs based on the identifiers, this does not provide sufficient information to inspect the source codes as suspicious cases of plagiarism. They may have presented the programs the way the instructor had taught them in the programming class.

Level 3 - Changes in Level 2 and changes in variable position: At **Level 3**, the variables expressed in the following programs in Listing 2.5 and Listing 2.6 may have been plagiarised. If pairs of programs show a slight change in the position of the variable of a program, it may have been plagiarised. Students may deceive their instructors by altering the position of the variable, they may have yielded the same result but structurally it may differ in presentation. Sample programs to show **Level 3** are indicated in Listing 2.5 and Listing 2.6.

```
1 // Sample code "I love wits"
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    string A = "I love wits";
6    System.out.println(A);
7   }
8 }
```

LISTING 2.5: Learner A

```
1 /* Program to show "I love wits" */
2 public class Lovewits {
3   public static void main(String[] args)
4   {
5    string Z;
6    Z = "I love wits";
7    System.out.println(Z);
8   }
9 }
```

LISTING 2.6: Learner B

The program modifications as shown in **Level 3** may constitute plagiarism. At **Level 3**, Learner A declared the *variable* (String A="I love wits") and its *value* on the same line as seen in Listing 2.5. Contrary, Learner B applied the *identifier*, the *variable declaration* and the *value* on separate lines as seen in Listing 2.6. Significant programming skill is not required to plagiarise a source code at this level.

Level 4 - Changes in Level 3 and changes in procedure combinations: At **Level 4**, programs may have been plagiarised. Pairs of programs to demonstrate **Level 4** are illustrated in Listing 2.7 and Listing 2.8.

In the pair of programs that follow, we can conclude that two programs may have been plagiarised as seen in Listing 2.7 and Listing 2.8. If we inspect the code, we may conclude that they are similar with regard to the program modules used in the program, but the students may have changed the variable identifiers.

```
1  package acceptinput;
2  import java.util.Scanner;
3  // Sample code to get input text
4  public class Acceptinput {
5    public static void main(String[] args)
6    {
7     int Y = in.nextInt();
8     int Z = in.nextInt();
9     System.out.println(Y);
10    System.out.println(Z);
11   }
12 }
```

LISTING 2.7: Learner A

```
1  package receiveinput;
2  import java.util.Scanner;
3  public class Receiveinput {
4  /* Receive inputs from users */
5    public static void main(String[] args)
6    {
7     int num1, num2;
8     num1 = in.nextInt( );
9     num2 = in.nextInt();
10    System.out.println(num1);
11    System.out.println(num2);
12   }
13 }
```

LISTING 2.8: Learner B

Level 5 - Changes in Level 4 and changes in the statements of programs: We can stipulate a case of source code plagiarism at **Level 5** as presented in Listing 2.9 and Listing 2.10. Students may possess excellent programming skills to produce a source code at this level. If we have pairs of programs as shown next, we can clearly examine the source code and observe cases of plagiarism at this stage.

If we thoroughly examine the pairs of programs displayed in Listing 2.9 and Listing 2.10, we can clearly investigate that the two programs may have been plagiarised.

```java
package acceptinput;
import java.util.Scanner;
public class Acceptinput {
// Sample code to get input text
  public static void main(String[] args)
  {
   int Y = in.nextInt();
   int Z = in.nextInt();
   if (Y>Z)
   System.out.println("Y is greater");
   else
   System.out.println("Z is greater");
  }
}
```

LISTING 2.9: Learner A

```java
package receiveinput;
import java.util.Scanner;
/* Receive inputs from users */
public class Receiveinput {
  public static void main(String[] args)
  {
   int num1, num2;
   num1 = in.nextInt( );
   num2 = in.nextInt();
   if (num1>num2)
   System.out.println("num1 is greater");
   else
   System.out.println("num2 is greater");
  }
}
```

LISTING 2.10: Learner B

Level 6 - Changes in Level 5 and changes in the control logic: This is the last stage where we can conclusively infer that pairs of programs may have been plagiarised. Students may possess excellent programming skills to produce a source code at this level. An example of Level 6 plagiarism is illustrated in Listing 2.11 and Listing 2.12.

```java
import java.util.Scanner;
public class TimesDrill {
        public static void main(String[] args) {
                int a, b; // times operands
                int ansR; // right answer
                int ansU; // user's answer
                int score; // user's score
                String s; // output string

                  Scanner scanner = new Scanner( System.in );
                  System.out.println("Practice multiplication");
                  System.out.println("To quit, enter -1");
                  score = 0; // initialise score
                  do
                  { // generate question
                        a = (int)( Math.random( )*11 );
                        b = (int)( Math.random( )*11 );
                        s = a + " X " + b + " = ";
                        System.out.print( s + "? " );
                        ansR = a * b; // compute right answer
                        ansU = scanner.nextInt( ); // get user's answer
                        if ( ansU == -1 ) // user wants to quit
                        {
                                System.out.println( "Goodbye" );
                                break; // quit
                        }
                        if ( ansU != ansR ) // user answered wrong
                        {
                                System.out.print( "Sorry, you failed!!, " + s + ansR );
                                break; // quit
                        }
```

```
                        // user entered a correct answer so add one to
                        // score and encourage continuing
                                score++;
                                System.out.println(score+" right. Keep going!");
                    } while ( true );
          }
 }
```

LISTING 2.11: Learner A

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Random;

public class TimesDrill {

        public static void main(String[] args) {
           int a = 0;
           int b = 0; // times operands
           int ansR = 0; // right answer
           int ansU = 0; // user's answer
           int score = 0; // user's score
           String question; // output string

           System.out.println("Practice multiplication");
           System.out.println("To quit, enter -1");
           score = 0; // initialise score
           BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
           do {
            // generate question
            Random rand = new Random();
            a = rand.nextInt(12 - 1 + 1) + 1;
                b = rand.nextInt(12 - 1 + 1) + 1;
            question = a + " X " + b + " = ";
            ansR = a * b; // compute right answer

            while(true){
                    try{
                      System.out.print(question);
                      ansU = Integer.parseInt(in.readLine()); // get user's answer
                      break;
                     }catch(IOException | NumberFormatException e){
                      // if ans invalid show message
                             System.out.println("Please enter a number !");
           }
        }

                       if (ansU == -1) // user wants to quit
                          {
                            System.out.println("Goodbye");
                            break; // quit
                          }
                        if (ansU != ansR) // user answered wrong
                        {
```

```
46                          System.out.print("Sorry, you failed!!, " + question + ansR);
47                             break; // quit
48                          }
49                            // user entered a correct answer so add one to
50                            // score and encourage continuing
51                          score++;
52                            System.out.println(score + " right. Keep going!");
53
54                          }while(true);
55    }
56
57 }
```

LISTING 2.12: Learner B

We can decisively concede at **Level 6** that the programs may have been plagiarised.

The six source code modifications that can constitute source code plagiarism are presented in Figure 2.1. Source code programs at **L1** require no programming skills to produce a program. If we proceed up to the highest level, **L6**, we are able to establish that excellent programming skills is required at **Level 6** to produce a program. Other source code modifications carried out by students were mentioned in the work of Joy and Luck [1999]; Jones [2001]; Dick et al. [2002]; Prechelt et al. [2002]; Sheard et al. [2003]; Mozgovoy [2006]; Roy and Cordy [2007]. In conclusion, checking pairs of plagiarised source code using the Faidhi and Robinson [1987] categorisation of source code modification, is plausible.

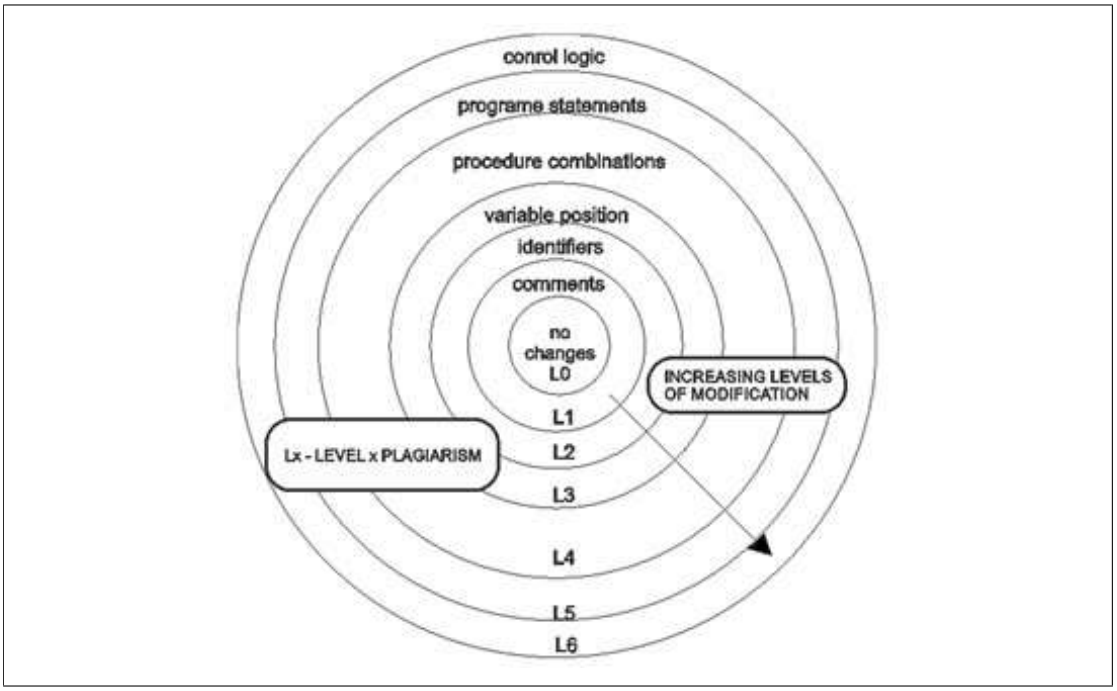


FIGURE 2.1: The different modifications in source codes [Parker and Hamblen, 1989].

2.3 Reasons why students plagiarise source codes

There are several reasons highlighted by Hattingh et al. [2013] regarding the cause of students engaging in source code plagiarism;

- **Poor knowledge of a programming language:** Inadequate knowledge of programming can affect academic performances. Students who detest programming usually perform poorly hence, they engage in cheating in order to achieve better grades.
- **Academic pressure:** Taking multiple courses can compel a student to cheat. The "fear of failure" would often make a student indulge in plagiarism. Academic pressure is invariably "self-perpetuating", and students would resort to any means of copying just to attain good grades.
- **Poor time management:** There are a lot of extra-curricular activities that students devote time to, which further diminishes the interest to learn programming. Hence, poor time management contributes to students plagiarising each other.
- **Self-plagiarism:** Self-plagiarism happens when a student reuses pieces of code between assignments to carry out programming tasks. The effect of this will lead to a significant issue when the source codes are compared for plagiarism.

Bennett [2005] gave a different opinion on factors associated with student plagiarism. The author listed means and opportunity (using the internet resource), personal traits (desire to achieve a high grade) and individual circumstances (high volume of academic workloads) as contributory factors to student plagiarism. Students' desire to cheat would escalate even further if appropriate mechanisms are not enforced to curb the menace. Mann and Frew [2006] highlighted a number of reasons why students plagiarise in the programming assignments. The reasons provided by Mann and Frew [2006] are that students have zero knowledge of what programs are, students have been taught by the same instructor, hence, they are introduced to a particular coding style. Students are made to work in a group, hence, there is likelihood of submitting the same answers. They acquire external help to write their assignments early on in the course, which increases their desire to cheat when the need arises. Furthermore, Vogts [2009] opined that the main reasons why students plagiarise in their source code assignments, were due to a lack of knowledge about the programming language and largely as a last resort, to pass in most cases. All these factors mentioned by these authors contribute a great deal to reasons why students plagiarise in their assignments.

In the next section, we categorise the different plagiarism detection systems and their application to detecting plagiarism.

2.4 Plagiarism Detection Systems

Students are aware of various modes of cheating to disguise plagiarism detection tools. They carry out plagiarism effortlessly with some creative techniques. To them, plagiarism is seen more as a rewarding exercise. One obvious attempt at cheating in a programming assignment would be to obtain a copy of a working program and change the comments, rename the variable names and rearrange the statements of the program. A manual inspection of the copied program can reveal traces of plagiarism, this is however impossible if the class size were large. Plagiarism detection systems for programming assignments are of paramount importance, because they can identify traces of plagiarism in a large class size. Lancaster and Culwin [2005] defined plagiarism detection systems as *"the software programs that compare student programs with each other or with other potential sources for plagiarism"*. The algorithms employed by source code plagiarism detection systems are able to detect plagiarism at the different levels of program modifications carried out by students.

In evaluating plagiarism detection systems, we will examine two important systems in measuring similarities in a program, which are: the attribute counting system and the structure counting systems.

2.4.1 Attribute Counting System

Early automated plagiarism detection systems for student-generated programs employed the attribute counting system for detecting plagiarism in FORTRAN programs [Ottenstein, 1976]. The attribute counting system relied on Halstead software metrics in detecting plagiarism in student-generated programs by counting the number of operators and operands. Halstead [1977] suggested the following metrics for measuring the level of similarities:

1. η_1 - The number of unique operators: The η_1 is the total number of unique operators in a program. The operators in a program can be mathematical, relational, logical, assignment and a bitwise operators. They perform a function on an operand.
   Example: A + B - C
   The unique operators are "+" and "-". Hence, the value of $\eta_1 = 2$.

2. η_2 - The number of unique operands: The η_2 is the total number of unique operands in a program. The operands are the entity upon which the operators act.
   Example: A + B - C + D
   The unique operands are "A", "B","C", and "D". Hence, the value of η_2 = 4

3. N_1 - The total number of occurrences of unique operators: The N_1 is the total number of times an operator occurs in a program.
   Example: A + B - C + D
   The unique operators are "+", which occurs twice in the above example, and "-", which occurs once. Hence, the value of N_1 = 3.

4. N_2 - The total number of occurrences of unique operands: The N_2 is the total number of times an operand occurs in a program.
   Example: A + B - C + D
   The unique operators are "A", "B","C", and "D", and they occur once in each of the examples presented. Hence, the value of N_2 = 4.

The system employed by Ottenstein [1976] indicated that if corresponding values of η_1, η_2, N_1 and N_2 for pairs of student-generated programs have the identical values, the programs should be inspected for plagiarism.

The attribute counting system led to the introduction of other systems using metrics like counting the number of loops and number of procedures [Grier, 1981; Verco and Wise, 1996; Bandara and Wijayarathna, 2011]. These systems were effective in detecting plagiarism because they applied essential features to the attribute of a program to find similarities in programs. The Grier [1981] system was called the "Accuse" system and was intended for Pascal programs. The system used the following parameters for detecting plagiarism in Pascal programs in different combinations to find the most effective combination:

1. The number of unique operators in a program
2. The number of unique operands in a program
3. Total operators in a program
4. Total operands in a program
5. The code lines (excluding comments lines) in a program
6. The used and declared variables
7. Total control statements

The Accuse system [Grier, 1981] applied more sophisticated metrics in detecting source code plagiarism compared to those used by Halstead [1977] in measuring similarities in source codes.

Another application of the attribute counting system was the system introduced by Faidhi and Robinson [1987] for finding similarities in student-generated programs. Their system employed the counts of control flow structures in a program. In the set of metrics employed by them, they argued that it was difficult for a novice programmer to alter a program. They argued their method was more effective in detecting plagiarism than the other plagiarism detection systems, because their system was less prone to false results generated (false positives) due to some hidden measures employed by the set of metrics used by their system. It is important to note that if two programs have the same number of variables used, counts of loops, counts of methods and counts of conditional statements might be marked as suspicious programs if an attribute counting system were used for finding similarities. Obviously, this might not clearly indicate that the two programs are similar. It is worth further inspection into the programs. Although attribute counting systems are effective, they do not consider the structure of a program when finding similarities in programs. Hence, attribute counting systems are limited in detecting plagiarism in source codes [Burrows et al., 2007]. Another limitation of the attribute counting system is that they are only effective in finding similarities with shorter programs. It is difficult to find similarities in larger programs [Verco and Wise, 1996].

In the next section, we will examine the structure counting system in measuring a level of similarity in a program.

2.4.2 Structure Counting System

The structure counting system compares the representations of programs' structures [Ottenstein, 1976; Chen et al., 2004]. The representation can be a program's string, data flows, work flows and parse trees [Tresnawati et al., 2012]. The structure counting system does not compare exact matches as in the case of the attribute counting system, but checks for the similarity of a program's token string for suspicious plagiarism. Burrows et al. [2007] argued that in a structure counting system, comments, whitespaces, and variable names are discarded because they are easily customisable. It only changes the structure of programs and compares them for plagiarism. Recent detection systems as indicated by Verco and Wise [1996] used the structure counting system in detecting plagiarism in programs. Arwin and Tahaghoghi [2006] introduced a novel approach,

XPlag, into structure counting systems to detecting plagiarism across programs written in different programming languages. XPlag detects interlingual plagiarism, where source code is copied from one programming language to another by using a *compiler suite* to convert the programs. The reason for this, is because, programs have the same structure, hence, it is easy to plagiarise a source code written and converted from one programming language to another. Moreso, a code written in C, may be converted into Java. Hence, their plagiarism detection system explores detecting programs converted from one programming language to another to mask plagiarism. Another example of a structure counting system is the Student Submissions Integrity Diagnosis (SSID) developed by Poon et al. [2012]. Their detection system ensured a three-step pipeline in finding similarities in programs. The steps are the tokenisation of the program structure, pairwise similarities of the program pair and considering plagiarism clusters, which use a similarity threshold to assign each submission to groups. Ganguly and Jones [2014] extended the structure counting system by introducing an Information Retrieval (IR) approach. The information retrieval approach considers each document as a pseudo-query and retrieves a list of programs in a decreasing order of their similarity values. Other systems that apply the structure counting system approaches are Measure of Software Similarity (MOSS), Software Similarity Tester (SIM), JPlag, Plague, and Yet Another Plague (YAP). MOSS is considered in depth in the following section and is used as the back-end to generate pairwise similarity scores of programs.

2.5 Plagiarism Detection using MOSS

Aiken [1994] presented MOSS (Measure of Software Similarity) as an automated system for determining the similarity of programs. Presently, the main contribution of MOSS has been in detecting plagiarism in source codes. Since its introduction in 1994, MOSS has been very effective in detecting plagiarism for educators around the world. MOSS generates fingerprints for different aspects of a document. The MOSS similarity output is the number of matching fingerprints in pairs of programs. Additionally, MOSS sorts these results and shows the highest percentage matches to the user. MOSS supports C, C++, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, x86 assembly and HCL2.

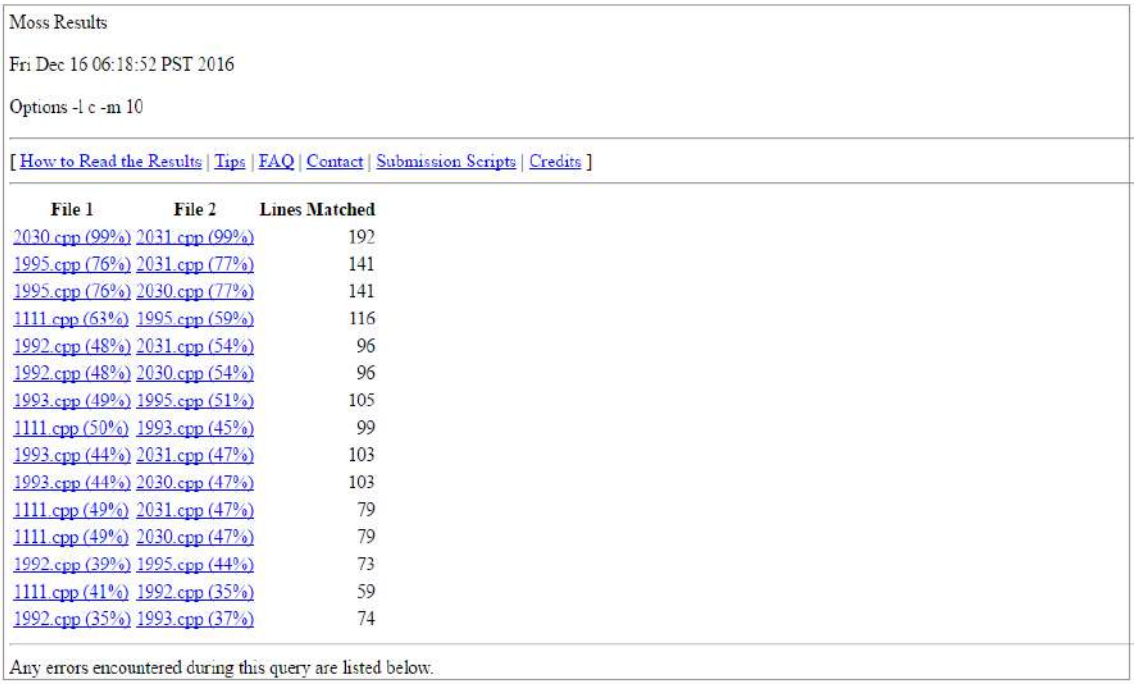
MOSS is a free, online service [Aiken, 1994] that performs a pairwise comparison on a large collection of student submissions, assigns each pair a percentage, and the percentage indicates a measure of similarity between two or more student-generated programs. MOSS compares programs by dividing programs into a certain length, k-grams,

where each k-gram is hashed and MOSS selects the hash values as the fingerprint of the program [Schleimer et al., 2003]. Two programs' similarity is determined by the number of fingerprints shared by the program. In other words, the more fingerprints they share, the more similar the programs are.

Before programs are converted into hashes, MOSS pre-processes source code files, calculates a numeric fingerprint for each file, and then performs the longest common sequence search on the two fingerprints [Schleimer et al., 2003]. At the preprocessing stage, MOSS replaces all function and variable names with a single token, and removes all comments and whitespace from the source code. Bowyer and Hall [1999] gave a succinct explanation on the usage of MOSS in checking for similarities in programs. To use MOSS, it would require a user to send an email to *moss@moss.stanford.edu*. A mail sent to this address would result in a reply email which contains a perl script, which can be downloaded into the instructor's computer.

Assignments to be submitted to MOSS can be in a subdirectory or a directory from which the MOSS script is to be executed. An instructor can also upload an *instructor-supplied* template alongside the assignments to be checked for plagiarism. Hence, when the *instructor-supplied* code is supplied, lines of codes that also appear in the students' programs, are not counted as plagiarism. Hage et al. [2010] explained that the *instructor-supplied* code improves the result by eliminating a number of false positives, but this is not usually necessary to include when checking programs for plagiarism. The *instructor-supplied* code might be a partial solution, a *code snippet* or a program outline that the students are expected to adhere to. After sending the programs to MOSS, the result is available as a web page. The report shows the pairwise similarity of each of the program pairs. Figure 2.2 shows an example of a MOSS results page for some program pairs. Bowyer and Hall [1999] added that to view the plagiarism report, an instructor needs to click the link displayed in the report. Assessing the link, shows the section at which the two programs appear similar. Figure 2.3 shows the side-by-side comparison of the similarity report.

In the result shown in Figure 2.3, we can clearly see that MOSS ignores a program's comments and variable declaration. This is seen between line number 8 - 199 in the first program and 9 - 200 in the second program, shown in red. The colour in both programs shows the sections at which the two students plagiarised each other. We observed that MOSS reported a higher similarity value of 90 for both programs, which is an indication that the two programs are similar. Hence, the two students' programs in Figure 2.3 were plagiarised. MOSS uses the *winnowing algorithm* to measure similarities in programs [Schleimer et al., 2003].



Moss Results

Fri Dec 16 06:18:52 PST 2016

Options -l c -m 10

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
2030.cpp (99%)	2031.cpp (99%)	192
1995.cpp (76%)	2031.cpp (77%)	141
1995.cpp (76%)	2030.cpp (77%)	141
1111.cpp (63%)	1995.cpp (59%)	116
1992.cpp (48%)	2031.cpp (54%)	96
1992.cpp (48%)	2030.cpp (54%)	96
1993.cpp (49%)	1995.cpp (51%)	105
1111.cpp (50%)	1993.cpp (45%)	99
1993.cpp (44%)	2031.cpp (47%)	103
1993.cpp (44%)	2030.cpp (47%)	103
1111.cpp (49%)	2031.cpp (47%)	79
1111.cpp (49%)	2030.cpp (47%)	79
1992.cpp (39%)	1995.cpp (44%)	73
1111.cpp (41%)	1992.cpp (35%)	59
1992.cpp (35%)	1993.cpp (37%)	74

Any errors encountered during this query are listed below.

FIGURE 2.2: A pairwise similarity of the programs

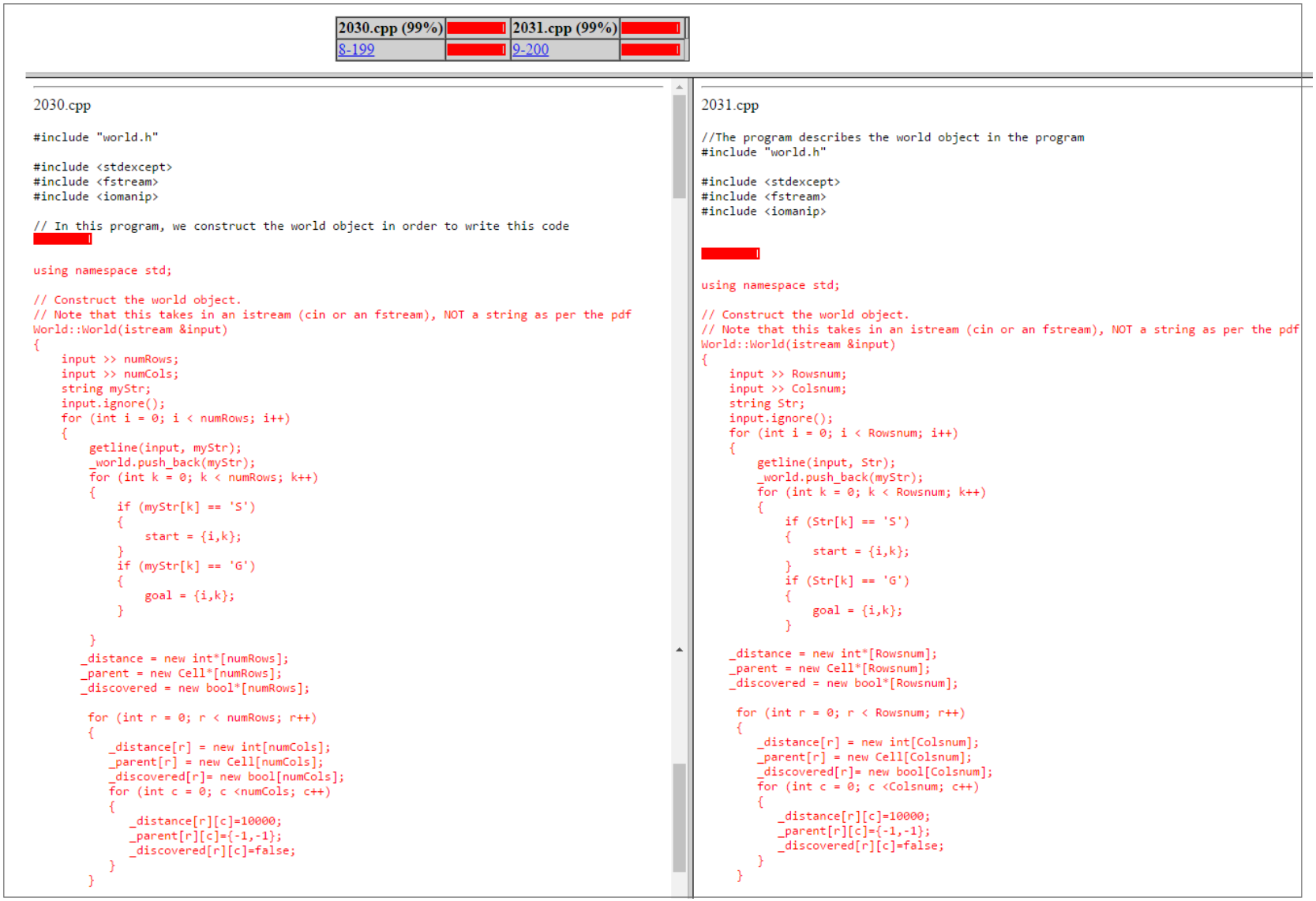
We will examine the winnowing algorithm in the next section to better understand how MOSS uses the algorithm to check similarities in programs.

2.6 Algorithms used in Plagiarism Detection

In this section, we look at some of the algorithms used in plagiarism detection and understand how they perform their operations.

2.6.1 The Winnowing Algorithm

The winnowing algorithm is one of the similarity matching algorithms used for identifying similar documents [Schleimer et al., 2003]. Sorokina et al. [2006] identified the winnowing algorithm as an instance of the fingerprinting algorithm used for finding duplicates of smaller chunks of a document in a larger document collection. MOSS employs the winnowing algorithm concepts for measuring similarities in programs [Chen et al., 2004; Smith and Horwitz, 2009; Al Jarrah et al., 2011; Djuric and Gasevic, 2012]. The winnowing algorithm identifies similarities in programs by dividing them into k-grams, converting the k-grams into hashes, selecting subsets of the hashes as fingerprints and comparing the fingerprints as the similarities between two documents. A k-gram is the simplest approach used by fingerprinting algorithms to identify similarities [Wolpers et al., 2010]. Ohmann [2013] explained that the idea of using k-grams is to



2030.cpp (99%)	[redacted]	2031.cpp (99%)	[redacted]
8-199	[redacted]	9-200	[redacted]

2030.cpp

```
#include "world.h"

#include <stdexcept>
#include <fstream>
#include <iomanip>

// In this program, we construct the world object in order to write this code
[redacted]

using namespace std;

// Construct the world object.
// Note that this takes in an istream (cin or an fstream), NOT a string as per the pdf
World::World(istream &input)
{
    input >> numRows;
    input >> numCols;
    string myStr;
    input.ignore();
    for (int i = 0; i < numRows; i++)
    {
        getline(input, myStr);
        _world.push_back(myStr);
        for (int k = 0; k < numRows; k++)
        {
            if (myStr[k] == 'S')
            {
                start = {i,k};
            }
            if (myStr[k] == 'G')
            {
                goal = {i,k};
            }

        }
       _distance = new int*[numRows];
       _parent = new Cell*[numRows];
       _discovered = new bool*[numRows];

        for (int r = 0; r < numRows; r++)
        {
           _distance[r] = new int[numCols];
           _parent[r] = new Cell[numCols];
           _discovered[r]= new bool[numCols];
           for (int c = 0; c <numCols; c++)
           {
              _distance[r][c]=10000;
              _parent[r][c]={-1,-1};
              _discovered[r][c]=false;
           }
        }
```

2031.cpp

```
//The program describes the world object in the program
#include "world.h"

#include <stdexcept>
#include <fstream>
#include <iomanip>


[redacted]

using namespace std;

// Construct the world object.
// Note that this takes in an istream (cin or an fstream), NOT a string as per the pdf
World::World(istream &input)
{
    input >> Rowsnum;
    input >> Colsnum;
    string Str;
    input.ignore();
    for (int i = 0; i < Rowsnum; i++)
    {
        getline(input, Str);
        _world.push_back(myStr);
        for (int k = 0; k < Rowsnum; k++)
        {
            if (Str[k] == 'S')
            {
                start = {i,k};
            }
            if (Str[k] == 'G')
            {
                goal = {i,k};
            }

    _distance = new int*[Rowsnum];
    _parent = new Cell*[Rowsnum];
    _discovered = new bool*[Rowsnum];

     for (int r = 0; r < Rowsnum; r++)
     {
        _distance[r] = new int[Colsnum];
        _parent[r] = new Cell[Colsnum];
        _discovered[r]= new bool[Colsnum];
        for (int c = 0; c <Colsnum; c++)
        {
           _distance[r][c]=10000;
           _parent[r][c]={-1,-1};
           _discovered[r][c]=false;
        }
     }
```

FIGURE 2.3: A side-by-side comparison of the MOSS report

create a list of all sub-elements that appear in a particular document. Hence, a *k*-gram is a contiguous substring of length *k*, generated by a string of length *N* in a program, where *N* is the number of characters and *k* is the contiguous substring, *N*-(*k*-1). Figure 2.4 illustrates the *k*-gram generation process, where *N* = 22, *K* = 5. Hence, *N*-(*k*-1) = 18.

for (int i = 0; i <10; i++){}
(a) A program statement

for(inti=0;i<10;i++){}
(b) The irrelevant features removed

for(i, or(in, r(int, (inti, inti=, nti=0, ti=0;, i=0;i, =0;i<, 0;i<1, ;i<10, i<10;, <10;i, 10;i+, 0;i++, ;i++), i++){, ++){}

(c) The unique 18-grams generated.

FIGURE 2.4: The *k*-gram generation process [Ohmann, 2013]

The winnowing algorithm is used for source code plagiarism in order to select fingerprints from hashes of *k*-grams of programs and to make a comparison of the fingerprints of each program for measuring similarities [Marinescu et al., 2013]. Purwitasari et al. [2011] gave a succinct explanation to the calculation of the winnowing algorithm by creating a hash value of each ASCII character using the rolling hash equation. By comparing the fingerprint values of each document; the window size, basis and k-gram values were being experimented to indicate source code plagiarism. Huston [2008] emphasised that the main purpose of using the hashing technique in the winnowing algorithm is that it usually converts k-grams which are represented as *strings*, into *numbers* for easy comparison.

The winnowing algorithm is insensitive to whitespaces (ignores spaces in a program), discards unnecessary noise (suppresses the appearance of common keywords, like ”print”) and is independent of the position (re-ordering part of a program portion) of a document [Schleimer et al., 2003; Zdziarski et al., 2006]. The steps in the winnowing algorithm are shown in Figure 2.5. As we move along the stages of the winnowing algorithm, we are able to generate the similarity value of each of the fingerprint of the documents. The similarity of each fingerprint gives us an idea of plagiarism in a set of documents. Using the winnowing algorithm on each level of source code modifications as shown in Figure 2.1 would provide a more interesting approach. From **Level 1 - Level 3**, we may not apply the winnowing algorithm to identify similarities in programs. Since, comments, keywords and the position of variables are not essential in identifying similarities, we may choose to ignore the use of the winnowing algorithm. On the other hand, if comments in a program are the same, then it probably indicates

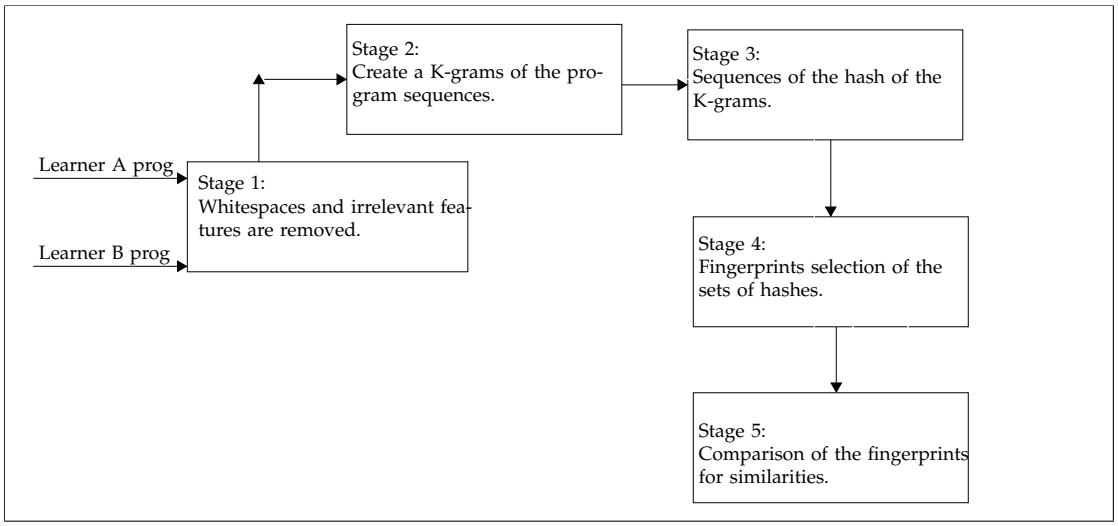


FIGURE 2.5: Steps in the Winnowing Algorithm [Schleimer et al., 2003]

copying. **Level 4 - Level 6** would provide a better usage of the winnowing algorithm to identify similarities.

2.6.2 Levenshtein Distance

The Levenshtein distance also known as *edit distance* provides a similarity measure between two strings [Ristad and Yianilos, 1998; Bilenko and Mooney, 2003]. The algorithm was first considered by Vladimir Levenshtein to measure the distance between strings in a larger document collection [Biswas and Paul, 2010]. Plagiarism detection using the Levenshtein distance has been feasible with texts, but is less plausible in the application of the algorithm to source code plagiarism due to comparison issues. This is as a result of whitespaces, characters or strings in programs, and the presence of different variable names that exist in source codes to be able to match similarities [Siregar, 2015].

Su et al. [2008] highlighted that a minimum number of operations such as insertion, deletion and substitution occur when applying the Levenshtein distance algorithm to transform one string into another string. To calculate the edit distance between two strings *"Universe"* and *"University"*, the edit distance would be 3 as shown in Table 2.1. We would have to substitute the last *e* for *i*, add *t* and add *y*. The steps of the Levenshtein algorithm would apply three operations (insertion, deletion and substitution) when transforming the first string into the other string. Applying the Levenshtein distance algorithm to the modification steps highlighted in Figure 2.1 is not convincing enough to detect plagiarism, because its operation (insertion, deletion and substitution) can be easily altered by even an inexperienced programmer to deceive an instructor.

TABLE 2.1: Levenshtein Distance between two Strings (Universe and University)

		U	n	i	v	e	r	s	i	t	y
	0	1	2	3	4	5	6	7	8	9	10
U	1	0	1	2	3	4	5	6	7	8	9
n	2	1	0	1	2	3	4	5	6	7	8
i	3	2	1	0	1	2	3	4	5	6	7
v	4	3	2	1	0	1	2	3	4	5	6
e	5	4	3	2	1	0	1	2	3	4	5
r	6	5	4	3	2	1	0	1	2	3	4
s	7	6	5	4	3	2	1	0	1	2	3
e	8	7	6	5	4	3	2	1	1	2	3

A similarity value for measuring the similarity of two strings holds if the following conditions apply [Siregar, 2015]:

1. *The similarity value for entirely two different strings should be 0* : This condition holds if the two strings are completely different.
2. *The similarity value for identical strings should be 1*: This condition holds if the two strings are completely identical.
3. *The similarity value for two strings should be between 0 and 1*: This condition holds for any similarity value for two strings.

2.7 Graph-based Plagiarism Detection

Over the years, a number of graph-based plagiarism detection tools have been developed to address the issues of source code plagiarism. In the next sections, we describe the contributions of these tools. The drawbacks posed by these tools are also compared to the plagiarism detection system designed in this work.

2.7.1 Program Dependency Graph

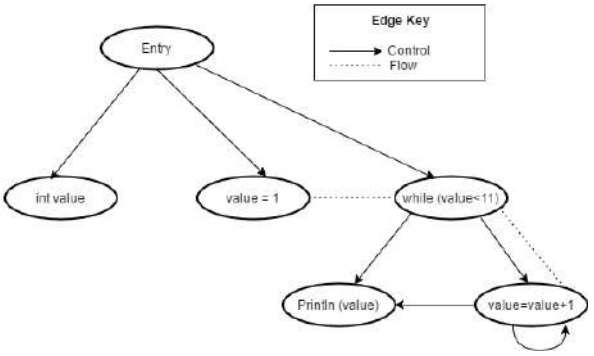
Ferrante et al. [1987] introduced the *Program Dependency Graph* (PDG), a *directed* graph, that provides a unifying framework for program optimisation. Horwitz and Reps [1992] defined the PDG for a program as a *directed* graph, whose vertices are connected by several sets of edges. Krinke [2001] proposed the use of the PDG approach in identifying similarities in programs. The approach considers the syntactic structures and data flows within programs to measure plagiarism. In PDGs, basic statements such as procedure calls, variable declarations and assignments, are represented by program vertices

[Liu et al., 2006]. The *vertices* represent program statements and the control predicates that occur in a program. The directed *edges* represent the dependencies given by the data flow in the program.

Komondoor and Horwitz [2001] proposed the use of PDGs and program slicing to identify similarities in programs. Although the approach was successful in finding similarities in programs in which statements were reordered and the programs intertwined to deceive an instructor, the running time was considerable. For example, it took 3 hours to analyse 250 programs, which is much slower compared to the plagiarism detection tool chosen for this work. Another PDG tool by Krinke [2001] suffered the same problems. Figure 2.6a and 2.6b represent the program and its dependency graph.

```
1 public class Testloop {
2  public static void main(String[] args) {
3     int value = 1;
4     while (value < 11) {
5       System.out.println(value);
6        value++;
7     }
8  }
9 }
```

(A) A sample program



(B) A program dependency graph of the program

FIGURE 2.6: *An example of a program with its program dependency graph. The arrows show the control information between the statements of the program and the dotted lines represent the flow of information in the program.*

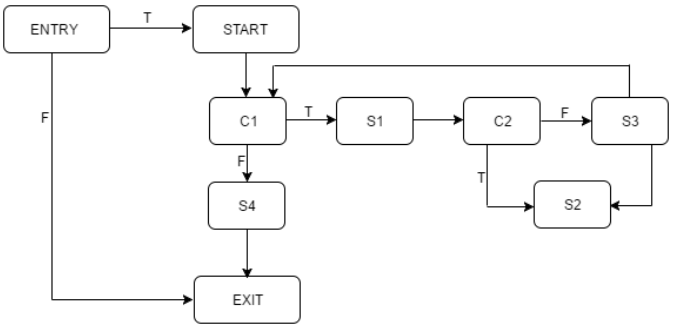
2.7.2 Control Flow Graph

A *control flow graph*, CFG[1], is a *directed* graph, whose *nodes* are the basic blocks of a program represented by two additional nodes, *Entry* and *Exit* [Cytron et al., 1991]. Furthermore, an edge connects the basic statements of the program from the *Entry* to the *Exit*. CFG has been successfully applied to data mining and artificial intelligence approaches to analyse smartphone malware samples and categorise them into families based on code structures presented by CFG [Jiang and Tan, 2005; Cesare and Xiang, 2010; Grace et al., 2012; Suarez-Tangil et al., 2014]. Their approaches ensure the investigation of similarities between malware samples. Chae et al. [2013] proposed a plagiarism detection system using an *Application programming interface* (API) labelled CFG (A-CFG). The A-CFG was used to abstract the functionality of a program which hardly changes

[1]Control Flow Graph

```
1  int main( )
2  {
3    while (C1)
4    {
5          S1;
6          if(C2)
7            S2;
8          S3;
9    }
10   S4;
11 }
12
```

(A) A sample program



(B) Control flow graph of the program

FIGURE 2.7: *An example of a Java program with its control flow graph. C1 and C2 are the predicate nodes which represent boolean expressions. The regular statements are represented by S1 to S4.*

by semantic-preserving transformation attacks in programs. Another CFG approach for detecting source code plagiarism was proposed by Chan and Collberg [2014]. The method considered CFGs of known edit distances to measure plagiarism in programs. Figure 2.7a and Figure 2.7b illustrate a control flow graph with its dependencies.

A number of flaws were reported in the system developed by Chan and Collberg [2014]. First, CFGs are arbitrarily expensive and complex to implement. The nodes are usually bounded with two or more nodes (represented by exceptions) which consist of exception handling and lots of *goto* statements. The CFG relies heavily on basic block contents (instructions) to compute the similarity score between programs. A slight change in a program's basic block would manipulate CFG easily. This is a limitation CFG possesses over our plagiarism detection system. Our plagiarism detection system is not easily manipulated because it considers the structural information of programs, rather than just a number of basic statements.

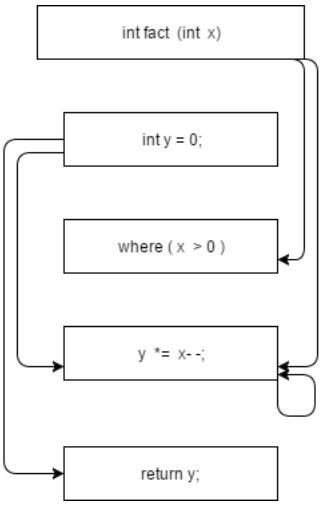
2.7.3 Data Flow Graph

Another graph considered in detecting plagiarism in source code is the *Data Flow Graph* (DFG). Li et al. [2016] proposed the use of the DFG in measuring similarities in programs. The authors highlighted that DFGs are frequently used in program analysis and compiler organisation as they capture the flow of data. Zhang et al. [2012] asserted that a DFG is similar to a CFG, since it represents the *data flows* between the basic blocks in a CFG.

Figure 2.8a and 2.8b shows the program and its data flow diagram. Lutz and Diehl [2014] developed an interactive model for comparing plagiarism using the DFG. The model leverages visual dataflow to allow users to implement comparison strategies

```
1  int fact (int x ) {
2    int y = 1;
3    while (x > 0){
4    y *= x--;
5    }
6    return y;
7    }
8
9  }
10
```

(A) A sample java program



(B) Data flow diagram of the program

FIGURE 2.8: *An example of a Java program with its data flow diagram. The variables used in this example are x and y. The program calculates a number factorial. The instruction in the program are represented by the rectangle. The line around the instructions shows the data flows.*

for source code plagiarism. Beyond that, the model reported a number of flaws. A basic block splitting can disrupt the plagiarism result. That is, a slight modification of program statements can affect the similarity score disproportionately. It would also require a lot of zooming and panning to investigate plagiarism, especially for large program analysis. The DFD possesses a significant flaw compared to the plagiarism detection tool chosen for this work.

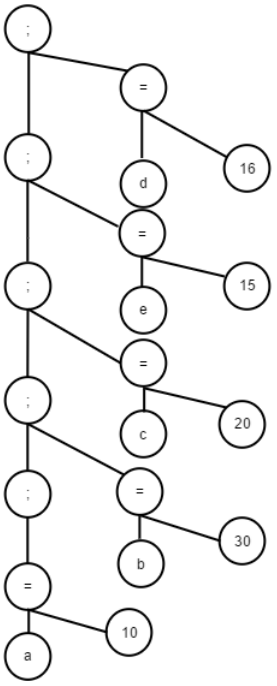
2.7.4 Abstract Syntax Tree

Kontogiannis et al. [1996] proposed an *Abstract Syntax Tree* (AST) representation of a program for measuring similarities in programs. An AST representation was used because, it maintains all necessary information (control or data flow) about programs. Mayrand et al. [1996] defined an AST as a tree-based representation of tokens contained in programs. The AST is an exact replication of a program. Baxter et al. [1998] proposed a clone detection approach using the AST for the detection of exact or near miss clones for arbitrary fragments of programs. Figure 2.9a and 2.9b show an example of a program and its AST used for the detection of clones.

Although the approach used the AST for detecting similarities in the clones and also considered program structures in its metrics, clones can be easily factored out of the original source using a number of conventional methods to obfuscate plagiarism. Their

```
1  void y( )
2   {
3    a = 10;
4    b = 30;
5    c = 20;
6    e = 15;
7    d = 16;
8   }
9
```

(A) A program statement



(B) An Abstract syntax tree of the program

FIGURE 2.9: *An example of a program statements with its abstract syntax tree. The variables used in this example are represented by a, b, c, d and e.* [Baxter et al., 1998]

method was straightforward, easy to implement for large scale program comprehension and abstraction, it however suffers a great deal of *false positives* because of the usage of the tree method in detecting similar clones.

2.7.5 Dotplots and Duploc

Dotplots were proposed by Church and Helfman [1993] for visualising similar codes by converting the source codes into *lines*. For visualising the output of the program, the output can be represented in *scatter plots* which show the number of times each line occurs in matching sections of code. The technique allows interactive manipulation of the *scatter plots* to compare programs. Grozea et al. [2009] developed the *encoplot*, shown in Figure 2.10, which is an alternative to *dotplot* and compares similarities in programs. The *encoplot* uses a linear time sequence matching technique. Another approach considered in this section is the *Duploc*. Ducasse et al. [1999] introduced *Duploc* which uses the *scatter plots* visualisation for finding similarities in programs. *Dotplots* and *Duploc* use the *scatter plots* visualisations for finding similarities in programs. One flaw noticed in both approaches is that they can only detect identical source code pairs, and are intolerant to renaming of program statements. This is a substantial flaw which makes these systems vulnerable to even low level plagiarism obfuscation techniques.

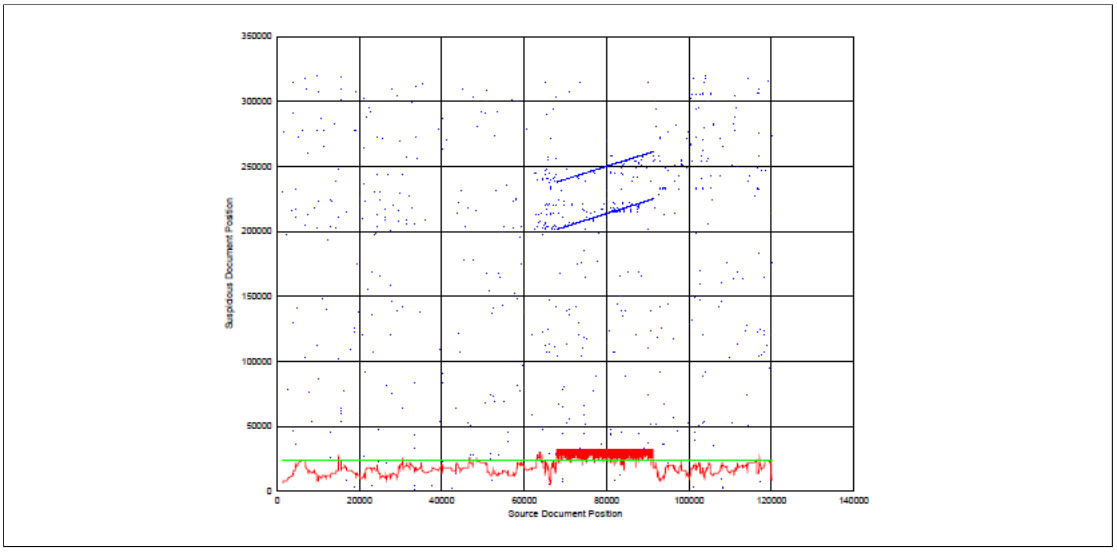


FIGURE 2.10: An encoplot used to compare similarities in programs [Grozea et al., 2009].

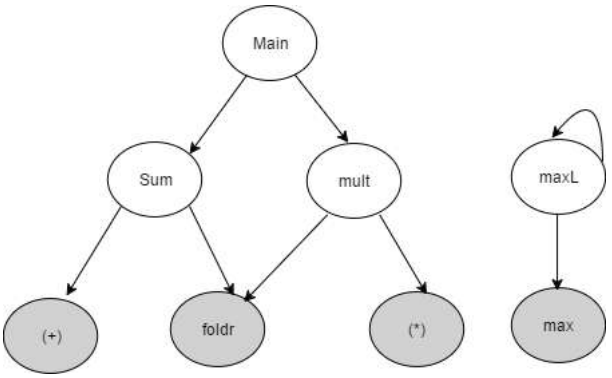
2.7.6 Call Graph

Gascon et al. [2013] proposed the idea of using *call graphs* in detecting malware in the Android operating system. The technique was based on using the structural features of *call graphs* to detect malware. For plagiarism detection, Choi et al. [2007] applied *call graphs* to detect software clones, identify software ownership and similarity of binary executables. The credibility of *call graphs* was tested with other techniques; functions and API calls. The result revealed a success when investigating the techniques for similarities. Kammer et al. [2011] extended the use of *call graphs* in detecting plagiarism in Haskell programs. As described by Kammer et al. [2011], a *call graph* is a *directed*, *labelled* graph that can be *cyclic* if the program contains self-recursive functions or call circles of two or more functions. In a call graph, each edge represents a function calling another function. Figure 2.11a and Figure 2.11b show an example of an Haskell program and its call graph.

Song et al. [2015] proposed a novel approach of using *call graphs* and *parse trees* to find similarities in programs written in Java, C, C++ and Python. The approach considered the structural representation of programs using *parse trees* and the dependencies among function calls within programs, represented as *call graphs*. Although the approaches detected plagiarism easily, the structure is altered easily with a number of *insertions* and *deletions*.

```
1 module Main where
2 main = sum [1..4]
3 sum = foldr (+) 0
4 multL = foldr (*) 1
5 maxL [a] = a
6 maxL (a:as) = max a (maxL as)
7
```

(A) A sample Haskell program



(B) Call graph of the program

FIGURE 2.11: *An example of an Haskell program and its call graph. The sum and foldr functions are part of the haskell prelude functions. The edges are directed to each of the functions [Kammer et al., 2011]*

2.8 Classification Scheme

Classification of program pairs is important to adjudicate actual and coincidental similarities in programs. This is a common machine learning problem. In this section, we take an holistic view on the *confusion matrix concept* to measure the performance of the plagiarism detection system. In this work, we are faced with the challenging issue of classifying which cases of plagiarism are *true* or *false*.

2.8.1 Confusion Matrix Concept

A *Confusion Matrix* contains information about predicted and actual cases carried out by a classification system [Provost et al., 1998]. The *confusion matrix concept* has a size *n x n*, for a classifier for representing actual (columns) and predicted (rows) classification labels, where *n* is the number of the different classes. In the *confusion matrix* for a classifier, there are four metrics used to classify cases. The metrics are *True positive*, *True Negative*, *False Negative* and *False Positive*.

TABLE 2.2: The confusion matrix for a classifier model [Hamel, 2009]

		Observed cases	
		True	False
Predicted cases	True	True Positive (TP)	False Positive (FN)
	False	False Negative (FN)	True Negative (FN)

Table 2.2 shows the confusion matrix of a classifier. In Table 2.2, the metrics that appear along the diagonal of the Table are the correct classifications. The metrics are

True Positive and *True Negative*. The metrics with incorrect or erroneous classification is *False Positive* and *False Negative*. For a model with 100% accuracy, the *False positive* and *False negative* would both be equal to zero.

Each metric is now described in the context of plagiarism detection.

- *True Positive*: The program pair was classified as plagiarised and was actually an instance of plagiarism.
- *True Negative*: The program pair was classified as unplagiarised and was actually not an instance of plagiarism.
- *False Negative*: The program pair was classified as unplagiarised and was actually an instance of plagiarism.
- *False Positive*: The program pair was classified as plagiarised and was actually not an instance of plagiarism.

Given the above metrics, a number of performance evaluation metrics can be derived for a classification system. The performance metrics are *Precision*, *Recall* and *Accuracy* as defined by Powers [2011], are presented:

- *Precision* comprises the proportion of *predicted positive cases* that are classified correctly (*true positives*).
$$\text{Precision}(P) = \frac{TP}{TP + FP} \tag{2.1}$$
- *Recall* comprises the proportion of *positive cases* in the classification system that were identified.
$$\text{Recall}(R) = \frac{TP}{TP + FN} \tag{2.2}$$
- *Accuracy* comprises the total number of *predicted cases* that were identified correctly.
$$\text{Accuracy}(ACC) = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

We can see from the metrics that they evaluate a system performance and are important in evaluating a plagiarism detection system. Furthermore, a confusion matrix can be used to construct a Precision-Recall (PR) curve [Davis and Goadrich, 2006]. In a PR curve, the *recall* is plotted on the *x-axis* and the *Precision* is plotted on the *y-axis* of the PR curve graph. The PR curve is an evaluation tool used for binary classification that allows the visualisation of the performance of a classification system at a range of *thresholds* and is commonly used in the analysis of the classifier [Boyd et al., 2013]. Besides

the visual representation, it is generally used as a general measure of performance, irrespective of any range of thresholds [Moussiades and Vakali, 2005; Goadrich et al., 2006; Liu and Shriberg, 2007; Awad and Elseuofi, 2011; Soliman and Girdzijauskas, 2016]. Figure 2.12 represents a PR curve used to compare the performance of two methods. From the graph in Figure 2.12, the thresholds are represented on the *x-axis* and *y-axis* of the graph. When a classifier uses a threshold, the PR Curve shows the trade-off between false negatives and false positives as one move along the PR curve. A typical example is JPlag [Prechelt et al., 2002]. If the threshold is set to 0, then everything the JPlag system identifies is classified as plagiarism. Therefore, this yields 0 False negatives because, there are no negatives predicted, hence, more false positive exists. As the threshold increases, the false negatives increases than the false positives that exists. The PR curve helps to illustrate this trade-off.

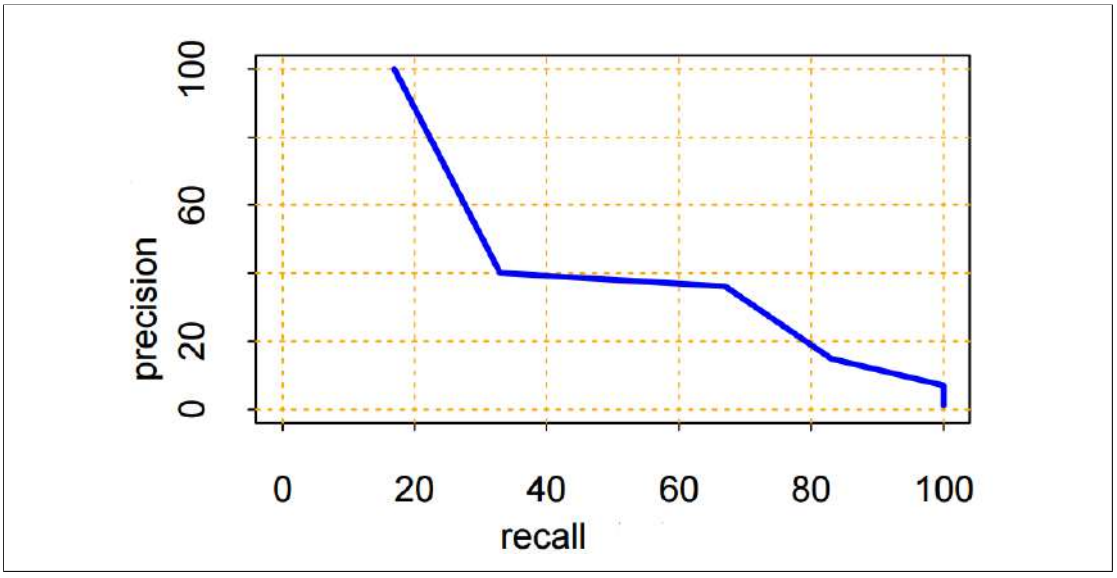


FIGURE 2.12: An example of a PR curve used in evaluating the performance of JPlag over data set of Java submissions [Prechelt et al., 2002].

A *threshold* is needed when accessing the classifier's performance using the metrics provided in the confusion matrix [Manel et al., 2001]. It is a key component to transform the data into performance components. Hence, this transformation of the data will be useful to evaluate our model.

2.9 Conclusion

In this chapter, we have presented the background related to this research. The idea behind source code plagiarism was presented. A number of techniques that students use to transform and obfuscate source code to avoid being caught were also discussed

along with some reasons why students plagiarise their assignments. The reasons generally include *poor knowledge of a programming language*, *academic pressure*, *poor time management* and *self-plagiarism*. Other reasons were highlighted regarding the cause of students plagiarising their assignments; a *lack of knowledge* of a programming language, and largely as a *last resort* when failing a class, are contributory factors as to why students plagiarise. The plagiarism detection methods used to uncover plagiarism in source code were presented. A number of algorithms the plagiarism detection systems considered for detecting plagiarism were presented to unravel plagiarism in students source code assignments. The algorithms considered are the *Levenshtein Distance* and the *Winnowing* algorithm. A number of graph-based plagiarism detection tools were discussed: *Program Dependency Graph*, *Control Flow Graph*, *Abstract Syntax Tree*, *Data Flow Graph*, *Call Graph*, *Dotplots* and *Duploc*. These are all methods of direct plagiarism detection based on the source code itself. This work differs from the methods presented in that, it provides a graph-based analysis of the generated similarity scores. None of the methods surveyed provided this information or looked at the structure of similarities across the class. These methods rather considered pairwise comparisons of the submitted codes.

Lastly, the classification scheme metrics were introduced to better evaluate our model and the visualisation was presented using the *Precision-Recall curve*. The next chapter presents graph theory concept definitions and metrics applied in the area of source code plagiarism.

Chapter 3

Graph Theory Concepts

3.1 Introduction

In the previous chapter, background relating to plagiarism detection was discussed. This chapter is focused on graph theory concepts and relates them to the idea of a plagiarism graph. Section 3.2 introduces the concept of graphs and their application to plagiarism analysis. Section 3.3 is focused on the size and order of graphs. Section 3.4 considers adjacent and incident vertices. Section 3.5 presents the concept of neighbourhoods and vertex degree. Section 3.6 discusses connected components. Section 3.7 is focused on articulation points. Section 3.8 discusses the union and intersection of graphs. Section 3.9 concludes the chapter.

3.2 Concept of Graphs

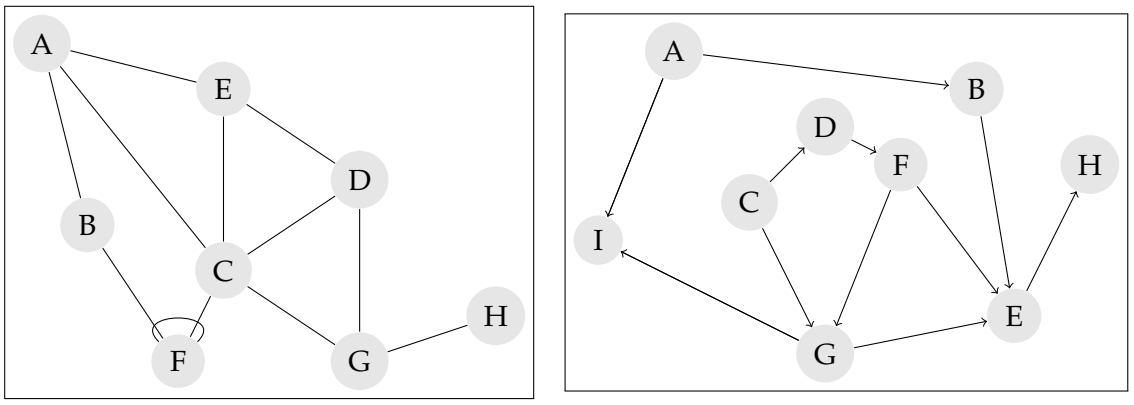
Graphs provide a convenient way of representing many real-world situations [Bondy and Murty, 1976]. In this section, the result generated by MOSS is expressed as a *graph* to visually and accurately represent program similarity.

Definition 3.1. A graph is a mathematical structure which comprises a finite set of *vertices* **V**, and a finite set of *edges* **E** connecting the *vertices* [Gross and Yellen, 2005]. Examples of generic graphs are presented in Figure 3.1.

Graphs are *undirected* and *directed*. A graph is said to be an *undirected* graph, if there is no direction in the edges that link the vertices in the graph. Figure 3.1a is an example of an *undirected* graph, the set of *vertices* **V** or *nodes* and the set of *edges* **E** connecting the *vertices* are given as:

$\mathbf{V} = \{A, B, C, D, E, F, G, H\}$
$\mathbf{E} = \{\{A, B\}, \{A, C\}, \{A, E\}, \{B, F\}, \{C, D\}, \{C, E\}, \{C, F\}, \{C, G\}, \{D, E\}, \{D, G\}, \{F, F\}, \{G, H\}\}$

A graph is said to be a *directed* graph if there exists direction in the edges that link the vertices in the graph. Figure 3.1b is an example of a directed graph, the set of *vertices* **V** or *nodes* and the set of *edges* **E** connecting the *vertices* is given as:
$\mathbf{V} = \{A, B, C, D, E, F, G, H, I\}$
$\mathbf{E} = \{\{A, B\}, \{A, I\}, \{B, E\}, \{C, D\}, \{C, G\}, \{D, F\}, \{E, H\}, \{F, E\}, \{F, G\}, \{G, E\}, \{G, I\}\}$



(A) An undirected graph (B) A directed graph

FIGURE 3.1: Types of graphs

Graphs can also be *weighted* or *unweighted*. In *weighted* graphs, such as that shown in Figure 3.2, each edge is assigned a numerical value. The meaning depends upon the interpretation of the graph. For example, if the graph represented roads and cities, *weighted* edges may indicate the travel time, distance, or cost of traversing the road.

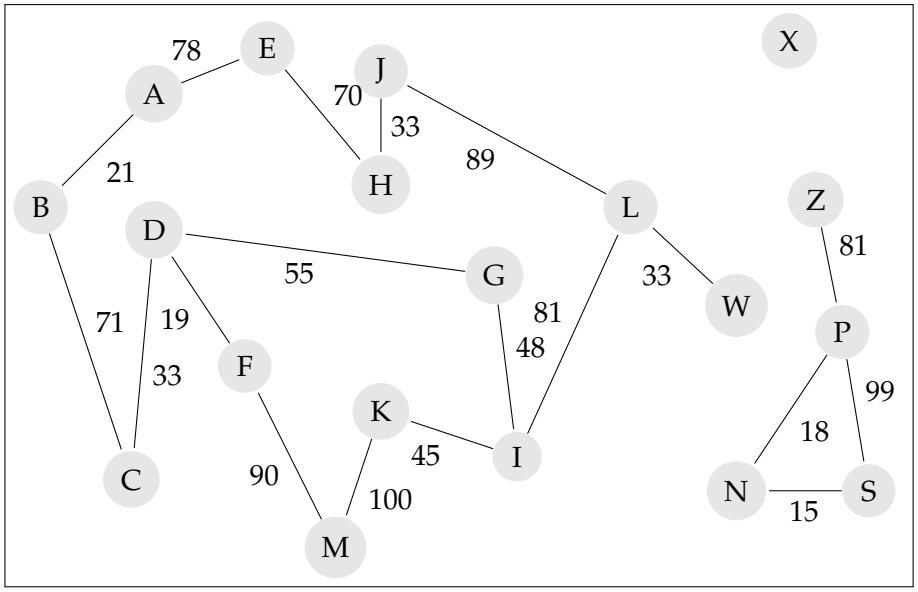


FIGURE 3.2: A weighted graph

In an *unweighted* graph, such as that presented in Figure 3.3, there is no numerical value assigned to an edge. For example, if the graph represented a social network where nodes represent individuals, the *unweighted* edges between them would indicate if they are friends or not.

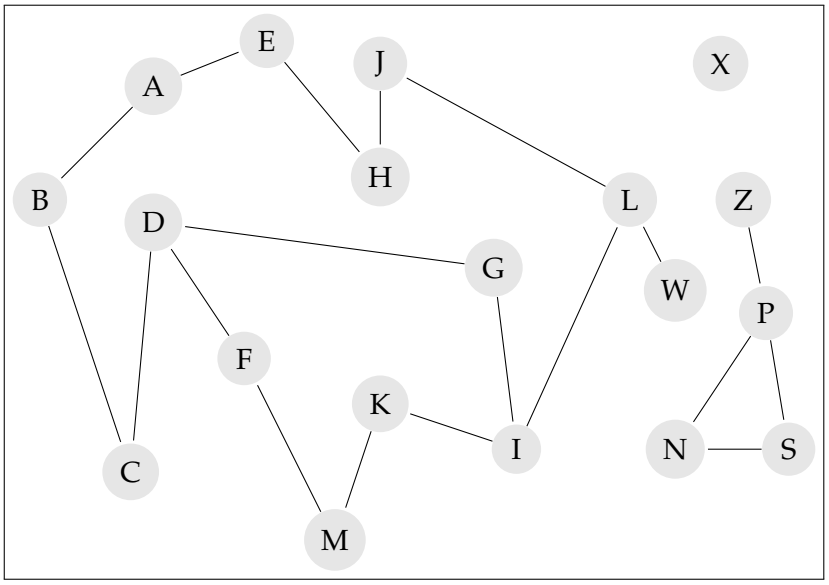


FIGURE 3.3: An unweighted graph

In this work, graphs are used as *plagiarism graphs*. A plagiarism graph is an *undirected, weighted* graph, which represents the structure of plagiarism in the class. Suppose that each *node* (A to Z) as depicted in Figure 3.2 represents a student submission. An edge between the nodes indicates that the submissions are similar, while the numerical *weight* of that *edge* indicates the level of similarity. For example, the level of similarity between the "Student A" program and "Student E" program is 78. This representation allows the intuitive visualisation of the structure of plagiarism in the class.

From the definition of *graphs*, many extensions to this basic definition exist from which *vertices* of a *graph* and its *edges* have several attributes. Other concepts of graphs and their application to source code plagiarism will be discussed holistically.

3.3 Size and Order

Definition 3.2. The **size** of a graph **G**, is defined as the number of *vertices* that exist in the graph **G**. The **order** of a graph **G**, is defined as the number of *edges* that exist in the graph **G**.

Mathematically, the size and order of a graph are given as:

$$\text{Size}(\mathbf{G}) = |V|$$
$$\text{Order}(\mathbf{G}) = |E|$$

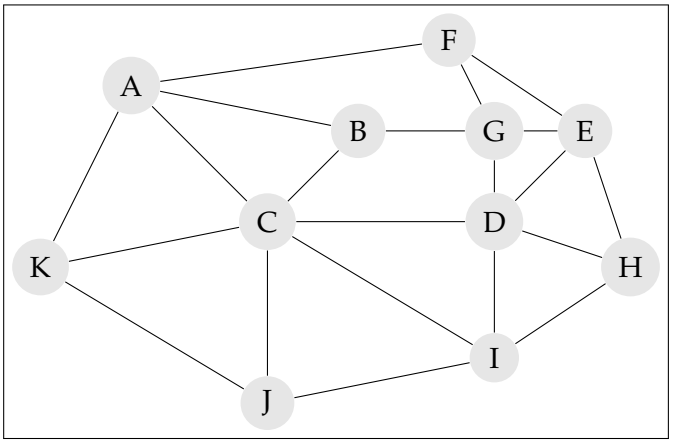


FIGURE 3.4: An undirected graph: Size and Order

In the graph **G** shown in Figure 3.4, the number of *vertices*, denoted as $|V| = 11$, is the **size** of the graph. The number of *edges*, denoted as $|E| = 20$, is the **order** of the graph.

In a plagiarism graph where the *vertices* correspond to the students' programs and *edges* correspond to the pairs of the copied programs above some similarity *threshold*, we can infer that the number of students who participated in plagiarism is 11. Also, the number of pairs of the copied students' programs in the class is 20. We can establish that a student program is similar to two or more students' programs above the *threshold*. In this section, the size and order of the graph have been used to establish relationships between the students' programs. In the next section, the adjacency and incident vertices that exist in a graph are discussed.

3.4 Adjacency and Incidence

Definition 3.3. In a graph **G**, *vertices* are **adjacent** to each other if they share an edge. A *vertex*, v, and an edge, e, are **incident** if one endpoint of e is v.

Mathematically, the adjacency and incidence of a graph are given as:

$$\{v_1, v_2\} \text{ are adjacent } \textit{iff } \textit{edges } (v_1, v_2) \in \text{E}$$

$$e \in \text{E is incident to a } \textit{vertex } v \in \text{V } \textit{iff } v \in f(e)$$

From the graph **G** in Figure 3.5, the *vertices*, A and K are **adjacent** and the *edge* $\{B, E\}$ is **incident** to the *vertex* B and *vertex* E.

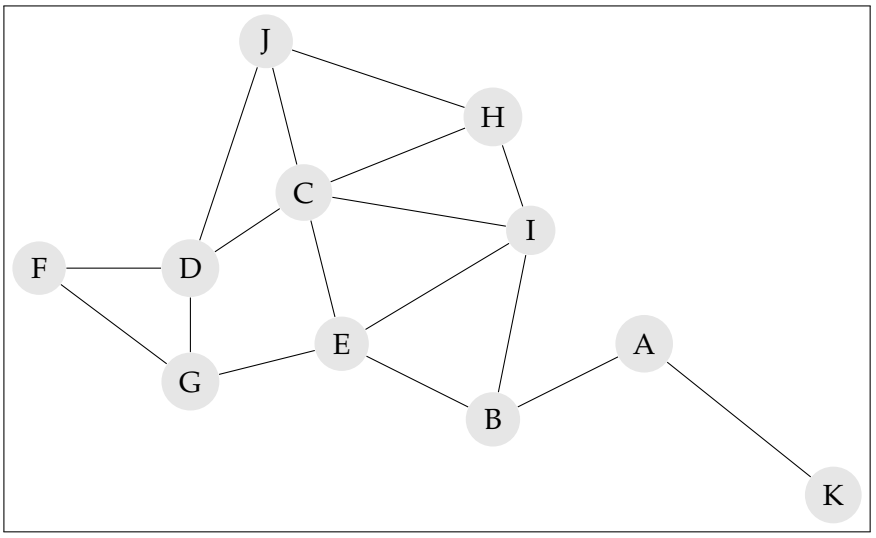


FIGURE 3.5: An undirected graph: Adjacency and Incidence

In a plagiarism graph where the *vertices* correspond to the students' programs and *edges* correspond to the pairs of the copied students' programs above some similarity *threshold*, **adjacency** of the graph would indicate that the students' programs are similar above some similarity *threshold* and there is potential plagiarism. In addition, **incidence** of the graph would indicate that the pairs of the students' programs share a relationship. In this section, the adjacency and incident vertices of a graph have been presented. The next section discusses the neighbourhood and degree of a graph.

3.5 Neighbourhood and Degree

Definition 3.4. The **neighbourhood** of a *vertex* in a graph **G** is the set of *vertices* adjacent to the *vertex*. The **degree** of a *vertex* in a graph is the number of incident *edges* or the size of the **neighbourhood** of that *vertex*.

Mathematically, the neighbourhood and degree of a graph are given as:

$$\text{Neighbourhood} = \{v \in \text{V} \mid edge(u,v) \in \text{E}\}$$

$$\text{degree} = |\text{Neighbourhood}|$$

In the graph **G** presented in Figure 3.6, the **neighbours** of the *vertex E* are $\{F, D, J\}$. The **degree** of the *vertex E* is 3. Table 3.1 shows the **degree** and **neighbours** of each *vertex* in the graph illustrated in Figure 3.6.

In a plagiarism graph where the *vertices* correspond to the students' programs and *edges* correspond to the pairs of the copied students' programs above some similarity

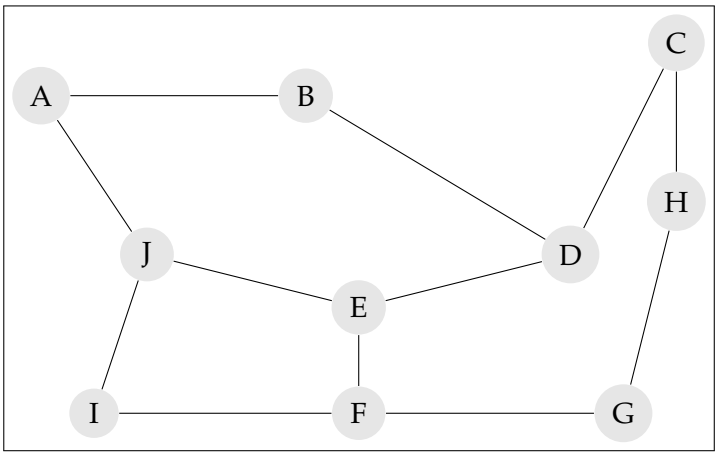


FIGURE 3.6: An undirected graph: Neighbourhood and Degree

TABLE 3.1: Neighbours and Degree in the graph

Vertex	**Neighbours**	Degree
A	B , J	2
B	A , D	2
C	D , H	2
D	B , C , E	3
E	D , F , J	3
F	E , G , I	3
G	F , H	2
H	C , G	2
I	F , J	2
J	A , E , I	3

threshold, the **neighbours** of each program would indicate that the students are actually copying each other. The **degree** would provide us with the total number of students' programs that are similar to that specific program. The next section introduces connected components in graphs.

3.6 Connected Components

Definition 3.5. For **connected components** in a graph, there is a *path* from one *vertex* to every other *vertex* in the component.

Mathematically, the connected components of a graph are defined as:

C $\subseteq$ V is a connected component *if* $\forall\, v \in C$, $\forall\, w \in C$, $\exists$ a *path* starting at v and ending at w and all *vertices* in the *path* are in C.

In the graph presented in Figure 3.7, the **connected components** are $\{A, B, C, D\}$ and $\{F, G, H, I\}$

FIGURE 3.7: An undirected graph: Connected Components

In a plagiarism graph, **connected components** will provide us with the groups of students that are actually plagiarising each other. The next section introduces the articulation point in a graph.

3.7 Articulation Point

Definition 3.6. An **articulation point** in any plagiarism graph **G** is a set of *vertices* in which, when a *vertex* in the graph is removed, it breaks the plagiarism graph into two or more pieces (connected components). A graph without an articulation point is called a *biconnected* graph.

Mathematically, the articulation points in a graph is given as:

A *vertex* $v \in V$ is an articulation point of G if (G-v) has more connected components than G. i.e. c(G-v) $>$ c(G). Where c(G)= Number of connected components in G.

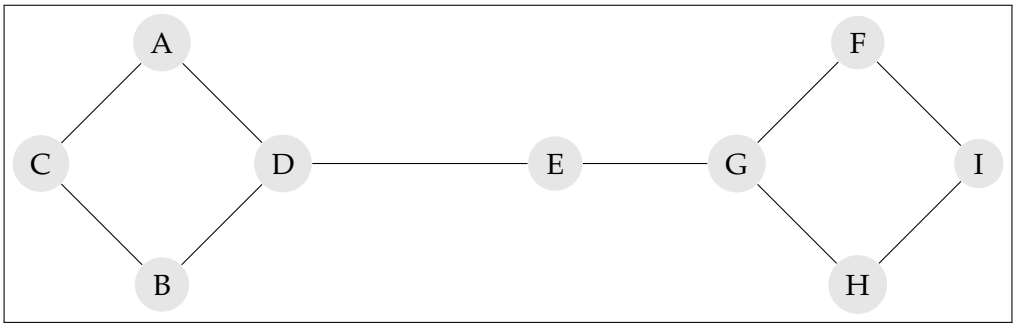


FIGURE 3.8: An undirected graph: Finding the articulation points

In the graph presented in Figure 3.8, to find the **articulation points** in the graph, removing any of the *vertices* $\{D, E, G\}$, divides the graph. Removing the vertex E, divides the graph into two biconnected graphs as depicted in Figure 3.9. If the set of vertices, $\{D, E, G\}$, are removed from the graph, the **articulation points** of the graph would be at C and I as presented in Figure 3.10.

FIGURE 3.9: An undirected graph: Two biconnected graphs

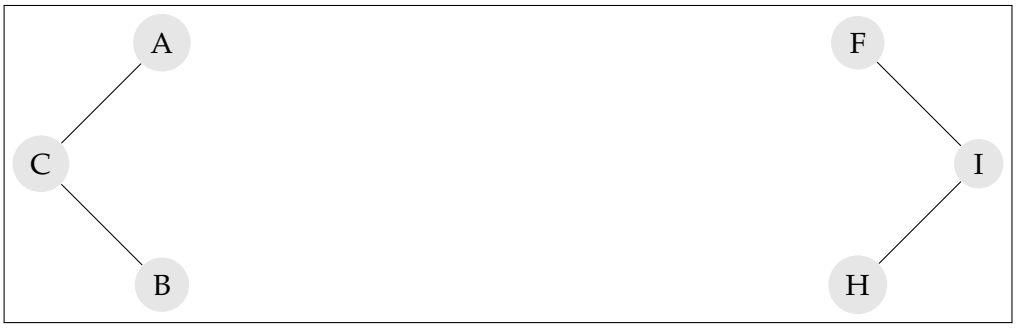


FIGURE 3.10: An undirected graph: Articulation points at *C* and *I*

In a plagiarism graph where the *vertices* correspond to the students' programs and *edges* correspond to the pairs of the copied students' programs above some similarity *threshold*, **articulation points** will provide us with students that are not connected but are copying sections in a program.

3.8 Union and Intersection

Definition 3.7. The **union** of any three graphs is written as $G_1 = \{V_1, E_1\}$, $G_2 = \{V_2, E_2\}$ and $G_3 = \{V_3, E_3\}$ which denotes a simple graph with the set of *vertices* $\{V_1 \cup V_2 \cup V_3\}$ and the set of *edges* $\{E_1 \cup E_2 \cup E_3\}$. For any three graphs, their **union** is $\{G_1 \cup G_2 \cup G_3\}$.

Definition 3.8. The **intersection** of any three graphs written as $G_1 = \{V_1, E_1\}$, $G_2 = \{V_2, E_2\}$ and $G_3 = \{V_3, E_3\}$ denotes a simple graph with the set of *vertices* $\{V_1 \cap V_2 \cap V_3\}$ and the set of *edges* $\{E_1 \cap E_2 \cap E_3\}$. For any three graphs, the **intersection** between them is $\{G_1 \cap G_2 \cap G_3\}$.

Mathematically, the union and intersection of graphs is given as:

A graph **G** is said to be an intersection graph, *if* $\exists$ a *subset* of $E = \{E_1, E_2, ..., E_n\}$ with a set of vertices $\{V_1, V_2, ..., V_n\}$ and a set of edges $\{E_i, E_j\}$ for all *i* and *j* ($i \neq j$) with $E_i \cap E_j \neq \emptyset$

A graph **G** is said to be a union graph, *if* ∃ a *subset* of $E = \{E_1, E_2,...,E_n\}$ with a set of vertices $\{V_1, V_2,...,V_n\}$ and a set of edges $\{E_i,E_j\}$ for all *i* and *j* (*i*=*j*) with $E_i \cup E_j = \emptyset$

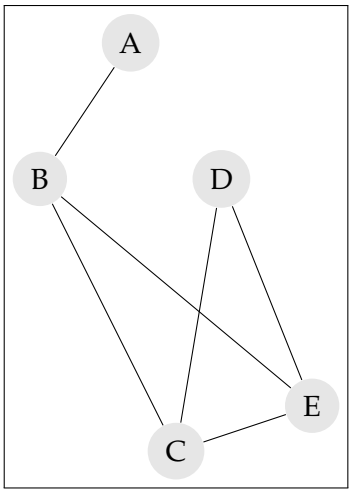


FIGURE 3.11: Graph G_1: An undirected graph: Union and Intersection

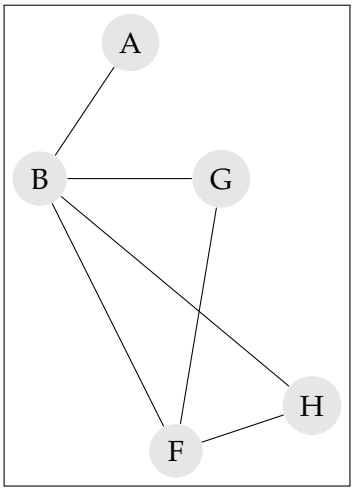


FIGURE 3.12: Graph G_2: An undirected graph: Union and Intersection

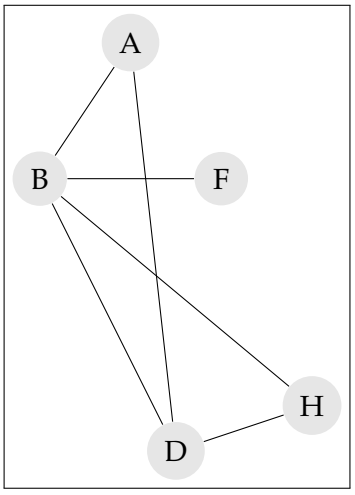


FIGURE 3.13: Graph G_3: An undirected graph: Union and Intersection

To find the **union** of these graphs as presented in Figure 3.11, Figure 3.12 and Figure 3.13, we would make a combination of the three graphs and derive some relationships amongst them.

The first graph, G_1, with the set of *vertices* $\{A, B, C, D, E\}$ and the set of *edges* are {{A, B}, {B, C}, {B, E}, {C, D}, {C, E}, {D, E}}.

The second graph, G_2, with the set of *vertices* $\{A, B, F, G, H\}$ and the set of *edges* are {{A, B}, {B, F}, {B, G}, {B, H}, {F, G}, {F, H}}.

The third graph, G_3, with the set of *vertices* $\{A, B, D, F, H\}$ and the set of *edges* are {{A, B}, {A, D}, {B, D}, {B, F}, {B, H}, {D, H}}.

The **union** of the three graphs is $G_1 \cup G_2 \cup G_3$ with the set of *vertices* $\{A, B, C, D, E, F, G, H\}$ and set of edges {{A, B}, {A, D}, {B, C}, {B, D}, {B, E}, {B, F}, {B, G}, {B, H}, {C, D}, {C, E}, {D, E}, {D, H}, {F, G}, {F, H}}.

To find the **intersection** in the graphs represented in Figure 3.11, Figure 3.12 and Figure 3.13, we would make a combination of the three graphs and derive the relationship between them. $G_1 \cap G_2 \cap G_3$ is the intersection of the graph, we represent each relationship as $G_1 \cap G_2$, $G_1 \cap G_3$ and $G_2 \cap G_3$.

The **intersection** of the two graphs, $G_1 \cap G_2$, produces the *vertex* set $\{A, B\}$ and the *edge* set $\{A, B\}$. This shows that there is a relationship between the two graphs.

The **intersection** of the two graphs, $G_1 \cap G_3$, produces a set of relationship between the two graphs. The graph produces the *vertex* set $\{A, B, D\}$ and the *edge* set $\{A, B\}$.

The **intersection** of the two graphs, $G_2 \cap G_3$, produces the *vertex* set $\{A, B, H\}$ and the *edge* set {A, B}. This shows that there is a relationship between the two graphs.

The result of taking the union of multiple plagiarism graphs is a plagiarism graph that indicates if two students have ever plagiarised each other over a number of assignments. For example, if "Student B" plagiarised "Student C" in the first assignment and plagiarised "Student F" in the second assignment, then the union will have edges connecting "Student B" with both of the other two students.

On the other hand, the intersection of plagiarism graphs over a number of assignments indicates that the students have plagiarised each other in every assignment. This is an extremely strong indication of plagiarism. For example, if "Student A" plagiarises from "Student B" in the three assignments, then the students will be adjacent in the intersection graph, and one can be confident that the similarities were not coincidental as they occurred in every submission.

3.9 Conclusion

In this chapter, we have presented the graph theory concepts and related them to the analysis of source code plagiarism in a class. The definition of graphs was presented. A number of graph metrics were provided. The definitions provided would be important in this research, since this research is focused on using the idea of graphs to present cases of plagiarism.

The next chapter introduces the questions for this research and the methods used to carry the detection meticulously.

Chapter 4

Research Questions and Methods

4.1 Introduction

The previous chapters presented the background and related work in plagiarism detection, and a number of graph definitions which are used to analyse the structure of plagiarism within a class were discussed. The aim of this research is discussed in Section 4.2. The questions for this research are formulated in Section 4.3. The methodology for carrying out this research is outlined in Section 4.4. Section 4.5 concludes the chapter.

4.2 Aims

This research aims to determine which cases of measured similarity might be plagiarism. In order to fulfil the aim of this research, we would have to carry out detection on the students' programs, represent the measured programs in form of *graphs*, and manually differentiate actual plagiarism from coincidental similarities. MOSS, discussed in the background of this work, Chapter 2, is to be implemented as a plugin, integrated into graphs to check for plagiarism in source codes on a series of programming languages taught at the *University of the Witwatersrand, Johannesburg*. Graphical representation of source code similarities is very vital. This helps to identify the actual representation of corresponding nodes as discussed in Chapter 3, which is an area yet to be explored in the area of source code plagiarism. Implementations should run on the CSAM (Computer Science and Applied Mathematics) LMS[1] of the University, which must be easy to integrate and effortless to use.

[1] Learning Management System

4.3 Research Questions

The following questions will be answered in this work to better understand the idea of identifying cases of source code plagiarism.

1. *At what threshold is it sufficient to indicate that a program has been plagiarised?*
2. *Does the graph-based approach highlight useful structural information to help instructors understand detected program similarity?*
3. *Can we identify culprits/plagiarists across multiple assignments of the same course?*

4.4 Methodology

Even though all of the tools discussed in the literature of this work have been effective in detecting plagiarism, it is understood that no detection system is *fool-proof*. The methodology of this work was conducted and completed in a number of predefined phases:

1. *Pairwise program similarity check*
2. *Graph visualisation of programs*
3. *Manual classification of the result*

At the *pairwise program similarity check* phase, a plugin for MOSS, which already existed but had some *deprecated* codes was used in this work. The MOSS plugin sends the submissions to MOSS which extract programs into tokens, and generate a fingerprint of the tokens as the similarity between programs. This was discussed in the literature of this work, Chapter 2. The MOSS plugin aims to achieve the following:

- *Detection of programs from a pool of programming assignments*.
- *Perform a pairwise comparison between two or more programs indicated by a similarity value*.
- *Representation of the programs in histograms in order of precedence*.

More details of this phase are provided in Chapter 5.

At the *graph visualisation* phase, the programs will be represented in the form of graphs, and this is important to structurally analyse the copied programs. This aspect of the work was discussed in the introduction of the concept of graphs and their application to plagiarism analysis, Chapter 3. Graphs can aid an instructor to grasp the information that the detection conveys, although, the MOSS plugin represents a number of similarities by means of pairwise comparison. With the aid of the graph, the information conveyed by MOSS will be easily understood and will save an instructor extra time in identifying actual plagiarism. At the *graph visualisation* phase, the aim is to achieve the following:

- *Representation of the information generated by MOSS in a compact way to convey the information*.
- *Assist an instructor in effective decision-making*.
- *Ensure a comparative analysis of the students programs*.

This phase is important in this work because our aim is to make it easier for an instructor to carry out the detection of source codes meticulously. More explanation of this phase is provided in Chapter 6.

The *manual classification of the result* phase is important to make an accurate decision of the detection and visualisation phases. The classification will be done at the ground truth analysis phase. Our aim at this phase is to *differentiate actual plagiarism* from *coincidental cases* and to *evaluate the plagiarism detection system*. More details of this phase are discussed in Chapter 7.

4.5 Conclusion

In this chapter, the aim of this research was discussed. The questions that will be answered in this research were outlined and the methods for carrying out this research were introduced. The idea of using the MOSS plagiarism detection system to check for similarities in programs, was introduced. While this approach offered numerous advantages, the idea of representing each source code pair as graphs, was introduced. The graph visualisation will make it possible to trace section of the copied programs. Finally, the report of the detection system will be classified in a number of groups with some specified characteristics. This is important to differentiate actual plagiarism from coincidental cases.

Chapter 5 is concerned with the design and implementation of the plagiarism detection system.

Chapter 5

Design and Implementation

5.1 Introduction

The previous chapter outlines the aims, questions to be answered and methods of this research. This chapter is focused on the design and implementation of the plagiarism detection system. Section 5.2 introduces the overview of the plagiarism detection system. Section 5.3 is focused on the development of the MOSS Plugin. Section 5.4 is focused on the graph visualisation tool. Section 5.5 discusses the inclusion of these tools into the VLE[1]. Section 5.6 concludes the chapter.

5.2 System Overview

A plugin was developed which uses MOSS [Aiken, 1994] to check for similarities in programs and expose techniques students use to obfuscate plagiarism in programs. The plugin adapted several features of the native MOSS system for checking similarities in programs. The features of the plugin include; authenticating a user, uploading different program files with an *instructor-supplied* code to ignore lines of code that may appear suspicious, and the pairwise comparison of the source code pairs. A portion of the implementation of the MOSS plugin was adapted from the work of Le Nguyen et al. [2013]. The following additions/changes were made to the MOSS plugin developed by Le Nguyen et al. [2013]:

1. Most of the PHP library functions were *deprecated*: The MOSS plugin by Le Nguyen et al. [2013] was designed to work on the old Moodle's VLE which only supported

[1]Virtual Learning Environment

Algorithm 1 : Increase size of MOSS

```
1: Input: A set of student programming assignments
2: Output: The plagiarism report of the student programs
3: while the user clicks the button to scan for plagiarism do
4:     generate the plagiarism report and display similarities
5:     from MOSS according to n-size
6:     if n-size ≤ 5000 then
7:         generate the plagiarism report and show all matching
8:         similarities within the similar files up to 5000
9:     else
10:        display the plagiarism report and show the matched
11:        similarity up to the default MOSS size of 250
12:    end if
13: end while
```

older PHP library functions. To work for the current version of Moodle, the PHP library functions were replaced to support the current version of Moodle. For example, $HTTP_COOKIES_VAR was replaced by $_COOKIES.

2. The plugin performed better for a small class size: The work of Le Nguyen et al. [2013] performed better for a small class size but not suitable for a larger class size. The MOSS plugin was optimised to work better for a large class size. The pseudocode of this changes is included in Algorithm 1 displaying changes made to the plugin.

3. The graph code was added to the MOSS plugin to visually annotate the program similarities: The Graphviz code was added to the MOSS plugin to visually represent the program similarities. More details about the Graphviz code is presented in Section 5.4.

The implemented plagiarism detection system consists of two parts as shown in Figure 5.1; the development of a MOSS plugin to check for plagiarised pairs of programs, and a graph tool to visually annotate the pairs of the copied programs. Graph visualisation is vital, this shows the structural information of the source code plagiarism.

The next sections explain the development of the MOSS plugin, the graph visualisation, Graphviz tool, to annotate the student programs, and the inclusion of these tools into the VLE.

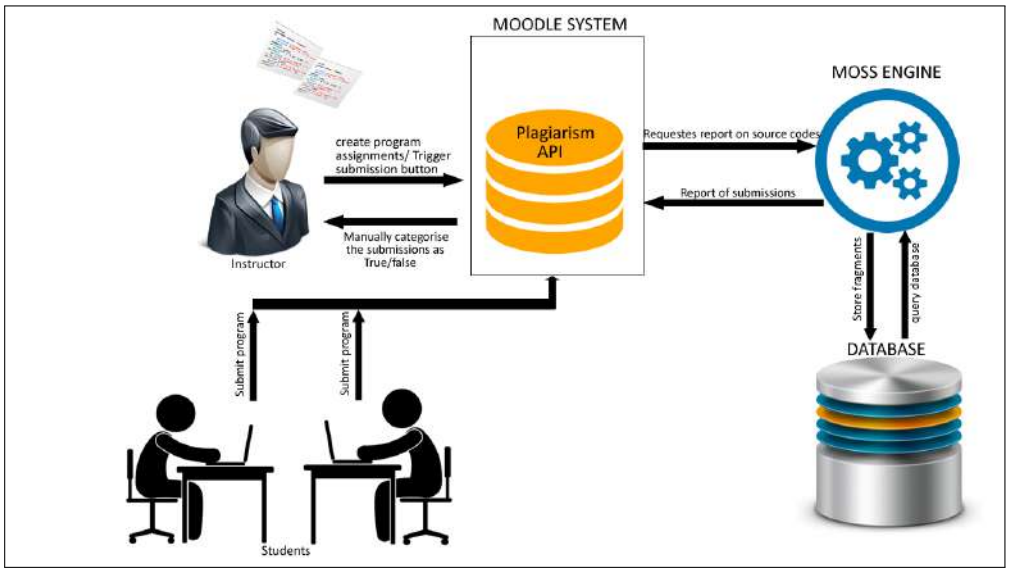


FIGURE 5.1: The plagiarism detection engine

5.3 The Development of a MOSS Plugin

To expose copied programs, a MOSS plugin already existed for the Moodle VLE, developed by Le Nguyen et al. [2013], and was added into the VLE. The plugin was developed based on Moodle's plagiarism *Application programming interface* (API) by the Moodle Developer, Dan Marsden, for developing new plagiarism plugin instances. The Moodle plagiarism API is a core set of functions that Moodle uses to send users' generated contents to external plagiarism detection systems. The plugin employed features of the native MOSS system which requires an instructor to be authenticated before creating and submitting the source code assignments for plagiarism, as presented in Figure 5.1. At first instance of using the plugin, it would require an instructor to obtain a MOSS user ID by sending a request to *moss@moss.stanford.edu*. In sending an email to the MOSS server, the body of the message must contain the following:

registeruser
mail *username@domain*

The *username@domain* is your email address. An auto-generated response is sent from the MOSS server in a Perl script, which contains the MOSS user ID. The MOSS user ID from MOSS authenticates a user whenever a request for a similarity measure for source codes is required from the MOSS server. The authentication activity was embedded in the MOSS plugin, presented in Figure 5.2. Before using the MOSS plugin, a user can copy the entire text generated from the MOSS server or copy the MOSS user ID generated from the Perl script into the textbox as shown in Figure 5.2. The MOSS user ID

is then available to all assignments in the course or even site wide, depending on the choice of the instructor. After authentication, a plagiarism instance is triggered under

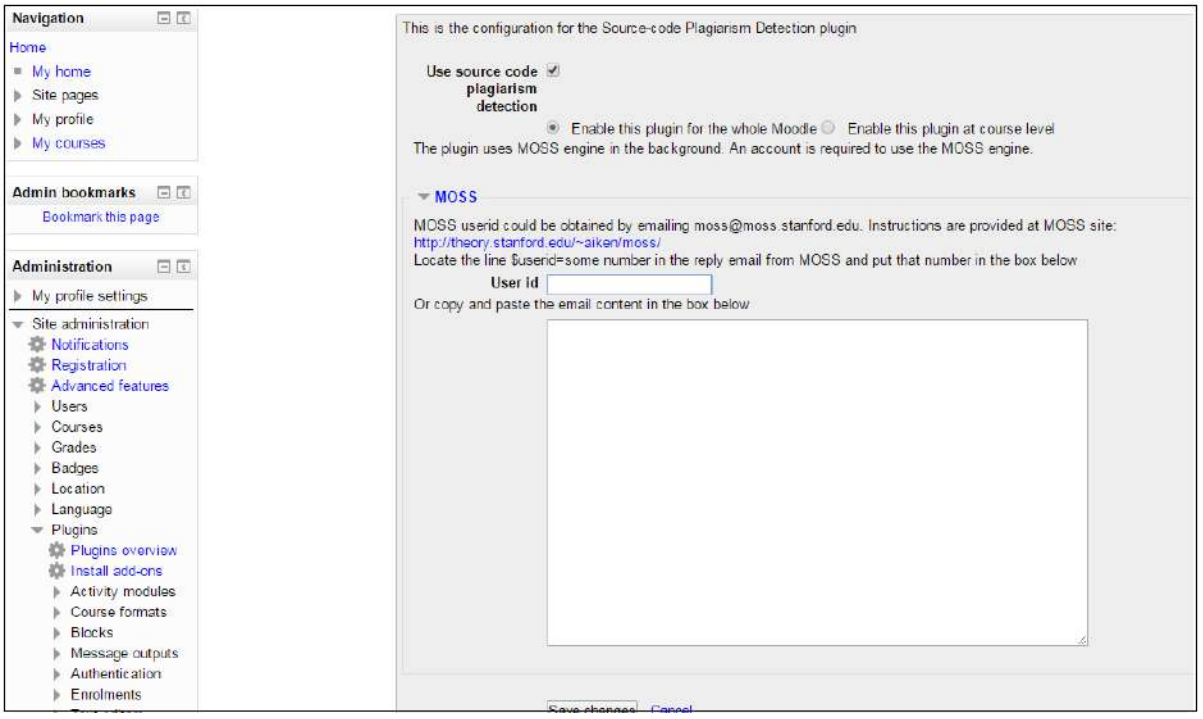


FIGURE 5.2: The authentication stage of MOSS

the assignment module set by an instructor. The assignment module allows instructors to collect assignments from students, check for plagiarism in their assignments and provide feedback, as well as for grading purposes. Recent versions of Moodle presented in Figure 5.3 ensured two file submission types; *the online text type* (copy/pasting source codes) and *the file submissions type* (uploading source code attachments). Students are expected to upload their assignments for plagiarism via the assignment module.

Event handlers are triggered and processed by the plagiarism plugin based on the Moodle API, which contains the details about the student, the module (course) and the submissions that have been carried out. The report of the plagiarism detection is sent to the instructor and a URL for displaying the report is available. One key feature of the MOSS plugin is the similarity rate distribution. The MOSS plugin presents connected pairs of programs and groups them according to a range of similarity thresholds. The similarity rate distribution feature, presented in Figure 5.4 would assist in identifying connected students' programs which fall within a certain threshold, and relationships can be easily established from that distribution.

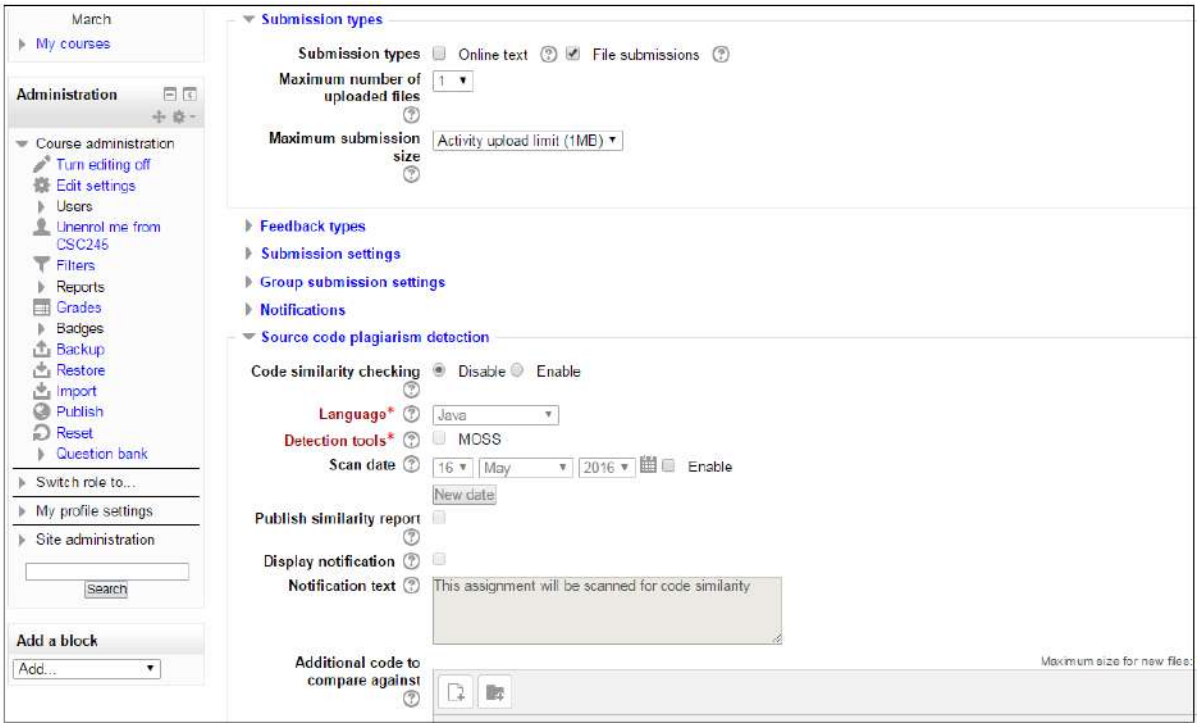


FIGURE 5.3: The assignment activity module of Moodle

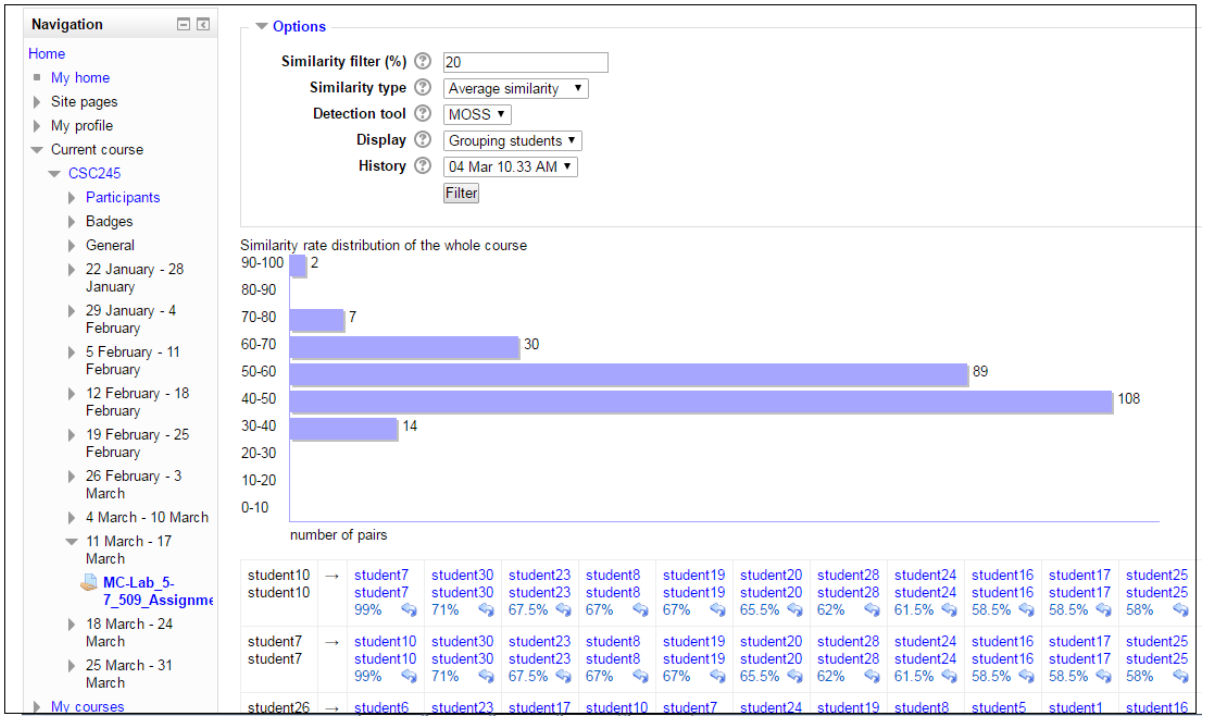


FIGURE 5.4: The similarity rate distribution of the MOSS plugin

5.4 The Graph Visualisation, Graphviz tool

Another tool that was explored during the implementation was the Graphviz tool, developed at the AT&T Research Labs. The Graphviz tool was added to the MOSS plugin developed by Le Nguyen et al. [2013] to visually annotate program similarities. The Graphviz tool is an open source collection of graph drawing tools, specified in the DOT language for drawing graphs (objects are composed of *vertices* and *edges*) [Conte et al., 2003]. The graph visualisation provides a visual representation of structural information such as networks, diagrams and abstract graphs. It has wider application in software engineering, databases, distances between locations, bioinformatics and visual systems. The DOT language is a simple text language from which the graph tool can produce diagrams in useful formats such as images and SVG[2] formats for web pages, in PDF or Postscripts for inclusion into documents, or presenting in an interactive graph browser.

Graphviz provides several features for presenting graphs. The features include options for colours, hyper links, fonts, pen width, label, line width and custom shapes. To render graphs in our implementation, the data file which contains the report generated from MOSS was parsed and the graph was generated. See Algorithm 2 on the implementation of the graph visualisation tool. The generated graphs are stored in a local cache as DOT files, which are easily downloadable to derive useful information. The Graphviz image is generated and displayed within the plugin as seen in Figure 5.5.

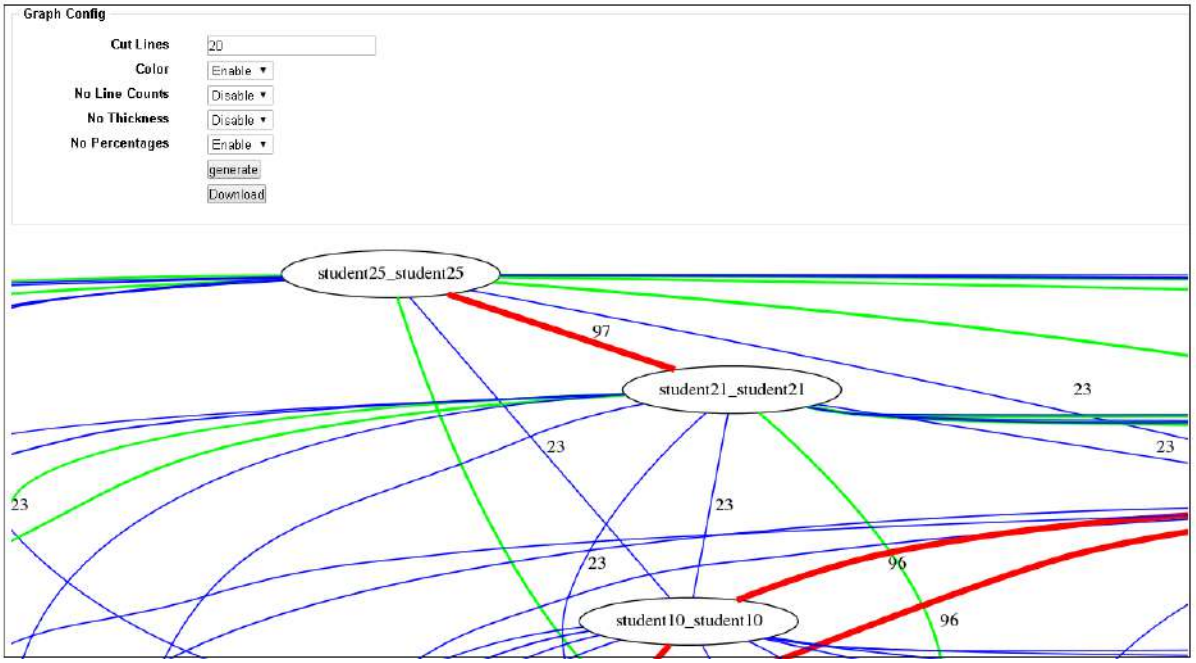


FIGURE 5.5: The plagiarism graph visualisation

[2]Scalable Vector Graphics

Algorithm 2 : Render the graph

```
1: Input: The student programs
2: Output: The plagiarism graph
3: while the user clicks the button to scan for plagiarism do
4:     generate the plagiarism report and display similarities
5:     from MOSS according to n-size
6:     if n-size ≤ 5000 then
7:         generate the plagiarism report and shows all matching
8:         similarities within the similar files up to 5000
9:     else
10:        display the plagiarism report and show the matched
11:        similarity up to the default MOSS size of 250
12:        for each n-size generated from MOSS do
13:            plot a graph with a node connected
14:            by a number of edges from the
15:            similarities generated from MOSS
16:        end for
17:    end if
18: end while
```

Within the plugin, the Graphviz tool possessed different properties which include cutlines, colours, line counts and percentage. The terminologies are explained next. **Cutlines:** Cutlines are a property which is used to prune the plagiarism graph. In the plagiarism graph, when a threshold is set, cutlines delete the nodes and edges that appear below the set threshold. Cutlines are important in plagiarism graphs to show the structure of a graph within a set threshold. For example, if an instructor wants to investigate programs with a higher plagiarism scores of 80 and above, the instructor can determine the plagiarised pairs by setting a threshold score. An illustration of the example is shown in Figure 5.6.

In Figure 5.6, if the threshold is set to 80, the selection will include the students' programs and plagiarism scores that have endpoints attached to that selection.

Colours: The colour attribute specifies colours as hexadecimal values in the *RGB*-Red, Green and Blue format. Graphviz uses colours to identify the edges in the graph. In the plagiarism detection system, if no colour is set, the default colour "black" is enabled. When a colour property is set, the graph shows the colour of each copied program of the class. The colour used in the graph was specified within a certain threshold range. The colour "red" was used to identify a threshold range between 70-100, "green" for a threshold 50-69 and "blue" for a threshold between 0-49. Figure 5.7 shows the colour attribute specified to show the paired portion of the class. For example, an instructor can determine plagiarism in a pool of assignments by setting the colour property of the plugin to "enabled". This makes it easy to identify each program pairs, rather than manually inspecting every program pairs for plagiarism.

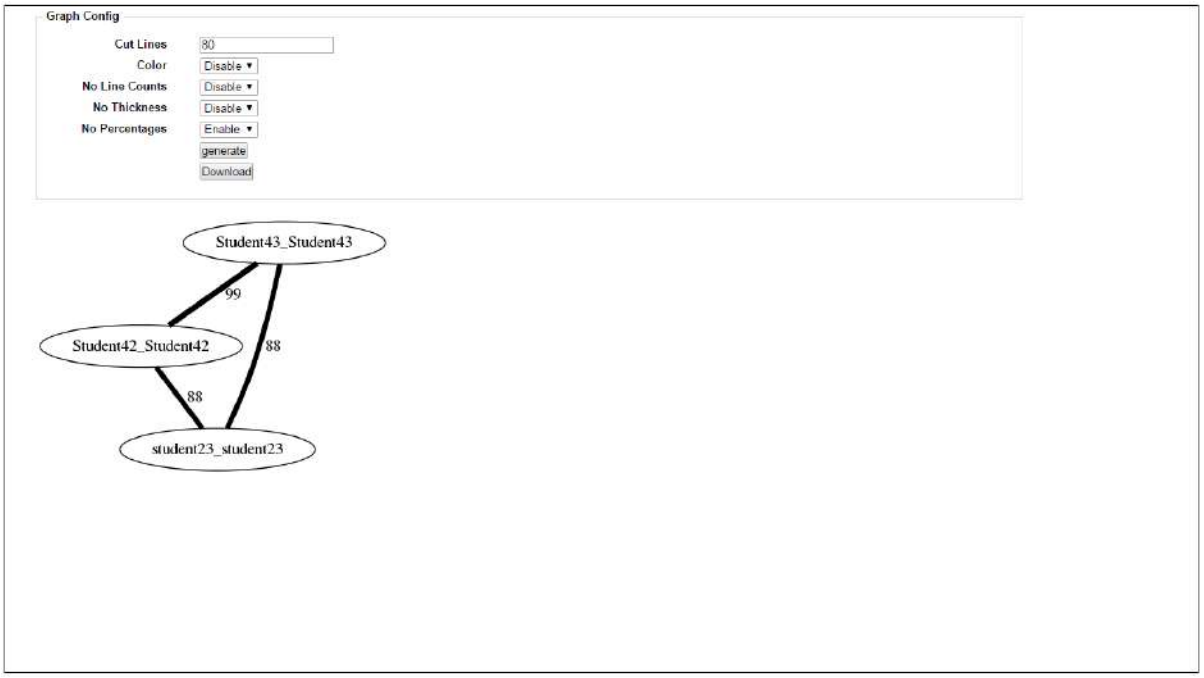


FIGURE 5.6: The structure from a set threshold of the plagiarism graph

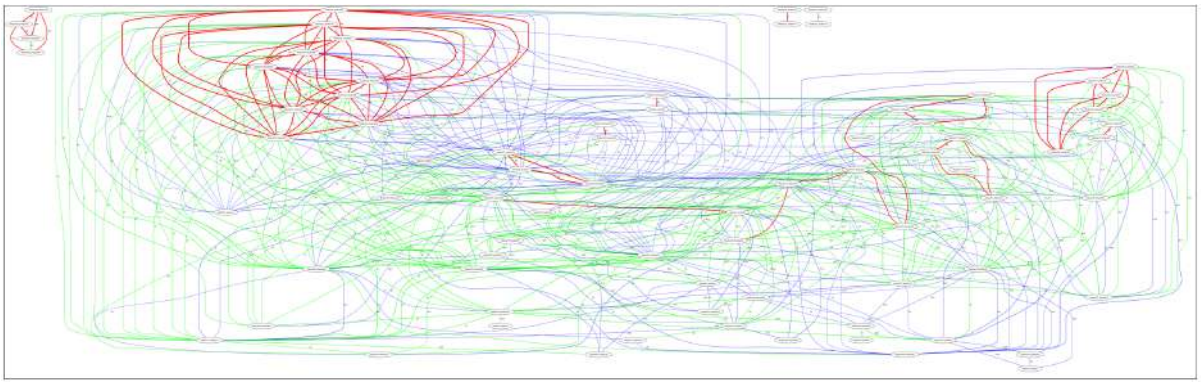


FIGURE 5.7: Colours of the graph structure of a class

Line counts: The line counts of any graph show the number of edges in the graph. The line counts property of a graph was implemented in the plagiarism detection system. If set to "enabled", it shows the number of lines that appeared in the graph. Otherwise, if the line counts property is set to "disabled", the corresponding lines of the graph disappears. An example of the line counts property adapted from the plagiarism graph in the plagiarism detection system is presented in Figure 5.8.

Thickness: The thickness property of the graph is a property of the edges which specifies the borderline of the line weight of the edge. The thickness property of the graph works along with threshold values in the plagiarism detection system. A higher threshold yields the thickness of the edges, while a lower threshold reduces the thickness of the lines in the graph. If set to "enabled", the line of the edges thickens. With the

FIGURE 5.8: The line count feature of the plagiarism graph

thickness property, we can investigate programs whose thresholds are higher and investigate the program pairs for plagiarism. A sample graph which shows the thickness of a graph is showed in Figure 5.9.

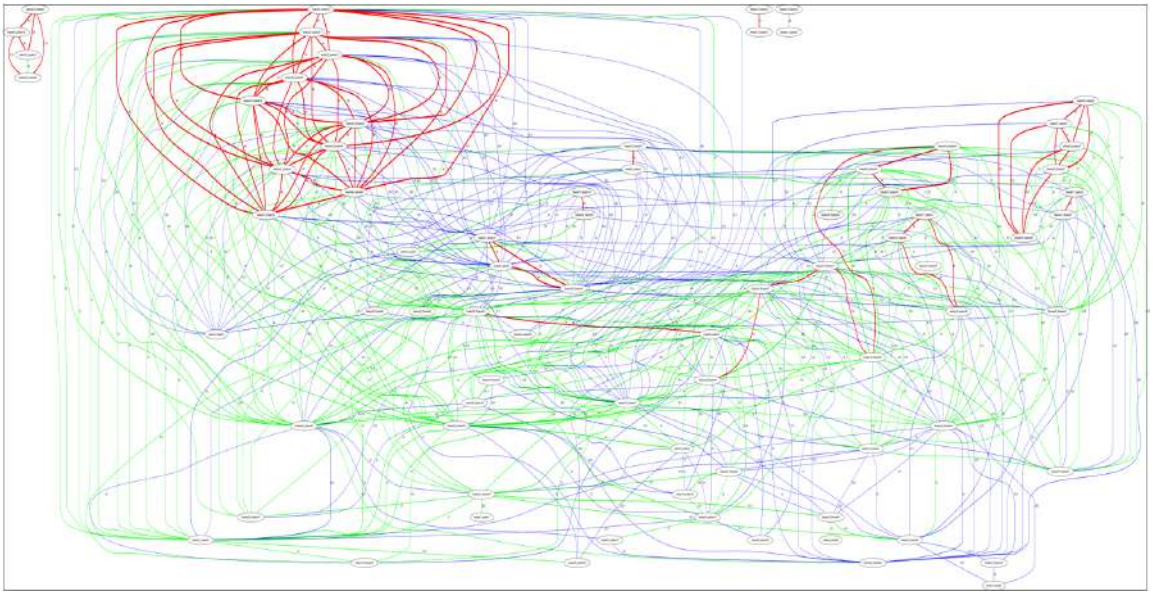


FIGURE 5.9: The thickness of the edges of the plagiarism graph (adapted from the plugin)

Percentage: The percentage property specifies the weight/threshold of the edge in a graph. In the graph, the heavier the weight of the graph, the closer the edge line is to another node. Invariably, the higher the weight of the graph, the shorter the line of the

graph. In the plagiarism graph, the weight corresponds to the similarity index between the two programs. If the percentage property is set to "enabled", the weight appears on the graph. The red circled illustrations are the weight of the graph as illustrated in Figure 5.10. The weights of this graph are 88 and 99.

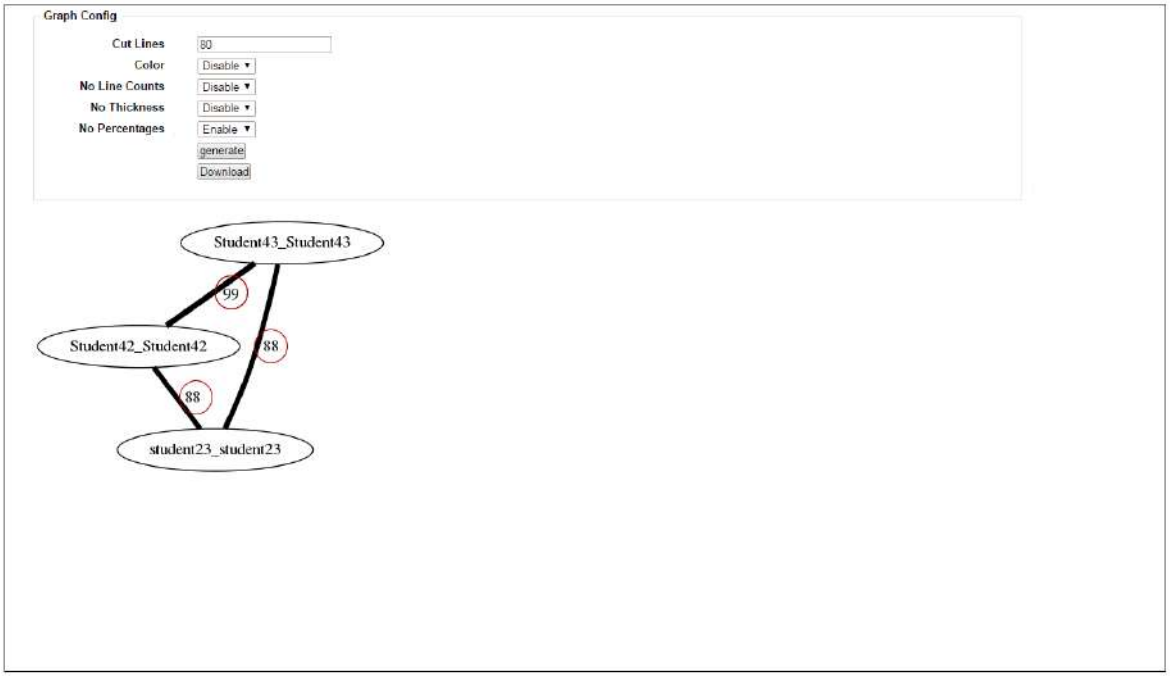


FIGURE 5.10: The weight of the plagiarism graph (adapted from the plugin)

Using the configuration option provided, we can potentially show portions of plagiarised contents of the plagiarism graph.

5.5 Inclusion of these tools into the VLE

The plugin was developed in PHP. Experiments were carried out on the undergraduate students' Java program lab work. An *instructor-supplied* code was added to eliminate matches that one expects to be copied, thereby eliminating coincidental similarities that may arise from the experiment. The inclusion of these tools into the VLE offered numerous features; the plugin showed side-by-side comparison of the plagiarism report, presented in Figure 5.11.

One key attribute offered by the system is that it is easy to show the overall structure of the class, rather than just the pairwise similarities presented in Figure 5.12. In addition, the structure of the class represented in form of a plagiarism graph would aid an instructor to differentiate actual cases of plagiarism from coincidental similarities in a

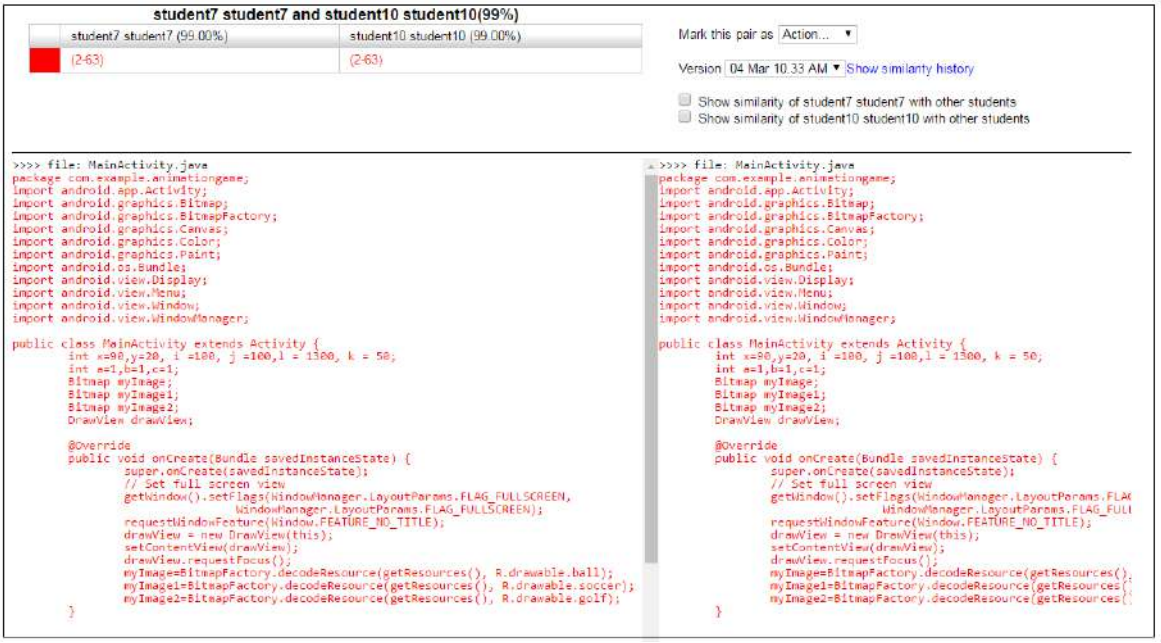


FIGURE 5.11: A pairwise comparison of the plagiarism report

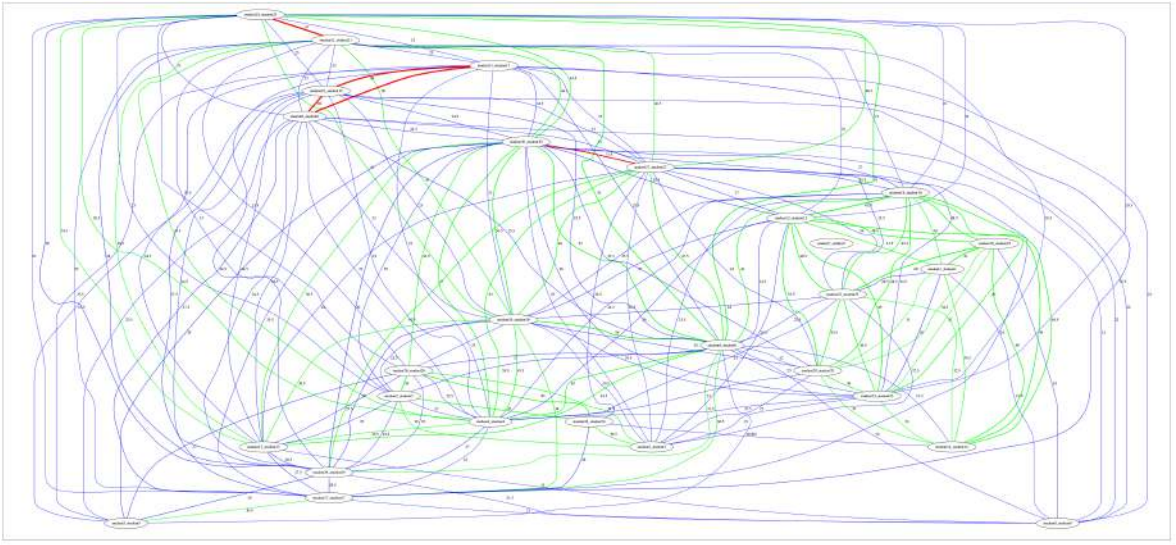


FIGURE 5.12: Structure of a plagiarism graph (adapted from the plugin)

programming class. The usefulness of the graph structure is its ability to show the relations amongst multiple students presented in Figure 5.13. Hence, connected students' programs were found to show that they engaged with each other to carry out the programming task. The connected student programs are presented according to the range of thresholds. Figure 5.13 shows the connected pairs at a threshold of 90. At different thresholds, the system shows groups of connected programs.

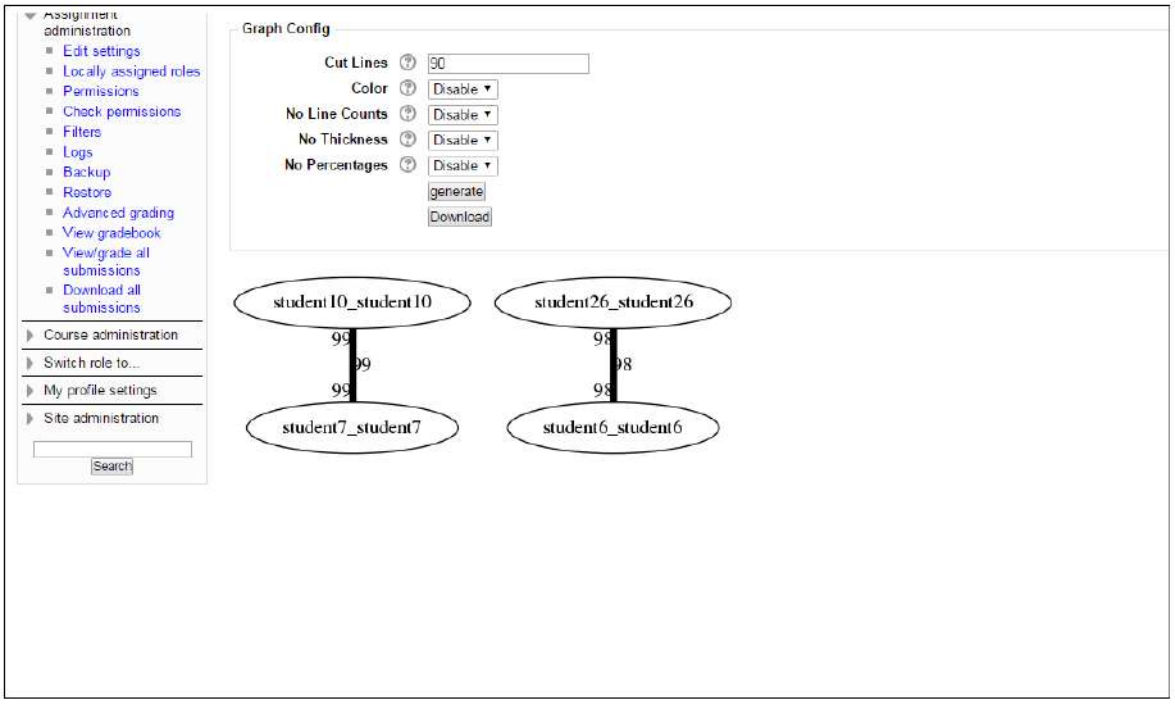


FIGURE 5.13: The connected pairs of the similar programs

5.6 Conclusion

This chapter described the method for this research. We presented the different approaches to creating an efficient plagiarism detection system. At first instance, the MOSS plagiarism detection system was integrated into the VLE. Using MOSS would also require an instructor to obtain a MOSS user ID and plagiarism instances will be triggered for the entire course. MOSS offered numerous features in presenting a plagiarism report. The features include; pairwise similarities of each source codes and a side-by-side comparison of a plagiarism report.

Secondly, the Graphviz tool was added into the MOSS plugin to visually show the pairs of the source codes. The Graphviz tool possesses features for customising the generated graph. The features include cutlines, colours, line count, thickness and percentage. The Graphviz features ensured easy identification of the plagiarism report. It was also found that the inclusion of these tools would make the detection process effortless to explore.

The next chapter focuses on the structures of graphs and the different operations carried out on plagiarism graphs.

Chapter 6

Graph Structures

6.1 Introduction

The previous chapter outlines the design and implementation of the plagiarism detection system. This chapter is focused on the graph data structures. Section 6.2 discusses structures of graphs. Section 6.3 explains connected components in plagiarism graphs. Section 6.4 discusses the intersection across multiple graphs. Section 6.5 concludes the chapter.

6.2 Graph Data Structures

Based on the ideas of graphs discussed in the concept of graphs and their application to plagiarism analysis, presented in Chapter 3, graph data structures are introduced, especially for program representation.

The application of source code plagiarism to graph data structure is the representation of the structural information of the programs. One key advantage of representing programs as graphs is that they allow keeping the structural information of the programs. The specifics of the representation of programs as graphs depend on the information needed to unravel from the structure of the graph. For example, the generated graph of the plagiarism detection system is represented as an *undirected graph*, which represents all *nodes* in the graph as the programs and connecting *edges* as the copies. This structural representation of the programs may be suited for determining if two plagiarists engage with each other to produce the same program. In this chapter, the graph metrics used in this work will be discussed holistically.

6.3 Connected Components

In a graph, a *node* can connect to any other *node* in a graph by means of a path *if and only if* the graph is a *connected* graph. Taking this into account, we can infer that a graph is *connected* if for each and every *node* in the graph, a path exists. Although, there are some instances of graphs which are not connected. For example, a social network graph shows individuals which may not be connected. For these, we can not construct a path between them to establish relationships.

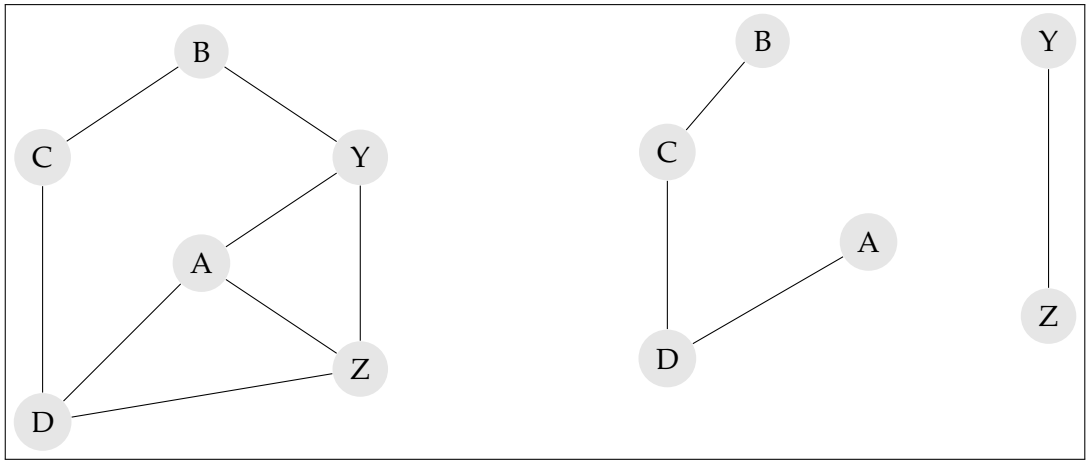


FIGURE 6.1: A graph that is connected, and a graph that consists of two connected components

Connected components in a graph are a subset of the *nodes* of a graph such that: (i) every *node* has a path to every other *node* in the graph; and (ii) the subset is not a member in the larger set with the property that every *node* can reach every other *node* in the set. Dividing a graph into connected components is a first step in describing its structure. The graph showed in Figure 6.1 shows a graph with extracted connected components in the graph. From the example in Figure 6.1, the six *nodes* in the graph represent the students' programs {A, B, C, D, Y, Z} with the pairs of programs indicated by {A,D}, {A,Y}, {A,Z}, {B,C}, {B,Y}, {C,D}, {D,Z}, {Y,Z}. We can extract the student programs{Y,Z} from the graph and the connected pairs from the extracted graph. From the generated graph of the plagiarism detection system, we can derive useful information from connected components. An image extracted from a larger image from the plagiarism detection system is presented in Figure 6.2. The image is scaled to show networks of students copying each other in a programming class. The connected components in the graph are indicated with the red circled region of students engaging with each other to produce the programs. The components showed *smaller* units of programs, *medium-sized* components and *larger* components.

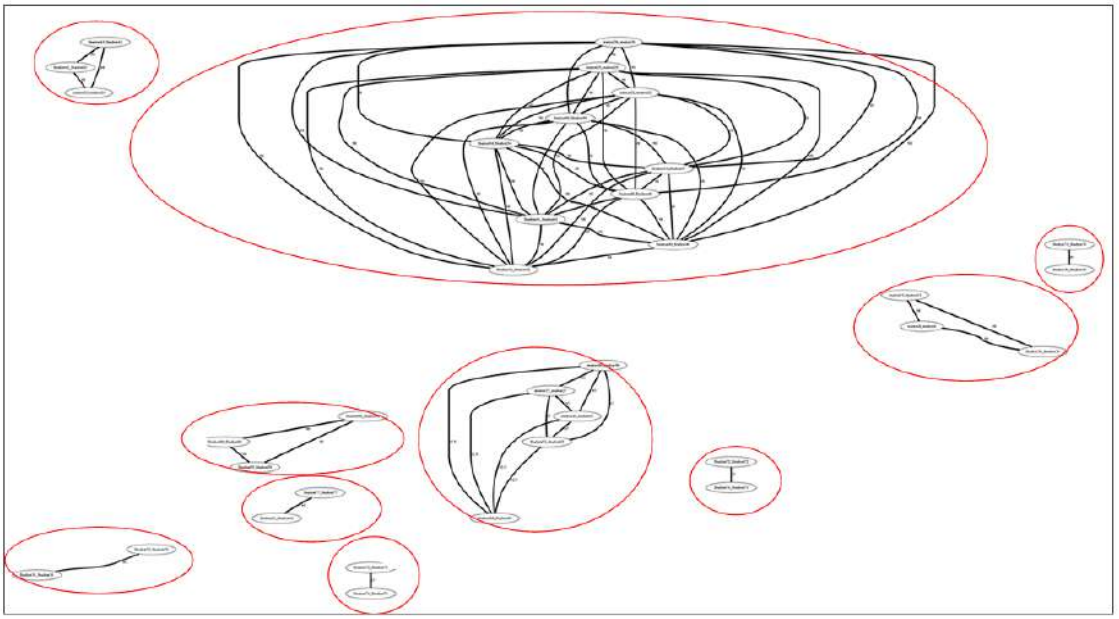


FIGURE 6.2: The connected components of the plagiarism graph

In *smaller* components of two program pairs, the pair may have copied each other and this would have educational benefit to the two students. Hence, the two students may have engaged with each other to produce the program only so, this is less of a problem for the instructor to investigate the two programs to find plagiarism. The instructor simply has to look at the program pairs and judge the programs as plagiarised pairs.

In *medium-sized* components, a student may have produced a program and distributed it to the rest of the class. It is worth noting that the code may be redistributed by another student, and this may have gotten into a few other hands. It is worth trying for the instructor to investigate the source of where the code was copied from.

In *larger* components, the code may have been copied from sources on the Internet. This is much easier to explore the origin of the copied code, and the culprits who were found to have copied from that source. It is also clearer that the instructor needs to take precautions when setting assignments, and should endeavour to set a question for which no answer can be found on the Internet.

6.4 Intersection

Graphs are useful tools to model problems of any area of specialisation, especially in the application to source code plagiarism. We introduce intersections across multiple graphs to unravel plagiarism across multiple students' programming assignments. A

graph is said to be an *intersection* graph if it contains a family of a non-empty set *F*, in which each set in *F* is represented by a *node*, and connected by two *nodes* with an *edge* if and only if their corresponding sets intersect [Golumbic et al., 1983]. In other words, an intersection number in a graph is the subset of *vertices* (cliques) of the graph that covers all the *edges* of the graph. Across two or more graphs, $A(V_A, E_A)$ and $B(V_B, E_B)$, the *intersection* represents the union of their *node* sets and *intersection* of their *edges* sets of the graphs. The *intersection* across the two graphs would mean $A \cap B = V_A \cup V_B, E_A \cap E_B$.

In source code plagiarism, an intersection graph is derived from two or more plagiarism graphs with the aim to uncover historical plagiarism across multiple programming assignments. A typical example of *intersection* across multiple graphs is shown in Figure 6.3.

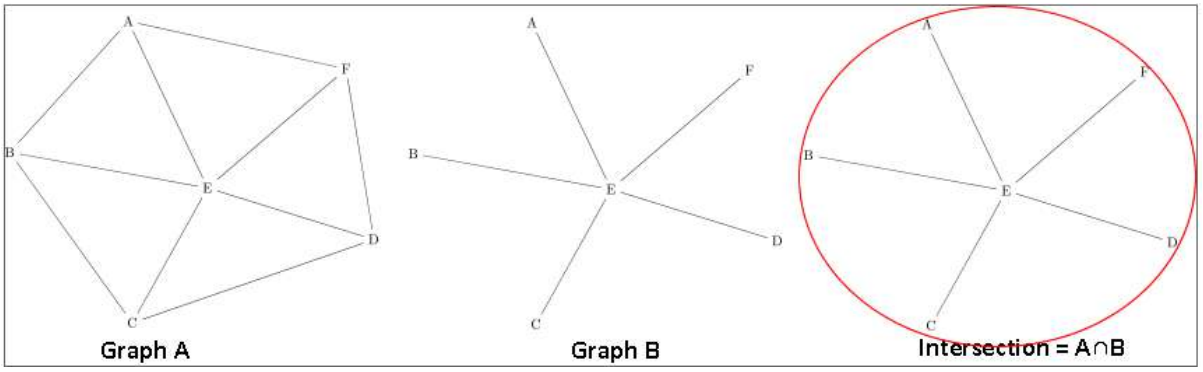


FIGURE 6.3: The intersection across multiple plagiarism graphs

Given two graphs A and B, Graph A with sets of *nodes* {A, B, C, D, E, and F} and sets of *edges* represented by {A,B}, {A,E}, {A,F}, {B,C}, {B,E}, {C,D}, {C,E}, {D,E}, {D,F}, {E,F} and Graph B with sets of *nodes* (A, B, C, D, E, and F) and sets of *edges* represented by {A,E}, {B,E}, {C,E}, {D,E}, {E,F}. Hence, the *intersection* across the two graphs is given by: $A \cap B$ = {A,E}, {B,E}, {C,E}, {D,E}, {E,F}. The *intersection* of the two graphs is represented by the red colour circled graph. The five *edges* represent the *intersection* sets of the two graphs.

Taking the *intersection* of the three graphs, it was found that there were unlikely false positives generated from the report. These results strongly suggest that the students were actually copying each other. Hence, repeated plagiarism is an indication that the students were actually copying each other in the assignments.

6.5 Conclusion

This chapter described the structures of graphs and their application to plagiarism detection. The basic definition of graphs was highlighted and a number of operations on graphs were discussed. The different graph structures were introduced. Connected components in graphs were explored in this chapter. In connected components, we derive a number of components that comprise a graph. The components are *small*, *medium-sized* and *large* components. The *intersection* across multiple graphs was also discussed in this chapter. The explanation of the structures would be useful in evaluating the graph model.

Chapter 7 is concerned with evaluating the performance of the plagiarism detection system.

Chapter 7

Evaluation

7.1 Introduction

In the previous chapter, the structure of the graph was discussed. This chapter is focused on the evaluation thereof. Section 7.2 introduces the data set used for our experiment. Section 7.3 is focused on the ground truth data set taken on the system. Section 7.4 is focused on evaluating the system's metrics. Section 7.5 discusses the performance metrics of the system. Section 7.6 discusses the results and discussions from the experiment. Section 7.7 is focused on performing the intersection across multiple assignments. Section 7.8 concludes the chapter.

7.2 Data Set

The training data set used for our initial experiment was sufficient to analyse system performance. This is significantly better than our prior work [Obaido et al., 2016]. The data set was collected with arbitrary usernames from "1892 to 1980". For the sake of confidentiality, the student identities were replaced with arbitrary names "Student 1", "Student 2",..., "Student 80". For the experiments, a number of 240 second-year undergraduate Java programs that had an average of 71 lines of code per programmer were taken from the three different lab work submissions. In each of the course's activities (MC_Lab_6_512, MC_Lab_5_509 and MC_Lab_3f_506), there were 80 programs in each of the lab work. The data set allowed us to test the idea of working with multiple graphs as they contained the same students performing multiple course activities. Table 7.1 shows the summary of the data set used in the experiment.

TABLE 7.1: Summary of the data set used in the evaluation of the Plagiarism Detection System.

Assignments	Number of Programs	Counts of lines of code	Counts of Tokens streams	Counts of Detections
MC_Lab_6_512	80	7447	167643	2864
MC_Lab_5_509	80	6662	135874	1642
MC_Lab_3f_506	80	3024	48615	1429
Total	240	17133	352132	5935

In the data set of 240 student submissions written in Java, each of the program's token strings and lines of code were counted. A number of **17,133** lines of code and **352,132** number of token streams were realised from the data set. In the data set, there were 3 programs which had similar token strings and lines of codes. To carry out the experiment, the data set was manually entered into the Virtual Learning Environment (VLE), and the programs were scanned for plagiarism. The plagiarism detection system reported the 3 programs which had similar lines of code and token streams as plagiarised sets of programs. The findings of the report are indexed in Appendix B.

7.3 Ground Truth Data

The key to choosing an effective source code plagiarism detection system is to obtain a ground truth. Ground truthing ensures the calibration of the plagiarism detection system report. To establish the ground truth for the collection of programs, each of the programs was examined manually to find patterns of plagiarism. In the collection of 240 programs in the three assignments, the plagiarism detection system reported **5935** distinct pairs of programs. The ground truth work was carried out in two phases:

TABLE 7.2: Summary of the Ground Truth Data Work

Assignments	Programs	Threshold Value	Counts of Program Pairs	Counts of Detection
MC_Lab_6_512	80	30% - 50%	655	225
MC_Lab_5_509	80	30% - 50%	166	68
MC_Lab_3f_506	80	30% - 50%	189	81
Total	**240**	-	1010	374

Phase 1: The plagiarism detection system was used to inspect the programs for plagiarism and the programs were manually classified as suspicious/coincidental similarities. At the initial phase of the ground truth work, pairs of programs between the threshold of 30% and 50% were manually investigated to find plagiarism. There were **1010** pairs of programs between the threshold of 30% and 50% in the collection. Out

of the **1010** pairs of programs in the collection, **374** program pairs were judged to have been plagiarised. Programs between the threshold of 50% and 100% were also investigated for plagiarism. Between the threshold of 50% and 100%, **3408** pairs were judged to have been plagiarised. At this phase, we were able to establish plagiarism by manually inspecting the pairs of programs.

Phase 2: At the second phase, a number of techniques used by students to hide plagiarism were found. The techniques mentioned in the work of Faidhi and Robinson [1987] and Seo-Young et al. [2006] outlined in the background of this research were found in the collection; reconstructed comments, variable declarations and output statements. The program constructs were interchanged, *if/else* swapped to *switch/case*, and a number of function definitions were reordered. These types of plagiarism were difficult to trace, especially if examining all pairs of source codes manually. There were a few cases of plagiarism which was easy to detect, and this showed similar usage of *comments*, identical *variables/constants* declarations, similar *function* definitions and identical program constructs. Table 7.2 shows the summary of the ground truth work.

7.4 Evaluation Metrics

In automatic plagiarism detection systems, a number of false positives/false negatives usually occur. Hence, an inaccurate report or missing plagiarism that does exist, usually produces misleading results. Several factors are responsible for this; shorter lines of codes, students working in teams and the absence of an instructor-supplied code given to students by an instructor [Granzer et al., 2013]. In source code plagiarism, a false positive (FP) results when a plagiarism detector reports pairs of programs as plagiarised when in actual fact, they were not plagiarised. A false negative (FN) results when a plagiarism detector reports pairs of programs as unplagiarised when in actual fact, they were plagiarised code pairs. Alternatively, a true positive (TP) shows the actual plagiarised pairs of programs, while a true negative (TN) indicates pairs of programs that were not plagiarised [Prechelt et al., 2002].

In evaluating the plagiarism detection system, each of the program pairs was manually classified as positives/negatives. Out of the **5935** pairs of programs reported by the plagiarism detection system, **3782** of the pairs of programs were judged to have truly engaged in plagiarism. A number of **2760** of the report were found to be falsely reported by the system. A number of false negatives/false positives errors were also encountered. For instance, some programs were found suspicious due to similar program identifiers and control structures used. These cases do not provide convincing

TABLE 7.3: Summary of metrics reported during evaluation of the system

Assignments	Threshold	False Positive	False Negative	True Positive	True Negative
MC_Lab_6_512	0% - 99%	1165	1066	1798	1698
MC_Lab_5_509	0% - 99%	443	564	1078	1199
MC_Lab_3f_506	0% - 99%	1152	523	906	277
	Total	2760	2153	3782	3174

proof that the programs were plagiarised. The students may have adopted the instructor's coding style to write their code, but the plagiarism detection system flagged the programs as plagiarised pairs. Table 7.3 shows the summary of metrics reported during the evaluation of the system. The next section is focused on the plagiarism detection system's performance.

7.5 System Performance Metrics

The reliability measure of our plagiarism detection system is based upon precision and recall as discussed in the literature of this work, Chapter 2. Taking into perspective that our plagiarism detection system detects source code programs in pairs, assigns a numerical similarity value to the pairs represented as highest similarity rate with a specified cut-off threshold, and displays the similar program pairs as graphs, Prechelt et al. [2002] stipulated that

- Precision constitutes the number of actual plagiarised pairs that have been detected in the entire sets of program pairs detected altogether.
- Recall constitutes the number of actual plagiarised pairs that have been detected in the entire sets of program pairs that have truly engaged in plagiarism.

From our dataset collection of 240 programs of the three assignments, to evaluate the system to find the precision and recall in our given dataset, we identified the pairs of programs as positives and negatives. Each pair from our system is classified as False positive (FP), True Positive (TP), True negative (TN), False negatives (FN), and the accuracy (ACC) of the system has been determined from the computation as follows:

$$\text{Recall}(R) = \frac{TP}{TP + FN} \tag{7.1}$$

$$\text{Precision}(P) = \frac{TP}{TP + FP} \tag{7.2}$$

Given the above metrics, we derive the accuracy of the system as follows:

$$\text{Accuracy}(ACC) = \frac{TP + TN}{TP + TN + FP + FN} \tag{7.3}$$

7.6 Results and Discussion

To evaluate the effectiveness of our system over the three assignments, as presented in Table 7.3 of the generated metrics, the precision and recall of the system evaluated on the three assignments are presented in Figure 7.1, 7.2 and 7.3 respectively.

Table 7.4, shows how the Precision/Recall curve was generated against the threshold of similarities in the plagiarism detection system.

TABLE 7.4: Summary of the evaluation metrics of the system

	Assignment 1		**Assignment 2**		**Assignment 3**	
Threshold	Precision	Recall	Precision	Recall	Precision	Recall
90 - 99%	1	0.0432106	1	0	1	0.134045403
80 - 89%	0.888889	0.2212834	0.869565	0.123244	0.857143	0.222345433
70 - 79%	0.906627	0.3892348	0.808314	0.234432	0.736842	0.440537344
60 - 69%	0.625571	0.4392453	0.690789	0.394843	0.642857	0.639485543
50 - 59%	0.540872	0.4839213	0.635701	0.520588	0.635417	0.537243943
40 - 49 %	0.551769	0.539403	0.626506	0.658228	0.554622	0.684845747
30 - 39%	0.491272	0.723483	0.628205	0.687754	0.50365	0.634849324
20 - 29%	0	0.723483	0	0.648148	0.024324	0.734212334
10 - 19%	0	0.824745	0	0.777738	1	0.89327332
0 - 9%	0	1	0	0	1	1
	Accuracy = 84%		Accuracy = 79%		Accuracy = 52%	

During the first assignment in Figure 7.1, we presented an interpolated Precision-Recall graph over the first assignment. We discovered that there was a high precision rate and a few false positives were found in the assignment. The system accuracy at the first assignment showed an 84%. Hence, we can deduce that the system reported a number of false positives in the first assignment. We believe there is room for improvement.

The system's evaluation report of the second assignment, Figure 7.2, was better compared to the first case. The system reported an accuracy of 79%. Although a number of false positives were found, the result fared better than in the first case.

Finally, the third assignment in Figure 7.3 reported a huge number of false positives with an accuracy of 52%. We found that the absence of an *instructor-supplied* code and

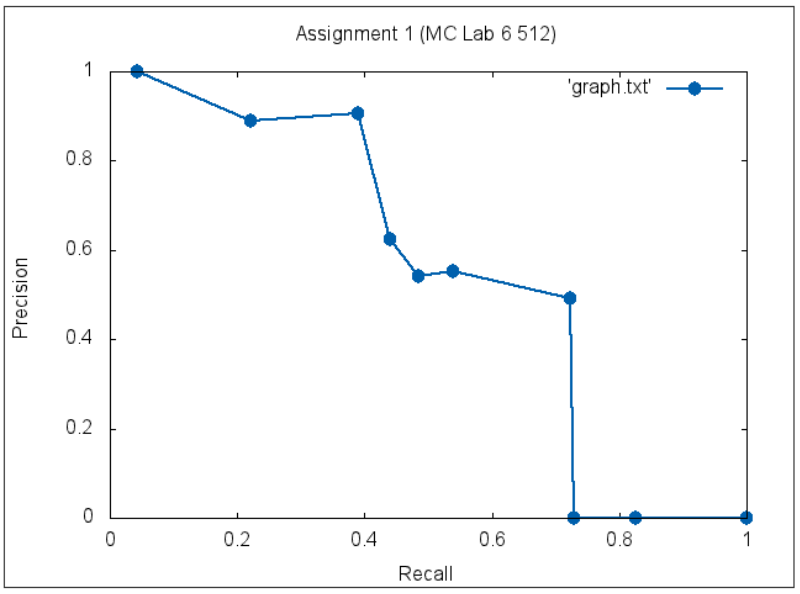


FIGURE 7.1: Assignment 1: Precision-Recall curve of the first assignment

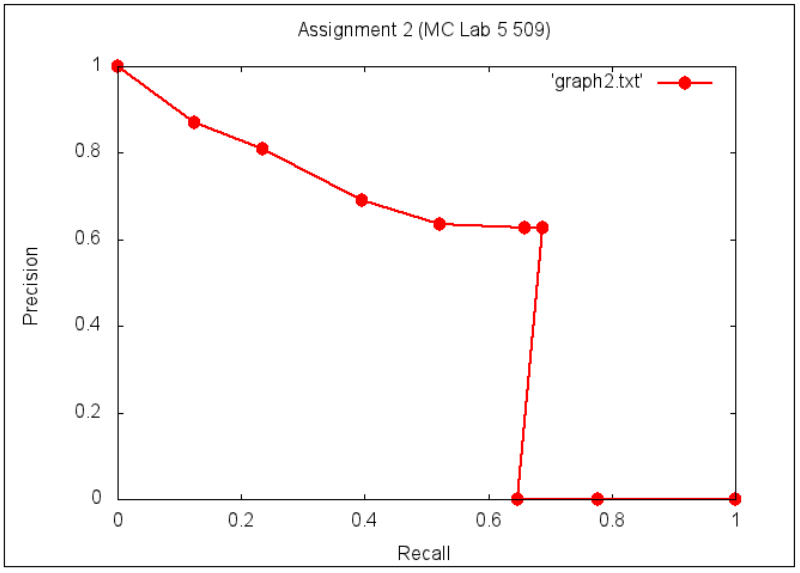


FIGURE 7.2: Assignment 2: Precision-Recall curve of the second assignment

the length of the programs, contributed to the low accuracy of the report. Hence, the accuracy of the third assignment performed poorly compared to the first and second assignments respectively.

7.7 Intersection Across Multiple Assignments

During the final experiments, a Python script was used to prune the graph and find similar occurrences in the data set. Seven occurrences of historical plagiarism were identified from the experiment indicated by the number of *edges* across the three graphs.

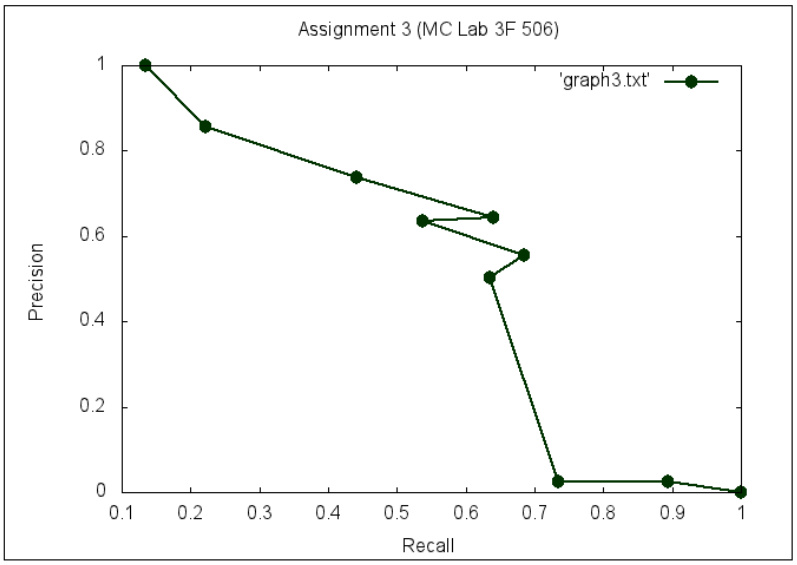


FIGURE 7.3: Assignment 3: Precision-Recall curve of the third assignment

The image on Figure 7.4 shows the historical plagiarists who engaged in plagiarism over the three assignments. The images were regenerated into four different images to view the structure of the class. The images were regrouped into a number of connected components. Figure 7.4 shows the connected components of the intersection plagiarism graph of the three assignments.

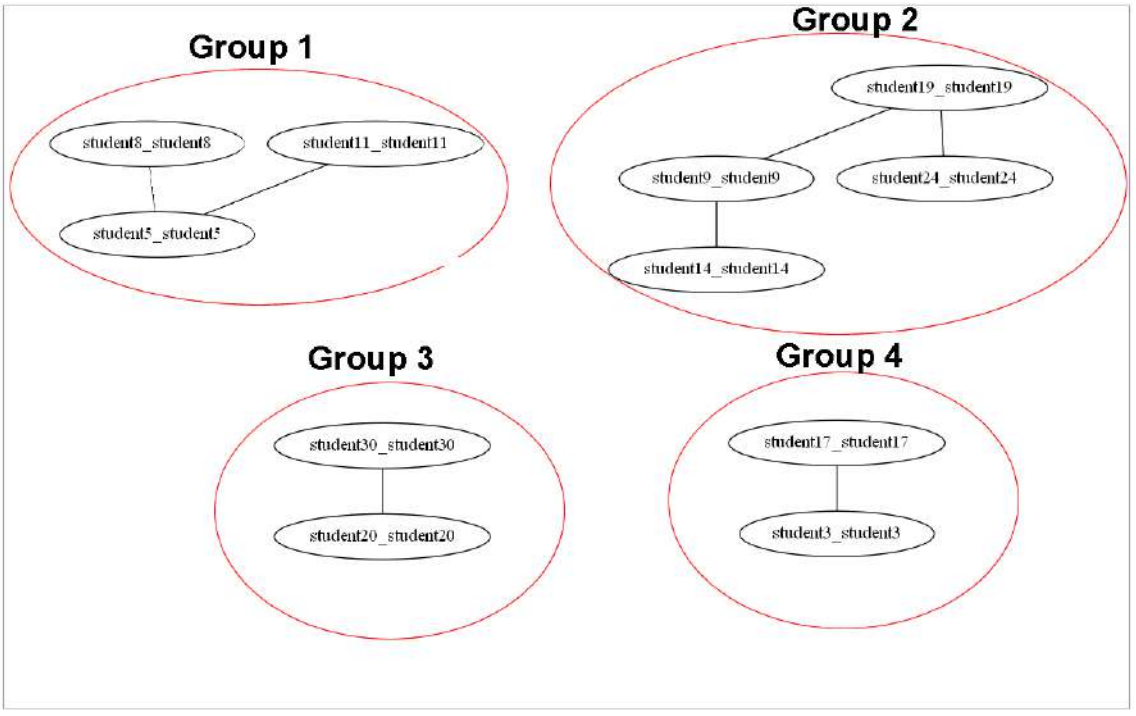


FIGURE 7.4: The connected components of the intersection graph.

In **Group 1**, there are three plagiarised programs. The "Student 8" and "Student 11" programs are connected to the "Student 5" program, but are not connected to each other. It can be ascertained that both students copied from "Student 5". **Group 2** shows four plagiarised programs. The programs are weakly connected to each other. In the **Group 2** graph, the "Student 9" and "Student 24" programs are adjacent over the "Student 19" program. The students may have collaborated to produce the plagiarised program. **Group 3** and **Group 4** are strongly connected graphs. In other words, the students may have collaborated to produce the set of plagiarised programs. In Figure 7.4, **Group 1** and **Group 2** are weakly connected. Each source code is not connected to every other source codes in the graph. So it is possible that a student can copy from one or two other students' assignment, but not to every other member of the group. Group 3 and Group 4 are strongly connected to each other. Invariably, two or more students can copy from each other to write their source codes. It was discovered that the pairs of programs (connected components) appeared in the three assignments, hence, we can conclude that if a set of students are adjacent over a number of different submissions, it shows that they are really copying each other.

Manual verification of the edges in the intersection graph indicated that the programs were truly plagiarised pairs. Hence, this reduces the rate of false positives in the result to zero.

7.8 Conclusion

This chapter presents an evaluation of the investigation component of the plagiarism detection tool. The basic idea of the graph tool was to annotate the source code pairs as graphs, rather than pairwise similarities of source codes. Experiments were performed on the source codes and a number of similarities were found. We found out that a number of errors; false positive, false negative, true negative and true positive usually occur in detecting plagiarism in source codes. A comparison of the result gathered during the analysis of the system with human judgement, also revealed a high correlation. The plagiarism detection system has shown to report a higher number of plagiarism cases than what a human judgement would have been able to achieve.

To conclude the experiment, we introduced the idea of taking the intersection across a number of assignments. This reduces the number of false positives to zero. We can conclusively establish those students who are really plagiarising in their assignments. From the evaluation report, we also found that source code plagiarism is a serious issue and supporting evidence that consists of more than just a pairwise comparison of

source codes can be helpful to academics with regards to making sound judgement on cases of plagiarism.

The next chapter draws conclusion, provides the limitation of the research, discusses the contribution and provides future work pertaining to this research.

Chapter 8

Conclusion, Limitations, Contribution and Future Work

8.1 Conclusion

The ubiquitous cases of source code plagiarism in programming assignments are increasing rapidly [Ohno and Murao, 2011]. The problem has been on the rise and students are devising new techniques in deceiving their instructors. Hence, the reduction of source code plagiarism is the primary goal of source code plagiarism detection systems. A number of suspicious cases of source code plagiarism can be easily missed by humans, especially if the class size is large and the means of checking plagiarism in programming assignments is the side-by-side comparison of each individual source code against another. In this research, we applied graph-based approaches in detecting suspicious cases of source code plagiarism in programming assignments.

The first approach undertaken in this research in detecting plagiarism in programming assignments, was using the MOSS system to extract and compare the representation of the program structure and measure similarities in the programs. Whilst this approach seems like a good option, there were traces of *false positives*. These are however also evident in most source plagiarism detection systems. An informal approach was to manually look up each source code programs to inspect *false positive/false negative* in the similarity report.

The second approach undertaken was the application of the graph-based approach to convert programs to *nodes*, the relationship between the programs as a set of *edges*,

and the degree of copies as the *weight* specified in the graph model. Other graphs metrics (intersections of graphs from multiple assignments) were applied to convincingly prove that students are really copying from each other.

It was found that, at a lower threshold of 30%, programs have to be investigated for plagiarism. It was also easily shown that the ground truth work proved to show a knowledge of plagiarism and was effective to prune the plagiarism graph. We also investigated that taking the intersection of multiple plagiarism graphs, reduces *false positives/false negatives*. It was easily proved that if a student plagiarises in the first, second and third assignments, there is a likelihood for them to even plagiarise more in many circumstances.

Although, the source code plagiarism detection process was easily carried out, the result should not be used as a final benchmark to indicate whether or not a student has plagiarised. The result should be used as a suggestion to aid instructors carrying out manual investigation meticulously. By applying this tool for detecting similarities in source code, assessment of source codes for plagiarism will become more efficient and worthwhile. Using this tool for large programs also works better than for smaller programs. The result might not be accurate for shorter programs due to the presence of more coincident similarities in shorter programs, which may lead to a greater number of *false positives*. By using this automatic source code plagiarism detection, we hope that students will be discouraged from plagiarism, as they will be aware we are watching them conscientiously. Hence, it will encourage them to work independently and exert much effort in writing their program assignments themselves, which will increase their understanding of programming and contribute effectively to their academic achievements.

8.2 Limitations of the Study

One major limitation of this research was that we could not test all programming languages supported by MOSS. The only programming language tested in this research was Java. This limited the scope of our research somewhat; however, in this case, we only considered the programming language taught at the university. Few of the data sets provided also had shorter lines of code. We discovered that shorter programs contributed to a high false positive rate, which produces unreliable results. To avoid this issue, we had to carefully select the three lab works to find programs with reasonable program *lengths* to perform the experiment.

8.3 Contributions and Future Work

This work has contributed a new extension to plagiarism detection that allows the representation of programs as graphs. The inclusion of the graph-based approach to the structure representation of programs shows a major contribution. This part of the research has been published in Obaido et al. [2016].

Other contributions of this work are the *detection* of the programs, *long-term storage* of the similarity report, *enhanced functionality* of the plagiarism detection engine compared to other plagiarism detection tools, *real-time analysis* by both tools, other plagiarism detection engines takes considerable number of hours to detect programs, an example is the *program dependency graph tool* and effective *decision-making* for instructors, with the features offered by the graph-based approach.

There are still unexplored areas relating to graph-based techniques in detecting source code plagiarism. Given the results obtained from the experiments carried out using the plagiarism detection tool, it will be interesting to explore more graph operations on the graph model. One unexplored area that would be interesting in this research is modeling the graph tool to find the *shortest possible path*.

Applying this technique would be interesting to deduce if the shortest possible path is best suited to indicate plagiarism. Another area of continued research would be to find plagiarism across *multiple courses* to investigate historical plagiarism. It would be worthwhile to check the history of a known plagiarist in a class amongst multiple courses. Achieving this would assist in minimising the spread of plagiarism across multiple programming courses.

Bibliography

[Ahmadzadeh et al., 2011] Marzieh Ahmadzadeh, Elham Mahmoudabadi, and Farzad Khodadadi. Pattern of plagiarism in novice students' generated programs: An experimental approach. *Journal of Information Technology Education*, 10:1–11, 2011.

[Aiken, 1994] Alex Aiken. A system for detecting software plagiarism. *Retrieved May*, 1:2015, 1994. URL `https://theory.stanford.edu/~aiken/moss/`.

[Al Jarrah et al., 2011] Abeer Al Jarrah, Izzat Alsmadi, and Zakariya Za'atreh. Plagiarism detection based on studying correlation between Author, Title, and Content. In *Proceedings of the International Conference on Information Communication System*, pages 22–24, 2011.

[Arwin and Tahaghoghi, 2006] Christian Arwin and Seyed M. M. Tahaghoghi. Plagiarism detection across programming languages. In *Proceedings of the 29th Australasian Computer Science Conference*, volume 48, pages 277–286. Australian Computer Society, Inc., 2006.

[Atkins and Nelson, 2001] Thomas Atkins and Gene Nelson. Plagiarism and the Internet: Turning the tables. *English Journal*, pages 101–104, 2001.

[Awad and Elseuofi, 2011] W. A. Awad and S. M. Elseuofi. Machine learning methods for E-mail classification. *International Journal of Computer Applications*, 16(1), 2011.

[Bandara and Wijayarathna, 2011] Upul Bandara and Gamini Wijayarathna. A machine learning based tool for source code plagiarism detection. *International Journal of Machine Learning and Computing*, 1(4):337–343, 2011.

[Barnhart, 1988] Robert K. Barnhart. Dictionary of Etymology. *Edinburgh, Chambers Harrap*, 1988.

[Baxter et al., 1998] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE, 1998.

[Bennett, 2005] Roger Bennett. Factors associated with student plagiarism in a Post-1992 University. *Assessment & Evaluation in Higher Education*, 30(2):137–162, 2005.

[Bilenko and Mooney, 2003] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM Special Interest Group on Knowledge Discovery and Data Mining*, pages 39–48. ACM, 2003.

[Biswas and Paul, 2010] Goutam Biswas and Dibyendu Paul. An evaluative study on the open source digital Library softwares for institutional repository: Special reference to Dspace and Greenstone Digital Library. *International Journal of Library and Information Science*, 2(1):1–10, 2010.

[Bondy and Murty, 1976] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Citeseer, 1976.

[Bowyer and Hall, 1999] Kevin W Bowyer and Lawrence O Hall. Experience using "MOSS" to detect cheating on programming assignments. In *Proceedings of the Frontiers in Education Conference*, volume 3, pages 13–18. IEEE, 1999.

[Boyd et al., 2013] Kendrick Boyd, Kevin H. Eng, and C. David Page. Area under the Precision-Recall curve: Point estimates and confidence intervals. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 451–466. Springer, 2013.

[Britannica, 2005] Encyclopedia Britannica. plagiarism, 2005. URL `http://global.britannica.com/topic/plagiarism`. Accessed: 2015-8-7.

[Burrows et al., 2007] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. In *Journal of Software: Practice and Experience*, volume 37, pages 151–175, 2007.

[Cesare and Xiang, 2010] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing*, volume 107, pages 61–70. Australian Computer Society, Inc., 2010.

[Chae et al., 2013] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. Software plagiarism detection: A graph-based approach. In *Proceedings of the 22nd International Conference on Information & Knowledge Management*, pages 1577–1580. ACM, 2013.

[Chan and Collberg, 2014] Patrick P.F. Chan and Christian Collberg. A method to evaluate CFG comparison algorithms. In *Proceedings of the 14th International Conference on Quality Software*, pages 95–104. IEEE, 2014.

[Chen et al., 2004] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.

[Choi et al., 2007] Seokwoo Choi, Heewan Park, Hyun-il Lim, and Taisook Han. A static birthmark of binary executables based on API call structure. In *Annual Conference on Asian Computing Science*, pages 2–16. Springer, 2007.

[Chunhui et al., 2013] Wang Chunhui, Liu Zhiguo, and Liu Dongsheng. Preventing and detecting plagiarism in programming course. *International Journal of Security and Its Applications*, 7(5):269–278, 2013.

[Church and Helfman, 1993] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993.

[Conte et al., 2003] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Graph matching applications in pattern recognition and image processing. In *Proceedings of the International Conference on Image Processing*, volume 2, pages II–21. IEEE, 2003.

[Cytron et al., 1991] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13 (4):451–490, 1991.

[Davis and Goadrich, 2006] Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240. ACM, 2006.

[DeVoss and Rosati, 2002] Dànielle DeVoss and Annette C Rosati. "It wasn't me, was it?" Plagiarism and the Web. *Computers and composition*, 19(2):191–203, 2002.

[Dick et al., 2002] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, and Cary Laxer. Addressing student cheating: definitions and solutions. In *ACM Special Interest Group on Computer Science Education Bulletin*, volume 35, pages 172–184. ACM, 2002.

[Djuric and Gasevic, 2012] Zoran Djuric and Dragan Gasevic. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2012.

[Ducasse et al., 1999] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

[Faidhi and Robinson, 1987] J. A. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a University programming environment. *Journal of Computers & Education*, 11(1):11–19, 1987.

[Ferrante et al., 1987] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimisation. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[Ganguly and Jones, 2014] Debasis Ganguly and Gareth J.F. Jones. An information retrieval approach for source code plagiarism detection. In *Proceedings of the Forum for Information Retrieval Evaluation*, pages 39–42. ACM, 2014.

[Gascon et al., 2013] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of the ACM workshop on Artificial Intelligence and Security*, pages 45–54. ACM, 2013.

[Goadrich et al., 2006] Mark Goadrich, Louis Oliphant, and Jude Shavlik. Gleaner: Creating ensembles of first-order clauses to improve Precision-Recall curves. *Machine Learning*, 64(1-3):231–261, 2006.

[Goffe and Sosin, 2005] William L. Goffe and Kim Sosin. Teaching with Technology: May you live in interesting times. *Journal of Economic Education*, 36(3):278–291, 2005.

[Golumbic et al., 1983] Martin Charles Golumbic, Doron Rotem, and Jorge Urrutia. Comparability graphs and intersection graphs. *Discrete Mathematics*, 43(1):37–46, 1983.

[Grace et al., 2012] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, volume 14, page 19, 2012.

[Granzer et al., 2013] Wolfgang Granzer, Friedrich Praus, and Peter Balog. Source code plagiarism in computer engineering courses. In *Proceedings of the 3rd International Conference on Education, Training and Informatic*, 2013.

[Grier, 1981] Sam Grier. A tool that detects plagiarism in Pascal programs. In *ACM Special Interest Group on Computer Science Education Bulletin*, volume 13, pages 15–20. ACM, 1981.

[Gross and Yellen, 2005] Jonathan L Gross and Jay Yellen. *Graph theory and its applications*. CRC Press, 2005.

[Grozea et al., 2009] Cristian Grozea, Christian Gehl, and Marius Popescu. ENCOPLOT: Pairwise sequence matching in linear time applied to plagiarism detection. In *3rd PAN Workshop on Uncovering Plagiarism, Authorship and Social Software Misuse*, page 10, 2009.

[Hage et al., 2010] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28, 2010.

[Halstead, 1977] Maurice H. Halstead. *Elements of software science (operating and programming systems series)*. Elsevier Science Inc., New York, USA, 1977.

[Hamel, 2009] Lutz Hamel. Model assessment with ROC curves. In *Encyclopedia of Data Warehousing and Mining, Second Edition*, pages 1316–1323. IGI Global, 2009.

[Hattingh et al., 2013] Frederik Hattingh, Albertus Buitendag, and Jacobus Van Der Walt. Presenting an alternative source code plagiarism detection framework for improving the teaching and learning of programming. In *Proceedings of the Informing Science and Information Technology Education Conference*, volume 12, 2013.

[Horwitz and Reps, 1992] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411. ACM, 1992.

[Howard, 2007] Rebecca Moore Howard. Understanding "Internet plagiarism". *Computers and Composition*, 24(1):3–15, 2007.

[Huston, 2008] Cate Huston. Fingerprinting Jar files using Winnowing. pages 1–5, 2008. URL `https://cate.blog/files/fingerprintingJarfiles`.

[Introna et al., 2003] Lucas Introna, Niall Hayes, Lynne Blair, and Elspeth Wood. Cultural attitudes towards plagiarism. *Report of the Lancaster University*, 2003.

[Jiang and Tan, 2005] Xing Jiang and Ah-Hwee Tan. Mining ontological knowledge from domain-specific text documents. In *Proceedings of the 5th International Conference on Data Mining*, pages 1–4. IEEE, 2005.

[Jones, 2001] Edward L. Jones. Metrics based plagarism monitoring. *Journal of Computing Sciences in Colleges*, 16(4):253–261, 2001.

[Joy and Luck, 1999] Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.

[Kammer et al., 2011] ML Kammer, HL Bodlaender, and J Hage. *Plagiarism detection in Haskell programs using call graph matching*. PhD thesis, Utrecht University, 2011.

[Komondoor and Horwitz, 2001] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the International Static Analysis Symposium*, pages 40–56. Springer, 2001.

[Kontogiannis et al., 1996] Kostas A. Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. In *Reverse Engineering*, pages 77–108. Springer, 1996.

[Krinke, 2001] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[Lancaster and Culwin, 2005] Thomas Lancaster and Fintan Culwin. Classifications of plagiarism detection engines. *Innovation in Teaching and Learning in Information and Computer Sciences*, 4(2):1–16, 2005.

[Le Nguyen et al., 2013] Thanh Tri Le Nguyen, Angela Carbone, Judy Sheard, and Margot Schuhmacher. Integrating source code plagiarism into a virtual learning environment: Benefits for students and staff. In *Proceedings of the 15th Australian Computing Education Conference*, volume 136, pages 155–164, 2013.

[Li et al., 2016] Wenchao Li, Hassen Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. Detecting similar programs via the Weisfeiler-Leman graph kernel. In *Proceedings of the International Conference on Software Reuse*, pages 315–330. Springer, 2016.

[Liu et al., 2006] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM Special Interest Group on Knowledge Discovery and Data Mining International Conference*, pages 872–881. ACM, 2006.

[Liu and Shriberg, 2007] Yang Liu and Elizabeth Shriberg. Comparing evaluation metrics for sentence boundary detection. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 1–6. IEEE, 2007.

[Lutz and Diehl, 2014] Rainer Lutz and Stephan Diehl. Using visual dataflow programming for interactive model comparison. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 653–664. ACM, 2014.

[Mallon, 1988] Thomas Mallon. Stolen words foray into the origins and ravages of plagiarism. *Language Arts & Disciplines*, 1988.

[Manel et al., 2001] Stéphanie Manel, H Ceri Williams, and Stephen James Ormerod. Evaluating presence–absence models in Ecology: the need to account for prevalence. *Journal of Applied Ecology*, 38(5):921–931, 2001.

[Mann and Frew, 2006] Samuel Mann and Zelda Frew. Similarity and originality in code: Plagiarism and normal variation in student assignments. In *Proceedings of the 8th*

Australasian Conference on Computing Education, volume 52, pages 143–150. Australian Computer Society, Inc., 2006.

[Marinescu et al., 2013] D Marinescu, A Baicoianu, and S Dimitriu. A plagiarism detection system in computer source code. *International Journal of Computer Science Research and Application*, 3(1):22–30, 2013.

[Marshall and Garry, 2005] Stephen Marshall and Maryanne Garry. How well do students really understand plagiarism. In *Proceedings of the 22nd Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education*, pages 457–467, 2005.

[Mayrand et al., 1996] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253. IEEE, 1996.

[McCabe, 2005] D McCabe. Levels of cheating and plagiarism remain high. *Center for Academic Integrity, Duke University*, 2005.

[Moussiades and Vakali, 2005] Lefteris Moussiades and Athena Vakali. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.

[Mozgovoy, 2006] Maxim Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. *International Journal on Informatics in Education*, 5(1):97–112, 2006.

[Noynaert, 2006] J. Evan Noynaert. Plagiarism detection software. In *Midwest Instruction and Computing Symposium*, 2006.

[Obaido et al., 2016] G. Obaido, P. Ranchod, and R. Klein. Catching plagiarists: Detecting plagiarism in student source code assignments in a virtual learning environment. In *the 10th International Technology, Education and Development Conference*, pages 7369–7376. IATED, 2016. ISBN 978-84-608-5617-7. doi: 10.21125/inted.2016.0074. URL http://dx.doi.org/10.21125/inted.2016.0074.

[Ohmann, 2013] Anthony Ohmann. Efficient clustering-based plagiarism detection using IPPDC. pages 4–43, 2013.

[Ohno and Murao, 2011] Asako Ohno and Hajime Murao. A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM. *International Journal of Innovative Computing, Information and Control*, 7(8):4729–4739, 2011.

[Olt, 2009] Melissa R Olt. Seven strategies for plagiarism-proofing discussion threads in online courses. *Journal of Online Learning and Teaching*, 5(2):222–229, 2009.

[Ottenstein, 1976] Karl J Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM Special Interest Group on Computer Science Education Bulletin*, 8(4):30–41, 1976.

[Owunwanne et al., 2010] Daniel Owunwanne, Narendra Rustagi, and Remi Dada. Students' perceptions of cheating and plagiarism in higher institutions. *Journal of College Teaching & Learning*, 7(11), 2010.

[Parker and Hamblen, 1989] A. Parker and J. O. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, 1989.

[Pawelczak, 2013] Dieter Pawelczak. Online detection of source-code plagiarism in undergraduate programming courses. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering*, page 1, 2013.

[Poon et al., 2012] Jonathan YH Poon, Kazunari Sugiyama, Yee Fan Tan, and Min-Yen Kan. Instructor-centric source code plagiarism detection and plagiarism corpus. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, pages 122–127. ACM, 2012.

[Powers, 2011] David Martin Powers. Evaluation: from Precision, Recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37—-63, 2011.

[Prechelt et al., 2002] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8 (11):10–16, 2002.

[Provost et al., 1998] Foster J. Provost, Tom Fawcett, and Ron Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proceedings of the International Conference on Machine Learning*, volume 98, pages 445–453, 1998.

[Purwitasari et al., 2011] Diana Purwitasari, I Wayan Surya Priantara, Putu Yuwono Kusmawan, Umi Laili Yuhana, and Daniel Oranova Siahaan. The use of Hartigan index for initialising K-means++ in detecting similar texts of clustered documents as a plagiarism indicator. *Asian Journal of Information Technology*, 10(8):1–8, 2011.

[Ristad and Yianilos, 1998] Eric Sven Ristad and Peter N Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[Roy and Cordy, 2007] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[Schleimer et al., 2003] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM Special Interest Group on Management Of Data*, pages 76–85. ACM, 2003.

[Selwyn, 2008] Neil Selwyn. 'Not necessarily a bad thing...': A study of online plagiarism amongst undergraduate students. *Assessment & Evaluation in Higher Education*, 33(5): 465–479, 2008.

[Seo-Young et al., 2006] N. Seo-Young, Sangwoo K., and J. Cheonyoung. A lightweight program similarity detection model using XML and Levenshtein distance. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering*, pages 3–9, 2006.

[Sheard et al., 2003] Judy Sheard, Angela Carbone, and Martin Dick. Determination of factors which impact on IT students' propensity to cheat. In *Proceedings of the 5th Australasian conference on Computing education*, volume 20, pages 119–126. Australian Computer Society, Inc., 2003.

[Siregar, 2015] Kesumo Siregar. Levenshtein distance calculation using dynamic programming for source code plagiarism checking. *STE dan Informatika*, 12:1–8, 2015.

[Smith and Horwitz, 2009] Randy Smith and Susan Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International Workshop on Software Clones*, pages 1–7, 2009.

[Soliman and Girdzijauskas, 2016] Amira Soliman and Sarunas Girdzijauskas. Adaptive graph-based algorithms for Spam detection in social networks. In *Proceedings of the International Conference on Networked Systems*, pages 1–18, 2016.

[Song et al., 2015] Hyun-Je Song, Seong-Bae Park, and Se Young Park. Computation of program source code similarity by composition of parse tree and call graph. *Mathematical Problems in Engineering*, 2015, 2015.

[Sorokina et al., 2006] Daria Sorokina, Johannes Gehrke, Simeon Warner, and Paul Ginsparg. Plagiarism detection in arXiv. In *Proceedings of the International Conference on Data Mining*, pages 1070–1075. IEEE, 2006.

[Sraka and Kaučič, 2009] Dejan Sraka and Branko Kaučič. Source code plagiarism. In *Proceedings of the 31st International Conference on Information Technology Interfaces*, pages 461–466. IEEE, 2009.

[Su et al., 2008] Zhan Su, Byung-Ryul Ahn, Ki-Yol Eom, Min-Koo Kang, Jin-Pyung Kim, and Moon-Kyun Kim. Plagiarism detection using the Levenshtein distance and Smith-Waterman algorithm. In *Proceedings of the 3rd International Conference on Innovative Computing Information and Control*, pages 569–569. IEEE, 2008.

[Suarez-Tangil et al., 2014] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4): 1104–1117, 2014.

[Tresnawati et al., 2012] Dewi Tresnawati, Arief Syaichu, and Kuspriyanto. Plagiarism detection system design for programming assignment in virtual classroom based on Moodle. *Procedia-Social and Behavioral Sciences*, 67:114–122, 2012.

[Verco and Wise, 1996] Kristina L. Verco and Michael J. Wise. Software for detecting suspected plagiarism: comparing structure and attribute-counting systems. In *Proceedings of the 1st Australian Conference on Computer Science Education*, pages 86–95. ACM, 1996.

[Vogts, 2009] Dieter Vogts. Plagiarising of source code by novice programmers a cry for help? In *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 141–149. ACM, 2009.

[Wagner, 2004] Neal R Wagner. Plagiarism by student programmers. *The University of Texas at San Antonio Division Computer Science San Antonio, TX*, 78249, 2004.

[Wolpers et al., 2010] Martin Wolpers, Paul A Kirschner, Maren Scheffel, Stefanie Lindstaedt, and Vania Dimitrova. *Sustaining TEL: From Innovation to Learning and Practice*, volume 6383. Springer, 2010.

[Young, 2001] Jeffrey R Young. Plagiarism and plagiarism detection go High Tech. *The Chronicle of Higher Education*, 6, 2001.

[Zdziarski et al., 2006] Jonathan Zdziarski, Weilai Yang, and Paul Judge. Approaches to Phishing identification using match and probabilistic digital fingerprinting techniques. In *Proceedings of the MIT Spam Conference*, pages 1115–1122, 2006.

[Zhang et al., 2012] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 111–121. ACM, 2012.

Appendix A

Evaluation

The table shows the evaluation information of the system.

TABLE A.1: Summary of the evaluation metrics of the system

	Assignment 1		**Assignment 2**		**Assignment 3**	
Threshold	Precision	Recall	Precision	Recall	Precision	Recall
90 - 99%	1	0.0432106	1	0	1	0.134045403
80 - 89%	0.888889	0.2212834	0.869565	0.123244	0.857143	0.222345433
70 - 79%	0.906627	0.3892348	0.808314	0.234432	0.736842	0.440537344
60 - 69%	0.625571	0.4392453	0.690789	0.394843	0.642857	0.639485543
50 - 59%	0.540872	0.4839213	0.635701	0.520588	0.635417	0.537243943
40 - 49 %	0.551769	0.539403	0.626506	0.658228	0.554622	0.684845747
30 - 39%	0.491272	0.723483	0.628205	0.687754	0.50365	0.634849324
20 - 29%	0	0.723483	0	0.648148	0.024324	0.734212334
10 - 19%	0	0.824745	0	0.777738	1	0.89327332
0 - 9%	0	1	0	0	1	1
	Accuracy = 84%		Accuracy = 79%		Accuracy = 52%	

Appendix B

Ground Truth Work

The graph structure of the Historical plagiarism found during the intersection of the multiple assignments.

```
Graph{
        student8_student8--student5_student5;
        student9_student9--student14_student14;
        student17_student17--student3_student3;
        student19_student19--student24_student24;
        student11_student11--student5_student5;
        student30_student30--student20_student20;
        student19_student19--student9_student9;
}
```

LISTING B.1: The graph code of the historical plagiarists

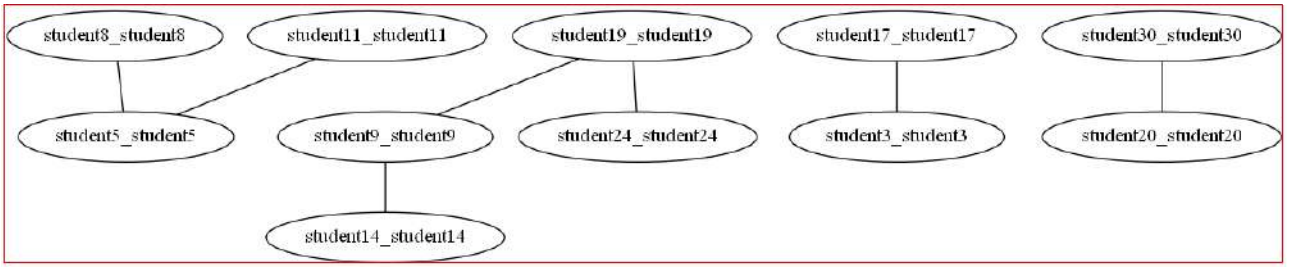


FIGURE B.1: The generated graph of the historical plagiarists.

TABLE B.1: The similar pairs during the manual inspection of the source code programs.

Assignments	Source code programs	Counts of Lines of Code	Counts of Tokens
MC_Lab_6_512	Student8, Student10 Student11	87	2030
MC_Lab_3f_506	Student8, Student11	28	574

Assignment 1: Student8 Program

```
1  package com.example.animationgame;
2  import java.util.ArrayList;
3  import java.util.Random;
4  import android.annotation.SuppressLint;
5  import android.app.Activity;
6  import android.graphics.Canvas;
7  import android.graphics.Paint;
8  import android.os.Bundle;
9  import android.view.Display;
10 import android.view.MotionEvent;
11 import android.view.View;
12 import android.view.View.OnTouchListener;
13 import android.view.Window;
14 import android.view.WindowManager;
15
16 public class MainActivity extends Activity implements OnTouchListener{
17 ArrayList <Ball> balls = new ArrayList<Ball>();
18 DrawView drawView;
19 Ball b1;
20 Ball b2;
21 Ball b3;
22 int     width;
23 int height;
24 int yup =0;
25 int zup=0;
26 Random r = new Random();
27 @SuppressWarnings("deprecation")
28 @SuppressLint("ClickableViewAccessibility") @Override
29 public void onCreate(Bundle savedInstanceState) {
30 super.onCreate(savedInstanceState);
31 // Set full screen view
32 getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
33 WindowManager.LayoutParams.FLAG_FULLSCREEN);
34 requestWindowFeature(Window.FEATURE_NO_TITLE);
35 drawView = new DrawView(this);
36 setContentView(drawView);
37 drawView.requestFocus();
38 drawView.setOnTouchListener((OnTouchListener) this); //Add this line when doing touch
      events
39 Display display = getWindowManager().getDefaultDisplay();
40 width = display.getWidth();
41 height = display.getHeight();
42 balls.add(new Ball(100, 100, 2, 0, (Integer) width, height));
```

```
balls.add(new Ball(200, 200, 3, 0, (Integer) width, height));
balls.add(new Ball(300, 180, 1, 0, (Integer) width, height));
}

public void doDraw(Canvas canvas, Paint paint) {

for(int i =0; i < balls.size(); i++)
{
Ball ball = balls.get(i);
canvas.drawCircle((int) ball.x, (int) ball.y, 5, paint);
ball.update(0.5);
}
}
@Override
public boolean onTouch(View arg0, MotionEvent arg1) {
// TODO Auto-generated method stub
Display display = getWindowManager().getDefaultDisplay();
width = display.getWidth();
height = display.getHeight();
int action = arg1.getAction();
int newx = r.nextInt(width-width/2-1);
int newy = r.nextInt(height- height/2);
yup = (int) arg1.getY();
zup = (int) arg1.getX();
if(action==MotionEvent.ACTION_DOWN){
int x= (int)arg1.getX();

if(x<=width*0.5){
balls.add(new Ball(newx, newy, 2, 0, width, height));
}
else
balls.clear();
}
return false;
}
}

```

LISTING B.2: Student8 Program

Assignment 1: Student10 Program

```
package com.example.animationgame;
import java.util.ArrayList;
import java.util.Random;
import android.annotation.SuppressLint;
import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.os.Bundle;
import android.view.Display;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.view.Window;
```

```
14 import android.view.WindowManager;
15
16 public class MainActivity extends Activity implements OnTouchListener{
17         ArrayList <Ball> balls = new ArrayList<Ball>();
18         DrawView drawView;
19         Ball b1;
20         Ball b2;
21         Ball b3;
22         int     width;
23         int height;
24         int yup =0;
25         int zup=0;
26         Random r = new Random();
27         @SuppressWarnings("deprecation")
28         @SuppressLint("ClickableViewAccessibility") @Override
29         public void onCreate(Bundle savedInstanceState) {
30                 super.onCreate(savedInstanceState);
31                 // Set full screen view
32                 getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
33                 WindowManager.LayoutParams.FLAG_FULLSCREEN);
34                 requestWindowFeature(Window.FEATURE_NO_TITLE);
35                 drawView = new DrawView(this);
36                 setContentView(drawView);
37                 drawView.requestFocus();
38                 drawView.setOnTouchListener((OnTouchListener) this); //Add this line
    when doing touch events
39                 Display display = getWindowManager().getDefaultDisplay();
40                 width = display.getWidth();
41                 height = display.getHeight();
42                 balls.add(new Ball(100, 100, 2, 0, (Integer) width, height));
43                 balls.add(new Ball(200, 200, 3, 0, (Integer) width, height));
44                 balls.add(new Ball(300, 180, 1, 0, (Integer) width, height));
45         }
46
47
48         public void doDraw(Canvas canvas, Paint paint) {
49
50                 for(int i =0; i < balls.size(); i++)
51                 {
52                         Ball ball = balls.get(i);
53                         canvas.drawCircle((int) ball.x, (int) ball.y, 5, paint);
54                         ball.update(0.5);
55                 }
56                 }
57         @Override
58         public boolean onTouch(View arg0, MotionEvent arg1) {
59                 // TODO Auto-generated method stub
60                 Display display = getWindowManager().getDefaultDisplay();
61                 width = display.getWidth();
62                 height = display.getHeight();
63                 int action = arg1.getAction();
64                 int newx = r.nextInt(width-width/2-1);
65                 int newy = r.nextInt(height- height/2);
66                 yup = (int) arg1.getY();
67                 zup = (int) arg1.getX();
```

```
68                  if(action==MotionEvent.ACTION_DOWN){
69                          int x= (int)arg1.getX();
70
71                          if(x<=width*0.5){
72                                  balls.add(new Ball(newx, newy, 2, 0, width, height));
73                          }
74                          else
75                          balls.clear();
76                          }
77
78                  return false;
79                  }
80  }
81
```

LISTING B.3: Student10 Program

Assignment 1: Student11 Program

```
1   package com.example.animationgame;
2   import java.util.ArrayList;
3   import java.util.Random;
4   import android.annotation.SuppressLint;
5   import android.app.Activity;
6   import android.graphics.Canvas;
7   import android.graphics.Paint;
8   import android.os.Bundle;
9   import android.view.Display;
10  import android.view.MotionEvent;
11  import android.view.View;
12  import android.view.View.OnTouchListener;
13  import android.view.Window;
14  import android.view.WindowManager;
15
16  public class MainActivity extends Activity implements OnTouchListener{
17          ArrayList <Ball> balls = new ArrayList<Ball>();
18          DrawView drawView;
19          Ball b1;
20          Ball b2;
21          Ball b3;
22          int     width;
23          int height;
24          int yup =0;
25          int zup=0;
26          Random r = new Random();
27          @SuppressWarnings("deprecation")
28          @SuppressLint("ClickableViewAccessibility") @Override
29          public void onCreate(Bundle savedInstanceState) {
30                  super.onCreate(savedInstanceState);
31                  // Set full screen view
32                  getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
33                  WindowManager.LayoutParams.FLAG_FULLSCREEN);
34                  requestWindowFeature(Window.FEATURE_NO_TITLE);
35                  drawView = new DrawView(this);
36                  setContentView(drawView);
```

```
37                drawView.requestFocus();
38                drawView.setOnTouchListener((OnTouchListener) this); //Add this line
      when doing touch events
39                Display display = getWindowManager().getDefaultDisplay();
40                width = display.getWidth();
41                height = display.getHeight();
42                balls.add(new Ball(100, 100, 2, 0, (Integer) width, height));
43                balls.add(new Ball(200, 200, 3, 0, (Integer) width, height));
44                balls.add(new Ball(300, 180, 1, 0, (Integer) width, height));
45        }
46        public void doDraw(Canvas canvas, Paint paint) {
47                for(int i =0; i < balls.size(); i++)
48                {
49                        Ball ball = balls.get(i);
50                        canvas.drawCircle((int) ball.x, (int) ball.y, 5, paint);
51                        ball.update(0.5);
52                }
53        }
54        @Override
55        public boolean onTouch(View arg0, MotionEvent arg1) {
56                // TODO Auto-generated method stub
57                Display display = getWindowManager().getDefaultDisplay();
58                width = display.getWidth();
59                height = display.getHeight();
60                int action = arg1.getAction();
61                int newx = r.nextInt(width-width/2-1);
62                int newy = r.nextInt(height- height/2);
63                yup = (int) arg1.getY();
64                zup = (int) arg1.getX();
65                if(action==MotionEvent.ACTION_DOWN){
66                        int x= (int)arg1.getX();
67
68                        if(x<=width*0.5){
69                                balls.add(new Ball(newx, newy, 2, 0, width, height));
70                        }
71                        else
72                        balls.clear();
73                        }
74        return false;
75                }
76 }
```

LISTING B.4: Student11 Program

Assignment 3: Student8 Program

```
1  import java.util.*;
2  import java.util.Scanner;
3  public class Program{
4
5          public static void main(String args[]){
6                  Scanner in = new Scanner(System.in);
7                  ArrayList<Rating> allratings = new ArrayList<Rating>();
8                  String input = in.nextLine();
9                  while (!input.equals("-1")){
```

```
10                          String str = input;
11                          String[] vals = str.split(";");
12                          allratings.add(new Rating(vals[0],Integer.parseInt(vals[1])));
13                          input = in.nextLine();
14                  }
15                  System.out.println(getAverage(allratings));
16          }
17          public static double getAverage(ArrayList<Rating> v){
18                  double average =0, total=0;
19                  for(int i = 0;i<v.size();i++){
20                          total = total +v.get(i).getScore();
21                  }
22                  average = total/v.size();
23                  return average;
24          }
25  }
```

LISTING B.5: Student8 Program

Assignment 3: Student11 Program

```
1   import java.util.*;
2   import java.util.Scanner;
3   public class Program{
4
5           public static void main(String args[]){
6                   Scanner in = new Scanner(System.in);
7                   ArrayList<Rating> allratings = new ArrayList<Rating>();
8                   String input = in.nextLine();
9                   while (!input.equals("-1")){
10                          String str = input;
11                          String[] vals = str.split(";");
12                          allratings.add(new Rating(vals[0],Integer.parseInt(vals[1])));
13                          input = in.nextLine();
14                  }
15                  System.out.println(getAverage(allratings));
16          }
17          public static double getAverage(ArrayList<Rating> v){
18                  double average =0, total=0;
19                  for(int i = 0;i<v.size();i++){
20                          total = total +v.get(i).getScore();
21                  }
22                  average = total/v.size();
23                  return average;
24          }
25  }
```

LISTING B.6: Student11 Program