Linnæus University
School of Computer Science, Physics and Mathematics

Degree Project

Plagiarism Detection in Java Code

Ahmad Gull Liaqat & Aijaz Ahmad
2011-06-29
Subject: Software Technology
Level: Master
Course code: 4DV01E

Linnæus University
School of Computer Science, Physics and Mathematics

SE-351 95 Växjö / SE-391 82 Kalmar
Tel +46-772-28 80 00
dfm@lnu.se
Lnu.se

Acknowledgement

We feel a great sense of accomplishment to be able to complete this thesis, as part of the requirement for our Master's degree. We would have not been able to do this on our own without the supports of the people whom are important to us.

Firstly, we would like to thank our supervisor, Associate Professor Dr. Jonas Lundberg for his continuous support and guidance. His advice and comments are crucial in our progress to complete this thesis.

We would also like to thank our parents and family for their words of encouragement and their supports throughout our study period. Our friends are also important to us, for without them this experience would be less meaningful.

Abstract

The issue of plagiarism is often discussed in academia, it relates to the act of one person citing or quoting another individual's work without giving credit to the original writer. In programming, students are submitting assignments every year in the form of Java code files. There is a possibility that students may copy the Java code files from another source without properly crediting the original writer or programmer, intentionally or unintentionally. This is also a form of plagiarism. The main focus of this thesis is divided into two parts. The first part is focused on a written Java code program that could assist academician in detecting plagiarism in Java programming. The program should be able to read a large number of Java code files and to identify programming codes that are identical or almost identical when the files are being compared to one another. The second part of the thesis is a report on the properties of different approaches in detecting plagiarism in Java code files and recommendation on best approaches for future work or study.

Keywords

Plagiarism, Plagiarism detection in Java Code, Plagiarism Detection, Java Code plagiarism detection tool, Plagiarism detection related work, SIM, Software Similarity Tester, YAP, Yet another Plague, MOSS, Measure of Software Quality, Normalization in Plagiarism Detection, Text Based Plagiarism Detection tools, Levenshtien Algorithm.

Table ofContent

Abbreviations

SIM: Software Similarity Tester
YAP: Yet another Plague
MOSS: Measure of Software Quality
LCS: Longest Common Subsequence
RKR-GST: Running Karp Rabin Greedy String Tilling
SC: Source Code
TS: Token Stream
RC: Remove Comments
Ren: Renaming
FC: Full Class
M: Method

List of Tables

List of Figurs

1. Introduction

This chapter consists of the introduction of the research, which includes the problem description and motivation and sturcture of our report.

1.1 Plagiarism

The act of plagiarizing is when you use someone's else words and ideas without giving due credits to the original writer and regard the work as your own.[2].

Plagiarism is defined by S. Hannabuss as

> "is the act of imitating or copying or using somebody else's creation or idea without permission and presenting it as one's own [4]."

Plagiarism is similar to the act of stealing. If the act of stealing someone's car, watches, cell phone and others is punishable by law, surely the act of stealing someone ideas, thoughts, writings or words is also considered wrongful.

However, this does not mean that students should not view other's works or sources for references. Taking in opinions and ideas from experts in order to increase knowledge is a good thing, but most importantly is to make sure that the origin of the sources and the relevant cited references are duly credited. Failure to credit the original writer is considered as plagiarism.

If someone is caught for committing plagiarism in a college or a university, the consequences could be severe, from failing the course to being expelled from the institution. Recently German Defense minister Karl-Theodor zu resign after being surrounded by a plagiarism scandal at University of Bayreut in PhD.

1.2 Plagiarism in Coding

Plagiarism in coding is not entirely a new phenomenon. The issue has been discussed and studied previously by researchers to identify the severity of the problem and what factors contribute to the act of plagiarism. In programming assignment [11], plagiarism does not necessary only involve copying the source code but if someone include comments, program input data and interface designs that can also be considered as plagiarizing..

In 1977 the people start show concerns about plagiarism in source code. A survey in 2002 shows that 85.4% in a class of 137 students at Monash University and 69.3% in a class of 150 students at Swinburne University are involved in source code plagiarism and they admit this dishonesty [1]. Both samples show a high tendency among students to cheat or plagiarize. Involvement of students in plagiarism is due to many reasons like sometime there are some drawbacks in checking assignments in academics and sometime student just feel lazy to write his own code. One example of its practice in programming courses is when a student submits work of which part was copied from another student's work, assuming that instructors will not find out about the truth.

Generally, plagiarism in coding is hard to be detected because of the similar coding used for the same application. Plagiarism in coding is easy to do, but difficult to detect (as cited in N. Wagner, 2000)[4]. Students copy all or part of a program from some source or from

different sources and submit the copy as their own work. This includes students who collaborate and submit similar work. Such plagiarism is felt to be common, though the true extent is hard to assess. When a teacher in a programming course gives same assignment problems to all students then all students have to work on same problems. So it is the possibility that some students write source code of problems by their own and remaining students just take the code from them and amend it like changing of variable names, changing the order of statements, functions and variables of class and submit it. These types of modifications in source code are very difficult to catch.

There are two types of source code modification they are lexical change and structural change. It is very easy to do lexical change. One can do lexical change without any knowledge of programming language by using just a simple editor. Structural changes ca not be done without any programming language.

> ”Structural changes are changing iterations, changing conditional statements, changing the order of statements, changing procedure to function and vice versa, changing procedure call within the body of the procedure and vice versa, adding statements that will not affect the output of the program [4].”

We illustrate in the following example. Suppose the simplest scenario, if someone writes a code to add two numbers like;

```
packageImplementation;
publicclass test
 {
        publicstaticint add(int a, int b)
{
        returna+b ;
}
        publicstaticvoid main(String[] args)
 {
        int c;
        c= add(2,3);
        System.out.println("Sum = " +c);
 }
}
```

Someone else use the same logic to return sum without seeing this code, then this will considered as plagiarism? Surely the answer is no. So we can say that two persons can have the same logic and they can write same code for same functions.

For the above scenario of same code we have to define some conditions on the size of code like that if 10 lines in a sequence are similar in two person’s code then it will be consider as plagiarism or we can calculate some percentage (%) of similarity of code between two java files by using some formula.

1.3 Goal of Thesis

The goal of the thesis is to create a plagiarism detection tool to assist lecturers in higher institutions to detect plagiarism of Java code in students’ assignments. The program will

enable lecturers to identify the authenticity of student's works by comparing the submitted assignments with the other student's assignments. Lecturers will be able to detect if the students commit plagiarism, if the comparison between two Java files resulted in at least threshold value or higher degree of similarity.

The second goal of the thesis is to analyze the normalization techniques. The normalization techniques are applied to the plagiarism detection tool, in order to analyze the data. Based on the result, the technique with the highest percentage is considered to be more reliable in detecting plagiarism in Java codes and thus, will be recommended.

1.4 Problem Description

In this scenario, the Java code files that the students have been submitting, as part of their assignments are being analyzed using the plagiarism detection program that have been written in Java. Every new submitted Java code file is being compared with the other student's assignments to detect plagiarism. In this case, 'our system' must be scalable because of the increasing number of Java code files. This type of plagiarism detection is called 'Historical Plagiarism Detection'. The result of the research is presented in different approaches in this report.

1.5 Motivation

The motivation to carry out the research is based on the need for a program that is able to simplify the task of lecturer in checking accurately through student's coding of Java files, line by line in ensuring no duplication is being made from other sources. This is to avoid too much time spent on examining a single assignment and to ensure that equal time is given on the checking of each assignment. The research includes suggestion on different methods in detecting plagiarism in Java code files and recommendation on best approaches for future work and study references.

1.6 Report Structure

Chapter 2 is related to the work of plagiarism detection. It contains information on the current software that is used for plagiarism detection and also includes the algorithms and approaches used in the software.
Chapter 3 describes the main idea i.e. normalization process and Levenshtien algorithm.
Chapter 4 explains experimental setup.
Chapter 5 is about experimental results.
Chapter 6 is conclusion and suggestions for future work.

2. RelatedWork

As an educator, it is important to detect plagiarism in programming assignment because it is a nuisance to the educational procedure. Plagiarism in programming assignment has long existed and it is a conventional form of dishonesty [5].

In evaluating plagiarism detection system, we are looking at two main systems, which are: text-based system and code-based system. Text-based systems approaches are not suitable for code plagiarism detection as discussed by Burrows et al. [6] and Arwin and Tahaghoghi [3]. Burrows et al. [6] explains in details below the two systems on plagiarism detection, namely text-based system and code-based system, including the attributed oriented and structure-oriented approaches. We also include the discussion on some of the available tools for checking plagiarism detection code like SIM, MOSS, Plague, YAP and JPLAG.

2.1 Text-based systems

In determining whether a text has been plagiarized or not, text analysis can be carried out. Statistics is used to decide whether a text has indeed been plagiarized based on the frequency counts on the words and sentences. For example, a text is being compared to other texts in the database to look for similarity in the words used and sentence composition. The higher frequency of words and sentences count will indicate higher probability of plagiarism [15].

CopyCatch and WordCHECK are two available standalone tools and plagiarism.org is a web based tool. When a document is submitted to plagiarism.org to check for plagiarism, first of all it will generate a representation in its database for comparison with other representations in the database and also with the documents in World Wide Web. WordCHECK match keyword profiles by maintaining internal database of submitted documents and document to check [15].

> "CopyCatch measures pair wise similarity between texts based on word frequencies [15]."

Text-based systems approaches are not suitable for code plagiarism detection as it ignores coding syntax and furthermore, changes in copied code will be able to avoid detection.

Source code is less rich in textual and literacy properties then text, so due to this it is not helpful to apply text-based tools for source code plagiarism checking. Source code contains more significant information then lexical ones in formal and structural properties. For example in source code plagiarism checking, algorithmic structure of operations is more meaning full then systemic difference between identifiers in the code [16].

The same opinion also shared by Arwin and Tahaghoghi that state

> "the nature of program source code makes it difficult to apply simple text-based detection techniques. Copied code is typically altered to avoid detection [3]."

2.2 Attribute-oriented code-based systems

An attribute count is defined as the attribute counting systems that measure some property of an individual system [13].

Attribute-oriented system targets the key properties of code and appraises these key properties. A system that is explained by Donaldson et al. is using four attributes and measuring them for plagiarism detection [6].

These four attributes are.

- "The number of unique operators
- The number of unique operators
- The number of occurrences of operators
- The number of occurrences of operands [6]."

Two code programs are approximated by measuring the difference between these above attributes. It is observed that attribute-oriented systems are very narrow .Attribute oriented systems are only helpful in close copies of code. A most used way of copying code is to add or remove unnecessary code. As this system check code line by line so there is possibility of a big difference between the number of occurrence of operands and operators and number of operands and operators. It is also very difficult to detect plagiarism in a program of very large code

2.3 Structure-oriented code-based system

Structure metric systems is based on the combination of two techniques; searching for similarity in a structural representation of two pieces of source code and also applying the attribute counting techniques [13].Structure-oriented systems change the structure of program and then compare them to check plagiarism .In these systems elements like comments, white spaces and variable names are overlooked because they can be customized easily [6].

There are many structure oriented approaches are available to detect plagiarism in source code. Every approach focuses on definite characteristics of code. Like, some approaches are only designed to check plagiarism of source code written in different programming languages. Some approaches are available to check plagiarism of very complex code modifications but they need a long time to detect similarity. In structure-oriented systems the similarity between two programs is measured on the base of similarity of structure of two source codes. In this approach source code is compared in two phase [3]. In first phase token stream of programs is generated and the second phase is the comparison phase of token streams by using string matching algorithms. Software Similarity Tester (SIM), JPlag, Measure of Software Quality (MOSS), Plague and Yet Another Plague (YAP) are the existing structure-oriented code-bases systems.

2.3.1 SIM

SIM is used to detect plagiarism of code written in Java, C, Pascal, Modula-2, Lisp, Miranda [6]. SIM is also used to check similarity between plain text files. SIM converts the

source code into strings of token and then compare these strings by using dynamic programming string alignment technique. This technique is also used in DNA string matching [14]. The alignment is very expensive and exhaustive computationally for all applications because for large code repositories SIM is not scalable. The source code of SIM is available publically but it is no more actively supported.

2.3.2 MOSS

MOSS is available free to use in academics and it is accessible as an online service .Moss support Ada programs, Java, C, C++, plain text and Pascal. At the same time MOSS also support UNIX and windows operating systems [1] .First of all MOSS convert source code into tokens and then use robust winnowing algorithm. Robust Winnowing Algorithm is introduced by Schleimeretal but the internal detail of working of this algorithm is confidential. This algorithm takes the document fingerprints by selecting a subset of token hashes [6].

In the comparing process of set of files,

> "MOSS creates an inverted index to map document fingerprints to documents and their positions within each document. Next, each program file is used as a query against this index, returning a list of documents in the collection having fingerprints in common with the query [6]."

The number of matching fingerprints of each pair of document in the set of files is the result of MOSS. MOSS sort these results and show highest-score matches to user.

2.3.3 Plague

One of the earliest structure-oriented systems is Plague. Plague only support programs written in C. This tool works in several steps. In the first step source code is converted into structure profiles. After this Plague use Heckel algorithm to compare generated structure profiles of first step. The algorithm is basically designed for plain text files and it is introduced by Paul Heckel. Plague returns results in list and then use an interpreter to process this list to show results in a way, so that common user can understand it easily [4].

2.3.4 YAP

YAP stands for yet another plague [1]. As by name we can see that YAP is an enhancement of Plague. YAP has three versions. In 1992 Micheal Wise introduced YAP1. Then after this YAP2 came into market a bit later after YAP1 and finally the YAP3 of YAP family was released in 1996. YAP family process have two phases. In the first phase a token file is created of each source file and this phase is called as generation phase. In the second phase every token file is compared to other token file and the result of this comparison is called as percent match value. This match value varies from 1 to 100. The user of YAP can set the lowest value. YAP shows result in a plan text file. If token pairs have percent match value larger then lowest value set by user then the matching pair will be consider as plagiarized pair [14].

All three versions of YAP use different algorithms and this implementation. The difference of these three algorithms is considered as main difference between these three versions. In YAP1 LCS algorithm was implemented. The main drawback of this algorithm was order of token strings. This algorithm affects the order of tokens. If someone just change the order of statements in source code, then it is very difficult to caught plagiarism in YAP1 due to implementation of LCS algorithm. In the YAP2 Hackel algorithm was implemented.

> "However, this algorithm does not prioritize longer substrings Identical substrings being identified by this algorithm are mostly in the form of several short substrings scattered over the source code [4]".

YAP3 overcome the drawbacks of YAP1 and YAP2 .YAP3 implements the Running Karp- Rabin Greedy String Tiling(RKR-RGST) algorithm [4].

2.3.5 JPlag

JPlag is available publically as free accessible service [12]. JPlag can be used to check plagiarism of source code written in C, C++, Scheme and Java. We gave a directory of programs as input in JPlag. First of all source code in the directory is parsed and then transformed into token strings. After transformation JPlag compare these strings by using Running Karp-Rabin Greedy String Tiling(RKR-RGST) algorithm. The comparison result then shown in HTML file, which can be visited by using any browser. In the HTML file of results main page consist of pairs of programs that are assume to be plagiarized. User can see results of different pairs separately. Different fonts in result file of HTML shows different things, like the pairs with similar code will have different font from other pairs. In this way user can analyze results very easily [4]. JPlag has been used extensively by various academic institutions, both at the undergraduate and the graduate level, reaching the submissions as many as 500 participants since fall 2001, receiving and processing dozens of submission each month [12].

3. Main Idea

This chapter is about our main idea of thesis. In this chapter we will discuss how we convert java files into normalized form throgh normalization. We will also explain Levenshtein Distance algorithm with example. Our plagiarism detection formula is also explained in this chapter. At the end we will explain token stream.

3.1 Introduction

Our plagiarism detection tool works in two steps. First it needs to convert Java code into a form which is easy to compare. This is done in the Normalization phase. After normalization we use Levenshtein distance Algorithm on normalized files to detect plagiarism in Java code. After getting result of from Levenshtein Distance Algorithm we apply our Plagiarized Value formula to get result.

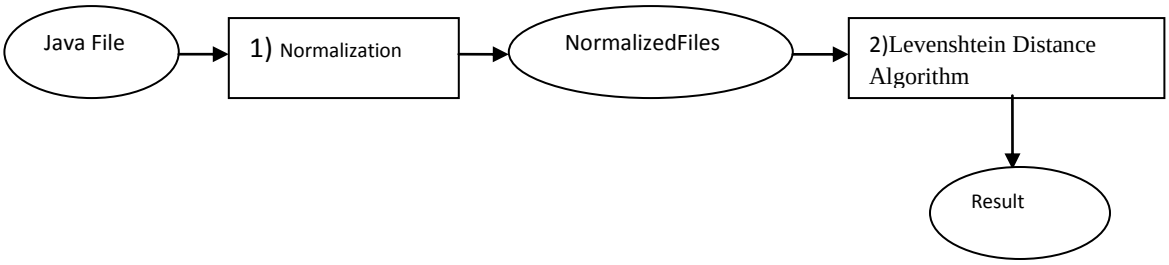

Figure3.1: Main Idea

3.2 Normalization

Normalization is the process of rewriting all Java files in a certain way that will simplify the comparison later on. Normalization can be applied on two formats i.e. Source file or Token stream. It includes:

- Removing comments.
- Uniform renaming of identifiers.
- Sorting of all class members according to their size.

We use the RECODER 0.92 library in this process of Normalization.

3.2.1 Removing Comments

The first step of normalization is to delete all comments of Java files. We know that comment have no effect on functionality of a code. We remove comments because someone can just add extra comments in code so that it looks like different from copied one. Comments can also affect the results of our plagiarism detection tool, as the algorithm we use is comparing the source code character by character or token by token after normalization phase.

3.2.2 Uniform Renaming of Identifiers

In the second step we rename fields and methods of classes. We can choose to rename full class or either just fields or just methods of class. We do renaming of identifiers because if a student who want to copy code of his fellow can just change the name of fields and methods of Java code to make it look like different. This is the easiest way in programming assignments to copy code.

3.2.3 Sorting of all Class Members According to their Size

In the third step of normalization, class members are sorted in ascending order of a selected Java file. Class members are sorted in this order; fields come first on top then after fields constructors will come and at the end methods of classes will come. Methods of class are sorted on the basis of statements i.e. if a method has two statements and other have three statements then the method with two statements will come first and the method with three statements will be sorted after this. Inner classes are sorted according to number of methods in classes. Reason for sorting of class members is that, if we have a code of 200 lines and someone just change the order of fields, methods, constructors and of inner classes then it is very difficult to detect code plagiarism.

3.3 LevenshtienAlgorithm

The Levenshtien Distance Algorithm (LD) measure the similarity between source string(s) and target string(t).The similarity between two strings is measured as distance between source and target string. Number of substitutions, insertion and deletion required to convert source strong into target string are referred as difference between these files. The Lavenshtien Algorithm result increase with difference between the strings.

If source string(s) is "fellow" and target string (t) is "follow" then LD=1. It means that one substitution is required to change source string into target string [8].

If s is "this thesis is related to plagiarism" and t is "the thesis is about plagiarism", then LD(s,t) = 10 [8].

Levenshtien distance was used in Plagiarism Detection. It is also been used in Spell Checking, Speech recognition and DNA Analysis [8].

3.3.1 The Levenshtien Algorithm

1. "Set n to be the length of s.
   Set m to be the length of t.
   If n = 0, return m and exit.
   If m = 0, return n and exit.
2. Initialize the first row to 0..n.
   Initialize the first column to 0..m.
3. Examine each character of s (i from 1 to n).
4. Examine each character of t (j from 1 to m).
5. If s[i] equals t[j], the cost is 0.
   If s[i] does not equal t[j], the cost is 1.
6. Set cell d[i,j] of the matrix equal to the minimum of:

a. The cell immediately above plus 1: d[i-1,j] + 1.
b. The cell immediately to the left plus 1: d[i,j-1] + 1.
c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.

7. After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m] [8].”

3.3.2 Explanation

The algorithm calculates the number of operations needed to change String1 to String2. We will explain it manually here [10].

Suppose we want to calculate Levenshtien Distance between strings “MORNING” and “EVENING”. First of all we have to take “MORNING” and “EVENING” horizontally and vertically in Rows and Columns in a table. It does not matter which string we are taking horizontally or which one vertically. We have to insert one extra row and one extra column in the table. Now we will fill first row and first column from left to right and top to bottom in ascending order from 0...n and 0…m. The table after this step will become like this.

		E	**V**	**E**	**N**	**I**	**N**	**G**
	0	1	2	3	4	5	6	7
M	1							
O	2							
R	3							
N	4							
I	5							
N	6							
G	7							

Table 3.1: After first step

Now the real part begins after this. It’s the calculation part. We start it by checking the column of “E” (from Evening) from top to down, then “V” from top to down and so on…

We have to match letter in the column with the letter in the line.
If they match then we simply put the value “Above Left” cell in the matching cell.
If they do not match then we have to take minimum of three values and add 1 to it. The three values will be from cell above, above left and left of current cell.

Let’s take “E” from “EVENING” and “M” from “MORNING” .They are not matching so we have to take minimum of cell above(that is 1) , above left(that is 0) and left(that is 1) of current cell. Minimum is 0 here. When we add 1 in 0 then it will become 1 which we put in current cell.

		E	**V**	**E**	**N**	**I**	**N**	**G**
	0	1	2	3	4	5	6	7
M	1	1	2	3	4	5	6	7
O	2	2	2	3	4	5	6	7
R	3	3	3	3	4	5	6	7
N	4	4	4	4	3	4	5	6
I	5	5	5	5	4	3	4	5
N	6	6	6	6	5	4	3	4
G	7	7	7	7	6	5	4	**3**

Table 3.2: After matching all characters

We will fill the table in the same way until we reach at some matching character "N" of "EVENING" and "N" of "MORNING", here at this cell we simply have to take 'Top Left'(which is 3 here) value of current cell and put it in current cell. We will treat all matching characters in this way. When the whole table is filled out then the last value of last column and last value of last row (here it is 3) is the Levenshtien Distance value. So it shows that we need 3 substitutions to change one string into to other string.

We apply Levenshtein Algorithm on Source file/Token Stream, Source file after removing comments, Source file/Token Stream after renaming identifiers, Source file/Token Stream after sorting of all class members according to their size.

3.3.3 Complexity

If both strings (Source and Target) have length ´n` then the time-complexity of Levenshtien Algorithm will be $O(n^2)$. Space-complexity of this algorithm is also $O(n^2)$ if we need to keep a track-back of whole matrix and it is O (n) for two rows of matrix [9].

3.3.4 Results by LD

If we select two directories of Java files for checking plagiarism of source directory then our tool take one Java file of source directory at a time and compare it with all Java files of target directory and this process of picking a Java file from source directory and comparing it with all files of target directory will continue till the end of all files of source directory.

We apply the same normalization process on both files of source directory and target directory one by one. After normalization we get normalized files in the form source string and target string. Lets

$Source\ String = CS$

$Target\ String = TS$

After this our program gives these two strings to Levenshtein Distance Algorithm. This algorithm gives us numeric value as difference of two strings. Let

$DIistance = Diff$

When we have difference of these two strings then we apply our formula to calculate plagiarized value. The Plagiarized Value formula is

$$Plagiarized\ Value = \left\{1 - \frac{Diff}{Max(CS, ST)}\right\} * 100$$

Our goal is to calculate the similarity of two strings. As we know that Levenshtein Distance algorithm gives us difference between two strings. Max(CS,ST) will give us larger length value of either of target string or of source string. We are taking Max(CS,ST) to divide our distance value because we need to calculate percentage of similarity Lets explain this formula with a simple example

Suppose

$CS = plagiarismdetectiom$

$TS = plagiarism$

Then by applying Levenshtein Distance algorithm

$Diff = 14$

$Max(CS, ST) = 19$ *(Length of CS)*

Let's put these values in our formula

$$PlagiarizedValue = \left\{1 - \frac{Diff}{Max\ (CS,ST)}\right\} * 100 = \left\{1 - \frac{14}{19}\right\} * 100 = 26$$

$$Plagiarized\ Value = 26$$

This means that 26 % is similarity between two strings.

If this value of plagiarism will be more than threshold value then we suppose that source file is plagiarized with target Java file and the result, names and location of both files will be shown.

3.4 Token Stream

When characters of a Java program are grouped into symbols then it is called as token. A token can be an identifier, keyword, separator, operator, literal and comment. Programmer choose the identifies; keywords are names that are already defined in a programming language; separators are punctuators; operators are symbols that produce results by operating arguments; Literals can be Numeric Textual, Logical and reference and comments are line or block. Although Comments are accepted as token by the Java compiler, it abandons these for further processing [14].

4. Experimental Setup

This chapter is about tool and experimantal setup.We will expalin user interface of tool in detail and then we will explain benchmark for experiment.

4.1 Tool

The user interface of our plagiarism detection tool is shown in Figure 4.1. First of all we select source directory and target directory by pressing Browse buttons corresponding to 'Select Directory 1' and 'Select Directory 2'. Source Directory should contain Java files which we want to check for plagiarism and Target Directory should have Java files to compare. If we need Jar then we can select Jar by pressing Browse button against 'Select jar's Directory'.

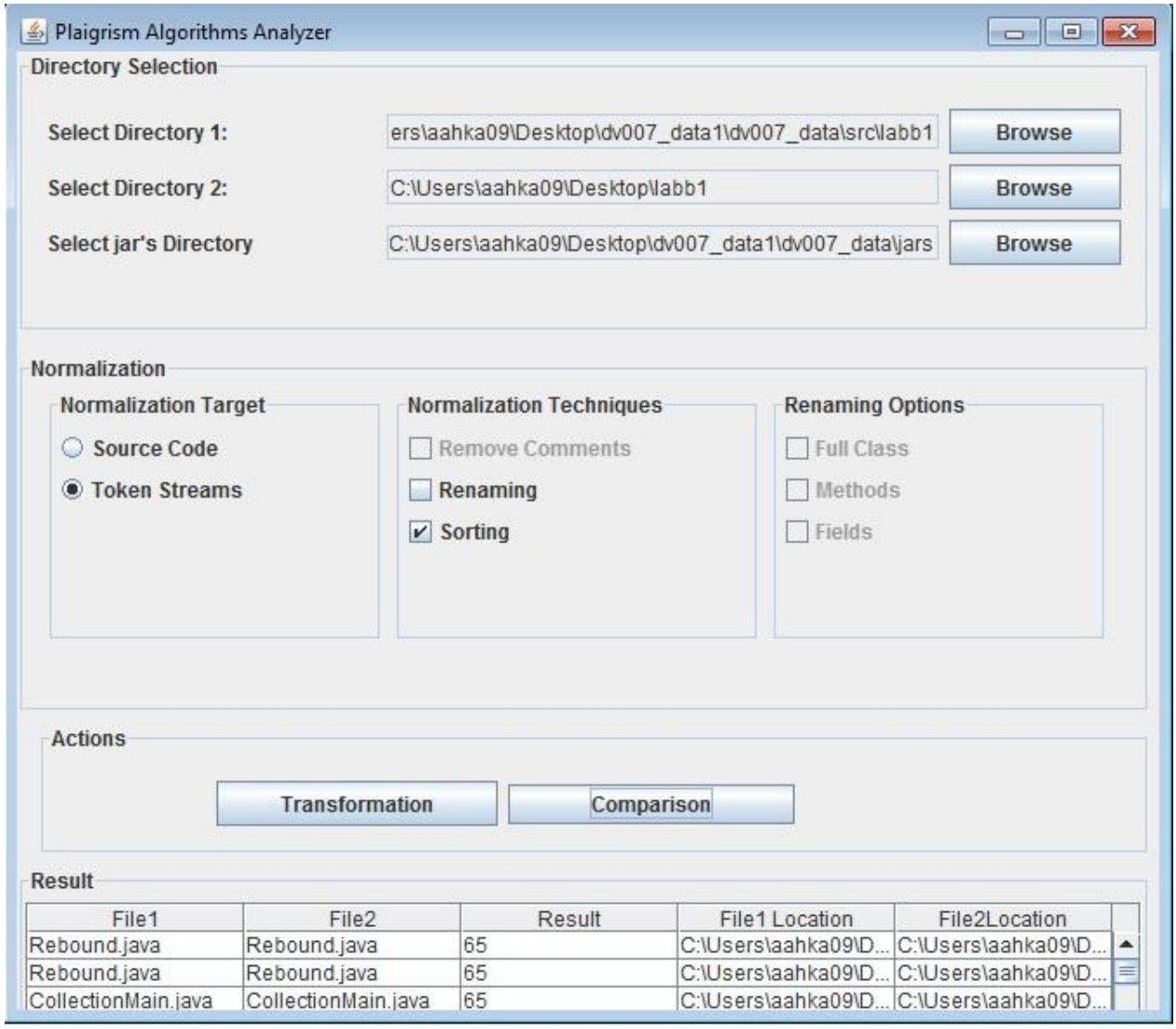

Figure4.1: User Interface

After selecting directories or Jar we go to the Normalization phase. In this phase first of all we have to select one method i.e. Source Code or Token Stream in Normalization Target. If we want to compare source code of our selected Java files then we have to enable 'Source Code 'option or if we want to compare token stream of Java files then we have to enable 'Token Stream' option

After selecting a Normalization Target we select which Normalization Techniques we want to apply on the Normalization Target. We have three options to select i.e. Remove Comments, Renaming and Sorting. We can select all three options, two of them or either just one. If our selections is 'Source Code' in Normalization Target then all three options of Normalization Techniques can be selected and if we select 'Token Stream' in Normalization Target then only two options 'Renaming' and 'Sorting 'will be available to select because when we tokenize Java code it will delete all comments from our source code, so Remove Comments option will be disable to select.

The last part of Normalization phase is Renaming Options. These renaming options will be only available to select if we select 'Renaming 'in Normalization Techniques. If we select 'Full Class' in 'Renaming Options' then we ca not select Methods or Fields and vice versa.

Now after all these selections of Normalizations we have actions part of our tool. There are two Buttons to press there. One is Transformation and other is Comparison. The transformation button is just to transform our Java code according to selections in Normalization and save the transform Java Target and Source files in project directory. It is here to press so that user can see that the tool is working correctly by examining Normalized files. The Comparison button is doing two tasks at the same time. It transforms the files and also calculates the result for plagiarism. If we do not want to examine our transform files then we can just press Comparison button and there is no need to press Transformation button.

Result is the final part of our tool. This part only shows the name, result and location of those file which are similar more than a given threshold value. File1 and File2 are the names of Source and Target Java files which are supposed to be plagiarized. Our tool does not check Interface Java files for plagiarism detection because in student assignments most of Interfaces are supposed to be same and teachers are always interested in Implementation of interfaces. However we will hope that in next version of this tool there will be an option for user in User Interface to include or ignore Interfaces. This plagiarism detection tool also does not check the directories with same student names. The Result column shows the percentage of similarity between File1 and File2. This result is calculated by Plagiarized Value formula presented in Chapter 3. Our tool shows result in ascending order. File1 location and File2 location give the path of File1 and File2. If we click on this path then our tool will open the Java file in notepad.

Our Plagiarism Detection tool will give error and will not work, if there is any code error in any Java file which is under examination for plagiarism.

4.2 Benchmark

We take data (Java files) from the practical assignments which students have been submitted for marking grades. We have 3 directories LAB1, LAB2, LAB3. LAB1 have 17 groups of students and each student group folder have approximately 20 Java files in it. LAB2 have 13 groups and each group folder have approximately 18 Java files in it. LAB3 have 12 different groups of students and each student group folder have approximately 13 Java files in it. We run our tool with different Normalization Techniques on the given test data. If we find a pair of files with a similarity greater than the threshold value, we consider it to be the case of suspicious plagiarism. We verify results by manual checking of each suspicious pair.

4.3 Experiments

We run our tool with all available options and present all plagiarized files in table 4.1. We found this list of files with a threshold value of above 70, which means that they are similar to more than 70%. We are taking this threshold value on the base of experiments by running our tool again and again with different threshold values. On the base experiments, we came to know that in most cases those suspected pairs of files which are giving result more than 70 have plagiarism in them.

Here is the list of pairs of plagiarized files in table 4.1.

Result

Case	File1	File2
P1	labb2.Group1.PrintJavaMain.java	labb2.Group6.PrintJavaMain.java
P2	labb3.Group2.Word.java	labb3.Group8.Word.java
P3	labb2.Group1.ExceptionMain.java	labb2.Group9.ExceptionMain.java
P4	labb2.Group3.PascalMain.java	labb2.Group6.PascalMain.java
P5	labb2.Group3.MP3Track.java	labb2.Group7.U5_MP3.java
P6	labb2.Group3.MP3Track.java	labb2.Group10.MP3Track.java
P7	labb2.Group4.U5_MP3.java	labb2.Group10.MP3Track.java
P8	labb1.Group5.Creature.java	labb1.Group10.Creature.java
P9	labb1.Group5.spelplan.java	labb1.Group10.CreatureRitPanel.java
P10	labb3.Group2.LinkedQueue.java	labb3.Group11.java

Table 4.1: List of plagiarized files

We will use this list (Table 4.1) in next chapter for analyzing different combinations of Normalization Target and Normalization Techniques.

As an example, here is the source code of two plagiarized Java files.

Group2.Word.java

```
public class Word implements Comparable<Word> {
	private String word;

	/**
	 * Konstruktor, skapar ett objekt frÃ¥n ett visst ord.
	 * @param str Ordet som skall representeras av objektet.
	 */
	public Word(String str) {
		this.word = str;
	}

	/**
	 * Returnerar en strÃ¤ngrepresentation av ordet.
	 * @return String
	 */
	public String toString() {
		return word;
	}

	public boolean equals(Object other) {
		if (this.compareTo((Word) other)==0) {
			return true;
		} else {
			return false;
		}
	}
	/**
	 * Compute and return a hashcode for the word.
	 * @return int hashcode
	 */
	public int hashCode() { // Heltal baserat pÃ¥ strÃ¤ngen
		int hc = 0;
		for ( int i =0;i<word.length();i++) {
		   char c = word.charAt(i);
		   hc += Character.getNumericValue(c); // ASCII number
		}
		return hc;
	    }

	/**
	 * Compares two words lexicographically
	 * @param w Objekt av typen Word som skall jÃ¤mfÃ¶ras
	 */
	public int compareTo(Word w) {
		return (word.compareToIgnoreCase(w.word));
	}
}
```

Group8.Word.java

```
public class Word implements Comparable<Word>{


	private String word; //skälva ordet alltid små bokstäver

	public Word(String str){

		word = str.toLowerCase();

	}


	public int hashCode() { //returnerar hashcode
		int hc = 0;

		for ( int i=0;i<word.length();i++) {
			char c = word.charAt(i);
			hc += Character.getNumericValue(c);
		}
		return hc;

	}

	  public boolean equals(Object other) {
		//använder compareTo för att se om 2 ord är lika

		if (this.compareTo((Word) other) == 0) return true;
		else return false;

	}


	public int compareTo(Word o) {	//returnerar värde 0 om de
är lika, - om den ena är mindre + om den samma är större
		return this.word.compareTo(o.word);

	}
	public String toString() {

		return word;
	}

}
```

Table 4.2: Source Code of plagiarized files

Table 4.2 shows a pair of files that we consider being plagiarism. If we examine the pair of files in Table 4.2, we can see that both files have same number of methods and fields. Even logic is same but the order of methods and commenting style is different. If we examine after removing the comments and reordering the methods according to their size then we can see that, these two files are almost identical. If we see non-trivial methods of this pair of files like hash code or equal, we can easily examine that these non-trivial methods are the same, so there are very less chances that they can be same by accident. That's why, we are considering this pair of files as plagiarized.

5. Experimental Result

In this chapter, we present the comparison of LAB1 with LAB1, LAB2 with LAB2 and LAB3 with LAB3.

5.1 Introduction

To conduct these experiments, we use different normalization techniques and select one normalization targets at a time. In all our comparison results we are ignoring GUI main program files, exception handling files and teacher provided files. Because most GUI main program follow a template structure provided by the teacher. And exception handling files are very small file which has either 4 or 5 lines of code and that code is also very similar for everyone. We will from now on refer to these files as *excluded files.*

We are comparing the remaining results like this, with every run of our tool we get a list of files, we list file pairs having a similarity above the threshold. The results part of table 5.1 is a summary of this. We will use these results to motivate why one normalization target is better than other and which normalization technique is giving better results.

Here are some abbreviations which we are using in statistics tables.
SC = Source Code, TS = Token Stream, RC = Remove Comments,
Ren = Renaming, FC= Full Class, M =Method

Table 5.1 shows the statistics of LAB1, LAB2 and LAB3 with `Source Code` as Normalization Target with all combinations of Normalization Techniques.

	SC	**SC+RC**	**SC+RC +Ren(FC)**	**SC+RC +Sorting**
No of PairsDetected	70	80	95	88
Suspicious Cases	5	11	13	13
True Positives	P4,P7,P8	P3, P4, P5,P6, P7,P8,P9	P3, P4, P5,P6, P7,P8,P9	P1, P2, P3, P4, P5, P6,P7,P8,P9,P10
False Positives	2	4	6	3
Precision	0.60	0.63	0.53	0.77
ComparisonTime (Min)	2.93	1.70	1.61	1.64

Table 5.1: Source code comparison for lab1, lab2 and lab3

In this table `No of pairs Detected` is the total number of results that we get after running our tool with a threshold value of above 70.`Suspicious Cases ` are number of pairs when we have removed all *excluded files*. `True Positive` means number of actual plagiarism in Table 4.1 and `False Positive `the number of wrong results. Precision is the ratio of True Positives and Suspicious Cases which tells us that how accurate plagiarism detection rate is Pn(n=1,2,3…10) are the pairs of those plagiarized files which we gave in Table 4.1(Chapter 4).

5.2 Conclusion of Source Code Comparison

Statistics in Table 5.1 shows that Source Code as Normalization Target with Remove Comments and Sorting (SC + RC + Sorting) has higher precision than other combinations. We found 10 pairs of suspected cases and after comparing Table4.1,we found 8 True

Positive pairs, which is showing highest rate of reliably and accuracy. The comparison time of this combination is not higher than other combinations so we can say that this combination is efficient too.

Now we select Token Stream as Normalization Target and get the statistics with different combinations in Table 5.2

	TS+RC	**TS+RC +Ren(FC)**	**TS+RC +Sorting**
No of PairsDetected	86	98	91
Suspicious Cases	12	18	13
True Positives	P1,P5, P6,P7, P8,P9	P1, P3, P5, P6, P7,P8	P1, P2,P3,P4, P5, P6, P7, P8,P9 ,P10
False Positives	6	12	3
Precision	0.50	0.33	0.77
ComparisonTime (Min)	0.62	0.61	0.58

Table 5.2: Token Stream comparison for lab1, lab2 and lab3

5.3 Conclusion of Token Stream Comparison

If we look at table 5.2, we can see once more that removing comments with renaming has less precision compared to other combinations. So, we can see that the renaming improvement is questionable from a teacher viewpoint. Although better than pure remove comments in detecting plagiarism, it will generate a lot more work due to higher number of false positives. Removing comments and sorting has once again higher precision as compared to other techniques.

So after analyzing the Table 5.1 and Table 5.2, we have reached the conclusion that removing comments always give good results but we have even more accurate results when we combine removing comments with sorting. Sometimes removing comments with renaming also give good results but renaming always detects a large number of false positives as compared to any other normalization techniques.

We also observed that token stream is just as accurate and faster as compared to source code. Although token stream detect large number of false result as compared to source code, the performance of our tool with token stream is faster than when using source code.

6. Conclusion and Future Work

This chapter is about conclusion and of future work. We will present conclusion of different normalization techniques and our recommendations for future work.

6.1 Conclusion

Our Plagiarism detection tool is a structured oriented code-base system which used a different approach then existing structured oriented systems like MOS, YAP, SIM, JPlag and Plague. We use normalization techniques such as Renaming, Sorting and Removing Comments techniques and apply them on both source code and token stream. No other plagiarism detection tool implements all these normalization techniques together. We used Levenshtien algorithm in our analyzer for comparison of normalized files, which is not being used in above mentioned systems. Table 5.1 and Table 5.2 are showing that our Plagiarism detection tool is working well for large directories containing hundreds of Java code files for comparison and plagiarism detection.

By using different Normalization techniques with different combinations, we can detect different kinds of plagiarism like if someone has added comments in Java source code or changed and other has changed the fields and methods order etc. We also observe from the results in Table 5.1 and Table 5.2 that some combinations of normalization techniques and normalization target are good and some are not, like removing comments and sorting mostly give better results than renaming technique. Similarly, in the case of Normalization Target, token stream mostly give better performance than source code. We have shown in Table 5.2 that we can get best results by using removing comments and sorting as normalization techniques with token stream.

We implement Levenshtein algorithm for the actual file comparison which is also known as edit distance algorithm. It takes two files in the form of string as input and returns a difference of two files as output. We are getting good result using this algorithm but for the better performance, we can use another algorithm in future.

We achieved both goals of our thesis. Our first goal was to create a plagiarism detection tool to assist lecturers. In chapter 4 we explain our tool in detail and in table 4.1 we show the results of our plagiarism detection tool. Our second goal was to analyze different normalization techniques. Table 5.1 and Table 5.2 are showing this analysis.

6.2 Future Work

Our first suggestion is to use Abstract Syntax Tree (AST) as normalization target with source code and token stream and then compare results because by comparing Abstract Syntax Tree (AST) of two Java programs, we can get more precise results as compare to source code an token stream, because Abstract Syntax Tree comparison is done by comparing node by node and there are very less chances to getting wrong results.

If user wants to ignore some Java files during plagiarism detection then we recommend adding this option in the tool User Interface in which user can add these file names. In some cases teacher ask students to design and implement their own interface, so we recommend

to add an option in User Interface, so that user can include or ignore Interfaces for plagiarism detection.

We also suggest trying more algorithms like LCS, RKR-RGST, TokenCompress etc on normalized files and then compare the results with our tool results. These algorithms can improve the efficiency of our tool.

References

[1] Ameera Jadalla, Ashraf Elnagar. PDE4Java: Plagiarism Detection Engine for Java Source Code: A Clustering Approach. City, 2007.

[2] Chris Park. Rebels Without a Clause: Towards an Institutional Framework for Dealing with Plagiarism by Students. Journal of Further and Higher Education Vol. 28, No. 3, August 2004.

[3] Christian Arwin and S.M.M. Tahaghoghi. Plagiarism Detection across Programming Languages. Proceedings of the 29th Australasian Computer Science Conference, 48:277–286, 2006.

[4] Cynthia Kustanto, Inggriani Liem. Automatic Source Code Plagiarism Detection.10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, pp 481-482, 2009.

[5] Lefteris Moussiades, Athena Vakali. PDetect: A Clustering Approach for Detecting Plagiarism in Source CodeDatasets, The Computer Journal Vol. 48 No. 6, 2005.

[6] Steven Burrows, S. M. M. Tahaghoghi, Justin Zobel. Efficient Plagiarism Detection for Large Code Repositories. School of Computer Science and Information Technology, RMIT University, Melbourne, Australia, pp 158-159, 2006/

[7] http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html, reviewed 10th April 2011

[8] http://www.merriampark.com/ld.htm, reviewed 20th April 2011

[9] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic/Edit/, reviewed 05th May 2011.

[10] http://blog.exec-tunes.de/2010/10/11/levenshtein-distance-edit-distance-easily-explained/, reviewed 17th May 2011

[11] http://eprints.dcs.warwick.ac.uk/121/1/cosma_joy_ieeetoe_51.pdf, reviewed 17th May 2011.

[12] Lutz Prechelt, Guido Malpohl, Michael Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. Journal of Universal Computer Science, vol. 8, no. 11 (2002).

[13] http://www.acs.ase.ro/mcs/SourceCodePlagiarism/Lancaster_2003.pdf, reviewed 17th May 2011.

[14] Edward L. Jones. Plagiarism Monitoring and Detection-Towards and Open Discussion. Department of Computer and Information Sciences Florida A&M University, 2001

[15] http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/tokens/lecture.html, reviewed 23rd May 2011.