CSCI – 788 Computer Science MS Project

Capstone Project Report

Title: Implementation and Analysis of Graph Algorithm Used for Software Plagiarism Identification

Advisor: Dr. Carlos Rivero

Student: Sowgandh Krishnaa Nandamuri (sn2638@rit.edu)
Contents

Figures 4
Tables 6

1. Abstract 7
2. Introduction 7
3. Summary of previous plagiarism detection techniques used 8
4. Project Objective 9
5. System Specifications 9
6. System Overview 10
   6.1 Plagiarized Program 10
   6.2 Code Parser 11
      6.2.1) List of files needed to generate parser and lexer files 11
      6.2.2) Steps to be followed to setup environment for ANTLR 11
      6.2.3) Mapping that defines the names for the nodes 13
      6.2.4) Recognition of edges for a pair of nodes for the graph 15
   6.3 Graph Generator 16
   6.4 Graph Visualization 17
   6.5 Graph Storage 20
      6.5.1) Setting up environment for Neo4j Database 20
   6.6 Subgraph Isomorphism 22
      6.6.1) Isomorphism Testing 22
7. Testing the system with Real World Examples. 23
   7.1 List of websites considered for real world examples 23
   7.2 Sample examples of real world examples 25

8. Comparison of the PDG based with other plagiarism detection tools 28
   8.1 Analysis and comparison of the results with the current implementation 31
   8.2 (MOSS & JPLAG) VS Current Implementation 32
   8.3 A brief description of some other tools used to detect plagiarism 33

9. Conclusion 34

10. Challenges 34

11. Future Improvement of Project 34

12. References 34
Figures

6. System Overview

Figure 6 (A) Block diagram of the system architecture

6.2 Code parser

Figure 6.2 (A) Block diagram indicating usage of ANTLR.jar

Figure 6.2 (B) Code sample used by ANTLR to extract syntax of language

Figure 6.2 (C) Code sample used by ANTLR to recognize nodes and edges

Figure 6.2 (D) Image indicating recognition of nodes by ANTLR

Figure 6.2 (E) Image indicating node relations

Figure 6.2 (F) Image indicating the edges created between a pair of nodes

6.3 Graph generator

Figure 6.3 (A) Block diagram for graph generation

6.4 Graph visualization

Figure 6.4 (A) Block diagram for graph visualization

Figure 6.4 (B) Sample programs

Figure 6.4 (C) visualization of original program

Figure 6.4 (D) visualization of plagiarized program

6.5 Graph Storage

Figure 6.5 (A) Block diagram describing storage of graph

Figure 6.5 (B) Image visualizing the graphs stored in Neo4j database

6.6 Subgraph Isomorphism

Figure 6.6 (A) Block diagram indicating testing of isomorphism for graphs
7. Testing the system with Real World Examples

7.2 Sample real world examples

Figure 7.2 (A) Image 1 of real world example program 25
Figure 7.2 (B) Image 1 of real world example to test system 26
Figure 7.2 (C) Image 2 of real world example program 26
Figure 7.2 (D) Image 2 of real world example to test system 27

8. Comparison of the PDG based tool with other plagiarism detection tools

Figure 8 (A) Image Indicating similarities in code detected by MOSS 29
Figure 8 (B) Image Indicating similarities in code detected by JPLAG 31
Tables

Table 1: Mapping of the node names of the graph to use case 13
Table 2: Mapping of the nodes along with their different colors 18
Table 3: Comparison of PDG based technique with other tools 32
Table 4: Summary of some tools used for plagiarism detection 33
1. Abstract:

The issue of software plagiarism occurs at various levels, ranging from programming courses taken by students at schools to industries that deal with the creation of state of art software products. Plagiarism that occurs in programs developed by students during their course of study is less challenging to detect (in most cases) than the plagiarism that occurs in industries. Consider a scenario, where a software company is developing a product in a short span of time, in such cases software developers, of the company may be tempted to use some existing open source software code as part of their development. In such a typical scenario, plagiarism occurs in a small portion of a large software code written for the product being developed, detecting plagiarism in these situations will be quite a challenge. There have been quite a few software plagiarism detection tools that were created in the past but techniques such as changing order of statements and random insertion of code may create issues with these tools.

In order to reduce the number of failures in detecting software plagiarism, this project deals with a different technique to detect plagiarism in software. It deals with a methodology of building graphs over the code. Graphs are automatically built over the various functions present in the original and plagiarized program. Now the graphs are compared and checked if they are subgraph isomorphic to one and another, if isomorphism is detected between each other, then plagiarism is reported.

2. Introduction:

The following project report describes a project that is mainly used to carry out detecting of core-plagiarism in software. Consider a situation in which a company is dealing with a development of a complex software product which involves a number of lines of code. Development of such type of code right from the base would take a lot of time and effort, hence in such a case developers may be tempted to use some existing software code obtained from open source projects directly into their product. For example, developers may use code pertaining to user interface, input/output components that are open source. In these cases plagiarism of code is present in a very small portion of a large software code of the product. Such a scenario is referred to as core-plagiarism. Though there are quite a few plagiarism detection tools that are available they are susceptible to some complex plagiarism techniques that are applied. Some disguises may include usage of alternate structures like using a while loop instead of a for loop, changing the order of statements without affecting the meaning of the program etc. The project introduces a mechanism that helps us detect core part plagiarism more effectively than its predecessor techniques. The mechanism used in the project tries to obtain patterns from the original program and tries to find these patterns in the plagiarized code. A specific set of graphs known as Program Dependence Graphs (PDG’s) are constructed over the various procedures or functions present in both the original and plagiarized program. Next the graphs from the original program are compared with those of the plagiarized program.
to check if one is subgraph isomorphic to one and another. If isomorphism is detected between them then plagiarism is reported. The advantage of PDG based methodology is that in such a graph both, flow of data and control in a function or procedure are taken into consideration. Thus any plagiarism that is attempted must not alter these two factors in order for the program to run in a similar manner, since the data and control dependencies remain the same even in the plagiarized code, checking for subgraph isomorphism works fine.

3. **Summary of previous plagiarism detection techniques used:**

In an attempt to understand and study some other techniques used to detect plagiarism, a number of papers were studied and the section below summarizes the techniques used.

3.1) **Text-Based**

Some plagiarism detection tools are text/string based. In such tools the program is seen as a sequence of strings or lines of text. So in these cases, the original program is taken as an input and sequences of lines are studied and stored. Once the plagiarized program is also fed to the system, it is searched for the corresponding sequence of text present in the original program. A main drawback present in this approach is that it does not take into consideration the structural elements or syntax of the language. Another disadvantage with this type of tool is that, the tool can be easily tricked with identifier renaming and using an alternate syntax present in the language if possible. Some tools that use this concept apply a few types of transformations such as removal of comments, whitespace etc.

3.2) **Token-Based**

Tools that use this kind of approach, parse or transform the entire original program into a set of tokens. The tokens obtained from the original program are then stored in the memory. Later the plagiarized code is also fed as input to the tool. Since the kind of parsing or transformation applied to a program remains the same, the plagiarized code is changed to a set of tokens in a similar manner. These tokens are now compared with those of the original program, if the level of similarity is greater than a particular threshold, then plagiarism is identified. This token based approach, is slightly more robust when compared to the test based approach as it remains language independent and the tokens formed are no way related to the syntax of programming language. However, even these tools are vulnerable to some plagiarism techniques such as usage of alternate control structures.

3.3) **Tree-Based**

Tools that use a Tree-Based technique usually have parsers and lexers associated with each programming language. So, based on the programming language used in the source program the corresponding lexer or parser is used. The tools in general, use the CFG (Context free language) concept to build the trees. The entire source program is parsed and stored as a tree
structure. The tools also provide a manner to visualize these trees. These trees ignore variable and literal names to remain generic. In a similar fashion trees over the plagiarized code are also built. Now subtrees of the plagiarized code are searched for in the tree of the original code. Various tree traversal algorithms such as preorder, postorder and inorder are used internally.

3.4) PDG-Based

Program Dependence Graphs (PDG) based methodology are more advanced when compared to the techniques described previously. These graphs contain information that have control and flow of data. They provide more abstraction than other tools and provide more semantic understanding of the program present underneath. Once program dependence graphs are built over both the original and the plagiarized code, graph isomorphism testing can be done on both. If the level of isomorphism is considerably large then plagiarism is identified.

3.5) Metric-Based

Tools that are Metric-Based compare metrics of the original and plagiarized code rather than comparing the programming codes directly. They have some functions called as fingerprinting functions present in them. These functions can be used over a particular method or class or even a single statement. Thus these fingerprinting functions are applied on both the source and plagiarized procedures and their results are compared. They are applied on various levels of the program such as class or method to improve the plagiarism detection process.

4. Project Objective:

To design a system to be able to detect plagiarism that occurs in a small portion of a large software code. Typically the system assumes that programming logic is present in a single function or procedure, plagiarism is detected based on comparison of the functions of original program and the plagiarized program.

5. System Specifications:

The system is designed to work for programs developed in Java only.

Operating System: Windows 10

Coding Language : Java

Tools : Eclipse
6. **System Overview**

The block diagram given below describes the overall architecture of the system.

![Block diagram of the system architecture](image)

The first step in the system is to parse the program fed as input to generate the graph. The parser can be used to identify the nodes and edges of the graph. Once the nodes and edges are identified, the graph generator uses this information to construct the graph, now that the graph is ready the graph visualizer provides a couple of graphical user interface options to visualize the graph. Now to have a permanent record of constructed graphs, a graph database, namely Neo4j is used, which stores the nodes and edges between the nodes of the graph. Lastly, the constructed graphs of both the original and plagiarized program is fed as input to the isomorphism testing framework that identifies if one is subgraph isomorphic to another or not.

6.1) **Plagiarized Program**

The plagiarized program that is introduced to the system is assumed to have the following alterations in code. The logic of the program remains the same, but plagiarism disguises applied to the code corresponds to one or more of the techniques mentioned below.

**Identifier Renaming**

Some of the identifiers, literals and variables were deliberately renamed in the plagiarized code.

**Rearranging Statements**

The syntax statements of the original program were taken and used in the plagiarized code with rearrangement of their order without the program logic getting affected.

**Alternate Control Structures**

Some of the code in the plagiarized program has also been written by using alternate control structures in the plagiarized code. For example, the for-loop in the original program was replaced by a while-loop in the plagiarized code.
Random Code Insertion.

Another attempt of plagiarism that was made, was to insert an abrupt and unnecessary code into the original program to try and break the graphs that are constructed.

6.2) Code Parser

In order to parse a code a tool called ANTLR is used. ANTLR stands for (Another tool for language Recognition) that helps to read, parse and interpret the code written in various programming languages such as JAVA, C#,C++. ANTLR can be used to parse and interpret programs written in Java and extract the components of programs such as datatypes, control structures such as for-loop, while-loop etc.

ANTLR takes a grammar file as input and generates a lexer and a parser.

It basically defines how to split the program into language specific tokens. A grammar file defines the tokens of the programming language. Thus, we have different grammar files for Java, C#, C++ etc. that define the tokens for the syntax of the programming language. They are of extension .g4. They also determine if Java or Python classes must be generated. It is provided with a variety of options like the generation of abstract syntax trees, code interpretation etc.

![ANTLR Diagram](image)

Figure 6.2 (A) Block diagram indicating usage of ANTLR.jar

Lexer runs through the code and splits the code into components into tokens which is then passed on to the parser which helps interpreting the code.

6.2.1) List of files needed to generate parser and lexer files

- antlr-4.5.1-complete.jar file – Jar file needed to parse programs
- Java8.g4 – Grammar file defined for programs written in Java
- Test.java – Helps in creating the listener file to walk through the code
- Sample Program to parse

6.2.2) Steps to be followed to setup environment for ANTLR

- Download and install the jdk and jre (which is required to install Java Virtual Machine and development environment).
- Download the jar file (antlr-4.5.1-complete.jar) from ANTLR official website.
• Setup the class-path JAVA_HOME variable, that points the jdk location and setup a location to the bin present inside the jre folder.
• Download the grammar file and the sample program to parse to a folder where you have the jar file.

Upon running the command I obtained Java8Lexer.java, Java8Parser.java, Java8BaseListener.java and Java8Listener.java and various other class files are generated to parse the code.

Test.java is another Java file that is used to actually feed in the sample Java program being parsed and it calls the Listener class file to walk through the program.

To compile Test.java file the following command is used.

javac –cp .;antlr-4.5.1-complete.jar Test.java

The Test.java file accepts the sample program that is being parsed as a runtime argument,

After setting up the environment for ANTLR we get a separate auto generated class known as Java8BaseListener.java whose image is given below. The class has a number of overridden methods. Each of these methods are related to the syntax of the programming language. Also, each of these methods are classified into two, one named entry and other named exit. The corresponding methods are called as and when the appropriate syntax is encountered by the tool. For example, if ANTLR encounters an object creation statement, then the entry method of object creation is called and once ANTLR goes onto the next statement the corresponding exit method is called. The appropriate program logic is written in these methods to construct the nodes and connect two nodes by means of edges. The image shown below gives a snapshot of a of the methods generated by Java8BaseListener.java
Once the nodes for the graphs were identified, the nodes of the graph were classified as declaration, assignment, control and increment/decrement. The section given below defines the names given to nodes for the graph that is being constructed, the names are provided based on their corresponding use case in the program.

### 6.2.3) Mapping that defines the names of the nodes

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Name</th>
<th>UseCase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decl</td>
<td>Declaration</td>
<td>Used to declare a variable</td>
</tr>
<tr>
<td>Control</td>
<td>Control structure</td>
<td>Used for structures such as for-loop, while-loop, if etc</td>
</tr>
<tr>
<td>Assign</td>
<td>Assignment</td>
<td>Used to assign a value or variable to another variable</td>
</tr>
<tr>
<td>inc/dec</td>
<td>Increment/decrement</td>
<td>Pre and Post decrement/increment</td>
</tr>
<tr>
<td>Expr</td>
<td>Expression</td>
<td>Any expression other than previous ones</td>
</tr>
<tr>
<td>Return</td>
<td>Return</td>
<td>Returns a particular value</td>
</tr>
</tbody>
</table>

Table1: Mapping of the node names of the graph to the use case

Given below is an image that show a sample program that was parsed using ANTLR.
Figure 6.2 (C) Code sample used by ANTLR to recognize nodes and edges

The image given below indicate the recognition of nodes and edges by ANTLR after the program is parsed.

Identification of nodes for the graph after using ANTLR

Figure 6.2 (D) Image indicating recognition of nodes by ANTLR
6.2.4) Recognition of edges for a pair of nodes for the graph

The edges between a pair of nodes for the Program dependence graph (PDG) that is constructed can be classified into two namely,

Data Dependency Edge:

There is a data dependency edge between two nodes (say) n1 and n2, if there is some value or variable that is assigned to n1 (either directly or indirectly), which is used n2 either directly or indirectly.

Control Dependency Edge:

There is a control dependency edge between two nodes (say) n1 and n2 if, where n1 is a control vertex and the truth of the condition tested in n1 causes n2 to be executed.

In order to preserve the nodes and edges created, some user defined classes namely Vertex.java and Edge.java were created. The image given below shows the sample edges between a pair of nodes of the graph.

![Diagram showing sample edges between nodes](Figure 6.2) (E) Image indicating node relationships

The image above shows a set of sample edges between a pair of nodes that were constructed between a pair of nodes.
Hence, likewise the various edges have been constructed between a pair of nodes and those edges that were recognized using ANTLR are given below in the image.

![ANTLR Edge Creation Log](image)

Figure 6.2 (F) Image indicating the edges that have been created between a pair of nodes.

6.3) **Graph Generator**

Now that the nodes and edges of the graphs are recognized and identified by using ANTLR.jar file, the next step was construction of the graph.

The block diagram shown below gives a brief description of the methodology used to generate the graphs.

![Graph Generation Block Diagram](image)

**Figure 6.3 (A) Block diagram for graph generation**
In order to make this possible user defined classes such as Node.java and Edge.java have been defined, with the corresponding setters and getters methods.

Hence once the nodes and edges of the graphs have been identified and constructed successfully using ANTLR, we can generate the nodes and edges for the graph using the user defined classes mentioned above.

Next an ArrayList of nodes and edges obtained from the user defined classes are created, they maintain not only the nodes present in the graph, but also the edges between two nodes that were defined. To preserve these highly informative lists we serialize objects of the class where these lists are defined. Serialization of these objects are done to preserve them in the local file system, this enables us to get back to the graph when needed, thus until this stage the system has in memory solution to preserve the graphs.

6.4) Graph Visualization

The block diagram given below shows methodology for visualizing the graph

![Figure 6.4 (A) Block diagram for graph visualization](image)

The nodes and edges that were previously stored in the file system as objects are now recovered and stored into temporary lists, with one list maintaining the nodes of the graph and the other list maintaining the edges between a pair of nodes.

A separate applet class is written to implement the JGraphT library, the library has some special methods associated with them, through which we can specify the coordinates of the nodes, their color and the edges between them.

Using the JGraphT library we can directly specify the names of the vertices along with their coordinates to be displayed on the Applet, by calling a specific method, namely addVertex(), similarly the edges can be constructed by specifying the source and target vertex in the addEdge() method.

In order to differentiate between the different types of vertices present in the graph, each is associated with a different color. The mapping for the colors is defined in the table below.
<table>
<thead>
<tr>
<th><strong>Vertex-Type</strong></th>
<th><strong>Color</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>decl,( declaration)</td>
<td>Red</td>
</tr>
<tr>
<td>assign,(assignment)</td>
<td>Green</td>
</tr>
<tr>
<td>control,(control)</td>
<td>Blue</td>
</tr>
<tr>
<td>inc/dec,(increment/decrement)</td>
<td>Magenta</td>
</tr>
</tbody>
</table>

Table2: Mapping of the nodes along with their different colors

The image given below shows, the two programs namely, BubbleSort.java and BubbleSortWhile.java of which the former is the original program, while the latter one is the plagiarized one.

![Sample programs](image.png)

Figure 6.4 (B) Sample programs

The images given below show the visualization of graphs generated for both the original and plagiarized programs.
Figure 6.4 (C) visualization of original program

Figure 6.4 (D) visualization of plagiarized program
6.5) **Graph Storage**

Until now the graphs are constructed and stored in the file system in terms of a serialized Java object. At this stage we have only an in memory technique, used to hold the graph structures, but in order to have a permanent record of the graphs being considered we needed a database to hold the graphs. Thus, in order to permanently store graphs, a graph database called Neo4j was used.

![Block diagram describing storage of graph](image)

The above block diagram indicates the methodology used to store the graphs constructed permanently using the database. The Neo4j Database helps us store the nodes and the relationships between the nodes. Thus after deserializing the object the list containing the nodes and edges are obtained.

6.5.1) **Setting up environment for Neo4j Database**

- Download the jar file associated with Neo4j and add it onto the class path.
- Download Neo4j-CE to visualize the graph constructed.

A separate Java class is used to store the nodes and edges in the database. This class basically relies on the Neo4j jar files to store nodes and edges. The database is created in the local file system by means of this class. The types of nodes such as assignment, declaration, control are defined by means of enumerations. Also the types of edges such as data dependency or control dependency are also defined by another enumeration. Each node is associated with a label name and a color is associated with the type of node. The edges between a pair of nodes in Neo4j are referred to as relationships.

A single database of Neo4j can be associated with more than one graph, hence as a result, for a given program, a Neo4j database is created to store the graphs of both the original and the plagiarized program.
Now in order to fetch the stored graphs from Neo4j, it supports a separate language like SQL known as CQL (Cypher query language). Thus, for example, in this case we can fetch all nodes that have both a control or data dependency incoming edge.

Many graph traversal algorithms such depth first search, breadth-first-search are built in within the Neo4j database. So whenever a query is fired from the user to fetch a set of nodes from the graph, it actually calls one of these built in algorithms.

Some indexing mechanism are also available with the Neo4j database just as other databases. Usually indexing is done for nodes that are frequently accessed in order to reduce the cost of the corresponding input/output operation.

Neo4j also provides us with another visualization tool namely Neo4jCe. Once this software is downloaded it can be made to point to the local databases that hold graphs, it helps us have a visualization of graphs, present in the database. The image given below shows two graphs one of the original program and the other of the plagiarized program.

Figure 6.5 (B) Image visualizing the graphs stored in Neo4j database.
6.6) **Subgraph Isomorphism**

Prior to this step, the program dependence graphs are constructed and stored using two ways, first one being an in memory solution where the nodes and edges for the graphs are stored as Java objects and the second one deals with storing the graph directly in a graph database such as Neo4j. The next step would be testing of subgraph isomorphism between the graphs of the original and plagiarized program. The manner in which this step is done is explained in the block diagram given below.

![Figure 6.6 (A) Block diagram indicating testing of isomorphism for graphs.](image)

As the above block diagram indicates, the graph information is read from the file system and later, used in the generation of a .gdl file. This file is basically fed as input to the isomorphism testing framework. It’s the representation of the graph, it is basically a text file that has the nodes present in the graph and the edges that connect these nodes. Thus, we have two .gdl files, one that corresponds to the original program and the other that corresponds to the plagiarized code. Now both these .gdl files are fed as input to the isomorphism testing stage.

6.6.1) **Isomorphism Testing**

In order to run the isomorphism testing, a separate framework is loaded onto the Eclipse IDE. This framework basically is a large Java project that has different algorithms associated with it to carry out isomorphism. Some of the algorithms include GraphQL, Nema etc. The implementation of the framework is beyond the scope of this project and its use-case is only taken into consideration.

The given below image shows the results of isomorphism testing on the original and plagiarized programs of the BubbleSort program mentioned previously.
7. **Testing the system with Real World Examples.**

In order to carry out testing of the system with programs obtained from the real world, a number of websites were studied. These websites host basically programming contests/challenges for students. Some of these websites also have various student solutions to a particular challenge/contest held in the past, hence these programming solutions were obtained, graphs were constructed for the same and tested for inter-student solution plagiarism.

7.1) **List of websites considered for real world examples**

The section below describes the list of web sites considered, to take some real world examples into account

1) **Google code Jam 2013 :**

Reference:  [https://code.google.com/codejam/archive.html](https://code.google.com/codejam/archive.html)

Google Code Jam is a programming competition conducted by Google .

2) **TopCoder**

Reference: [https://www.topcoder.com/](https://www.topcoder.com/)

TopCoder is a company that runs programming contests. They conduct computer programming competitions in the Java, C++, and C# languages

3) **Sphere Online Judge**
Reference: http://www.spoj.com/problems/classical/

Sphere online judge is a professional coding competition, with a support for a large number of compilers and languages

4) CodingBat
Reference: http://codingbat.com/
CodenBalt is a website, which provides problems in Java and Python.

5) ACM-ICPC
Reference:https://icpc.baylor.edu/
This contest/challenge is conducted by IBM which mainly involves algorithmic programming problems. They support two languages like C/C++ and Java.

6) HackerEarth

7) CodeChef
Reference: https://www.codechef.com/viewsolution/9680759
CodeChef is an Indian company. It is a programming contest conducting organization which hosts online contests and events for programmers from across the globe.

8) Codeforces.com
Reference: http://codeforces.com/
Codeforces is another popular website where students post their solutions for programming contests/challenges. This website does have past coding solutions

Among the above stated competitions, only CodeChef and Codeforces provided student-solutions for past programming contests. However, there was one restriction on Codeforces, the solutions had a lot of complex syntax such as static classes, bit operations, multiple procedure calls within a function. In such a scenario the graph builder would not perform well as we are taking into consideration certain limitations on the syntax being parsed, hence CodeChef was chosen as the website from where we could get a few examples to test for plagiarism.
7.2) **Sample examples of real world examples**

![Image of real world example program](image1)

---

**Figure 7.2 (A) Image 1 of real world example program**
Figure 7.2 (B) Image 1 of real world example to test system

```java
public static void main (String[] args) throws java.io
{
    int size;
    Scanner scanner = new Scanner(System.in);
    size=scanner.nextInt();
    int[][]a=new int[size][size];
    for(int i=0;i<size;i++)
    {
        int y=0;
        while(y<i){
            int []u = new int[3];
            y++;
            int y_val=0;
            while(y_val<size){
                int y_val = scanner.nextInt();
                y_val++;
            }
        }
    }
    int sum=0;
    int h=0;
    int x_val=0;
    while(x_val<size){
        sum=sum+a[x_val][h];
        x_val++;
        h++;
    }
    int a[2][0];int y_val=0;
    for(int val=size-1;val>=0;val--)
    {
        a2=a2+a[0][y_val];
        y_val++;
    }
    System.out.println(Math.abs(sum-a2));
}
```

Figure 7.2 (C) Image 2 of real world example program
Figure 7.2 (D) Image 2 of real world example to test system
8. **Comparison of the PDG based with other plagiarism detection tools**

There are some popular detection tools such as MOSS and JPLAG that are used for plagiarism checking in coding samples. The tools are explained below.

**Moss( Measure of Software Similarity )**

Reference: http://theory.stanford.edu/~aiken/moss/

Moss is a plagiarism detection tool that was found in Stanford, it was initially invented to detect plagiarism that occurs in programming classes. It can be used for programs written in languages such as Java,C++,C, etc. It compares the program structure of the original program with that of the plagiarized program. It remains robust in most cases where a user may attempt to introduce white spaces, comments and random code insertion.

The next section gives a brief description of the algorithm used by Moss to detect plagiarism

**Algorithm for Moss**

1. Removal of white spaces in code and conversion of characters to lowercase
2. Dividing the remaining characters into continuous substrings of fixed size.
3. Application of hashing on each of these substrings
4. Selection of a subset of these substrings and store them
5. Run comparison on the subset of substrings.

Moss can be used by external users by registering for the Moss service by means of email, once the user is registered he/she is provided perl script. This perl script can be run from the command prompt, upon running we get a http link as a response. This http link gives us the analysis report of plagiarism.

The next section shows testing of a few problems using Moss and results obtained.

**Problem-1**

BubbleSort.java and BubbleSortWhile.java

BubbleSort.java
JPLag is another commercially available tool that is used to test plagiarism. It operates in a variety of platforms. JPlag uses a token-based approach in order to compare two programs. JPlag does not merely compare the text of the programs, but is also aware of the programming language syntax. It supports many programming languages such as Java, C#, C and C++

**Algorithm for JPlag**

1) Convert the program into tokens robust to whitespaces and comments.
2) Try to find a set of substrings present in token sequences in the plagiarized program.
3) If a substring match is found store it and update it, if a longer substring with the same characters of the short one is found in the future.
Using the JPlag service

A jar file namely Jplag-2.11.8-SNAPSHOT.jar was downloaded to run the service. In order to run the service we run the jar file from command prompt by providing the directory to be searched for and the directory to which the results have to be written.

The command

```
java -jar jplag-2.11.8.jar -l java17 -r "C:\Users\sn2638\Desktop\Test_Files_Jplag" -s C:\Users\sn2638\Desktop\Test_Files_Jplag\output_BubbleSort
```

-r: Indicates the jar file to recursively search all folders present in the path for the java file.
-s: Indicates the output directory to which the files have to be written.

We get a HTML file called “index” written to our output directory, where we can find the analysis report. The image given below shows the analysis report.
From the above results of JPlag, the blue section indicates the similarities present between both the programs. We observe that its performance is even more poor when compared to that of Moss. In fact, the above problem it recognizes the main program as being plagiarized, which is not of our interest.

8.1) Analysis and comparison of the results with the current implementation

1) Both Moss and JPlag are token-based systems that do not take into account semantic information, such as data flow and control flow in a program.

2) Both the systems are unable to recognize the change in control structures such as replacement of for-loop with while-loop statement etc.

3) Both the systems seem to have an error if, there are rearrangement of statements involved. For example if two variable declarations in the plagiarized code are made at a
different stage in the program, (as long as the program logic associated with these variables are the same) this may confuse both these tools.

4) However, both these tools are robust to whitespace and comment insertions.

5) Moss seems to be more effective to handle random code insertion when compared to that of JPlag.

<table>
<thead>
<tr>
<th>Plagiarism Disguises</th>
<th>MOSS</th>
<th>JPLAG</th>
<th>PDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Code Insertion</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Alternate Control Structures</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Identifier or variable Renaming</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rearrangement of statements</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table3: Comparison of PDG based technique with other tools

Apart from the above tools, I tried to use other websites such as http://similix.com//, it presents a commercial implementation of the Moss tool known as MOSS-Plus. Unfortunately, we were not able to get an implementation of it to test it.

8.2) (MOSS & JPLAG) VS Current Implementation

The current PDG based implementation is more robust than both MOSS and JPlag, mainly because it does not rely on such token/string based technique, but rather takes into account the data and control flow of the program. The robustness of the system is explained in various scenarios given below.

Random code insertion does not affect the graph constructed as long as it does not alter the logic of the program. It may only make the graph more complex but still the graphs will remain subgraph isomorphic to each other.

Alternate control structures does not affect the graph, as the manner of construction of nodes and edges does not take into account the type of control structure. The control nodes and control edges remain the same even if they belong to a while-loop or for-loop etc.
Identifier or literal renaming cannot change the graph, as the names of identifiers or literals are merely labels given to the graph nodes and they do not affect the structure of the graph.

Rearrangement of statements remains independent of the graph, as they act as mere declaration or assignment nodes that will be created at a different point of time in the plagiarized code, keeping the structure of the graph the same.

The above scenarios indicate that a clear advantage of the current implementation over tools such as MOSS, JPlag, etc. A key feature of the current technique is that it is able to identify similarities in code even if alternate control structures are used. For example, if a while-loop is used instead of for-loop as an attempt to plagiarize, the tool can still find out similar nodes present in the graph.

8.3) A brief description of some other tools used to detect plagiarism

The following section describes a summary of other code plagiarism detection tools that are available and discussed in some of the other research papers that were related to the topic.

<table>
<thead>
<tr>
<th>No</th>
<th>Tool</th>
<th>Languages Supported</th>
<th>Domain</th>
<th>Technique</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dup</td>
<td>c,c++,Java</td>
<td>Unix</td>
<td>Text-based</td>
<td>Academic</td>
</tr>
<tr>
<td>2</td>
<td>CloneDr</td>
<td>c,c++,Java,Text</td>
<td>Windows</td>
<td>Tree-Based</td>
<td>Commercial</td>
</tr>
<tr>
<td>3</td>
<td>DupLoc</td>
<td>Language Independent</td>
<td>VisualWork</td>
<td>String matching</td>
<td>Academic</td>
</tr>
<tr>
<td>4</td>
<td>CP-Miner</td>
<td>c,c++</td>
<td>WindowsNT</td>
<td>Database-Data mining</td>
<td>Academic</td>
</tr>
<tr>
<td>5</td>
<td>SIM</td>
<td>c</td>
<td>Linux</td>
<td>Tree-based</td>
<td>Academic</td>
</tr>
<tr>
<td>6</td>
<td>Coogle</td>
<td>Java</td>
<td>Independent</td>
<td>Tree-based</td>
<td>Academic</td>
</tr>
<tr>
<td>7</td>
<td>PDG-DUP</td>
<td>c,c++</td>
<td>Windows/Unix</td>
<td>PDG/Slicing</td>
<td>Academic</td>
</tr>
<tr>
<td>8</td>
<td>Clones</td>
<td>c,c++,Java,COBOL,VB</td>
<td>Independent</td>
<td>Tree-Based</td>
<td>Academic</td>
</tr>
<tr>
<td>9</td>
<td>Duplix</td>
<td>C</td>
<td>Linux</td>
<td>Tree-Based</td>
<td>Academic</td>
</tr>
<tr>
<td>10</td>
<td>Cpdecltector</td>
<td>C,c++</td>
<td>Linux</td>
<td>AST/suffix tree</td>
<td>Academic</td>
</tr>
</tbody>
</table>

Table4: Summary of some tools used for plagiarism detection
9) Conclusion

Thus the report describes a code plagiarism detection system that is based on construction of graphs, namely Program Dependence Graphs (PDG). The system was tested and verified with programs from real world examples such as CodeChef.com, which is a website that hosts programming contests/challenges for students. The system was also compared with the other state of art plagiarism detection tools such as MOSS and JPLAG. It proved to be a better detector plagiarism than its counterparts.

10) Challenges

A major challenge that was faced during the development of the project was developing an algorithm to construct graphs taking into consideration different coding styles in Java. Secondly, there was a steep learning curve about ANTLR, parsers, lexers and how the compiler recognizes the language. Another major challenge faced was the integration of Neo4j visualizer with the constructed program dependence graphs.

11) Future Improvement of the project

There are certain features that can be incorporated into the graph so that more meaning can be added to PDG. The current implementation does not indicate the invocation of methods present in the language itself, i.e it does not differentiate the user defined method calls and in-built method calls. Also there are no separate nodes created in the graph for the object creation and import statements of libraries, such additional nodes can increase semantic information associated with the graph. We can also develop a mechanism to have a large PDG for the whole program, with smaller PDG’s for individual functions. This kind of a structure would create an edge from one smaller PDG to other smaller ones, to indicate function invocations within the program.

12) References

[1] Philip S.Yu. , Chao Liu, Chen, Jiawei Han, Detection of Software Plagiarism by Program Dependence Graph Analysis. Industrial and Government Application Paper, 2006.


[6] https://jplag.ipd.kit.edu/

