EPlag: A Two Layer Source Code Plagiarism Detection System

Omer Ajmal, M. M. Saad Missen, Tazeen Hashmat, M. Moosa, Tenvir Ali
Dept. of Computer Science & IT
The Islamia University of Bahawalpur
Pakistan
{omer.ajmal, saad.missen, muhammad.moosa, tenvir.ali}@iub.edu.pk, tazeen_hashmat@yahoo.com

ABSTRACT: *In academic environments where students are partly evaluated on the assignments, it is necessary to discourage the practice of copying assignments of other students. The detection of plagiarism in code from large source code repositories, manual detection is fairly complex, if not impossible. Therefore, for fair evaluation there must be a fast, efficient and automated/ semi-automated way to detect the assignments copied. Source Code metrics can be used to detect the source code plagiarism in programming assignments submitted by university students. In this paper we have developed a source code plagiarism detection system and tried to improve the existing techniques by separating the suspected files and the non-plagiarized files, thus reducing the dataset for further comparison. A number of source code metrics have been calculated, combined using similarity detection formula to give an aggregate view of the source code metrics. After that the suspected files are separated and then performed string-matching to detect the level of similarity.*

Keywords: Plagirism Detection, Information Retrieval, Java, Greedy String Tiling

Received: 29 May 2014, Revised 8 July 2014, Accepted 18 July 2014

1. Introduction

Fair Evaluation is a key factor for the success of any educational system. It is vital at all levels especially at higher educational levels. If Fair Evaluation is to be done, the originality of the work must be determined. Students, worldwide, at higher education are found to be involved in some forms of academic dishonesty [6, 7]. Programming courses are no exception. Students plagiarize the programming assignments. They copy code segments from all sources available (internet, books, their class fellows and their seniors), thereby the dataset from which the plagiarism is to be detected increases. In this paper, we will restrict to the two sources: assignments already submitted by the seniors and peers. Manual detection of such plagiarism at large scale is fairly complex rather impossible. There is a need for an efficient technique that detects the source code plagiarism and does it with great performance.

A lot of work has been done on detecting plagiarism in programs. Some techniques involved finding similarity based on source code metrics (code lines, variables declared, total number of statements, total no. of variables, subprograms, input statements, conditional statements, loop statements etc). The systems built on this technique are termed as attribute-counting systems. Other methods include structured metrics techniques and some very successful systems (like *JPlag, MOSS, and YAP* series) have been built using this technique. The underlying technique in such systems includes string matching, which is a costly process. However, both of the techniques suffer from some disadvantages when used in isolation. The attribute-counting

systems are easy to implement but have low performance in detecting plagiarism as compared to the structure metric systems. The structured metric systems which use string matching do not make up efficient systems when the data set is large.

In this paper, we have combined the two techniques so that it radically improves the performance of such systems.

Our work is primarily based on the work done by Andrew Granville in his report submitted at University of Sheffield in May 2002 [14].

Our system works in two stages. In the first step we start with selecting source code file which we call *seed file* (or the original file) from the dataset and rank the Top K "*suspected*" code files from the dataset by using source code metrics techniques. In the second step, we compare the suspected file contents with the *seed file* through a greedy string tiling algorithm. The files, whose contents match to the *seed file* with similarity more than the threshold value (in our case we set it to 50%), are finally said to be plagiarized. The threshold value is set by carefully analyzing the dataset. We will explain the selection of threshold in Section V, Experimentation.

The remaining part of the paper is organized as follows: Section II presents the previous work done in the same domain, Section III describes our approach in detail, Section IV explains the Prototype of the system, Section V presents the experiments, dataset and the results, and Section VI describes the conclusion.

2. Related Work

There has been lot of work on plagiarism detection done by several researchers working in this domain. In this section, we describe some major approaches proposed already.

Most of the approaches involve use of one of the very basic techniques called *attribute-counting. In attribute counting* a program P may be considered as a sequence of tokens classified as either operators or operands: $N1$ is the number of operator occurrences, $N2$ the number of operand occurrences, $n1$ the number of distinct operators and $n2$ the number of distinct operands. In [3] a four-tuple of ($n1$, $n2$, $N1$, $N2$) is assigned to each program such that programs with the same four-tuples are considered suspicious [1]. In later [2]-[5] approaches, the attributes used were enhanced and were made to include code lines, variables declared, total number of statements, total no. of variables, subprograms, input statements, conditional statements, loop statements, call to sub programs and other more sophisticated approaches which includes information such as average number of characters per line, the average identifier length, the number of reserved words, the conditional statement percentage etc. While these attribute counting techniques are easy to implement, they do not make up high performance systems in terms of detecting source code plagiarism, especially when they are compared with the Structured Metric Systems. The structure metric systems rely on the structure of the program and some very successful systems like MOSS, YAP series and JPlag have been built on this technique. A good review of web-based plagiarism detection methods is given in [8][9]. In [10] the author uses Latent Semantic Analysis, an information retrieval technique to extract semantic information and detects similar files from the data collection.

The systems which use attribute-counting or structure metrics suffer from two major limitations:

1) The system require one source/binary code file comparison with many (sometimes thousands or even more) files. Moreover, to find the copied code segments most of the systems use string matching algorithms. The string-matching algorithms (like GST) matches the contents of files; it can easily be imagined the performance of such systems which compare files' contents having thousands of lines of code and hundreds of files.

2) There is a strong need for having clusters of files rather than just having a "*one-to-many*" comparison.

Our system makes contribution to the first type of problems/limitations in the existing systems. We have tried to reduce the dataset on which the string-matching algorithm works. We first reduce the dataset by separating the "suspected" files from those which are not plagiarized. Next we apply the string matching algorithm to the *reduced* dataset. Our approach is presented in the next section in detail.

3. Approach

Our system works in two phases: 1) one-to-many comparison of source code files using the code metrics calculated for each file. This reduces the dataset and thus returns the "*suspected*" files out of the hundreds or even thousands of source code files submitted by the students as a solution to a programming assignment. 2) The contents of the files are compared using our own implementation of the Greedy String Tiling algorithm.

1. Reducing the Dataset using Similarity detection through Source Code Metrics: In our system we calculate source code metrics for each source code file that is submitted as a solution to an assignment. The source code metrics we have used include:

1) **McCabe's Cyclomatic Complexity**: Cyclomatic complexity (also called conditional complexity) is a software complexity measure developed by Thomas J. McCabe in 1976 [11], and it measures the no. of linearly independent paths of a program. The two similar programs will have a similar cyclomatic complexity value. We have used the CyVis implementation to calculate the cyclomatic complexity of our programming assignments.

The remaining code metrics, although not very useful to detect plagiarism when used alone, can give some idea of *copied* files when used with other metrics.

2) **Count of Logical lines**: A logical line of code is termed as a line which is termed as a single statement in a language. In some language (like Java, C++) a statement is terminated by a semi-colon. Some relatively newer languages like Python use line-break and/or semi-colon to separate the statements from one another. We have calculated the total number of logical lines in a program.

3) **Count of Physical lines**: A physical line is a line in the source code. Two physical lines are separated by a line break in a file. One logical line of code may be divided into two or more physical lines in a program and two or more logical lines of code may be combined into a physical line.

4) **Count of Comment lines**: A comment line is a line used for enhancing readability and does NOT become part of the translated (compiled) output file. Different languages use different symbols (//, /**/, #) to separate source code statements from comment lines.

5) **Count of Blank lines**: A blank/empty line is a line which has no text in it.

6) **Count of Keywords**: In this paper we will deal with the Java programs, so the keywords we consider will be restricted to the Java language [12]. A keyword is a reserved word and cannot be used as identifiers in a program.

7) **Count of Fields** (called instance/class variables in JAVA, data members in C++): In object oriented programs, a field is a data member of a class; it may be an instance variable (exists in object's memory) or a static/class variable (exists in shared memory for all instances of a class).

8) **Count of Methods** (also called member functions in C++): A java method is a member function of a class. Like a data member, a method may be instance method or a static/class method.

Note that for code metrics in 2-8 we have our own implementation of calculating these code metrics.

Once the above code metrics are calculated for each of the programs in our dataset, we find the similarity in the files with a simple Euclidean Distance formula [13]. It is a distance between two points and is given by Pythagorean formula. It can be one-dimensional, two-dimensional or N-dimensional. The dimensions will be specified by the attributes you are using for two points. An object with two attributes can be thought of as a point in two-dimensional plane (one for each attribute). Similarly, an object with N-attributes can be thought of as a point in N-dimensional space. The similarity between two points is calculated as their distance in an N-dimensional space. The formula is given by (1).

$$d(p, q) = d(q, p) = \sqrt{\sum_{i=1}^{n} (q_i, p_i)^2} \quad (1)$$

Where $\boldsymbol{p}$ and $\boldsymbol{q}$ are two points in Euclidean n-space $\boldsymbol{q_i}$ and $\boldsymbol{p_i}$ are the axes of the two points (or the attributes of two objects) and $\boldsymbol{d(p, q)}$ and $\boldsymbol{d(q, p)}$ are the distance from $\boldsymbol{p}$ to $\boldsymbol{q}$ and $\boldsymbol{q}$ to $\boldsymbol{p}$ respectively. Note that the two distances are equal. The two objects which are more similar will have a *low* Euclidean distance value and the objects which are more different will have a *large*

Euclidean distance value. In our system if $\boldsymbol{p}$ and $\boldsymbol{q}$ are two source files and $\boldsymbol{p_i}$ and $\boldsymbol{q_i}$ represent the source-code metrics for the two files respectively, the larger value of $\boldsymbol{d}$ means different files and lower values of $\boldsymbol{d}$ means similar files. If the attributes are normalized (values are in the range 0 –1), then similarity = (1 – *distance*). We define a number "*K*" and select "*Top K*" files that are *suspected* to be plagiarized with a particular file (a one-to-many comparison). The larger the value of *K* the more chance is there to include all suspected files for further analysis i.e. string matching through GST. Now we have reduced our dataset to only *K* files rather than hundreds or thousands of files being passed for string matching.

2. String matching through the Greedy String Tiling algorithm: The GST algorithm [15] attempts to compute the degree of similarity between two files of the source code. These will be named the source and target files, where the target is suspected of being plagiarized from the source. The overall method works in two stages. The first one converts the source and target files into token strings. The overall working of GST is given below:

1) **Tokenization**: The contents of the source and target files are divided into tokens. The Tokenizer is made to do a lot of tasks that range from removing comments, strings constants, translating upper case letters into lower case to mapping of synonyms to a common form (e.g., double mapped to float) and like tasks. We limit our tokenizer to do the following:

a) Remove comments

b) Remove string constants

c) Translate upper case to lower case

d) Remove all tokens that are not from the lexicon

2) **Tiling**: A Tile is a one-to-one pairing of a substring from the source file and a substring from the target file [14]. Some important definitions related to GST algorithm are given below:

a) Minimum Match Length: It is a number with potential tiles below this length are ignored. Analyzing our dataset we set it to 3.

b) Maximum Match Length: This is similar to a Tile, but it is a temporary pairing of substrings between the source and target files.

c) The Dice Score Formula: The Dice Score Formula is used to quantify the plagiarism scores. The formula is given by (2), where sFile represents the Source File and tFile represents the Target File.

Our implementation of GST can deal with the following level of changes in the target file that is plagiarized from the source file.

1. Changing comments or formatting

2. Changing identifiers

3. Changing the order of operands in expressions

4. Changing data types

5. Replacing expressions by equivalents

6. Adding redundant statements or variables

7. Changing the structure of selection statements (nested IF and Switch – Case statements)

8. Changing the structure of iteration statements (for, while loops)

3) Quantifying the Similarity: The *diceScore* formula measures the similarity by the fraction of tokens between the source file and target file that are covered by matches. It gives output between 0 and 1, where 0 represents no similarity and 1 represents the equivalent files. Now we show with the following example [14] how it works. Consider the two files with minimum match *length* = 1.

File 1	***int i; static double j;***
File 2	***static double j; int i;***

Table 1. Sample Files for Dice Score Formula

The two matching token sequences found between them are of int *j*, of length 3 and static double *x*, of length 4. Note that the length of files 1 and 2 is 7 each. The diceScore formula can be implemented as follows:

$$diceScore\ (sFile, tFile) = 2 * (3 + 4) / 7 + 7 = 1$$

The above equation shows that the files are identical although the sequence of tokens is different in both files. The details of the GST algorithm may be seen in [14] [15].

4. EPLAG Prototype

We have implemented the above mentioned approach by developing a prototype of the system. We have implemented it for two types of users: Teachers and Students.

1. Teacher can do the following through his/her interface

a) Register into the system

b) Announces assignment for a course

c) Views all submissions for an assignment that are submitted as solution for an assignment

d) Selects a particular solution submitted by a student and views all Top *K* "*suspected*" files through Step 1 (given in Section III-1) of our approach.

e) Views the final results as files plagiarized from a particular source file. The final results are displayed as a list of files sorted with the percent similarity (1 means 100% and so on).

2. Student can do the following through his/her interface

a) Register into the system

b) Selects a particular course and submits solution for an assignment.

5. Experimentation

Our dataset consists of four assignments with 10 submissions for each assignment submitted for the course of Object Oriented Programming using Java taught at The Islamia University of Bahawalpur during Spring 2010. Even our dataset was small but the files on which we have tested were analyzed carefully to and they were found sufficient to uncover system performance. We have selected the following from among many assignments that were given to the students during their course:

1. Binary Search

2. Merge Sort

3. Bubble Sort

4. A small Add and printing algorithm

As we have already mentioned that our system consists of two stages and we have discussed them in detail, now we will see how our dataset works on the system.

Step 1: Finding *suspected* files with similarity calculated through code-metrics: First of all we separated Top *K* (where $K = 5$) submissions. These Top *K* submissions passed to Step 2, where we quantify the similarity using our implementation of GST algorithm.

Step 2: Finding the similarity of Top K files with the *original* file based on GST algorithm: We applied GST algorithm on the Top K files of the Step 1.The files whose similarity is greater than 50% (i.e. the threshold value in our case), we present them as *Plagiarized File*s to the user. The results of the system are given in Table 2. File 1, 2 and 3 represents the submissions against which we calculated the similarity with original file for each assignment. In binary search, the original file was written with all logic in a single method i.e. the main method. File 1 was written with the following logic:

1) Separate methods for getting data from user, displaying it and printing the results.

2) The relational operators were also different from original file (instead of $a < b$ use $b > a$).

3) All identifiers were different from original file.

4) The whole method works with making object of the same class and calling methods for adding data to an array, getting value to search from user, searching value in the array and printing the result.

	Similarity %age with the *Original/Seed* File		
Assignment	**File 1 [Non-Plagiarized File]**	**File 2 [Plagiarized]**	**File 3 [Plagiarized]**
Binary Search	17.78	49.38	-
Merge Sort	**67.95**	57.62	-
Bubble Sort	30.25	82.81	65.30
Simple add and print	45.16	86.26	-

Table 2. Results of EPlag System

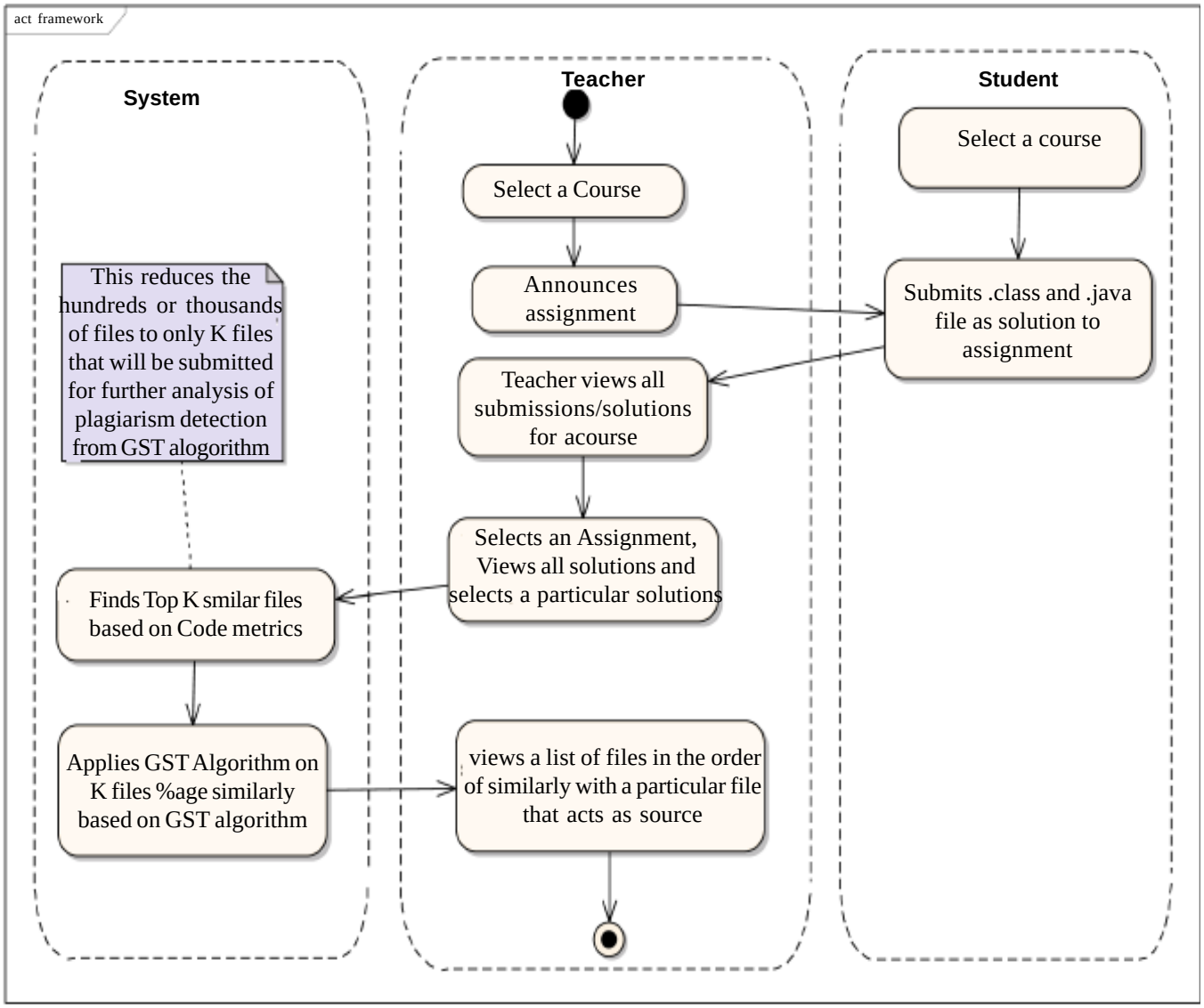

Figure 1. EPlag System Flow

It may be noticed that with such major differences in a small code like Binary Search, the similarity value is only 17%.

File 2 for Binary Search was developed from File 1 but all the code placed in single method again. This time the similarity value approaches 50%.

An exception can be seen in Table 2 where a non-plagiarized file of Merge sort gives the Similarity of 67.95%. We have found that this is due to following reasons:

a. Complexity of the algorithms implemented in the files.

b. Our prototype was not developed to detect all of the levels/types of changes that can be made in target files that are plagiarized from the source file.

c. It is also noticed that two programs that (are NOT plagiarized, but) implement the same algorithm; the similarity for such programs is high.

6. Conclusion

In this work we have proposed a strategy for detecting source code plagiarism. Our strategy works in two phases, at the first phase it selects a *seed* file from the dataset, compares the *seed* file with the rest of the dataset based on code metrics and filters the Top files. These files are then passed to the second phase where their similarity is quantified by the greedy string tiling algorithm. We have shown that by combining code metrics and string-tiling and dividing the system in two stages, it works efficiently as shown by the results. We have also observed that while reducing the dataset through similarity based on code metrics, the Recall of the system can be increased by including more complex code metrics. Moreover, it was also noticed that if the system parameter K is increased there are more files for the processing of the GST algorithm. Our system has one-to-many comparison at both stages/layers i.e. similarity through the code-metrics and the GST algorithm. As part of our future work, we are working on techniques to make the clusters of assignments based on the degree of similarity between submissions that will help identifying study groups in the class.

$$diceScore\ (sFile, tFile) = \frac{2 * \sum_{i \epsilon tiles} (length_i)}{|\ sourceFileSize\ | + |\ targetFileSize\ |} \quad (2)$$

References

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

[2] Grier, S. (1981). A Tool that detects plagiarism in Pascal Programs, *In*: Proceedings of the twelfth SIGCSE technical symposium on Computer science education, p. 15-20, NY.

[3] Donaldson, J. L., Lancaster, A. M., Sposato, P. H. (1981). A Plagiarism Detection System, *In*: Proceedings of the twelfth SIGCSE technical symposium on Computer science education, p. 21-25, NY.

[4] Faidhi, J.A.W., Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment, Computers & Education, 11, p. 11-19.

[5] Allen, F. E., Cocke, J. (1976). A program data flow analysis procedure, Communications of the ACM, 19, p. 137.

[6] Burrows, S. M. M. Tahaghoghi and J. Zobel. (2006). Efficient plagiarism detection for large code repositories, Software Practice and Experience, 37, p. 151-175.

[7] Noh, Seo-Young, Sangwoo Kim, Gaida, S. K. (2004). An XML plagiarism detection model for procedural programming languages. *In*: Proceedings of the 2nd International Conference on Computer Science and its Applications.

[8] Lukashenko, R., Graudina, V., Grundspenkis, J. (2007). Computer-Based Plagiarism Detection Methods and Tools: An Overview, *In*: Proceedings of International Conference on Computer System and Technologies.

[9] Culwin, F., Lancaster, T. (2000). A Review of Electronic Services for Plagiarism Detection in Student submissions, *In*: Proceedings 8th Annual Conference on Teaching of Computing, Edinburgh.

[10] Cosma, G. (2008). An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. PhD Thesis, University of Warwick.

[11] MCCABE, T. J. A Complexity Measure. IEEE Transactions on Software Engineering SE-2 (4) 308 - 320.

[12] Java Language Keywords. from http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html.

[13] Deza, E. D. M., Michel. (2009). Encyclopedia of Distances, Springer: 94.

[14] Granville, A. (2002). Detecting Plagiarism in Java Code. Bachelor of Engineering with Honours in Software Engineering, Thesis, The University of Sheffield.

[15] Wise, Michael J. (1993). String similarity via greedy string tiling and running Karp-Rabin matching. Online Preprint, Dec.