Similarity Detection in Java Programming Assignments

Mohamed El Bachir Menai, Nailah Salah Al-Hassoun
Department of Computer Science
CCIS - King Saud University
P.O. Box 51178, Riyadh 11543, Saudi Arabia
menai@ksu.edu.sa

Abstract— **Similarity detection tools are nowadays commonly used by instructors to prevent student cheating and to enforce academic integrity. Systems identifying similarity in programming assignments are generally classified as either attribute-based or structure-based systems. Attribute-based methods make statistical analysis of the program attributes to detect lexical changes. Whereas structure-based methods complete a deeper analysis of the program structure to detect hidden structural similarities. Both methods can be useful for student programming assignments which consist generally of small to medium size source codes. In this paper, we introduce a method that encompasses both approaches to fit characteristics of student Java programming assignments. Similarities between pairs of programs can be detected by either profiling their source codes and measuring their distance or parsing them and comparing their encodings using a method inspired by DNA sequencing. We describe our experimental prototype, called CAPlag (Computing Assignment Plagiarism), and illustrate the results of some exploratory experiments. We demonstrate that our method is able to accurately find similarities in Java programs by comparing our results against those obtained with JPlag, a Web based service, and show that our system can be useful for instructors to deal with different programming assignment cases.**

Keywords- Similarity detection, Programming assignment, Java, Attribute-based method, Structure-based method

I. INTRODUCTION

Plagiarism is an attempt to pass off someone's work, in a whole or part, as his/her own work without giving credit [26, 22]. Most frequent cases appear among university students who copy materials from different sources (journals, books, peer course work, etc.) without citing references. Program source code can be particularly reproduced easily by including a small number of changes without a need to a detailed understanding. This means that with a few simple editor operations it is possible to produce a plagiarized program with a very different visual appearance [3, 7].

Changes in program code fall into two main categories: lexical and structural changes. Lexical changes require little knowledge of the programming language and generally there is no need to program parsing. These changes might include rewording, adding or deleting comments, changing formatting and changing variable names [16]. Structural changes require knowledge of the programming language to be able to change the structure without altering the program significance. This is highly language dependent and might include replacement of equivalent iteration structures or operand ordering [16]. Faidhi and Robinson [12] characterize six levels of program modification in a plagiarism spectrum. Level 0 is the original program without modifications. In level 1, only comments and indentation are changed. In level 2, identifier names are changed. In level 3, positions of variables, constants, and procedures are modified. In level 4, some combinations and separations are introduced to the procedures. In level 5, program statements are changed. In level 6, loop control structures are changed to an equivalent form using different control structures [26, 4, 32].

While similarity can be detected manually for isolated cases, it is often made automatically. Manual detection methods are often costly in time and effort especially as class sizes and assignments length increase. Manual detection is difficult but it could be efficient to assess the plagiarism case once an automatic tool has been used [11, 26, 24]. Indeed, the usage of automatic methods of detection aids the manual inspection of suspect assignments by reducing the effort required in comparing a large number of assignments.

There are three different classes of similarity detection methods: quiz methods, writing style methods, and comparison methods with original sources [18, 5, 26]. In this paper, we focus on the last class of methods, since an instructor compares student's programming assignments against a collection of other works.

The rest of the paper is organized as follows. In Section 2, we present existing comparison methods and existing program-code similarity detection systems. In Section 3, we describe the proposed system, CAPlag, and discuss its details. In Sections 4, we present some experimental results including comparative results of CAPlag and JPlag. We conclude and discuss future work in Section 5.

II. RELATED WORK

A. Existing methods

Comparison methods can be roughly grouped into two categories: Text-based methods and code-based methods. The most used text-based methods include fingerprinting [11, 5, 28], Greedy String Tiling (GST) [31, 15, 21], and Running Karp-Rabin Matching and Greedy String Tiling (RKR-GST) [24, 31]. RKR-GST appears to be the principle algorithm that is used in most commercial plagiarism detection systems. It is a string searching algorithm that uses hashing to find the longest possible string common to two documents.

Code-based methods fall in two categories: Attribute-based and structure-base methods [3, 5]. Attribute-based methods consist of extracting various metrics which capture a simple quantitative analysis of some program attributes, such as the number of tokens, distribution of identifiers, and other authors or program specific characteristics, such as the usage of a particular reserved word. The earliest attribute-counting metric systems used Halstead's software metrics [14] to measure the level of similarity between programs. These methods are easy to implement and to use. However, they are not very effective, since it is difficult to select rationally a set of metrics to profile a program.

Structure-based methods [5, 25, 15, 30] have been introduced to capture the logical structure of a program. They consist of comparing string representations of the programs, rather than comparing measures extracted from their structure. Programs are typically converted into sequences of tokens (string representation) using a language-dependent parser. This process of normalization is used to reduce the effect of differences due to systematic changes, such as renaming identifiers, and to characterize the essence of a program's structure (which is difficult to change by a plagiarist). Structure-based methods give an improved measure of similarity. They have been shown to be more effective than attribute-based methods in similarity detection [3, 5]. Recent research tends to focus on efficient program encodings into a normalized stream of tokens rather than finding additional comparison methods.

B. Existing systems

This section presents some of the most popular similarity detection tools for source code. The first tool was developed by Ottenstein [25] to detect similarities in FORTRAN programs based on Halstead's metrics [14]. Most of modern tools implement structure-based methods [4, 15].

1) MOSS (Measure Of Software Similarity)

MOSS is a free online similarity detection tool (available at www.cs.berkeley.edu/~aiken/moss.html). MOSS was developed in 1994 by Alex Aiken at UC Berkeley. It works with programs written in C, C++, Java, Pascal, ML, Lisp, Ada, or Scheme [1, 28]. Neither information nor test results are provided about the algorithm except what is mentioned on MOSS Web site: "… more sophisticated than systems based on counting the frequency of certain words in the program text".

2) JPlag

JPlag is an online similarity detection tool (available at http://www.jplag.de) developed by Guido Malpohl in 1996 at the University of Karlsruhe. JPlag finds similarities between pairs of programs written in Java, C, C++, Scheme, and free text. It uses Greedy String Tiling comparison algorithm [31], adds different optimizations for improving its run time efficiency [21], and provides a powerful graphical interface. However, no test results are published.

3) CodeMatch

CodeMatch is a commercial similarity detection tool produced as part of the CodeSuite software (available at http://www.safe-corp.biz/products_codematch.htm). CodeMatch compares thousands of source code files in multiple directories and subdirectories to determine which files are the most highly correlated. It works with different programming languages, such as C, C++, C#, Delphi, Java, JavaScript, and Pascal. CodeMatch uses several string matching algorithm to determine similarity between two source code files.

4) CPD (Copy/Paste Detector)

The PMD open source tool (http://sourceforge.net /projects/pmd/) provides a Copy/Paste Detector (CPD) tool for finding duplicate code. CPD [6] uses the Karp-Rabin string matching algorithm. It works with Java, JSP, C, C++, Fortan and PHP code. It provides guidance on how to add other programming languages to the tool. This tool scans the files themselves for duplicate code, also it is successful in returning similar code across different files.

III. PROPOSED SYSTEM

Source code implemented by Bachelor students are characterized by several common traits, since they are generally related to the same programming assignments. In particular, those related to the first graduate levels (first programming courses) could not be checked correctly using a structure-based tool, as their global structures are almost the same. A system that provides tools to compare program profiles and/or program structures might help instructors to detect lexical and structural changes. Moreover, such a system might limit the number of false positives and balance the tradeoff between the speed and the reliability of the algorithms used. Indeed, for short programming assignments, it might be useless to compare source code structures.

CAPlag (Computing Assignment Plagiarism) is the similarity detection system we propose for Java programs, based around a two phase-algorithm. The first phase consists of a fast screening process that determines program profiles and compares them. The second phase could be useful to detect more intricate plagiarism cases by parsing the programs and comparing them.

A. First phase: Attribute-based comparison

A program profile is represented by a set of features extracted from its source code according to different software metrics [13, 20]. Specific metrics have been

introduced for different programming languages, such as Java [9], C [17], and C++ [20]. Three classes of software metrics for Java source code have been statistically derived for authorship identification [9]: programming layout metrics (STY), programming style metrics (PRO), and programming structure metrics (PSM). The layout metrics have been shown to play a more important role in the classifications than the style and structure metrics [9]. Our study of software metrics leads us to consider those having the highest impact on the similarity measure [26, 19, 23, 9]. Table 1 gives a description of the program metrics used in CAPlag.

We measure the similarity between two programs *A* and *B* by the Weighted Mean (*WM*) of each class of metrics. Given a program *P* and *n* layout metrics, *m* style metrics, and *p* structure metrics, the weighted means for each class of metrics are defined by the equations (1), (2), and (3).

$$WM_{Layout}(P) = \sum_{i=1}^{n} w_i \cdot STY_i \Big/ \sum_{i=1}^{n} w_i \qquad (1)$$

$$WM_{Style}(P) = \sum_{i=1}^{m} w_i \cdot PRO_i \Big/ \sum_{i=1}^{m} w_i \qquad (2)$$

$$WM_{Structure}(P) = \sum_{i=1}^{p} w_i \cdot PSM_i \Big/ \sum_{i=1}^{p} w_i \qquad (3)$$

The match percentage between *A* and *B*, *M* (*A, B*), can be evaluated for every *WM* as follows.

- Match according to programming layout:

$$M_{Layout}(A,B) = \left(1 - \frac{\left|WM_{Layout}(A) - WM_{Layout}(B)\right|}{Max(WM_{Layout}(A), WM_{Layout}(B))}\right) \times 100 \qquad (4)$$

- Match according to programming style:

$$M_{Style}(A,B) = \left(1 - \frac{\left|WM_{Style}(A) - WM_{Style}(B)\right|}{Max(WM_{Style}(A), WM_{Style}(B))}\right) \times 100 \qquad (5)$$

- Match according to programming structure:

$$M_{Structure}(A,B) = \left(1 - \frac{\left|WM_{Structure}(A) - WM_{Structure}(B)\right|}{Max(WM_{Structure}(A), WM_{Structure}(B))}\right) \times 100 \qquad (6)$$

Overall, it can be estimated by *M* (*A*, *B*):

$$M(A,B) = \left(1 - \frac{\left|WM(A) - WM(B)\right|}{Max(WM(A), WM(B))}\right) \times 100 \qquad (7)$$

TABLE I. PROGRAM METRICS USED IN CAPLAG [9]

Program Measure (metric)		
Metric	***Description***	***Weight***
STY1c	Percentage of open braces ({) that are the last character in a line	0.39
STY1e	Percentage of close braces (}) that are the first character in a line	0.41
STY1f	Percentage of close braces (}) that are the last character in a line	0.29
STY1g	Average indentation in white spaces after open braces ({)	0.25
STY1h	Average indentation in tabs after open braces ({)	0.4
STY2a	Percentages of pure comment lines among lines containing comments	0.39
STY2b	Percentages of End Of Line style comments among End Of Line and Traditional style comments ‘‘//’’ End Of Line style comment and ‘‘/*’’ Traditional style comment	0.23
STY4	Average white spaces to the left side of operators: One of (= > < ! ~ ? : == <= >= != && || ++ - + - & | ^ % << >> >>> += -= &= |= ^= %= <<= >>= >>>=)	0.38
STY5	Average white spaces to the right side of operators	0.4
PRO1	Mean program line length in terms of characters	0.14
PRO2b	Mean function name length	0.22
PSM5	Ratio of primitive variable count to lines of non-comment code Primitive variable : One of (Int, Long, Float, Double, Boolean, Char)	0.32
PSM6	Ratio of function count to lines of non-comment code	0.03
PSM7c	Ratio of keyword ‘‘class’’ to lines of non-comment code	0.1
PSM7e	Ratio of keyword ‘‘implements’’ to lines of non-comment code	0.38
PSM7j	Ratio of keyword ‘‘new’’ to lines of non-comment code	0.2
PSM7l	Ratio of keyword ‘‘private’’ to lines of non-comment code	0.27

where *WM* (*P*) is defined by the equation (8).

$$WM(P) = \frac{\sum_{i=1}^{m+n+p} w_i \cdot metric_i}{\sum_{i=1}^{m+n+p} w_i} \qquad (8)$$

B. Second phase: Structure-based comparison

The structure-based comparison in CAPlag consists of a normalizing and alignment processes. First, a parser generates a string representing the structure of the input program code. Next, program string encodings are aligned using a DNA local alignment method [8, 27].

Similarity has both quantitative and qualitative aspects. A similarity measure gives a quantitative answer, saying that two sequences show a certain degree of similarity. An alignment is a mutual arrangement of two sequences which is a sort of qualitative answer it exhibits where the two

sequences are similar and where they differ. Optimal alignment is evaluated in CAPlag using dynamic programming alignment [2, 10]. An optimal alignment is the one with the maximum number of matches and minimum number of mismatches and gaps between the two sequences.

There are two types of dynamic programming algorithms: Global and local. Global sequence alignment compares two sequences throughout their lengths. This is clearly not the case when comparing a program sequence against an entire program [27].

In CAPlag, we use the Smith-Watermann local sequence alignment algorithm [29]. It computes the best score and finds the highest possible scoring substrings. This algorithm is described as follows.

1. Initialization:
   The two normalized strings are assigned to variables, A and B and their lengths to i and j.
2. Create a Scoring Matrix:
   A matrix V of dimensions (i+1) by (j+1) is created to save the scores, using the following scoring matrix:
   a. Perfectly matched get a high score $w(match) = 1$
   b. Matches between related get a modest score $w(mismatch) = -1$
   c. Matches with gaps get a low score $w(gap) = -2$. Finding the local alignments of two sequences starts from the highest score of a block until zero.

$$V[i+1.j+1] = \max\begin{cases} V[i,j] + match \text{ or } mismatch \\ \mathrm{V}[i,j+1] + gap \\ V[i+1,j] + gap \\ 0 \end{cases} \quad (9)$$

3. Finding the optimal alignments of two sequences. The score between two sequences is the maximum score among all alignments.
4. Calculate similarity measure. A similarity measure (%) between two sequences A and B is defined as follows (normalized formula):

$$M(A,B) = \left(2 \cdot \frac{\text{score}(A,B)}{(\text{score}(A,A) + \text{score}(B,B))}\right) \times 100 \quad (10)$$

Figure 1 shows a block diagram of our algorithm.

IV. Experimental Evaluation

CAPlag was implemented in Java programming language and tested on more than 200 Java code source programs of various sizes grouped in 3 sets. They consist of a dummy test set created by hand, with modified levels in plagiarism spectrum (level 1 to level 6; level 0 represents the original program).

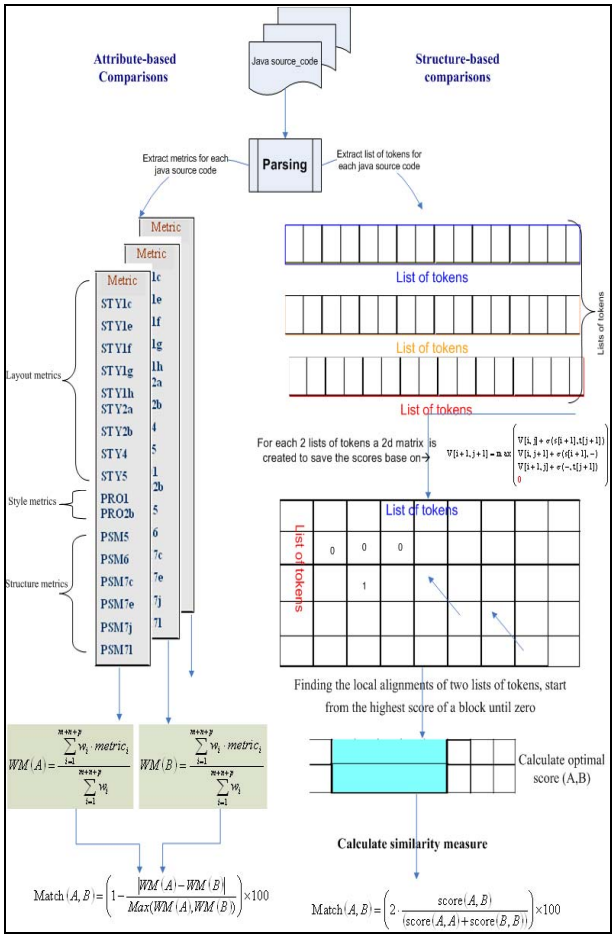

Figure 1. Block diagram of CAPlag algorithm

Program sizes range from small to large: small (average number of lines without comments: 10-50); medium (average number of lines without comments: 51-150); large (average number of lines without comments: 151-300).

Figures 2-4 present the results obtained in terms of average similarity percentage for each level of plagiarism when attribute-based or structure-based comparisons are used. Overall, the structure-based comparison method outperforms the attribute-counting one. Both methods return high similarity percentage for the plagiarism levels 1-2. Moreover, their performances for the other plagiarism levels are closely related on small programs.

CAPlag was compared against JPlag on the same test sets. Figure 5 illustrates the results obtained. JPlag outperforms CAPlag on medium and large programs. However, the performance of CAPlag with the structure-based comparison method remains comparable to JPlag's performance. Average results obtained with CAPlag on small programs with attribute-based or structure-based comparison methods are much better than those obtained with JPlag.

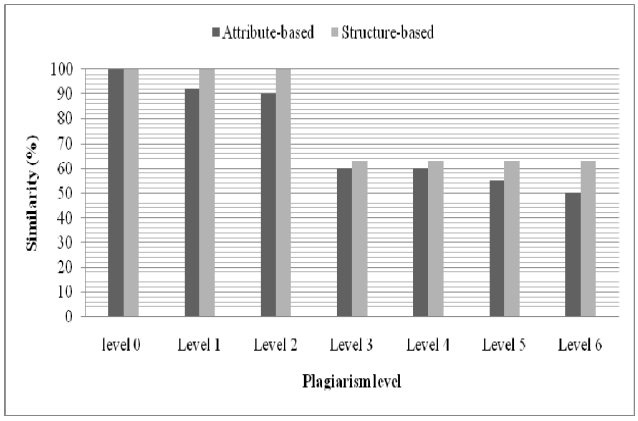

Figure 2. Histogram of similarity % (y-axis) against plagiarism levels (x-axis) for small Java programs.

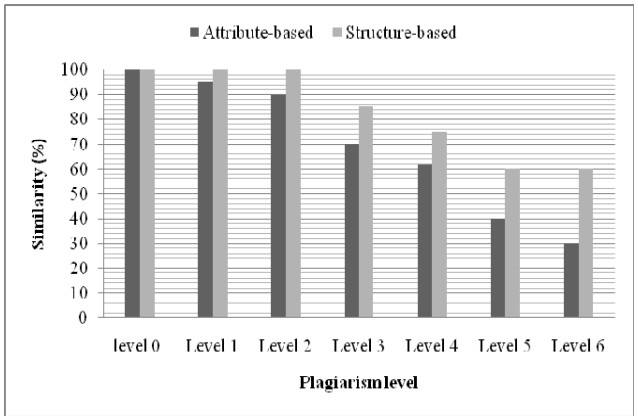

Figure 3. Histogram of similarity % (y-axis) against plagiarism levels (x-axis) for medium Java programs.

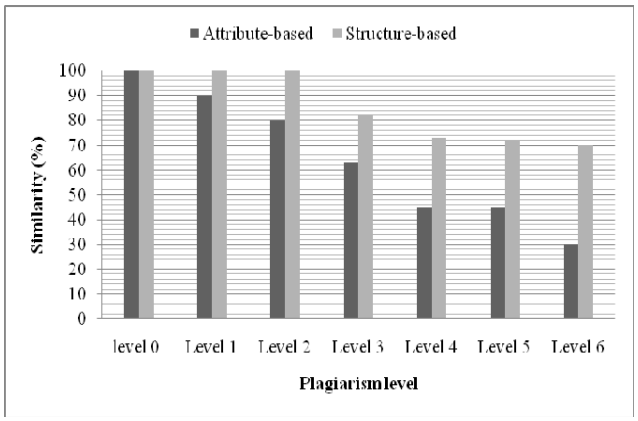

Figure 4. Histogram of similarity % (y-axis) against plagiarism levels (x-axis) for large Java programs.

V. Conclusion and Future Work

We introduced a new similarity detection system (CAPlag) allowing both attribute-based and structure-based comparisons of Java source codes.. Attribute-based comparison consists of programs' profiling and comparing their software metric features. Structure-based comparison is based on programs' normalization and alignment using a dynamic programming alignment method.

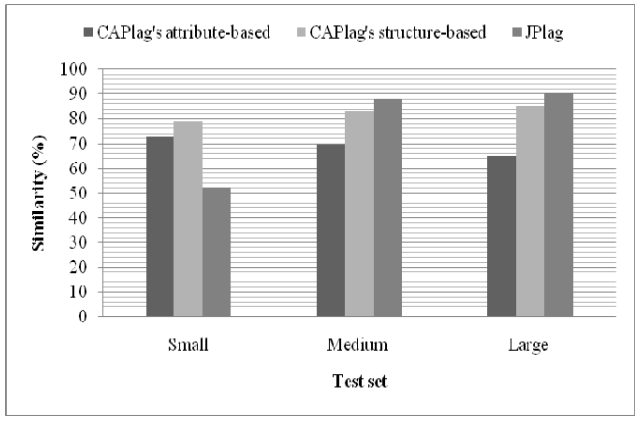

Figure 5. Histogram of similarity % (y-axis) against test set size (x-axis) comparing CAPlag with JPlag.

Experimental results demonstrate the effectiveness of CAPlag. Its comparison against JPlag proves its competitiveness. Moreover, we show that attribute-based comparison method can be useful for student programming assignments, even if it is not regarded in modern plagiarism detection systems. Indeed, programming assignments of the first graduate levels are generally characterized by their small size, and a fast shallow parsing might be sufficient to detect similarities in the source codes. Structure-based comparison method is suitable for larger programs. CAPlag can be used by instructors to detect lexical and structural similarities in their students' programming assignments through an interactive and easy-to-use graphical interface.

The performance of CAPlag could be improved by expanding the set of metrics used for program profiling. Indeed, a statistical analysis of student programming assignments might lead to derive specific metrics. CAPlag could be also easily extended to handle other programming languages.

References

[1] W. K. Bowyer and O. L. Hall, "Experience Using 'MOSS' to Detect Cheating On Programming Assignments," In Proc. 29th ASEE/IEEE Frontiers in Education Conference, pp.18 - 22, 1999.

[2] G. M. Cannarozzi, "String alignment using dynamic programming", 2005. http://biorecipes.com/DynProgBasic/code.html

[3] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker, "Shared Information and Program Plagiarism Detection," IEEE Transactions on Information Theory, vol.50, no. 7, pp. 1545-1551, 2004.

[4] P. Clough, "Plagiarism in Natural and Programming Languages: An Overview of Current Tools and Technologies," Research Memoranda, Department of Computer Science, University of Sheffield, UK, Tech. Rep. CS-00-05, 2000.

[5] P. Clough, "Old and new challenges in automatic plagiarism detection," Plagiarism Advisory Service, Department of Information Studies, University of Sheffield, UK, 2003.

[6] T. Copeland, "Detecting Duplicate Code with PMD's CPD," OnJava, March 2003. http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.htm

[7] F. Culwin, A. MacLeod, and T. Lancaster, "Source Code Plagiarism in UK HE Computing Schools, Attitudes and Tools, " South Bank University, London, Tech. Rep. (SBU-CISM-01-01), 2001.

[8] D. Das and D. Dey, "A New Algorithm for Local Alignment in DNA Sequencing," In Proc. of the IEEE INDICON 2004, vol.1, no. 20-22, pp. 410 – 413, 2004.

[9] H. Ding and M. H. Samadzadeh, "Extraction of Java program fingerprints for software authorship identification," Journal of Systems and Software, vol.72, no.1, pp.49-57, June 2004.

[10] S. R. Eddy, "What is dynamic programming?," Nature BioTechnology, vol. 22, no. 7, 2004.

[11] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism Detection Using Feature-Based Neural Networks," Artificial intelligence, vol. 39, no. 1, pp. 34 -38, 2007.

[12] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," Comput. Educ., vol. 11, pp. 11-19, 1987.

[13] A.Gray, P. Sallis, S. MacDonell, "IDENTIFIED (integrated dictionary-based extraction of non-language-dependent token information for forensic identification, examination, and discrimination): a dictionary-based system for extracting source code metrics for software forensics," In Proc. of Software Engineering: Education & Practice Conf., pp. 252–259, 1998.

[14] M. H. Halstead. Elements of Software Science. New York: Elsevier,1977.

[15] M. Hoffmann, "The Plagiarism Detector COPY-D-TEC," Department of Computer Science, University of Stellenbosch, South Africa, Tech. Rep. Final Project Report, 2004.

[16] M. S. Joy and M. Luck, "Plagiarism in programming assignments," IEEE Transactions on Education, vol. 42, no. 2, pp.129-133, 1999.

[17] I. Krsul, E.H. Spafford, "Authorship Analysis: Identifying the Author of a Program," Purdue University, West Lafayette, IN, Tech. Rep. (CSD-TR-96-052), 1996.

[18] T. Lancaster and F. Culwin, "Classifications of Plagiarism Detection Engines," ITALICS, vol. 4 , no.2, 2005.

[19] H. F. Li, "An Empirical Study of Software Metrics," IEEE Transactions On Software Engineering, vol. 13, no. 6, pp. 697-708, 1987.

[20] S. G. McDonell, A. R. Gray, G. McLennan, and P. J. Sallis, "Software forensics for discriminating between program authors using case based reasoning, feed forward neural networks, and multiple discriminate analysis," In Proc. of the 6th Inter. Conf. on Neural Information, vol. 1, pp. 66–71, 1999.

[21] G. Malpohl, L. Prechelt, and M. Phlippsen, "Finding plagiarisms among a set of programs with JPlag," Journal of Universal Computer Science, vol. 8, no. 11, pp. 1016-1038, 2002.

[22] H. Maurer, F. Kappe, and B. Zaka, "Plagiarism - A Survey," Journal of Universal Computer Science, vol. 12, no. 8, pp. 1050-1084, 2006.

[23] E. Merlo, "Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity," in Proc. Of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, Dagstuhl, Germany, 10pp, 2007.

[24] E. Noynaert, "Plagiarism Detection Software," In Proc. Of 38th Annual Midwest Instruction and Computing Symposium, Eau Claire, Wisconsin, 2005.

[25] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," SIGCSE Bull., vol. 8, no. 4, pp. 30-41, 1976.

[26] A. Parker and J. Hamblen, "Computer Algorithms for Plagiarism Detection," IEEE Transactions on Education, vol. 32, no. 2, pp. 94–99, 1989.

[27] E.C. Rouchka, "Aligning DNA sequences using dynamic programming," ACM Crossroads, vol.13, no.1, pp.18-22, 2006.

[28] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," In Proc. of the ACM SIGMOD Inter. Conf. on Management of Data, pp. 76-85, 2003.

[29] T. F. Smith, and M. S. Waterman, "Identification of Common Molecular Subsequences," Journal of Molecular Biology, vol. 147, pp. 195–197, 1981.

[30] G. Whale, "Identification of program similarity in large populations," The Computer Journal, vol. 33, no. 2, pp.140–146 , 1990.

[31] M. J. Wise , "String similarity via greedy string tiling and running Karp-Rabin matching," ftp://ftp.cs.su.oz.au/ michaelw/doc/RKR GST.ps, 1993.

[32] L. Zhang, Y. Zhang, and Z. Yuan, "A Program Plagiarism Detection Model Based on Information Distance and Clustering," In Proc. of Intelligent Pervasive Computing, pp. 431-436, 2007.