ISSN No. 0976-5697

Volume 3, No. 3, May-June 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Abstract Syntax Trees with Latent Semantic Indexing for Source Code Plagiarism Detection

Resmi N.G.*
Department of Computer Science and Engineering
Sahrdaya College of Engineering and Technology
Thrissur, India
resming@sahrdaya.ac.in

K.P. Soman
Centre for Excellence in Computational Engineering and Networking, Amrita Vishwa Vidyapeetham
Coimbatore, India
kp_soman@amrita.edu

Abstract: In this paper, we study and analyze the results of combining two source code plagiarism detection approaches by making some modifications as compared to the existing systems to detect source code plagiarism in academic field. Structure based techniques which have increased efficiency in detecting similarity compared to software metric based techniques are generally computationally complex. Here, we combine an attribute-metric based detection approach - Latent Semantic Indexing (LSI), with a structure based approach - Abstract Syntax Tree (AST) comparison. LSI is first used for identifying a set of potentially plagiarized programs which are further tested for similarities by comparing their abstract syntax trees. Use of LSI for screening reduces the computational cost involved in tree generation and comparison. Moreover, we have modified the preprocessing stage of LSI and have added a post processing stage for improved performance. Our method was tested for C, C++ and Java source code files. Both the approaches were initially tested individually for a collection of student programs of varying functionality and size. These were then combined and found to give better results than executing independently. The performances are evaluated by calculating the precision and recall.

Keywords: abstract syntax trees; latent semantic indexing; plagiarism detection; singular value decomposition

I. INTRODUCTION

Source code plagiarism occurs when source code is copied and edited without proper acknowledgement of the original author [1]. Plagiarism can be defined in many ways by identifying the causes, sources, and types of plagiarism in written text as well as in programming languages [1],[2],[3]. Plagiarism of any type is always considered a serious problem and has to be detected. Since the task of manually detecting plagiarism in a large database of programs is very tedious and time-consuming, efforts are continuously being made to automate the process. An analysis of methods used by a number of tools currently available to detect source code plagiarism is done in [1],[2],[3]. An earlier study shows that systems which gather details of program structure are more effective in detecting plagiarism than those that employs attribute counting mechanism [4].

The first algorithm for plagiarism detection, by Ottenstein [5], was based on attribute counting using Halstead's software science metrics [6]. Vector space model is used for automatic indexing of text documents in [7], and it facilitates the retrieval of documents matching a user query or in identification of similar documents. It considers each document as a vector in a document space and similarity between two documents is given by their dot product. Documents with high similarity tend to cluster together and document space consists of several such clusters. A user query might return all documents belonging to a particular cluster. Information retrieval using Latent Semantic Indexing (LSI), [8] also treats documents as vectors and applies term weighting but can automatically identify the latent semantic structure of the data through Singular Value Decomposition (SVD). This method too computes cosine similarity and is more efficient than raw term matching method used for information retrieval. In [1], Cosma uses LSI for detecting similarity in source code files. Section 2 explains this method in more detail and we also discuss how our approach differs from that adapted by Cosma.

In [9], Baker uses a lexical analyzer to generate a parameterized string for which a compact representation of trie, a parameterized suffix tree, is constructed to identify clones in large software systems. Parameterized match is detected when one part of the code differs from the other only by a change in parameter names. A parser is used to construct Abstract Syntax Trees (ASTs) and find the source code clones in [10]. Each subtree is assigned a hash value and those subtrees with the same hash value, grouped into the same set, are compared to detect tree matches. Even though an existing parser is modified to produce ASTs, and the number of subtree-pair comparisons is reduced through hashing, the time required is still high. In [11], the authors describe a clone detection method which uses abstract syntax suffix trees but avoids direct tree comparison. A source code file is first parsed and its AST is produced on which a preorder traversal is performed to obtain a sequence of AST node types. A string based algorithm then replaces the parameterized string matching used in [9]. The use of parsers in source code similarity detection makes the system highly language dependent and hence, less scalable.

Moreover, parser-dependent techniques allow the users to check only those program files which are free of compilation errors. Another line-by-line comparison approach described in [12] uses hashing and string matching after a simple preprocessing of source files and is language independent. JPlag, a tool to detect code plagiarism, converts each source program to a string of tokens after parsing [13] and then uses greedy string tiling [14] to detect token matches.

Our main objective in this paper is to study and analyze the results of combining two source code plagiarism detection approaches by making some modifications as compared to the existing systems to detect source code plagiarism in academic field. We executed both the methods independently and in a combined manner. The results of our experiments are discussed in section 3.

Rest of the paper is organized as follows: Section 2 briefly describes how the two plagiarism detection approaches – Latent Semantic Indexing and Abstract Syntax Tree comparison, were modified for use in our system. It also discusses the method we adapted by combining these two approaches. Results and discussions are given in Section 3. Section 4 discusses the possible extensions and future scope of our work.

II. METHODS USED

A. Latent Semantic Indexing:

Latent Semantic Indexing is an established technique in the field of information retrieval [8] and document classification. It is used in web search engines to retrieve documents that match a user query. Automatic indexing for text documents first uses a simple word analyzer to identify the words in a document. It discards words which are used very often, as they are irrelevant in identifying a particular document. The list of relevant words which is used for indexing is formed out of a sample document corpus and a term document matrix is constructed with normalized term frequencies. Term frequency (tf) for the ith term in jth document is given by

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \qquad (1)$$

Where $n_{i,j}$ is the number of times the term i occurred in document j and denominator gives the total number of occurrences of all the terms in document j. The inverse document frequency (idf) is computed for each term to know its significance among a set of files. It is given by

$$idf_i = \log \frac{|D|}{|\{d : t_i \in d\}|} \qquad (2)$$

Where D is the total number of files to be compared and d is the number of files in which the term i has occurred. tf-idf transform is then computed as

$$(tf - idf)_{i,j} = tf_{i,j} \times idf_i \qquad (3)$$

This matrix, say A, is decomposed into three matrices using Singular Value Decomposition (SVD) as

$$A = USV^T \qquad (4)$$

Where U is an orthonormal matrix with columns as eigenvectors of AA^T, S is a diagonal matrix with square roots of eigenvalues of AA^T or A^TA in descending order, and V is an orthonormal matrix with eigenvectors of A^TA. Term-term similarity, document-document similarity and term-document association can be obtained using these matrices. By retaining only columns corresponding to k largest singular values, we can also achieve dimensionality reduction.

B. Modified LSI for Source Code Plagiarism Detection:

The general approach for text comparison takes into account all the words in a document, except for a short list of stop-words. Our approach is designed to detect similarity between source code files for a specific programming language. It takes into account only the keywords specific to a particular language. Each language has its own set of keywords. Only this small set of words along with the set of operators is considered while creating the term-document matrix. This greatly reduces the number of terms in the matrix, thereby making the matrix compact and hence, reduces the computational cost involved in the decomposition of a huge matrix.

In our analysis, we have considered 3 widely used programming languages in the academic field - C, C++ and Java. Separate token files are maintained for each language taking into account their language features.

C. Abstract Syntax Trees for Source Code Plagiarism Detection:

Abstract Syntax Tree (AST) is an intermediate representation of the source code. A parser generator is required to produce ASTs. The trees generated by parser generators are called parse trees and are usually huge in size. These trees can be reduced in size by making suitable modifications in the parser definition [15] for a specific language to remove redundant nodes which do not add any extra information to the program structure. This reduced tree will contain only those nodes which carry useful information and hence the name abstract syntax tree. LSI treats source code file just as a collection of words and cannot keep track of the structure information. This limitation is overcome by the use of ASTs.

Each source code file is parsed and its AST is generated. Once the ASTs are generated, comparison of ASTs can be done in different ways. One simple way is to compare the ASTs node by node. However, this method is not very efficient since such an algorithm halts whenever it encounters two nodes with different labels. There is hardly any meaning in this approach if root nodes of the ASTs have immediate children with different labels. Another method is to partition the ASTs into subtrees and compare each subtree in one AST with each subtree in another AST. However, this requires huge amount of time as well as space. A better approach employing hash functions is used by the authors in [10] as already discussed in the introduction section.

In [15], Ligaarden proposes an AST based approach to detect plagiarism in Java source code, which we have modified for C and C++ source code files. The author modifies the parse tree generated using open source scanner and parser generator JavaCC and tree builder JJTree to obtain the corresponding AST. A preorder traversal is done through the ASTs to be compared as done in [11] to generate node sequences. Sequence matching algorithms- Top Down Unordered Maximum Common Subtree Isomorphism, Needleman-Wunsch(NW) algorithm and Longest Common Subsequence(LCS) algorithm, are then used to compare the node sequences and find matches. Top down unordered maximum common subtree isomorphism is used for the tree as a whole. Since it does an unordered matching, it can find good match between independent structures. However, it fails to find good match between the statements and local variable declarations of two blocks.

This is solved by using NW algorithm. It also finds low similarity between the trees of different loops and different selection statements. This is solved by using LCS algorithm. Maximum weight bipartite matching algorithm is used to find the size of maximum common subtree of two tree structures. This approach proved to be very efficient in terms of similarity detection, but for a huge program database the runtime was found to be very high. Hence, we use an LSI based approach to reduce the number of programs given as input to the AST algorithm.

D. Combining AST Based Approach and LSI Based Approach:

LSI algorithm can identify sets of potentially plagiarized programs in a large program database but is incapable of identifying which portions of the programs tend to be similar. Moreover, attribute-based detection is less accurate for large program files and larger number of files. Hence, we use LSI only for extracting groups of possibly plagiarized files. To improve the overall performance of the AST-LSI combined approach, we have modified the preprocessing stage of LSI and have added a post processing stage to LSI.

a. Preprocessing Stage:

The preprocessing of files occurs during the lexical analysis phase of LSI. In [1], Cosma removes comments, Java reserved words, terms occurring in a single file or all files, and single character tokens during preprocessing. A different method is followed in [16] which takes into account only the identifier names (variable names and method names) in source files and term document matrix is created out of these identifier names obtained from the corpus.

During this phase, our system skips the comments and uses a token file for lookup which consists of keywords and predefined words for a specific programming language and the set of operators. For each source file to be compared, we count the number of occurrences of each of these tokens. It also stores the number of distinct variable names and method names in each file. This is different from the normal approach where the algorithm is allowed to create index automatically by analyzing words in the sample corpus. In our approach, we achieve this by dropping the zero rows from the matrix corresponding to terms that occur in none of the documents.

b. Similarity Score Calculation and Evaluation Criteria:

Document-document similarity for all files in the database is obtained by finding the product A^TA as

$$A^TA = (SV^T)^T(SV^T) \qquad (5)$$

Where A is the term-document matrix referred to in (4) and S and V are the matrices of singular vectors obtained using SVD. We have used the evaluation criteria given in [13]. During testing, it was observed that LSI could retrieve documents with high recall but there was a fall in precision as we increased the number of files. Precision and recall depend on the similarity score cut-off and therefore the similarity threshold should be so chosen that it is low enough to accommodate all the true positives but high enough to reduce the number of false positives. The number of false positives showed a slight increase with increasing number of files. In order to reduce the number of false positives to zero and thus reduce the number of files to be given as input to AST algorithm, which is our main objective, we add a post processing stage to LSI.

c. Post processing Stage:

The vectors in the initial term document matrix corresponding to the files identified as plagiarized are extracted and a different similarity score, Dice's coefficient, is calculated. A high value for both LSI and Dice's coefficient indicates that the files are potentially similar. This stage could filter out all the false positives obtained with LSI.

III. RESULTS AND DISCUSSIONS

We tested our approach for a student program database with 80 C source code files of varying functionality and size and a smaller database of C++ and Java files. The initial database had implementations of different sorting algorithms, greedy algorithms, algorithms for numerical analysis, and linear algebra algorithms. These files were manually plagiarized using the different student cheating strategies listed in [15], which includes changing identifier names, replacing for with while, while with for, while with do-while, do-while with while, if-else-if with switch, switch with if-else-if, function calls with function bodies, group of statements with function calls and so on. This database was initially tested using AST algorithm and we obtained maximum precision and recall of 1. It was observed that increase in the number of files did not affect the precision and recall. However, runtime of AST increased with increase in the number of files. However, runtime of AST increased with increase in the number of files. Fig. 1 shows plots of runtimes of AST and LSI algorithms against number of files under comparison.

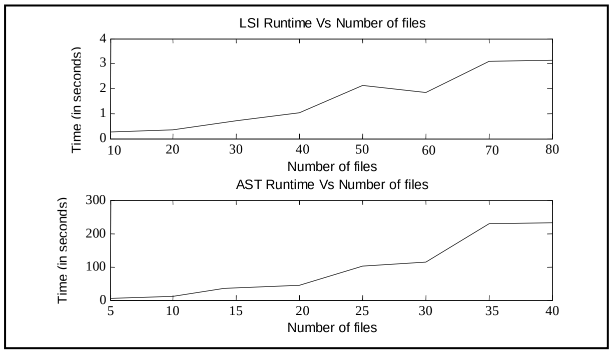

Figure 1. This figure shows the plots of runtimes of LSI and AST algorithms against number of files under comparison.

In [15], Ligaarden makes an estimate of runtime for 2000 files using modified ASTs which is also very high. Hence, we have used LSI to reduce the number of files to be given as input to AST algorithm.

A. Choice of LSI and Selection of Tokens in TokenFile:

Initially, we represented each file as a vector with counts of occurrences of all terms in each file. We used two similarity measures - Dice's coefficient and cosine similarity. We computed precision and recall by using different token lists - with only the keywords, with only the identifiers, with keywords and operators, and with keywords, operators and punctuators. The results were not satisfactory because there were a large number of false positives. A high threshold for similarity identified most of the highly similar files but decreasing the threshold, to accommodate files with slightly lower similarity, decreased the precision.

We then used LSI and tested it using the different token lists and found it to be more promising than the previous measures. The token file we used finally to create the term document matrix was obtained after numerous trials. Considering all the tokens in a file was not found to be effective. A list of identifiers alone produced high similarity scores between very small programs and large programs with same number of variable and method counts. The final token list contains keywords and a set of operators. LSI could effectively retrieve all the similar documents in our database (high recall) for a certain similarity threshold.

Fig. 2 shows the plots of precision and recall against number of files on applying LSI on C source code database.

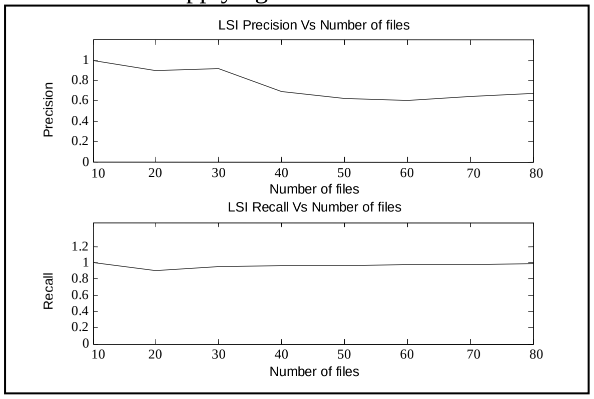

Figure 2. This figure shows the plots of precision and recall against number of files on applying LSI on C source code database.

The fall in precision is due to increase in the number of False Positives(FPs) with increase in number of files (Fig.3).

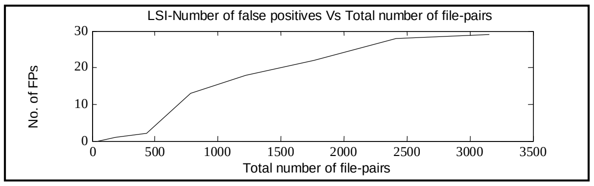

Figure 3. This figure shows the plot of number of false positives against the number of file-pairs on applying LSI.

For the postprocessing of LSI output, we first used cosine similarity of identifier vectors. The counts of occurrences of distinct variable names and distinct method names were stored as two vectors. Cosine similarities were found after making suitable modifications to these identifier vectors to account for their distinct occurrences in the program files. However, it produced many false positives. Then, we tested using Dice's coefficient on vectors extracted from term document matrix which gave better results when applied on LSI output and reduced the false positives to zero.

LSI approach was tested on C, C++ and Java files by selecting respectively the token lists designed for C, C++ and Java.

B. Applying AST Algorithm on Potentially Plagiarized Files:

To generate ASTs for C, C++ and Java files, we have used their respective grammars and have used JavaCC and JJTree tree builder.

The parse tree generated, without any modification in C grammar, for a simple C program consisted of 41 levels and 69 nodes(including 5 leaf nodes). Fig. 4 shows the AST generated for the same program with suitable modifications in the grammar.

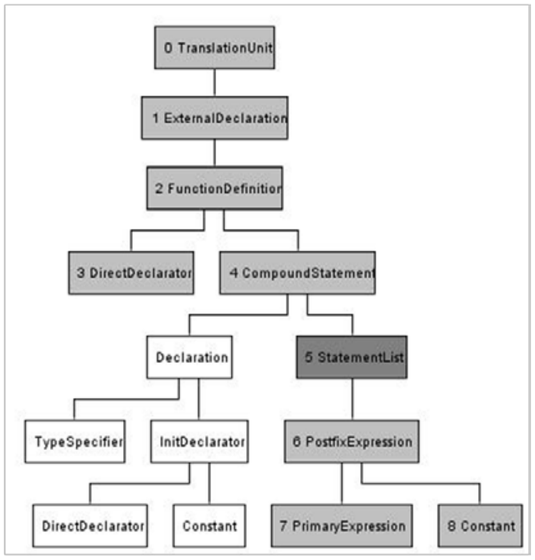

Figure 4. This figure shows the AST for a simple C program.

The size of AST is considerably small compared to the parse tree. Hence, there is a significant reduction in the computational cost involved in AST-based comparison in plagiarism detection as compared to parse tree based comparison.

IV. CONCLUSION

In this paper, we have studied and analyzed the results of combining Abstract Syntax Trees (ASTs) and Latent Semantic Indexing (LSI) for detecting plagiarism in source code files written in C, C++ and Java. We look forward to try different modifications on the parse tree generated by JavaCC to reduce its size and also try different sequence matching algorithms for comparison. We are also exploring methods which can detect cross-language plagiarism. We limit our method for detection of plagiarism in student assignments in educational institutions, the reason being the fact that LSI results are less reliable since it greatly depends on the token list and chosen dimensionality. Our method gave good results on our database. However, finally, it is upto the investigator to decide whether the files flagged as plagiarized are actually plagiarized. Our system just identifies the highly similar files.

V. ACKNOWLEDGMENT

We would like to thank O. S. Ligaarden, research fellow at SINTEF ICT, Norway, for his help and guidance in carrying out this research.

VI. REFERENCES

[1] G. Cosma, An approach to source-code plagiarism detection and investigation using latent semantic analysis, Ph.D. Thesis, University of Warwick, July 2008.

[2] P. Clough, “Plagiarism in natural and programming languages: an overview of current tools and technologies,” 2000.

[3] R. Koschke, “Survey of research on software clones,” In Proc. of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software 2007, 962.

[4] K.L. Verco, and M.J. Wise, “Software for detecting suspected plagiarism: comparing structure and attribute-counting systems,” First Australian Conference on Computer Science Education, Sydne y, Australia, July 3-5, 1996.

[5] K.J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” CSD-TR 200, August, 1976.

[6] M.H. Halstead, Elements of software science, Elsevier, 1977.

[7] G. Salton, A. Wong, and C.S. Yang, “A vector space model for automatic indexing,” Commun. ACM 1975, 18(11): 613-620.

[8] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman, Indexing by latent semantic analysis. J. Am. Soc. Inform. Sci. 1990, 41(6):391-407.

[9] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proc. 2nd WCRE 1995; 86-95.

[10] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proc. IEEE ICSM 1998; (Cat. No. 98CB36272):368-377.

[11] Koschke R, Falke R, and Frenzel P, Clone detection using abstract syntax suffix trees. 13th WCRE 2006; 253-262.

[12] Ducasse S, Rieger M, and Demeyer S, A language independent approach for detecting duplicate code. Proc. IEEE ICSM 1999; 109-118.

[13] Prechelt L, Malpohl G, and Philippsen M, Finding plagiarisms among a set of programs with JPlag. J. Univers. Comput. Sci. 2002; 8(11): 1016-1038.

[14] Wise MJ, String similarity via greedy string tiling and running Karp-Rabin matching. TR, Dept. of CS, University of Sydney, December 1993; 1-17.

[15] Ligaarden OS, Detection of plagiarism in computer programming using abstract syntax trees. Master Thesis, University of Oslo, November 2007.

[16] Kawaguchi S, Garg PK, Matsushita M, and Inoue K, Automatic categorization algorithm for evolvable software archive. Proc. 6th International Workshop on Principles of Software Evolution, 2003.