Code analyzer for an online course management system

Jong Yih Kuo*, Fu Chu Huang

Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106, Taiwan

ARTICLE INFO

Article history:
Received 14 December 2009
Accepted 15 July 2010
Available online 24 July 2010

Keywords:
Intelligent agent
Program similarity
Program plagiarism

ABSTRACT

The online course management system (OCMS) assists online instruction in various aspects, including testing, course discussion, assignment submission, and assignment grading. This paper proposes a plagiarism detection system whose design is integrated with an OCMS. Online assignment submission is prone to easy plagiarism, which can seriously influence the quality of learning. In the past, plagiarism was detected manually, making it very time-consuming. This research thus focuses on developing a system involving code standardization, textual analysis, structural analysis, and variable analysis for evaluating and comparing programming codes. An agent system serves as a daemon to analyze the program codes for OCMS. For textual analysis, the Fingerprinting Algorithm was used for text comparison. Structurally, a formal algebraic expression and a dynamic control structure tree (DCS Tree) were utilized to rebuild and evaluate the program structure. For variables, not only the relevant information for each variable was recorded, but also the programming structure was analyzed where the variables are positioned. By applying a similarity measuring method, a similarity value was produced for each program in the three aspects mentioned above. This research implements an Online Detection Plagiarism System (ODPS) providing a web-based user interface. This system can be applied independently for assignment analysis of Java programs. After three comparison experiments with other researches, the results demonstrated the ODPS has many advantages and good performance. Meanwhile, a combined approach is proven that it is better than a single approach for source codes of various styles.

1. Introduction

With growing attention being devoted to internet-assisted teaching, many online course management systems have been developed (Broder, 1997; Adaptive Brusilovsky, 2001; Perkowitz and Etzioni, 1997). Online course management systems support chiefly various activities in online teaching, including testing, course discussion, assignment submission, and assignment grading (Frasson and Aimeur, 1998). However, the coding style for programming often varies according to each programmer’s personal habit, and is reflected in the use of different techniques in designing with the same programming language. If the similarity between the two samples is high, then it is possibly a case of plagiarism. To plagiarize a program, most people rewrite the original version, including part or all of its interface or contents, to avoid easy human identification.

If the differences between codes are compared manually, it will be very time-consuming. As the coding scale becomes larger and more complex, the success rate of identification also diminishes. In the past, purely textual or structural comparison methods have generally been employed to compare documents and programs, though these methods all have their limitations. This is in part because slight differences in textual comparison will cause the similarity rate to vary significantly. For example, “System.out.println(“This is my first java program, hello world!”);” and “String str = “This is my first java program, hello world!”;” involve completely different programming source codes, but there appears to be many similarities. If it is only evaluated by textual comparison, the output similarity value may be greater than expected. In addition, both structural analysis and variable analysis are liable to identify two programs of the same structure, but for different purposes as having a high similarity match. To improve this situation, combining code standardization, textual analysis, structural analysis, and variable analysis was proposed for enhancing the comparison ability.

There are a number of methods for detecting plagiarism in assignments, and most methods focus only on how to improve similarity. Although increasing the similarity is useful for detecting plagiarism in assignments, how to make a correct detection is also very important. Using a single method will probably get an overly high or low similarity in different cases and it is very easy to make wrong estimation. In view of this, three analysis methods were combined to build an Online Detection Plagiarism System...
(ODPS) plugged into an online course management system (OCMS) for detecting similarity. The ODPS serves as a daemon module for performing analysis. Pre-analyzing the sample through structural and variable analyses before textual analysis will increase the precision of textual analysis. For textual analysis, the Document Fingerprinting Algorithm is employed. For structural analysis, the formal algebraic expression and the DCS Tree structure are used to rebuild and examine the program composition. For variables, not only the information of each variable is a relevant record, but also the formation of variables for calculating similarity is also structurally analyzed. As for fingerprinting, the document is translated into k-grams, and finds a unique fingerprint representing this document. Structural analysis uses formal algebraic expression to simplify the code structure, and applies textual comparison to the structural comparison problem to increase the accuracy of output similarity. In terms of variable analysis, each variable in the code is focused, and both statistical and architectural analyses were performed. This allows the comparison process to be more flexible, and produces a more precise similarity output. Thus, by combining textual analysis, structural analysis, and variable analysis, programs that are likely to have been intentionally plagiarized can be distinguished.

This research examines previously developed online course management systems, because submitted assignments are easily copied. Programming courses are focused, and the processes such as code standardization, textual analysis, structural analysis, and variable analysis are applied to code evaluation. A hybrid approach is proposed by combining and advancing three methods including the Fingerprinting Algorithm (Schleimer et al., 2003), algebraic expression (Canfora et al., 1998) and variable statistics (Donaldson et al., 1981). The rest of the paper is organized as follows. Section 2 discusses the references used in our research. Section 3 presents the proposed analysis method for the codes. Section 4 presents the system design and testing results, and Section 5 contains conclusions.

2. Background and related work

2.1. Textual analysis

Schleimer et al. (2003) proposed winnowing as a method for textural comparison of web data. In this method, the tokens of a document are first partitioned into k-grams. k signifies the number of characters in each k-gram block dictated by the user according to the length of the document. After partitioning, all k-grams are stored into a hash table for distribution. Then a partition is made according to the hashed results, creating “fingerprints”, the smallest unit of comparison. In other words, fingerprints are sets of k-grams after implementing the hash algorithm. The compared document creates one or more fingerprints. The possibility of collision is minimized because the hash algorithm is utilized. To compare two documents, the same fingerprints mean the same document. This means that the fingerprints are unique and can be used for detecting similarity. Therefore, only the minimum hash value in the first window is possible that this value is also apparent in an adjacent window. Last, by juxtaposing the hash values in every window, the section that is being plagiarized can be efficiently detected. This is because when a minimum value appears inside one window, then it is quite possible that this value is also apparent in an adjacent window. Therefore, only the minimum hash value in the first window is recorded for overlapping windows. Only the unrecorded minima are called fingerprints. At the same time, the positions of each fingerprint are also recorded as a basis for comparison, thus increasing the accuracy of analysis.

Gitchell developed a plagiarism detection system to compare token sequences using a dynamic programming string alignment approach (Gitchell and Tran, 1998). This approach first assigns a score to each pair of characters in an alignment score. For example, a match score of 1, a mismatch score of –1, and a gap score of –2. The highest score of a block gives the score of an alignment. The score between two sequences is then defined as the maximum score among all alignments, which is easily calculated by dynamic programming techniques. With this definition a similarity measure between two sequences is defined as follows:

\[ S = \frac{2 \times \text{score}(s, t)}{\text{score}(s, s) + \text{score}(t, t)} \]

which is calculated from the individual score for each block, thus giving a similarity value between 0.0 and 1.0. The higher the value, the more similar the two sequences are.

Chen et al. (2004) proposed a metric according to Kolmogorov complexity (Li and Vitányi, 1997) for measuring the amount of shared information between two sequences. They use a compression algorithm involving the LZ data compression scheme (Ziv and Lempel, 1977) to approximate heuristically the Kolmogorov complexity. An information-based sequence distance is then defined as

\[ d(x, y) \approx \frac{1 - \text{Comp}(x) - \text{Comp}(x[y])}{\text{Comp}(x[y])} \]

where \( x \) and \( y \) represent the strings, \( d(x, y) \) represents the similarity value, and \( \text{Comp}(x) \) measures the amount of absolute information the sequence \( x \) contains. That is, \( \text{Comp}(x) \) is length, in number of bits, for the input \( x \) after being compressed by LZ data compression. Given another sequence \( y \), \( \text{Comp}(x[y]) \) measures the amount of information of \( x \) given \( y \) for free. By definition, \( \text{Comp}(x[y]) \) is the length of the shortest program that on input \( y \) prints \( x \) after being compressed by LZ data compression. The denominator \( \text{Comp}(x[y]) \) is the total amount of information in the compressed concatenated string \( xy \).
2.2. Structural analysis

\( F(p) \) algebraic expression is presented by Canfora et al. (1998) to denote the structure of programming codes. \( F(p) \) allows the analyzer to define a strategy for comparison, wherein the composition of the code is represented algebraically, and it is then reconstructed using a tree-type structure, namely DCS (Dynamic Control Structure) Tree (shown in Table 2). \( F(p) \) is a structure using symbols to represent a process by modularizing the code and transforming it into an algebraic expression. Each procedure in coding is treated as the principal target to represent the code structure and process flow. D-Structure is defined as \( \text{DCS} = \{ \text{sequence, if-then, if-then-else, while-do, repeat-untill, and both number of layers } i \text{ and number of rows } j \} \) employed to represent the program structure. Further, ‘\( \cdot \)’ denotes inclusion, ‘+’ denotes alternative operators, ‘/’ denotes serialization, ‘\( \lambda \)’ (lambda) denotes null (i.e. an empty block), and ‘\( \times \)’ denotes the sequence structure.

A similarity score used in the YAP system (Whale, 1996) is a value ranging from 0 to 100, termed the percent-match, representing the range from “no-match” to “complete-match”. It is obtained by the following formulas:

\[
\text{Match} = \frac{\text{same} - \text{diff}}{\text{maxfile} - \text{minfile}} = \frac{\maxfile - \minfile}{\maxfile}
\]

\[
\text{PercentMatch} = \max(0, \text{Match} \times 100)
\]

where maxfile and minfile are the lengths of the larger and smaller of the two files, respectively. The variable “same” is the number of common tokens in both files, and the variable “diff” is the number of single-line differences within blocks of matching tokens.

The programming structure metrics were used by Ding and Samadzadeh (2004) for authorship identification of Java source code. The metrics were extracted and collected first, and then measured by a statistical process called canonical discriminant analysis (CDA).

2.3. Statements metrics

Ottenstein (1977) applied two program metrics to measure the level of similarity between program pairs:

\[
V = (N_1 + N_2) \log_2(\eta_1 + \eta_2),
\]

\[
E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2(\eta_1 + \eta_2)}{2 \eta_2}
\]

where \( \eta_1 \) is number of distinct operators, \( \eta_2 \) is number of distinct operands, \( N_1 \) is the total number of operand occurrences over all distinct types, and \( N_2 \) is the total number of operand occurrences over all distinct types. However, Whale (1990) has demonstrated that a system developed from this approach cannot detect sufficiently similar programs.

Donaldson et al. (1981) have used statistical analysis to compare similarities for the FORTRAN language. There are two phases for completing the comparison for those assignments: (1) a data collection phase; and (2) a data analysis phase, and the two tables are defined in advance for analysis. The first table is intended to define some important counter items for counting each statement type (shown in Table 3). The second table is for defining the statements as characters (shown in Table 4).

In the data collection phase, the program parses those assignments and records the related information according to the two tables. First, the number of times certain types of statements occur is computed, and separate counters are established for each statement type. Then, assignments are characterized by the order in which statements occur in an assignment.

In the data analysis phase, counter algorithms and string comparison are employed to calculate the similarity. The counter algorithms, including three similarity calculation methods, are as follows: (1) The corresponding counter values are subtracted, and the absolute values of the difference are summed. The smaller the sum value, the more similar the two assignments are. (2) To compare the corresponding counter items of two assignments, if their corresponding values are the same, then the counter is increased by one. The larger the counter value, the greater the similarity between the assignments are. (3) This method, which employed weight to replace the counter value, is an extension of the above algorithm. It enables the instructor to weight each statement type according to the demand of the particular programming problem assigned. The string comparison algorithm would compress the strings first. For example, if the original string is “VVVR==HI=EDDI”, it would be reduced to “VR=HI=EDI”. After compressing the strings, the similarity can be calculated by comparing the compressed strings.

In Eric Wong and Gokhale (2005), the authors use an execution slice-based technique to identify a set of codes for implementing each feature. Depending on whether the execution frequency is considered during the construction of such sets of code, a static as well as a dynamic distance are computed for each pair of features. They proposed using metrics to measure the static and dynamic distance between two features. The former depends exclusively on how the features are implemented in the program, while the latter considers how the features are executed by their invoking inputs according to a user’s operational profile.
3. Source code analysis

This research proposes an agent model with three modules, namely belief module, goal module and plan module. When the agent model gets code files and parameters such as winnowing parameters, hash type or plagiarism term, the agent starts to handle the process. Finally the agent creates an output, which is a plagiarism filename that records the plagiarism list. The proposed agent model integrates textual analysis, structural analysis, and variable analysis to address the similarity comparison function. The agent model is developed with reference to our previous works (Kuo and Chu, 2005; Kuo and Chang, 2007) (as Fig. 1) for the representation.

![Agent model](image)

**Fig. 1.** Agent model.

3.1. Textual analysis

For textual analysis, some meaningless tokens of source code such as comments, space and invisible characters are first removed. Then the Fingerprinting Algorithm (Schleimer et al., 2003) is utilized for textual analysis of source code. This algorithm provides three control variables: the number of characters in a window (W), the number of characters in a k-gram (K), and the “distance” between two windows (G), also in number of characters. The range for G is defined as 1 < G < K + 1. The G value must be larger than 1 to create different k-grams for comparison; it should also be smaller than K + 1, to prevent its effect on output precision due to indistinguishable characters. The value of W should be controlled, so it approximates K, preferably within the range of K + 1 > W > K – 1.

3.2. Structural analysis

Simply performing textual analysis is likely to result in inaccurate similarity output due to the miniscule difference in the documents. This leads to the addition of a structural analysis process to compensate the inadequacy of textual analysis. However, to evaluate the structure, it must first be represented in a standard format, and then followed by analysis (Gitchell and Tran, 1998). In this paper, a formal algebraic expression was employed, namely the F(p) algebraic expression (Canfora et al., 1998), to describe the entire structure. Analysts can then define a strategy for code comparison, and focus on processing the structure rebuilt by the F(p) algebraic expression. By using the F(p) algebraic expression, it is possible to employ a tree-type data structure to build a DCS Tree (Adaptive Brusilovsky, 2001) for each code. For ease of implementation, F(p) is simplified using a tree structure to replace the number of layers and number of rows. Note the symbols for the DCS Tree nodes are capitalized and the conjunction between the nodes is in lower case or non-alphabetical symbols (see Tables 5 and 6). By utilizing a DCS Tree to represent the code structure, a set of DCS Trees can be obtained. Performing preorder traversal on each DCS Tree will then yield a DCS expression in string form. The structural comparison problem then equals a string comparison problem, allowing the use of a textual comparison algorithm. In this study, the scope was extended to textual comparison, which is divided into three phases: division, search, and inspection. (1) Division Phase: divide the target string into units of length n, and right shift one character at a time. The set of DCS expressions with length n is defined as a “table”. (2) Search Phase: use a hash table to compare the content of tables and find units with the same hash value. (3) Inspection Phase: inspect whether the consecutive units

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>seq</td>
</tr>
<tr>
<td>F</td>
<td>if</td>
</tr>
<tr>
<td>W</td>
<td>w</td>
</tr>
<tr>
<td>R</td>
<td>r</td>
</tr>
<tr>
<td>X</td>
<td>Another algebraic expression</td>
</tr>
<tr>
<td>L</td>
<td>λ (lambda), denotes null.</td>
</tr>
<tr>
<td>o</td>
<td>Inclusion</td>
</tr>
<tr>
<td>c</td>
<td>Continue</td>
</tr>
<tr>
<td>+</td>
<td>Alternative operator</td>
</tr>
<tr>
<td>(</td>
<td>Layer(j) + 1, denotes a move to child layer</td>
</tr>
<tr>
<td>)</td>
<td>Layer(j) – 1, denotes a move to parent layer</td>
</tr>
</tbody>
</table>

Table 5
Symbols and corresponding descriptions.

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Algebraic representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>So(XcXcXcXc...cX)</td>
</tr>
<tr>
<td>if-then</td>
<td>Fo(X + L)</td>
</tr>
<tr>
<td>if-then-else</td>
<td>Fs(X + X)</td>
</tr>
<tr>
<td>while-do</td>
<td>Wv(X + L)</td>
</tr>
<tr>
<td>repeat-unti</td>
<td>Ro(X)</td>
</tr>
</tbody>
</table>

Table 6
Algebraic representation of control structures.
found during the Search Phase are identical as well. If so, combine the consecutive units into a larger one.

### 3.3. Analysis of variables

Structural analysis is also liable to miscalculating the similarity results since two original codes with different purposes may have high similarity simply because of their similar structures. This research also adds a process for variable analysis to reduce miscalculation. Through extensive testing results, reliable weights can be found to minimize any miscalculations.

Since programs are written to produce a predictable output by manipulating control structures, the variables play a significant role in program execution. Plagiarism may occur sometimes simply in the form of renamed variables, and so it is important to incorporate variable analysis in code comparison. This study involves statistical calculation of variables in code, as proposed by Donaldson et al. (1981). Processing variables statistically and calculating their similarities are proposed. Each variable goes through the following three processes: (1) Recording basic information phase: information including variable name, data type, and scope (class variable or local variable), (2) Statistical recording phase: the structure where the particular variable has appeared is recorded. If variable x appears inside a “for loop”, then the number of its appearances inside a “for loop” is determined. The structures considered are the if-condition, for loop, do-while loop, increment, decrement, assignment statement, expression, and referenced statements. Calculations are made to see if two variables display the same statistical count under identical control structure. If so, then it is noted in the similarity calculation. (3) Formation recording: two variables with different positions in code may have the same records nonetheless. Therefore the program also records at which layer the variable appears, called its “formation”.

Besides the control structures outlined in Table 7, the list is expanded with a switch/case structure, treating it as an if-else condition. The number of layers is determined by observing code blocks, i.e. the use of left braces “{” and right braces “}” to compute an increase or a decrease in number of layers, respectively. If two variables appear in different layers, then although both may display the same control structure count, the formation record would still be dissimilar.

### 3.4. Similarity computation method

When comparing the level of differences between programming source codes, a formula must first be defined to compute the similarity. According to our previous research (Kuo and Chu, 2005), when evaluating two documents, the similarity for documents A and B is

\[
\text{Sim}(A, B) = \frac{A \cap B}{A \cup B}
\]

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Code example</th>
</tr>
</thead>
<tbody>
<tr>
<td>if-condition</td>
<td>if (...) { x ... }</td>
</tr>
<tr>
<td>for loop</td>
<td>for (...) { x ... }</td>
</tr>
<tr>
<td>do-while loop</td>
<td>do (...) while (...) ;</td>
</tr>
<tr>
<td>increment</td>
<td>x++;</td>
</tr>
<tr>
<td>decrement</td>
<td>x = --;</td>
</tr>
<tr>
<td>assignment</td>
<td>x = 3;</td>
</tr>
<tr>
<td>expression</td>
<td>y = x + 5;</td>
</tr>
<tr>
<td>referenced</td>
<td>y = x;</td>
</tr>
</tbody>
</table>

According to an optimistic judgment, the value of final similarity (FinSim) is

\[
\text{FinSim}(A, B) = w_1 \times \text{Sim}_1 + w_2 \times \text{Sim}_2 + w_3 \times \text{Sim}_3
\]

where Sim$_1$ is the similarity of the Text algorithm, Sim$_2$ is the similarity of the Structural algorithm and Sim$_3$ is the similarity of the Variable algorithm. Each algorithm can be assigned a weight, and the sum of three weights is 1.0. Finally, the similarity ranges are between 0.0 and 1.0.

### 4. System design and implementation

This system focuses on processing Java programming assignments, and novice students tend to code assignments in non-standard formats, which may interfere with code analysis. Then the characteristics of decompilers can be exploited to unify the formats of programming codes (Navarro, 2001). Decompilers (Dyer, 2002) are used for converting class files back into.java files, causing the converted.java files to be in a uniform format. This study uses a JAD decompiler (Kouznetsov, 2004) to pre-process the target codes, thus easing the ensuing processes. It is convenient for parsing and helpful for similarity using the process of compiling and decompiling. In particular, it also enables us to find directly the wrong assignments.

#### 4.1. System structure and process flow

Software system maintenance requires a deep understanding of the existing system so as to modify and integrate it with new or changing requirements. Design patterns represent useful architectural information that can support a rapid understanding of software design and source code (De Lucia et al., 2009). Therefore, the MVC model (Yu et al., 2003) was applied to the design system architecture and object-oriented programming was used to implement the system. Fig. 2 is a diagram of the system structure that is designed to preserve the flexibility of system. The entire system is separated into three sub-systems: model, view and controller. The model module includes textual analysis, structural analysis, and variable analysis. The view module allows the user to operate this system through a simple graphical user interface (GUI). The three analysis algorithms are controlled by the Controller Module that feeds those related parameters and data. Using the MVC model employed in the proposed system now makes it easy to replace all the algorithms in the future. Then Controller Module gets feedback and the result will be displayed on the View Module. Fig. 3 depicts the system process flow. After the user selects the target folder of the source code and sets up the configurations, the system will...
Renaming variables: The only difference between the two test cases is that all variables have been renamed. A.java has variables i and x, whereas B.java named it j and y, respectively. All other details remained the same. After system execution, the two test cases have a similarity of 100%.

Syntax modification: The content of A.java is copied into C.java, except the “for loop”, which is now reconstructed into a while loop. After system execution, the structural similarity is calculated to be 100%. Performing textual analysis on the identical structures yields a similarity output of 78%. According to variable analysis, the similarity output is 94%. The similarity computation method in Section 3.4 was employed to calculate the final similarity, which is about 91%. In fact, A.java is extensively similar to C.java. Then, why is the output similarity only 91%? This is because the analysis is performed using the decompiled class files. In the experiment, an extraneous but unused variable flag in C.jad that reduces the similarity during variable analysis and structural analysis was found. To solve this problem, the extraneous variable will have to be removed prior to analysis.

If the only revisions are variable renaming and syntax modification, then the execution result would be identical to that of the single-method comparison.

(1) Rewriting two methods into a single method: If two codes are analyzed textually, the similarity is only 23%. After structural analysis, the complete similarity result is increased to 67%, with a variable analysis similarity of 88%.

(2) Mixing method sequences: Rewrite two identical codes with the methods in opposite order. The output similarity is 100%, with a variable analysis similarity of 87.5%.

Multiple-classes multiple-methods comparison

In multi-class and multi-method comparisons, the only revision for the testing cases is variable renaming, with other details remaining unchanged. If the target code has inner classes, individual files will be created for each inner class after de-compilation, thus reducing the accuracy. In this case, the comparison should be targeted at one directory (containing one program) at a time. The directory comparison method gives a result of 100% similarity. The original code comparison yields a structural similarity of 99%, with variable similarity at 100%. Therefore, the total similarity is about 99.5%.

Multiple-files comparison

The ODPS system can reciprocally compare multiple files. An instructor just assigns the folder to contain all checked files, and it will compare those files automatically and find suspicious files with a similarity value higher than the value set by the instructor. It was tested by running 36 assignments for this comparison and obtained the plagiarism list that is written as a text file.

Making well-known tricks

A sample code was created to make some well-known tricks, including class renaming, function rename, adding comment, change variable position, deleting space line and change parameter position (shown in Fig. 5).

Table 8 is the result of executing ODPS and Software Integrity Diagnosis system (SID) (Chen and Francia, 2004). The ODPS finds a high similarity because those tricks cannot influence its structural and variable analyses. Only textual analysis is slightly affected. The detailed similarity of the modified file is (Textual: 0.77, Structural: 1.0, Variable: 1.0, Average: 0.92). This case illustrates the advantages of combining different methods. If only textual analysis is adopted, the result of similarity is different from the real situation.

<table>
<thead>
<tr>
<th>Similarity</th>
<th>Original</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODPS</td>
<td>1.0</td>
<td>0.92</td>
</tr>
<tr>
<td>SID</td>
<td>0.96</td>
<td>0.49</td>
</tr>
</tbody>
</table>
4.4. Random insertions

Plagiarizers can also use a sample trick to confuse instructions or overcome a checking system by inserting irrelevant statements such as int x = 0. Chen et al. (2004) compared their system with JPlag (Malpohl) and Measure Of Software Similarity (MOSS) (Aiken) by random insertions. The same test files from http://software.bioinformatics.uwaterloo.ca/SID/ built by Chen were downloaded, and the ODPS system was executed to compute similarity value and determine the level of performance. Fig. 6 clearly shows that with increase in number of irrelevant statements, both JPlag and MOSS deteriorate to the point that they find the programs to be dissimilar. Although the SID shows a substantial improvement, it still deteriorates with an increased in insertions. On the other hand, the ODPS maintains a stable and high similarity that never drops below a 60% similarity. There are two main reasons why the ODPS can obtain a stable and high similarity. First, an average value for text structural and variable similarity is adopted. Second, a winnowing algorithm is employed to compare fingerprints, and if the fingerprints are not destroyed, they can still be compared easily.

4.5. Comparing with other systems and single method

The ODPS adopted an approach combining textual analysis, structural analysis and variable analysis to calculate similarity. In this section, a clear comparison is presented if the methods are supported by the ODPS or other systems as shown in Table 9. Meanwhile, the combined approach is separately compared with every single method. Analyzing the experimental result shows that every single method has its inherent shortcomings, which make possible easy attacks or wrong judgments. However, the ODPS can easily avoid these problems, and has a more reasonable detection.

4.5.1. Comparing with other systems

Table 9 shows the results of comparing our ODPS system with other plagiarism detection systems. The comparison schema of Table 9 includes “Text Analysis”, “Structural Analysis” and “Variable Analysis”.

SID (Chen and Francia, 2004) is based on Kolmogorov (Li and Vitányi, 1997) complexity and its universal. Universality guarantees that their measure will detect similarity, if any, between two sequences under any computable similarity metric. Donaldson et al. (1981) implemented a detection system using the SNOBOL4 programming language. The analysis of assignments is done in two phases, namely data collection phase and data analysis phase. In the data collection phase, each assignment is read, line-by-line, and information is gathered on the characteristics of the assignment. In the data analysis phase, these characteristics are compared and tables of results are constructed. Ding and Samadzadeh (2004) extracted a set of software metrics of a given Java source code as a fingerprint to identify the author of the Java code. The contributions of the selected metrics to authorship identification were measured by a statistical process, namely canonical discriminate analysis, using the statistical software package SAS. JPlag (Malpohl) is a system that finds similarities among multiple sets of source code files. This way can detect software plagiarism. JPlag does not merely compare bytes of text, but is aware of programming language syntax and program structure and hence is robust against many kinds of attempts to disguise similarities between plagia-
ized files. Details of the algorithm used by MOSS system (Aiken) are not given to prevent circumvention. However, But it is believed to be a tokenizing procedure followed by a fast substring-matching procedure. Experimental results show that pairs of files are sorted by the size of their matching token blocks.

The ODPS provides a web-based interface for the user to operate. It also supports to detect multiple classes and files and considers many aspects when calculating similarity for more objective detection of plagiarism. Table 9 shows that the ODPS supports all comparison methods, whereas other systems only perform one or two comparison methods.

4.5.2. Comparing with structural analysis

Comparing ODPS with structural analysis, a pairs of assignments that have simple program structures are provided for this experiment. The main features of the assignments are as follows: (1) it can be inputted in any format like string, integer and so on; and (2) the if-else condition is utilized to check the input and print to the screen of the application. The ODPS system was executed to run this sample including two assignments and found no plagiarism. The similarity of the combined approach is 0.62 (text: 0.616, structural: 1.0, variable: 0.25), but the structural analysis reaches 1.0. In general, the assignments of students are simple cases whose structure is not very complex. If the plagiarizing system is merely using structural analysis to check the assignments, it cannot precisely judge those assignments when the structure of program is not very complex. However, this problem can be avoided using the combined approach.

4.5.3. Comparing with textual analysis

According to the above case, the sample is modified into a more complex structure. The parts added are as follows: (1) adding the while-loop condition to many times let the user input data from the screen of application, and (2) the program needs to count the number of times user inputs the strings. Therefore, a variable has to be added to this sample to record the counter. One of the assignments tried to plagiarize another assignment, and it modifies many characters of the source code including the variable name and printed strings. After executing the ODPS system to check this sample, the similarity of ODPS is 0.94 (text: 0.82, structural: 1.0, variable: 1.0). However, the textual similarity is only 0.82. Clearly, when the system is using textual analysis only, it can be easily cheated by changing some strings. However, the ODPS is only slightly affected. The main cause is that the structure of program and variable retains the original plagiarized assignment.

4.5.4. Comparing with variable analysis

By using the experiment of random insertion, the combined approach was compared with variable analysis. When more and more statements “double x=0;” are inserted into the source code, the curve of variable analysis drops rapidly. However, the combined approach keeps a relatively stable decline (shown in Fig. 7). The main cause is that there are many leaks in the variable analysis, and it can be easily attacked. The variable analysis focuses only on comparing variables, and wrong judgments easily occur.

5. Conclusion

This study focuses on programming assignment analysis for OCMS. Moreover, how to judge whether programming codes are identical to each other by building a code comparison system to replace conventional exhaustive methods (performed manually) was discussed. This research also proposes a method of combining code standardization, textual analysis, structural analysis, and variable analysis methods to improve the similarity comparison function. For code standardization, the JAD decompiler (Kouznetsov, 2004) is employed to unify the format of the source code and the Document Fingerprinting Algorithm is applied to textual analysis. Both the formal algebraic expression and the DCS Tree structure are utilized to rebuild and evaluate code formation in structural analysis. For variables, not only is relevant information about each variable recorded, but a formation analysis for each variable is also provided. This similarity comparison program outputs the similarity for three individual aspects. A document comparison system allowing the user to compare instructive documents aside from programming codes is also provided. The proposed method implements a web-based user interface, permitting this system to be used independently. In addition, an ODPS system connected to an online course management system has been developed. The system is also designed to output various details, enabling users to check each file without difficulty. By changing the belief module and analysis rules, the analysis power can be further improved.

Further, some case analyses to illustrate our system functions described in Section 4.2 are also provided. Those comparison methods include the “single class single method”, “single class multiple method” and “multiple files”. It mainly proves that the ODPS system provides relatively complete support and obtains high similarity. In addition, some well-known tricks and random insertions are also utilized to compare with the SID system. In the comparison using some well-known tricks, the similarity of the ODPS system is much higher than the SID system. The cause is the SID only supports textual comparison. In the comparison using random insertions, the curve of similarity presents a stable decrease on ODPS while other systems drop rapidly through inserting more and more irrelevant statements. This is because the ODPS system avoids being cheated and supports more comparison methods. At the same time, the combined method is clearly verified to be better than the single method using individual comparison.

However, there are still some areas for the ODPS system to improve in the future. First, the intelligent agent function has to be enhanced, thus enabling the system to make more accurate judgments. Therefore, replacing manual operation with an intelligent agent is needed. Second, the similarity weight is defined by the user. It would be the better to enable the system to automatically set weight values according to different assignments. Finally, although the ODPS can also compare programs other than Java by textual analysis, supporting more popular programming languages to enable the ODPS to be more widely employed is desired. In particular, some languages like C language inherently have different programming structures and concepts. Implementing those popular languages require much effort. Faced with such a situation, the MVC model was adopted to develop system architecture and object-oriented programming was utilized to design the system. The main purpose is to reserve future extensive flexibility. Moreover, another noticeable solution is to provide a language-
independent approach like SCAP (Frantzeskou et al., 2008) using a high-level feature to measure distance.

Acknowledgement

This work was supported by the National Science Council under grant number NSC NSC-97-2752-E-008-001-PAE and NSC-97-2218-E-027-004.

References


Jong Yih Kuo received his BS degree from National Tsing Hua University, Taiwan, Republic of China, in 1991, and his PhD degree from the National Central University, Taiwan, in 1998. He is now an Assistant Professor in the Intelligent System Laboratory of the Department of Computer Science and Information Engineering at the National Taipei University of Technology in Taiwan. His research interests include agent-based software engineering and fuzzy logic.

Fu Chiu Huang is a PhD student at the National Taipei University of Technology in Taiwan. He is now in the Intelligent System Laboratory of the Department of Computer Science and Information Engineering. His research interests include agent-based software engineering and fuzzy logic.