THE JK SYSTEM TO DETECT PLAGIARISM

Khair Eddin M. Sabri
Computer Science Department
University of Jordan
Amman, Jordan
sabrikm@ju.edu.jo
and
Jubair J. Al-Ja'afer
Computer Information System Department
University of Jordan
Amman, Jordan
jubair@ju.edu.jo

ABSTRACT

In this research a system, referred to as Jubair-Khaireddin (JK), has been developed to assess the degree of similarity between two programs even though they look superficially dissimilar. The JK system has the capability to detect deliberate attempts of plagiarism. Reverse engineering technique is used to bring each suspected program back to its initial specification stage. This operation enables us to extract the structure of the program which is an important factor in detecting plagiarism. This can be achieved through the extraction of the Static Execution Tree (SET) for each program. The SET is then transformed into Terminating Binary Sequence (TBS). The TBSs generated from the tested programs are compared in order to get similar branches. Reengineering technique is then applied on these similar branches in order to compute its entropy (information content). The entropy is computed to prove or disprove the existence of similarities between programs. The JK system has been tested on different Java programs with different modifications, and proved successful in detecting almost all cases including those of partially plagiarised programs.

Keywords: Entropy, Plagiarism, Reverse Engineering, Software Engineering, Java Programs

1. INTRODUCTION

Students of various disciplines, especially Computer Science, develop large numbers of software systems worldwide every year as part of their learning process. In most cases, their instructors have the feeling that some of these programs are in fact copies (with some modifications) from software developed by others. Such unacceptable conduct of claiming others' work partially or completely is called "plagiarism".

Conventional inspections for plagiarism are proved ineffective and time-consuming. The instructor might discover plagiarism incidentally by observing that a student has forgotten to replace the name of his/her friend in the program source text, or if two programs produce the same weird failure for a test input. Therefore, better automated inspections that can find similar pairs among a set of programs would be practically more effective.

Plagiarism detection is a pattern analysis problem. A plagiarized program is either an exact copy of the original, or a variant obtained by applying various textual transformations. A method to detect plagiarism must produce a measure that quantifies how closely the two programs are similar. Except for the case of a verbatim copy, detection approaches that use direct comparison of text files are usually weak, since there is no obvious closeness measure. Most techniques adopt a lexical approach, where the program tokens are classified as language keywords and user symbols.

In this research, the JK system is developed to be capable of detecting deliberate attempts of plagiarism. The system is intended to be used by instructors to detect plagiarism acts among students. The JK system also has wider applications, as it offers a means of characterizing programming style and attaching a "Fingerprint" to the program. Another incentive for developing the system is the protection of intellectual property. It is of note that plagiarism in software is considered an extremely common practice of violating intellectual properties and copyrights.

2. RELATED WORK

Early attempts to detect plagiarism were usually based on feature comparison. Most systems were based on computing a number of different software metrics for each program. The systems then consider sets of programs that lie close to be possibly plagiarized [1].

The earliest system is Ottenstein dated 1976. It applied only basic Halstead metrics [2] on FORTRAN language (number of unique operators n1, number of unique operands n2, total number of operators N1, and total number of operands N2) and considered programs to be plagiarized if all four values coincide [2]. Later systems such as those of [2,3] introduced much larger number of metrics (up to 24) in order to improve performance. Also, other metrics-based systems for monitoring similarities between programs are presented [4,5].

On the other hand, there are systems that depend on the structure of the program rather than on summary indicator. Such systems are "MOSS" [6], "YAP3" [7], "JPlag" [8] and "SIM" [9]. Measure Of Software Similarity (MOSS) [6] is an automatic system for determining the similarity of C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs. To date,

MOSS has been mainly used in detecting plagiarism in software programs. Since its development in 1994, MOSS has been very effective in this role. Moss is being provided as an Internet service.

The "Yet Another Plague (YAP)" [7] series of tools are based on the Plague plagiarism detection tool. Michael Wise created the original version of "YAP1" followed by "YAP2". In 1996, Wise produced the final version of "YAP3" that depends on an algorithm called "Karp-Rabin Greedy-String-Tiling (RKS-GST)" [1]. Also, "SIM" [9] is used to detect plagiarism among a large set of programs in the C language. "SIM" works by converting each program into a stream of tokens. These tokens are compared using "string alignment techniques" to detect similarity.

"JPlag" system [8] is used to find pairs of similar programs among a given set of programs. It has been successfully used in practice for detecting plagiarisms among student Java program submissions. It also provides support to the languages C, C++ and Scheme and if is widely available as a web service. It takes a set of programs as input, compares them pairwise (computing for each pair a total similarity value and a set of similarity regions), and provides a set of HTML pages as output that allows for exploring and understanding observed similarities in detail. JPlag works by converting each program into a stream of canonical tokens, and then trying to cover one token string by substrings taken from the other one (string tiling).

It is concluded that when very close copies are involved, the attribute-counting-metric systems perform better than the structure comparison system in the detection of plagiarism. On the other hand, if a student copy is only a part of another student's program, the attribute-counting-metric systems will not be able to detect plagiarism [10].

There are other systems which depend on both structure and metrics comparisons, such as [3,11]. For example, Jankowitz [11] constructed a 'template' for each program to extract similar regions or areas from the set of programs. Then a statistical analysis was performed on these regions in order to detect plagiarism. Whereas, Donaldson et al [3] used eight attributes counting metrics and generated a string representation of the program text. Each letter in the string represents single or multiple adjacent occurrence of program structure, such as variable declarations, assignment statements and procedure calls. These Strings are compared in order to detect similar programs.

There are other techniques used to detect plagiarism, such as [12,13]. Baker and Manber used the bytecode to detect similarity in Java programs. They adopted three tools designed to find similarity in both source code and text in order to work with bytecode files [12]. Cunningham and Alexander discovered another approach to detect plagiarism in computer assignments using a Case Based Reasoning (CBR) approach [13]. The problem of finding similarity in programs is made analogous to the problem of case retrieval in CBR [1].

The mentioned tools are effectively updated and extended to cover other programming languages such as C#. Also, new tools are developed to detect plagiarism such as [14,15].

3. ENTROPY MEASUREMENT

One of the most fundamental results of the statistical communication theory is the Shannon's information theory [17]. By ignoring the meaning of a message and focusing on the probability of choosing any symbol out of the message, Shannon was able to establish an Entropy function which measures the statistical information content. By applying the Shannon's concept to a program, its Entropy can be calculated using the following formula:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where:

p_i: probability of occurrence of the i^{th} symbol in a message.

n: total number of symbols

Entropy concept can be used in a program by considering the program as a message and the symbol is either an operator or operand. The Entropy software metric, discussed by Davis and LeBlanc [18], explains the notion of Entropy at a higher conceptual level by considering the so-called chunks of code. A chunk could be a single statement, a block of code, or a module itself. An important notion is an equivalence class of chunks. The concept of the equivalence class is based on chunks' in-degree and out-degree. Two chunks are considered equivalent if they have the same number of in-degree and out-degree of links.

4. THE PROPOSED APPROACH

Two techniques are used to detect plagiarism. The first one is based on comparing the structure of the programs and extracting similar branches. The second one is based on analyzing the similar branches by comparing the general characteristics and computing the Entropy (information content in the programs).

Program Structure

Plagiarism could be detected in the programs by comparing the Terminating Binary Sequence (TBS). TBS can be obtained by constructing the Static Execution Tree (SET), transforming it to a Strictly Binary Tree (SBT), and then generating the TBS as shown in figure 1.

Determining the Static Execution Tree (SET): The SET represents the interconnection between the "main" method of a program to its other methods (functions). SET can be constructed by parsing the source program and doing the following:

Step1: The "main" method is made the root of the SET.

Step2: A branch is added to each method called from the main method.

Step3: The same recursive algorithm is applied on each method until no further call is made to another method.

The generated tree is then slightly altered to generate another SET free of user-dependent method name. The algorithm is quite simple: each time a different method is encountered, a unique number is assigned to replace the method's name. By using this technique, all user-dependent information is stripped out leaving the skeleton structure of the program.

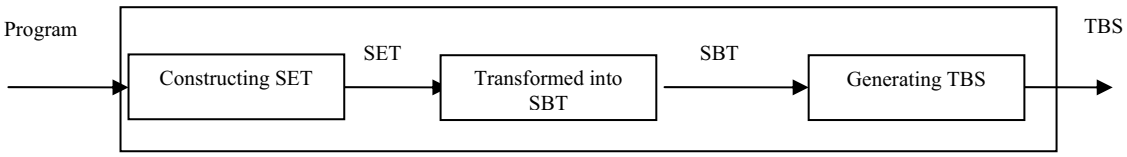


Figure 1: The process of generating the Terminating Binary Sequence (TBS)

Determining the Strictly Binary Tree (SBT): A SBT is defined as a tree in which every node has either 0 or 2 children. It can be produced by transforming the original SET. If the original SET has "n" nodes, the resulting SBT will have (2n-1) nodes [11].

The transformation can be accomplished by using the following algorithm [19]. Given an ordered rooted tree, the SBT can be constructed as follows:

Step1: If the tree is a single node, the SBT is just the root.

Step2: If the tree is not a single node, cut the branch between the root and its eldest son. This divides the original tree into two parts: the left and right subtrees of the root of the SBT. The left subtree is the part of the original tree rooted in the eldest son of the root, and the right subtree is the remainder of the tree including the root.

Step3: Recursively, repeat steps (1) and (2) on the two parts of the original tree.

Determining the Terminating Binary Sequence (TBS): TBS is defined as a binary set of numbers that includes only 0 or 1 numbers. For a general tree, the TBS can be constructed as follows:

1. Transform the tree into a strictly binary tree.
2. Traverse the resulting SBT using the pre-order walk, and when a node is visited put a "1" in the binary sequence if the node is a branch node and a "0" if the node is a terminal node.

Some interesting properties of the TBS are the following:

1. The original SBT can be reproduced without losing any information from the TBS
2. A tree with "n" nodes has a TBS with "n" 0s and "n-1" 1s.
3. A group of "k" 0s corresponds to a node with k sons in the original tree. (In our model, this group represents a method which has called k methods. The followed '1' is the method that calls 'k' methods.
4. This technique enables us to find in a given tree, all the occurrences of a subtree with a certain structure. This sequence forms the basis of our analysis.

Performing the Analysis

Once two identical branches are identified, their corresponding methods (functions) are extracted and analyzed. There are two separate analysis techniques. The first one inspects the global characteristics of the procedures. The second one, involving Entropy measurement, requires more detailed analysis.

Inspecting the General Characteristics: The statistics gathered in the first part are:

1. code lines
2. Attributes
3. reserved words
4. assignment statements
5. IF statements
6. FOR statements
7. WHILE-DO statements
8. CASE statements
9. function calls

Each of the above factors is computed for the two programs (the authentic and the suspected).

Measuring the Entropy: The Entropy measurement is used to further analyze the extracted branches and as a crucial step to detect plagiarism. Based on Entropy, four measurements are conducted:

1. Entropy based on the interconnection between the classes: The Entropy of the class connection can be computed by considering the chunk as a class. The class which has the same number of sub and super classes will be in the same equivalence classes.
2. Entropy based on the interconnection between the methods: The Entropy of method connections is computed by considering the chunk as a method. The methods that have the same number of "call and called" methods are placed in the same equivalence class.
3. Entropy based on operators count: This factor is used to compute the information content of each method in the program as well as the whole program depending on the number of operators. The Entropy is computed as follows:

$$H = -\sum_{i=1}^{j} \frac{n_i}{n} \log_2 \frac{n_i}{n}$$

Where:
j: number of distinct operators.
n_i: frequency of occurrence of the operator i.
n: total number of operators

4. Entropy based on the operands count: This factor is also used to compute the information content of each method in the program as well as the whole program depending on the number of operands. The Entropy is computed as follows:

$$H = -\sum_{i=1}^{j} \frac{n_i}{n} \log_2 \frac{n_i}{n}$$

Where:
j: number of distinct operands
n_i: frequency of occurrence of the operand i.
n: total number of operands.

The Algorithm

Input: Two Programs
Output: Plagiarized programs or none
Step 1: Compare the two programs by using the four Entropy factors.
Step 2: If the factors (1, 2, 3, 4) of the two programs have similarity of more than 95%, the system indicates plagiarism
Step 3: If there is no significant plagiarism in the whole program, branches are compared. SET is used to extract similar branches.

1. Parse the two programs and construct SET for each program.
2. Convert SET into SBT.
3. Generate the two TBSs.

Step 4: The system searches for the largest segment existing in both TBSs.

Step 5: If the above segment consists of more than three zeros, the methods corresponding the zeros are extracted using the mapping function. The following analysis is then performed:

1. Comparing the general characteristic:

   a: For i=1 to n, where n is the total number of factors

   If (similarity > 95%) depending on factor i, Then Increment COUNTER by 1

   b: If ((COUNTER / n) * 100) > 50, then Perform the deep analysis (Entropy) Else, there is no similarity

2. These functions are then compared by using the Entropy measurement (Factors 3, 4).

   If (similarity between the two branches based on the two entropy measurements > 95 %), Then There is Plagiarism. Else, there is no similarity

Step 6: If the segment found in step 5 has less than three zeros, then no indication for plagiarism.

5. EXPERIMENTAL RESULTS AND ANALYSIS

To test the system's capability of detecting plagiarism, three test examples are given. First, two simple programs are presented; one of them is plagiarized from the other. Second, two relatively large programs are given: one of them is applied for a bookstore which adds new records and view existing records. Each record consists of the ISDN, name, and price of each book. This program is plagiarized to be used for adding and viewing employee name, number, and salary. The third example represents a partially plagiarized bookstore program which adds new employee record. All three plagiarism cases are successfully detected by the system, as shown in Appendix A.

The experimental results indicate that the JK system is capable of detecting plagiarism among Java programs even in the case of partial plagiarism. First, we will show the effects of common modifications on the Entropy values based on the operators and operands.

Case1:

Table 1: Similarity based on Entropy case 1

Entropy1			Entropy 2		
Prog 1	Progr 2	Similarity	Prog 1	Prog 2	Similarity
3.84	3.84	100%	4.89	4.89	100%

Modifications:

1. Code formatting.
2. Insertion, modification or deletion of comments.
3. Changing the names of variables methods or classes.
4. Alteration of modifiers such as private, final … etc.
5. Modification of constant values.
6. Replacing a for-loop by a while-loop or vice versa.
7. Reordering the cases of a switch-statement.
8. Recording independent statements within a basic block.
9. Promote an int to a long.
10. Reordering a cascading if-statement.

Analysis:

The above modifications could not change the value of the Entropy, since no changes were made on the operators or operands of the program. As we can see from the modification, changing the comment or output will not change the operator or operand in the program. The same thing applies to the changing of variable names, or the order of independent statements. The value of entropy is not affected as the number of operators and operands in the program remains the same. In the case of replacing the "for-loop" by "while loop", only the structure of the loop is changed and the operator "for" is replaced with "while".

Case 2:

Table 2: Similarity based on Entropy case 2

Entropy1			Entropy 2		
Prog 1	Progr 2	Similarity	Prog 1	Prog 2	Similarity
3.84	3.89	98%	4.89	4.89	100%

Modifications:

1. Splitting or merging variable declaration lists.
2. Replacing a sequence of if-statement by a switch-statement.
3. Replacing an int[2] by two separate int or vice versa.

Analysis:

These modifications have no effect on the Entropy value which depends on the operands that doesn't change in the program. On the other hand, there is a minor change in the Entropy value which depends on the operators. For example, replacing "if-statement" with "switch-statement" would increase the Entropy as new operators are defined (switch, case, break, default) instead of (if, else) operators. However, the changes have a slight effect on the similarity (98%). Splitting or merging the declaration of variables will not affect the number of declared variables (operands), but a new operator will be defined that affects the value of Entropy 1.

Case 3:

Table 3: Similarity based on Entropy case 3

Entropy1			Entropy 2		
Prog 1	Progr 2	Similarity	Prog 1	Prog 2	Similarity
3.84	3.89	98%	4.89	4.92	98%

Modifications

1. Moving an initialization away from the declaration.
2. Explicitly initializing with defaults value.
3. Adding or removing unused code.
4. Importing additional packages and classes.
5. Inserting output statements.
6. Moving a block of statement into a new method.

Analysis:

The modifications affect, to some extent, the value of both entropy 1 and entropy 2. This effect depends on the size of the program and on the degree of modifications. In general, the system can detect plagiarism in all cases expect when there is a large modification in a small program. Any modification that changes the number of operands or operators will affect the value of Entropy. For example, adding output statements will increase the number of operands (output) and operators (print functions) and, therefore, the Entropy value will change. The same result is obtained if new methods are created, or explicit initialization variables are added. It can be concluded that any changes to the number of operands and operands will change the Entropy.

It is worth mentioning that the other two Entropy factors, based on the methods and classes, are changed if the structure of the program is changed. The system is tested on various programs including small-sized programs, medium-sized programs, relatively large-sized programs and partially plagiarized programs.

Small-Sized Programs

A set of programs is selected from elementary classes to detect plagiarism. It was found that most of the programs are similar due to the fact that the structure and the idea of elementary programs are almost the same. In order to discover plagiarism, the similarity value should be 99 or 100 %.

Medium-Sized Programs

Two programs (calculator, analogue clock) are given to students to perform some modifications. It is noted that all changes are based on renaming variable names, reordering statements, and changing the place of functions. Therefore, the system detected all cases and the similarity was more than 98%. Also, as there was no modification on the structure of the methods or classes in all programs, the measurement of similarity-based structure was 100 %.

Relatively Large-Sized Programs

A relatively large-sized program, which contains more than one class and a large number of methods, has been selected to test the system. The program was given to students in order to make some modifications. The four Entropy measurements indicated a high similarity between the original and modified programs. However, there were few cases where the students made some changes to the structure of the program and the Entropy measurements failed to detect them. However, in such cases, the system was able to extract similar branches by determining the TBS.

Partially Plagiarized Program

In this case, a part of a relatively large-sized program has been plagiarized. The Entropy measurements of the whole program successfully detected some cases of partial plagiarism. However, when TBS is used, the similar branches are discovered and then detected.

6. CONCLUSIONS AND FUTURE WORK

The following conclusions have been reached based on the application of the JK system on different programs using various modifications.

1) Entropy measure is suitable for detecting plagiarism.
2) The values of the structure-based Entropy are in general be the same in both the original and modified programs.
3) The TBS indicator is particularly useful in the case a partial plagiarism.
4) Frequent addition or merge of methods would change the TBS of the program. Therefore, in some cases, the system fails to extract similar branches.
5) Changing the Entropy of a program, by adding redundant statements, may affect the Entropy values and mislead the system.
6) In case of small-sized programs, it is not recommended to compare the TBS as it is almost equal for all programs. Also, it is better to assume that two programs are plagiarized if the similarity value is more than 99 %. This is due to the fact that the structure and implementation of the programs are almost the same.
7) The system is able to detect many plagiarism attempts that many other systems fail to detect such as: replacing "while" statements with "for", replacing "if" statement with "switch-case" and reordering independent statements

Suggested future work

1. Apply the concept of entropy measures to cover all software quality indicators subjected to modifications such as complexity, effort, difficulty … etc.
2. Apply the concept of entropy to detect plagiarism in textual products (Articles, Novels, … etc).

APPENDIX

Example 1

Original

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class program1 extends JApplet implements ActionListener {
  private Container container = getContentPane ();
  private JTextField textfield= new  JTextField (10);    // TextField to
read the mark
  // The initialisation of the user interface
    public void init () {
              container.setLayout (new FlowLayout());
              JLabel label = new JLabel ("Enter the mark..");
              container.add (label);
              textfield.addActionListener(this);
```

```
            container.add (textfield);
}
// Automatically invoked when the use press Enter.
  public void actionPerformed (ActionEvent e ) {
  int mark=Integer.parseInt (textfield.getText()) ;  // read the mark
  String msg ;
   if (mark > 100 || mark < 0 ) {
     msg="Mark is out of range" ;
     container.setBackground(Color.blue);
  } else if (mark>=90) {
  msg="Your Grade is A" ;
  container.setBackground(Color.green);
} else if (mark>=80) {
  msg="Your Grade is B" ;
  container.setBackground(Color.cyan);
} else  if (mark>=70) {
  msg="Your Grade is C" ;
  container.setBackground(Color.black);
} else  if (mark>=60) {
  msg="Your Grade is D" ;
  container.setBackground(Color.red);
} else {
  msg="Your Grade is F" ;
  container.setBackground(Color.gray);
} showStatus(msg);  // print the message in status bar
```

Plagiarized

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class program2 extends JApplet implements ActionListener
 { JTextField textfield= new  JTextField (15);
   byte m;
   Container container = getContentPane ();
   String grade ;
   public void init ()
    { container.setLayout (new FlowLayout());
      JLabel label = new JLabel ("Enter the mark..");
      container.add (label);
      textfield.addActionListener(this);
       container.add (textfield);
    }
    public void actionPerformed (ActionEvent e )
     { m=Integer.parseInt (textfield.getText()) ;
       if (m > 100 || m < 0 )
        { grade="Error" ;
          container.setBackground(Color.blue);
        } else if (m>=90)
        { grade="A" ;
          container.setBackground(Color.green);
         } else if (m>=80)
         { grade="B" ;
             container.setBackground(Color.cyan);
          } else  if (m>=70)
           { grade="C" ;
               container.setBackground(Color.black);
            } else  if (m>=60)
              { grade="D" ;
                container.setBackground(Color.red);
              } else
              { grade="F" ;
                container.setBackground(Color.gray);
              } showStatus(grade);
    }  }
```

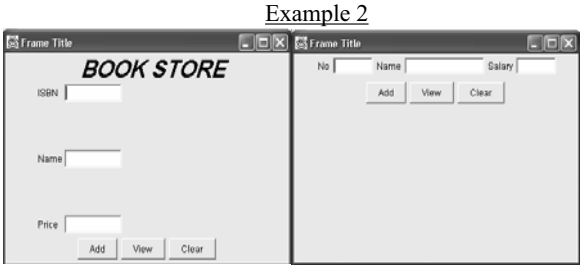
Figure 1: example of two programs

Table 1: The output using JK system

Entropy Factors	Program 1	Program 2	Similarity
Entropy 1	1.0	1.0	100%
Entropy 2	1.0	1.0	100%
Entropy 3	3.90	3.83	98%
Entropy 4	4.89	4.87	99%

Result: Detecting plagiarism

Example 2



(a) Original (b) Plagiarized

Figure 2: Example of two plagiarism programs,

Table 2: The output using JK system

Entropy Factors	program1	program2	Similarity
Entropy 1	0	0	100%
Entropy 2	2.59	2.59	100%
Entropy 3	4.48	4.48	99%
Entropy 4	6.21	6.02	97%

Result: Detecting plagiarism

Example 3: This program plagiarized only add and clear functionality

Entropy Factors	program1	program2	Similarity
Entropy 1	0	0	100%
Entropy 2	2.70	2.70	99%
Entropy 3	4.48	4.47	99%
Entropy 4	6.21	6.03	97%

Result: Detecting plagiarism.

If a TBS technique is applied, it works as follows:

Step 1:

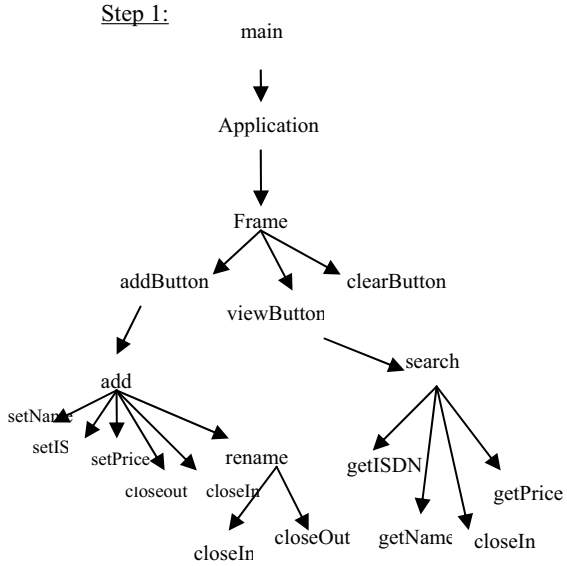


Figure 2: The Static Execution Tree (SET) of the original program

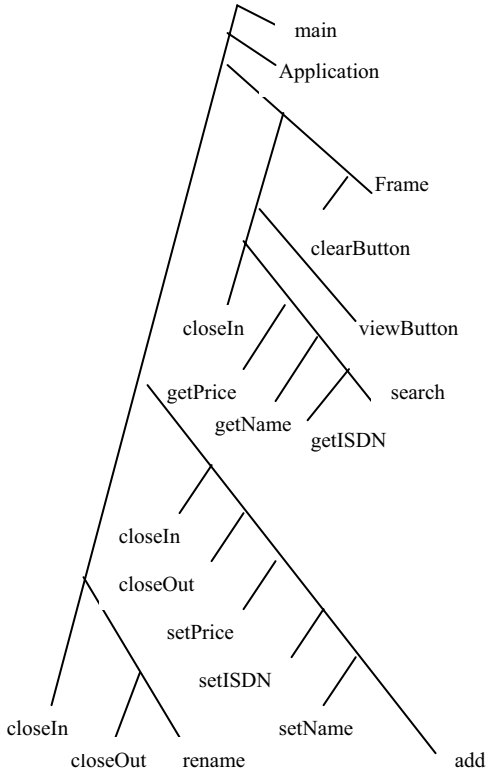


Figure 3: The Strictly Binary Tree (SBT) of the original program

The TBS generated from the original program is:
10101110010111100000**1011111101100000000**

The TBS generated from the plagiarized program is:
10101100**1011111101100000000**
The identical pattern is: **101111110110000000**
The extracted methods from the original program are: *addButton, add, rename, closeOut, closeIn, setPrice, setISDN, setName*. The extracted methods from the plagiarized program are: *addButton, add, rename, closeout, closeIn, setSalary, setNo, setName*

Step 2: Analysis of the extracted methods:

Table 3: The first phase analysis

Factor	Prog 1	Prog 2	Similarity	outcome
size of methods	46	46	100%	Pass
Attributes	14	14	100%	Pass
reserved word	39	39	100%	Pass
Assignment statements	46	46	100%	Pass
If statements	1	1	100%	Pass
For statements	0	0	100%	Pass
While-do statements	2	2	100%	Pass
Case statements	0	0	100%	Pass
Function calls	36	36	100%	Pass

Table 4: The second phase analysis (Entropy)

Entropy Factors	Prog 1	Prog 2	Similarity
Entropy 1	4.01	4.01	100%
Entropy 2	4.83	4.77	98%

Result: Detecting plagiarism

6. REFERENCES

[1] P. Clough, "Plagiarism in Natural and Programming Languages: An overview of current tools and technologies". Internal report, Department of Computer Science, University of Sheffield, 2000 http://www.dcs.shef.ac.uk/~cloughie/papers/Plagiarism.pdf

[2] S. Grier, "A tool that detects plagiarism in Pascal programs", ACM SIGCSM Bulletin, Vol. 13, No. 1, 1981, pp. 15-20.

[3] J.L. Donaldson, L. Ann-Marie, and P.H. Sposato, "A plagiarism detection system", ACM SIGCSE Bulletin, Vol. 13, No. 1, 1981, pp.21-25.

[4] L.J. Edward, "Metrics based plagarism monitoring", The Journal of Computing in Small Colleges, Vol. 16, No. 4, 2001, pp. 253-261.

[5] S.D. Stephens, "Using metrics to detect plagiarism" (Student paper). The Journal of Computing in Small Colleges, Vol. 16, No. 3, 2001, pp.191-196.

[6] A. Aiken, "Measure of software similarity", URL http://www.cs.berkeley.edu/-aiken/moss.html.

[7] M.J. Wise, "YAP3: Improved Detection of similarities in computer program and other Texts", ACM SIGCSE, 1996, pp. 130-134.

[8] L. Prechelt, G. Malpohl and M. Phillippsen, "JPlag: Finding Plagiarisms among a Set of Programs", Technical Report, 2000 http://www.ipd.uka.de/~prechelt/Biblio/Biblio/jplagTR.pdf

[9] D. Gitchell and N. Tran, "Sim: A Utility for Detecting Similarity in Computer Programs", ACM SIGCSE Technical Symposium, Vol. 31, No. 1, 1999, pp. 266-270.

[10] K.L. Verco and M.J. Wise, "Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems", Proceedings of First Australian Conference on Computer Science Education, Sydney, Australia, July 3-5 1996, pp. 81-88.

[11] H.T. Jankowitz, "Detecting plagiarism in student Pascal programs", Computer Journal, Vol. 31, No 1, 1988, pp 1-8.

[12] B. Baker and U. Manber, "Deducing similarities in java sources from bytecode", Proceeding of USENIX Annual Technical Conference, New Orleans, 1998, pp. 179-190.

[13] P. Cunningham and A. Alexander, "Using CBR techniques to detect plagiarism in computing assignments", Proceedings of the First European Workshop on Case-Based Reasoning EWCBR-93, Kaiserslauten, Germany, 1993, 178-183.

[14] C. Daly and J. Horgan, "Patterns of Plagiarism", Proceedings of the 36th SIGCSE technical symposium on Computer science education, 2005, pp. 383-387.

[15] P. Vamplew and J. Dermoudy, "An anti-plagiarism editor for software development courses", Proceedings of the 7th Australian conference on Computing education, Australia, 2005, pp. 83-90.

[16] L.S. Shooman, Software Engineering Design, Reliability and Management, McGraw-Hill Book Company, 1983.

[17] J. Davis and R. LeBlanc, "A Study of the Applicability of Complexity Measures", IEEE transactions on Software Engineering, Vol. 14, No. 9, 1988, pp. 1366-1372.

[18] E.S. Page and L.B. Wilson, Information Representation and Manipulation in a Computer. Cambridge University press, 1973.

Received: Oct. 2005. Accepted: Jun. 2006.