An XML Plagiarism Detection Model for Procedural Programming Languages

Seo-Young Noh, Shashi K. Gadia
Department of Computer Science
Iowa State University
Ames, Iowa, USA
{rsyoung, gadia}@cs.iastate.edu

Abstract

Plagiarism is a common place in academics, especially in courses involving programming. In this paper, XPDec, an XML-based model is introduced to detect similarities among programs that arise under plagiarism. Based upon the syntax of a specific programming language, XPDec uses an XML scheme that is suitable for the detection of plagiarism. XML documents are generated from given program sources and XQuery is used to extract information relevant to the detection of plagiarism. The XML's tree-like representation of query results is exploited to ignore common forms of reordering that arise in plagiarism. The level of similarity between a pair of programs is numerically quantified and reported. The usefulness of XPDec in detection of plagiarism is discussed. XPDec has been implemented, and its architecture is presented.

Keywords

XML Plagiarism detection model, XPDec

1. Introduction

Plagiarism in academic world is a significant problem. Prevention of plagiarism among students is an ongoing struggle for instructors. In spite of such efforts plagiarism continues to be a commonplace. There are many surveys reporting plagiarism. According to a national survey by Donald McCabe, of approximately 6,000 students, 74% of engineering students reported engaging in some form of academic dishonesty, compared with 87% for business majors, and 63% for humanities majors [1]. Currently, many plagiarism detection systems have been developed [2]. Based on the domain that they deal with, the plagiarism detection systems can be classified into two categories: plagiarism detection systems for written text and plagiarism detection systems for software programs. In this paper, we focus on the latter: plagiarism detection systems for programs.

Hamblen and Parker [3] define software plagiarism as follows: A program that has been produced from another program with a small number of routine changes. A simple approach for detection of plagiarism is to compare two programs as a pair of strings using a Unix tool such as diff. But such approach would ignore interesting aspects of plagiarism such as reordering of program segments. A quantum improvement would be to represent the program source into a tree structure that disregards certain common reorderings that do not alter the overall semantics of the programs. A tree structure consists of nodes, and each node can have its child nodes. Because it is up to us to consider child nodes ordered or unordered, generating a tree structure from a programming source has merits for detection of reordering plagiarism. Such ordering criteria can be applied recursively to the entire tree. The problem that we face is how to generate a tree structure from a given program source. Fortunately, a program in languages like C or Pascal is a well-structured document because it has to conform to precise rules in the corresponding language. We use XML to represent such programs. XML gives the program the desired tree structure and at the same time provides us access to various components of the program through its tagging mechanism. We can use the XML based tree structure to find similarities between given program sources.

The organization of the rest of this paper is as follows: in Section 2, we discuss related work. In Section 3, XPDec, our XML based system for the detection of plagiarism, will be introduced. In Section 4, we will discuss the architecture of XPDec system that we have implemented. In Sections 5 and 6, we will discuss the test results and conclusions, respectively.

2. Related Work

In this section, we discuss plagiarism detection systems for programs written in procedural languages such as C or Pascal. A plagiarized program can be defined as a program that has been produced from another program with a small number of routine transformations. Faidhi and Robinson [5] characterize six levels of program modification in plagiarism spectrum as shown in Figure 2.1. The direction from inside to outside represents the difficulty to detect plagiarism.
A number of algorithms to detect plagiarism can be found in the technical literature, and an easy to implement algorithm based on string comparisons is as follows [5]:

- Remove all comments
- Ignore all blanks and extra lines, except when needed as delimiters
- Perform a character string comparison between the two files using Unix utilities, diff, grep, and wc
- Maintain a count of the percentages of characters which are the same

This simple algorithm will detect many cases of plagiarism, and it is easy to implement. According to [6], the majority of students who copy programs change comments, the white space, and a few variable names in the program. It, however, requires a substantial amount of run time. In the following subsections, we will discuss plagiarism detection systems for programming such as SIM, YAP series, and JPlag.

2.1. SIM

SIM (Software Similarity Testor) was developed by Dick Grune at Vrije Universiteit. According to Grune, SIM is able to detect potentially duplicated code fragments in large software projects, in program text, in shell scripts and in documentation, and to detect plagiarism in software projects. SIM adopts a three-phase algorithm. In the first phase, the system reads program files, and prepares a forward-reference table, and determines the set of interesting runs (substrings). In the second phases, it determines the line numbers of the interesting runs. In the last phase, it prints the contents of the runs in order. The SIM consists of five main sub modules as shown in Figure 2.2, and Table 2.1 shows the functions for each module in SIM.

2.2. YAP Series

YAP, which stands for Yet Another Plague, is a series of systems. YAP series are based on the Plague plagiarism detection system. According to Wise [8,9], YAP series is able to detect following plagiarism:

- Changing comments or formatting
- Changing identifiers
- Changing the order of operands
- Changing data types
- Replacing expressions with equivalents
- Adding redundant statements or variables
- Changing the order of independent statements
- Changing the structure of iteration statements
- Changing the structure of selection statements
- Replacing procedure calls by the procedure body
- Introduction of non-structured statements
- Combining original and copied program segments

Basically YAP series detect similarities between given source files by using two phases. The first phase in all
three systems, YAP1, YAP2, and YAP3, is same, and details of steps in the phase are as follows:

- Remove comments and print-strings
- Translate upper-case letters to lower-case
- Remove letters not found in legal identifiers
- Form a list of primitive tokens
- A range of synonyms are mapped to a common form

In the second phase, each system has its different computing algorithm, and Table 2.2 [7] shows the summary of the algorithms adopted in YAP series.

<table>
<thead>
<tr>
<th>YAP Series</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>YAP1</td>
<td>Mixture of Unix utilities and a Bourne-shell script</td>
</tr>
<tr>
<td>YAP2</td>
<td>Heckel’s algorithm [9]</td>
</tr>
<tr>
<td>YAP3</td>
<td>RKR-GST algorithm [9]</td>
</tr>
</tbody>
</table>

2.3. JPlag

JPlag is a system that finds similarities among multiple sets of source code files. JPlag computes program syntax and structures. Current version of JPlag supports Java, C, C++, Scheme, and natural language text. It also can be used to detect stolen software parts among large amount of source text or modules that have been duplicated.

JPlag computes the similarity between a pair of programs in two phases as follows:

1. Generating token strings from source programs
2. Comparing token strings in pairs for determining the similarity of each pair.

In the second phase, each token string can be divided into small substrings called “tiles.” The percentage of the token streams that can be covered is the similarity value between two program source files. In JPlag system, the similarity between two source files, \( A \) and \( B \), is computed as follows [10]:

\[
\text{Sim}(A, B) = \frac{2 \times \text{Coverage}(\text{tiles})}{|A| + |B|}
\]

\[
\text{Coverage}(\text{tiles}) = \sum \text{length of } \text{match}(a, b, \text{length of } \text{tiles})
\]

where \( |A|, |B| \) : the lengths of token string of file \( A \) and \( B \)

\[
\text{match}(a, b, \text{length of } \text{tiles}) : \text{identical substrings } a \text{ and } b \text{ with length } n
\]

3. XML Plagiarism Detection Model

In this section, XPDec\(^2\), our XML plagiarism detection model is introduced. XPDec consists of five main parts: generating an XML document from a source program, generating a tree structure from a generated XML document, generating a decimal frame matrix from the generated tree structure, extracting control sequences (or constructing decimal control matrix) from the XML document using XQuery query, and calculating a similarity from the generated matrices.

3.1. Generating an XML Document

The plagiarism detection model suggested in this paper is based on general procedural programming languages. First of all, we need to take a closer look at the structure of procedural programming languages. Most of the procedural programming languages consist of three main structure blocks: Headers block, Global Variables block, and Functions block as shown in Figure 3.1 Function block in Functions block may have many nested blocks, and a nested block can also have structure blocks as its child blocks.

![Figure 3.1: Block diagram of a general procedural programming language](image)

In order to find transformation rules between XML and the structure of procedural programming languages, each substructure should be restructured into a tree structure because every tree structure can be transformed into its equivalent XML document. The outer-most block in Figure 3.1 represents a program source, and it consists of three main structure blocks. Therefore, the tree structure for a given source program consists of four nodes in a tree (1: root node, 3: child nodes) as shown in Figure 3.2.

\(^2\) In this paper, we use the term, XPDec, as XML plagiarism detection model and the system implemented based on the model interchangeably.
Figure 3.2: A tree structure of a program source

Each node in Figure 3.2 can be extended until it does not contain internal nodes. Headers node in Figure 3.2 is extended as shown in Figure 3.3. As seen, Headers node may have many leaf nodes which represent header information in a given program source.

Figure 3.3: A tree structure of Headers

Figure 3.4 shows an extended Global Variables node in Figure 3.2. Like Headers node, it can be made up of several leaf nodes without internal nodes. Each leaf node has information on a global variable defined in a given program source.

Figure 3.4: A tree structure of Global Variables

The last child node of Program Source node is Functions node. Functions node consists of Function nodes which represent individual functions in a given program source. The basic tree structure of Functions node is shown in Figure 3.5.

Figure 3.5: A tree structure of Functions in a program source

As seen, each Function node is made up of four child nodes: Return Type, Name, Arguments, and Blocks. The node for Return Type has information on a return type of its parent node. Name node contains information on the name of a function in a program source. The third node in Function node represents argument lists appearing in the function. It is possible for Arguments node not to have any child node. The last child node of Function node is Blocks node. The basic structure of Blocks node is depicted in Figure 3.6.

Figure 3.6: A tree structure of Blocks

Blocks node has only one child node—Block. Block node is made up of three different nodes: Local Variables, Contents, and Control nodes. Local Variables node is same as Global Variables node. Each leaf node in Local Variables represents a local variable defined in a block of a function. Contents node contains information on assignment statements, function calls, individual statements in a block of a function, and so forth. As seen, it might contain a Block node. Control node is made up of Control Type and Block nodes, and it is possible to be an empty node meaning there is no control statement in a block. Control node should have exactly one Control Type node, containing information on a control type appearing in the block. The last child node is Block node. Since a block in Control node might have a nested block, Block node may contain Block node itself.

Figure 3.7: An equivalent XML of Figure 3.2

In order to generate an XML document for a program source, we need to have a mapping between a tree structure for a programming language and an XML document. As we have seen in Figure 3.2, Program
Source node has three different its child nodes. The tree structure can be transformed into an equivalent XML document as shown in Figure 3.7.

The equivalent XML document contains the exactly same information that the tree structure has. The frame of an XML document is made up of three elements: <headers>, <globalvariables>, and <functions> as same as its equivalent tree structure. Each element in the XML document can be extended. Figure 3.8 shows the extended XML documents for <headers> and <globalvariables>.

Figure 3.8: An equivalent XML of Figure 3.3 and Figure 3.4

In Figure 3.8, Headers node in the tree structure is mapped to <headers> element. Each leaf node in Headers node is mapped to <header> element, and the mapped element has a child element named <name> containing the name information that the leaf node has. Global Variable node can be mapped in the same way like Headers node. <functions> elements can be extended in the same way from Functions sub tree in Figure 3.5.

3.2 Generating a Tree Structure

The XML document generated from a given program has a natural tree structure. However the XML document is a text document that is not lend itself for tree-based algorithms. Therefore the next step is to represent it as a tree. It should be clear that the transformation of a program to an XML document to a tree is reversible. This means that the information content of a program, the XML document, and the tree is the same. Figures 3.9 show the part of a tree resulting from a C program shown in Appendix.

3 For the space reason, we omit the detailed explanation on <functions> element, please refer to [11] for more information.

Figure 3.9: A tree structure from an XML document generated from a program source

References:
A: XMLRoot  G: type  M: blocks
B: headers  H: functions  N: block
C: header  I: function  O: localvariables
D: name  J: returntype  P: contents
E: globalvariables  K: arguments  Q: controls
F: variable  L: argument

3.3 Generating a Decimal Frame Matrix

After generating a tree structure from an XML document, the next step in XPD is to generate a decimal frame matrix for the tree structure. Figure 3.10 depicts the relationship between a decimal frame matrix (DFM) and a tree structure generated from an XML document. The reason that XPD transforms the tree structure into a matrix is to provide convenient way to calculate the similarity between a pair of program sources. Once we generate matrices, we can apply arithmetic operations over the matrices. It, however, should be clear that generating matrices is not for general matrix operations. The number of columns in the matrix might be different from a procedural programming language to a language. We assume that is large enough.

In the decimal frame matrix, the first row is the information on headers node in the tree structure. Mapping relationship between headers node and the first row in the matrix is shown in Table 3.1.
For example, suppose that the tree structure contains six headers, \( h_{dr_3}, h_{dr_{10}}, h_{dr_5}, h_{dr_{n-1}}, h_{dr_{11}}, \) and \( h_{dr_n} \).

The decimal sequential representation for these headers is shown below.

\[
0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, \ldots, 1, 1
\]

It says that of all possible headers in a procedural programming language, header numbers 3, 5, 10, 11, n-1, and n are present in the source program.

The decimal sequential representation for global variables node in the tree structure is the second row of the matrix shown in Figure 3.10. The way of generating the decimal sequential representation for the global variables node is the same as that of headers node. The mapping between global variables node and the matrix is shown in Table 3.2.

Table 3.2: The mapping relationship between global variables node and a decimal sequential representation

<table>
<thead>
<tr>
<th>Position</th>
<th>( d_{hl} )</th>
<th>( d_{h2} )</th>
<th>( \ldots )</th>
<th>( d_{hn-1} )</th>
<th>( d_{hn} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Name</td>
<td>( g_v_1 )</td>
<td>( g_v_2 )</td>
<td>( \ldots )</td>
<td>( g_v_{n-1} )</td>
<td>( g_v_n )</td>
</tr>
</tbody>
</table>

For instance, when global variables node has six child nodes \( g_v_n, g_v_3, g_v_4, g_v_2, g_v_{10}, \) and \( g_v_{n-1} \), an instance of the decimal sequential representation will be as follows:

\[
0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, \ldots, 1, 1
\]

Next two rows in the matrix show the information on the first function node which is the first child node of functions node. A decimal sequential representation for a function node consists of two rows. The first row contains the information on a return type and argument types. The second row contains the information on local variable and control type information. Tables 3.3 and 3.4 show the mapping for return types and argument types, and the mapping for local variable types and control types, respectively.

Table 3.3: The mapping for return types and argument types in decimal sequential representations

<table>
<thead>
<tr>
<th>Position</th>
<th>( d_{c1} )</th>
<th>( \ldots )</th>
<th>( d_{ck} )</th>
<th>( \ldots )</th>
<th>( d_{cn-1} )</th>
<th>( d_{cn} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( r_{1} )</td>
<td>( \ldots )</td>
<td>( r_{k} )</td>
<td>( a_{1} )</td>
<td>( \ldots )</td>
<td>( a_{n-k-1} )</td>
</tr>
</tbody>
</table>

Table 3.4: The mapping for local variable types and control types in a decimal sequential representation

<table>
<thead>
<tr>
<th>Position</th>
<th>( d_{c1} )</th>
<th>( \ldots )</th>
<th>( d_{ck} )</th>
<th>( d_{ck+1} )</th>
<th>( \ldots )</th>
<th>( d_{cn-1} )</th>
<th>( d_{cn} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( l_{1} )</td>
<td>( \ldots )</td>
<td>( l_{k} )</td>
<td>( c_{1} )</td>
<td>( \ldots )</td>
<td>( c_{n-k-1} )</td>
<td>( c_{n-k} )</td>
</tr>
</tbody>
</table>

For example, suppose that a return type is \( r_{t_2} \), argument types are \( a_{t_2}, a_{t_3}, \) and \( a_{n-k} \), where type \( a_{t_2}, a_{t_3}, \) and \( a_{n-k} \) appear 2, 3, and 4 times, respectively, and suppose that there are \( l_{t_2} \), \( l_{t_7} \), and \( l_{t_k} \) as local variables, where each variable appears in the function 2, 3, and 7 times, respectively. Finally, \( c_{t_1}, c_{t_2}, \) and \( c_{n-k-1} \) exist, and occurrences are 1, 1, and 4 times, respectively. The decimal sequential representations for the function having this information will be generated as follows:

\[
\begin{array}{c|c|c}
\hline
\text{Return type} & \text{Arguments} & \text{Control types} \\
\hline
010000000 & 00 & 0230000000000000 \ldots 04 \\
020000050 & 07 & 1100000000000000 \ldots 40 \\
\hline
\end{array}
\]

3.4. Extracting Control Sequences from an XML Document by using XQuery

In this section, the method for extracting control sequences from an XML document by using XQuery [12] will be discussed. In section 3.3, we have extracted information on headers, variables, and functions. Even though the decimal frame matrix on functions contains some control information used in some functions, it is not good enough to detect plagiarisms because the control structure can vary from one program to another. In order for XPDec to be more informative, we have to deal with control sequences appearing in functions. To do this, XPDec extracts control sequences from an XML document by using XQuery.

In the XML plagiarism detection model, the XQuery shown in Figure 3.11 is used to extract control sequences from an XML document. The XQuery defines a function, control_summary, to extract control types. This function is recursively called in \(<controlsequence>\) element. The result of the XQuery is also an XML document starting with \(<xqueryresult>\), and ending with \(<xqueryresult>\) tags. The first FLWR expression [12] in \(<xqueryresult>\) element extracts \(<name>\) element that is a child element of \(<function>\) element. After extracting a name element of a function, the XQuery starts extracting control sequences as recursively calling control_summary function. Each control type is wrapped with \(<control>\) and \(</control>\) tags, and \(<control>\) element with control type will be returned to the calling function. The control_summary is like table of contents of a
document; in extracting the summary, it preserves the sequence and nesting of the control structure.

```xml
DEFINE FUNCTION control_summary(ELEMENT $f) RETURNS ELEMENT

  <content>
    FOR $i IN $f/controlType
      RETURN
      | <type>$(i)/</type>
      | control_summary($i)
  </content>
</queryread>
```

**Figure 3.11:** An XQuery to extract control sequences from an XML document

As the final step of this section, XPDec generates information on functions. Each control type has its unique decimal number as shown in Table 3.5. The unique decimal number given to each control type depends on the control types sequence for a given function. Each control number is assigned to a slot in the array in order of the control types sequence shown in Table 3.6.

<table>
<thead>
<tr>
<th>Control Type</th>
<th>$ct_1$</th>
<th>$ct_2$</th>
<th>…</th>
<th>$ct_{n-1}$</th>
<th>$ct_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec. Number</td>
<td>1</td>
<td>2</td>
<td>…</td>
<td>$n-1$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Suppose that the control sequence in a function contains $ct_1$, $ct_2$, $ct_{m-k}$, $ct_1$. The decimal sequential representation for this control sequence will be a decimal sequence shown in Table 3.6.

<table>
<thead>
<tr>
<th>Slot Number</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
<th>…</th>
<th>$n-1$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec. Number</td>
<td>1</td>
<td>2</td>
<td>m-k</td>
<td>1</td>
<td>…</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Control Type</td>
<td>$ct_1$</td>
<td>$ct_2$</td>
<td>$ct_{m-k}$</td>
<td>$ct_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As seen in the table, the first four slots are assigned to the control types based on the size of the programming language. In order to generate a decimal number given to each control type, we can build a decimal control matrix (DCM) combining all decimal sequential representations.

### 3.5. Similarity Calculation

Above we have shown how to transform a program to an XML document, and document to a tree. Then we discussed how to generate a decimal frame matrix for the tree structure, and extract decimal control matrix from an XML document by using an XQuery. XPDec combines two decimal matrices into an integrated decimal matrix (IDM). An IDM might be depicted as shown in Figure 3.12:

**Figure 3.12:** Integrated decimal matrix from DFM and DCM

The operation, $OP(IDM)$, is defined as follows:

<table>
<thead>
<tr>
<th>$CSMatrix_{[m,n]}$</th>
<th>Control Sequence Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PFMatrix_{[2m+1,n]}$</td>
<td>Program Source Matrix</td>
</tr>
<tr>
<td>$IDMMatrix_{[2m+2,n]}$</td>
<td>IDM Matrix</td>
</tr>
</tbody>
</table>

```
int p = 3 /* row position of IDMMatrix */
IDMMatrix[2m+2, n] = null
MatrixCopy(PFMatrix[1, 2, 1 n], IDMMatrix[1, 2, 1 n])
for i = 1 to n
  u = i + j + 2 /* second row of FSMatrix */
  v = i + j + 1 /* first row of FSMatrix */
  MatrixCopy(PFMatrix[u, v, 1 n], IDMMatrix[u + 2, v, 1 n])
endfor
```

The right hand side matrix shown in Figure 3.12 after applying $OP(IDM)$ operation represents a more detailed structure of IDM matrix. As we have discussed in
previous sections, the notations, \( h_i \) and \( g_i \), are a header and a global variable whose position is \( i \), respectively. The first two rows represent the information on headers and global variables in a given source program. The third, fourth, and fifth rows contain the information on the first function of a decimal frame matrix and a decimal control matrix. In the third row, \( r_{i_1}, r_{i_2}, \ldots, r_{i_k}, a_{i_1}, \ldots, a_{i_{n-k}} \), the first \( k \) elements are the information on the return type of the first function in the given source program, and the second \( n-k \) elements are the information on the argument types in the function, respectively. In the fourth row, \( l_{i_1}, l_{i_2}, \ldots, l_{i_k}, c_{i_1}, \ldots, c_{i_{n-k}} \), the first \( k \) elements contain the information on the local variables in the first function, and the second \( n-k \) elements contain the information on the control types. In this row, the information on the control types is only the number of controls appearing in the function. In the fifth row, \( cd_{i_1}, cd_{i_2}, \ldots, cd_{i_k} \), \( n \) elements represent the information on the sequence of controls in a decimal control matrix. The rest of rows in the IDM matrix represent the information on the rest functions as the same as the way of the first function.

Since we can generate an IDM matrix from a tree structure, we can also apply arithmetic calculations to get similarities between two given trees. If there exist two tree structures to be compared, each tree can be seen as an IDM. In Figure 3.13, the program sources A and B are represented as \((m+2) \times n\) and \((p+2) \times n\) matrices, respectively. We can think that finding the similarity of the headers is to find how many same headers appear in both A and B program sources.

\[ \begin{array}{ccccccccccccc} h_1 & h_2 & \ldots & h_k & h_{k+1} & \ldots & h_n \\ a_1 & a_2 & \ldots & a_k & a_{k+1} & \ldots & a_n \\ c_1 & c_2 & \ldots & c_k & c_{k+1} & \ldots & c_n \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \ldots & a_k & a_{k+1} & \ldots & a_n \\ c_1 & c_2 & \ldots & c_k & c_{k+1} & \ldots & c_n \\ \end{array} \]

Figure 3.13: IDM matrices for two given program sources, A and B

Therefore, the similarity of headers depends on the result of the multiplication of the first row of IDM matrix A and the transpose of the first row of IDM matrix B as shown in Figure 3.14. In this paper, we name the result \( \text{HeaderVal} \).

\[
\text{HeaderVal} = A[1][1..n] \times B^T[1][1..n]
\]

\[
= \left( h_1, h_2, \ldots, h_{n-k}, h_n \right) \times \left( h'_1, h'_2, \ldots, h'_{n-k}, h'_n \right)^T
\]

\[
= \sum_{i=1}^{n} h_i \cdot h'_i
\]

\[
= \sum_{i=1}^{n} h_i \cdot h'_i = 1 \quad \text{if } h_i = h'_i = 1
\]

\[
= 0 \quad \text{otherwise}
\]

Figure 3.14: Header value (\( \text{HeaderVal} \)) between IDM matrix A and B

As applying the strategy of getting the header value, \( \text{HeaderVal} \), to get the intermediate value for global variables, we can calculate the value for global variables since it also depends on the result of the multiplication of the second row of IDM matrix A and the transpose of the second row of IDM matrix B. Figure 3.15 shows the way to calculate the \( \text{GlobalVal} \) for global variables between two IDR matrices A and B.

\[
\text{GlobalVal} = A[2][1..n] \times B^T[2][1..n]
\]

\[
= \left( g_1, g_2, \ldots, g_{n-k}, g_n \right) \times \left( g'_1, g'_2, \ldots, g'_{n-k}, g'_n \right)^T
\]

\[
= \sum_{i=1}^{n} g_i \cdot g'_i
\]

Figure 3.15: Global value (\( \text{GlobalVal} \)) between IDR matrix A and B

When trying to get similarities between functions, we need to carefully apply this strategy since we cannot guarantee whether the first function of IDM matrix A has any relationships with the first function of IDM matrix B. Therefore, to find similarities between functions, every function in IDM matrix A should be compared with every function in IDM matrix B. Each result will be various depending on the contents of the rows of matrices. The method in XPDec chooses the maximum value, and considers it as an intermediate value before calculating the similarity for the functions.

In addition to the intermediate values introduced above, \( \text{RetVal}, \text{ArgVal}, \text{LocVal}, \text{CtrlVal}, \text{and CtrlSeqVal} \), representing intermediate values for a return value, argument types, local variables, control types, and control sequences, are defined as shown in Figure 3.16.

\( \text{RetVal} \) is the maximum value among multiplications of return elements in IDM matrix A and the transpose of return elements of IDM matrix B with \( k \) elements. The way to get the intermediate values for similarities is similar to the way of calculating each intermediate value except \( \text{CtrlSeqVal} \). \( \text{CtrlSeqVal} \) is an intermediate value for the similarity of control sequence. Since each control sequence contains a decimal sequence representing the order of control types, the similarity between two given control sequence is calculated as comparing two decimal
numbers whose positions are same in the sequences. If the control types, $cd_{ik}$ and $cd_{jk}$ at the $k$th position in IDM matrices $A$ and $B$ are same, the difference $cd_{ik} - cd_{jk}$ should be 0. In the other hand, if two values are similar, for example, for and while, but they are not same, then the difference $cd_{ik}$ and $cd_{jk}$ will be close to 0. Therefore, if two control sequences are totally different, $CtrlSeqVal$ will be a large number.

$$
\text{RetVal} = \max \left\{ \{1, \ldots, 4\} \mid |A| \cdot |B| + |3\{1, \ldots, 4\}\} \cdot [1, \ldots, 4], \quad 1 \leq a \leq m, 1 \leq b \leq \frac{P}{3} \right\} 
$$

$$
\text{ArgVal} = \max \left\{ \{1, \ldots, 4\} \mid 1, \ldots, 4\} \cdot |B| + |3\{1, \ldots, 4\}\} \cdot [1, \ldots, 4], \quad 1 \leq a \leq m, 1 \leq b \leq \frac{P}{3} \right\} 
$$

$$
\text{LocVal} = \max \left\{ \{1, \ldots, 4\} \mid 1, \ldots, 4\} \cdot |B| + |3\{1, \ldots, 4\}\} \cdot [1, \ldots, 4], \quad 1 \leq a \leq m, 1 \leq b \leq \frac{P}{3} \right\} 
$$

$$
\text{CtrlVal} = \max \left\{ \{1, \ldots, 4\} \mid 1, \ldots, 4\} \cdot |B| + |3\{1, \ldots, 4\}\} \cdot [1, \ldots, 4], \quad 1 \leq a \leq m, 1 \leq b \leq \frac{P}{3} \right\} 
$$

$$
\text{CtrlSeqVal} = \max \left\{ \{1, \ldots, 4\} \mid 1, \ldots, 4\} \cdot |B| + |3\{1, \ldots, 4\}\} \cdot [1, \ldots, 4], \quad 1 \leq a \leq m, 1 \leq b \leq \frac{P}{3} \right\} 
$$

Figure 3.16: Intermediate values between IDM matrix $A$ and $B$ for a function

These intermediate values, however, are ranged from 0 to an unknown bound. Therefore, the XPDec normalizes these intermediate values as ranging them from 0 to 1. Figure 3.17 shows the normalized intermediate values.

In order to get the similarity between two IDR matrices $A$ and $B$, the XPDec combines the normalized intermediate values as shown in Figure 3.18.

$$
\text{sim} = (W_0 \cdot \text{NormRetVal} + W_1 \cdot \text{NormArgVal} + W_2 \cdot \text{NormLocVal} + W_3 \cdot \text{NormCtrlVal}) \cdot \text{NormCSVall} \cdot \text{NormCtrlVa} \cdot \text{NormLocVal} \cdot \text{NormArgVal} \cdot \text{NetVal} \cdot \text{NormRW} \cdot \text{NormGlbVal} \cdot \text{NormHdrVal} 
$$

Figure 3.18: Similarity between two IDM matrix $A$ and $B$

As seen in Figure 3.18, the final similarity is calculated as the summation of the normalized intermediate values for headers, global variables, and so on. Even though each calculated similarity is significantly used to detect plagiarism, the level of importance will vary from a similarity to a similarity. For example, a similarity of functions should be considered as more important similarity compared with that of headers. Therefore, each similarity can have its weight value $w_i$.

4. System Architecture

![System Architecture Diagram](image)

Figure 4.1: System architecture of XPDec system

The implementation of XPDec, the XML plagiarism detection model, consists of primary three main layers: Tagging & Treeing Layer (Layer 1), XQuerying &
Decimalization Layer (Layer 2), and Analyzing Layer (Layer 3). Figure 4.1 depicts the system architecture of XPDec system with three layers.

4.1. Tagging& Treeing Layer (Layer 1)
Tagging& Treeing Layer is made of three subsystems: Tagging& XML Generating System, Tree Generating System, and Tree Restructuring System. The program source is passed to Tagging& XML Generating System as an input to generate an XML document for the given source program. When an XML document is generated for the given source program, the Tagging& XML Generating System references a dictionary-Tagging Info. The dictionary contains tagging information such as mapping relationship between types in a program language and names of tags. After tagging the program source using XML notations based on Tagging Info dictionary, it generates an XML document. In the XML document, all comments in the program source are removed to make the document lighter. Removing all comments in the program sources helps the XPDec system to deal with lighter program sources without losing the ability to detect plagiarisms in computer programming. The generated XML document is used as an input of the Tree Generating System. Tree Generating System generates a tree structure for the given XML document. In the XPDec system, the tree structure generated by Tree Generating System is called FDT (First Draft Tree). Even though FDT contains whole information on the XML document, it is not efficient to deal with FDT directly because of many redundancies of tags. Therefore, the XPDec system modifies the FDT. The subsystem that modifies the FDT is Tree Restructuring System. Tree Restructuring System receives an FDT from the upper subsystem, and restructures the FDT to RT (Restructured Tree). When the subsystem modifies the FDT, it uses Restructuring Info dictionary containing the modification information such as which nodes should be combined and removed without losing any semantics of the FDT. The purpose of the Tagging& Treeing Layer in the XPDec system is to generate an optimized tree structure from an XML document containing information on the given source program. The optimized tree structure is used as an input of XQuerying& Decimalization Layer.

4.2. XQuerying& Decimalization Layer (Layer 2)
The second layer of the XPDec system is XQuerying& Decimalization Layer. This layer consists of five distinct subsystems: XML Regenerating System, XQuery System, DFM Generating System for an RT, DCM Generating System for Control Sequence, and IDM Matrix Generating System. In the XQuerying& Decimalization Layer, two dictionaries are used to provide information to two decimal matrix generating systems.

The RT from the upper layer will be provided to two subsystems as an input. XML Regenerating System receives the RT from the upper layer, and regenerates an XML document for the RT. The regenerated XML document is also equivalent to the RT, and it contains exactly same information that the original XML document has. The other subsystem, DFM Generating System for the RT, receives the RT, and extracts a decimal frame matrix for the RT. The decimal frame matrix contains the information that we discussed in section 3. The result of the subsystem is integrated with the result of DCM Generating System for Control Sequence. The DFM Generating System for RT uses a dictionary named Decimal Format Info. This dictionary contains the information on decimal format such as which positions represent global variables and the relationship between types and positions. The third subsystem is XQuery System. What XQuery System does in this layer is to extract control sequences from the result provided by XML Regenerating System. XQuery System gives an XML query discussed in section 3 over the modified XML document. It returns the query result to DCM Generating System for Control Sequence. The result from the XQuery System is also an XML document. Since the result from XQuery System is an XML document, it can be also generated to its equivalent tree structure. As the DFM Generating System for RT generates a decimal frame matrix for the RT, DCM Generating System for Control Sequence generates a decimal control matrix for the XQuery result based on Sequence Format Info dictionary.

The results from two matrix-generating systems are used in the IDM Matrix Generating System as inputs. As discussed in section 3, a decimal representation for an RT (or program source) is a \((2m+2) \times n\) matrix, where \(m\) is the number of functions, and a decimal control matrix for control sequences for the XQuery results (or functions in a given source program) is a \(m \times n\) matrix. IDM Matrix Generating System integrates two given matrices into an IDM matrix containing two matrices. Therefore, the result of this layer will be an IDM matrix.

4.3. Analyzing Layer (Layer 3)
Analyzing Layer consists of Matrix Analyzing System and Weight Info. Matrix Analyzing System calculates the similarities between the IDM matrix and target IDM matrices by using several formulas that were discussed in the previous section. When calculating the similarities, it refers to Weights Info dictionary. Weights Info dictionary maintains the weights information on similarities such as headers similarity, global variables similarity, functions similarity, and so on.

5. Result
In this section, the results generated by XPDec system will be discussed. In order to validate the XPDec system, 15 program sources are tested using XPDec system. Those program sources are modified versions of an origin source. In the test, 6 different categorized tests are used as shown in Table 5.1.

Table 5.1: Test categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category 1</td>
<td>Reordering</td>
</tr>
<tr>
<td>Category 2</td>
<td>Comment Changing</td>
</tr>
<tr>
<td>Category 3</td>
<td>Unnecessary Headers' Addition</td>
</tr>
<tr>
<td>Category 4</td>
<td>Function Name Changing</td>
</tr>
<tr>
<td>Category 5</td>
<td>Variable Name Changing</td>
</tr>
<tr>
<td>Category 6</td>
<td>Unnecessary Statements' Addition</td>
</tr>
</tbody>
</table>

In the first 5 categorized tests, XPDec system identified all plagiarized program sources in the categories as identical program sources compared with the origin program source. Figure 5.1 shows the results. As we can see, the test results for category 1 through category 5 indicate all 1. It means that XPDec system considers all program sources in these categories are identical to each other.

![Figure 5.1: Results for category 1 through category 5](image)

The reason that XPDec system identifies all source programs in these 5 categories as identical source programs is because of the detection rules adopted in XPDec system. Since XPDec system uses an XML document, and the document is used to generate a tree structure. The tree structure does not consider ordering. For the comment changing plagiarism, XPDec system does not keep all comments in a given source program, that is, it removes all comments. Therefore, those two program sources are considered as identical. For the unnecessary headers’ addition, XPDec system uses minimum base header plagiarism detection rule that is different from the other detection rules. If one program source contains many headers and the headers of the other program source are subset of the first source program, XPDec system considers that these two header formations are identical. For the function name changing, XPDec system does not contain any names for the functions. It only considers the prototype of each function such as the return type, argument types, local variable types, control sequences and so on. Therefore, even though the names of two functions are different, the two functions are considered as identical functions if the prototypes are identical. For variable name changing, XPDec system considers the variable-changed program sources as identical programs regardless of the changing percentage because XPDec system is not interested in the names of variables or functions. Instead, it is interested in the types that specify the variables or functions.

Figure 5.2 shows the result for Category 6 test. In this category test, unnecessary statements are added to the origin source program. The added statements include some local variables, global variables, general program statements such as printing statements, and so on. As we can see in the figure, XPDec system gives good results for the test. As 5% statements are more added to the origin source program, the detection rate only decreases by at most 4%. The average detection rate decreases by only 2% as 5% additional statements are added to the origin program source. The results for the category 5 might be dependent on what kinds of statements have been added to the origin statements because additional control sequences can affect the results compared with the general programming statements. But, from the programming viewpoints, different control sequences may be considered as different program source and most plagiarized program sources are not beyond the changing control sequence level as discussed in section 2.

![Figure 5.2: Results for category 6](image)

6. Conclusions

In this paper, XPDec, an XML based plagiarism detection model and its implementation have been presented. Because of XML, XPDec has many merits of a
tree structure. Reordering plagiarism, for example, can be easily detected by simply disregarding ordering of nodes. XPDec preserves important attributes used in a program source such as global variables, local variables, control types, and so on. Those attributes are used in constructing the decimal frame matrix for the XML document generated from a given program source. For the structural metrics, XPDec extracts prototypes for each function, instead of preserving variable strings. Since the prototypes may not be changed significantly in a plagiarized program source, comparing prototypes for their structures captures important similarities between the programs. The prototypes are elements in the decimal frame matrix of the XML document. In order to emphasize the use of similar variables and functions on one hand, and structural mechanisms on the other, XPDec maintains the decimal matrix for attribute-counting metrics and structural metrics.

For detection of plagiarism in control sequences, currently XPDec takes a simple approach. The issue of detection of plagiarism in control sequences is inherently very difficult. To determine if two programs with different control sequences compute the same functions is obviously an unsolvable problem. The question is what differences are to be absorbed under the banner of plagiarism? A good approach to this will be to develop some pattern based rules of thumb and integrate them in plagiarism detection systems. Due to its XML based underpinnings, XPDec should be quite amenable to such extensions. In XPDec, dealing with control sequences is a two step process. First, an XML query is used to extract control sequences from each function. This idea has been deployed in Section 3 and its usefulness has been shown in the previous section. In the second step, some aggregate information is incorporated into the decimal matrix. It is likely that both steps will need some extensions. The XQuery query should perhaps extract a more fine grained version of the control sequence that includes some information about variable assignment so that a more accurate aggregation can be done that better reflects the extent of plagiarism.

7. References

8. Appendix
#include <stdio.h> /* stdio.h */
#include <file.h> /* file.h */
t int abc; /* global variable */
void main(int t, int j, char g)
{
 int k;
 summation(); /* call a function */
multiply(); /* call a function */
 priprintf(“Hello “);
}
bool summation(int k, char c) /* function summation */
{
 ....
}
bool multiply(int k, char c) /* function multiply */
{
 ....
}