

A Lightweight Program Similarity Detection Model using XML and Levenshtein Distance

Seo-Young Noh
Department of Computer Science
Iowa State University
Ames, IA, USA
rsyoung@cs.iastate.edu

Sangwoo Kim
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA, USA
skim1@andrew.cmu.edu

Cheonyoung Jung
School of Computer Science & Multimedia
Hyecheon College
Daejeon, Korea
cyjung@hcc.ac.kr

Abstract

Program plagiarism is one of the most significant problems in Computer Science education. Most common plagiarism includes modifying comments, reordering statements, and changing variable names. Such simple changes, however, require excessive string comparisons. This paper presents a lightweight program similarity detection model. Unlike other detection models, our model avoids globally involved string comparisons. String matching is only involved locally when comparing control sequences. To this end we use XML and Levenshtein distance algorithm. The XML's tree-like representation reduces intensive string comparisons for the simple modifications. Levenshtein distance algorithm makes our model reliable for logic changes. Our approach is based on the XPDec model and is capable of distinguishing a flat structure from a nested structure of control sequences. Such improvement will lead to simple and reliable implementation of program similarity detection systems.

Keywords: Computing Ethics, Program Similarity Detection Model, Program Similarity Checker, Levenshtein Distance

1 Introduction

The general concept of plagiarism is defined as activities of reproducing someone else's work without acknowledging or mentioning the source. Plagiarism is a significant problem in the academic world and prevention of plagiarism among students is an ongoing struggle for instructors.

Plagiarism in the computer education (or simply computer programs) is somewhat different from the general definition of plagiarism because students can use the same variable types, similar names for them, or similar logics for a given problem. Parker and Hamblen[1] define software plagiarism as follows:

a program that has been produced from another program with a small number of routine changes.

Faidhi and Robinson[2] have characterized six levels of program modification in plagiarism spectrum such as comments, identifiers, variable position, procedure combination, program statement, and control logic. Parker and Hamblen[1] have pointed out that "the majority of students who copy programs change comments, the white space, and a few variable names in the program".

Since not many logics are changed from source programs, pattern matching or string sequence matching algorithms may be used for checking suspected program sources. However, such approaches would ignore interesting aspects of plagiarism such as reordering of program segments. Reordering plagiarism is simple, but it is difficult to detect changes of sequence orders by using string matching algorithms solely.

This reordering problem can be elegantly resolved by adapting tree properties because a tree structure disregards certain common reordering that does not alter the overall semantics of the programs. Since we may consider child nodes ordered or unordered, generating a tree structure from a programming source has merits for detection of reordering plagiarism.

Our approach extends the XPDec model (XML Plagiarism Detection Model)[3] which can provide simi-

larity between two given procedural source programs. One shortcoming of the XPDec is that it lacks of ability of distinguishing a flat structure from a nested structure of control sequences.

The EXPDec (Extended XPDec) overcomes the problem of the XPDec for distinguishing a flat structure from a nested structure of a control sequence. It uses tree structures for determining the skeleton of program sources and a string matching algorithm for comparing control sequences. We use XML to transform a given source program to a tree structure. Despite the fact that a program source itself is a tree, transforming it to an XML document provides many benefits, for example, we can use XQuery (XML Query Language)[4] to extract specific information from the XML document. For control sequence comparisons, we use Levenshtein distance algorithm[5]. This algorithm is able to count the differences between two control sequences not only when two control types are different in the same length, but also when two sequences are in the different lengths. This improvement of EXPDec will lead to simple and reliable implementation of program similarity detection systems. We hope that our lightweight program similarity detection model benefit the community of computer science education or related fields.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we will introduce the EXPdec detection model. In Sections 4, we will discuss an implementation methodology. In section 5, we will conclude this paper with our observations.

2 Related Work

There are many different types of plagiarism detection models and systems. Plagiarism detection systems can be categorized into two groups. Plagiarism detection systems for written texts and detection systems for computer programming. Clough[6] surveyed these systems and their base detection models.

In this section, we will briefly introduce program plagiarism detection systems such as SIM, MOSS, SID, and YAP.

SIM (Software Similarity Tester)[7] has been designed to detect potential duplicated code fragments in large software projects. SIM uses a three-phase algorithm: 1) preparing a forward reference table and determining the set of interesting substrings, 2) determining the line numbers of the substrings, and 3) printing the substrings in orders.

Aiken[8] has developed MOSS (Measure Of Software Similarity). MOSS is an automatic system for determining the similarity of programming languages such as C, C++, and Java. MOSS uses an algorithm based on code-sequence matching. However, it is hard to find explicit information about the algorithm.

The Bioinformatics groups at University of California, Santa Barbara and University of Waterloo have developed SID (Shared Information Distance)[9]. The base algorithm of SID was used to compare the similarity of genomes. They extended Kolomogorov complexity to measure the distance between two program sources.

YAP (Yet Another Plague)[10] is a series of systems. YAP series are based on the Plague plagiarism detection system. The procedure consists of multiple steps such as removing comments and print-strings, translating upper-case letters to lower-case, removing letters not found in legal identifiers, forming a list of primitive tokens, mapping a range of synonyms to a common form, and applying computing algorithms.

3 EXPDec Model

The EXPDec detection model consists of five steps: 1) generating an XML document from a given source file, 2) generating a frame matrix, 3) extracting control sequences from the XML document, 4) creating a control matrix, and 5) measuring the similarity by using a frame matrix and a control matrix.

3.1 XML Representation

In general, procedural programming languages consist of three main structure blocks: *Headers*, *Global Variables*, and *Functions* blocks. These blocks may consist of multiple headers, global variables, or functions in a corresponding block. Figure 1 shows a partial tree of *Function* block that consists of multiple functions. A single function consists of four elements such as a *return type*, a *function name*, *argument(s)*, and *blocks*.

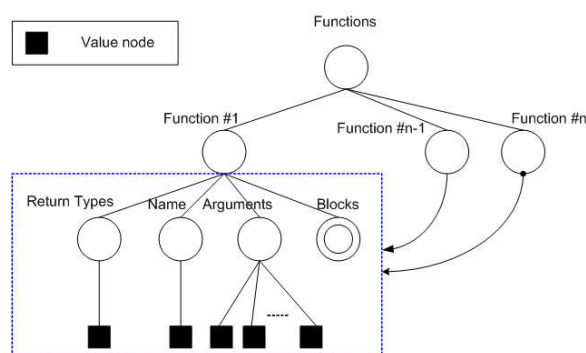


Figure 1. A tree structure of Functions[3]

Figure 2 shows the structure of *Blocks*. *Blocks* node has a single child called *Block* that consists of three child nodes: *Local Variables*, *Contents*, and *Control* nodes. The most important node is the *Control* node. It consists of *Control Type* and a *Block* node. We

must note that a *Block* node may be recursively nested because a *Control* node may have a nested block.

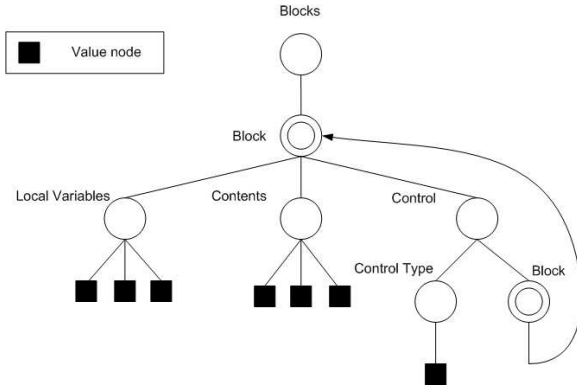


Figure 2. A tree structure of Block[3]

Since a program source is a tree, we can transform it to an XML document. Figure 3 shows the skeleton of a program source represented in XML.

```

<XMLRoot>
  <Headers>
    <header>...</header>
    <!-- multiple headers -->
    <header>...</header>
  </Headers>
  <GlobalVariables>
    <variable>...</variable>
    <!-- multiple variables -->
    <variable>...</variable>
  </GlobalVariables>
  <Functions>
    <function>
      <returnType>...</returnType>
      <name>...</name>
      <arguments>...</arguments>
      <blocks>...</blocks>
    </function>
    <!-- multiple functions -->
    <function>...</function>
  </Functions>
</XMLRoot>

```

Figure 3. XML representation

3.2 Frame Matrix Generation

A frame matrix represents a skeleton of a program source in a matrix. It stores the quantity information such as the number of specific variable types and the number of control types that appear in a function.

Figure 4 shows a mapping relationship between a frame matrix and an XML document (or tagged tree). The main purpose of mapping a tree structure to a

matrix is to provide an elegant method to measure similarities in terms of number.

In the frame matrix, the first row represents header information. Each header has its unique position number. For example, suppose that there are four headers, where type numbers are 3, 10, $n - 1$, and n , respectively. This information plays a role as a fingerprint for headers. It will be represented in the first row of the frame matrix as follows:

$$0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, \dots, 1, 1$$

It says that of all possible headers, header numbers 3, 10, $n - 1$, and n are present in the source program. The representation for global variables is the second row of the matrix. The way of generating a fingerprint for global variables is the same as that of headers.

Next two rows in the frame matrix show the information about the first function. Each function is assigned to two rows in a frame matrix. Two rows are divided into two columns at the k th column as shown below:

return type	argument type
local variable type	control type

Each column represents its partial fingerprint for a function. The left upper column represents a return type of the function. The right upper column represents argument types. Left down column has local variable information. Right down column has control type information. For example, let us consider a function in a program source as follows:

- return type: 2(1)
- argument type: 2(1), 3(1), $n - k(1)$
- local variable type: 2(3), 7(3), $k - 1(5)$
- control type: 1(1), 2(1), $n - k - 1(4)$

In the function, a number indicates a type number and a number in a parenthesis indicates the number of appearances, respectively. This information is fingerprinted in the frame matrix as follows:

$$\begin{array}{cccccccc|cccccccc}
 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 \\
 0 & 3 & 0 & 0 & 0 & 0 & 3 & 0 & \dots & 5 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 & 0
 \end{array}$$

k

It should be noted that since a frame matrix maintains quantity information, the right down column does not preserve the control sequence of the function. Therefore, this fingerprint cannot distinguish between a nested structure and a flat structure of control sequences.

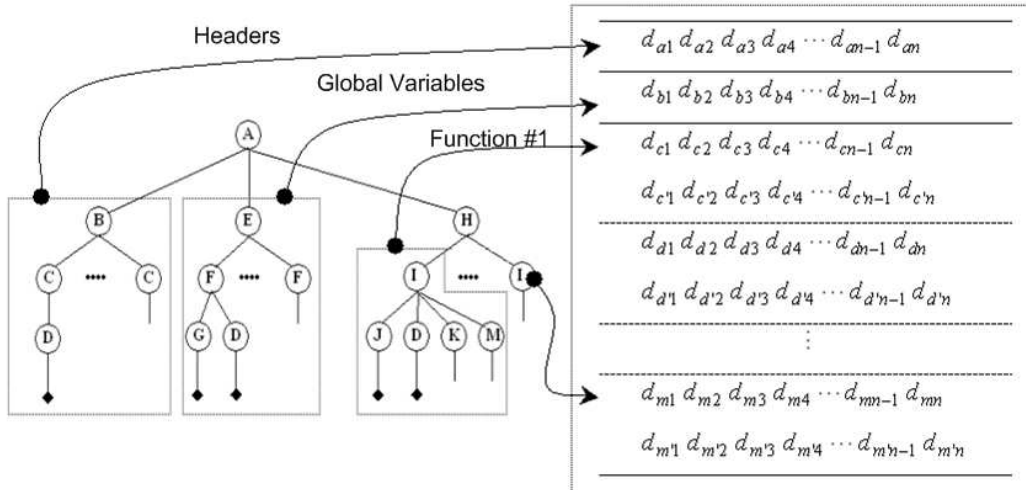


Figure 4. Mapping a tree to a frame matrix[3]

3.3 Control Sequence Extraction

Control sequences play important roles to determine how much two program sources are similar. However, as we have seen, a frame matrix represents the quantity information about control types. It is insufficient for a similarity checker to capture differences between control sequences which have the same quantity values, but different structures.

In the EXPDec model, extracting control sequence is relatively simple because of XML. Since XML documents are used to represent program sources, we can query over the XML documents to extract interesting information by using XQuery.

Figure 5 shows an XML Query used to extract control sequences. The XQuery defines a function, **control_summary**, to extract control types. This function is recursively called in `<controlsequence>` element and generates a control sequence for every function in a program source.

3.4 Control Matrix Generation

In order to compare control sequences for suspected codes, each control sequence is stored in a matrix called *control matrix*.

Figure 6 shows a code segment and its control matrix. As shown in the previous section, the XML query extracts control statements from an XML document. While the query extracts control statements, it converts the name of a control statement to its first letter. This approach makes it possible for the EXPDec model to use Levenshtein distance, which is the most effective algorithm in string comparisons. We must note that a control statement can be nested. This feature requires a mechanism to distinguish a nested structure from

```

DEFINE FUNCTION control_summary(ELEMENT $s)
RETURN ELEMENT
{
  <control> {
    FOR $ss IN $s//controltype
    RETURN {
      <type>($ss)</type>
      control_summary($ss)
    }
  }</control>
}

<xqueryresult>
{
  <function> {
    LET $doc := document("source.xml")
    FOR $func IN $doc/functions/function
    RETURN $func/name {
      <controlsequence> {
        FOR $s IN document("source.xml")//block
        RETURN control_summary($s)
      }</controlsequence>
    }
  }</function>
}</xqueryresult>

```

Figure 5. XQuery for control sequence extraction[3]

a flat structure of a control sequence (or un-nested structure). To distinguish two structures, the name of the nested control statement is defined as an outer statement's first letter and an inner statement's first letter. As shown in Figure 6, for example, *fi* indicates *if* statement is nested by a *for* statement.

```

for(...) {
    if(...)
    else(...)
} while(...) {
    if(...)
}

```

Cell position	1	2	3	4	5
Sequence	f	fi	fe	w	wi

Figure 6. Code segment and control matrix

3.5 Similarity Calculation

The similarity calculation consists of two parts. The first part measures how much two program's skeletons are similar by using frame matrices. The second part analyzes the similarity of the control sequences by using Levenshtein distance and control matrices. Given two program sources, they can be seen as two frame matrices. Figure 7 describes a frame matrix.

$$A = \begin{pmatrix} h_1 & \cdots & h_k & h_{k+1} & \cdots & h_n \\ g_1 & \cdots & g_k & g_{k+1} & \cdots & g_n \\ r_{11} & \cdots & r_{1k} & a_{11} & \cdots & a_{1n-k} \\ l_{11} & \cdots & l_{1k} & c_{11} & \cdots & c_{1n-k} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ r_{p1} & \cdots & r_{pk} & a_{p1} & \cdots & a_{pn-k} \\ l_{p1} & \cdots & l_{pk} & c_{p1} & \cdots & c_{pn-k} \end{pmatrix}$$

Figure 7. A frame matrix

If two programs have p and q functions, they can be considered as $(p+2) \times n$ and $(q+2) \times n$ matrices, respectively. The similarity of headers is obtained by comparing two fingerprints from the frame matrices as shown in Eq.(1).

$$H = A[1][1..n] \times B^T[1][1..n] \quad (1)$$

Since the first rows in the frame matrices represent the fingerprint of headers, the multiplication of both rows shows how many identical headers appear in two program sources. We can apply this strategy to find the similarity of global variables as shown in Eq.(2).

$$G = A[2][2..n] \times B^T[2][2..n] \quad (2)$$

We must note that the same strategy may not work for measuring the similarity of the functions because of reordering plagiarism. Every pair of functions from two matrices must be compared. Eq.(3) through Eq.(6) demonstrate how to estimate the similarities of return type, argument(s), local variables, and control

types. In the equations, the terms R , A , L , and C represent the similarity of return type, argument, local variables, and control types, respectively.

$$R = \max(A[2a][1..k] \times B^T[2b][1..k]) \quad (3)$$

$$A = \max(A[2a][k+1..n] \times B^T[2b][k+1..n]) \quad (4)$$

$$L = \max(A[2a+1][1..k] \times B^T[2b+1][1..k]) \quad (5)$$

$$C = \max(A[2a+1][k+1..n] \times B^T[2b+1][k+1..n]), \quad (6)$$

$$\text{where } 1 \leq a \leq \frac{p}{2}, \text{ and } 1 \leq b \leq \frac{q}{2}$$

Capturing the similarity of control sequences is important for a similarity checker to be reliable. The EXPDec model uses Levenshtein distance to compare two control sequences. It assigns various weights to each control statement based on the order of control statements. The weights are used to guarantee that the front portion of two control sequences dominate the rear portion of control sequences.

Let n be a length of a control sequence in a function. Let W_{mid} be $\lceil \frac{100}{n} \rceil$, which is the weight for the middle cell of a control matrix. We can define the increment, W_{inc} , as follows:

$$W_{inc} = (W_{mid})^2$$

The distance from the middle cell to the i^{th} cell is determined by function $\text{Distance}(i)$. By using the definitions, we can establish a weight function as follows:

$$W(i) = \begin{cases} W_{mid} + \text{Distance}(i) \times W_{inc} & \text{if } i \leq n/2 \\ W_{mid} + \text{Distance}(i) \times (-W_{inc}) & \text{if } i > n/2, \end{cases}$$

where A and B are control matrices. Please note that $W(i)$ is a weight at the i^{th} cell in the control matrix. The weight function is used to measure the similarity between two control sequences in Levenshtein distance algorithm.

Algorithm 1 shows how the EXPDec model compares two control sequences. Algorithm 1 counts the differences between two strings not only when they are different at the same positions, but also when they have different lengths. Based on the weight function, weights are granted to elements in the control matrix. In the distance matrix, the last element in the algorithm, e.g. $D[n+1][m+1]$, has the similarity between two given control matrix A and B .

Algorithm 1 Control sequence comparison

```
1: procedure LEVENSHTEINDISTANCE( $A, B$ )
2:   define  $D[n+1][m+1]$   $\triangleright |A| = n, |B| = m$ 
3:   set  $D[i][0..m] \leftarrow i$   $\triangleright 0 \leq i \leq n$ 
4:   set  $D[0..n][j] \leftarrow j$   $\triangleright 0 \leq j \leq m$ 
5:   for  $i \leftarrow 0, n$  do
6:      $a \leftarrow \text{getCharAt}(A[i])$ 
7:     for  $j \leftarrow 0, m$  do
8:        $b \leftarrow \text{getCharAt}(B[j])$ 
9:       if  $a = b$  then
10:         $cost \leftarrow 0$ 
11:       else
12:         $cost \leftarrow 1$ 
13:       end if
14:        $D[i][j] = \min\{D[i-1][j]+1, D[i][j-1]+1,$ 
15:                     $D[i-1][j-1]+cost\}$ 
16:        $w \leftarrow \text{getWeighAt}(i)$ 
17:        $D[i][j] = w \times D[i][j]$ 
18:     end for
19:    $d \leftarrow D[n+1][m+1]$   $\triangleright$  distance
20:   return  $d$   $\triangleright$  the similarity
21: end procedure
```

The EXPDec model allows users to assign weights to similarity fingerprints. It is useful when every program source has the identical prototypes for functions. By assigning a large value to the most important fingerprint, we can obtain more appropriate results. Eq.(7) shows the formula which is used in the EXPDec to measure the similarity between two program sources. Eq.(7) is derived by using Eq.(1) through Eq.(6) with weight values for fingerprints.

$$\begin{aligned} sim &= W_h \cdot H + W_g \cdot G + W_r \cdot R + W_a \cdot A \\ &+ W_l \cdot L + W_c \cdot C + W_d \cdot D \end{aligned} \quad (7)$$

where, W_x : weights, D : is distance

4 System Architecture

We can implement a program similarity checker based on our proposed model. In our discussion, we only consider C programming language, but we must note that the EXPDec model is orthogonal to programming languages. It can be extended to another programming language. Figure 8 shows the system architecture, which consists of three layers.

Layer 1 generates an XML document from a program source. It uses a C language grammar written in JavaCC[11]. The XML Generator uses a C language parser generated by JavaCC and it generates an XML document based on tag information.

Layer 2 generates a frame matrix and a control matrix from an XML document. When generating

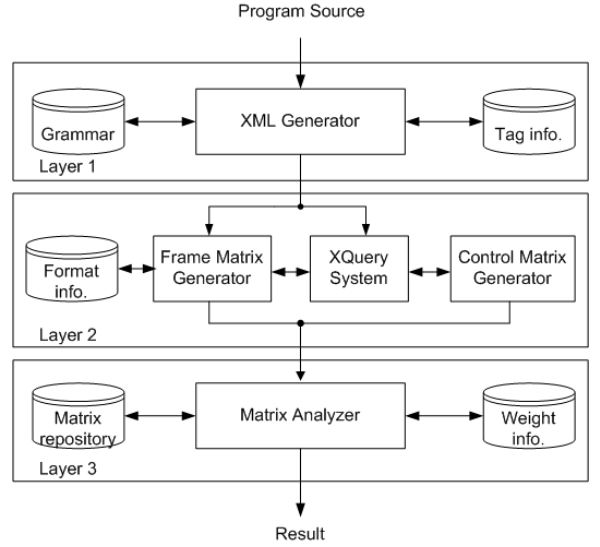


Figure 8. The EXPDec's system architecture

a frame matrix, the Frame Matrix Generator uses a dictionary that contains the information of position formats. The XQuery System extracts control sequences from every function. It is worth noting that the XQuery System may be hard-wired as a code segment because the EXPDec model uses a single query. It may reduce the system size. The Control Matrix Generator creates a control matrix. The frame matrix and the control matrix are passed to the Matrix Analyzer in layer 3.

Layer 3 analyzes two matrices and compares them with the other matrices generated from the other program sources. The weight information dictionary is used to assign different weight to fingerprints of a program source.

The XPDec model[3] showed the reasonable performance results. The system was tested under six different categories such as 1) reordering statements, 2) comment changing, 3) unnecessary header additions, 4) function name changing, 5) variable name changing, and 6) unnecessary statement additions. The test results show that two suspected program sources are determined as identical for the first five categories. For category 6, the average detection rate decreases by 2% as 5% additional statements are added to the original program source. Since the EXPDec model extends the XPDec model, the EXPDec system is expected to show the similar results. However, we must note that the EXPDec data model has the ability of distinguishing a nested structure from a flat structure of a control sequence. This ability makes the EXPDec model more reliable. The claim can be made that the possibility such that two logics are different, but have the same control sequence is very low. However, it is only true when two control sequences are long enough. That is,

if two control sequences are short, e.g. 2 or 3, the possibility is relatively high. Therefore, it is necessary for a similarity checker to have the distinction ability.

5 Conclusion

Computer program plagiarism is one of the most significant problems and common in the computer education field. Due to the plagiarism, it is important to detect suspected programs for accurate assessments. In order to find suspected program sources, $O(n^2)$ pairs of program sources must be compared, leading to excessive string comparisons. String matching algorithms are used in many program similarity detection models and systems. However, these algorithms need intensive processes for simple changes, such as statement reordering and name changing, which are the most common program plagiarism.

We have proposed the EXPDec detection model which is extended from the XPDec model. The EXPDec model uses a tree structure for measuring the skeleton of a program source. Tree structures provide an elegant mechanism to avoid intensive string comparisons for statement reordering. Unlike most similarity detection models, the EXPDec model uses a string comparison method only when two control sequences are compared. It reduces comparison processes and makes the EXPDec model light. As noted, the EXPDec model uses XML for representing a tree structure and Levenshtein distance for control sequence comparisons.

The EXPDec model overcomes the shortcoming of the XPDec model which lacks the ability to distinguish a nested structure from a flat structure of control sequences. In order to solve this problem, the EXPDec model assigns a synthesized name to a control type depending on its previous control types and the structural relationship between two control types. The synthesized names are filled in a control matrix and used by Levenshtein distance algorithm.

Since the EXPDec model extends the XPDec model, the system performance is expected to show the similar performance. However, it is expected that the EXPDec model is more reliable than the XPDec model because of structural distinction mechanism. Unlike the XPDec model, the EXPDec model assigns various weights to each cell of a control matrix. This approach makes it possible to set more weights to specific portions of a control sequence.

The EXPDec model can be simply implemented and customized by using a weight setting mechanism. We hope that our proposed approach helps other instructors and researchers to find valuable insights to implement a simple and effective program similarity detection system.

References

- [1] A. Parker and J. Hamblen, "Computer algorithms for plagiarism detection," *IEEE Transactions on Education*, vol. 32, no. 2, pp. 94–99, 1989.
- [2] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computer Education*, vol. 11, pp. 11–19, 1987.
- [3] S.-Y. Noh, S. Kim, and S. K. Gadia, "An xml plagiarism detection model for procedural programming languages," in *Proceedings of the 2nd International Conference on Computer Science and its Applications*, (San Diego, California, USA), pp. 320–326, June 2004.
- [4] W3C, "XML query language (xquery)." Website, April 2005. <http://www.w3.org/XML/Query>.
- [5] Levenshtein, "XML query language (Xquery)." Website, April 2005. <http://www.levenshtein.net/>.
- [6] P. Clough, "Plagiarism in natural and programming languages: an overview of current tools and technologies," Tech. Rep. CS-00-05, Department of Computer Science, University of Sheffield, 2000.
- [7] SIM, "The software and text similarity tester SIM." Website, April 2005. <http://www.cs.vu.nl/~dick/sim.html>.
- [8] MOSS, "A system for detecting software plagiarism." Website, April 2005. <http://www.cs.berkeley.edu/~aiken/moss.html>.
- [9] SID, "SID-plagiarism detection." Website, April 2005. <http://genome.math.uwaterloo.ca/SID/>.
- [10] M. J. Wise, "Yap3: improved detection of similarities in computer program and other texts," in *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, (Philadelphia, Pennsylvania, USA), pp. 130–134, February 1996.
- [11] JavaCC, "JavaCC grammar repository." Website, April 2005. <http://www.cobase.cs.ucla.edu/pub/javacc/>.