IJCAT International Journal of Computing and Technology, Volume 1, Issue 6, July 2014
ISSN : 2348 - 6090
www.IJCAT.org

Abstract Syntax Tree Generation using Modified Grammar for Source Code Plagiarism Detection

[1] **Resmi N.G.** , [2] **Soman K.P.**

[1] CEN, Amrita Vishwa Vidyapeetham, Coimbatore, Tamilnadu, India

[2] CEN, Amrita Vishwa Vidyapeetham, Coimbatore, Tamilnadu, India

Abstract – Abstract Syntax Tree (AST) matching has been used for detecting plagiarisms in source code files by many researchers. ASTs are usually constructed from parse trees. The generation of ASTs and structure of ASTs used may however differ in each approach. In this paper, we propose a few modifications to C, C++, and Java grammars to generate ASTs. The ASTs generated using modified grammar are further modified to allow subtree matching. These ASTs are traversed to generate node sequences which are compared using sequence matching algorithms - Needleman-Wunsch algorithm and longest common subsequence algorithm. A comparison of results obtained for ASTs generated using original and modified grammars for C, C++, and Java languages is done which shows that the results are better for ASTs generated using modified grammar for the most common plagiarism strategies.

***Keywords* - Abstract Syntax Tree, Source Code Plagiarism Detection, Modified Grammar**

1. Introduction

Plagiarism, the practice of taking someone else's work or ideas and passing them off as one's own without proper acknowledgement of the original author, has become a serious issue in today's world. Plagiarism has become very common in educational institutions. Students copy other students' assignments, both text and source code, without any hesitation to complete their work in time or to complete their work in a better way. Many students seldom care to put their time and effort into doing the assignments on their own when it is far simpler and effortless to copy from someone else. However, it is necessary to differentiate the original work from plagiarized work.

There is an alarming rise in plagiarism due to the widespread use of internet. Internet is an enormously huge repository of information which can be accessed easily from almost anywhere. This has made it very difficult to control plagiarism. Since the task of manually detecting plagiarism in a large document database is very tedious and time-consuming, efforts are continuously being made to automate the process.

Source code plagiarism occurs when source code is copied and edited without proper acknowledgement of the original author [1]. Plagiarism in source code can occur by changing variable names, method names, data types, replacing expressions with their equivalent expressions, replacing one loop statement with another, replacing one selection statement with another, replacing procedure or function calls with procedure or function bodies, and so on [2].

A study [3] shows that structure-metric-based methods tend to outperform attribute-counting-based information retrieval or similarity detection methods. One such method for code similarity detection is AST matching. AST matching has been used for detecting plagiarisms in source code files by many researchers. ASTs are usually constructed from parse trees. The generation of ASTs and structure of ASTs used may however differ in each approach. In this paper, we propose a few modifications to C, C++, and Java grammars to generate ASTs.

2. Abstract Syntax Trees

AST is the output of the syntax analysis phase of a compiler. An AST is an intermediate tree representation of the source code. It represents the abstract syntactic structure of the program. Each node in the AST represents a construct in the source code. A terminal node in AST is either an identifier or a constant. A parser generator is required to produce ASTs. The trees generated by parser generators are called parse trees and are usually huge in size. Parse trees contain large number of nodes that carry no structure information. These trees can be reduced in size by making suitable modifications in the parser definition for a specific language to remove redundant nodes which do not add any extra information to the program structure. The most common nodes which are eliminated include nodes that represent punctuation marks such as semi-colons and commas. The reduced tree will contain only those nodes which carry useful structural information and hence the name ***Abstract Syntax Tree***.

Each source code file is parsed and its AST is generated. Once the ASTs are generated, comparison of ASTs can be done in different ways. Ligaarden [4] proposes an AST based approach to detect plagiarism in Java source code. The author modifies the Java grammar to obtain the corresponding AST. A preorder traversal is done through the ASTs to be compared as done in [5] to generate node sequences. Top Down Unordered Maximum Common Subtree Isomorphism (TDUMCSI) algorithm [6,7] along with sequence matching algorithms – Needleman-Wunsch (NW) algorithm and Longest Common Subsequence (LCS) algorithm, are then used to compare the node sequences and find matches.

We have earlier extended Ligaarden's approach to detect plagiarisms in C and C++ [8] by making similar modifications in the C and C++ grammars as done in [4] for Java grammar. We have now improved upon Ligaarden's modified Java grammar to generate ASTs that would give better results on comparison.

3. AST Generation using Modified Grammars

Parse trees are huge because most of the parsers create nodes for all the non-terminals in the grammar. A simple method to reduce the size of a parse tree and produce an AST is to retain all the terminal nodes and only those non-terminal nodes which have more than one child. Further reduction in size of parse trees can be done by modifying the grammars for each programming language.

3.1 Modifying C Grammar

ASTs generated using original C grammars do not allow proper comparison between subtrees of different iteration statements or between subtrees of different selection statements. The C grammar is modified so as to incorporate the changes required in order to allow comparison between subtrees of different iteration or selection statements.

3.1.1. Original C Grammar

Statement() : (LOOKAHEAD(2) LabeledStatement()
    | ExpressionStatement() | CompoundStatement()
    | SelectionStatement() | IterationStatement()
    | JumpStatement())
LabeledStatement() : (<IDENTIFIER> ":" Statement()
    | <CASE> ConstantExpression() ":" Statement()
    | <DFLT> ":" Statement())
CompoundStatement() : "{" [LOOKAHEAD
    (DeclarationList()) DeclarationList()]
    [StatementList()] "}"
StatementList() : (Statement())+
SelectionStatement() : <IF> "(" Expression() ")"
    Statement() [LOOKAHEAD (2) <ELSE> Statement()]
    | <SWITCH> "(" Expression() ")" Statement())
IterationStatement(): <WHILE> "(" Expression() ")"
    Statement()
    | <DO> Statement() <WHILE> "(" Expression() ")" ";"
    | <FOR> "(" [Expression()] ";" [Expression()] ";" [Expression()] ")" Statement())

3.1.2. Modified C Grammar

A new rule StatementBlock is added and the rules for IterationStatement and SelectionStatement are redefined using StatementBlock. The AST generated with this modified grammar will always have StatementBlock as root of the subtree which corresponds to the body of iteration or selection statement. The rule for LabeledStatement is also redefined.

The subtree for a StatementBlock either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single statement as its body then the node label StatementBlock in the AST generated is changed to CompoundStatement. If the iteration statement or selection statement has a compound statement as its body then the node StatementBlock is removed.

The rule for *switch* statement is changed so that subtree rooted at CaseBlock corresponding to each *case* block with zero, one, or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from CaseBlock to CompoundStatement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have CompoundStatement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at CompoundStatement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

StatementBlock() : Statement()
Statement():(LOOKAHEAD(2) LabeledStatement()
| ExpressionStatement() | CompoundStatement()
| SelectionStatement() | IterationStatement()
| JumpStatement())
LabeledStatement() : <IDENTIFIER> ":" Statement()
CompoundStatement():"{" [LOOKAHEAD (DeclarationList()) DeclarationList()]
[(Statement())+] "}"
SelectionStatement() : (<IF> "(" Expression() ")" StatementBlock() [LOOKAHEAD(2) <ELSE> StatementBlock()]
| <SWITCH> "(" Expression() ")" "{" (CaseLabel() CaseBlock())* "}")
CaseLabel() : (<CASE> ConstantExpression() ":"
| <DFLT> ":")
CaseBlock():(Statement())*
IterationStatement() : (<WHILE> "(" Expression() ")" StatementBlock()
| <DO> StatementBlock() <WHILE> "(" Expression() ")" ";"
| <FOR> "(" [Expression()] ";" [Expression()] ";" [Expression()] ")" StatementBlock())

3.2 Modifying C++ Grammar

AST generation using original C++ grammar also faces the same problem as with C grammar. The modifications to C++ grammar are hence similar to that of C grammar.

3.2.1. Original C++ Grammar

statement_list() :(LOOKAHEAD(statement()) statement())+
statement() : LOOKAHEAD(declaration()) declaration()
| LOOKAHEAD(expression()";") expression() ;"
| compound_statement() | selection_statement()
| jump_statement() | ";" | try_block()
| throw_statement()| LOOKAHEAD(2) labeled_statement()| iteration_statement()
labeled_statement():<ID> ":" statement()
| "case" constant_expression() ":" statement()
| "default" ":" statement()
compound_statement() : "{" (statement_list())? "}"
selection_statement() : "if" "(" expression() ")" statement()
(LOOKAHEAD(2) "else" statement())?
| "switch" "(" expression() ")" statement()
iteration_statement() : "while" "(" expression() ")" statement()
| "do" statement() "while" "(" expression() ")" ";"
| "for" "("(LOOKAHEAD(3) declaration() |

Fragment 1 (*if* without block)	Fragment 2 (*if* with block)

if (ch==1)
cout<<"one";
else
cout<<"wrong choice!";
	if (ch==1)
	{ cout<<"one";}
	else
	{
	cout<<"wrong choice!";
	}

expression() ";" | ";") (expression())? ";"
(expression())? ")" statement()
Consider code fragments 1 and 2. Figure 1 shows the ASTs generated using original C++ grammar for the code fragments.

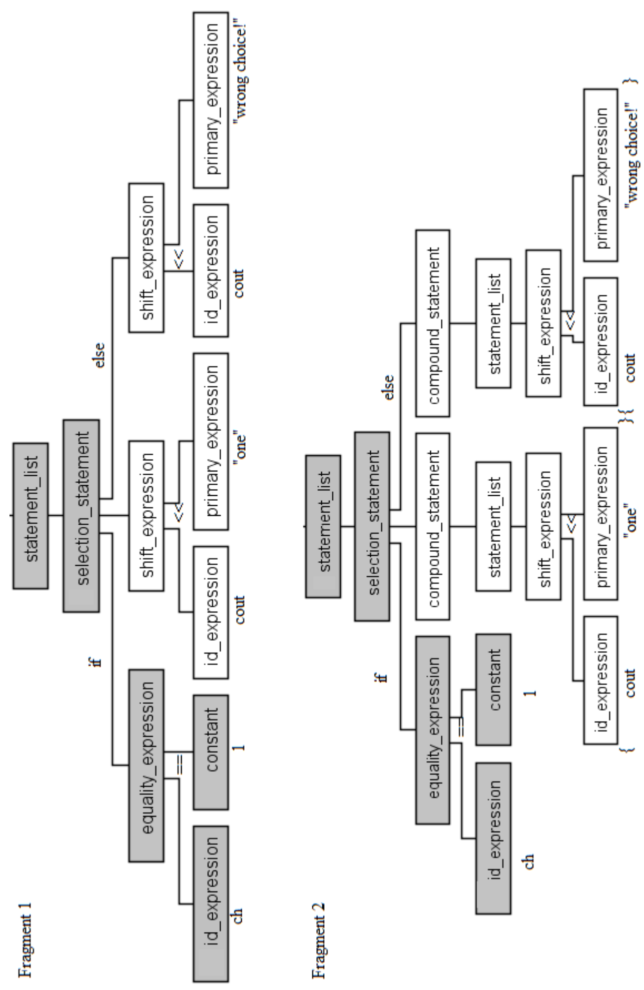

Fig. 1 ASTs generated using original C++ grammar for code fragment 1 – *if* without block and code fragment 2 – *if* with block.

IJCAT International Journal of Computing and Technology, Volume 1, Issue 6, July 2014
ISSN : 2348 - 6090
www.IJCAT.org

3.2.2. Modified C++ Grammar

The modification done to C++ grammar is similar to that done to C grammar. A new rule statement_block is added and the rules for iteration_statement and selection_statement are redefined using statement_block. The AST generated with this modified grammar will always have statement_block as root of the subtree which corresponds to the body of iteration or selection statement. The rule for labeled_statement is also redefined.

The subtree for a statement_block either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single statement as its body then the node label statement_block in the AST generated is changed to compound_statement. If the iteration statement or selection statement has a compound statement as its body then the node statement_block is removed.

The rule for *switch* statement is changed so that subtree rooted at case_block corresponding to each *case* block with zero, one, or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from case_block to compound_statement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have compound_statement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at compound_statement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

```
statement_block():statement()
statement() :  LOOKAHEAD( declaration()) declaration()
        | LOOKAHEAD(expression() ";") expression()";"
        | compound_statement() | iteration_statement()
        | LOOKAHEAD(2) labeled_statement()
        | selection_statement() | jump_statement() | ";"
        | try_block() | throw_statement()
labeled_statement() :<ID> ":" statement()
compound_statement() :"{"(statement())* "}"
iteration_statement() :"while" "(" expression() ")"
        statement_block()
        | "do" statement_block() "while" "(" expression()
        ")" ";"
        | "for" "("(LOOKAHEAD(3) declaration() |
        expression() ";" | ";") (expression())? ";"
        (expression())? ")" statement_block()
selection_statement() : "if" "(" expression() ")"
        statement_block() (LOOKAHEAD(2) "else"
        statement_block())?
        | "switch" "(" expression() ")" "{" ( case_label()
        case_block() )*  "}"
case_label():"case" constant_expression() ":" | "default"
":"
case_block():( statement() )*
```

Figure 2 shows the AST generated using modified C++ grammar for code fragment 1 – *if* without block. It also shows the same AST obtained after changing statement_block to compound_statement. The resultant modified AST is same as that of *if* with block obtained using modified grammar thereby allowing a comparison between the body of *if* with and without blocks.

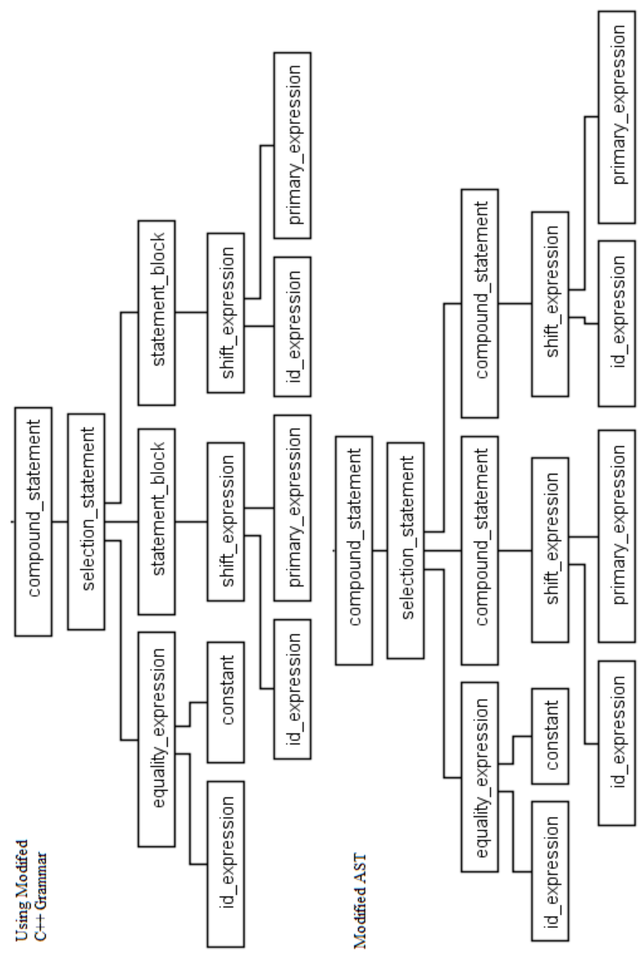

Fig. 2 AST generated using modified C++ grammar and AST modified by changing statement_block to compound_statement for code fragment 1 – *if* without block

3.3 Modifying Java Grammar

3.3.1. Original Java Grammar

```
Statement():LOOKAHEAD(2)  LabeledStatement()
        | AssertStatement() | Block() | EmptyStatement()
        | StatementExpression() ";" | SwitchStatement()
        | IfStatement() | WhileStatement() | DoStatement()
        | ForStatement() | BreakStatement()
        | ContinueStatement() | ReturnStatement()
        | ThrowStatement() | SynchronizedStatement()
        | TryStatement()
SwitchStatement():"switch" "(" Expression() ")" "{" (
        SwitchLabel() SwitchLabelBlock() )* "}"
SwitchLabel():"case" Expression() ":" | "default" ":"
SwitchLabelBlock():( BlockStatement() )*
IfStatement():"if" "(" Expression() ")" Statement()
        [LOOKAHEAD(1) "else" Statement() ]
WhileStatement()   :"while" "(" Expression() ")" Statement()
DoStatement():"do" Statement() "while" "(" Expression()
        ")" ";"
ForStatement():"for""("(LOOKAHEAD(Type()
        <IDENTIFIER> ":")Type() <IDENTIFIER> ":"
        Expression() | [ ForInit() ]";" [ Expression() ] ";"
        [ ForUpdate() ] ) ")" Statement()
```

3.3.2. Modified Java Grammar

Ligaarden [4] makes a distinction between the different types and between the literals of different types on modifying the grammar. Making a type distinction and literal distinction will only help to discriminate the files rather than finding their similarity. It is therefore necessary to retain the original grammar rules for primitive types and literals to identify plagiarisms involving changing identifiers and constants, and changing data types effectively.

In the original Java1.5 grammar, there are separate rules for the selection statements *if* and *switch*. In case of different rules for the selection statements, the comparison stops at nodes labeled IfStatement and SwitchStatement since the labels do not match. Similarly, there are separate rules for *for*, *while*, and *do-while*. The comparison stops at nodes labeled ForStatement, WhileStatement, and DoStatement since the labels do not match. Hence, the rules are modified so that the separate rules for *if* and *switch* are combined to form a new rule SelectionStatement and the separate rules for *for*, *while*, and *do-while* are combined to form a new rule IterationStatement.

The modification done to C and C++ grammar is also done to Java grammar. A new rule StatementBlock is added and the rules for IterationStatement and SelectionStatement are redefined using StatementBlock. The AST generated with this modified grammar will always have StatementBlock as root of the subtree which corresponds to the body of iteration or selection statement.

The subtree for a StatementBlock either corresponds to a single statement or a compound statement. If the iteration statement or selection statement has a single statement as its body then the node label StatementBlock in the AST generated is changed to CompoundStatement. If the iteration statement or selection statement has a compound statement as its body then the node StatementBlock is removed.

The rule for *switch* statement is changed so that subtree rooted at CaseBlock corresponding to each *case* block with zero, one, or more statements is separated from the other. The root of each of these subtrees corresponding to each of the *case* blocks in *switch* statement is changed from CaseBlock to CompoundStatement to allow subtree matching. In the modified AST, the iteration statements *for*, *while* and *do-while* or the selection statement *if-else* will always have CompoundStatement as root of its subtree which corresponds to the body of iteration or *if-else* statement and the *case* blocks in *switch* will also be rooted at CompoundStatement. This modification allows the subtrees of iteration or selection statement with and without block to be matched.

```
StatementBlock() :Statement()
Statement():LOOKAHEAD(2)  LabeledStatement()
        | AssertStatement() | Block() | EmptyStatement()
        | StatementExpression() ";" | IterationStatement()
        | SelectionStatement() | BreakStatement()
        | ContinueStatement() | ReturnStatement()
        | ThrowStatement() | SynchronizedStatement()
        | TryStatement()
SelectionStatement():"if" "(" Expression() ")"
        StatementBlock()[LOOKAHEAD(1)
        "else" StatementBlock() ]
        | "switch" "(" Expression() ")" "{" (CaseLabel()
        CaseBlock() )* "}"
CaseLabel():"case" Expression() ":" |  "default" ":"
CaseBlock(): ( BlockStatement() )*
IterationStatement():"while" "(" Expression() ")"
        StatementBlock
        | "do" StatemenBlock() "while" "(" Expression()
        ")" ";"
```

```
| "for" "(" (LOOKAHEAD(Type() <IDENTIFIER> ":") Type() <IDENTIFIER> ":" Expression() | [ ForInit() ] ";" [ Expression() ] ";" [ ForUpdate() ] ) ")" StatementBlock()
```

Consider a code fragment: if with block

```
if (ch==1)
{
    System.out.println("one");
}
else
{
    System.out.println("wrong choice!");
}
```

Figure 3 shows the AST generated using original and modified grammars for the code fragment – *if* with block. There is no separate rule for *if* in modified grammar so as to allow comparison between subtrees of different selection statements.

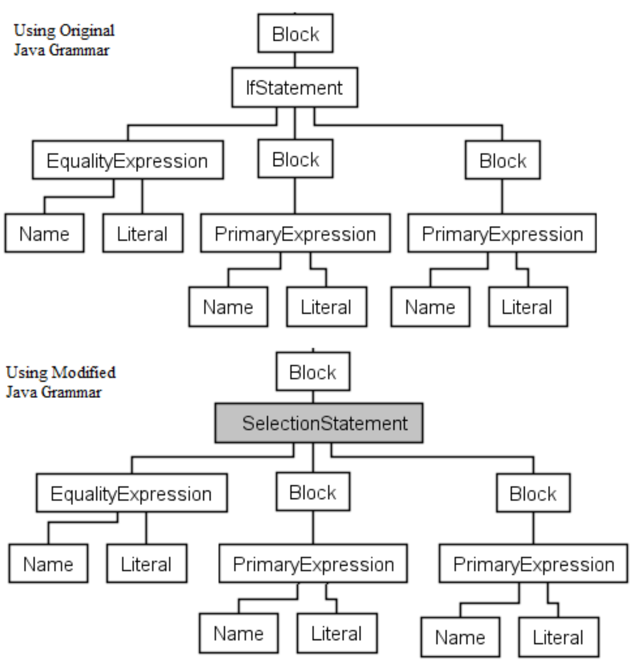

Fig. 3 ASTs generated using original and modified Java grammar for code fragment – *if* with block.

3. Results and Discussions

The similarity scores obtained on applying LCS and NW algorithms on ASTs generated and modified using original and modified C, C++, and Java grammars for the common plagiarism strategies are given in Tables 1, 2, and 3.

Table 1:. Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated using Original and Modified C Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	***C***		***Modified C***	
	NW	***LCS***	***NW***	***LCS***
Changing identifiers	100	100	100	100
Changing data types	100	100	100	100
Changing the order of operands in expressions	100	100	100	100
Changing the order of independent code	100	100	100	100
Replacing an expression with an equivalent expression	83.3	83.3	81.8	81.8
Replacing one loop statement with another: a) *for* without block – *for* with block	72.7	72.7	100	100
b) *while* without block – *while* with block	62.2	62.2	88.4	88.4
c) *for* without block – *while* without block	74.4	74.4	74.4	74.4
d) *for* without block – *while* with block	50	50	76.2	76.2
e) *for* with block – *while* without block	48.9	48.9	74.4	74.4
f) *for* with block – *while* with block	78.3	78.3	76.2	76.2
g) *do-while* – *for* without block	50	50	76.2	76.2
h) *do-while* – *for* with block	78.3	78.3	76.3	76.3
i) *do-while* – *while* without block	62.2	62.2	88.4	88.4
j) *do-while* – *while* with block	100	100	100	100
Replacing one selection statement with another: a) *if* without block – *if* with block	57.9	57.9	100	100
b) *if* without block – *switch*	43.2	43.2	81.1	81.1
c) *if* with block – *switch*	63.4	63.4	81.1	81.1
Replacing a statement block with a function call	56.4	56.4	56	56

Table 2: Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated using Original and Modified C++ Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	***C++***		***Modified C++***	
	NW	***LCS***	***NW***	***LCS***
Changing identifiers	100	100	100	100

Changing data types	100	100	100	100
Changing the order of operands in expressions	100	100	100	100
Changing the order of independent code	81.5	100	80.8	100
Replacing an expression with an equivalent expression	83.3	83.3	81.8	81.8
Replacing one loop statement with another: a) *for* without block – *for* with block	80	80	100	100
b) *while* without block – *while* with block	68.3	68.3	87.2	87.2
c) *for* without block – *while* without block	76.9	76.9	76.9	76.9
d) *for* without block – *while* with block	60	60	79	79
e) *for* with block – *while* without block	58.5	58.5	76.9	76.9
f) *for* with block – *while* with block	81	81	79	79
g) *do-while* – *for* without block	60	60	79	79
h) *do-while* – *for* with block	81	81	79	79
i) *do-while* – *while* without block	68.3	68.3	87.2	87.2
j) *do-while* – *while* with block	100	100	100	100
Replacing one selection statement with another: a) *if* without block – *if* with block	60	60	100	100
b) *if* without block – *switch*	46.2	46.2	82.1	82.1
c) *if* with block – *switch*	65.1	65.1	82.1	82.1
Replacing a statement block with a function call	48.8	48.8	48.1	48.1

Table 3: Similarity Scores Obtained on Applying LCS and NW Algorithms on ASTs Generated using Original and Modified C++ Grammar for the Common Plagiarism Strategies

Plagiarism Strategy	***Java***		***Modified Java***	
	NW	***LCS***	***NW***	***LCS***
Changing identifiers	100	100	100	100
Changing data types	100	100	100	100
Changing the order of operands in expressions	100	95.4	100	100
Changing the order of independent code	90.9	75.8	90.9	100
Replacing an expression with an equivalent expression	89.5	89.5	89.5	89.5
Replacing one loop statement with another: a) *for* without block – *for* with block	86.8	86.8	100	100
b) *while* without block – *while* with block	84.6	84.6	98.1	98.1
c) *for* without block – *while* without block	54.9	54.9	83.0	83.0
d) *for* without block – *while* with block	52.8	52.8	81.5	81.5
e) *for* with block – *while* without block	53.9	53.9	83.0	83.0
f) *for* with block – *while* with block	51.9	66.7	81.5	81.5
g) *do-while* – *for* without block	52.8	52.8	81.5	81.5
h) *do-while*– *for* with block	51.9	66.7	81.5	81.5
i) *do-while* – *while* without block	67.9	83.0	98.1	98.1
j) *do-while* – *while* with block	66.7	85.2	100	100
Replacing one selection statement with another: a) *if* without block – *if* with block	63.2	52.6	100	100
b) *if* without block – *switch*	50	50	85.2	85.2
c) *if* with block – *switch*	54.9	54.9	85.2	85.2
Replacing a statement block with a function call	61	61	61	61

A comparison of results obtained for ASTs generated using original and modified grammars for C, C++, and Java languages shows that the results are better for ASTs generated using modified grammar for the most common plagiarism strategies.

4. Conclusions

The ASTs generated using modified grammars were found to be more effective than those with original grammar for source code plagiarism detection. The results of AST matching are found to be highly reliable since they take into account the structural information of the programs. AST based approach proved to be very efficient in terms of similarity detection, but for a huge program database the runtime was found to be very high.

References

[1] G. Cosma, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis", Ph.D. Thesis, University of Warwick, 2008.

[2] G. Whale, "Software Metrics and Plagiarism Detection", Journal of Systems and Software, 13, 1990, 131-138.

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

[4] O.S. Ligaarden, "Detection of Plagiarism in Computer

Programming Using Abstract Syntax Trees", Master Thesis, University of Oslo, 2007.

[5] R. Koschke, R. Falke and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees", 13th WCRE 2006, 253-262.

[6] G. Valiente, "Simple and Efficient Tree Pattern Matching", Technical Report LSI-00-72-R, Technical University of Catalonia, 2000.

[7] G. Valiente, Algorithms on Trees and Graphs, Springer-Verlag, Berlin, 2002.

[8] N. G. Resmi and K. P. Soman, "Abstract Syntax Trees with Latent Semantic Indexing for Source Code Plagiarism Detection", International Journal of Advanced Research in Computer Science, 3(3), 2012, 546-550.

N. G. Resmi The author secured her master's degree in Computational Engineering and Networking from Amrita Vishwa Vidyapeetham (2008). She is currently a doctoral student there. The author has also worked as Assistant Professor in Sahrdaya College of Engineering and Technology. Her current research interests include compilers, wavelet theory, kernel methods and linear algebra.

K. P. Soman The author secured his Ph.D. from IIT Kharagpur and was scientific officer in the Reliability Engineering Centre, IIT Kharagpur. The author currently serves as Head and Professor at Amrita Center for Computational Engineering and Networking (CEN), Coimbatore. He has been in the research field for more than 25 years and his current interests are Software Defined Radio, Statistical Digital Signal Processing (DSP) on Field Programmable Gate Array (FPGA), Wireless Sensor Networks, High Performance Computing, Machine learning using Support Vector Machines, Signal Processing, and Wavelets & Fractals.