I.J. Modern Education and Computer Science, **2021, 3, 34-56**
Published Online June 2021 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2021.03.04

A Methodology for Reliable Code Plagiarism Detection Using Complete and Language Agnostic Code Clone Classification

Sanjay B. Ankali
Research Scholar at VTU-RRC, & Faculty, KLE College of Engineering and Technology, Chikodi-India-591201
Email: sanjayankali123@gmail.com

Latha Parthiban
Department of Computer Science, Pondicherry University, Community College, Lawspet-India-605008
Email: lathaparthiban@yahoo.com

Received: 17 January 2021; Revised: 26 February 2021; Accepted: 14 March 2021; Published: 08 June 2021

Abstract: Code clone detection plays a vital role in both industry and academia. Last three decades have seen more than 250 clone detection techniques with lack of single framework that can detect and classify all 4 basic types of code clones with high precision. This serious lack of clone classification impacts largely on the universities and online learning platforms that fail to validate the projects or coding assignments submitted online. In this paper, we propose a complete and language agnostic technique to detect and classify all 4 clone types of C, C++, and Java programs. The method first generates the parse tree then extracts the functional tree to eliminate the need for the preprocessing stage employed by previous clone detection techniques. The generated parse tree contains all the necessary information for detecting code clones. We employ TF-IDF cosine similarity for the proper classification of clone types. The proposed technique achieves incredible precision rate of 100% in detecting the first two types of clones and 98% precision in detecting type-3 and type-4 clones for small codes of C, C++, and Java containing an average line count of 5. The proposed technique outperforms the existing tree-based clone detection tools by providing the average precision of 98.07% on the C, C++, and Java programs crawled from Github with an average line count of 15 which signifies that cosine similarity measure on ANTLR functional tree accurately detects all 4 types of small clones and act as proper validation tools for identifying the learning level in the submitted programming assignment.

Index Terms: Clone types, functional tree, TF-IDF, cosine similarity, Code plagiarism.

1. Introduction

Code cloning is the process of creating functionally similar codes with syntactic modifications. It can also be defined as semantically similar code fragment pairs with or without syntactical change [1]. Many researchers refer this process with different terms like similar code [2], identical code [3] or duplicate code [4]. Large systems contain 10-15% and 20-50% of duplicate code in the codebase [5]. Based on the milestone, literature like [6,7, 8, 9], and based on the study of Wang, W. L. (2020) [1], the code clones are of 4 types that can be categorized into Type-1 which is also called as exact clones, Type-2 which is also called as renamed clones and Type-3 which is also called as near-miss clones. Semantically similar codes that are implemented differently are called as Type-4 clones.

Language agnostic code clone detection has a great role to play in building reliable code plagiarism detection. In order to provide justification to academic integrity, an attempt to code plagiarism detection has already started in late 1976. Based on the survey conducted by Chivers [10] The code plagiarism detection is based on the 3 different techniques a) attribute-based b) structure-based c) hybrid technique. Attribute counting technique was first conducted by Ottenstein [11] The study was based on metrics of Halstead [12] considering the number of unique operators and operands. In the year 1981 Grier [13] added 16 new attributes to the existing metrics of Ottenstein [11] that include looping statements, conditional statements, and tokens like white space, line. A study by J. L. Donaldson et al [14] is based on counting the programming constructs like looping and conditional statements. An empirical approach proposed by Faidhi and Robinson [15] for detection of program similarity is based on 24 metrics. These initial studies were completely based on the text or strings and counting the attributes present in the program. In a comparative study made by Whale [16] argue that more application-specific metrics and structural features of code need to be considered for accurate detection of code plagiarism.

Existing plagiarism detection tools like MOSS, JPag calculates similarities in terms of percentage which can present the amount of similarity between two codes but fail to validate the submitted coding assignment when they are implemented differently with type 4 clones. Correlating clone type classifications (type 1, 2, 3, and 4) will give a better understanding of learning of students from submitted programming assignments. Type 1 and 2 are ugly practices that breach academic integrity. Type 3 is bad practice and type 4 is good practice as it increases the level of learning by making students implement functionally similar codes using different syntax. This research paper contributes in following way.

- We use the capability of freely available ANTLR parser generator to extract functional tree by providing corresponding grammar files for the input programs. Extraction of functional tree eliminates the need for the preprocessing phase employed by earlier clone detection techniques.
- Vector representation of the functional tree using TF-IDF is given as input to cosine similarity which proved to be a more accurate classification of all 4 clone types for the micro programs with line count of 5, 15 and 32.
- Existing code plagiarism detection tools that work on similarity matching and report type 4 clones as plagiarism but with respect to academia, it is a good learning practice. We relate clone detection to academic code plagiarism to identify the good, the bad and ugly practices of students.

2. Background and Related Work

In this section, we present the examples to understand the various clone types, literature on clone detection and literature on code plagiarism. To justify our understanding of clone types, we present the examples based on [6]. According to [8], there are 9 types of clones. Based on the editing taxonomy there exist 4 basic clone types [6].

2.1 Background

In the following section we present small programs of our data set to define clone types.

Type-1 clone: Syntactical and semantically similar codes with a change in white space and comments [1].

```
main()    // addition program
{
int first=10, second=20, sum;
sum= first+ second; //logic
printf(“sum of two numbers=%d”, sum);
}
```

Code-1

```
/* addition program */
main()
{
int first=10, second=20, sum;
sum= first+ second; //logic
printf(“sum of two numbers=%d”, sum);
}
```

Code-2

Code-1 and Code-2 are an example of type 1 clones. These are also called as exact clones or copy/paste clones. This practice of copying the program as it is from the peer needs to be detected to stop the ugly practice of learning in students and also breach software integrity in industry.

Type-2: syntactically similar codes with a change in variable, function, and class name.

```
main()
{
int i=1,fact=1,n;
printf("Enter the number");
scanf("%d",&n);
while(i<=n)
{
    fact=fact*i;
   i++;
}
   printf("factorial of number is=%d", fact);
       }
```

Code-3

```
main()
{
   int x=1,res=1,n;
   printf("Enter the number");
   scanf("%d",&n);
   while(x<=n)
  {
      res=res*x;
      x++;
   }
 printf("factorial of number is=%d", res);
 }
```

Code-4

Code-3 and 4 are examples of type 2 clones. These clones are also called as renamed clones. This practice of renaming the multiple entities in the program like identifier, method name, and the class name is a bad practice of coding by students which breaches academic integrity.

Type 3: types-2 clones with addition and deletion of lines creates type-3 clones. below code-1 and code-5 are type-3

```
main()    // addition program
{
int a=10,b=20,c;
c=a+b;            //logic
printf("sum of two numbers=%d",c);
}
```

Code-1

```
main()    // addition program
{
 int a=10,b=20,c;
 c=a+b;            //logic
 printf("program find addition")
 printf("sum of two numbers=%d", c);
    }
```

Code-5

Type 3 clones are a matter of interest for many researchers in the past where many tools mentioned in [7] struggled to detect type3 clones. In academia, these are just superset of type2 clone which is considered as bad practice by students.

Type 4: These are semantically similar codes with change in the syntax [1]. For example, consider following code fragments.

```
int fact(int x)
{
for(i=1;i<=n;i++)
fact=fact*i;
printf("factorial of number is=%d",fact);
}
```

Code-6

```
int fact(int n)
{
if(n==0)
return 1;
else
return n*fact(n-1);
}
```

Code-7

Code-6 and Code-7 are type 4 clones. Type 4 clones are matter of interest to both industry and academia. Type 4 clone detection was out of the scope of many great scalable tools mentioned in the introduction. A major issue with existing code plagiarism detection tools is that/, they report these codes as clones but with respect to the academic point of view it improve the learning levels in students.

2.2 Software clone detection

Significant research has happened in finding the software clone types. In this section, we present the summary of clone detection tools/ techniques. Based on the milestone literature works of [6,8], we group all the clone detection approaches into 5 classes like Text-based, Token-based, Tree-based, PDG-based, Metric-based [7] .

Text-based approaches: Text based approaches compare two code fragments based on the input text or string. The tools like Duploc[4], simian[18], EqMiner[19], NICAD[20], DuDe[21]. Except for NICAD none of the tools address detecting even small instances of Type 3. Whereas tool proposed by Johnson, Duploc, DuDe, and SDD detect only type 1 and other tools were meant to find first two clone types.. The work of (Kim, 2018) detects type 1 and 2 of C and C++ code. Highly scalable tool VUDDY [22] detects first two clone types of C/C++. The tool CCCD [23] has made an attempt to detect type 3 and 4 clones of C language. The tool vfdtect [24] detects type 3 and type 4 clones of Java code.

Token-based Techniques: The technique works by performing lexical analysis to extract the tokens from source code. These extracted tokens are used to form the suffix tree or suffix array for matching. Tools like Dup[25], CCFinder[26], iClones[27], CP-Miner[28]. These tools have detected both type 1 and type 2. The tool Siamese[29] detects first three clone types of java code and the tool CP-Miner detects type 3 clones moderately. The work of [30] finds the first 3 clone types of IJDataset. The tool CCAligner [31] detects first 3 clone types of C and Java language. Higly scalable tool SourcererCC [32] detects first 3 clone types of IJDataset. The language agnostic tool CCfindersw [33] detects only first two clone types.

Tree-based Techniques: Tree based approaches are good for refactoring and increase the precision of clone detection [1,34]. Tree based approaches work by parsing the source code to parse tree. tools like Deckard[35], CloneDR[36], simScan[37] , Asta[38], CloneDigger[39], sim[40], ClemanX[41], JCCD API[42], CloneDetection[43], cpdetector[34]. These techniques did not detect type 4 clones. The methodology by Yang [44] detects functional clones of java code. The work of [45] finds the type 3 and type 4 clones of Java. The work of [46] detects all 4 types of clones for Java codes.

PDG-based Techniques These techniques prepare the program dependency graph to represent the control and data flow of source code[47]. The technique has addressed the detection of type 4 clones. Tools like PDG-DUP[48], Scorpio[49], Duplix[50], Choi[51] concentrate on finding first 3 types of clones.

Metrics-based Techniques like CLAN/Covet[52], Antoniol[53], Dagenias[54] that counts a number of different category of tokens and stores them in a matrix. Both matrixes are matched to get the clones. The tool Vincent [55] detects first 3 clone types of Java code. These tools suffer from false positives for detecting type 3 and 4 clones.

In table 1, we summarize the number of clone detection tools developed to address various clone types and language they support.

Table 1. Number of tools to identify different clone types

Clone type	**Number of tool/techniques**
1	71
2	71
3	55
4	19
All 4 types	12
Function clones	3
File clone	1

With having a great number of studies in clone detection, we still find a lack in complete and accurate code clone detection techniques. Maximum of 68 tools work on java code, 30 tools work on C code and 13 tools work on C++ code for clone detection. We find only 4 language agnostic tools which is the big gap in clone detecion reasearch.

2.3 Code plagiarism detection

In order to provide justification to academic integrity, an attempt to code plagiarism detection was started in late 1976. From 2005 onwards detection of code plagiarism detection was based on string matching, fingerprinting, and tree-based. There are many state-of-art tools for conducting code plagiarism, those include JPlag[56], Marble, SIM, Plaggie, MOSS, Sherlock. JPlag works on tokenizing, greedy string tilling, and optimization. It supports C, C++, Java, C# and text files. Marble is structure-based tool, it works by the recursive splitting of the file till the top line reaches, then removes easily modifiable lexical patterns like class name, function name, identifier name, white space, and comments, and finally applies Linux diff to calculate the score of line similarity. Marble supports Java, Perl, PHP, and XSLT languages. MOSS works on the Linux platform based on document fingerprinting and supports more number of languages like C, C++, Java, C#, Python, VB, Javascript FORTRON, MIPS, and Assembly languages. Plaggie is the command line java application to find plagiarism in java codes. Tool SIM works by tokenizing the source code file then apply forward reference table for matching. It supports C, Java, pascal, Modula-2, Lisp, Mirad and text files. A comparative study made by (Hage et al, 2010) gives many insights a) Jplag, MOSS, and Marble perform better on java code. b) they are sensitive to small refactoring c) they present the similarity in terms of the percentage of similarity.d) MOSS supports 23 languages c) Plaggie works only for Java programs.

There are many plagiarism detection tools like one proposed by Birov, T. C. (2015) [57], it works only on java code, another tool proposed by M. Iwamoto, S. O. (2013.)[58] that works on C and Java codes. CPDP[59] works on Java to find copy/paste activities. This tool can be used in finding type 1, type 2 software clones. The study by [60] works on binary code to check file similarity, [61] works on any language to find file similarity. BCFinder [60] works on C/CP++. A tool PlaGate[62] uses Latent semantic analysis to improve the performance of current plagiarism detection tools. A more detailed and analytical comparative study was conducted by[55], which includes 30 code similarity analyzers including fuzzywuzzy and jellyfish. in his discussion he concludes by saying, the code similarity tools behave differently on pervasive code modification and boiler-plate code. Often used tool ccfx, and Python string matching algorithm, fuzzywuzzy work better on pervasive code modification. The experiment conducted on SOCO data sets for boilet-plate codes ranks jplag-text plagiarism detector followed by simjava, simian, jplag-java, and Deckard[55].

3. Proposed Methodology

Proposed work is based on the generation of ANTLR functional trees from the source code using corresponding language grammar. The proposed method works in 4 phases.

1. Repository Building and parse tree generation.
2. Functional tree generation.
3. Vector representation.
4. Measuring the functional tree similarity and displaying clone types.

Before we start explaining the methodology we present a brief introduction to ANTLR and similarity metrics.

3.1 Introduction to ANTLR (Another Tool for Language Recognition)

Terence Parr is the man behind ANTLR who is working with ANTLR since 1989. ANTLR is LL (*) parser generator that generates the parse trees for the program according to language freely available grammar [63]. Even though ANTLR is written in java, it generates lexer and parser that respectively perform lexical and semantic analysis to build the parse tree from the input files. In this research work, we generate the parse tree by using formal language description called grammar, along with lexer and parser. ANTLR generates various files like grammar tokens, lexer tokens, BaseListner, Listener, parse tree visitor and parse tree walker that can be used to process the parse tree according to our needs[64].

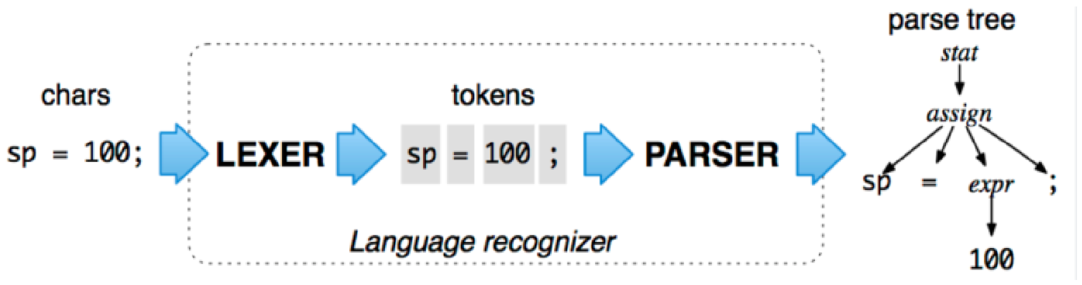


Fig. 1. Char stream to produce the parse tree

ANTLR parser creates lexer and parser based on the language grammar that later parses the input file based on the grammar. For example, we can write a grammar and include as file.g4 to parse the simple arithmetic expression like 100+2*34 as

```
grammar E;
start: (E NEWLINE)* ;
E:  E ‘*’ E | E ‘/’ E  | E ‘+’ E | E ‘-’ E  | INT    | (E);
NEWLINE : [\r\n]+ ;
IN     : [0-9]+ ;
```

Upon installing the latest ANTLR 4.8-v4 version and java (JDK) classpath set in the system, we have following ANTLR tutorial available at ANTLR site to generate parse tree for the arithmetic expression 2+3*4+(7-2).

- Execute the antlr command as “antlr file.g4” at command prompt we get various files such as file.tokens, fileBaseListener.java, fileLexer.java, fileListener.java, fileParser.java, file.interp.
- Compile all the generated java files to get the class files using the javac compiler.
- Run the java org.antlr.v4.gui.TestRig for the input file to obtain the parse tree as follows.

All the 3 steps have to be performed manually at the command prompt or in memory compilations can be done by using the automated APIs of “inmemantlr-tool” which has 14 releases so far and available at Github (Thome). Once we perform step 2 and get class files, using tree Listener class we can implement our own application to process the parse tree by creating the methods like getFirstChild(), getLastChild(), deChild(), getSubtree(), replaceSubtree() to access the basic ANTLR tree class.

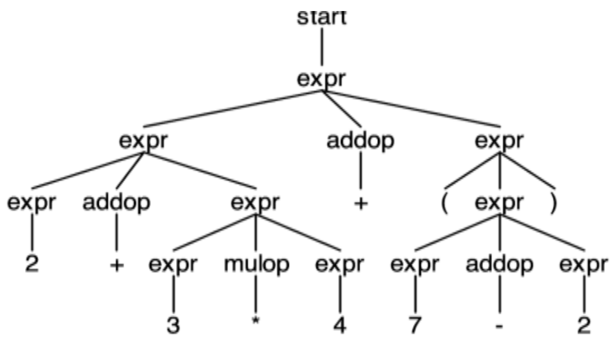


Fig. 2. Parse Tree for expression 2+3*4+(7-2)

3.2 *Introduction to similarity metrics*

According to [65] there exist several similarity metrics to find the similarity of the documents. Table 2 presents various similarity measures that classify the documents based on the data.

Table 2. Similarity metrics for document comparison

Sl.No	Name of algorithm	Concept used
1	Smith-Waterman	Edit Based
2	Levenstein distance	
3	Jaro	
4	Hamming	
5	Jaro	
6	Smith-Waterman	
7	Damerau-Levenstein	
8	Jaro-wrinkler	Token-based
9	Cosine similarity	
10	Jaccard	
11	Dice	
12	Word/N-gram	
13	Monge-Elkan	Hybrid
14	Soft –TFIDF	

These algorithms work by string matching or token matching by excluding the consideration about the position of tokens in the document hence they do not produce the proper results if applied on the input codes directly. Based on the motivation of [66] these metrics behave differently on the compiled and decompiled code.

We perform code similarity on the source code and corresponding ANTLR generated parse tree using a widely used code similarity metric cosine similarity which return 0 for no similarity and 1 for high similarity. The results of both similarity measures are shown in tables 3. Application of cosine similarity directly on two source codes addc and add1c gives matching similarity as 0.28 whereas we get similarity of 0.93 on the corresponding parse trees of addc and add1c. This significant difference is because of the fact that parse trees provide the syntactic and semantic information about the source code which provides evidence that application of cosine similarity on generated parse trees or subset of parse tree will work as accurate code clone detection technique.

Table 3. Similarity measures on source code and parse tree using cosine similarity.

Sl.No	**Source file**	**Destination file**	**Cosine similarity on source code**	**Cosine similarity on parse tree (dot file)**
1	addc	addc	1.0	1.0
2	addc	add1c	0.28	0.93
3	factdowhilec	factwhilec	0.67	0.94
4	factdowhilec	factforc	0.45	0.87
5	factforc	factwhilec	0.45	0.87
6	addcpp	addcpp	1.0	1.0
7	addcpp	add1cpp	0.30	0.97
8	factdowhilecpp	factwhilecpp	0.68	0.95
9	factdowhilecpp	factforcpp	0.54	0.94
10	factforcpp	factwhilecpp	0.54	0.93
11	addjava	addjava	1.0	1.0
12	Addjava	add1java	0.21	0.94
13	factdowhilejava	factwhilejava	0.84	0.90
14	factdowhilejava	factforjava	0.57	0.71
15	factforjava	factwhilejava	0.69	0.71

3.3 Repository Building and parse tree generation

Figure 3 presents the architecture of parse tree generation. It is the automated process of running the ANTLR tool through the java application that performs all the manual work explained in generating figure 2 from the input expression 2+3*4+(7-2). We have used “inmemantlr-tool-1.6” APIs available at maven central [67] to generate the parse tree in dot file. Since ANTLR provides grammars to parse all the language, the proposed method is language agnostic. We explain the clone detection and classification for C, CPP and Java codes that act as a evidence to language agnostic nature of the work. ANTLR tool generates various token and java files like Grammar.Tokens, Lexer.tokens, Lexer.java, Parser.java, Listner.java, BaseListner.java. All the java files are then compiled to get class files. Upon generation of java class files one has to provide input file (.C/.C++/.Java) files to generate the parse tree. This process can be done by writing the java application to read the grammar files and input file then calling the Listner class generated in the previous stage of ANTLR processing. This application can be made to work on the generated parse tree to extract the nodes of our interest.

Parse tree generation: as a case study we have considered small academic programs containing 135 C programs, 99 CPP programs and 33 codes of Java stored in a separate directory. The proposed method makes total of 9180 pair wise matching for C codes, 4950 comparisons for CPP codes and 561 comparisons for Java codes.

```
#include<stdio.h>
//prints hello
main()
{
printf("Hello World");
}
//end of program
```

C-Code

```
#include <iostream.h>
//prints hello
main()
{
cout << "Hello World!";
return 0;
}
//end of program
```

CPP-code

```
import java.util.Scanner
//prints hello
class Hello
{
public static void main(String[] args)
{
System.out.println(“Hello World”);
}
}
//end of program.
```

Java-Code

The advantage of using functional tree extraction is that, it automatically performs pre-processing step adopted by previous clone detection techniques to eliminate program header, and comments. Since the size of generated functional tree is very long, we take small example of printing “hello world” from our data set. The basic principle of ANTLR is to generate the complete parse tree that includes the node information of all the lines present in the input code. Functional tree is the subset of parse tree that includes the generation of nodes only for the line number 4 to 6 for the above C-code, line number 4-7 for above CPP-code and line number 4-8 for above Java-code. These are the statements that represent the main functionality of the code, and hence the tree generated by including all the information in the main function of C and CPP code and the class body of java code is named as functional tree.

Figure 4, 5 and 6 respectively shows the ANTLR parse tree for the above case studies of C, CPP and java code. Functional tree for both C and CPP code starts with the node name *statementseq* that contains node information for line number 5 for C and line number 5 and 6 for CPP. Sub tree generated by extracting only the lines of function definition is termed as functional tree in our paper. The parse tree of java code starts with node name *compilationunit* followed by left subtree *importDeclartion* that corresponds to importing the packages. A class definition starts from the right subtree with node name *typeDeclartion* followed by *classDeclaration*, *classBody*, *classBodyDeclaration*. The left children of this node correspond to modifier public static. The class definition starts from the rightmost child *memberDeclaration* followed by *methodDeclaration*. So far all the nodes of the parse tree represent in building the header information for the java class. These nodes just bring structural information of java code.

We first allow ANTLR to generate the complete parse tree by calling the base class method of antlr tree on the listener variable *dt*.

TreeListner dt;

dt.getParseTree() creates the ANTLR parse tree. We use common grammar file CPP14.g4 to parse C and CPP code to get parse tree. Parse tree for C and CPP code starts with node name *translationunit* followed by *declarationseq*, *declaration*, *functiondefinition*, *functionbody* and so on.

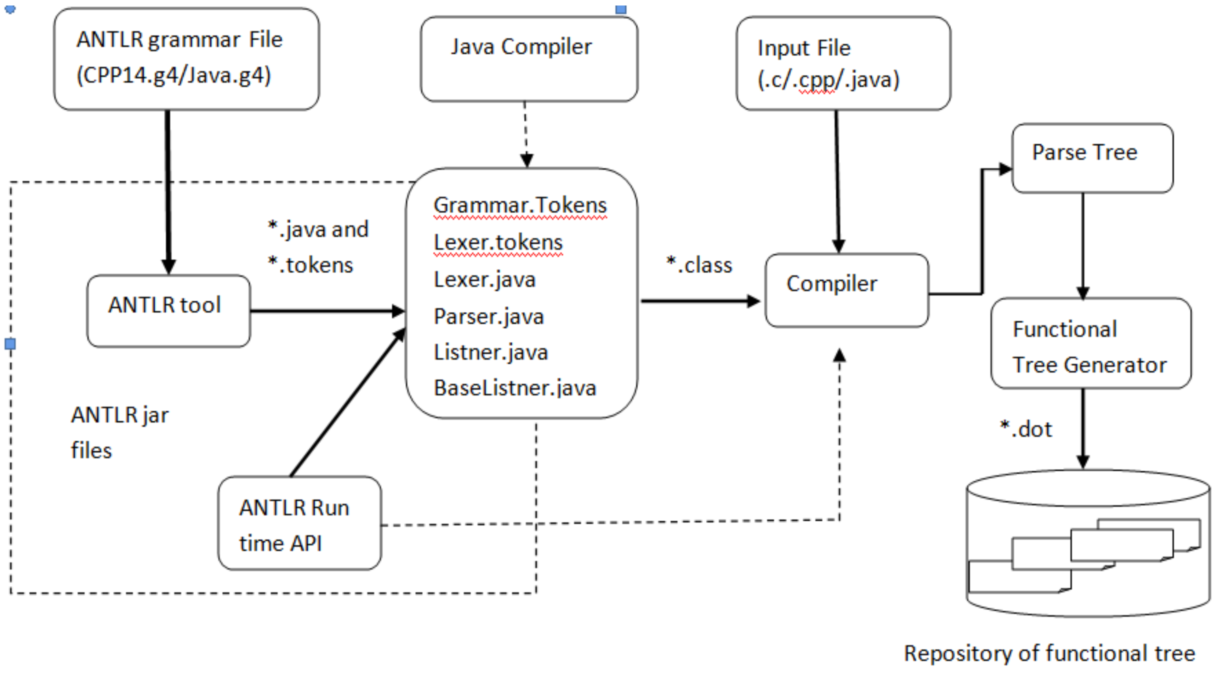


Fig.. 3. Repository Building and functional tree generation

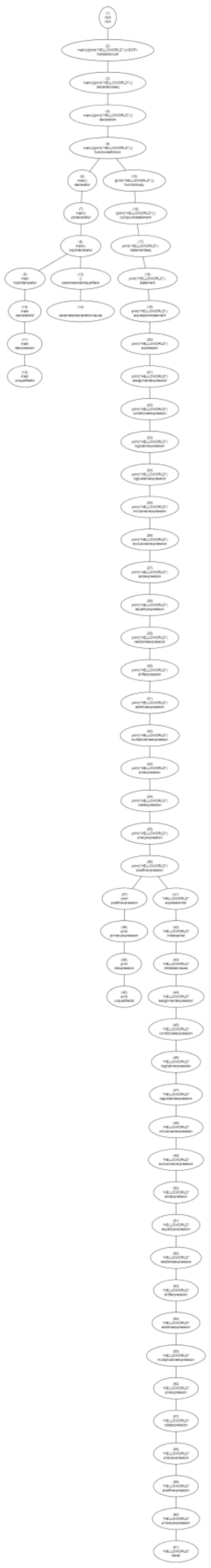


Fig. 4. Parse tree of C-code

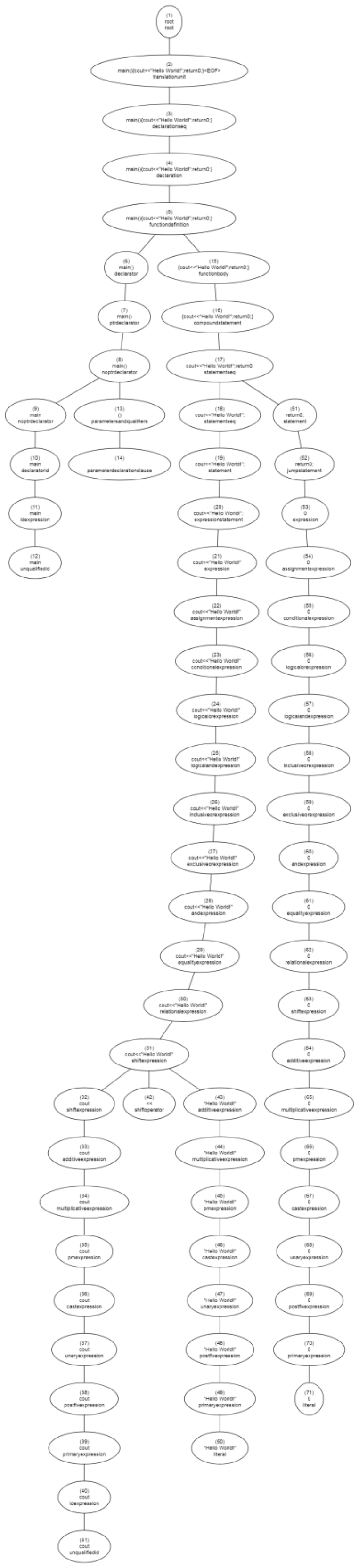

(1) root root
(2) main(){cout<<"Hello World!";return0;}<EOF> translationunit
(3) main(){cout<<"Hello World!";return0;} declarationseq
(4) main(){cout<<"Hello World!";return0;} declaration
(5) main(){cout<<"Hello World!";return0;} functiondefinition
(6) main() declarator
(7) main() ptrdeclarator
(8) main() noptrdeclarator
(9) main noptrdeclarator
(10) main declaratorid
(11) main idexpression
(12) main unqualifiedid
(13) () parametersandqualifiers
(14) parameterdeclarationclause
(15) {cout<<"Hello World!";return0;} functionbody
(16) {cout<<"Hello World!";return0;} compoundstatement
(17) cout<<"Hello World!";return0; statementseq
(18) cout<<"Hello World!"; statementseq
(19) cout<<"Hello World!"; statement
(20) cout<<"Hello World!"; expressionstatement
(21) cout<<"Hello World!" expression
(22) cout<<"Hello World!" assignmentexpression
(23) cout<<"Hello World!" conditionalexpression
(24) cout<<"Hello World!" logicalorexpression
(25) cout<<"Hello World!" logicalandexpression
(26) cout<<"Hello World!" inclusiveorexpression
(27) cout<<"Hello World!" exclusiveorexpression
(28) cout<<"Hello World!" andexpression
(29) cout<<"Hello World!" equalityexpression
(30) cout<<"Hello World!" relationalexpression
(31) cout<<"Hello World!" shiftexpression
(32) cout shiftexpression
(33) cout additiveexpression
(34) cout multiplicativeexpression
(35) cout pmexpression
(36) cout castexpression
(37) cout unaryexpression
(38) cout postfixexpression
(39) cout primaryexpression
(40) cout idexpression
(41) cout unqualifiedid
(42) << shiftoperator
(43) "Hello World!" additiveexpression
(44) "Hello World!" multiplicativeexpression
(45) "Hello World!" pmexpression
(46) "Hello World!" castexpression
(47) "Hello World!" unaryexpression
(48) "Hello World!" postfixexpression
(49) "Hello World!" primaryexpression
(50) "Hello World!" literal
(51) return0; statement
(52) return0; jumpstatement
(53) 0 expression
(54) 0 assignmentexpression
(55) 0 conditionalexpression
(56) 0 logicalorexpression
(57) 0 logicalandexpression
(58) 0 inclusiveorexpression
(59) 0 exclusiveorexpression
(60) 0 andexpression
(61) 0 equalityexpression
(62) 0 relationalexpression
(63) 0 shiftexpression
(64) 0 additiveexpression
(65) 0 multiplicativeexpression
(66) 0 pmexpression
(67) 0 castexpression
(68) 0 unaryexpression
(69) 0 postfixexpression
(70) 0 primaryexpression
(71) 0 literal

Fig. 5. Parse tree of CPP-code

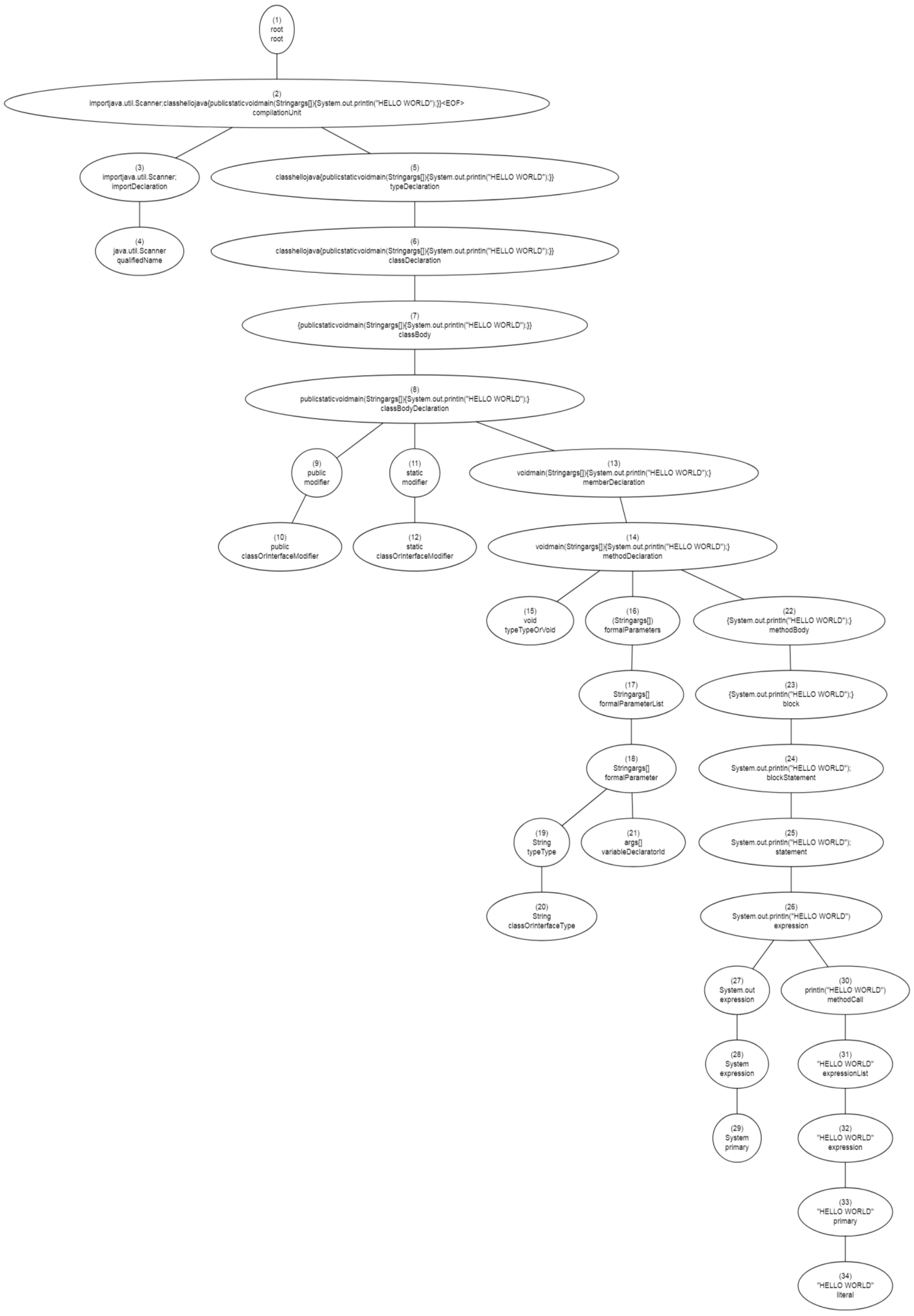

(1) root root
(2) importjava.util.Scanner;classhellojava{publicstaticvoidmain(Stringargs[]){System.out.println("HELLO WORLD");}}<EOF> compilationUnit
(3) importjava.util.Scanner; importDeclaration
(4) java.util.Scanner qualifiedName
(5) classhellojava{publicstaticvoidmain(Stringargs[]){System.out.println("HELLO WORLD");}} typeDeclaration
(6) classhellojava{publicstaticvoidmain(Stringargs[]){System.out.println("HELLO WORLD");}} classDeclaration
(7) {publicstaticvoidmain(Stringargs[]){System.out.println("HELLO WORLD");}} classBody
(8) publicstaticvoidmain(Stringargs[]){System.out.println("HELLO WORLD");} classBodyDeclaration
(9) public modifier
(10) public classOrInterfaceModifier
(11) static modifier
(12) static classOrInterfaceModifier
(13) voidmain(Stringargs[]){System.out.println("HELLO WORLD");} memberDeclaration
(14) voidmain(Stringargs[]){System.out.println("HELLO WORLD");} methodDeclaration
(15) void typeTypeOrVoid
(16) (Stringargs[]) formalParameters
(17) Stringargs[] formalParameterList
(18) Stringargs[] formalParameter
(19) String typeType
(20) String classOrInterfaceType
(21) args[] variableDeclaratorId
(22) {System.out.println("HELLO WORLD");} methodBody
(23) {System.out.println("HELLO WORLD");} block
(24) System.out.println("HELLO WORLD"); blockStatement
(25) System.out.println("HELLO WORLD"); statement
(26) System.out.println("HELLO WORLD") expression
(27) System.out expression
(28) System expression
(29) System primary
(30) println("HELLO WORLD") methodCall
(31) "HELLO WORLD" expressionList
(32) "HELLO WORLD" expression
(33) "HELLO WORLD" primary
(34) "HELLO WORLD" literal

Fig. 6. Parse tree of Java-code

3.4 Functional tree generation

This section provides the technical details to extract the functional tree from the parse tree. From the figure 4 and 5 we find that parse tree of C and CPP code starts with the node name translation unit followed by declarationseq, declaration, functiondefintion along with all these nodes till the left descending children of functiondefintion corresponds to C/CPP header. These rules do not contribute to finding the functional similarity of any program. The main computation or functionality of the code starts from line number 5 which corresponds to the the node name “statementseq” in the parse tree. Hence we generate the functional tree by extracting the subtree with node name statementseq for C/CPP parse tree. The code for extracting the functional tree from ANTLR generated parse tree is presented below.

parseTree=dt.getParseTree().getSubtrees(n-> n.getRule().equals("statementseq")).iterator().next();

similarly the parse tree of java code starts with node name compilationunit followed by left subtree importDeclartion that corresponds to importing the packages. A class definition starts from the right subtree with node name typeDeclartion followed by classDeclaration, classBody, classBodyDeclaration. The left children of this node corresponds to modifier public static. The class definition starts from the rightmost child memberDeclaration followed by methodDeclaration. So far all the nodes of the parse tree represent in building the header information for the java class. These nodes just bring structural information of java code. The functionality of the java code is written inside the two curly braces of the main method from the line number 7 of Java code. Hence functional tree for Java code can be extracted by passing the argument as node name “block” to equals method on getRule() as follows.

parseTree=dt.getParseTree().getSubtrees(n-> n.getRule().equals("block")).iterator().next();

the figures 8, 9 and 10 shows the corresponding functional trees for the parse tree 4, 5 and 6 respectively.

3.5 Finding the functional tree similarity and displaying clone types

The parse Functional tree generated in the previous phase can be stored in dot file using the APIs of “inmemantlr-tool” (Thome) that can be used for comparison using any of the natural language processing techniques such as Levenstein distance, Edit distance, Hamming distance, Longest Common Subsequence. By looking at the fact that parse tree generates a lot of information through repetitive rules we consider term frequency inverse document frequency with cosine similarity to find the similarity between two different parse tree representations in a dot file. The sequence of finding the functional tree similarity is shown in the figure 7.

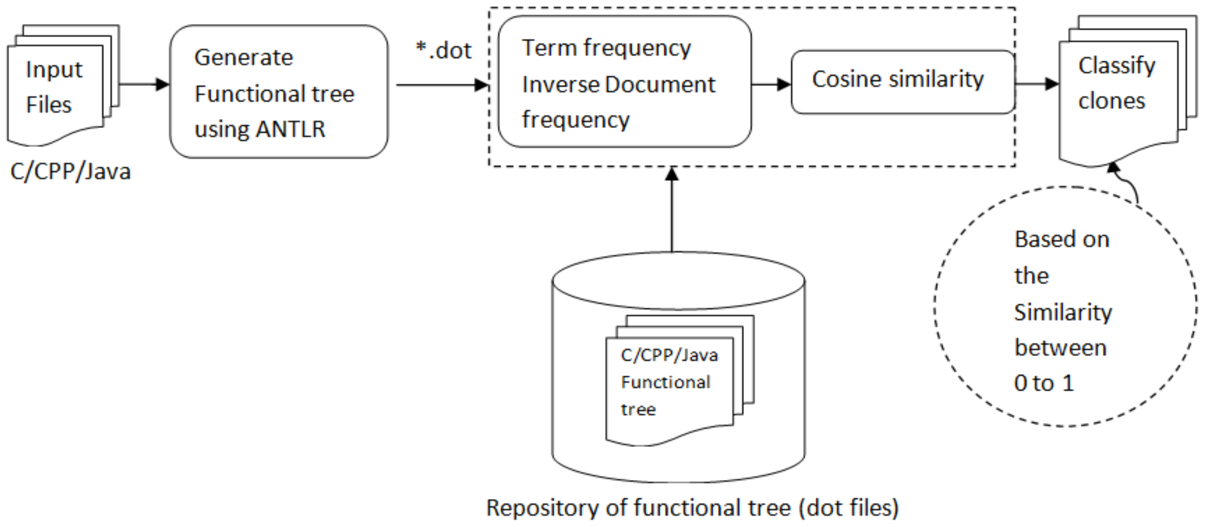

Input Files
C/CPP/Java
Generate Functional tree using ANTLR
*.dot
Term frequency Inverse Document frequency
Cosine similarity
Classify clones
Based on the Similarity between 0 to 1
C/CPP/Java Functional tree
Repository of functional tree (dot files)

Fig. 7. Finding functional tree similarity using TFID and cosine similarity

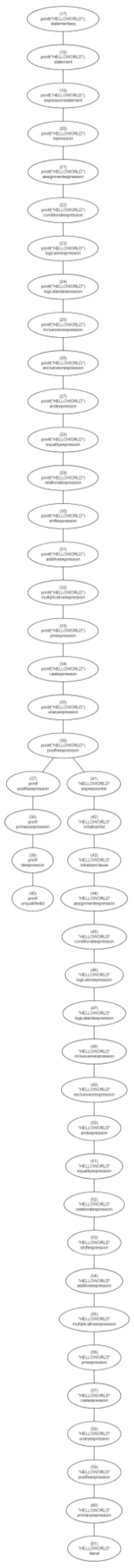

(17) printf("HELLOWORLD"); statementseq
(18) printf("HELLOWORLD"); statement
(19) printf("HELLOWORLD"); expressionstatement
(20) printf("HELLOWORLD") expression
(21) printf("HELLOWORLD") assignmentexpression
(22) printf("HELLOWORLD") conditionalexpression
(23) printf("HELLOWORLD") logicalorexpression
(24) printf("HELLOWORLD") logicalandexpression
(25) printf("HELLOWORLD") inclusiveorexpression
(26) printf("HELLOWORLD") exclusiveorexpression
(27) printf("HELLOWORLD") andexpression
(28) printf("HELLOWORLD") equalityexpression
(29) printf("HELLOWORLD") relationalexpression
(30) printf("HELLOWORLD") shiftexpression
(31) printf("HELLOWORLD") additiveexpression
(32) printf("HELLOWORLD") multiplicativeexpression
(33) printf("HELLOWORLD") pmexpression
(34) printf("HELLOWORLD") castexpression
(35) printf("HELLOWORLD") unaryexpression
(36) printf("HELLOWORLD") postfixexpression
(37) printf postfixexpression
(38) printf primaryexpression
(39) printf idexpression
(40) printf unqualifiedid
(41) "HELLOWORLD" expressionlist
(42) "HELLOWORLD" initializerlist
(43) "HELLOWORLD" initializerclause
(44) "HELLOWORLD" assignmentexpression
(45) "HELLOWORLD" conditionalexpression
(46) "HELLOWORLD" logicalorexpression
(47) "HELLOWORLD" logicalandexpression
(48) "HELLOWORLD" inclusiveorexpression
(49) "HELLOWORLD" exclusiveorexpression
(50) "HELLOWORLD" andexpression
(51) "HELLOWORLD" equalityexpression
(52) "HELLOWORLD" relationalexpression
(53) "HELLOWORLD" shiftexpression
(54) "HELLOWORLD" additiveexpression
(55) "HELLOWORLD" multiplicativeexpression
(56) "HELLOWORLD" pmexpression
(57) "HELLOWORLD" castexpression
(58) "HELLOWORLD" unaryexpression
(59) "HELLOWORLD" postfixexpression
(60) "HELLOWORLD" primaryexpression
(61) "HELLOWORLD" literal

Fig. 8. Functional tree of C-code

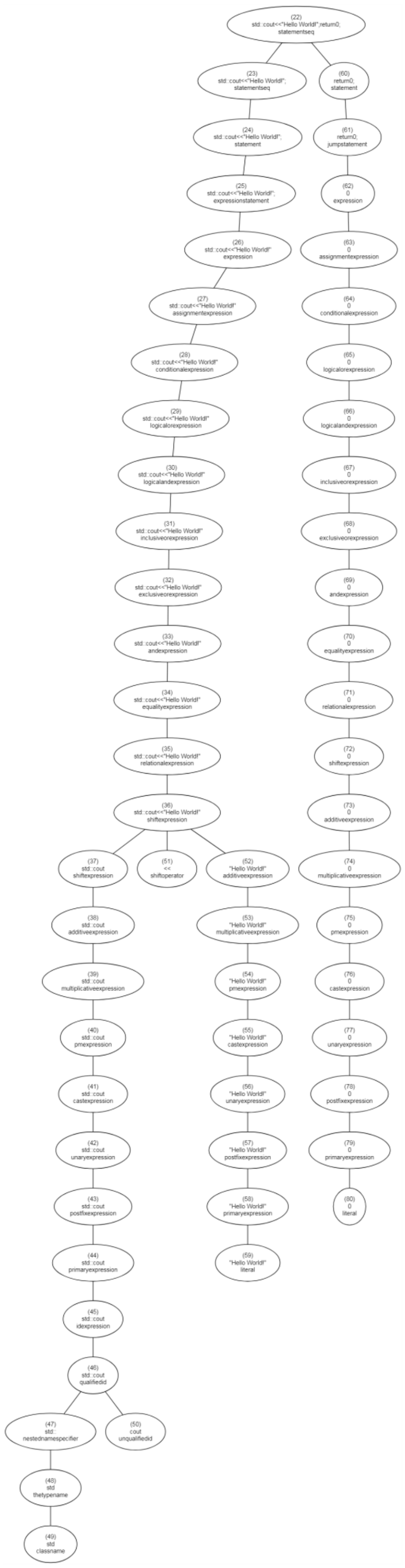

(22) std::cout<<"Hello World!";return0; statementseq
(23) std::cout<<"Hello World!"; statementseq
(60) return0; statement
(24) std::cout<<"Hello World!"; statement
(61) return0; jumpstatement
(25) std::cout<<"Hello World!"; expressionstatement
(62) 0 expression
(26) std::cout<<"Hello World!" expression
(63) 0 assignmentexpression
(27) std::cout<<"Hello World!" assignmentexpression
(64) 0 conditionalexpression
(28) std::cout<<"Hello World!" conditionalexpression
(65) 0 logicalorexpression
(29) std::cout<<"Hello World!" logicalorexpression
(66) 0 logicalandexpression
(30) std::cout<<"Hello World!" logicalandexpression
(67) 0 inclusiveorexpression
(31) std::cout<<"Hello World!" inclusiveorexpression
(68) 0 exclusiveorexpression
(32) std::cout<<"Hello World!" exclusiveorexpression
(69) 0 andexpression
(33) std::cout<<"Hello World!" andexpression
(70) 0 equalityexpression
(34) std::cout<<"Hello World!" equalityexpression
(71) 0 relationalexpression
(35) std::cout<<"Hello World!" relationalexpression
(72) 0 shiftexpression
(36) std::cout<<"Hello World!" shiftexpression
(73) 0 additiveexpression
(37) std::cout shiftexpression
(51) << shiftoperator
(52) "Hello World!" additiveexpression
(74) 0 multiplicativeexpression
(38) std::cout additiveexpression
(53) "Hello World!" multiplicativeexpression
(75) 0 pmexpression
(39) std::cout multiplicativeexpression
(54) "Hello World!" pmexpression
(76) 0 castexpression
(40) std::cout pmexpression
(55) "Hello World!" castexpression
(77) 0 unaryexpression
(41) std::cout castexpression
(56) "Hello World!" unaryexpression
(78) 0 postfixexpression
(42) std::cout unaryexpression
(57) "Hello World!" postfixexpression
(79) 0 primaryexpression
(43) std::cout postfixexpression
(58) "Hello World!" primaryexpression
(80) 0 literal
(44) std::cout primaryexpression
(59) "Hello World!" literal
(45) std::cout idexpression
(46) std::cout qualifiedid
(47) std:: nestednamespecifier
(50) cout unqualifiedid
(48) std thetypename
(49) std classname

Fig. 9. Functional tree of CPP-code

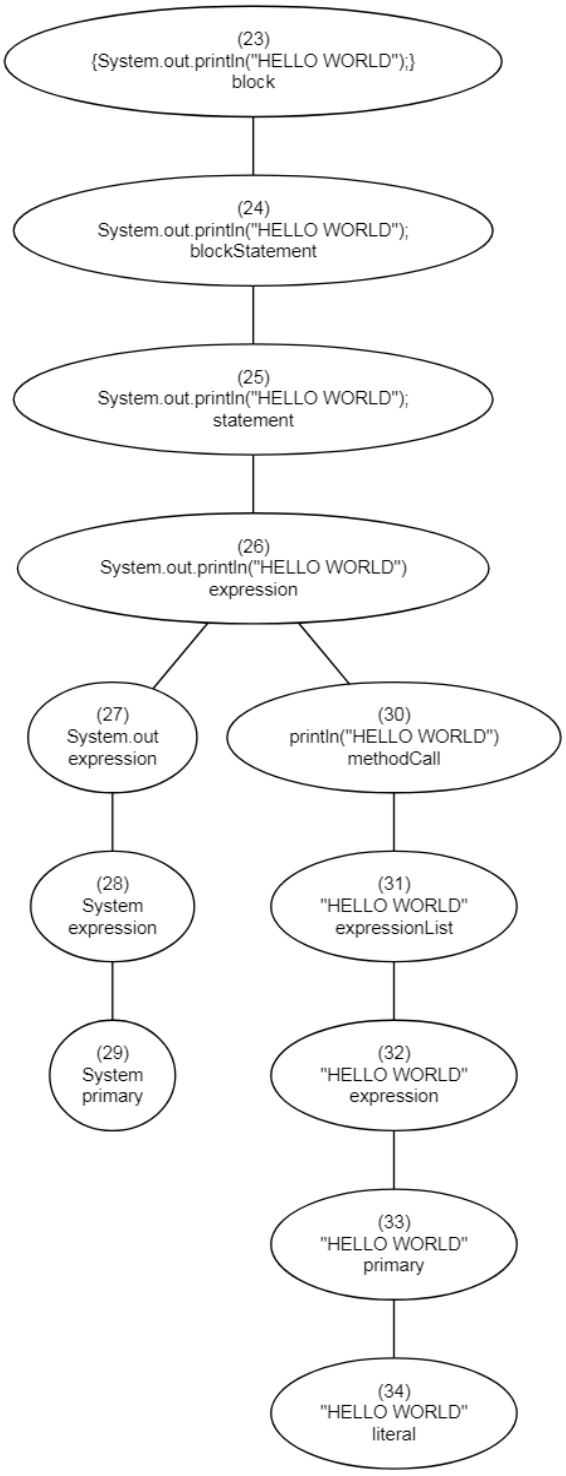


Fig.10. Functional tree of Java-code

The **Cosine Similarity** function

[68] Is widely used to compute the similarity between two given term vectors. Which is ratio of the inner product (v1•v2) to the product of vector length. Similarity between two vectors v1, v2 is given by

$$cos(v1, v2) = (v1\ X\ v2)/||v1||||v2|| \tag{1}$$

Where, ||v1||and ||v2|| are Euclidean norms of vector V.?

From table 3 results we can easily find the cosine similarity among two documents by inputting the vectors v1 andv2.

$$v1 = TF.IDF - A \ \ and\, vector\ \ v2 = TF.IDF - B \tag{2}$$

Following pseudo code, statements find cosine similarity between vectors v1 and v2 [68].

```
$For\ i = 1\ to\ lenghth(TF.IDF - A)$
  {
      $dotProduct \mathrel{+}= TF - IDF - A[i] * TF - IDF - B[i]$
      $magnitude1 \mathrel{+}= Math.pow(TF - IDF - A\ [i], 2);$
      $magnitude2 \mathrel{+}= Math.pow(TF - IDF - B\ [i], 2);$
  }
$magnitude1 = \sqrt{magnitude1}$
$magnitude2 = \sqrt{magnitude2}$
```

Finally, we get similarity between Doc A and Doc B as

$$cosine(TF - IDF - A, TF - IDF - B) = \frac{dotProduct}{magnitude1 * magnitude2} \tag{3}$$

4. Vector Representation

The Term Frequency Inverse Document frequency (tf.idf) [69] is a hash map-like data structure that finds frequencies of term occurrence in a document with the relative location. It is the SVM based metric that is used for document processing and comparison. Term frequency gives the count of each token in a document and inverse document frequency gives the uniqueness of each token in a document. The advantage of using this approach is it gives the relative position of every token in a document.

Equation (1) shows the use of TFIDF in our work. TF-IDF is used to find how relevant a term is in a document (Goel, 2014). Where TF measures how frequently a term occurs in a document and IDF gives log (number of documents/ number of documents with the term in it) (Elhadad, 2018).

$$tf - idf(t, D) = tf(t, D).idf(t, D) \tag{4}$$

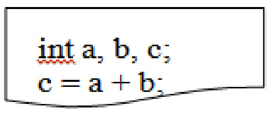
Where

$$tf(t, D) = freq\ t \in D \tag{5}$$

$$idf(t, D) = 1 + \log\left(N/|\{d \in D : t \in d\}|\right) \tag{6}$$

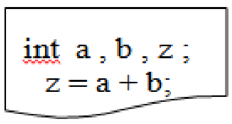
Where N is the total number of documents?

Table 4 shows the results obtained by combining TF and IDF for the below code-A and code-B.



```
int a, b, c;
c = a + b;
```

Code-A



```
int  a , b , z ;
 z = a + b;
```

Code-B

Significance of using TF-IDF is term frequency identifies the words having a unique occurrence of word in the documents. Term frequency is calculated by counting the tokens in the respective documents like for int in code-A is calculated as 1/7 which denotes the term int occurs once among 7 readable tokens. Similarly for a, b, c, and z in a code-A is 2/7, 2/7, 2/7, and 0/7 respectively. In the same way the term frequencies for all the tokens in code-B is calculated and presented in 5th columns of table 4. Inverse document frequency for each tokens of code-A and code-B are presented in column 6 and 7 respectively. Finally tf-idf value 0.4837 signifies uniqueness of token c in code-A and token z in code-B.

Table 4. TF-IDF for the tokens of code-A and code-B

Token	code A	code B	TF-A	TF-B	IDF-A	IDF-B	TF.IDF-A	TF.IDF-B
int	1	1	1/7=0.142	1/7=0.142	1+Log(2/2)=1	1+Log(2/2)=1	0.1428	0.1428
a	2	2	2/7=0.285	2/7=0.285	1+Log(2/2)=1	1+Log(2/2)=1	0.2857	0.2857
b	2	2	2/7=0.285	2/7=0.285	1+Log(2/2)=1	1+Log(2/2)=1	0.2857	0.2857
c	2	0	2/7=0.285	0/7=0	1+Log(2/1)=1.69	1+log(2/2)=1	0.4837	0
z	0	2	0/7=0	2/7=0.285	1+Log(2/2)=1	1+log(2/1)=1.69	0	0.4837

4.1 Classification of clone types

As the last step, we classify clone types by manually validating the matching percentage of all four clone types according to the following thresholds. The following judgment criteria are based on the validation of more than 4000 known clone pairs taken from sanfoundry.com. The classification threshold is based on the similarity value as shown in the table 5.

Table 5. Thresholds of clone type classification based on similarity values

Similarity matching	Clone type
1	1
>=0.95 && < 0.99	2
>=0.80 and <=0.90	4
>=0.65 and <0.80	3

5. Experimental Setup and Dataset Creation

The experiment is conducted on the Windows 7 operating system with Intel core 2 duos CPU having 2 GHZ speed and 2GB RAM on three sets of data samples. We plan 3 set of case studies as shown below.

i) **Dataset-1:** To understand the parsing ability of ANTLR, and clone classification accuracy of cosine similarity, we initially validated our approach on 135 of C codes, 99 C++ codes and 33 java codes with average line count of 5. We recorded time taken to parse the various inputs files.

ii) **Dataset-2:** We have collected sample of C, C++ and Java codes from sanfoundry.com which contains 1000 algorithm based codes of C, C++ and Java. We have edited all the codes according to clone type definitions to get total of 4234 true clone pairs of each code samples.

iii) **Dataset-3:** Next we perform systematic GitHub-web scrapping on 73,075 active repositories of C and 86,505 active repositories of C++ to collect 12,600 sample clone pairs of C and 14,480 clone pairs of C++ with average line count of 15. Since BigCloneBench is the standard data set for java, we use the sample dataset similar to that of (Wang, 2020) which contains 9,134 java codes type. The table 6 presents the details of three datasets on various known clone pairs.

Table 6. Number of clone types for dataset 1, 2, and 3.

Dataset	Language	Clone Type			
		T1	T2	T3	T4
1	C	24	28	48	36
	C++	28	27	22	22
	Java	8	9	8	8
2	C	1,112	1,026	896	1,200
	C++	1,286	950	1,018	980
	Java	1,448	1,020	860	906
3	C	4,238	4,290	1,996	2,076
	C++	5,432	5,180	1,846	2,022
	Java	3,298	2,696	1,860	1,288
Total		**16,874**	**15,226**	**8,554**	**8,538**

6. Results and Analysis

In section 3, we have applied TF-IDF cosine similarity directly on the C source files and found the results in table 3. The obtained similarity measure can only be used to find the type 1 clones and do not support detection of other three clone types. We analyze the results of parse tree similarity using TFID-Cosine similarity by presenting the precision and recall values that act as a validation parameters for our proposed approach.

Precision tries to find what proportion of positive instances where correct. (https://developers.google.com/) For instance we know that add1c and addc are type 2 clones. If we obtain type 2 as the result of classification then it is a true positive result otherwise it is said to be false-positive result. With this basic knowledge, we present precision as

Precision=True Positive/ True Positive + False Positive

Recall tries to find how many positives are identified correctly.

Recall= True Positive / True Positive + False Negative

In terms of precession and recall by selecting the few results randomly from C, CPP and Java to understand clone classification accuracy is as follows.

6.1 Results on Dataset-1

We present few samples of dataset 1 for both C and C++ in table 7 and 8 respectively using CPP14.g4 grammar.

Table 7. Functional tree similarity for C programs using TFID-cosine similarity

Sl. No	Source file	Destination file	Percentage of matching	Result obtained	Expected Result
1	add1C	add1C	1.0	Type1	Type1
2	add1C	addC	0.93	Type2	Type2
3	add1C	add2C	0.43	Type 3	Type 3
3	add1C	factdowhileC	0.26	Not a clone	Not a clone
4	add1C	factforC	0.25	Not a clone	Not a clone
5	add1C	factwhileC	0.26	Not a clone	Not a clone
6	add1C	helloC	0.12	Not a clone	Not a clone
7	factdowhileC	factforC	0.88	Type4	Type4
8	factdowhileC	factwhileC	0.95	Type4	Type4
9	factforC	factwhileC	0.87	Type4	Type4

Table 8. Functional tree similarity for CPP programs using TFID-cosine similarity.

Sl. No	Source file	Destination file	Percentage of matching	Result obtained	Expected Result
1	add1cpp	add1cpp	1.0	Type1	Type1
2	add1cpp	addcpp	0.96	Type2	Type2
3	add1cpp	add2cpp	0.46	Type3	Type3
4	add1cpp	factdowhilecpp	0.31	Not a clone	Not a clone
5	add1cpp	factforcpp	0.33	Not a clone	Not a clone
6	add1cpp	factwhilecpp	0.31	Not a clone	Not a clone
7	add1cpp	hellocpp	0.08	Not a clone	Not a clone
8	factdowhilecpp	factforcpp	0.91	Type4	Type4
9	Factdowhilecpp	factwhilecpp	0.95	Type4	Type4
10	Factforcpp	factwhilecpp	0.93	Type4	Type4

Table 9 shows the similarity of two functional tree's of ANTLR generated functional tree for Java files using JavaLexer.g4 and JavaParser.g4 grammar.

Table 9. Functional tree similarity for Java programs using TFID-cosine similarity

Sl. No	Source file	Destination file	Percentage of matching	Result obtained	Expected Result
1	add1java	add1java	1.0	Type1	Type1
2	add1java	addjava	0.92	Type2	Type2
3	add1java	factdowhilejava	0.22	Not a clone	Not a clone
4	add1java	factforjava	0.20	Not a clone	Not a clone
5	add1java	factwhilejava	0.22	Not a clone	Not a clone
6	add1java	hellojava	0.11	Not a clone	Not a clone
7	factdowhilejava	factforjava	0.82	Type4	Type4
8	factdowhilejava	factwhilejava	0.92	Type4	Type4
9	factforjava	factwhilejava	0.82	Type4	Type4

The experiment was extended to total of 135 C codes with 9180 comparisons, 99 C++ codes with 4950 comparisons and 33 Java codes with 561 comparisons were made to record the following precision and recall is shown in the table 10. The results show excellent precision and recall for type 1 and 2 but yields false positive values for type 3 and 4. The table 10 shows the improvement in the precision and recall of type 3 and type4.

Table 10. Precision and recall using cosine similarity

Clone types	Precision	Recall
Type1	(267/267) 100	(267/267) 100
Type2	(267/267) 100	(267/267) 100
Type3	(263/267)98.50	(252/267) 98.50
Type4	(262/267) 98.12	(234/267) 98.12

One noted good thing about TFID-cosine similarity is that it takes just 40 seconds to compare 9180 C code comparisons and takes 78 minutes to compare equivalent functional trees which make it computationally infeasible for exhaustive comparisons.

6.2 Results on Dataset-2

This section presents the classification results for the known clone pairs of dataset-2 containing exactly 4234 clone pairs of C, C++ and Java codes as shown in table 5. We have applied exhaustive comparison on the each dot file containing functional tree using TF-IDF cosine similarity. The results of clone classification for entire dataset-2 are presented in table 11 below.

Table 11. Clone classification result for dataset-2

Language	**Expected clone type results**			
	Type-1	**Type-2**	**Type-3**	**Type-4**
C	1,112/1,112	1,026/1,026	882/896	1,142/1,200
C++	1,286/1,286	943/950	1,002/1,018	962/980
Java	1,448/1,448	1,013/1,020	842/860	894/906
Average precision	**100**	**99.53**	**98.26**	**97.14**

The results presented in table 11 proves that the proposed method works very well in detecting the clone types 1 and 2 on all three languages with almost 100% precision. We have also obtained excellent results in detecting type 3 and 4 with precision of 98.26% and 97.14% respectively that outperforms the existing tree based clone detection tools for detecting type 3 and 4 clone types.

6.3 Results on Dataset-3

This section presents the classification results for the known clone pairs of dataset-3 containing exactly 12,600 sample clone pairs of C and 14,480 clone pairs of C++ with average line count of 15 and 9,134 java codes from BigCloneBench as shown in table 5. We have applied exhaustive comparison on the each dot file containing functional tree using TF-IDF cosine similarity. The results of clone classification for entire dataset-3 are presented in table 12 below.

Table 12. Clone classification result for dataset-3

Language	**Expected clone type results**			
	Type-1	**Type-2**	**Type-3**	**Type-4**
C	4,238/4,238	4,290/4,290	1,996/1,996	2,076/2,076
C++	5,432/5,432	5,046/5,180	1,824/1,846	2,008/2,022
Java	3,289/3,298	2,582/2,696	1,698/1,860	1,176/1,288
Average precision	**99.9**	**97.96**	**96.7**	**97.66**

The results presented in table 12 proves that the proposed method works very well in detecting the clone types 1 and 2 on all three languages with almost 98.93% precision. We have also obtained encouraging results in detecting type 3 and 4 with average precision of 97.18%.

7. Comparative Study

In this section, we present the recent tree based techniques on the two important parameters such as clone type classification and language supported by tool. The comparison is presented on table 13.

Table 13. Comparison of tree based techniques

Sl.No	**Author/Citation**	**Language Supported**	**Clone Type detection**
1	Yang, 2018 [44]	Java	Function
2	Pati, 2017 [70]	ArgoUML	1,3
3	Lavoie, 2019 [45]	Java	3,4
4	Clonemerge(Narasimhan, 2015)	C/C++	Near miss
5	Y. Yang, 2018 [71]	Java	1,2,3
6	J. Zeng, 2019 , [72]	Java	1,2,3,4
7	OOP, D. Li, 2014 [73]	Java/PHP	1,2
8	(Thompson, 2011) [74]	Erlang	Structural
9	Wang, 2020 [1]	Java(IJDataset)	1,2,3,4
10	**Proposed method**	**C, C++ and Java**	**1,2,3,4**

Table 13 provides evidence that only the work of J. Zeng, 2019[72] and Wang, 2020[1] have classified all 4 clone types but the major issue is they are limited to Java code for clone detection. Apart from this both the works are computationally infeasible. We are the first to apply the clone detection and classification to three languages. The evidences of current study gives the hint that since the grammars for parsing all the languages are freely made available by ANTLR, the current methodology can be extended to include all the programming languages that are in practice by all the universities in the world.

Next, we present table 14 to compare our study with the open source tree based techniques to record the precision in detecting the clone pairs on our dataset-3. We have considered Deckard[35], iclones [27], ccfx[75] and re-use the accuracy data presented in FA-AST+GMN[1] to validate our approach.

Table 14. Average precision comparison for tree based clone detection

Technique/Tool	**Precision (%)**
Deckard	94.34
iClones	91.24
ccfx	95
FA-AST+GMN	95
Proposed Approach	**98.077**

From the above table 14, we can conclude that, the proposed method is the more reliable, complete and language agnostic in detecting clone with the excellent average precision of 98.07% in detecting all four clone types thereby act as a proper validation tool for detecting learning levels by the students in submitted code assignments. The string based document comparison techniques for plagiarism detection in Arabic languages presented by Mohd. Binai [76] and plagiarism detection for Kurdish language proposed by Karzan Wakil et al. [77] can only detect type 1 and type 2 clones but fail to detect type 3 and type 4 clones because of the semantic factor involved in detecting code plagiarism hence cannot be applied to code plagiarism detection.

8. Conclusion

This research paper proposes more realiable code plagiarism detector by implementing complete and language-agnostic clone detection for C, C++ and Java languages on the datasets containing average line count of 5, 15, and 32 respectively. The technique works by extracting a functional tree from the ANTLR generated parse tree to eliminate the need of preprocessing stage employed by previous clone detection tools. We employ TF-IDF cosine similarity on the generated functional tree in dot file that takes less than 3 seconds to match the clone pairs of 1396 codes which provides evidence that the method works on large scale repositories. The results prove that classification of clone types-1 and 2 are done with 100% precision and precision of 98.50 and 98.12 respectively for detecting type 3 and type 4 clones on 30 small codes of C, C++, and Java. Proposed technique exhibits the precision of 99.9% for type-1, 97.96% for type-2, 96.7% for type-3, and 97.66% for type-4 clone detection on the C, C++ and Java programs crawled from active repositories of Github. As ANTLR grammars are made available freely, the proposed model can be extended to include other existing programming languages to detect code plagiarism with clone types classified to get proper validation in submitted coding assignments.

References

[1] Wang, W. L. (2020). Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 261-271). IEEE.

[2] Krinke, J. (2001). Identifying similar code with program dependence graphs. *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, (pp. 301–309). Stuttgart, Germany.

[3] Baxter, I. D. (1998). Clone detection using abstract syntax trees. *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 368--377). IEEE.

[4] Ducasse, S. R. (1999). A language independent approach for detecting duplicated code. *International Conference on Software Maintenance-1999 (ICSM'99)* (pp. 109-118). IEEE.

[5] Godfrey, C. K. (2006). clones considered harmful. *Reverse Engineering (WCRE'06)* (pp. 19-28). Benevento, Italy: IEEE.

[6] Chanchal Kumar Roy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR*, 64-68.

[7] Chanchal K. Roy, J. R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 470-495.

[8] Dhavleesh Rattan, R. B. (2013). Software clone detection: A systematic review. *Information and Software Technology*, 1165-1199.

[9] Ain, Q. U. (2019). A systematic review on code clone detection. . *IEEE access*, 86121-86144.

[10] Chivers, K. a. (n.d.). *https://www.researchgate.net/publication/337953514.* Retrieved April 2020, from ResearchGate.

[11] Ottenstein, K. J. (1976.). An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 30–41.

[12] Halstead., M. H. (1973). An experimental determination of the "purity" of a trivial algorithm. *ACM SIGMETRICS Performance Evaluation Review*, 10–15.

[13] Grier, S. (1981). A tool that detects plagiarism in Pascal programs. *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education.* New York, NY, USA: Association for Computing Machinery.

[14] J. L. Donaldson, M. P. (1981). A plagiarism detection system. *ACM SIGCSE Bulletin*, 21-25.

[15] J. A. W. Faidhi and S. K. Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming environment. Comput. Educ. 11, 1 (Jan. 1987), 11–19.

[16] Whale., G. (1990). Software metrics and plagiarism detection. *Journal of Systems and Software*, 131–138.

[17] Johnson. (1994). Substring matching for clone detection and change tracking. *International Conference on Software Maintenance (ICSM)* (pp. 120-126), Victoria, BC, Canada: IEEE.

[18] L. Barbour, H. Y. (2010). A technique for just-in time clone detection. *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10)* (pp. 76–79). Washington DC, USA: IEEE.

[19] S. Ducasse, O. N. (2006). On the effectiveness of clone detection by string matching, *Journal on Software Maintenance and Evolution: Research and Practice*, 37-58.

[20] Cordy, C. K. (2008). NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. *16th IEEE International Conference on Program Comprehension,* (pp. 172-181). Amsterdam: IEEE.

[21] Y. Higo, S. K. (2011). Code clone detection on specialized PDG's with heuristics, *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, (pp. 75-84). Oldenburg, Germany.

[22] Kim, S. S. (2017). VUDDY: a scalable approach for vulnerable code clone discovery. *In Security and Privacy (SP), 2017 IEEE Symposium* (pp. 595-614). San Jose, CA, USA: IEEE.

[23] Shihab, D. E. (2013). Cccd: Concolic code clone detection. *2013 20th Working Conference on Reverse Engineering (WCRE)* (pp. 489-490). Koblenz, Germany: IEEE.

[24] Z. Liu, Q. W. (2017). A vulnerable code clone detection system based on vulnerability fingerprint. *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)* (pp. 548-553). Chongqing, China: IEEE.

[25] Baker, B. S. (2007). Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering, 33(9),* 608-621.

[26] Kamiya, T. K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transaction on Software Engineering,* 54–67.

[27] Koschke., N. G. (2009). Incremental clone detection. . *In Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (pp. 219–228). IEEE.

[28] Li, Z. a. (2004). CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code. *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (p. 20). San Francisco, CA: USENIX Association.

[29] Ragkhitwetsagul, C. K. (2019). Siamese: scalable and incremental code clone search via multiple code representations. *Empir Software Eng.*, 2236–2284.

[30] Nishi, M. A. (2018). Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software 137 (2018),* 130-142.

[31] Wang, P. J. (2018). CCAligner: a token based large-gap clone detector. *Proceedings of the 40th International Conference on Software Engineering* (pp. 1066-1077). Gothenburg Sweden: ACM.

[32] Sajnani, H. V. (2016). SourcererCC: scaling code clone detection to big-code. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference* (pp. 1157-1168). Austin Texas: IEEE/ACM.

[33] Y. Semura, N. Y. (2017). Ccfindersw:Clone detection tool with exible multilingual tokenization. *24th Asia-Pacific Software Engineering Conference* (pp. 654-659). Nanjing, Jiangsu, China: A PSEC.

[34] R. Koschke, R. F. (2006). Clone detection using abstract syntax suffix trees. *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, (pp. 253–262.). Benevento, Italy.

[35] L. Jiang, G. M. (2007). DECKARD: Scalable and accurate tree based detection of code clones. *Proceedings of 29th International Conference on Software Engineering (ICSE'07)*, (pp. 96-105). Minneapolis, MN, USA.

[36] I. D. Baxter, A. Y. (1998). Clone detection using abstract syntax trees. *Proceedings of the 14th International Conference on Software Maintenance (ICSM '98),* (pp. Bethesda, Maryland, USA, 1998, pp. 368–). Bethesda, Maryland, USA.

[37] L. Barbour, H. Y. (2010). A technique for just-in time clone detection. *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10),* (pp. 76–79.). Washington DC, USA.

[38] W.S. Evans, C. F. (2009). Clone detection via structural abstraction, *Software Quality Journal*, 309–330.

[39] A. Corazza, S. D. (2010). A tree kernel based approach for clone detection. *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)* (pp. 1-5). Timisoara, Romania: IEEE.

[40] D. Gitchell, N. T. (1999). Sim: a utility for detecting similarity in computer programs, *ACM SIGCSE Bulletin 31 (1),* 266–270.

[41] T.T. Nguyen, H. N.-K. (2009). ClemanX:Incremental clone detection tool for evolving software. *Proceedings of 31st International Conference on Software Engineering (ICSE'09),* (pp. 437–438). Vancouver,Canada.

[42] B. Biegel, S. D. (2010). Highly configurable and extensible code clone detection. *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10),* (pp. 237–241). Beverly, MA, USA.

[43] V. Wahler, D. S. (2004). Clone detection in source code by frequent itemset techniques, *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04),* (pp. 128–135.). Chicago, IL, USA: IEEE.

[44] Yang, Y. Z. (2018). Structural Function Based Code Clone Detection Using a New Hybrid Technique. *IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (pp. 286-291). Tokyo, Japan: IEEE.

[45] Lavoie, E. M. (2019). Computing structural types of clone syntactic blocks. *16th Working Conference on Reverse Engineering* (pp. 274-278). Lille: IEEE.

[46] J. Zeng, K. B. (2019). Fast code clone detection based on weighted recursive autoencoders. *IEEE Access, 7*, 125062-125078.

[47] Ming Wu, P. W. (2020). LVMapper:A Large-Variance Clone Detector Using Sequencing Alignment Approach. *IEEE access*.

[48] R. Komondoor, S. H. (2001). Using slicing to identify duplication in source code. *Proceedings of the 8th International Symposium on Static Analysis (SAS'01),* (pp. 40–56). Paris, France.

[49] Y. Higo, K. S. (2009). Problematic code clones identification using multiple detection results. *Proceedings of the 16th Asia Pacific Software Engineering Conference (APSEC'09),* (pp. 365–372.). Penang, Malaysia.

[50] Krinke, J. (2001). Identifying similar code with program dependence graphs. *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01),* (pp. 301–309). Stuttgart, Germany.

[51] S. Choi, H. P. (2009). A static API birthmark for windows binary executables. *The Journal of Systems and software*, 862–873.

[52] Elizabeth Burd, J. B. (2002). "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)* (pp. 36-43). Montreal, Canada: IEEE.

[53] G. Antoniol, U. V. (2002). Analyzing cloning evolution in the Linux kernel, *Information and Software Technology,* 755-765.

[54] M. Balazinska, E. M. (1999). Measuring clone based reengineering opportunities. *Proceedings of the 6th International Software Metrics Symposium (METRICS'99),* (pp. 292–303). Boca Raton,Florida, USA.

[55] Ragkhitwetsagul, C. J. (2018). A picture is worth a thousand words: Code clone detection based on image similarity. *Software Clones (IWSC), 2018 IEEE 12th International Workshop* (pp. 44-50). Campobasso, Italy: IEEE.

[56] Prechelt, M. a. (2000). *JPlag: Finding plagiarisms among a set of programs.* . University of Karlsruhe, Department of Informatics.

[57] Birov, T. C. (2015). Duplicate code detection algorithm. *16th International Conference on Computer Systems and Technologies, CompSysTech '15* (pp. 104-111). New York, NY: ACM.

[58] M. Iwamoto, S. O. (2013.). A token-based illicit copy detection method using complexity for a program exercise. *Eighth International Conference on Broadband and Wireless Computing, Communication and Applications.* (pp. 575-580.). NW Washington, DCUnited States.: IEEE.

[59] B. Muddu, A. A. (2013). Cpdp: A robust technique for plagiarism detection in source code. *7th International Workshop on Software Clones (IWSC)* (pp. 39-45). San Francisco, CA, USA: IEEE.

[60] W. Tang, D. C. (2018). BCFinder: A Lightweight and Platform-Independent Tool to Find Third-Party Components in Binaries. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 288-297). Nara, Japan: IEEE.

[61] K. Ito, T. I. (2017). Web-service for finding cloned files usingb-bit minwise hashing. *2017 IEEE 11th International Workshop on Software Clones* (pp. 1-2). Klagenfurt, Austria: IEEE.

[62] A. Cosma, A. S. (2012). A novel approach based on formal methods for clone detection. *2012 6th International Workshop on Software Clones (IWSC)* (pp. 8-14). Zurich: IEEE.

[63] Parr, T. (2014). *ANTLR*. Retrieved April 10, 2020, from https://www.antlr.org/: https://www.antlr.org/

[64] Parr, T. (2014). *https://github.com/antlr/grammars-v4*. Retrieved April 10, 2020, from https://github.com/: https://github.com/antlr/grammars-v4.

[65] Naumann, F. (2013). *SImilarity Measures.*

[66] Ragkhitwetsagul, C. a. (2017). Using compilation/decompilation to enhance clone detection. *11th International Workshop on Software Clone (IWSC'17)* (pp. 8-14). Klagenfurt, Austria: IEEE.

[67] Thome, J. (n.d.). *https://github.com/julianthome/inmemantlr*. Retrieved April 10, 2020, from https://github.com: https://github.com/julianthome/inmemantlr

[68] Christopher D Manning, P. R. (2008). *Introduction to information retrieval.* Cambridge.: volume 1. Cambridge university press Cambridge.

[69] Hao Zhong, L. Z. (2009). Inferring resource specifications from natural language api documentation. *In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society.

[70] Pati, J. B. (2017). A Comparison Among ARIMA, BP-NN, and MOGA-NN for Software Clone Evolution Prediction. *IEEE Access 5*, 11841-11851.

[71] Y. Yang, Z. R. (2018). Structural function based code clone detection using a new hybrid technique. *IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (pp. 286-291). Tokyo,Japan: IEEE.

[72] J. Zeng, K. B. (2019). Fast code clone detection based on weighted recursive autoencoders. *IEEE Access, 7*, 125062-125078.

[73] D. Li, M. P. (2014). One pass preprocessing for token-based source code clone detection. *IEEE 6th International Conference on Awareness Science and Technology (iCAST)* (pp. 1-6). Paris, France: IEEE.
[74] Thompson, H. L. (2011). Incremental clone detection and elimination for erlang programs. *Fundamental Approaches to Software Engineering,Springer* , 356-370.
[75] Kamiya, T. (2013). Agec: An execution-semantic clone detection tool. *21st International Conference on Program Comprehension (ICPC)* (pp. 227-229). San Francisco, CA, USA: IEEE.
[76] Mohamed El Bachir Menai,"Detection of Plagiarism in Arabic Documents", *International Journal of Information Technology and Computer Science*, vol.4, no.10, pp.80-89, 2012.
[77] Karzan Wakil, Muhammad Ghafoor, Mehyeddin Abdulrahman, Shvan Tariq, "Plagiarism Detection System for the Kurdish Language", *International Journal of Information Technology and Computer Science*, Vol.9, No.12, pp.64-71, 2017.

Authors' Profiles

Sanjay Ankali is a research scholar at VTU-RRC, Belagavi-590018 and working as Assistant Professor in the Department of CSE at KLECET, Chikodi, India-591201. His research interest is in the field of Software Engineering, Software clone detection and code plagiarism detection.

Dr. Latha Parthiban is working as Assistant Professor in department of Computer Science at, Pondicherry University- Community College, India-605008. She has received Bachelors of Engineering in Electronics from Madras University in the year 1994. M. E from Anna University in the year 2008 and Ph D from Pondicherry University in the year 2010. Her research interest includes Software Engineering, Big Data Analytics, and Computer Networking.

How to cite this paper: Sanjay B. Ankali, Latha Parthiban, " A Methodology for Reliable Code Plagiarism Detection Using Complete and Language Agnostic Code Clone Classification", International Journal of Modern Education and Computer Science(IJMECS), Vol.13, No.3, pp. 34-56, 2021.DOI: 10.5815/ijmecs.2021.03.04