International Journal of Computer Applications Technology and Research
Volume 4– Issue 10, 728 - 736, 2015, ISSN: 2319–8656

Duplicate Code Detection using Control Statements

Sudhamani M
Department of Computer Science
University of Mysore
Mysore 570006, India

Lalitha Rangarajan
Department of Computer Science
University of Mysore
Mysore 570006, India

Abstract: Code clone detection is an important area of research as reusability is a key factor in software evolution. Duplicate code degrades the design and structure of software and software qualities like readability, changeability, maintainability. Code clone increases the maintenance cost as incorrect changes in copied code may lead to more errors. In this paper we address structural code similarity detection and propose new methods to detect structural clones using structure of control statements. By structure we mean order of control statements used in the source code. We have considered two orders of control structures: (i) Sequence of control statements as it appears (ii) Execution flow of control statements.

Keywords: Control statements; Control structure; Execution flow; Similarity value; Structural similarity.

1. INTRODUCTION

Duplicate codes are identical or similar code fragments present in software program. Two code fragments are similar if these code segments are similar in their structure of control statements and similar control flow between control lines [1, 15].

Different types of code clones are [15]

Type 1: Exact similar code fragments except white space and comments as shown in below example.

Ex 1:

Segment 1:

```
if(n>0)
{
 n=n*1;   //multiply by plus 1
}
else
n=n*-1;  // multiply by minus 1
```

Segment 2:

```
if ( n > 0 )
{
 n = n * 1;   //multiply by +1
}
else
n = n * -1;  // multiply by -1
```

Type 2: Syntactic similar code fragments except change in variable, literal and function names.

Ex 2:

Segment 1:

```
if (n>0)
{
 n=n*1;   //multiply by plus 1
}
else
n=n*-1;  // multiply by minus 1
```

Segment 2:

```
if ( m > 0 )
{
 m = m * 1;   //multiply by +1
}
else
m = m * -1;  // multiply by -1
```

Type 3: Similar code fragments with slight modifications like reordering/addition/deletion of some statements from already existing or copied code fragments.

Segment 1:

```
if (n > 0)
{
 n=n*1;   //multiply by plus 1
}
else
n=n*-1;   // multiply by minus 1
```

Segment 2:

```
if (n > 0)
{
 n=n*1;   //multiply by plus 1
}
else
n=n*-1;   // multiply by minus 1
x=5;    //newly added statement
```

In the above example a new statement x=5 is added.

Type 4: Functionally similar code fragments. Below example explains recursive and non recursive way of finding factorial of n. (same program implemented in two ways).

Ex:

Segment 1:

```
int i, j=1, n;
for (i=1; i<=n; i++)
j=j*i;
```

segment 2:

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

Output of program depends on the execution flow of effective source lines. Execution flow of source lines depends on the control lines used in the program. Control lines considered here are iterative statements (for, while and do-while), conditional statements (if, if-else and switch-case), and

```
Sorting program 1
main ()
{
int a[30],n,  i, j,  temp;
printf ("\n Enter the size of an array");
scanf ("%d",&n);
printf ("\n Enter the elements to
sort");
For (i=1; i<=n; i++)
scanf ("%d",&a[i]);

printf ("numbers to sort are");

for(i=1; i<=n; i++)
{
printf ("%d ",a[i]);
}

For (i=1; i <=n; i++)
{
for (j=1; j<=n; j++)
{
if (a[j] > a[j+1])
{
temp=a[j];
a[j]= a[J+1];
a[j+1] = temp;
}
}
}
printf ("Sorted numbers");
for(i=1; i<=n; i++)
printf ("%d ",a[i]);
}
```

```
Sorting program 2
Sort()
{
for (i=1; i <=n; i++)
for (j=1; j<=n; j++)
if  (a[j] >a[j+1])
{
temp = a[j];
a[j] = a[J+1];
a[j+1] = temp;
}
}
Print()
{
for(  i=1;  i<=n; i++)
printf  ("%d ",a[i]);
}
main () {
int a[30],n, i, j, temp;
Printf ("\n Enter the size of an array");
scanf ("%d",&n);
printf ("\n Enter the elements to
sort");
i=1;
while (i <= n)
{
Scanf ("%d",&a[i]);
i++;
}
Printf ("numbers to sort are");
Print();
Sort();
printf ("Sorted numbers");
Print();
}
```

```
Sorting program 3
Sort()
{
for (j=1; j<=n; j++)
if (a[j] >a[j+1])
{
temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
}
}

Print()
{
for( i=1; i<=n; i++)
Printf ("%d ",a[i]);
}

main ()
{
int a[30],n, i, j, temp;
Printf ("\n Enter the size of an
array");
Scanf  ("%d",&n);
Printf ("\n Enter the elements to
sort");
for (i=1; i<=n; i++)
Scanf ("%d",&a[i]);
printf ("numbers to sort are");
Print();
for (i=1; i <+n; i++)
Sort();
printf("Sorted numbers");
Print();
}
```

Fig 1: Different versions of bubble sort program

function call. Here we propose two approaches to find structural similarity. Approach 1 considers order of control statements present in the code segments and approach 2 depends on the execution flow of control lines in the program. Figure 1 shows three different ways of writing bubble sort program. To find similarity of these programs we compute control structure metrics. Rest of the paper is organized as follows. Section 2 covers key literature, section 3 describes proposed methods and results; section 4 concludes the work with suggestions on possible future work.

2. RELATED WORK

Duplicate code detection mainly consists of two phases where first phase is transformation and second phase is comparison. In transformation phase, source code is transformed in to an Internal Code Format (ICF). Depending on the ICF comparison, match detection techniques are classified as follows [15].

i. **String Based:** In these techniques source code is considered as an arrangement of characters/strings/lines and uses string matching techniques to detect duplicate code [2]. Dup tool compares lexemes on behalf of string match and finds partial match [2, 3, 4]. Ducass et al [5] proposed dynamic matching technique to detect code clones. String based techniques are simple, language independent and detect type I clones [13, 14, 15, 16].

ii. **Token Based:** In token based approach source code is transformed into sequence of tokens using lexer/parser. Then these sequences of tokens are compared to find duplicate code. This technique detects both type I and II clones. Kamiya et al's [5] CC Finder regenerate source file into a set of tokens and device single token from these set of tokens and uses suffix tree substring matching algorithm to detect code clones. CP Miner uses frequent substring matching algorithm to replicate tokenized statement. SIM correlate the chain of tokens using dynamic programming string alignment technique. Winnowing and JPlag are token based plagiarism detection tools [13, 14, 15, 16].

iii. **Tree Based:** Source text is parsed to obtain Abstract Syntax Tree (AST) or parse tree with appropriate parser. Then tree matching techniques are used to find similar sub trees. This approach efficiently detects type I, type II and type III clones [5, 6]. As AST does not address data flow between controls, it fails to detect type IV clones. Baxter et al's CloneDR find resemblance between programs by matching sub trees of corresponding source program [15].

iv. **Graph Based:** Source program is converted into Program Dependency Graph (PDG) where PDG contains the data flow and control flow information of the program [6]. Then isomorphic sub graph detection algorithms are used to find duplicate code. This technique efficiently identifies all types of clones. However generating PDG and finding isomorphic sub graphs is NP hard [8]. Komondoor and Horowitz PDG-DUP uses program slicing to find isomorphic sub graphs, Krinke uses iterative approach to detect highest comparable sub graphs. GPLAG is graph based plagiarism disclosure tool [11, 16].

v. **Metric Based:** In this technique different metrics are computed for code fragments and these metric values are compared to find duplicate code [9, 10, 11, 12]. AST/PDG representation can be used to calculate metrics like number of nodes, number of control edges present in the graph etc. Other common metrics are number of source lines, number of function calls, number of local and global variables and McCabe's cyclomatic complexity etc. eMetric, Covert and

Moss are metric based tools [15, 16]. Kontogiannis et al. [16] build an abstract pattern matching tool to identify probable matches using Markov models to measures similarity between two programs.

3. PROPOSED METHOD

Here we propose two approaches to find duplicate code. The different stages in the proposed method are preprocessing, metric computation, difference matrix computation and similarity value calculation. Architecture of proposed method is shown in figure 2 and each stage is explained subsequently.

Preprocessing and template conversion

In preprocessing stage extra space and comments are removed and input source program is transformed into its standard intermediate template form. Figure 3 shows the template form of versions of sort program in figure 1. This template is used to compute control structure metrics.

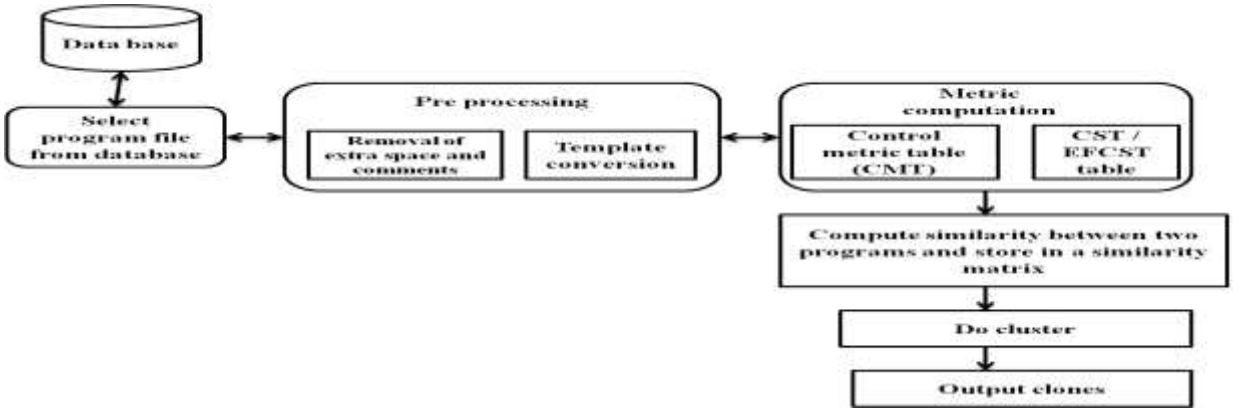

Fig 2: Architecture of proposed method

Fig 3: Templates of sort programs in figure 1

Note that the order / structure of control statements are different across versions. Some versions have function calls and some don't. Yet proposed approaches can detect duplicate to high accuracy.

3.1 Approach 1 – Computation of similarity using Control Structure Tables (CSTs)

Control Structure Table (CST): Control Structure Table contains the information about order of ingrained control lines used in the program [11]. CST of sort program 1 and sort program 2 in figure 1 are shown in table 1 and 2.

Table 1. Control structure table for sort program 1

Sl.No	Type of control statement	Loop	Condition
1	Loop	0	0
2	Loop	0	0
3	Loop	1	1
4	Loop	0	1
5	Condition	0	0
6	Loop	0	0

Table 2. Control structure table for sort program 2

Sl.No	Type of control statement	Loop	Conditio n
1	Loop	1	1
2	Loop	0	1
3	Condition	0	0
4	Loop	0	0
5	Loop	0	0

Difference Matrix (D) computation: Difference matrix is calculated using two CSTs. Difference matrix calculated from table 1 and 2 are shown in table 3. Difference matrix shows different between all pairs of control statement. Difference matrix (D) is computed from the respective control structure tables. A row of program 1 (corresponding to a control statement) is compared with every row of program 2. Row I and j of the programs are compared using city block distance formula |Ri1-Rj1|+|Ri2-Rj2|.

For example first row of table 1 is compared with second row of table 2 by computing |0-0| + |0-1| =1 is entered in (1, 2) of distance matrix (table 3). From this table we can find similar control lines present in two programs. Presence of zero in a position corresponding to similar control statement indicates structural similarity of the control statements in the two programs. For example zero at (3, 1) in table 3 imply that the iterative statements 3 of program 1 and 1 of program 2 are probably similar. Whereas zero at (5, 3) is not comparable because the control statements of the programs are different (fifth control statement of program 1 is conditional and third control statement of program 2 is iterative). The zeros that contribute to similarity are highlighted.

Table 3. Distance matrix computed from table 1 and 2

Control lines	Loop (L)	Loop (L)	Loop (L)	Cond (C)	Loop (L)
Loop(L)	2	1	**0**	0	**0**
Loop (L)	2	1	**0**	0	**0**
Loop(L)	**0**	1	2	2	2
Loop(L)	1	**0**	1	1	1
Cond(C)	2	1	0	**0**	0
Loop(L)	2	1	**0**	0	**0**

Similarity between codes is found, using the formula

$$\begin{cases} s = n & \text{if } r1 = r2 \\ s = \frac{n}{|r1 - r2|} & \text{otherwise} \end{cases} \quad \ldots\ldots (1)$$

where r1 and r2 are the number of control lines in two programs. From table 3 s = 9/1.

We conducted experiments using data set 1 of 5 distinct programs and 15 variants and similarity values are shown in table 4.

Table 4. Similarity table for data set 1 (s=n/|r1-r2|)

Programs	P1v1	P1v2	P1v3	P1v4	P2v1	P2v2	P2v 3	P3v1	P3v2	P3v3	P4v1	P4v2	P4v3	P5v1	P5v2
P1v 1	0.00	37.00	37.00	37.00	2.29	2.29	2.29	2.23	2.23	2.23	1.11	1.11	1.11	4.22	4.20
P1v2	37.00	0.00	37.00	37.00	2.29	2.29	2.29	2.23	2.23	2.23	1.11	1.11	1.11	4.22	4.20
P1v3	37.00	37.00	0.00	37.00	2.29	2.29	2.29	2.23	2.23	2.23	1.11	1.11	1.11	4.22	4.20
P1v4	37.00	37.00	37.00	0.00	2.29	2.29	2.29	2.23	2.23	2.23	1.11	1.11	1.11	4.22	4.20
P2v1	2.29	2.29	2.29	2.29	0.00	8.00	8.00	0.83	0.83	0.83	2.50	2.50	2.50	1.13	1.18
P2 2	2.29	2.29	2.29	2.29	8.00	0.00	8.00	0.83	0.83	0.83	2.50	2.50	2.50	1.13	1.18
P2v3	2.29	2.29	2.29	2.29	8.00	8.00	0.00	0.83	0.83	0.83	2.50	2.50	2.50	1.13	1.18
P3v1	2.23	2.23	2.23	2.23	0.83	0.83	0.83	0.00	199.00	199.00	0.61	0.61	0.61	10.92	13.83
P3v2	2.23	2.23	2.23	2.23	0.83	0.83	0.83	199.00	0.00	199.00	0.61	0.61	0.61	10.92	13.83
P3v3	2.23	2.23	2.23	2.23	0.83	0.83	0.83	199.00	199.00	0.00	0.61	0.61	0.61	10.92	13.83
P4v1	1.11	1.11	1.11	1.11	2.50	2.50	2.50	0.61	0.61	0.61	0.00	4.00	4.00	0.83	0.89
P4v2	1.11	1.11	1.11	1.11	2.50	2.50	2.50	0.61	0.61	0.61	4.00	0.00	4.00	0.83	0.89
P4v3	1.11	1.11	1.11	1.11	2.50	2.50	2.50	0.61	0.61	0.61	4.00	4.00	0.00	0.83	0.89
P5v1	4.22	4.22	4.22	4.22	1.13	1.13	1.13	10.92	10.92	10.92	0.83	0.83	0.83	0.00	161.00
P5v2	4.20	4.20	4.20	4.20	1.18	1.18	1.18	13.83	13.83	13.83	0.89	0.89	0.89	161.00	0.00

We may observe that in table 4 all programs show highest similarity only with its variants.

3.2 Approach 2: Computation of similarity using execution flow of control statements

In pre processing stage all functions are placed above the main function. Function Information
Table (FIT) and CST are generated in a single scan of the program.

Function Information Table (FIT): FIT gives starting and ending positions where a particular function begins and ends in CST. Here function calls are considered as a control lines. FIT of sort program 2 and 3 are shown in table 5a and 5b. CSTs of these programs are shown in table 6a and 6b.

Table 5a. Function Information Table (FIT) for sort program 2

Sl. No	Function name	Start position	End position
1	Sort	1	3
2	Print	4	4
3	main	5	8

Table 5b. Function Information Table (FIT) for sort program 3

Sl. No	Function name	Start position	End position
1	Sort	1	2
2	Print	3	3
3	Main	4	8

The line 1 (first control statement) of program 2 is function name ‘sort’ (beginning of function) is entered in FIT of table 5a (refer function name and start position). The control statements scanned from line 1 onwards are recorded sequentially in CST (table 6a) until end of the function. The end of the function namely line 3 is recorded in FIT. Thus in one scan FIT and CST are generated.

Execution Flow Control Structure Table (EFCST) is computed using CST and FIT by replacing the function calls by control lines of that particular function.

Table 6a. Control Structure Table (Order) for program 2 in figure 1

Sl. no	Control statement	Loop	Condition
1	Loop	1	1
2	Loop	0	1
3	Condition	0	0
4	Loop	0	0
5	Loop	0	0
6	Print	0	0
7	Sort	0	0
8	Print	0	0

Table 6b. Control Structure Table (Order) for program 3 in figure 1

Sl. no	Control statement	Loop	Condition
1	Loop	0	1
2	Condition	0	0
3	Loop	0	0
4	Loop	0	0
5	Print	0	0
6	Loop	0	0
7	Sort	0	0
8	Print	0	0

Execution Flow Control Structure Table (EFCST) of program 2 is given in table 7. Execution flow starts in 'main'. From FIT we see that flow starts at line 5 and ends at line 8. The entries in these lines are copied in EFCST. However if function call is present, FIT is referred as corresponding control lines of the function from the respective beginning and ending lines are copied to EFCST. The EFCST of programs 1, 2 and 3 in figure 1 are shown in table 7.

Table 7. EFCST of program 1, 2 and 3

Sl. no	Control statement	Loop	Condition
1	Loop	0	0
2	Loop	0	0
3	Loop	1	1
4	Loop	0	1
5	Condition	0	0
6	Loop	0	0

Difference matrix is computed using two EFCSTs as in section 3.1 and similarity value is computed using formula 1.

We conducted experiments on data set 1 and results are shown in below table. We conducted experiments on data set 1 and results are shown in table 8.

Table 8. EFCST and s=n/|r1-r2|

	P1v1	P1v2	P1v3	P1v4	P2v1	P2v2	P2v3	P3v1	P3v2	P3v3	P4v1	P4v2	P4v3	P5v1	P5v2
P1v1	0.00	36.00	36.00	36.00	2.14	2.14	2.14	2.14	2.14	2.14	1.00	1.00	1.00	3.55	3.30
P1v2	36.00	0.00	36.00	36.00	2.14	2.14	2.14	2.14	2.14	2.14	1.00	1.00	1.00	3.55	3.30
P1v3	36.00	36.00	0.00	36.00	2.14	2.14	2.14	2.14	2.14	2.14	1.00	1.00	1.00	3.55	3.30
P1v4	36.00	36.00	36.00	0.00	2.14	2.14	2.14	2.14	2.14	2.14	1.00	1.00	1.00	3.55	3.30
P2v1	2.14	2.14	2.14	2.14	0.00	7.00	7.00	0.76	0.76	0.76	2.00	2.00	2.00	1.00	0.88
P2v2	2.14	2.14	2.14	2.14	7.00	0.00	7.00	0.76	0.76	0.76	2.00	2.00	2.00	1.00	0.88
P2v3	2.14	2.14	2.14	2.14	7.00	7.00	0.00	0.76	0.76	0.76	2.00	2.00	2.00	1.00	0.88
P3v1	2.14	2.14	2.14	2.14	0.76	0.76	0.76	0.00	196.00	196.00	0.58	0.58	0.58	13.91	9.83
P3v2	2.14	2.14	2.14	2.14	0.76	0.76	0.76	196.00	0.00	196.00	0.58	0.58	0.58	13.91	9.83
P3v3	2.14	2.14	2.14	2.14	0.76	0.76	0.76	196.00	196.00	0.00	0.58	0.58	0.58	13.91	9.83
P4v1	1.00	1.00	1.00	1.00	2.00	2.00	2.00	0.58	0.58	0.58	0.00	3.00	3.00	0.75	0.63
P4v2	1.00	1.00	1.00	1.00	2.00	2.00	2.00	0.58	0.58	0.58	3.00	0.00	3.00	0.75	0.63
P4v3	1.00	1.00	1.00	1.00	2.00	2.00	2.00	0.58	0.58	0.58	3.00	3.00	0.00	0.75	0.63

P5v1	3.55	3.55	3.55	3.55	1.00	1.00	1.00	13.91	13.91	13.91	0.75	0.75	0.75	0.00	125.00
P5v2	3.30	3.30	3.30	3.30	0.88	0.88	0.88	9.83	9.83	9.83	0.63	0.63	0.63	125.00	0.00

Here also all programs show high similarity only with versions of the same program.

3.3 Similarity computation using CSTs, EFCSTs and Control Metric Table (CMT)

Control Metric Table (CMT): We compute control metric table which contains information about total number of iterative and conditional statements present in the program [11]. Table 9 shows CMT of data set 1 used for our experiment.

Table 9. Control Metric Table for data set 1 (CMT)

Sl. No	Programs	1		2		3		4	
		L	C	L	C	L	C	L	C
1	Beam search	10	2	10	2	10	2	10	2
2	Bubble sort	4	1	4	1	4	1	-	-
3	Min Max	15	19	15	19	15	19	-	-
4	Linear search	2	1	2	1	-	-	-	-
5	Queue	3	18	3	18	3	18	-	-

Computation of similarity value (s): Here similarity computation is based on CMT as well as CST/EFCST. First we generate CMT and CST for each program. Difference matrix (D) is computed from the respective CSTs as explained in earlier sub sections 3.1 and 3.2.

We compute similarity between programs only if programs are comparable in terms of number of loops and conditional statements. While duplicates are created it is unlikely to expect more than 20 % variation in number of control statements. Hence a threshold of 20 % variations in these numbers is fixed for computation of similarity. Suppose program 1 has x loops and y conditional statements. Program 2 is comparable with program 1 if the number loops and conditional statements are in the range [x – 20 % (x), x + 20 % (x)] and [y – 20 % (y), y + 20 % (y)]. Table 10 show computed similarity values with this additional consideration of CMT.

Table 10a. CST, CMT and s=n/|r1-r2|

	P1v1	P1v2	P1v3	P1v4	P2v1	P2v2	P2v3	P3v1	P3v2	P3v3	P4v1	P4v2	P4v3	P5v1	P5v2
P1v1	0	37	37	37	0	0	0	0	0	0	0	0	0	0	0
P1v2	37	0	37	37	0	0	0	0	0	0	0	0	0	0	0
P1v3	37	37	0	37	0	0	0	0	0	0	0	0	0	0	0
P1v4	37	37	37	0	0	0	0	0	0	0	0	0	0	0	0
P2v1	0	0	0	0	0	8	8	0	0	0	0	0	0	0	0
P2v2	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0
P2v3	0	0	0	0	8	8	0	0	0	0	0	0	0	0	0
P3v1	0	0	0	0	0	0	0	0	199	199	0	0	0	0	0
P3v2	0	0	0	0	0	0	0	199	0	199	0	0	0	0	0
P3v3	0	0	0	0	0	0	0	199	199	0	0	0	0	0	0
P4v1	0	0	0	0	0	0	0	0	0	0	0	4	4	0	0
P4v2	0	0	0	0	0	0	0	0	0	0	4	0	4	0	0
P4v3	0	0	0	0	0	0	0	0	0	0	4	4	0	0	0
P5v1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	161
P5v2	0	0	0	0	0	0	0	0	0	0	0	0	0	161	0

Table 10b. EFCST, CMT and s= n/|r1-r2|

	P1v1	P1v2	P1v3	P1v4	P2v1	P2v2	P2v3	P3v1	P3v2	P3v3	P4v1	P4v2	P4v3	P5v1	P5v2
P1v1	0	36	36	36	0	0	0	0	0	0	0	0	0	0	0
P1v2	36	0	36	36	0	0	0	0	0	0	0	0	0	0	0
P1v3	36	36	0	36	0	0	0	0	0	0	0	0	0	0	0
P1v4	36	36	36	0	0	0	0	0	0	0	0	0	0	0	0

P2v1	0	0	0	0	0	7	7	0	0	0	0	0	0	0	0
P2v2	0	0	0	0	7	0	7	0	0	0	0	0	0	0	0
P2v3	0	0	0	0	7	7	0	0	0	0	0	0	0	0	0
P3v1	0	0	0	0	0	0	0	0	196	196	0	0	0	0	0
P3v2	0	0	0	0	0	0	0	196	0	196	0	0	0	0	0
P3v3	0	0	0	0	0	0	0	196	196	0	0	0	0	0	0
P4v1	0	0	0	0	0	0	0	0	0	0	0	3	3	0	0
P4v2	0	0	0	0	0	0	0	0	0	0	3	0	3	0	0
P4v3	0	0	0	0	0	0	0	0	0	0	3	3	0	0	0
P5v1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	125
P5v2	0	0	0	0	0	0	0	0	0	0	0	0	0	125	0

In the above tables similarity is seen only with versions of the same program. All others are 0's.

3.4 Experimental Results

Five programs, 15 versions data set described in earlier sections is created in our lab and the experimental results with two approaches have been discussed in detail in sections 3.1 to 3.3.

For thorough testing of the proposed approaches we downloaded programs from 'sourcefoge.net' (www.sourceforge.net) and 'f1sourcecode' (www.f1sourcecode.com) and created many versions by changing loop statements, reordering control lines and also by refactoring. These are added to the sample data set in the earlier sections. Thus we have created 26 distinct programs and 100 versions data set. To find whether only versions of the same programs, show higher similarity when compared to similarities with other programs, we have done clustering of similarity values using k-means clustering algorithm with k=2. The clustering is done on set of similarity value corresponding to one version of a program (available in a column). The error in duplicate detection of a program 'j' is found as ratio of number of misclassification and total number of programs (inclusive of versions). Total misclassification in program 'j' includes number of false positives and true negatives. When a version of program 'j' is clustered with any other program it is true negative, where as when a version of program 'i' is clustered with program 'j' it is false positive.

Average error is computed total detection errors in each program by number of distinct programs. Table 11 shows the average error with two approaches with and without CMT for the sample data sets. Also shown in the table the similarity measurements using the formula s=n/D, where n is similar number of control lines and 'D' maximum dissimilarity [11].

Table 11. Error table for sample data sets.

Approaches	**Data structure used**	**Data set1**		**Data set 2**	
		S= n /D	**S= n / |r1-r2|**	**S= n / D**	**S=n / |r1-r2|**
Approach 1	Only CST	0.1465	0.0375	0.5794	0.1038
	CST and CMT	0	0	0.00923	0.00577
Approach 2	Only EFCSTs	0.04	0.0375	0.0866	0.009615
	EFCST and CMT	0	0	0.009615	0.00808

3.5 Time Complexity

Suppose two programs have n_1 and n_2 source lines and L_1 and L_2 control statements. Note that number of control statements in a program will be far less than number of source lines ($L << n$). Table 12 shows the detail of major steps in the computation of similarity and the corresponding complexities.

Table 12. Time complexity table

Steps	Complexity
Preprocessing	$\theta(n_1) + \theta(n_2)$
CST / EFCST	$\theta(n_1) + \theta(n_2)$
Difference matrix	$\theta (L_1 \times L_2)$
Similarity computation	$O(L_1 \times L_2)$

Hence total time complexity is maximum ($\theta(n)$ and $O(L^2)$) which is a polynomial time complexity.

3.6 Performance Evaluation

The experiments are done with three available tools **Duplo** (uses string matching technique), **PMD** (uses tokens to compare) and **CloneDR** (AST based) and the results obtained on data set 1 is shown in table 13.

PMD tool shows similarity with user defined function call and inbuilt function. Control lines for and while, from figure 1 are not shown as similar. CloneDR is sometimes sensitive to change in the type of loop statement.

We divided data set 2 which is used in section 3.4 into two data sets. First data set has 15 distinct programs and 50 variants. This data set has variation in sequence of control statements (independent control lines only) in versions of the same program. Second data set has 11 distinct programs and 50 variants. In this data set contents of control lines are replaced by function calls (refer fig 1).

Experiments are conducted on two data sets using two approaches. Tables 14a and 14b show performance analysis for proposed methods.

Table 13. Performance analysis table

Sl. no	Method	Error		Remarks
1	Duplo	1.8666		All versions of beam search show some similarity with all versions of minmax and bubble sort programs are not shown as similar programs.
2	PMD	1.6		All versions of beam search show some similarity with all versions of minmax and queue programs are not shown as similar programs.
3	Clone DR	1.8666		All versions of beam search show some similarity with all versions of minmax and queue programs are not shown as similar programs.
4	Proposed Approaches	Only CST	0.14658	Linear search and beam search programs show similarity with versions of other programs
		Only EFCST	0.04	Linear search program shows similarity with bubble sort programs
		CST & CMT	0	Similarity exists with its versions only
		EFCST & CMT	0	

Table 14a. Performance analysis table (without considering CMT)Without CMT			
Data structure and similarity measure used	**Data set1**	**Data set 2**	**Data set 3**
CST & s=n/d	0.14658	0.34	0.292727
CST & s=n/|r1-r2|	0.0375	0.0866	0.092727
EFCST & s=n/|r1-r2|	0.0375	0.0373	0.049

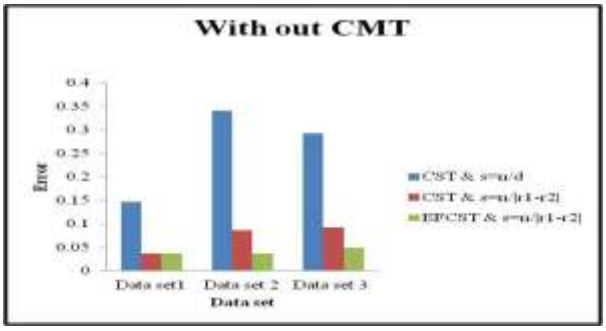

Fig 4: Error graph for proposed approaches without considering CMT

Table 14b. Performance analysis table (considering CMT)

With CMT			
Data structure and similarity measure used	**Data set1**	**Data set 2**	**Data set 3**
CST & s=n/d	0	0.0133	0.0436
CST & s=n/|r1-r2|	0	0.00933	0.02
EFCST & s=n/|r1-r2|	0	0	0

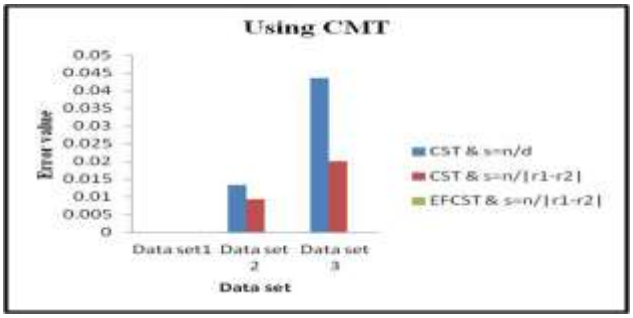

Fig 5: Error graph for proposed approaches without considering CMT

4. CONCLUSION AND FUTURE WORK

We have proposed two approaches Control Structure Table (CST) and Execution Flow Control Structure Table (EFCST) to detect duplicate code detection. We also suggested Control Metric Table (CMT) before computation of similarity measure. Performance with the addition of CMT has shown tremendous improvements.

The time complexity is max (θ(n) and $O(L^2)$) where 'n' is total number of source lines and 'L' is total number of control statements in the program. Time complexity is far less when compared to methods based on AST and PDG. The method also identifies all four types of clones.

The proposed algorithms do not take into consideration of statements inside control structures. The current similarity measure can be corrected to consider the statements together with operators and operands. Perhaps errors that are observed currently may decrease significantly.

5. REFERENCES

[1] Baker S., "A Program for Identifying Duplicated Code, "Computing Science and Statistics, vol. 24, 1992.

[2] Johnson J H., "Substring matching for clone detection and change tracking," in Proceedings of the International Conference on software Maintenance, 1994.

[3] Ducasse, S., Rieger M., and Demeyer S., "A Language Independent Approach for Detecting Duplicated Code." In Proceedings; IEEE International Conference on Software Maintenanace, 1999.

[4] Zhang Q., et . al., "Efficient Partial-Duplicate Based on Sequence Matching," 2010.

[5] Sadowski C., and Levin G., "SimHash: Hash-Based Similarity Detection," 2007.

[6]Jiang L., and Glondu S., "Deckard: Scalable and Accurate Tree-Based Detection of Code Clones".

[7] Baxter I D., Yahin I., Moura L., Anna M S., and Bier L., "Clone Detection Using Abstract Syntax Trees," in proceedings of ICSM. IEEE, 1998.

[8] krinke J., "Identifying Similar Code with Program Dependency Graphs," Proc. Eighth Working Conference ., Reverse Engineering., 2001.

[9] Vidya K and Thirukumar K, "Identifying Functional Clones between Java Directory using Metric Based Systems" International journal of Computer Communication and Information System (IJCCI)-Vol3, ISSN:2277-128x August 2013.

[10] Mayrand J, Leblanc C and Ettore Merlo M. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", proc of ICSM conference 1996.

[11]Sudhamani M and Rangarajan L, Structural Similarity Detection using Structure of Control Statements, proc. of International Conference on Information and Communication Technology, vol 46 (2015) 892-899.

[12] kodhai E, Perumal A, and Kanmani S, "Clone Detection using Textual and metric Analysis to figure out all Types of Clones" International journal of Computer Communication and Information System (IJCCI)- Vol2. No1.ISSN:0976-1349 July-Dec 2010.

[13] www.research.cs.queensu.ca.

[14] Bellon S., Koschke R., Antoniol G., Krinke J., and Merlo E., "Comparison and evaluation of clone detection tools," IEEE Transactions on.Software Engineering, September 2007.

[15]. Roy C K and Cordy J R. "A survey on software clone detection research". Tech. rep., 2007. TR 2007-541 School of Computing Queen's University at Kingston Ontario, Canada.

[16] Roy C K, Cordy J R and Koschke R, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", Science of Computr Programming, 74(2009) 470-495, 2009.

[17] Kostas Kontogiannis. "Evaluation Experiments on the Detection of Programming Patterns using Software Metrics". In Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'97), pp. 44-54, Amsterdam, the Netherlands, October 1997.