Analysis of a Hybrid Clone Detection Technique

Dr.G.Anil Kumar
Sr. Assistant Professor CSE MGIT Hyderabad T.S. India

Abstract:
Any software code clone detection technique should prove itself efficient in terms of some quality parameters. In this paper we discuss about two quality parameters, that is precision and recall. This method proved efficient in terms of these two parameters over the suffix tree method.

The proposed software clone detection system has been implemented in the working platform of JAVA (version JDK 1.6). Here we use the source code with different sizes of Software Lines of Code (SLOC). The main goal of the proposed method is to identify all four types of clones in the source code. This can be achieved by combining two methods called textual analysis and metrics method. In the proposed method metrics analysis is done through the twelve metrics. These metrics are used to identify the potential clones. Then, textual approaches are applied. These textual approaches include line by line comparison using string matching algorithm. Tokenization approach is also used to identify the similarity between language constructs which are divided as tokens. The step by step results obtained from the proposed method is described in following section. We first explained the functionality of the tool which we developed for the proposed method and then a case study which explains how this tool figure out clones from a set of code fragments.

System functionality
The working model of the system explained in this section. The system takes input files and processes them and gives results. This process is explained with captured screen shots.

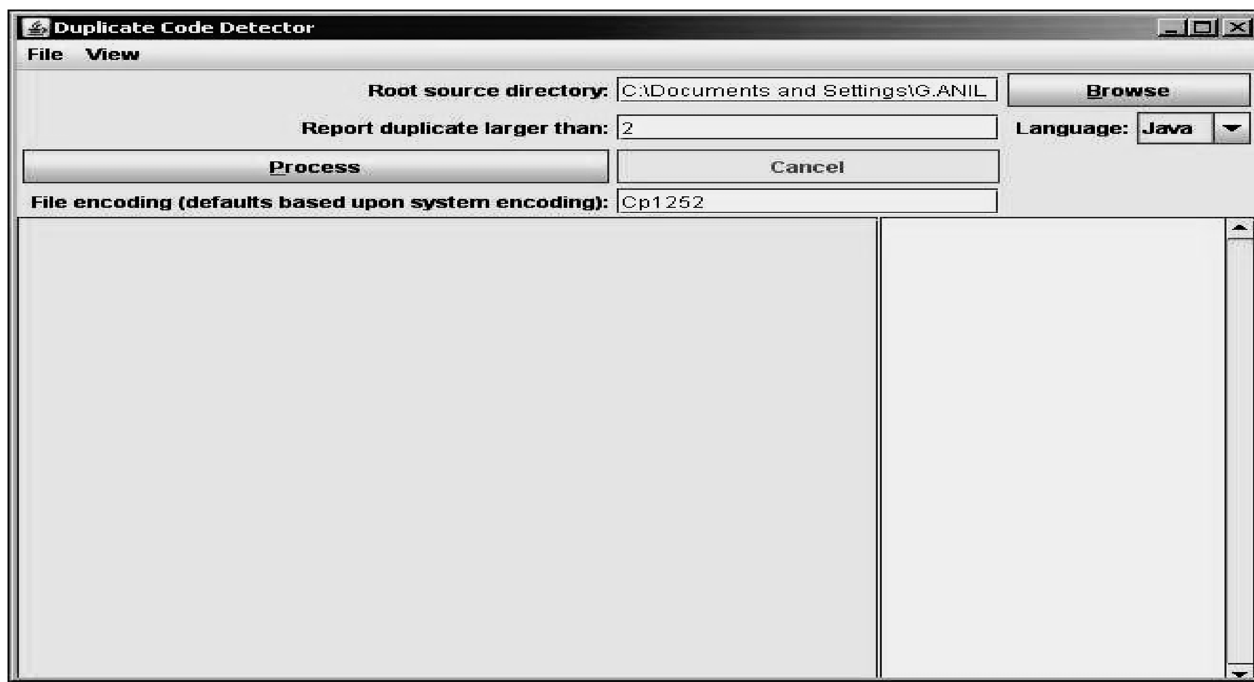


Figure 4.1: Initial Process

Figure 4.1 represents the initial screen obtained in the clone detection process to Load the database that is set of source programs using *browse* button. The Root *source directory* field will be filled if a particular folder or file is browsed from the source. Here there are two options to select a file. Selecting a file compares the file with the content of the same file which is shown in Figure 4.2 and selecting a folder compares the files contained in the folder (i.e. more than one file) which is shown in Figure 4.3. The

clone detection procedure is same for both the selections, only difference is that if a folder is selected, the tool will concatenate all the files in the folder. After concatenation normalization and the other procedure follows as a single file. Then selecting the *Report duplicate larger than* field to find the code fragments that are repeated specified number of times. *Language* field by default set to JAVA because this research is restricted to compare JAVA language constructs. File encoding for token based approaches the default system based encoding was adapted.

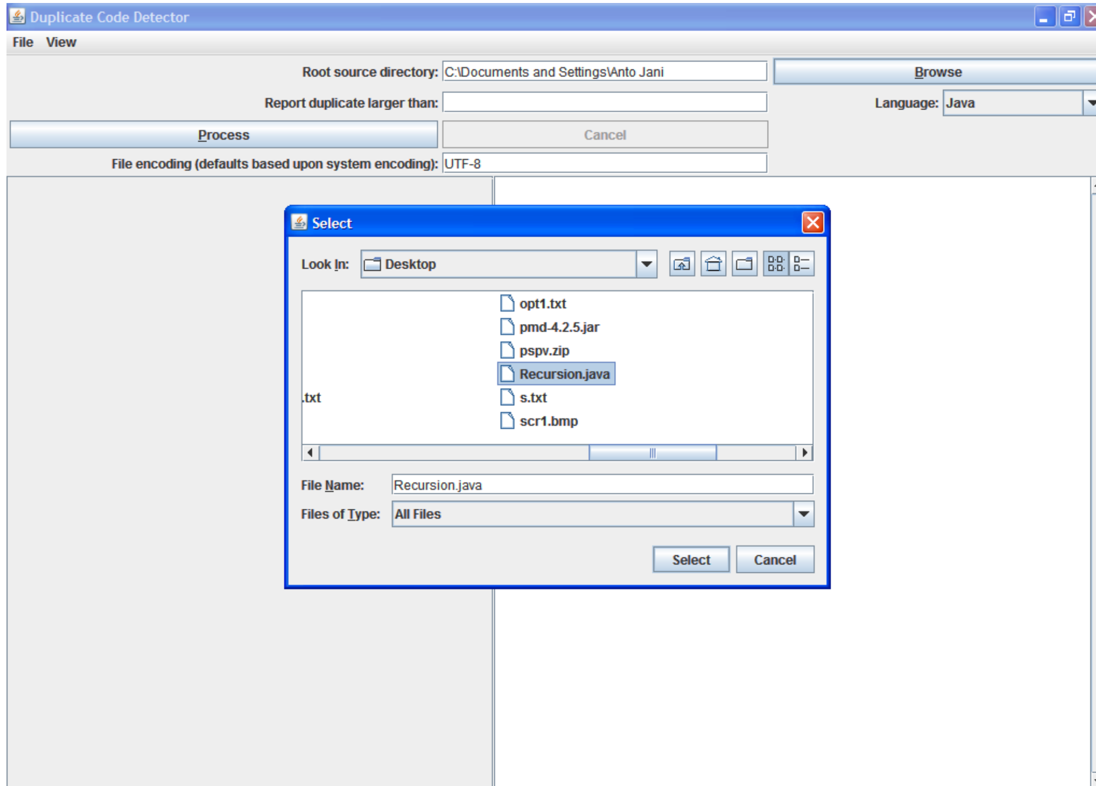


Figure 4.2: loading the database as a file

The source code will be divided into tokens. The tokenization happens exactly like what happens with our source code in a compiler. These tokens are defined in the tool according to code constructs of Java. The source code loaded as database to the tool, it will compare each token with all other tokens of the code fragment. The discovered clones are stored in a file for further analysis.

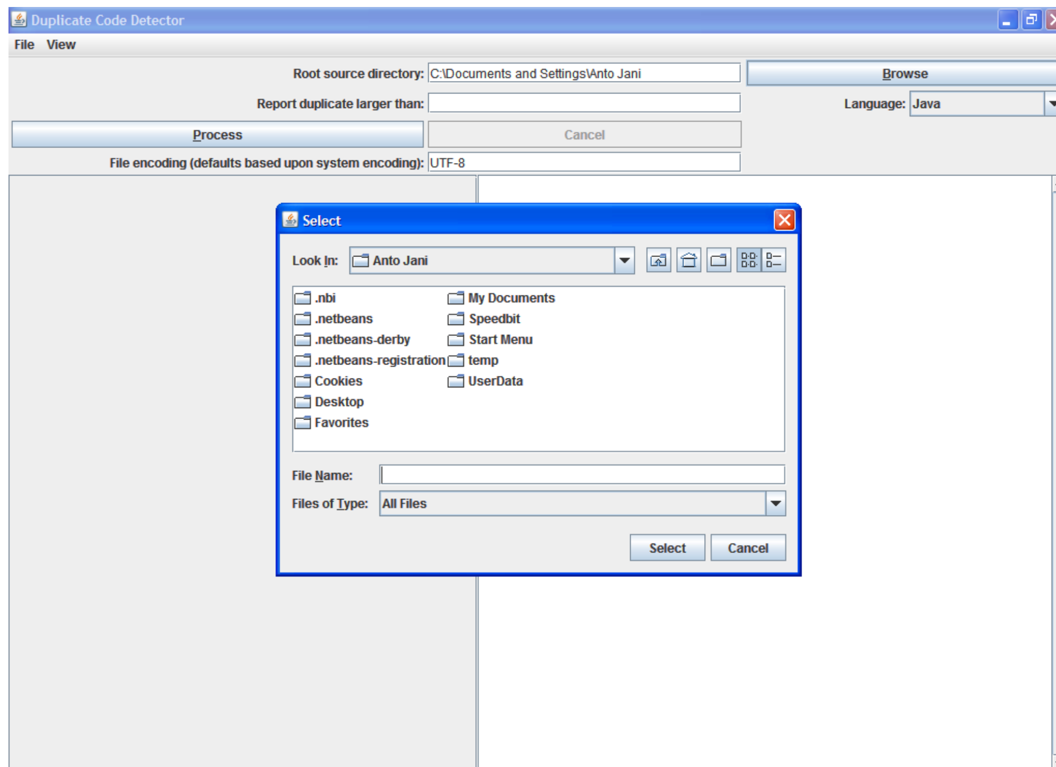


Figure 4.3: loading the database as a folder

After loading the database and selecting the input files/folder to detect the clones using *Process* button. In this process the preprocessing is done by concatenation of the files present in the folder. It makes any number of files present in the folder to form a single large file. Then it processes the file like processing a single file. Then, it normalizes the data of the source code in a single file. Normalization includes three activities which are

i. White space removal
ii. Comments removal and
iii. Unwanted code removal

Whitespace removal is a process of eliminating blank lines and blank spaces used to make the program structure understandable. Comments removal is the process of removing comment lines presented in the given Java code fragments. Finally, unwanted code removal is the process of removing code that is not making any functional difference to the code fragment. For example variables declared and not used anywhere in the program, increment or decrement operation which doesn't make any difference to either outcome of the method or may not affect any calculation further etc.

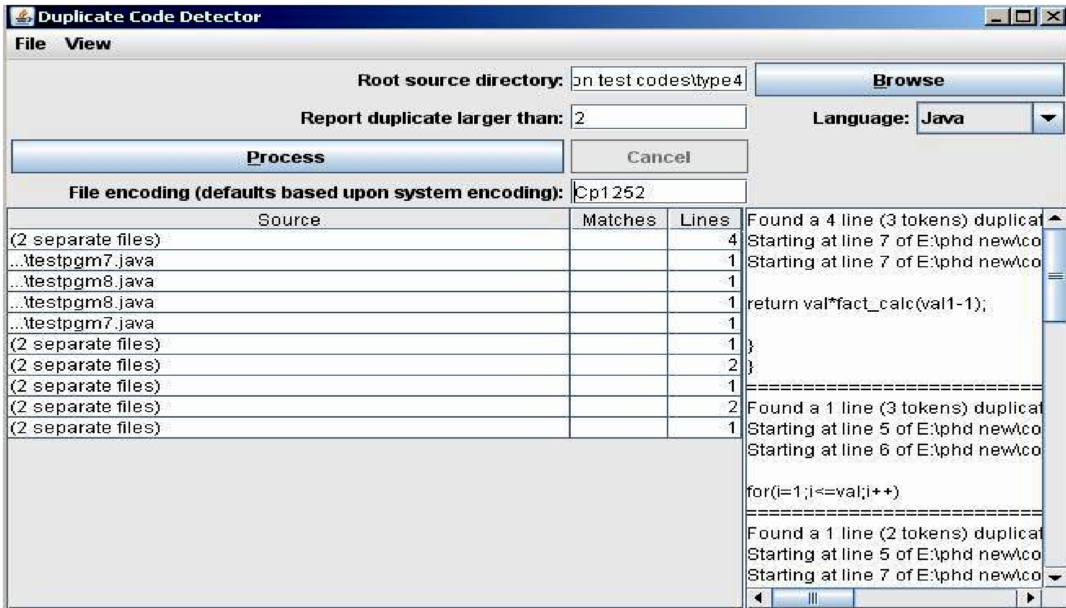


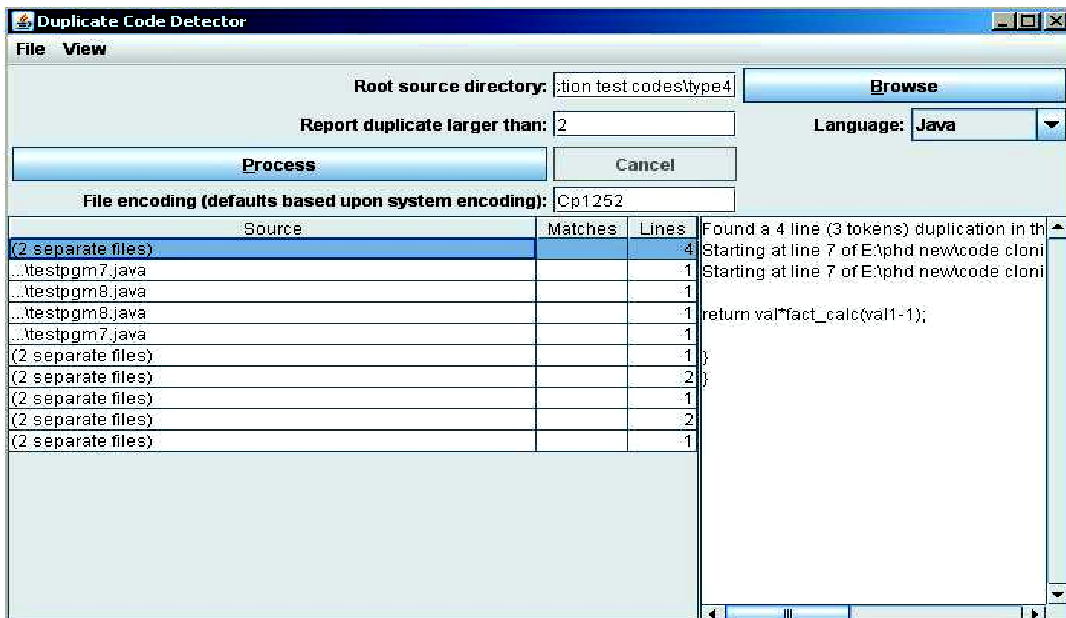
Figure 4.4: Metrics computation and Textual Analysis

The metric analysis finds the potential clones which are described in Figure 4.4. The twelve method level metrics are applied on to the source code to identify these potential clones. Analysis of metrics gives the understanding of the clones which are discovered. These metric values are analyzed and compared with the textual analysis. And clones will be finalized for further processing.

The lower half of the screen displays the results. The *Source* field specifies the files that system has compared and number of lines matched by the *Lines* field. Adjacent text area describes the details of the matched portions of the code fragments.

For detecting the clones presented in the input files, textual analysis is performed in the preprocessed codes. The textual analysis finds 2 types of clones such as type I and type II. It is presented in Figure 4.4.

Figure 4.5: Clone Detection Process



Finally, selecting one of the rows in the *Source* field clones available in the source files are detected in the efficient manner are displayed in the text area and the final output is presented in Figure 4.5. These clones identified through the process have to be analyzed manually and confirm that it is a genuine code and then the process of refactoring will start. And the refactoring is a process that allows us to nullify the negative effective of the code cloning.

Case Study

Consider the following program for a case study of the proposed clone detection technique. The program presented here is a part of a system which is the code of free software available on the World Wide Web. This is a part of a system which is developed for supporting some mathematical calculations and solutions for mathematical equation. To understand where the normalization and other activities happening we gave line numbers to the code.

Clone detection

The two programs were presented in the Appendix A. The following code fragment shown in Figure 4.6 after concatenation and normalization.

```
public class TestFileOne {

            intp,q=1,r;

            double VALUE;

publicint factorial(int n){

if(n == 0){

return 1;

}else{

return n * factorial(n-1);

     }

   }

publicintgcdOne(int a, int b) {

while (b != 0) {

if (a > b) {

        a = a - b;

      } else {

        b = b - a;

      }

    }

return a;

   }

publicintmul(int a, int b){
```

```
int n = 0, p=0;
    p=p+q;
for(int i = 0; i < b; i++){
    n += a;
   }
return n;
  }
        publicint factorial1( int VALUE ){
        for (p=1; p<=VALUE; p++)
        q = q*p;
        return q;
        }
 }
public class TestFileTwo {
publicint factorial2(int n){
if(n == 0){
return 1;
}else{
return n * factorial2(n-1);
   }
  }
publicintgcdTwo(int c, int d) {
while (d != 0) {
if (c > d) {
     c = c - d;
   } else {
     d = d - c;
    }
   }
return c;
```

```
  }
public double mulTwo(double a, long b){
double n = 0.0;
for(long i = 0l; i < b; i++)
      n += a;
return n;
  }
 }
```

Figure 4.6: Normalized Sample Code presented in Appendix A

In the process of file integration two files presented in appendix-A, are concatenated and white spaces and comments were removed. In program 1 line numbers 2,4,6,8,11,13,15,17,19,23,34,36,38,40,42,45,49,51 are the blank lines and were deleted before concatenation. When we look at program-2 of appendix-A, line numbers 2,4,5,7,15,26,31 are blank lines and were removed before concatenation. The comments present in line numbers 9,10,12,48 of program 1 and line number 8 of program 2 are removed in the above pre-processed code fragment. Pre-processing phase involves this file integration and white space and comment removal. Above code fragment is after preprocessing and will move to normalization phase.

The process of normalization is to replace all the identifiers with a common variable name. In this example we used common variable 'S' to replace all the variables presented in the code fragment. The normalization also involves removing the structure of the program. For reader of the program convenience we use tabs and spaces in a line. Normalization removes this structure for textual comparisons. The template conversion is another phase which is associated with normalization. Template conversion is to convert all language constructs into predefined templates. Table 4.1 shows normalized and template converted code of our example.

Table 4.1: Template conversion of sample plrograms

Code	**Normalized template**
public class TestFileOne {	ASP CLASS_NAME{
intp,q=1,r;	DAT S,S,S;
double VALUE;	DAT S;
public int factorial(int n){	ASP DAT FUN_NAME(DAT S){
if(n == 0){	IF{
return 1;	RETURN;
}else{	}ELSE{
return n * factorial(n-1);	RETURN REC_FUNCTION CALL;
}	}
}	}
public intgcdOne(int a, int b) {	ASP DAT FUN_NAME(DAT S, DAT S){
while (b != 0) {	LOOP{
if (a > b) {	IF{
a = a - b;	ASSIGNMENT STATEMENT;
} else {	}ELSE{
b = b - a;	ASSIGNMENT STATEMENT;
}	}
}	}
return a;	RETURN;
}	}

public intmul(int a, int b){
int n = 0, p=0;
p=p+q;
for(int i = 0; i < b; i++){
n += a;
}
return n;
}
public int factorial1(int VALUE){
for (p=1; p<=VALUE; p++)
q = q*p;
return q;
}
}
public class TestFileTwo {
public int factorial2(int n){
if(n == 0){
return 1;
}else{
return n * factorial2(n-1);
}
}
public intgcdTwo(int c, int d) {
while (d != 0) {
if (c > d) {
c = c - d;
} else {
d = d - c;
}
}
return c;
}
public double mulTwo(double a, long b){
double n = 0.0;
for(long i = 0l; i < b; i++)
n += a;
return n;
}
}
	ASP DAT FUN_NAME(DAT S,DAT S){
	DAT S,S;
	ASSIGNMENT STATEMENT;
	LOOP
	ASSIGNMENT STATEMENT;
	}
	RETURN;
	}
	ASP DAT FUN_NAME(DAT S){
	LOOP
	ASSIGNMENT STATEMENT;
	RETURN;
	}
	}
	ASP CLASS_NAME{
	ASP DAT FUN_NAME(DAT S){
	IF{
	RETURN;
	}ELSE{
	RETURN REC FUNCTION CALL
	}
	}
	ASP DAT FUN_NAME(DAT S, DAT S){
	LOOP{
	IF{
	ASSIGNMENT STATEMENT;
	}ELSE{
	ASSIGNMENT STATEMENT;
	}
	}
	RETURN;
	}
	ASP DAT FUN_NAME(DAT S,DAT S){
	DAT S;
	LOOP
	ASSIGNMENT STATEMENT;
	RETURN;
	}
	}

There are seven functions in the above code fragment and they are normalized with variables and templates created for each statement. Templates replaced with original statements like access specifier with ASP, name of the class with CLASS_NAME, variable declaration with DAT S, name of the function with FUN_NAME, conditional statement ‘if-else’ with IF and ELSE, ‘while’ and ‘for’ looping statements with LOOP, return statements of the function with RETURN, and all arithmetic statements with ASSIGNMENT statements etc. The opening and closing braces will be remains the same in converted template also for each functional block.

The next phase is clone detection process. It involves metric analysis and textual comparison. First metric computations will be done for each method, and then each of these templates will be divided into tokens similar to the process that happens with a compiler while compiling a program.

Metrics will be calculated for each method individually. Sample metrics calculations are shown for three methods as following

Table 4.2: Metric calculation for factorial method

Sl. No.	Metrics	Value
1.	No. of lines of code	8
2.	No. of local variables declared	0
3.	No. of conditional statements	1
4.	No. of looping statements	0
5.	No. of arguments passed	1
6.	No. of function calls	n-1
7.	No. of times function called	0
8	No. of return statements	N
9	No. of inherited objects or methods	0
10	No. of virtual functions	0
11	No. of overridden functions	0
12	No. of overloading constructors	0

In Table 4.2 metric calculation for number of function calls and number of return statements are depending on the input number and the value of 'n' will be replaced by that value accordingly at the time of metric calculation.

Table 4.3: Metric calculation for gcdone method

Sl. No.	Metrics	Value
1.	No. of lines of code	10
2.	No. of local variables declared	0
3.	No. of conditional statements	1
4.	No. of looping statements	1
5.	No. of arguments passed	2
6.	No. of function calls	0
7.	No. of times function called	0
8	No. of return statements	1
9	No. of inherited objects or methods	0
10	No. of virtual functions	0
11	No. of overridden functions	0
12	No. of overloading constructors	0

In Table 4.3 gcdone() function all metrics are clearly calculated well before compilation of the program unlike factorial method in the above calculation.

Table 4.4: Metric calculation for mul method

Sl. No.	Metrics	Value
1.	No. of lines of code	8
2.	No. of local variables declared	2
3.	No. of conditional statements	0
4.	No. of looping statements	1
5.	No. of arguments passed	2
6.	No. of function calls	0
7.	No. of times function called	0
8	No. of return statements	1
9	No. of inherited objects or methods	0
10	No. of virtual functions	0
11	No. of overridden functions	0
12	No. of overloading constructors	0

Like this metric calculations will be done for all the methods and will be analyzed by the system to identify functional similarities. This analysis is done to find type IV clones and returns them as clones. In this example it returns factorial and factorial1 functions as clones.

Let us observe the final results of the method for the given example.
Initial result screen shows as in the Figure 4.7. It shows the beginning lines of the two programs presented in the appendix –A and other line by line clones are displayed.

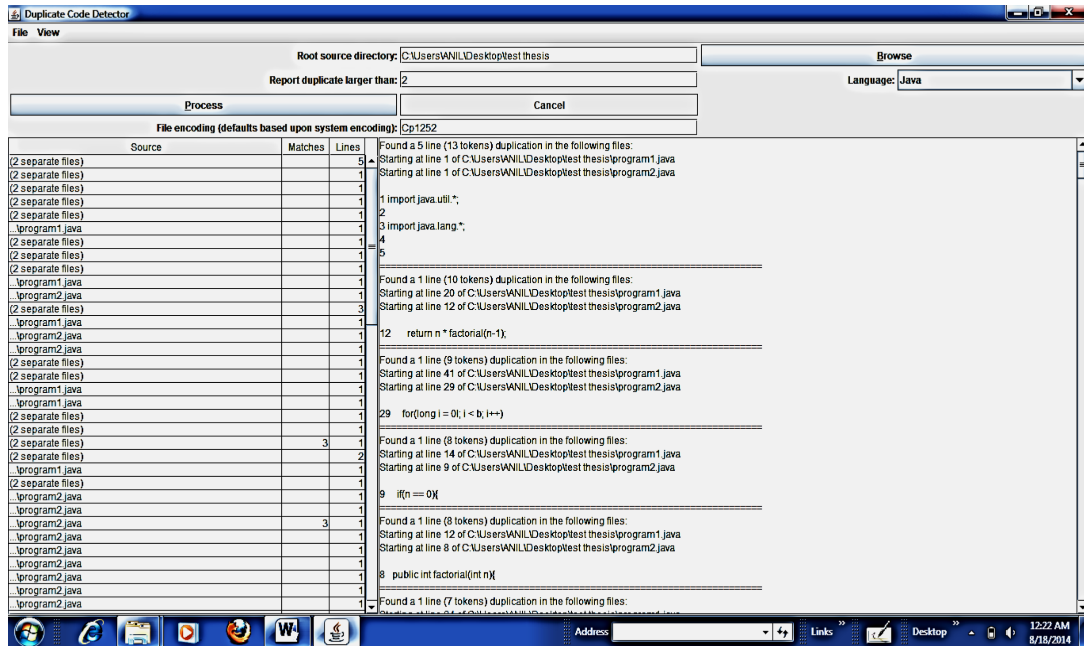


Figure 4.7: Initial results screen

When we scroll down and find the results for method level clones as we see in the example. In program1 factorial method implemented using recursive function and in program2 it is exactly implemented as same in program1 and represented as factorial2. It resembles code clone of type I and the result is shown in Figure 4.8.

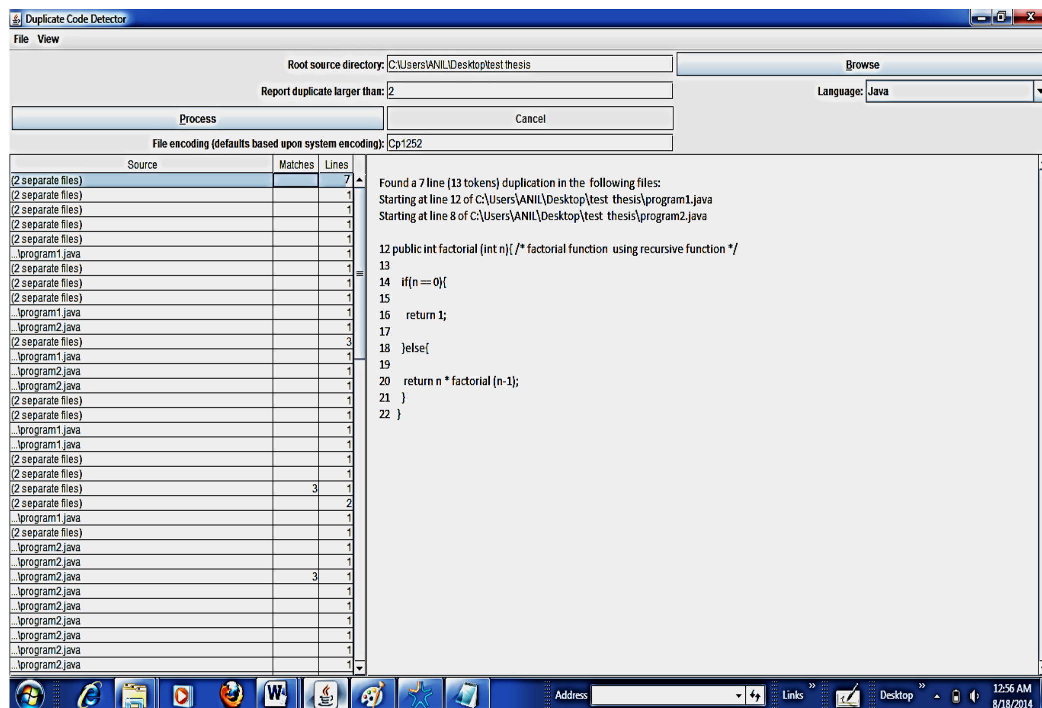


Figure 4.8: Detected type I clones

The other clones detected were gcd One method of program1 and gcd Two method of program2 is similar but not same. The variables presented in gcd One are named as 'a' and 'b', are replaced by 'c' and 'd' but other statements are same. System cannot identify it through simple textual comparisons, but by normalization method our system will be able to identify these clones. It resembles type II clones and results are as shown in Figure 4.9.

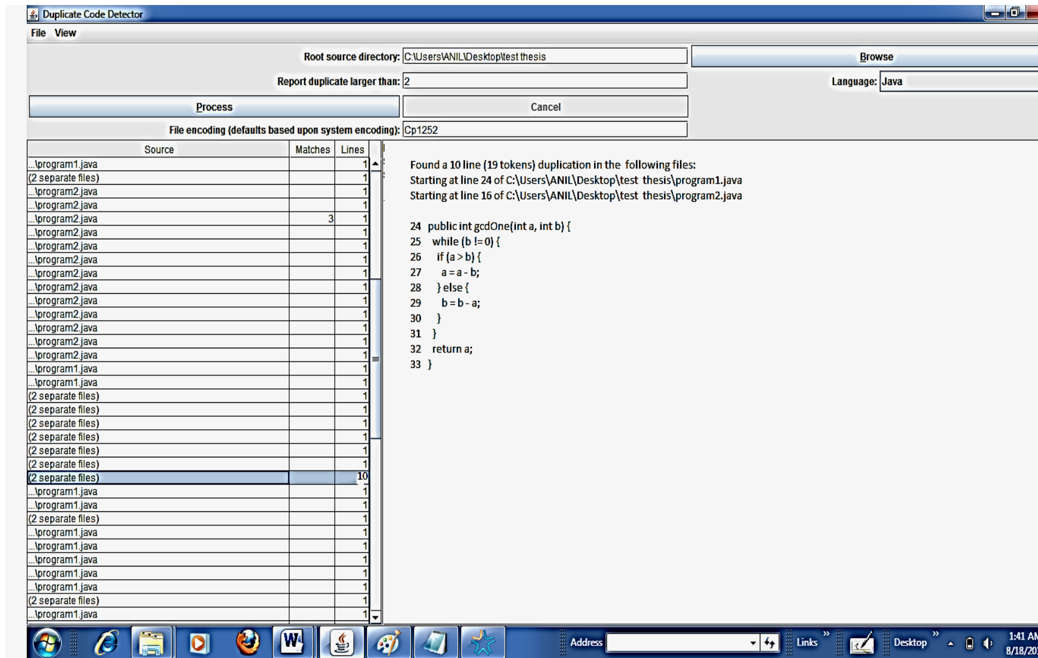


Figure 4.9: Detected type II clones

The method mul in program1 and mulTwo in program is not similar. The variables passed to this method are changed and their data types are also changed. The number of variables declared within the function is also changed, but an additional variable declared is 'p' and it has no functional value with in this function so, it can be considered as a clone. This resembles type III clone and is shown in Figure 4.10.

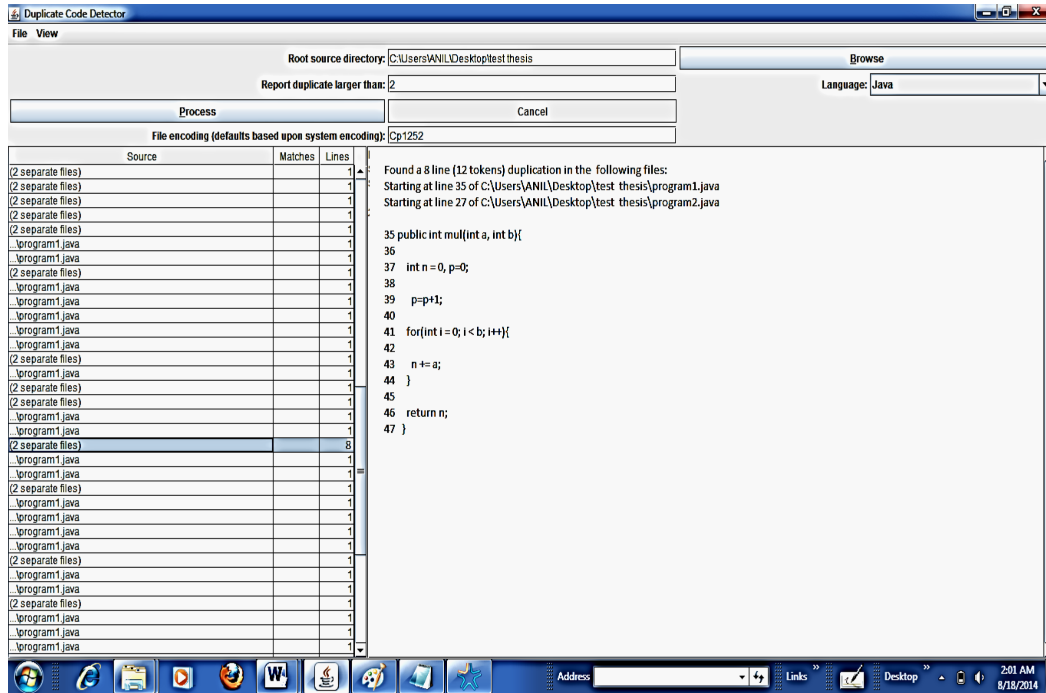


Figure 4.10: Detected type III clones

In program1 factorial method is implemented by recursive function but in the same program another method named as factorial1 is implemented using a looping structure. Though these two methods are implemented in two different ways both are calculating factorial of a number only. So, we consider these as clones of the system. Our system identifies these clones and resembles as type IV clones which is shown in Figure 4.11.

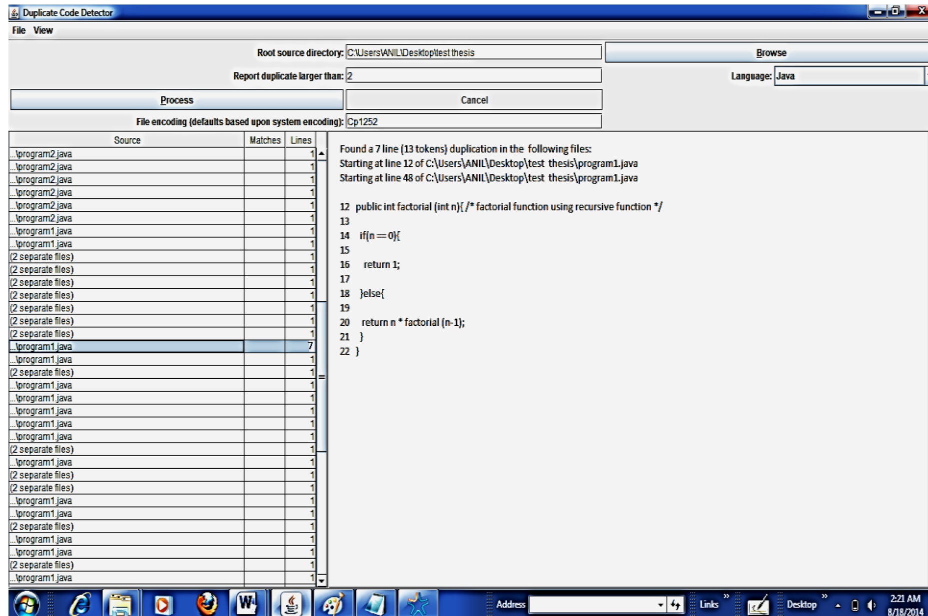
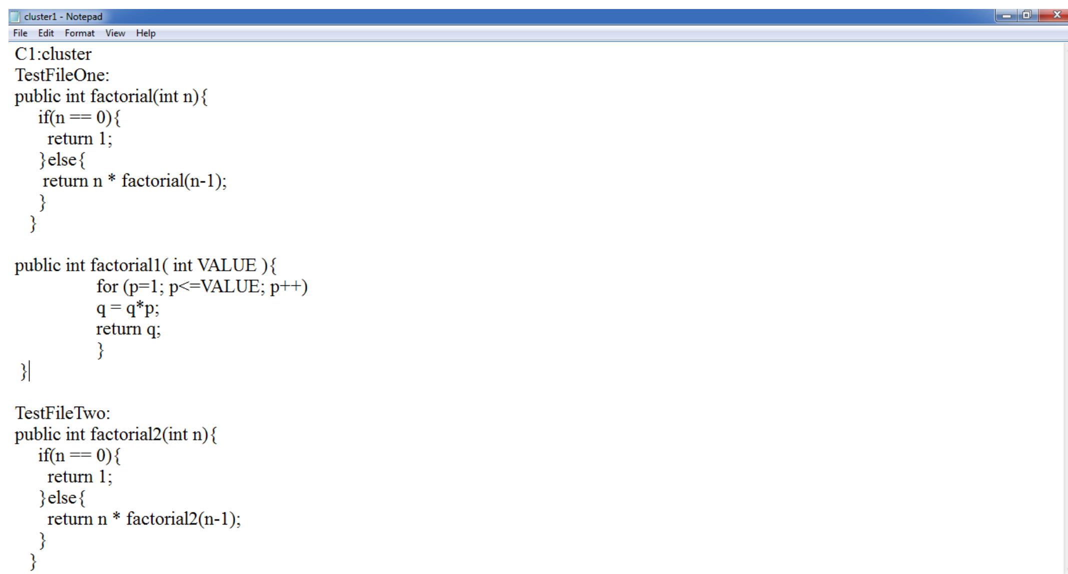


Figure 4.11: Detected type IV clones

Clone clustering

The identified code clones which are of different types will be grouped together to form clone clusters. These clone clusters will be useful to analyze the detected clones from the files which we have given as input. These clusters are stored in a text file and will be utilized to find the clones detected by the tool are actual clones are not. These clusters are named as C1, C2, C3 and so on. Some of the clone clusters identified for this example are presented in Figure 4.12.



```
cluster2 - Notepad
File Edit Format View Help
C2:cluster
TestFileOne:
  public int gcdOne(int a, int b) {
    while (b != 0) {
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
    return a;
  }

TestFileTwo:
public int gcdTwo(int c, int d) {
    while (d != 0) {
      if (c > d) {
        c = c - d;
     } else {
        d = d - c;
      }
    }
    return c;
  }
```

Figure 4.12 Clone Cluster representation

Refactoring

Finally refactoring is applied on the programs which are given as inputs. We applied *pull-up*method, hence we created a class named as InhClass. All three methods that are identified as clones were pulled to the parent class and these two classestestfileOne and testfileTwo were made as inherited classes (child classes) from the parent class. All the occurrences of these three methods were replaced by function calls to these methods in the parent class. Any changes required to these methods require modifications at one place now. All three occurrences of factorial(), factorial1() and factorial2() methods were made as function calls and one method with name factorial() is presented in InhClass. Similarly gcdOne() and gcdTwo() methods were replaced with function calls and a method gcd() was created in parent class. It is also same with the case of mul() and mulTwo() methods. A method named mul() was created and placed in parent class and these two methods are replaced with function calls.

Performance Measure

There are many parameters in the code clone literature to identify the efficiency of a clone detection technique. For this research work we stick to only precision and recall values to recognize the efficiency of the technique. Some clone detection techniques proved that they are efficient in either precision or recall but not both. Our method is assessed on both precision and recall values.

Clone detection result accuracy refers to a combination of both precision and recall. Precision denotes the probability that a randomly chosen candidate clone group is relevant. Recall denotes the probability that a relevant clone group, chosen from the hypothetical set of all relevant clone groups, is contained in a detection result.

$$\text{Precision},\ P = \frac{\textit{Number of clones correctly found}}{\text{Total Number of clones found}}$$

$$\text{Recall},\ R = \frac{\textit{Number of clones found correct}}{\textit{Total number of possible existing clones in the source code}}$$

Let us examine the precision and recall values observed in different popular clone detection techniques.

Table 4.5: precision and recall values of popular clone detection techniques

	CC Finder (type 1,2)	Clone Dr (type 1,2,3)	Jplag (type 1)	Moss (type 1)	Suffix Tree method (type 1,2,3)	Our method (type 1,2,3, 4)
Precision	72	100	82	73	92	98
Recall	72	9	12	10	75	96

In table 4.5 we can notice that CloneDr shows perfect 100% precision, which indicates the tool will not produce any false positives in its clone detection process. But the other side recall value is lowest, that is 9%. That shows tool is not able to find all major clones which are present in the system. CloneDr is a tool which is developed on the basis of Abstract syntax tree. CCFinder is a token based tool which shows a good balance between precision and recall values. Precision (72%) shows only 28% of false positives and Recall (72%) shows it is finding most of the original clones in the system. Suffix tree method shows better results in terms of Recall value (75%) when it compared with all other methods includingCCFinder. But it shows the low Precision value than cloneDr. The other two tools JPlag and Moss are basically plagiarism tools which can also be used as clone detection tools. Results of these two tools are almost similar in recall value but a little improvement in the precision value of JPlag. These are the results we obtained on checking the tools with an average size of software. Our method showed better results of precision and recall values. Because of identifying the functional clones, that is type-4 clones our methods precision value became 98% and we were succeeded in getting highest value for recall. The observations for CCFinder, CloneDr and Jplag and Moss are almost similar to the experiment conducted by Burd and Bailey [64]. Their experiment is conducted on a system which is having large sized Lines Of Code (LOC).

When we look at the above tools they are limited to detect particular type of clones a token based tool CCfinder was able to find clones of type I and type II only. Anti-plagiarism tools Jplag and Moss are able to show only type I clones. The other two tools which are tree based CloneDr and Suffix tree method were able to find type I, type II and type III. But our method was able to find all four types of clones with better precision and recall values.

The precision and recall of the proposed method will evaluate the proposed system's efficiency. In the process of proving the efficiency of the proposed method following table compares the Precision and recall values of the suffix tree method which is known to be an efficient method in terms of precision and recall values till now.

Table 4.6: Comparison table

Methods	**Performance Measure**	
	Precision	**Recall**
Proposed method	98	96
Suffix Tree method	92	75

The following graph describes the comparison of performance measure for Table 4.6. As the performance of the method is considered in terms of precision and recall only, following graph shows precision and recall values of the method from 0 to 100 in percentage on Y-axis. There is no model explicitly calculates all four types of clones so we compared our method to suffix tree method, which is used to find functional clones.

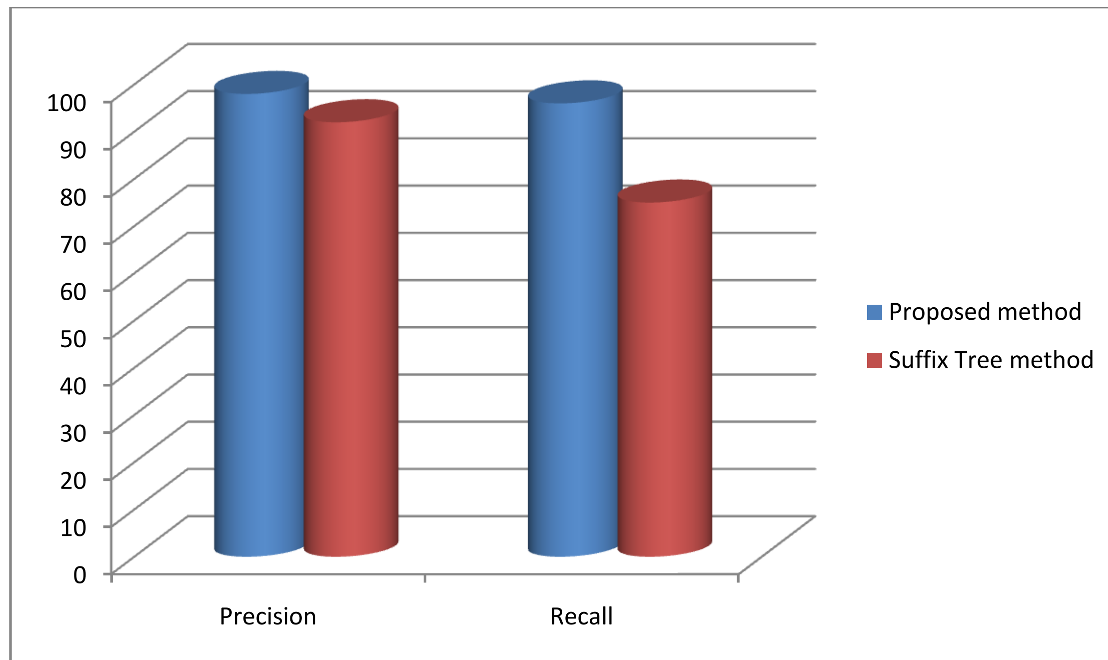


Figure 4.13 : Comparison graph of Precision and Recall
(Precison and Recall of proposed and suffix tree methods on X-axis is plotted against Percentage of performance for this precision and recall on Y-axis)

From the Figure 4.13, we observe that our proposed method detects the clones available in the source files in an efficient mannar. We compare the proposed work with the already existing clone detection tool which uses suffix tree method that will give less precision and recall rate when compared to our proposed method.

The precision and recall values are finalized for the method from calculating precision and recall values of the code segments of different sizes considered and took the average of all. These values are shown in Table 4.7

Table 4.7: Performance of the method for different lines of code

Number of Lines of code	500	1000	5000	10000
Precision	98.7	98.3	98.5	98.2
Recall	96.6	96.4	96.5	96.3

As shown above in Table 4.7 the precision value for the program which is about 500 lines of code is obtained as 98.7 and is the highest in the table for the minimum size program. If the size of the code is 1000 lines then it is 98.3. One can say about our method that if the size is increased the precision value is decreased but we got higher precision value for the code with size 5000 lines as 98.5 which is slightly higher than one with 1000 lines. Finally the precision value of 10000 LOC sized code is 98.2. so we conclude our method's precision value is not less than 98.
The recall value is also obtained almost similar to the precision value when it consider the difference between various sizes of code. It is 96.6 if the size is 500 lines. The recall value for 1000 lines 96.4 and 5000 lines is 96.5. like precision recall value also little less to 1000 lines and slight improvement to 5000 lines. When it comes to 10000 lines of code it became 96.3, so we conclude that our method's recall value will not be less than 96.

In addition our proposed approach supports refactoring of the identified clones. This refactoring can be done based on the clones that are discovered. Two types of refactoring approaches are used in our method. These two are *extract* and *pullup* methods. Extract method allows us to replace the idetified cloned lines of code to form as a method and make

calls each time it is repeated in the code. In Pullup method if we have methods in child classes repeated, these methods are pulled up to the parent class in the inheritance relationship. These are two method level refactoring techniques used in our approach.

Conclusion:

The working model of the system explains how to detect clones. Case study proved that proposed method is capable of detecting all types of clones. Comparison graph states that the method works efficiently.

Reference

[1] M. Fowlor. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.
[2] R. H. Page. http://www.refactoring.com/.
[3] MagielBruntink, Arie van Deursen,Remco van Engelen, and Tom Tourwe, "On the Use of Clone Detection for Identifying Crosscutting Concern Code", Ieee Transactions On Software Engineering, Vol. 31, No. 10,pp. 804-818, October 2005
[4] Abouelhoda M.I., Kurtz S. and Ohlebusch E, "The enhanced suffix array and its applications to genome analysis", In Proc. Workshop on Algorithms in Bioinformatics, vol. 2452,pp. 449–463, Berlin, 2002
[5] Hamid Abdul Basit and Stan Jarzabek, "Detecting Higher-level Similarity Patterns in Programs", European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp 1-10 Lisbon, Sept. 2005
[6] Lingxiao Jiang, Zhendong Su and Edwin Chiu, "Context-based detection of clone-related bugs", Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 55 – 64, New York, USA, 2007.
[7] Chanchal Kumar Roy and James R Cordy, "A Survey on Software Clone Detection Research", Computer and Information Science, Vol. 115, No. 541, pp. 115, 2007
[8] J Howard Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON'93)*, pp. 171-183, Toronto, Canada, October 1993.
[9] Zhenmin Li, Shan Lu, SuvdaMyagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In *IEEE Transactions on Software Engineering*, Vol. 32(3): 176-192, March 2006.