Code Clone Detection by Evaluating Combinations of Software Metrics

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Geetika
(801231008)

Under the supervision of:
Mr. Rajkumar Tekchandani
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

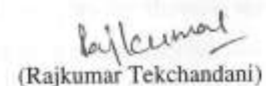
Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Code Clone Detection by Evaluating Combinations of Software Metrics*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Rajkumar Tekchandani* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

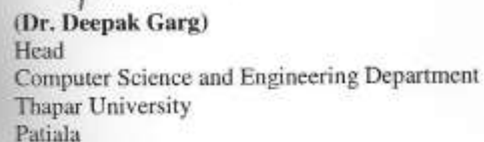
(Geetika)
801231008

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

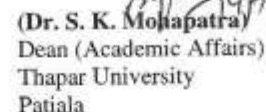


(Rajkumar Tekchandani)
Assistant Professor,
Computer Science and Engineering Department,
Thapar University,
Patiala.

Countersigned by



(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

No volume of words is enough to express my gratitude towards my guide, **Mr. Rajkumar Tekchandani**. I thank my supervisor for his time, patience, discussions and valuable comments. He has been very concerned and have aided for all the material essential for the preparation of this thesis report. His enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Deepak Garg**, Head of Computer Science and Engineering Department and **Ms. Damandeep Kaur**, P.G Coordinator, for motivation and inspiration that triggered me for the thesis work.

I also want to express my gratitude to **Dr. S. K. Mohapatra**, Dean of Academic Affairs, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their help, cooperation and affection, which made my stay at Thapar University memorable.

Last but not the least, I would like to thank my parents and Almighty for showing me the right direction, without their blessings none of this would have been possible.

Abstract

Code cloning can be defined as an act of reusing a code segment by copying it from one section of the software and then pasting it with or without some slight alterations into other sections of the software. It is a basic means of software reuse. Most of the large software systems consist of numerous code clones because code cloning has been extensively used within the software development design community. Although it is easy to develop softwares by applying code cloning but cloning is problematic for both maintenance and quality of a software system. Code cloning results in increasing maintenance effort because during the maintenance phase of a software if an error or bug is found in one code fragment, then all its corresponding clones should be find out to detect and correct the same error or bug. Because of the problems caused due to code cloning, there is a need to detect code clones and the detection process is known as clone detection. A number of code clone detection techniques have been proposed so far. In this thesis, an approach is proposed for selecting a set of relevant metrics for detecting code clones using metrics based code clone detection techniques. The proposed approach evaluates a set of independent metrics and these metrics are evaluated on the basis of their precision and recall values in clone detection starting from all combinations of one metric and then gradually increasing the number of metrics in the metrics combinations until the complete set of metrics involved in the approach are evaluated. The result of implementing the proposed approach on a C language software system is provided as example.

Table of Contents

Certificate...**i**

Acknowledgement...**ii**

Abstract.....**iii**

Table of Contents....**iv**

List of Figures...**vi**

List of Tables....**vii**

Abbreviations....**viii**

Chapter 1 Introduction **1**

1.1 Code Cloning..1

1.2 Clone Detection and Related Terminology..2

1.2.1 Clone Pair and Clone Class...2

1.2.2 Code Clone Types..2

1.3 Reasons of Code Cloning..4

1.4 Disadvantages of Code Cloning..7

1.5 Benefits and Need of Clone Detection..8

1.6 Motivation and Objective..8

1.7 Thesis Outline..9

Chapter 2 Literature Survey **11**

2.1 Existing Techniques of Detecting Code Clones.................................11

2.2 Literature Survey of Metrics Based Clone Detection Techniques.........13

2.3 Properties of Code Clone Detection Techniques...............................18

Chapter 3 Problem Statement **21**

3.1 Gap Analysis...21

3.2 Objective..22

Chapter 4 Proposed Approach and Implementation **23**

4.1 Metrics Used in the Approach..23

4.2 Proposed Approach..24

4.2.1 Comparison Algorithm to Detect Clone Classes..................25

4.3 Flow Diagram of Proposed Approach...27

4.4 Implementation...28

Chapter 5 Experimental Results **34**

Chapter 6 Conclusion and Future **38**

6.1 Conclusion...38

6.2 Future Scope...38

References **39**

List of Publications **43**

List of Figures

Figure No.	Figure Description	Page No.
Figure 1.1	Example of Code Clone	1
Figure 1.2	Clone Class and Clone Pair	2
Figure 1.3	Exact Clones	3
Figure 1.4	Renamed Clones	3
Figure 1.5	Near Miss Clones	4
Figure 1.6	Semantic Clones	4
Figure 1.7	Reasons of Code Cloning	6
Figure 2.1	Generic Flowchart of Metric Based Techniques	14
Figure 2.2	Cloning Scale	15
Figure 2.3	Parsing of a Sample Method	17
Figure 4.1	Steps Followed in the Proposed Approach.	27
Figure 4.2	Startup Page.	28
Figure 4.3	Add Project Directory	29
Figure 4.4	Export Metrics Detail as CSV File	29
Figure 4.5	CSV File of Metrics Detail	30
Figure 4.6	Specify the Source Code File Selection Directory	30
Figure 4.7	Export Metric as CSV File	31
Figure 4.8	Enter CSV File as Input	31
Figure 4.9	Metrics Values Stored into Database	32
Figure 4.10	Resulting Clone Classes	33
Figure 5.1	Resulting Clone Classes for Best Set of Metrics.	35
Figure 5.2	Resulting Clone Classes for Best Set of Metrics	35
Figure 5.3	Precision Graph for Different Number of Metrics Used	36
Figure 5.4	Highest Precision and Recall Values for Different Metric Combinations.	37

List of Tables

Table No.	Table Description	Page No.
Table 2.1	Clone Detection Techniques	13
Table 2.2	Method Level Metrics	16
Table 2.3	Summary of Comparison between Existing Metric Based Approaches	18
Table 2.4	Properties of Code Clone Detection Techniques	19
Table 4.1	Metrics Name and Meaning	24
Table 5.1	Description of Case Used	34
Table 5.2	Highest Precision and Recall Values	34
Table 5.3	Representation of Metric Combinations	37

List of Abbreviations

AST	Abstract Syntax Tree
PDG	Program Dependence Graph
CP	Clone Pair
CC	Clone Class
LOC	Line of Code
CSV	Comma Separated Values

Chapter 1

Introduction

Most of the software systems consist of a large number of identical code segments. These identical code segments are known as code clones. According to previous research, a software system consists of about 7% to 23% of cloned code [1]. Although code cloning makes the task of software development easy but at the same time it is problematic for the maintenance phase of the software because if the cloned code consists of an error, then the same error needs to be detected and corrected from the clones of that code in a consistent way. It also increases the development cost of software and lead to a bad quality software system as it results in increasing software size. Code clones are said to be undesirable for numerous reasons.

1.1 Code Cloning

Code cloning can be defined as an act of reusing a segment of code by copying it from one section of the software and then pasting it with or without some slight alterations into another section of the software. An example of code clone is shown in figure 1.1.

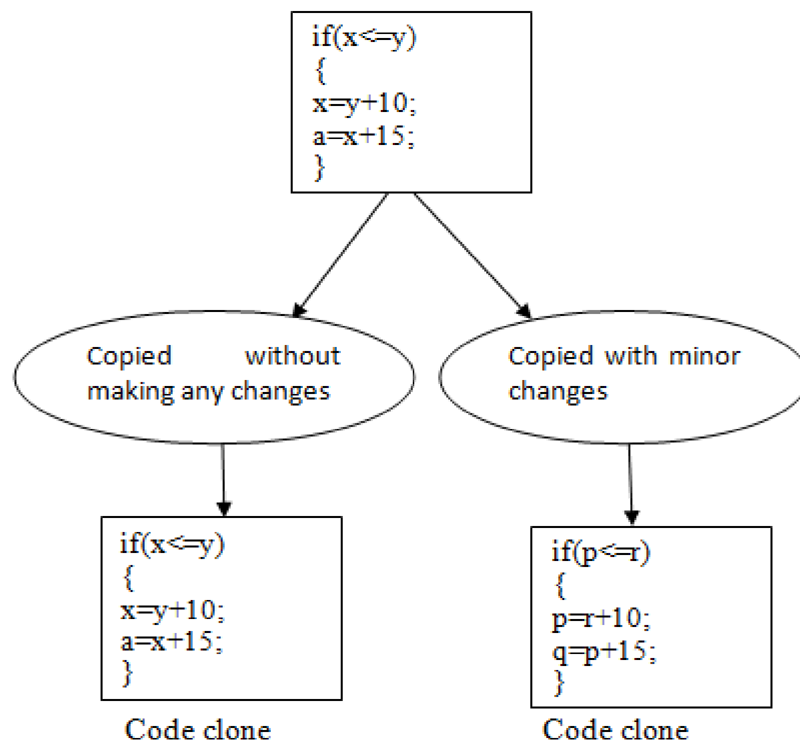


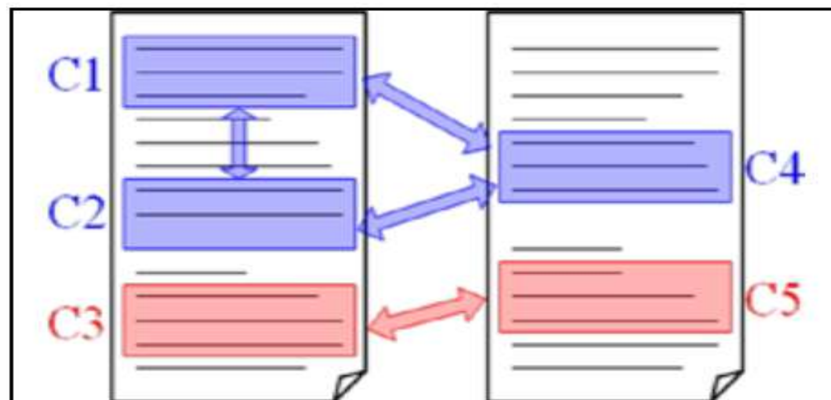
Figure 1.1: Example of Code Clone.

1.2 Clone Detection and Related Terminology

Maintenance is said to be the most expensive part of software development. Code cloning not only results in raising the maintenance effort but also increases the probability of errors in softwares. Because of the problems caused due to code cloning, it is required to identify code clones from softwares. The identification process is known as clone detection.

1.2.1 Clone Pair (CP) and Clone Class (CC)

Most of the clone detection tools give results as clone pairs or clone classes. Two code segments form a clone pair, if they are related to each other by an equivalence relation [2]. An equivalence relation holds all reflexive, symmetric and transitive relations. A clone pair is defined as a pair of matching code segments. Clone class is defined as a set of code segments with similar code portions. Each code segment in a clone class forms a clone pair with other code segments of that class. The concept of clone pair and clone class is clearly explained in figure 1.2.



Clone Pair	Clone Class
(C1, C2)	{C1, C2, C4}
(C1, C4)	{C3, C5}
(C2, C4)	
(C3, C5)	

Figure 1.2: Clone Class and Clone Pair [3].

1.2.2 Code Clone Types

Code segments can be identical in two ways. Either they can be identical on the basis of their program text or they can be functionally identical. On the basis of program text similarity, code clones can be classified into 3 types:

i. **Type-1 Clones**

If a code segment is copied as it is with some minor amendments in whitespaces, layout and comments then it comes under type-1 or exact clones. In figure 1.3, code segment 2 is an exact copy of code segment 1.

Code1:-	Code 2:-

int s=0, p=1, n=5;
while (n > 0)
{
s= s+n;
p=p* n;
n= n-1;
}
	int s=0, p=1, n=5;
	while (n > 0)
	{
	s= s+n;
	p=p* n;
	n= n-1;
	}

Figure 1.3: Exact Clones.

ii. **Type-2 Clones**

If a code segment is copied with some amendments in name of the variables, functions, types and identifiers as shown in figure 1.4 then it comes under type-2 or renamed clones.

Code1:-	Code 2:-

int s=0, p=1, n=5;
while (n > 0)
{
s= s+n;
p=p* n;
n= n-1;
}
	int a=0, b=1, c=5;
	while (c > 0)
	{
	a=a+c;
	b=b*c;
	c=c-1;
	}

Figure 1.4: Renamed Clones.

iii. **Type-3 Clones**

If a code segment is copied with some changes like insertion or deletion of statements along with change in name of variables, functions and type, then it comes under type-3 or near miss clones. An example of this type of clone is shown in figure 1.5.

Code 2:-	Code 3:-

int a=0, b=1, c=5;
while (c > 0)
{
a=a+c;
b=b*c;
c=c-1;
}
	int a=0, b=1, c=5,s=0;
	while (c > 0)
	{
	a=a+c;
	b=b*c;
	s= a+b;
	c=c-1;
	}

Figure 1.5: Near Miss Clones.

iv. Functional similarity

If two code segments perform the same functionality but they are having different syntax, then they are said to be type-4 or semantic clones. These clones are the most difficult to detect. In case of semantic clones, it is not necessary that the code clone is a copy of any other code segment. An example of type-4 clones is shown in figure 1.6.

Code 1:-	Code 2:-

int num1=2, num2=6, i, prod=0;
for(i=1; i<= num2; i++)
{
prod+=num1;
}
	int num1=2, num2=6, prod;
	prod=num1 * num2;

Figure 1.6: Semantic Clones.

1.3 Reasons of Code Cloning

There are numerous reasons of code cloning. Most of the times, it occurs because programmers find it easier to copy the code and paste it rather than designing the code from scratch. Another reason for the developers to do code cloning is that strict time constraints are given to them. Another situation where developer may duplicate code is lack of problem understanding, but they have the code that can do the required functionality. Code cloning is said to be very problematic for industrial software [4] [5]. Software cloning causes lot of negative effects on software quality [6]. It not only increases the maintenance efforts but it also increases the probability of bugs. Figure

1.7 gives a tree diagram of factors responsible for code cloning. A description of some of the factors responsible for code cloning is given below [7]:

i. **Time Limit**

A main reason responsible for existence of code clones is that a certain time constraint is assigned to developers to finish a project. To finish the project in time, developers just reuse the already existing code.

ii. **Developer's Inadequate Knowledge**

When developer doesn't have sufficient knowledge regarding a problem domain then he goes for existing solutions of the problem. Once he got a solution, he transforms that solution according to his requirement.

iii. **By Accident**

Code cloning may be performed accidentally [8]. There may be a case that two software developers may come with the same solution. Technically we can't say them clones because they are not copied knowingly. But they are identified as clones by clone detection tools.

iv. **Incorrect Way of Measuring Developer's Performance**

Sometimes a developer's productivity is computed according to the number of LOC (Line of Code) he writes in a given interval of time. Therefore, developer's attention is on increasing the number of LOC. He copies the existing code again and again to increase the number of LOC.

v. **Reuse**

The prime reason of code duplication is reuse approach. The easiest kind of reuse is to copy and paste already existing code with or without some amendments.

vi. **Integration of Two Identical Systems**

Sometimes developers integrate two software systems performing identical functions to create a new system. Even though separate team members have developed these systems, the integrated system may result in producing clones as identical functionalities are executed in the two systems.

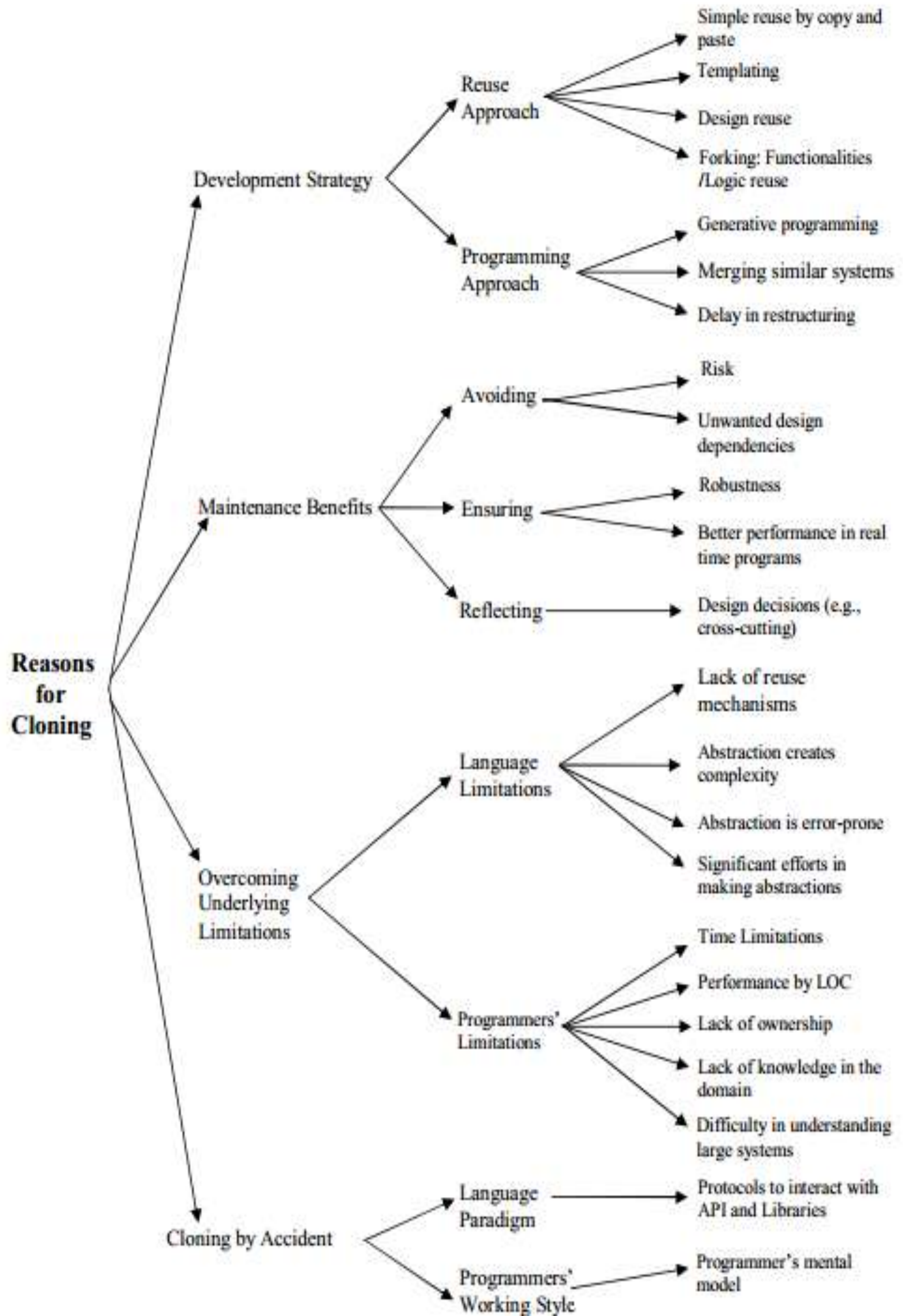


Figure 1.7: Reasons of Code Cloning [7].

vii. **Risk in Writing Fresh Code**

In order to avoid risk, the developers prefer to use existing code instead of writing new code. There may be chances of errors and bugs in new code but the existing code is already tested [9].

viii. **Using Generative Programming**

When generative programming approach is used to generate code, it results in large number of code clones because in this approach, same template is used to produce the similar logic.

1.4 Disadvantages of Code Cloning

Although it is easy to develop softwares by applying code cloning, cloning can be critical for both maintenance and quality of a software system [10]. Some of the problems caused due to cloning are [7]:

i. **Increased Probability of Defects**

Duplication of the source code may also increase the probability of bug propagation in the system because if the original code contains a bug then its clone will also contain the same bug [11].

ii. **Increased Resource Requirement**

Code cloning results in increasing the system size due to which compilation time as well as memory requirements for the system also get increased. It may also results in expensive software and hardware upgrades.

iii. **Increased Maintenance Work and Cost**

Code cloning may result in increased maintenance effort because during the maintenance phase if an error or bug is found in one code fragment, then all its corresponding clones should be find out to detect the same error or bug. This makes maintenance a complex and time consuming task. Code cloning multiplies the effort required during maintenance [12].

iv. **Increased Chances of Bad Design**

Code duplication also causes unfavourable effects on the system's design. It results in poor abstraction and raises difficulty in reusing the code in future projects [13].

v. **Inconsistent Updates**

Code cloning may result in inconsistent updates because if there is a need to modify a piece of code, one needs to modify all clone segments of that piece of code.

1.5 Benefits and Need of Clone Detection

Clone detection not only helps in improving the code quality through refactoring of duplicated code, there are various other advantages of clone detection which are listed below:

i. **Detection of Library Candidates**

If a code segment is duplicated and reused many times in a software system, then it demonstrates its usability [14]. Therefore, this segment can be used as library candidate.

ii. **Helps in Reducing Code Size**

If the detected code clones are replaced by function calls to a generic code segment performing the same functionality as that of the code clone, then it results in reducing the complexity and size of software system [15]. It also improves maintainability and readability of code.

iii. **Helps in Finding Usage Patterns**

When all cloned segments of a piece of code are detected, then the usage patterns of that piece of code can also be found out.

iv. **Better Understanding of Problem**

If working of a cloned segment is apprehended, one is able to understand the working of all duplicate code segments of the cloned segment.

v. **Detects Malicious Software**

Clone detection can be helpful in detecting malicious code. This can be done by matching one malicious piece of code with other and finding which piece of code matches with the malicious code.

1.6 Motivation and Objective

Code cloning is problematic for the maintenance phase of the software because if the cloned code consists of an error then we need to detect and correct the same error

from the clones of that code in a consistent way. It also increases the development cost of software and lead to a bad quality software system as it results in increasing software size. It also leads to increased effort of code enhancement and adaptation. Because of the problems caused due to cloning, there is a need to detect clones.

Detection of code clones is advantageous in many situations such as detecting library candidates, compacting software size, code understanding, detecting malicious software and usage patterns as explained above. For detection of cloned code, a number of techniques have been proposed so far that are quite efficient in detecting clones. But some of these techniques are very complex and some techniques take a huge amount of time. So, there is a need for a simple, time efficient and space efficient clone detection technique.

The objective of this thesis is to present an approach for selecting a set of relevant metrics for detecting code clones using metrics based code clone detection techniques. The aim is to propose an approach that is simple, light-weight and provides an accurate way of metrics based code clone detection. In the proposed approach the focus is on overcoming a major limitation of metrics based techniques of clone detection, which is less precision value.

To achieve this objective, the following steps need to be followed:

i. Study the various existing techniques of code clone detection.
ii. Propose an approach that can be used to select a set of relevant metrics which can be used in metrics based code clone detection.
iii. Implement the proposed approach and perform the experiment on a software system, so that the proposed approach and its results can be clearly understood.

1.7 Thesis Outline

The rest of the thesis consists of five main chapters:

Chapter 2 provides the review of the various existing code clone detection techniques and compares metrics based code clone detection techniques, their advantages, disadvantages and scope.

Chapter 3 defines the problem statement and a brief overview of how the problem is solved.

Chapter 4 explains the proposed approach and its implementation.

Chapter 5 gives the experimental results of implementing the proposed approach on a software system.

Chapter 6 concludes the thesis and gives directions for future work.

Chapter 2
Literature Survey

This chapter gives a review of various existing metrics based techniques for code clone detection. For detection of cloned code, numerous methodologies have been proposed which are explained in the next section.

2.1 Existing Techniques of Detecting Code Clones

i. **Text Based Techniques**

Text based techniques are the earliest and provide most easy way of clone detection. In this technique, code is compared line by line and the lines are in the form of simple strings. Unlike other techniques, this technique doesn't require any type of code transformation. But some latest text based techniques transform code by removing whitespaces and comments. Since these techniques don't require any semantic or syntactic examination of code, therefore the performance is fastest in case of these techniques. This technique was proposed by Ducasse et al. [16] to compare code line by line. Johnson [17] used an efficient string matching based on fingerprints.

ii. **Token Based Techniques**

In token based techniques, code is first transformed into tokens before comparing. A lexical analyzer is used to transform the code into tokens [18]. These techniques suffer from slow performance as compare to text based techniques because a significant amount of time is consumed in tokenization. Kamiya et al. [18] presented a token based algorithm for finding clones and developed a tool called CCFinder. Li et al. [19] proposed another tool based on tokens called CP-Miner. Basit et al. [20] proposed a tool repeated tokens finder (RTF) which make use of a suffix array for finding similar tokens.

iii. **Abstract Syntax Tree (AST) Based Techniques**

In AST based techniques, source code is transformed into AST using the required language parser. Tools which are build using this technique consumes a lot of time when they are applied on a very large source code. In this technique, code clones are identified by finding similar subtrees from the

AST. Jiang et al. [21] used an algorithm based on subtrees characterization using numerical vectors. Yang [22] gave a technique that syntactically differentiates two versions of the same program. Gitchell et al. [23] proposed a tool Sim which transforms code into trees. Wahler et al. [24] find parameterized and exact clones by transforming source code into XML form.

iv. Program Dependence Graph (PDG) Based Techniques

A PDG is a combination of a data flow graph and control flow graph [25]. In this technique the source code is transformed into its PDG. A PDG carries the semantic information of code. PDG based techniques can precisely find near miss clones. After getting the PDG, similar sub graphs are detected to get clones. Komondoor et al. [26] proposed an approach that uses PDGs and program slicing to find clones. This approach finds duplicate code fragments in C programs. Liu et al. [27] proposed a plagiarism detection algorithm and a tool Gplag to detect clones.

v. Metrics Based Techniques

In metric based techniques the code is not directly compared but different metrics of code are gathered and these metrics are compared to detect clones. Today most of the clone detection tools use these techniques to find clones. In comparison to other techniques, these techniques are easy to use, are less complex and less time consuming. The proposed approach is also based on these techniques to find clones. In this paper, the focus is on metric based techniques [28] [29] [30] [31] [32] [33] [34]. The next section gives a literature survey of metrics based techniques of detecting clones.

vi. Hybrid Techniques

In hybrid techniques, two or more techniques are combined to detect clones. Koschke et al. [35] proposed an approach that compares the tokens of the AST nodes rather than making a direct comparison between AST nodes. Leitao [36] proposed a hybrid technique that uses a combination of semantic technique (using call graphs) and syntactic technique based on AST metrics. Taurus et al. [37] proposed a method to find functional clones from a software. This method combines suffix tree and AST based techniques.

Table 2.1 gives information regarding the portability, efficiency and integrality of various clone detection techniques.

Table 2.1 Clone Detection Techniques

Technique	**Portability**	**Efficiency**	**Integrality**
Text Based	High	High	Low
Token Based	Medium	Low	High
AST Based	Low	High	Low
PDG Based	Low	High	Medium
Metric Based	Depend on metrics used	High	Medium

2.2 Literature Survey of Metrics Based Clone Detection Techniques

A number of metric based techniques of clone detection have been proposed. In these techniques the values for different metrics of the source code are calculated and then these values are compared to get the clone pairs or clone classes. In some of the metrics based techniques, the source code is first converted into an intermediate representation like AST or PDG or any other representation and then metrics are computed from the intermediate representation and in some cases metrics are computed directly from source code. Figure 2.1 shows a generic flow chart of metric based techniques.

Mayrand et al. [28] proposed a technique for identifying function level clones from large software systems. The tool DatrixTM which is a source code analyzer tool set was used for extracting a set of metrics from the source code. A set of 21 function metrics were used. Source code was first converted to AST form and then AST got converted into an intermediate representation i.e. control flow graph and data flow graph. Metrics were then calculated from the intermediate representation. The detected clones lie on a cloning scale that ranges from exact copy to distinct clones. The cloning scale is shown in figure 2.2. This technique can be helpful to improve maintainability of large softwares and results in negligible number of false positives.

But this technique works only for procedural languages. So, it can be extended so that it can be implemented to find clones from large object oriented softwares.

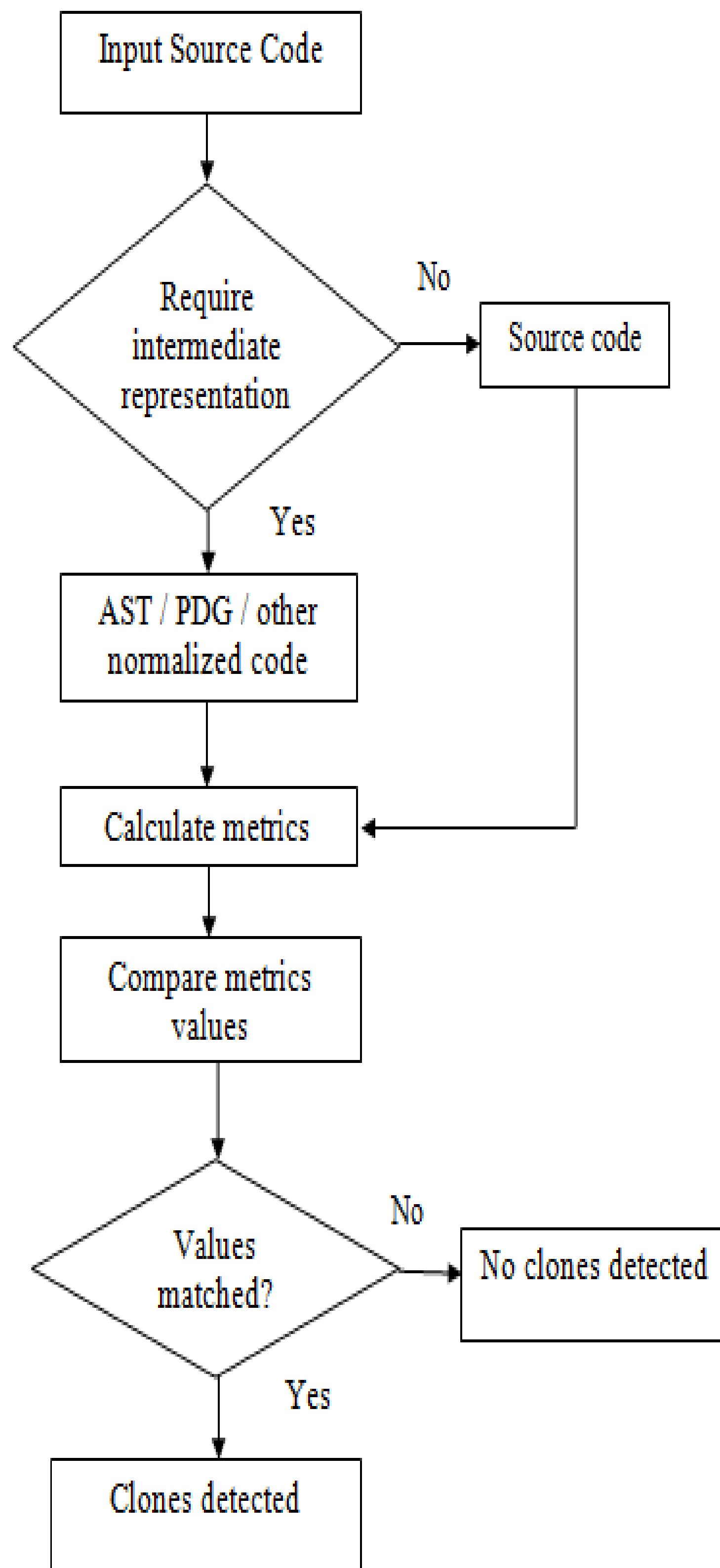


Figure 2.1: Generic Flowchart of Metric Based Techniques.

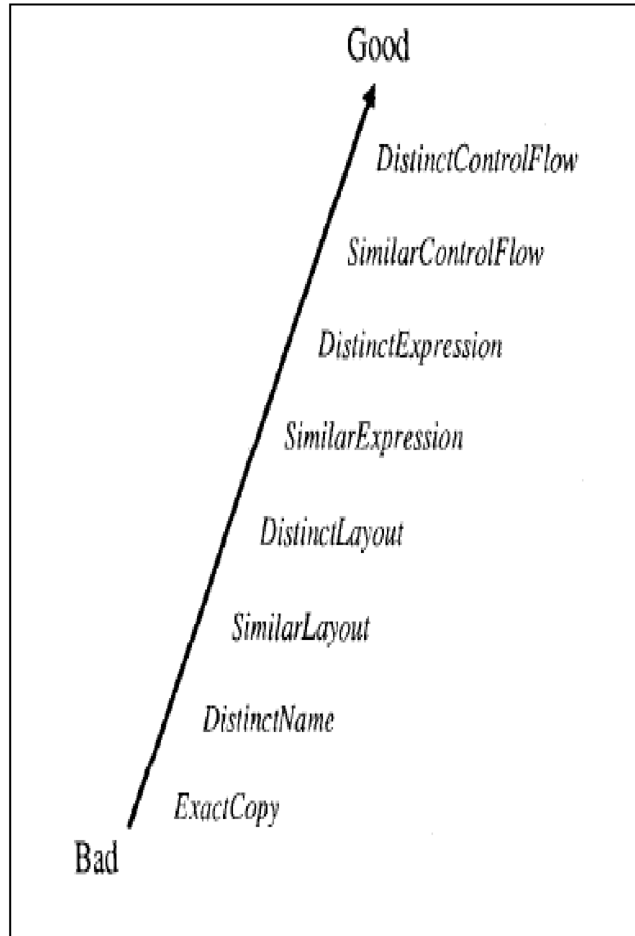


Figure 2.2: Cloning Scale [28].

Kontogiannis [29] performed experiments with some program features to get a set of five standard metrics that can be used to detect code clones. In the proposed approach, the code was first represented as AST and then some program features got computed from the AST representation. Using these features, a set of five standard metrics was computed. A 5-D space was created using the computed metrics. Euclidean distance between the points plotted on the space was used to detect code clones. The advantage of using this approach is that it is fast and easy to use. It can even detect clones if some local changes like change in variables name are done in the code. In future, the performance of the approach can be increased by computing hash values from the metrics to reduce the number of clone candidates.

Patenaude et al. [30] proposed a technique to identify code clones from java coded systems. This technique extended the tool DatrixTM to evaluate java source code. Six java specific method metrics were used to find cloned methods as shown in table 2.2. Methods with similar metrics values were reported as clones. A limitation of this

technique is that it may give false positives. So, manual verification is required to confirm whether the detected clones are actually clones or not.

Table 2.2 Method Level Metrics [30]

Abbreviation	Description
MetCOut	Number of calls from a method (Output).
MetStmt	Number of statements contained by a method.
MetMcC	McCabe's cyclomatic complexity.
MetPar	Number of parameters.
MetNLUD	Number of use-definition of non-local variables.
MetVar	Number of local variables.

Li et al. [31] gave a metric space based technique for detecting code clones. A metric space is a set containing the value of euclidean distance between the elements of the set. In this technique the source code was first converted to the elements of the metric space. The distance between elements of the space was used to detect code clones. The lesser is the distance between the elements, the more similar the elements are. After defining the metric space, a proximity query approach was used to get the clone members which are near to each other. Nearest neighbor query was used as proximity query. This technique was applied to a real industry project. This technique is an efficient and accurate clone detection technique. The disadvantage is that this technique is semi-manual and the results need to be verified and reviewed by experts. In future, the verification check can be made automatic.

Kodhai et al. [32] proposed a metric-based technique combined with text based comparison of the source code for the detection of function level clones from C language code. In this technique, the source code was first parsed into a standard format using a hand coded parser as shown in figure 2.3. Then methods were

identified and metrics values were computed for the methods and were compared to get clone candidates. Then potential clones were detected using a text based comparison of the standardized parsed code. The common candidates in both the cases were identified as cloned methods. As compare to the other metric based techniques, this technique is considered to be the less complex and provides more accurate and efficient way of detecting clones. In future, the technique can be extended to find clones from source code written in other programming languages.

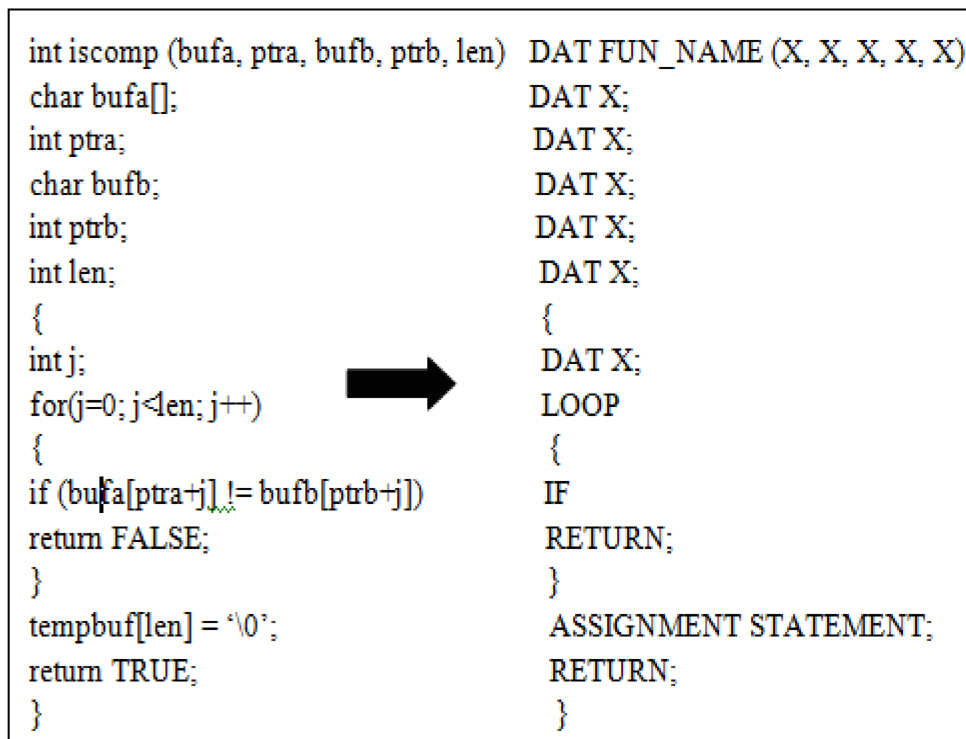


Figure 2.3: Parsing of a Sample Method [32].

Abd-El-Hafiz et al. [33] proposed a metric based data mining technique for detecting clones. Firstly, different functional metrics were computed for the source code. Then a data mining fractal clustering algorithm was used to divide the code into small number of clusters. In a metric space, a cluster comprises of those functions which are within a specific distance from each other. The last phase is the extraction of clone classes from the clusters. The technique is a space efficient technique for big softwares and it varies linearly with the size of data. In future, this technique can be experimented with different programming languages.

Shawky et al. [34] proposed a technique that applied sequential clustering to detect code clones. The proposed technique evaluated some of the metrics that are used for

detecting clones and generated a sequence of metrics that resulted in 100% precision. This technique can be used as a preliminary stage in metrics based clone detection. But the limitation of this technique is that it detects an optimal metrics sequence only for two systems under experiment. So in future, a metrics set can be generated that will be good for all systems.

Table 2.3 wraps up the scope and intermediate representation used in various metric based techniques.

Table 2.3 Summary of Comparison between Existing Metric Based Approaches

Author	**Scope**	**Intermediate representation**	**Reference**
Mayrand et al.	Procedural languages	AST, control flow graph and data flow graph	[28]
Kontogiannis	Procedural languages	AST	[29]
Patenaude et al.	Java	Not required	[30]
Li et al.	Language independent	Vector space representation	[31]
Kodhai et al.	C language	Standardized parsed code	[32]
Abd-El-Hafiz et al.	C language	Not required	[33]
Shawky et al.	C language	Not required	[34]

2.3 Properties of Code Clone Detection Techniques

All the code clone detection techniques are composed of a number of properties. These properties are helpful in understanding these techniques more clearly. An overview of these properties along with their meaning is given in table 2.4 [7].

Table 2.4 Properties of Code Clone Detection Techniques

Property	**Detail**
Source Transformation	It is defined as the transformation or filtering applied by each technique before performing the actual comparison. For ex. some techniques perform whitespace and comment removal before applying the comparison algorithm.
Comparison Granularity	Each of the techniques works on different levels of code granularity. Some of the techniques work on one LOC while some work on functions as code granularity.
Comparison Algorithm	This property means that which comparison algorithm is used by each technique to identify code clones.
Computational Complexity	The computational complexity of a technique is directly proportional to the comparison algorithm used.
Clone Similarity	This property means that which type of clones can be detected by each technique. For ex. some of the techniques can detect only type 1 code clones while others can detect type 2 clones.
Clone Refactoring	This property indicates if the clone detection technique supports clone refactoring or not.
Output	It means whether the clone detection technique returns clone pairs or clone classes as output.

Language Paradigm	This property means that for which language a particular technique works for. For ex. some clone detection techniques can detect clones from java language software systems only while some can detect clones from C language software systems.

Chapter 3
Problem Statement

There are numerous techniques for detecting code clones which are discussed in the literature survey. Each technique has its own benefits and limitations.

While the text based techniques provide the easiest way of detecting code clones but they can detect only type-1 clones. Though the token based techniques can detect both type-1 and type-2 clones but these techniques need a lexical analyzer to transform the code into tokens. A significant amount of time is consumed in tokenization. In AST based techniques, it is required to parse the source code which is a time and space consuming process. PDG based techniques can find near miss clones but these techniques take a huge amount of time and are very complex. To convert a program into its PDG representation, both its data flow graph and control flow graph are required. Metrics based techniques are least complex because they only require comparison of some numerical data i.e. metrics values of program units to find code clones. But these techniques may give false positives and result in less precision value.

3.1 Gap Analysis

i. An efficient and less complex approach for code clone detection is required.

ii. A metrics based approach is required that can work for all programming languages and that can be implemented to find clones from large object oriented softwares.

iii. An approach is required that can evaluate various metrics in such a way that these metrics should result in higher precision and recall values and less number of false positives when used with metrics based techniques of code clone detection.

iv. A disadvantage of most of the metrics based approaches is that they are semi-manual and the results need to be verified and reviewed. So, the verification check can be made automatic.

v. Various tools are required that can compute the values of required metrics for implementing the approach and that can export the results in such a way that the results can be stored into the database and can be used further.

3.2 Objective

The objective of the thesis is to solve a major problem of metrics based code clone detection techniques i.e. less precision value.

To solve the above stated problem, an approach is proposed for selecting a set of relevant metrics to be used in detection of cloned code. The metrics are relevant in the sense that using these metrics in metrics based code clone detection will result in higher precision and recall values. The proposed approach evaluates a set of independent metrics i.e. there is no correlation between the metrics which are evaluated. Metrics are evaluated on the basis of precision and recall values starting from all combinations of one metric and then gradually increasing the number of metrics in the metrics combinations until the complete set of metrics involved in the approach is used. The combination that results in highest precision and recall values is recognized as a set of relevant metrics for code clone detection.

Chapter 4

Proposed Approach and Implementation

The current work presents an approach to select a set of relevant metrics for metrics based techniques of code clone detection. From a large number of metrics given in the literature of code clone detection, a set of metrics is evaluated such that the metrics in the set are independent of each other i.e. there is no correlation between the metrics and also these metrics generated good results in case of code clone detection. The reason for taking independent metrics is to reduce the comparison overhead and to make the approach computationally efficient. To compute the values of required metrics for implementing the approach, two tools [38] [39] which are available for finding software metrics are used. These tools are able to export the metrics values as Comma separated values (CSV) file. Metrics are evaluated on the basis of precision and recall starting from all combinations of one metric and then gradually increasing the number of metrics in the metrics combinations until the complete set of metrics involved in the approach is used. The proposed approach overcomes a major limitation of metrics based techniques of code clone detection i.e. less precision value.

4.1 Metrics Used in the Approach

Software metrics give the numerical values of some characteristics of software or units of software. In the proposed approach, those metrics are required that measures functions as software units.

There exists a large number of metrics in the clone detection literature. But only a set of six metrics is chosen among those metrics. The reason for selecting these metrics is that they are independent of each other i.e. there is no correlation between the metrics and also these metrics generated good results in case of code clone detection. One new metric used is CountPath which results in increasing the precision to a great extent for the experiment conducted. Thus a total of seven metrics are evaluated in the proposed approach.

Table 4.1 gives the name, detail of each metric and number assigned to each metric which is used in the approach to refer these metrics.

Table 4.1 Metrics Name and Meaning

Metric Number	Metric Name	Metric Meaning
1	Complexity	Number of decision points + 1 or Edges – Nodes + Connected components
2	Depth	The maximum level of nesting of control constructs (for, while, switch, if etc.) found within each function.
3	CountInput	The number of inputs a function uses plus the number of unique subprograms calling the function i.e. Functions calledby + Parameters read + Variables read.
4	CountOutput	The number of outputs that are set i.e. Functions calls + Parameters set/change + Variables set/change
5	CountPath	Number of unique paths through a function.
6	CountStmtDecl	Number of declarative statements
7	CountStmtExe	Number of executable statements.

4.2 Proposed Approach

From the metrics explained in the above table, we started from applying a single metric which resulted in 7 different combinations i.e. (1), (2), (3), (4), (5), (6), (7). For each of these combinations, following steps are followed:

i. Metrics Calculation and CSV File Generation

The metrics specified within each combination are calculated using two tools. Six metrics i.e. Complexity, CountInput, CountOutput, CountPath, CountStmtDecl and CountStmtExe are calculated using a tool Understand [38] which is a tool available for metrics calculation. The remaining one metric i.e. Depth is calculated using a tool SourceMonitor [39]. These tools are able to export the metrics values as CSV file. Then the result contained in the CSV files is stored into the database.

ii. Generate Clone Classes and Clone Pairs

For each combination, clone classes and clone pairs are generated by applying comparison algorithm. Using this approach, type-1 and type-2 categories of clones can be detected.

4.2.1 Comparison Algorithm to Detect Clone Classes

The different notations used in the algorithm are:

N: Total number of functions in the software system from which clones need to be find out.

F: Set of all functions of the software system s.t. $F= \{f_1, f_{2,} ----, f_N\}$.

f_i: i^{th} function of the software system where i=1 to N.

M_k: Metric combination for which clone classes need to be detected.

m_{rk}: r^{th} metric in metric combination M_k s.t. $M_k= (m_{1k}, m_{2k,} ----, m_{rk})$.

$f_i.m_{rk}$: Value of metric m_{rk} for function $f_{i.}$

C: Clone class repository in which all the detected clone classes are stored.

Following are the steps of the comparison algorithm:

Input: $\forall$ $f_i \in F$ input values of $m_{1k}, m_{2k,} ---, m_{rk}$.

Output: Clone classes $c_1, c_2,$ ------, c_n for $M_{k.}$

1. C ← null
2. **for** i = 1 until N
3. {
4.   **if** ($f_i \notin c_i$) $\forall$ $c_i \in C$ **then**
5. c_i ← $\{f_i\}$
6.   **end if**
7.   **for** j = i+1 until N
8.   {
9.   **if** ($f_im_{1k} == f_jm_{1k}$ && $f_im_{2k} == f_jm_{2k}$ && ... $f_im_{rk} == f_jm_{rk}$) **then**
10. c_i ← $\{f_j\}$

11. **end if**

12. }

13. C ← c_i

14. }

iii. Calculate Precision and Recall

In this step, the number of clones generated in the above step is compared with the results available in an online repository of clone detection techniques [40] for calculating the precision and recall values for each combination of metrics. Precision is calculated as the fraction of detected clones that are correct. To calculate recall, the fraction of correct clones that are detected is taken.

For each combination (C) of metrics, precision and recall is calculated as:

N: Total number of clones identified by using metric combination C.

M: Number of correctly identified clones by metric combination C.

Q: Number of false positives detected by using metric combination C

s.t. N = M + Q.

S: Actual number of clones found from the code clone detection repository.

Now for metric combination C, precision (P) and recall (R) is calculated as:

$$P = M / N$$

$$R = M / S$$

iv. Precision and Recall Comparison

In order to determine which metric combination is best among all combinations, precision and recall values for all combinations were compared with each other. The combination which resulted in highest precision and recall is saved. Now keeping the metrics in the saved combination as fixed, other combinations are generated by adding one more metric to the saved combination and repeat step (i) to (iv) for these combinations. Keep on repeating steps (i) to (iv) until a combination containing all seven metrics is achieved.

v. Select Set of Relevant Metrics

After performing above steps, different combinations with their precision and recall values are achieved. Among these combinations, that combination which resulted in maximum precision is selected as relevant set of metrics. In case if

more than one combinations result in same precision value then decision is made on the basis of higher recall value.

4.3 Flow Diagram of Proposed Approach

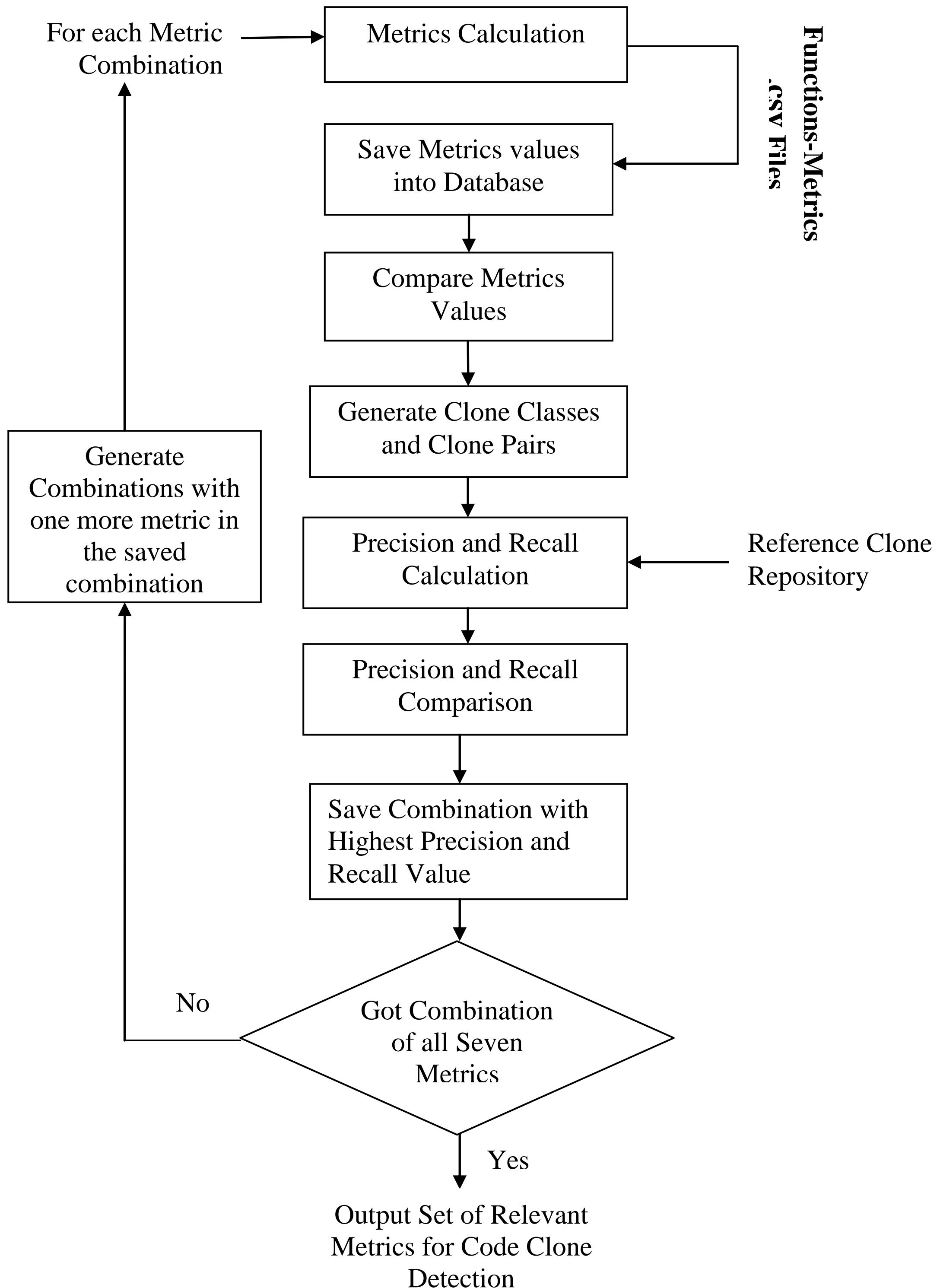


Figure 4.1: Steps Followed in the Proposed Approach.

4.4 Implementation

The proposed approach is implemented in PHP. Figure 4.2 shows the start up page of the clone detector. To get the clone classes for each combination of metrics, the name of the CSV file that contains the details of the metrics is entered into the textbox shown in figure 4.2.

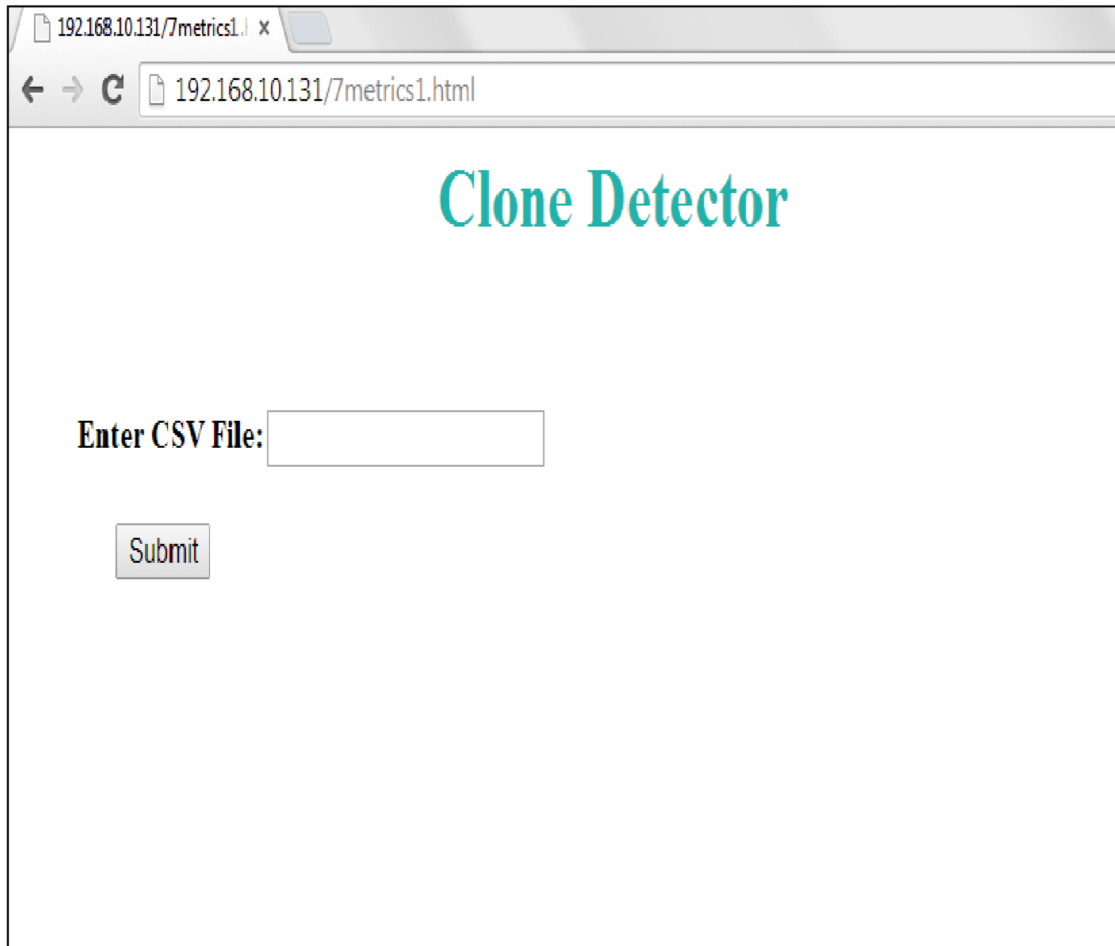


Figure 4.2: Startup Page.

To compute the required metrics values and to generate the results as a CSV file, two tools are used. Six metrics i.e. Complexity, CountInput, CountOutput, CountPath, CountStmtDecl and CountStmtExe are calculated using a tool Understand [38] which is a tool available for metrics calculation. The remaining one metric i.e. Depth is calculated using a tool SourceMonitor [39]. After computing the required metrics, the result is exported as CSV file. In figure 4.3, Understand tool is used and here that directory is added which contains the source code files of the software system for which the metrics values need to be calculated.

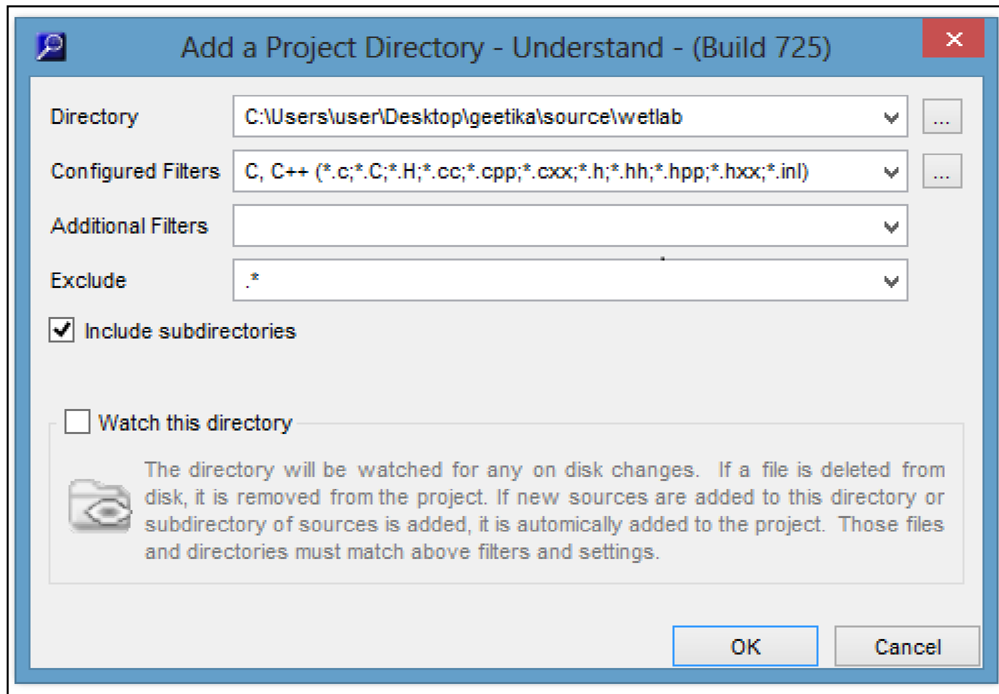


Figure 4.3: Add Project Directory.

In figure 4.4, the metrics which are required to be calculated are exported as a CSV file.

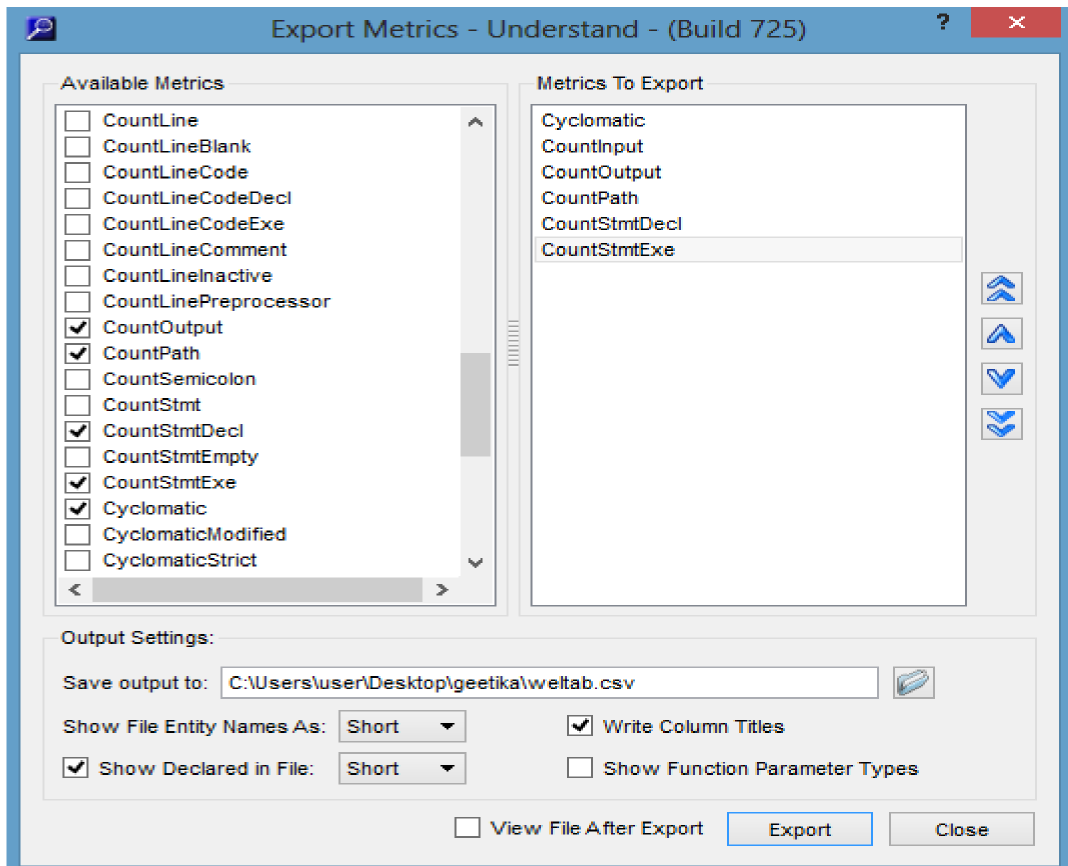


Figure 4.4: Export Metrics Detail as CSV File.

```
Kind,Name,File,Cyclomatic,CountInput,CountOutput,CountPath,CountS
tmtDecl,CountStmtExe
Function,"allowcard",ofca.c,22,11,8,137216,1,59
Function,"askchange",vfix.c,14,6,8,424,2,37
Function,"askchange",vedt.c,14,6,8,424,2,37
Function,"askchange",xfix.c,14,6,8,424,2,37
File,"baselib.c",baselib.c,,,,,57,202
Function,"blkbuf",baselib.c,2,62,0,2,1,5
File,"cand.c",cand.c,,,,,14,58
Function,"candhead",cand.c,1,7,4,1,2,15
File,"candnm.h",candnm.h,,,,,1,0
Function,"candtail",cand.c,1,6,4,1,2,15
File,"canv.c",canv.c,,,,,37,347
Function,"canvw",cnv1a.c,7,19,7,64,0,40
Function,"canvw",canv.c,7,19,7,64,0,40
Function,"canvw",cnv1.c,7,19,7,64,0,40
File,"cardrd.h",cardrd.h,,,,,1,0
File,"ccibm.c",ccibm.c,,,,,4,51
File,"ccibmx.c",ccibmx.c,,,,,4,30
File,"ccibmxp.c",ccibmxp.c,,,,,4,31
Function,"chead",cumt.c,1,9,7,1,2,23
Function,"chead",totl.c,1,9,6,1,1,22
Function,"cnread",wellib.c,4,6,6,6,2,17
File,"cnv1.c",cnv1.c,,,,,31,303
File,"cnv1a.c",cnv1a.c,,,,,23,305
Function,"confrm",wellib.c,8,9,5,40,2,18
Function,"cprint",wellib.c,2,31,4,2,1,8
File,"cprint.c",cprint.c,,,,,3,75
Function,"cread",wellib.c,4,5,6,6,2,19
Function,"ctail",cumt.c,1,8,6,1,1,21
```

Figure 4.5: CSV File of Metrics Detail.

Now to compute the Depth metric, SourceMonitor tool is used. In figure 4.6, that file is added which contains the source code of project for which the depth metric value need to be computed.

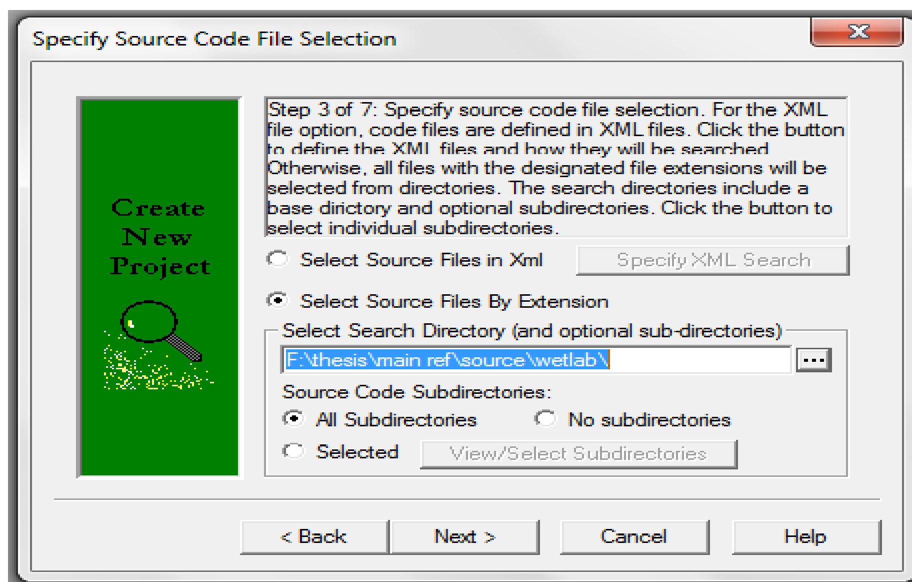


Figure 4.6: Specify the Source Code File Selection Directory.

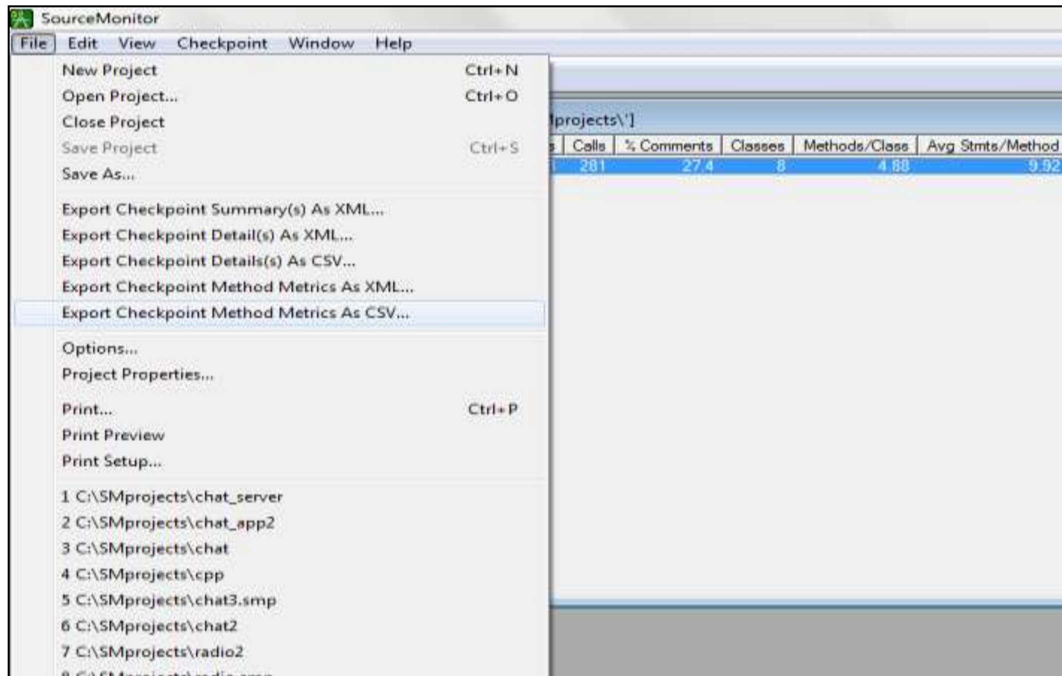


Figure 4.7: Export Metric as CSV File.

Now the detail of all the metrics is obtained in the form of CSV files. For each metrics combination, the CSV file is entered as input to the clone detector as shown in figure 4.8.

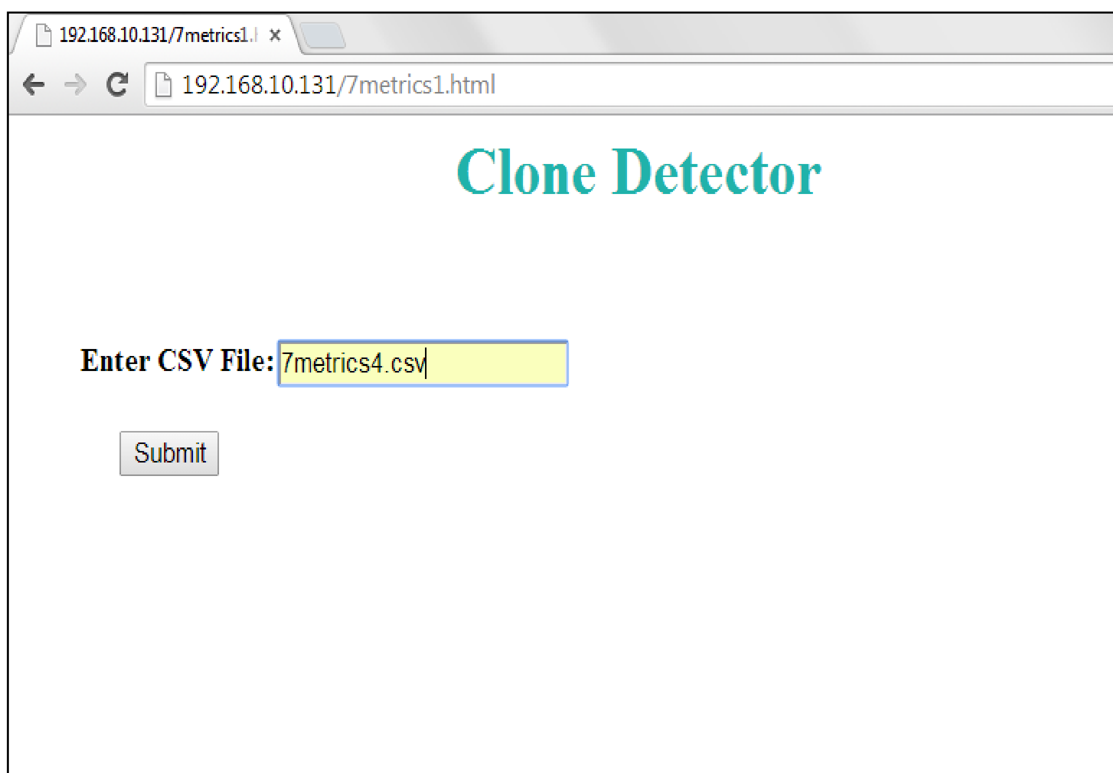


Figure 4.8: Enter CSV File as Input.

When the submit button is clicked in figure 4.8, the metrics values get stored into the database as shown in figure 4.9.

192.168.10.131/7metrics2.p ×

192.168.10.131/7metrics2.php

id	Kind	Name	File	Depth	CountInput	CountOutput	Cyclomatic	CountStmtDec	CountStmtExe	CountPath
1	Function	"allowcard"	ofca.c	3	11	8	22	1	59	137216
2	Function	"askchange"	vfix.c	3	6	8	14	2	37	424
3	Function	"askchange"	xfix.c	3	6	8	14	2	37	424
4	Function	"askchange"	vedt.c	3	6	8	14	2	37	424
5	File	"baselib.c"	baselib.c	3				57	202	
6	Function	"blkbuf"	baselib.c	1	62	0	2	1	5	2
7	File	"cand.c"	cand.c	1				14	58	
8	Function	"candhead"	cand.c	0	7	4	1	2	15	1
9	File	"candnm.h"	candnm.h	0				1	0	
10	Function	"candtail"	cand.c	0	6	4	1	2	15	1
11	File	"canv.c"	canv.c	4				37	347	
12	Function	"canvw"	cnv1.c	1	19	7	7	0	40	64
13	Function	"canvw"	canv.c	1	19	7	7	0	40	64
14	Function	"canvw"	cnv1a.c	1	19	7	7	0	40	64
15	File	"cardrd.h"	cardrd.h	0				1	0	
16	File	"ccibm.c"	ccibm.c	3				4	51	
17	File	"ccibmx.c"	ccibmx.c	3				4	30	
18	File	"ccibmxp.c"	ccibmxp.c	3				4	31	
19	Function	"chead"	cumt.c	0	9	7	1	2	23	1
20	Function	"chead"	totl.c	0	9	6	1	1	22	1
21	Function	"cnread"	wellib.c	1	6	6	4	2	17	6
22	File	"cnv1.c"	cnv1.c	4				31	303	
23	File	"cnv1a.c"	cnv1a.c	4				23	305	
24	Function	"confrm"	wellib.c	1	9	5	8	2	18	40
165	File	"vset.c"	vset.c	1				8	48	
166	File	"vtot.c"	vtot.c	3				11	91	
167	Function	"welcom"	wellib.c	1	35	6	2	0	11	2
168	File	"wellib.c"	wellib.c	3				35	344	
169	File	"weltab.h"	weltab.h	0				7	0	
170	Function	"whoentrer"	spol.c	2	4	4	7	1	19	48
171	Function	"whoentrer"	poll.c	2	4	4	7	1	19	48
172	Function	"wtcand"	wellib.c	1	15	5	3	0	10	4
173	Function	"wtdcnd"	wellib.c	1	8	5	3	0	9	4
174	Function	"wtdual"	wellib.c	1	8	5	3	0	9	4
175	Function	"wtoffi"	wellib.c	1	8	5	3	0	9	4
176	File	"xfix.c"	xfix.c	4				16	355	

Detect_Clones

Figure 4.9: Metrics Values Stored into Database.

When the Detect_Clones button is clicked in figure 4.9, the comparison algorithm explained in section 4.2.1 is applied on the metrics values and the clone classes are generated for the given metrics as shown in figure 4.10.

192.168.10.131/7metrics3.php

CLONE_CLASS 1

1:

Function = "askchange"
IN FILE vfix.c

2:

Function = "askchange"
IN FILE xfix.c

3:

Function = "askchange"
IN FILE vedt.c

CLONE_CLASS 2

1:

Function = "canvw"
IN FILE cnv1.c

2:

Function = "canvw"
IN FILE canv.c

3:

Function = "canvw"
IN FILE cnv1a.c

CLONE_CLASS 3

1:

Function = "gtdcnd"

Figure 4.10: Resulting Clone Classes.

After getting the clone classes for each combination of metrics, precision and recall are calculated as explained in point iii of section 4.2.

Chapter 5

Experimental Results

The proposed approach is implemented on weltab [40] which is a medium sized software system and is described in table 5.1.

Table 5.1 Description of Case Used

Case	Language	LOC	Functions	Description
Weltab	C	11K	123	Vote tabulation system

Table 5.2 shows the highest values of precision and recall for each case when we started from combinations of only one metric to a combination containing all seven metrics.

Table 5.2 Highest Precision and Recall Values

Number of Metrics used	Metrics Combinations	Precision	Recall	Number of Detected Clone Classes
One	(1)	.17	1	26
Two	(1, 3)	.23	1	24
Three	(1, 3, 6)	.5	.8	22
Four	(1, 3, 6, 7)	.66	.75	21
Five	(1, 3, 6, 7, 4)	.8	.7	19
Six	(1, 3, 6, 7, 4, 5)	1	.7	16
Seven	(1, 3, 6, 7, 4, 5, 2)	1	.25	14

Figure 5.1 and figure 5.2 shows the resulting clone classes for the best set of metrics.

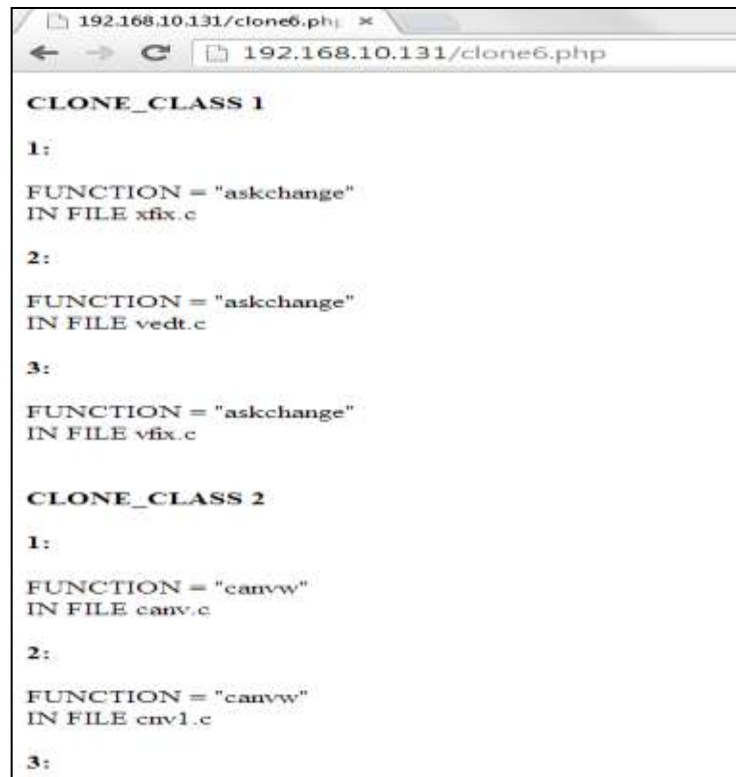


Figure 5.1: Resulting Clone Classes for Best Set of Metrics.

Figure 5.2: Resulting Clone Classes for Best Set of Metrics.

When clones are detected using combinations of only one metric, the highest retrieved precision is 17% and recall is 100%. These values are obtained when metric number one i.e. complexity is used. When a combination of two metrics is used, the highest precision and recall is 23% and 100% respectively in case of combination (1, 3). From table 5.2, it is analysed that the value of highest precision increases with increase in number of metrics but this case is valid only upto six metrics. When seven metrics were used, then the value of precision remained unaffected as shown in figure 5.3. The best combination is (1, 3, 7, 6, 4, 5) which obtained a precision of 100% with recall of 70%. When the metric numbered two i.e. Depth is added to this combination, it resulted in same precision value but lowers the recall value by 50%. This shows that metric Depth is less appropriate while detecting clones from weltab. So the combination (1, 3, 7, 6, 4, 5) is considered as a set of relevant metrics for code clone detection. Fig. 5.4 shows the highest precision and recall values for combinations of different metrics.

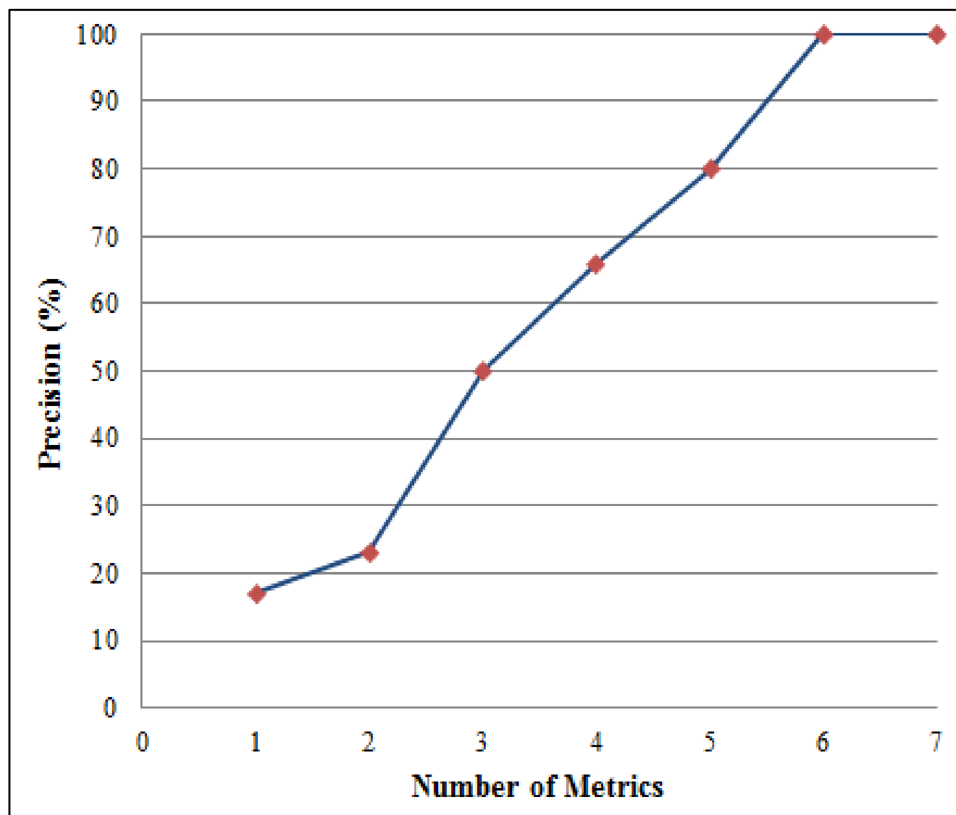


Figure 5.3: Precision Graph for Different Number of Metrics Used.

In figure 5.4, different metrics combinations are represented by english alphabets as shown in table 5.3.

Table 5.3 Representation of Metric Combinations

Alphabet	Metric Combination
A	(1)
B	(1, 3)
C	(1, 3, 6)
D	(1, 3, 6, 7)
E	(1, 3, 6, 7, 4)
F	(1, 3, 6, 7, 4, 5)
G	(1, 3, 6, 7, 4, 5, 2)

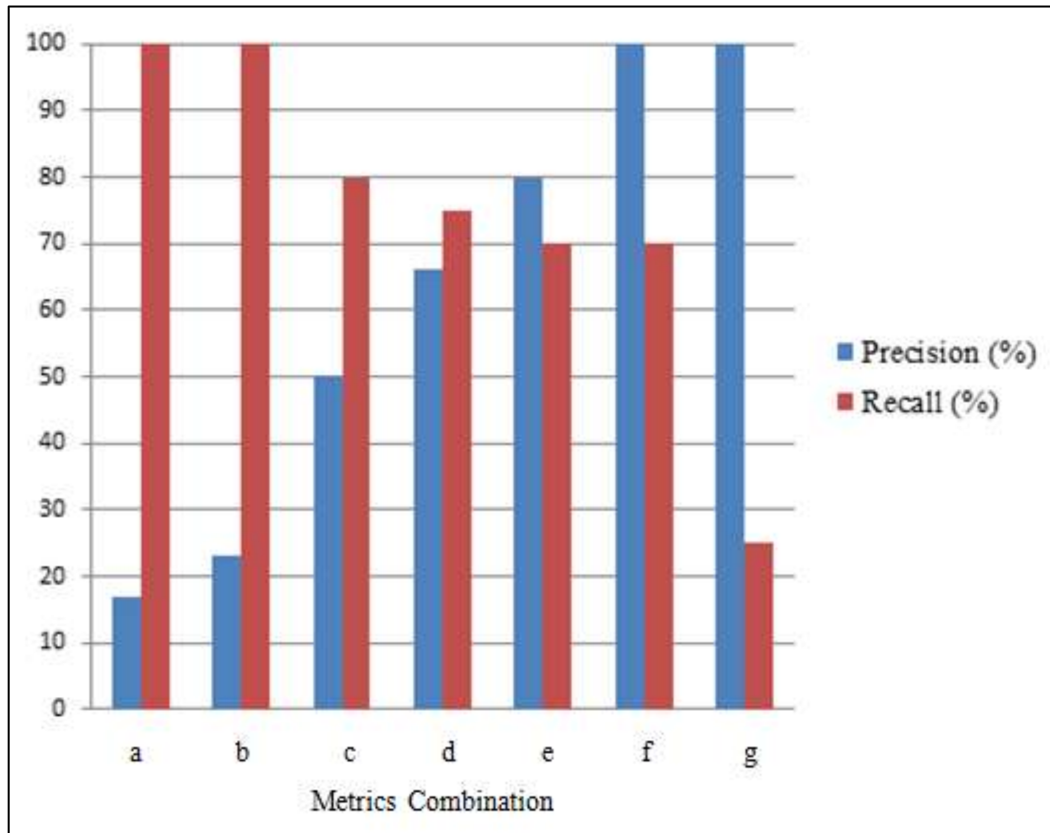


Figure 5.4: Highest Precision and Recall Values for Different Metric Combinations.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The proposed approach evaluates a set of software metrics for code clone detection. In the proposed approach, combinations of metrics are evaluated on the basis of the precision and recall values obtained when these metrics are used in metrics based code clone detection. A new metric i.e. CountPath is used in the proposed approach. The result of implementing the proposed approach on a C language software system have shown that the use of the metric CountPath have increased the precision value to a great extent. The proposed approach is computationally efficient and overcomes a major limitation of metrics based techniques of code clone detection i.e. less precision. Using the proposed approach, type-1 and type-2 categories of code clones can be detected.

6.2 Future Scope

- In future some other set of metrics can be used to perform the proposed approach which can achieve a higher value of recall.
- The proposed approach can be extended so that it can detect type-3 and type-4 categories of code clones.
- A redesigning approach can be applied after detection of cloned code using the proposed approach.
- The proposed approach can be applied to larger and object oriented software systems.

References

[1] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," *in Proceedings of the 15th Working Conference on Reverse Engineering*, Antwerp, pp. 81-90, 2008.

[2] Z. Jiang and A. Hassan, "A Framework for Studying Clones in Large Software Systems," *in Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, Paris, pp. 203-212, 2007.

[3] Y. Jia and M. Harman, "Clone Detection Using Dependence Analysis and Lexical Analysis," Ph.D. dissertation, Dept. of Computer Science, King's College, London, 2007.

[4] C. Kapser and M. W. Godfrey, " "Cloning Considered Harmful" Considered Harmful," *in Proceedings of the 13th Working Conference on Reverse Engineering*, USA, pp. 19-28, 2006.

[5] G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. D. Penta, "Identifying Clones in the Linux Kernal," *in Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, Italy, pp. 90-97, 2001.

[6] J. H. Johnson, "Substring Matching for Clone Detection and Change Tracking," *in Proceedings of the 10th International Conference on Software Maintenance*, Canada, pp. 120-126, 1994.

[7] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Queen's University at Kingston, Ontario, Tech. Rep. TR 2007-541, Sep. 2007.

[8] R. Al-Ekram, C. Kapser and M. Godfrey, "Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems," *IEEE International Symposium on Empirical Software Engineering*, Australia, pp. 376-385, 2005.

[9] J. R. Cordy, "Comprehending Reality: Practical Challenges to Software Maintenance," *in Proceedings of the 11th IEEE International Workshop on Program Comprehension*, USA, pp. 196-206, 2003.

[10] B. S. Baker, "On Finding Duplication and Near Duplication in Large Software Systems," *in Proceedings of the 2nd Working Conference on Reverse Engineering*, California, pp. 86-95, 1995.

[11] J. H. Johnson, “Navigating the Textual Redundancy Web in Legacy Source,” *in Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, Canada, pp. 7-16, 1996.

[12] G. Alkhatib, “The Maintenance Problem of Application Software: An Empirical Analysis,” *Journal of Software Maintenance: Research and Practice*, vol. 4, no. 2, pp. 83-104, 1992.

[13] A. Monden, D. Nakae, T. Kamiya, S. Sato and K. Matsumoto, “Software Quality Analysis by Code Clones in Industrial Legacy Software,” *in Proceedings of the 8th IEEE International Symposium on Software Metrics*, Canada, pp. 87-94, 2002.

[14] W. K. Chen, B. Li and R. Gupta, “Code Compaction of Matching Single-Entry Multiple-Exit Regions” *in Proceedings of the 10th Annual International Static Analysis Symposium*, USA, pp. 401-417, 2003.

[15] E. Burd and M. Munro, “Investigating the Maintenance Implications of the Replication of Code,” *in Proceedings of the 13th International Conference on Software Maintenance*, Italy, pp. 322-329, 1997.

[16] S.Ducasse, M. Rieger and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” *in Proceedings of the 15th International Conference on Software Maintenance*, Oxford, pp. 109-118, 1999.

[17] J. H. Johnson, “Substring Matching for Clone Detection and Change Tracking,” *in Proceedings of the 10th International Conference on Software Maintenance*, Canada, pp. 120-126, 1994.

[18] T. Kamiya, S. Kusumoto and K. Inoue, “CCFinder: a Multilinguistic Token Based Code Clone Detection System for Large Scale Source Code,” *IEEE Transaction on Software Engineering*, vol. 28, no. 7, pp. 654-670, July 2002.

[19] Z. Li, S. Lu, S. Myangmar and Y. Zhou, “CP-Miner: Finding Copy-Paste and Related Bugs in Large Scale Software Code,” *Software Engineering, IEEE Transactions*, vol. 32, no. 3, pp. 176-192, 2006.

[20] H. A. Basit and S. Jarzabek, “Efficient Token Based Clone Detection with Flexible Tokenization,” *in Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundation of Software Engineering*, USA, pp. 513-516, 2007.

[21] L. Jiang and D. Misherghi, Z. Su and S. Glondu, “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,” *in Proceedings of the 29th International Conference on Software Engineering*, USA, pp. 96-105, 2007.

[22] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software Practice and Experience*, vol. 21, no. 7, pp. 739 – 755, 1991.

[23] D. Gitchell and N. Tran, "Sim: A Utility for Detecting Similarity in Computer Programs," *in Proceedings of the thirtieth SIGCSE Technical Symposium on Computer Science Education*, USA, pp. 266-270, 1999.

[24] V. Wahler, D. Seipel, J. Gudenberg and G. Fischer, "Clone Detection in Source Code by Frequent Itemset Techniques," *in Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, USA, pp. 128–135, 2004.

[25] D. Rattan, R. Bhatia and M. Singh, "Software Clone Detection: A Systematic Review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165-1199, July 2013.

[26] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *in Proceedings of the 8th International Symposium on Static Analysis*, France, pp. 40-56, 2001.

[27] C. Liu, C. Chen, J. Han and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," *in Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining*, USA, pp. 872-881, 2006.

[28] J. Mayrand, C. Leblanc and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System using Metrics," *in Proceedings of the 12th International Conference on Software Maintenance*, USA, pp. 244-253, 1996.

[29] K. Kontogiannis, "Evaluation Experiments on the Detection of Programming Patterns using Software Metrics," *in Proceedings of the 3rd Working Conference on Reverse Engineering*, Netherland, pp. 44-54, 1997.

[30] J. F. Patenaude, E. Merlo, M. Dagenais and B. Lague, "Extending Software Quality Assessment Techniques to Java Systems," *in Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, pp. 49-56, 1999.

[31] Z. O. Li and J. Sun, "A Metric Space Based Software Clone Detection Approach," *in Proceedings of 2nd International Conference on Software Engineering and Data Mining*, China, pp. 111-116, 2010.

[32] Kodhai, S. Kanmani, A. Kamatchi, R. Radhika and B. V. Saranya, "Detection of Type-1 and Type-2 Code Clone using Textual Analysis and Metrics," *in Proceedings of the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, Kerala, pp. 241-243, 2010.

[33] Abd-El-Hafiz and S. K, "A Metrics-Based Data Mining Approach for Software Clone Detection," *in IEEE 36th Annual Computer Software and Applications Conference,* Izmir, pp. 35-41, 2012.

[34] D. M. Shawky and A. F. Ali, "An Approach for Assessing Similarity Metrics used in Metric-Based Clone Detection Techniques," *in 3rd IEEE International Conference on Computer Science and Information Technology*, China, pp. 580-584, 2010.

[35] R. Koschke, R. Falke and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *in Proceedings of the 13th Working Conference on Reverse Engineering*, Italy, pp. 253-262, 2006.

[36] A.M. Leitao, "Detection of Redundant Code Using R^2D^2," *Software Quality Journal*, vol. 12, no. 4, pp. 361-382, 2004.

[37] R. Tairas, J. Gray, "Phoenix-Based Clone Detection Using Suffix Trees," *in Proceedings of the 44th annual Southeast regional conference*, USA, pp. 679-684, 2006.

[38] Understand, A Tool for Source Code Analysis and Metrics Calculation [Online]. Available: http://www.scitools.com.

[39] The SourceMonitor Homepage [Online]. Available: http://www.campwoodsw.com.

[40] S. Bellon and R. Koschke. Detection of Software Clone: Tool Comparison Experiment [Online]. Available: http://www.bauhaus-stuttgart.de/clones.

List of Publications

Accepted

[1] Geetika Bansal and Rajkumar Tekchandani, “Selecting a Set of Relevant Metrics for Code Clone Detection,” in 3rd International Conference on Advances in Computing, Communications and Informatics (ICACCI), IEEE, Greater Noida, Delhi, India, 2014.

[2] Geetika Bansal and Rajkumar Tekchandani, “Assessing and Comparing Metrics Based Techniques of Detecting Code Clones,” in Global Summit on Computer and Information Technology (GSCIT), IEEE, Sousse, Tunisia, 2014.