HCDETECTOR: A HYBRID APPROACH TO DETECT CODE CLONES IN JAVA PROGRAMS

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

Master of Engineering
in
Software Engineering

Submitted By
Surbhi Sonika
(801231026)

Under the supervision of:
Mr. Rajkumar Tekchandani
Assistant Professor,CSED

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, "*HCDetector: A Hybrid Approach to Detect Code Clones in Java Programs*", 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.

surbhi

(Surbhi Sonika)

801231026

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

Rajkumar

(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

I express my gratitude and appreciation to all those who have helped me throughout the duration of my research work. It would not have been possible to complete this research without guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, I would like to express my deep and sincere gratitude to my supervisor Mr. Rajkumar Tekchandani, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala. It gives me immense pleasure to pay my gratitude for his valuable advice, constructive criticism and great patience at all the times.

I owe my sincere thanks to Dr. Deepak Garg, Associate Professor and Head of Computer Science and Engineering Department, Thapar University, Patiala, for his continuous motivation, cooperation and for providing facilities.

I wish to express my sincere thanks to all the staff members and my friends who always supported and encouraged me. I was very fortunate to have an unconditional support from my family. I want to acknowledge the contributions of my parents, for their constant motivation, inspirations and for supporting me spiritually throughout my life. Last but not the least; I would like to thank God for giving me inner peace and strength.

Surbhi Sonika
(801231026)

Abstract

In software programs, substantial amount of existing code has been reused by simply copying and pasting mechanism. This process is called software cloning which leads to introduction of similar codes called code clone in system. The impact of these clones are very severe and of great concern from maintenance and development point of view. Due to this, software code clone detection is an active research area and many code clone detection techniques and tools have been proposed to detect them based on the syntactic or semantic similarity existing between code fragments.

These techniques are mainly categorized into five types. Among them text based, token based, tree based, and metrics based clone detection techniques are used for detecting syntactically similar code fragments whereas program dependence graph based technique is able to detect both syntactically and semantically similar code fragments. Moreover there are certain tools which combine these techniques in order to detect code clones more efficiently and are termed as hybrid code clone detection tools.

This thesis presents such kind of hybrid code clone detection tool called HCDetector which combines PDG based and metrics based code clone detection technique in order to detect both structural as well as behavioral similar code fragments using java byte code. This tool is language dependent and works only on java programs.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Cloning

In software development process, significant amount of existing code has been directly used by copying and pasting mechanism or by making minor modification in it. This process leads to code clones in programs and process is known as Software Cloning [1]. Software cloning also referred as duplication of code or redundant code [3] [11]. This process supports reuse mechanism and consider as a one of the form of it [2]. Moreover this duplication is not limited to code level only but it has been started from first phase of software development life cycle. If a new project has similar functionality like already existing project then developers prefer to reuse requirements, design and code to save time and to reduce development cost [6]. It seems to be beneficial from this point of view but cloning leads to adverse effect on maintainability of system and increases maintenance cost [5]. Hence software cloning does not simply indicate to copy and paste mechanism but it is a reuse process which is harmful from quality and maintenance point of view.

There are certain question arises on whether software cloning is illegal or not but there is no definite answer to this question. Copyrights and trademarks are used to avoid duplication but inadequate to fulfill the cloning criteria. As copyrights are only for copying of source code but not for functionality and trademarks mainly concerned with some features of product but not for whole product.

1.2 Code Clones

When a code is duplicated copy of another code snippet, then they are termed as code clone. The developer can use these code clones directly or by making some modification in it. The program which contains this redundant code is known as clone.

Figure 1.1 indicates code clones which may be exist within file or between two different files.

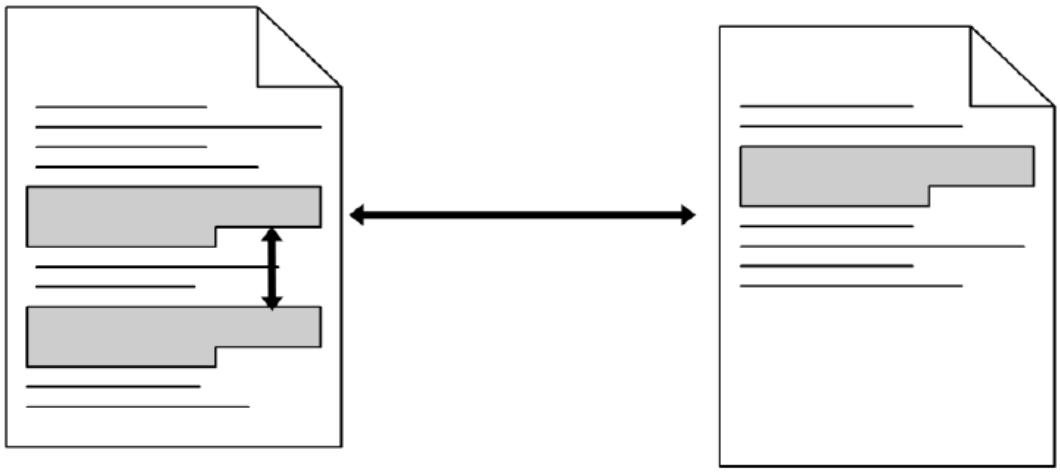


Figure 1.1: Code Clone Example.

Though, duplicate or similar code fragments are roughly known to be code clones, the definition of clone has remained more or less indefinite. This vagueness is reflected in the definition given by Ira Baxter [23], "Clones are segments of code that are similar according to some definition of similarity". This definition is relaying on how similarity is defined, and also raises question on how much of total code can be regarded as a code segment.

Mainly clones are distinguished on the basis of two types of similarity. These two are syntactic similarity and semantic similarity. If two code fragments are identical on the basis of text then they are syntactic clones while two code fragments are similar on the basis of behavior shown by them then they are semantic clones. However many definitions of code clones are available in literature according to detection technique and tool.

1.3 Code Clone Terminology

A clone detection tool always detects clones in terms of Clone pair and Clone Class. These terms helps to find similarity relation exist between different code fragments. If any sequence like same character strings, strings with no whitespace, parser generated token sequences and many more is exist between them then it means they posses clone relation and considered as code clones. The code relation is an equivalent relation in term of set theory [3]. In clone relation expressions, clones pair and clones class are defined as

follows.

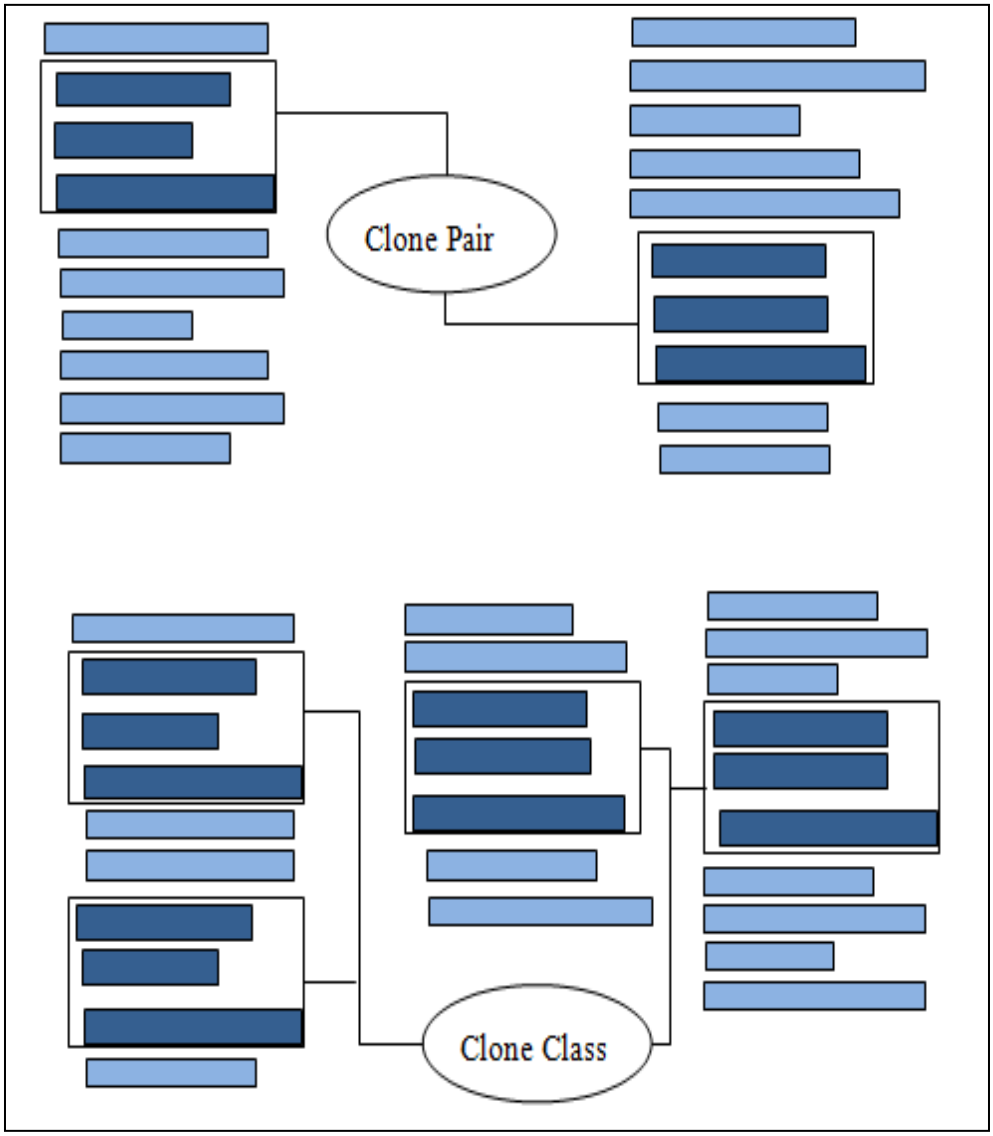


Figure 1.2: Code Clone Pair and Clone Class Example [15].

Clone Pair: If two code fragments are compared and clone relation can be found between them i.e. both are analogous to each other, then they are considering as clone pair [2]. A clone detection tool always tries to find clones with maximum possible length i.e. clone relation does not hold after this length. This is illustrated with the help of example shown in Figure 1.2. There are three code fragments in this figure each divided into several segments. When segment (a) merged with segment (b) of fragment 1 then they form maximal clone pair with combined segment (a) and segment (b) of fragment 2.

Clone Class: It is defined as set of code fragments in which clone-relation is exist between any two code fragments i.e. it is a set of all clone pairs [2]. For instance, in Figure 1.3, there is a clone class of segment (b) in fragment1, segment (b) in fragment2 and segment (a) in fragment3. Hence they form the largest set of possible clones. Although segment (b) in fragment1 and segment (b) in fragment2; segment (b) in fragment2, segment (a) in fragment3; and segment (b) in fragment1, segment (a) in fragment3 are three different clone pairs.

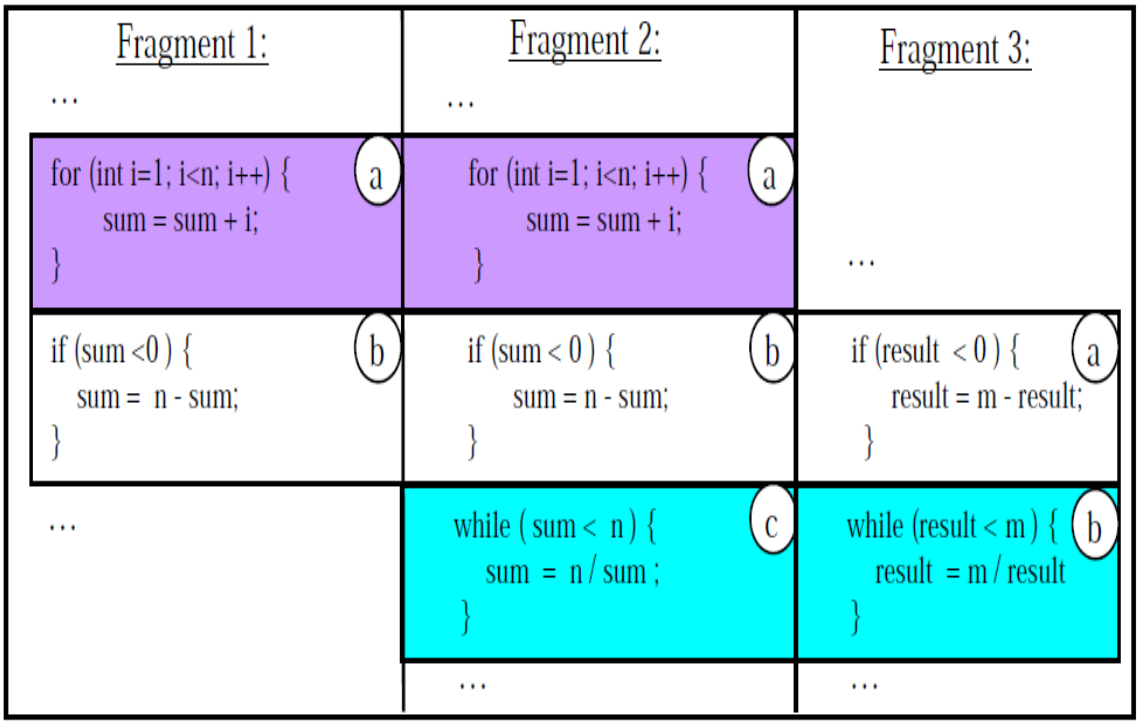


Figure 1.3: Clone Pair and Clone Class [2].

1.4 Code Clone Types

The regions of source code can be highly similar in terms of text, lexical and syntactic structure, or can be semantics, model based, and behavior. Based on this they are classified into following four types.

1.4.1 Type I (Exact Clones)

These are code fragments which are analogous to source code. Some minor modification in the presentation of code like blank spaces and comments can be allowed.

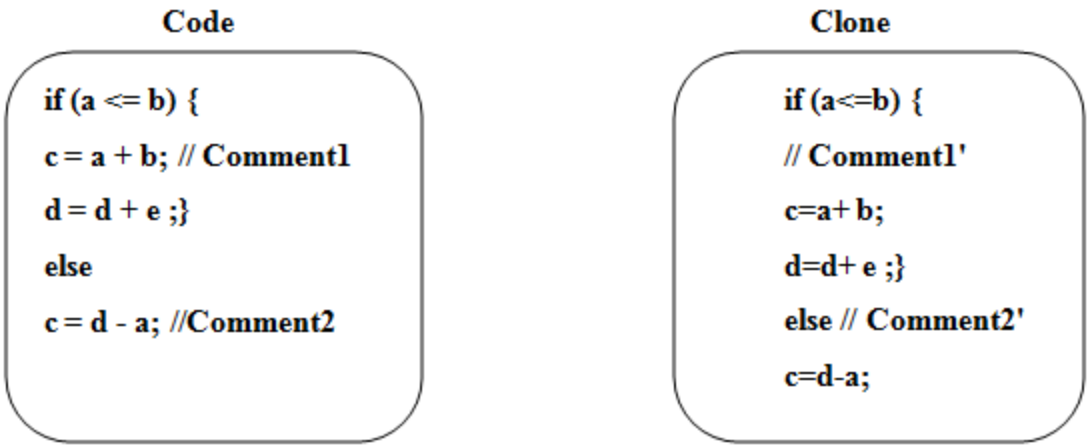


Figure 1.4: Exact Clones.

1.4.2 Type II (Renamed/Parameterized Clones)

These are structurally identical fragments except for slight renaming, layout and comments.

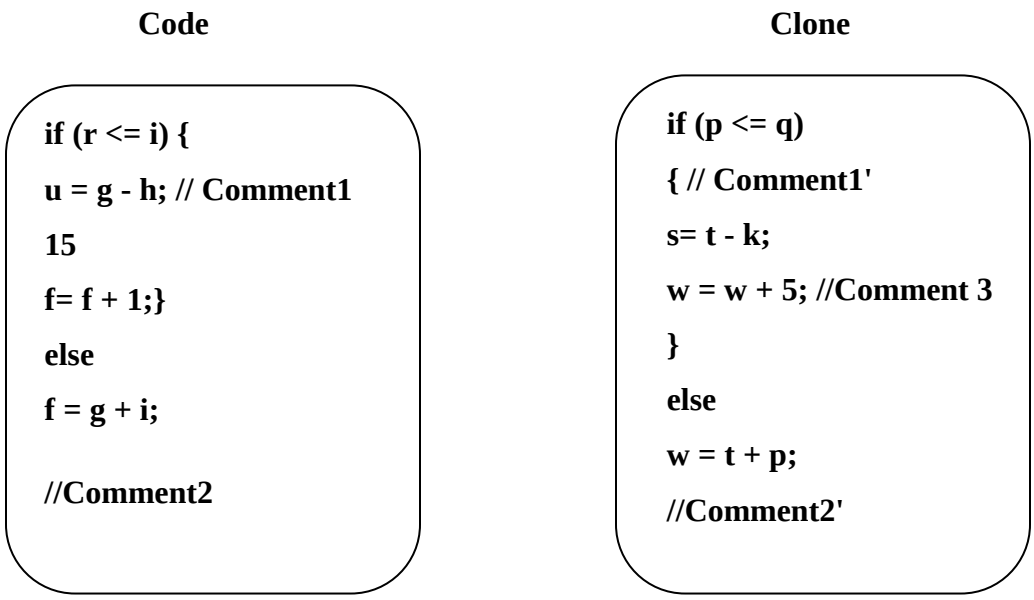


Figure 1.5: Renamed Clones.

1.4.3 Type III (Near Miss Clones)

These are copied fragments with more modifications. In this new statement can be added, removed or modified with renaming in identifiers, variation in literals, change in layout

and comments. If some modifications are done within the statements, then, also they are considered as Near Miss Clones.

Code

```
if (w <= x) {
y = w * 2; // Comment1
z = y - u;
p=p+1 //statement is
deleted in clone
}
else
y= w * 2; //Comment2
```

Clone

```
if (w<=x)
{
// Comment1'
y= w* 2;
z=y- u;
}
else // Comment2'
```

Figure 1.6: Near Miss Clones.

After deletion of one statement in second fragment they are consider as Type III clones.

1.4.4 Type IV (Semantic Clones)

If functionality of two code fragments is similar then they refer to Type IV or semantic clones. Functional similarity refers to both fragments have similar pre and post conditions. It is not necessary that clone is the copy of the native code. Sometimes same functionality can be implemented by different programmers by using different methods. Although they are not syntactically similar but performing the same functionality. Due to this these type of code fragments are considered as Semantic Clones.

Code

```
int m , p, w=1, t;
for ( p = 1; p <= t;  p++)
w = w * m;
```

Clone

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

Figure 1.7: Semantic Clones.

These two fragments find the power of a number although first is simple for loop code and second is recursive function. Hence no syntactic similarities exist between them. Therefore, these fragments are considered as semantic clones of each other. Following are the types of the clones which may be categorized under the Type IV category i.e. semantic clone:

i) Reordered Clone: If changes in sequence of statement in one code fragment do not affect control and data flow of naive program then this type of code fragments are consider as reordered clone.

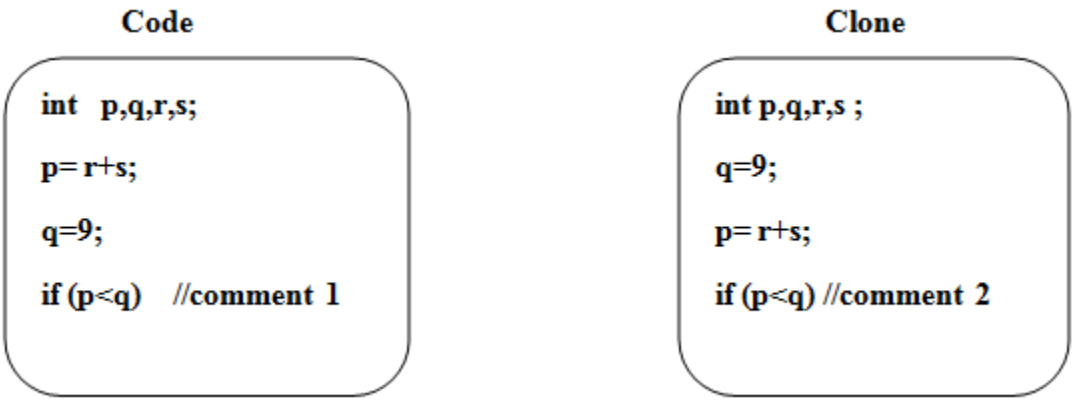


Figure 1.8: Reordered Clones.

ii) Structural Clone: Structural similarity works on the basic principle of software specification i.e. it tells similarities in the software specifications. For instance same functionality can be achieve by using while loop instead of for loop in the program.

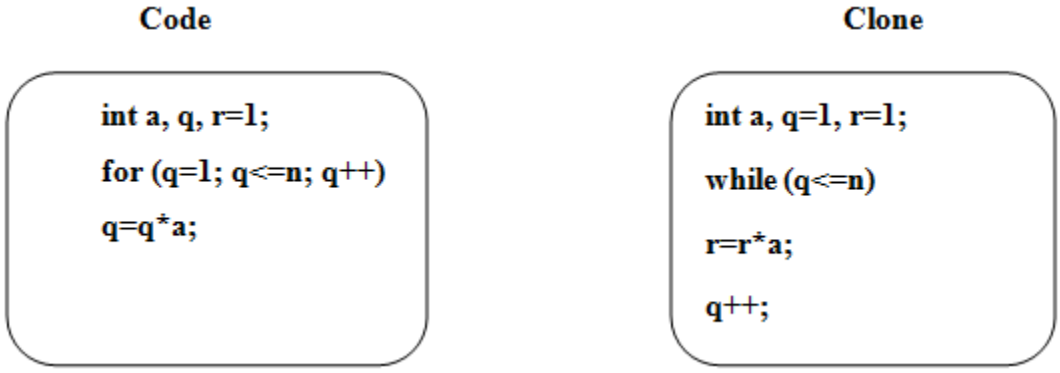


Figure 1.9: Structural Clones.

1.5 Reasons of Code Cloning

Code clones are intentionally used in system to get certain benefits of reuse mechanism. These are mainly added with intent of easiest way of providing similar functionality by using redundant code. Developer's tendency to complete project on time or to increase number of lines for better performance is another main reason for using clones in system. Sometimes it is difficult to know or proposed new approach for each requirement so clones provide solution for such requirements. Although clones create lots of maintenance problem but at the same time it helps developer in many ways. Due to which these are used in software systems. Following are the major reasons of code cloning [12] [13].

a) Time Constraint

During the process of software development, scheduling of each SDLC phase has been done. So time limit is allotted to every programmer involved in process of development. In order to meet this deadline, copying and pasting of existing code and make necessary changes according to the desired functionality is the easiest and fastest way available to developers.

b) Lack of knowledge about language

There are certain languages which does not support reuse and abstraction mechanism. In that case clones are the only solution to provide same functionality. Moreover, lack of developer's knowledge about programming language leads to clones in the software system.

c) Lack of Understanding of Requirements

Due to high number of specification in large system it is difficult to interpret and create novel approach for each and every requirement. Hence, use of existing code which provides same functionality is the easiest way left for developers.

d) Supports Reuse

Code cloning or redundant code supports the mechanism of reuse in software development. This reuse can be done in any phase of software development life cycle, i.e.

one can reuse requirement, design, code, test cases by simply using cloning.

e) By Co-incidence

There might be chance that programmer uses similar code to solve same kind of problem. However, one programmer is unaware about other programmer approach to solve problem but unfortunately these are detected as a clone by clone detection tools. For instance, same API is used to create button in different Java application. Hence these are those clones which are created unwittingly in system.

f) Developer's Performance

There are certain organizations which follow traditional method to measure performance of developers. For instance, if developer is producing more lines in one hour then he is more productive than others. This will encourage programmer to introduce code clones in system so that LOC of code is increased and can earn more incentives.

g) Risk in New Code

Novel code fragment developed by programmer may be more complex or more prone to bugs and errors. So developers often preferred to use already developed and tested code by simply copying or by making minor changes so that system is adaptive to new requirements.

1.6 Advantages of Code Clones

Clones are mainly considered as unfavorable for good programming activity still these are added by programmers to get certain benefits. Kapser and Godfrey [6] [13] studies positive impact of code clones and discuss beneficial part of them. The following points give the overview of few of them.

a) Fast method

Copying and pasting mechanism to develop a system is easier rather than to start a new from scratch. Hence, developers prefer to make clones to provide immediate solution of a given problem. Moreover, existing codes are tested properly so chances of bug are less as compared to newly developed software.

b) Basis for Templates

Many applications, for instance, website requires to follow the same design style in all pages, to obtain this same design code is used throughout the web pages. Hence code clone supports the template building.

c) Enhance Reuse Mechanism

There are certain languages which does not support reuse and abstraction mechanism. Reusing existing code by copying and pasting is the only way to achieve existing functionalities.

1.7 Drawback of Code Cloning

Besides negative effect on the maintenance of software, code clones has adverse impact on quality and modularity of code structure [11]. Due to which it become difficult to reuse that code for future project and additional increase of hardware and software requirement to run such bad design system [9]. Moreover, bugs are propagated more easily and it requires lots of effort to make consistent changes to remove them from whole system which results into high maintenance cost. The following list gives an overview of these problems.

a) Increased Maintenance Cost

It is necessary that changes made in system containing clones should be consistent [14] otherwise it is difficult to reuse. So when modification is done on clone then it would require performing on all replicated parts which increases maintenance cost of system.

b) Increases Difficulty in Maintenance Work

It is often seen in software development process that the developer team is different from the maintenance team. Moreover, code clones complicates the understanding of the code which in turn affects the maintenance team to maintain or modify the code.

c) Increased Resource Requirement

The introduction of code clone results into increase in size of program due to which large amount of code is needed to be compiled which in turn increases compilation time.

Moreover, clones make program huge and complex and to run such system, hardware and software specification also increases.

d) Increased Probability of Bad Design

A clone avoids the rule of modular and structured programming which leads to bad programming design. It also violates the basic concept of object-oriented programming like inheritance and encapsulation which leads to difficulty in reusing the code for future projects.

1.8 Motivation and Objective

These code clones are consider as beneficial in terms of fast development of software system or enhancing reuse mechanism [1] [6]. At the same time, they are deleterious when it comes to maintenance point of view [9] [11] because modification in one code snippet may lead to do changes in all cloned fragments results into higher maintenance cost. It is also assumed that changes made in clones should be consistent so that later version of software will not face any inconsistency and do not hindrances the evolution of system [14]. Moreover, increase in resource requirement, inconsistent code and bug propagation [12] are other serious factors associated with code clones. The previously made studies also suggest that considerable fraction of code which is 7% -23% is found to be duplicated code [1] [4] [22] and due to their negative impacts, they should be removed [11]. Thus, it becomes necessary to find all code clones throughout the software system with efficient clone detection tool.

The other factor which supports clone detection is that, this process is not only limited to detect code clones but it also plays an important part in software analysis and understanding software evolution [2] [3].

As various software engineering tasks like program understanding, code quality analysis, aspect mining and bug detection may requires information about these similar code fragments.

The main aim of this thesis is to propose a tool which is able to detect both syntactically as well as semantically similar codes. To design and implement such tool a hybrid

approach is used which is combination of program dependence and metrics based techniques. This tool works only for programs written in java language.
Following objectives should be fulfilled to achieve this aim:

- Study the process of clone detection in detail.
- Study the various clone detection techniques and tools based on them.
- Compare these techniques on the basis of their efficiency to detect type of clone.
- Propose a new methodology to detect clones in Java programs.
- Design and implement the proposed methodology to create novel clone detection tool.

1.9 Outline of Thesis

This report is further organized into following section:

Chapter 2: It gives a review of clone detection process, various existing clone detection techniques and tools and comparison between them.
Chapter 3: It identifies the problem in existing clone detection techniques and specifies the solution.
Chapter 4: It describes proposed methodology and its implementation.
Chapter 5: It discusses results evaluated from testing programs.
Chapter 6: It concludes proposed work and describes improvement scope for future.

Chapter 2

Literature Survey

This chapter covers literature survey of clone detection in detail. It includes generic clone detection process, various techniques and tools used for this purpose and comparison between these techniques.

2.1 Clone Detection

A lot of identical code has been used on the programs which are either under the development or maintenance phase. Due to this duplication code size and complexity, defect probability and resource requirement increases which in turn making program maintenance more difficult [1]. However refactoring is used to remove code clones but it is not possible to remove all types of clones from this process. Thus, it becomes necessary to find all clones from source code and the process of detecting these clones is called Code Clone Detection.

Clone detection has many other advantages like bug detection, detection of candidate libraries, enhancing program understanding and reduction of program size. For all these reasons it is an active research area from last many years and several tools and techniques are proposed to detect them.

2.2 Clone Detection Process

A clone detection process is used to find clone pairs in programs based on some definition of similarity. This definition varies from one clone detection tool to another because this definition depends on the technique used to detect them. But process of clone detection is generic for every clone detection technique and results into whether code fragments are actual clones or not. During this process several phases are encountered and each code fragment is required to compare with every other possible code fragment so that maximum number of clones can be detected. Due to this, clone detection process is costlier and requires high computational speed. In order to reduce the number of comparisons, preprocessing process has been done which filters several code

fragments. Following are the phases which are involve in general clone detection process.

2.2.1 Pre-processing

This is the first phase of clone detection process; it starts with definition of similarity used to detect code clones on the basis of technique chosen for this purpose. After this source code is divided into several code fragments and removes those parts which are not useful from comparison point of view. This filtered code is further divided into disjoint set of code fragments. Now these code fragments are known as source unit which are maximal length code snippets used for comparison purpose to detect clone pairs.

2.2.2 Transformation Phase

In order to detect clones, various comparison algorithms are applied on code fragments. These comparison algorithms are either directly applied on source code like in textual based clone detection techniques or on any intermediate representation of source code based on clone detection technique applied. Hence conversion of source unit into appropriate intermediate stage for more accurate comparison is the main objective of this phase and this process is also called extraction in terms of reverse engineering.

The intermediate stages which are extracted might be a removal of white space and comments from original code or series of tokens, abstract syntax trees, program dependence graph of code fragments used for finding clones.

After extraction process, certain tool does modification on fetched intermediate stage to enhance clone detection process. These modifications are generally removal of comments, normalization of identifiers and several structural transformations so that syntactic dissimilarities could be ignored.

2.2.3 Match Detection

This transformed source unit is given as input to suitable comparison algorithm so that similarity can be found by comparing them. Each source unit is compared and if similarity is found then merges with similar neighbor source units so that maximum length clone can be detected. As input to this phase is normalized code, so it find a clone

pair list and give their locations with respect to this new transformed code. Each clone detection tool uses their specialized comparison algorithm, among them suffix trees and hash functions are widely used in clone detection.

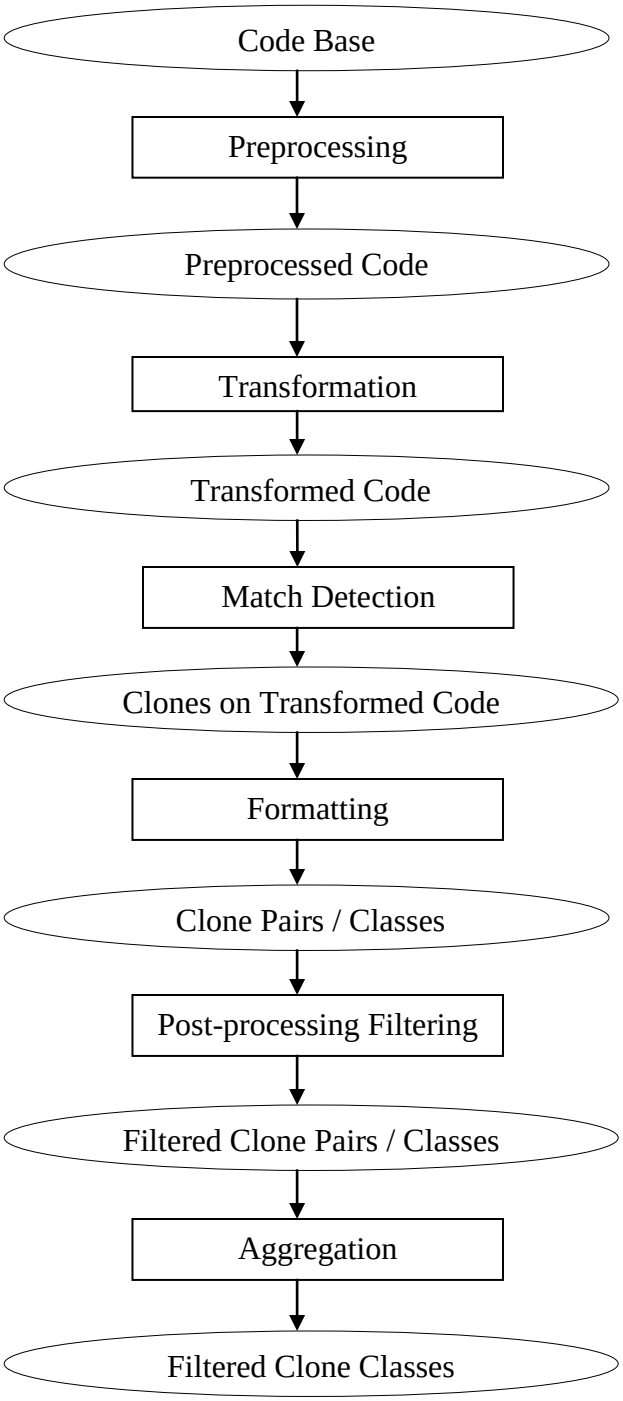


Figure 2.1: Generic Clone Detection Process.

2.2.4 Formatting

The clone pair list obtained in previous stage locates clone with respect to transformed code, it is required to map this list with original code. During this phase, list mapping has been done so that exact location of clone in original source code can be found. Now clone pair list contains a line number of source code where it actually exists.

2.2.5 Post Processing

Clones extracted from source code are checked to find whether they are actual clones or false positives. This phase finds actual clone and rank them either manually or with the help of automated heuristics.

Manual analysis: It requires human experts to compare each code clone with original source code to find whether they are actual clone or not.

Automated heuristics: It refers to use characteristics of clones like length, number of time particular clone is used and diversity in order to find and rank actual clones.

2.2.6 Aggregation

For proper analysis and data contraction, detected clone pairs are combining into clone classes and clone family.

2.3 Overview of Code Clone Detection Techniques and Tools

Clone detection is widely open research area from last many years [4]. There are several techniques and tools are mentioned in literature and broadly categorized into following types:

2.3.1 Text Based Code Clone Detection Techniques

In this type of clone detection techniques line by line comparison has been made on two code fragments on the basis of textual similarity exists between them. These techniques do not require any filtration or normalization process [3] and apply directly on source code. Hence, they can detect exactly identical code with slight variations in comment or layout. The clone detection process involves string based matching of code fragments and

results whether they are clones or not. This technique does not consider any renaming of variable or syntactical changes exist in code fragments. So it not highly efficient technique and can detect only Type I clones but certain enhancement has been made by using the concept of fingerprinting on substring of source code [16] [17]. Following are the tools which are proposed using this technique.

Johnson et al. [16] [17] enhances this process for better maintenance and reengineering of legacy systems. He finds fingerprints for substring present in source program and used them for comparison purpose. Clone detection process involves hashing of fixed length programs then finds line with same hash value with the help of sliding window and incremental hashing function and reported them as clones.

NICAD [10] is basically a hybrid approach which uses tree concept along with text based technique in order to detect clones. This tool works in two stages. Firstly flexible pretty printing and normalization process is used to identify potential clones and then line by line textual comparison is made on these potential clones in order to find actual clones.

SDD [18] is another text based tool which is efficient to detect near miss clones in large software system. SDD algorithm uses the concept of n- neighbor approach to find number of repetitions in system.

Ducasse et al. [19] uses string based matching along with scatter plot diagrams to visualized clones present in system. In this tool, detection process involves line by line comparison of source code. These comparisons generate results in Boolean form and store them in matrix and then resultant values are visualize in scatter plot diagram form.

2.3.2 Token Based Code Clone Detection Techniques

Token based clone detection technique uses concept of parsing or lexical analysis in order to detect code clones. In this technique, normalization process is used to convert source code into intermediate stage which is chain of tokens. These tokens are generated with the help of any parser and comparison algorithms are applied on them to detect

clones. If string of tokens is matched in source and target program, then they are code clone otherwise not. As source code is converted into tokens by using concept of lexical analysis so tools based on this are capable of detecting clones even when white spaces and comments are inserted. Hence this is more efficient technique then text based technique and can detect Type I and Type II clones however time complexity increases due to normalization process. This approach is exploited by various tools and trying to find clones.

Dup [22] is combination of text based and token based technique which divides program in parameterized and non parameterized tokens to find Type I and Type II clones. It uses hashing function in order to find Type I clone and position index for type II clones.

CCFinder [20] is one of the efficient token based tools which can detect code clones from Java, C, C++, COBOL and many other source program files. This tool convert source file into series of tokens and then comparison of these tokens are made with the help of suffix tree algorithm. It also provides clone metrics to find clone pairs and clone class. Moreover for better visualization scatter diagram and plot diagrams are used.

CP- Miner [21] uses least common subsequence approach in order to detect clone activities in large software systems. It is capable to detect these activities in operating system as well. Firstly, lexical analysis is done on source code to change into tokens and then data mining algorithms are applied to detect them. As this technique does not use sequential analysis it is proficient to detect reordering of statements. Moreover it helps to find bug related to clones and fix them accordingly.

2.3.3 Tree Based Code Clone Detection Techniques

Tree based clone detection is capable of detecting clones in which code is inserted or deleted i.e. Type 3 clones or near miss clones [1]. In this source code is represented into abstract syntax trees in contrast to tokens and then pattern matching is applied on them to find similar sub trees which are consider as code clones [12]. This technique is independent of various syntactic information like identifier name and their values and

literal changes so it requires advanced algorithms to find clones. Due to this it is complex technique and has some time complexity and portability issues [2] but it is efficient than text based and token based technique.

Ira D. Baxter et al. [23] presents a tool CloneDr which generates abstract syntax tree by using parser then three comparison algorithms are used to detect code clones. First algorithm is simply used to compare subtrees by categorizing them in buckets according to their hash values. Sub trees in same bucket are compared and detected as exact clone. The second algorithm is used to find semantic clones such as change in sequence of statements or declaration of statement by using "sequence detection algorithm" then third algorithm is used to find complex near miss clones.

William et al. [24] proposed a tool called Asta in which AST is converted into string of XML. It combines various AST's into XML string and data mining approach is applied on this XML string to find clones. It provides pattern of matched clones and supports variable renaming.

Jiang et al. [25] presents a tool named DECKARD introduced a novel approach of characteristic vector in Euclidian space to find similar subtrees. These vectors stores structural information about abstract syntax tree and then cluster them based on these vector values. For clustering concept of locality sensitive hashing is used in this tool. The subtrees which comes under same group were considered as clones. It is a language independent tool and finds clones with high accuracy and scalability.

2.3.4 Graph Based Code Clone Detection Techniques

In this technique, program dependence graph is obtained from source code as an intermediate state. Data and control dependency is carried by PDG's between statements of the source code i.e. it carries semantic information of program. Due to which it is independent of variable renaming, literal changes or any other syntactic dissimilarity exists in code fragments. In order to detect code clones, isomorphic sub graphs are identified [12]. However, finding similar subgraphs is tedious to obtain and has some scalability and portability issues [1] [7]. Following are the tools which are based on graph based clone detection technique.

Komondoor et al. [32] proposed an approach called PDG-DUP that uses program dependence graphs (PDGs) and program slicing to find non-contiguous clones, intertwined clones and clones that involves variable renaming and statement reordering. In this backward slicing is used for clone detection while forward slicing is only used for matching the predicate nodes. This tool finds duplicated code fragments in C programs and displays them to the programmer.

Krinke [33] presented an approach to identify similar code fragments in programs based on finding similar sub graphs in attributed directed graphs called Duplix. As this technique uses program dependence graphs and therefore considers both structural and semantically similar code segments. This approach uses an iterative approach (k-length patch matching) for detecting maximally similar sub graphs in the PDG.A prototype implementation of this technique shows that it solves the problem with non polynomial complexity with high precision and recall.

Liu et al. [35] proposed a tool called GPlag which uses PDG based algorithm to analyze the graph and to detect the clones. It uses algorithm called sequential pattern mining to discover copy/paste. It finds the clone with high precision and added the new feature like text clone and text clone file ratio.

Gabel et al. [34] proposed a scalable detection algorithm for finding semantic clones. This algorithm is based on selecting PDG subgraph based on its related structured syntax i.e., it converts program dependence based problem into easier trees like problem. Then this tree based problem is solved by scalable algorithm. This is the enhancement of Deckard abstract tree method which introduces concept of characteristic vectors [25]. This algorithm is implemented in a tool and its focus is only on semantic clones rather than finding copy paste activities.

Y. Higo et al. [36] proposed a tool called Scorpio which uses two-way slicing to detect clones i.e. forward and backward slicing. This is because if some clones are not able to detect by forward slicing then those are detected by backward slicing or vice-versa. This

tool is developed only for Java language. As earlier existing tool based on PDG clone detection approach is slow in detection of contiguous clones, therefore this new approach is introduced. This tool is further used by Keisuke et.al [39] for clone removal with the help of Form Template Method. It is a clone removal technique which uses the concept of separating duplicated data and non-duplicated data by managing base and derived classes. Similar code is put in base class and derived class has non-duplicated data. For similar functionality each class has common base class. Due to this similar code clone present in different methods are integrated in single class.

2.3.5 Metrics Based Code Clone Detection Techniques

In metric based code clone detection technique, different metrics of code are calculated and code clones should possess similar values of these metrics. These metrics are calculated on the basis of some relation that exists between the statements of program. Earlier technique involves calculation of metrics directly from the source code. Now a day's many tools using this technique with certain enhancements. Instead of applying directly on source code, textual representation of the program is converted into an intermediate stage like abstract syntax trees or program dependence graphs and then finds metrics on this intermediate stage. These metrics contains information about name of methods, variables, literals or about layout and control of program. Those code fragments which possess similar metric values are considered as code clones.

Jean Mayrand et al. [26] used this technique in tool named Datrix in which 21 metrics are calculated on the basis of four categories viz. name, layout, expression and control flow of program [27]. This tool change source code representation into abstract syntax trees and then these trees are converted into Intermediate Representation Language (IRL) to make tool language independent. This IRL deals with above defined categories but mainly control and data flow metrics are used to detect code clone as they stored semantic information.

Kontogiannis et al. **[31]** uses metrics technique in two different ways in order to detect

code clones. In first way metrics are calculated for whole program or function i.e. it includes full begin-end block or function specific metrics. It compares data on the basis of data and control flow among methods. These metrics are

1. The number of functions called (fanout).
2. The ratio of input/output variables to the fanout.
3. McCabe cyclomatic complexity.
4. Modified Albrecht's function point metric.
5. Modified Henry-Kafura's information flow quality metric.

In second way it uses to do statement by statement analysis of whole block by applying dynamic programming techniques.

Davey et al. [30] uses concept of neural network with metrics to detect code clones. Features of code blocks are stored in vectors and then trained in Dynamic Competitive Learning (DCL) Model to find similarity. Type I, Type II, Type III clones can be detected by this approach.

2.3.6 Hybrid Code Clone Detection Techniques

There are certain hybrid tools which uses combination of above discussed syntactic and semantic techniques to detect clones. By utilizing benefits of various techniques, it is able to detect all types of clones with more efficiency and accuracy.

Koschke et al. [28] presents a technique which overcomes limitation of token based techniques by using Abstract Syntax tree in combination with suffix tree algorithms. In this tokenization of abstract syntax tree has been done and then in contrast of comparing nodes of annotated tree their tokens are compared. It is efficient to detect Type I and Type II clones with high precision and recall values with linear time and space complexity.

Leitao [29] applies combination of both structural changes detection techniques and semantic techniques on programs to detect clones. Abstract syntax tree is generated for code fragments and based on them metrics are calculated then these metrics are merged with graph based techniques to detect clones.

Table 2.1: Various Existing Clone Detection Tools

Tool/Ist Author	Approach Used
Text Based Tool	
Johnson [16] [17]	Fingerprinting
Nicad [10]	Hybrid technique uses concept of tree along with it
SDD [18]	n-neighbor approach
Ducasse [19]	Stringbased matching
Token Based Tool	
CCFinder [20]	Suffix tree based search
CP-Miner [21]	Subsequence data mining
Dup [22]	Line wise suffix tree
Tree Based Tool	
CloneDr [23]	Hashing of syntax tree
Asta [24]	Syntax pattern matching
Deckard [25]	Generation of characteristic vector
Graph Based Tool	
Komondoor [32]	PDG + Program Slicing
Duplix [33]	k-length Pattern Matching
GPLAG [35]	Sequential Pattern Mining
Gabel [34]	Scalable Detection Algorithm
Scorpio [36]	Two-way Slicing
Metric Based Tool	
Jean Mayrand [26]	Control and data flow metric
Davey [28]	Training neural networks
Kontogiannis [20]	Compare begin-end blocks
Hybrid Tool	
Koschke et al. [29]	Tokenization and abstract syntax trees
Leitao [30]	Abstract syntax trees and Graphs

2.4 Comparison of Code Clone Detection Techniques

Clones are considered as harmful from maintenance point of view [11]. Due to which these clone detection techniques have been introduced. Based on them several tools have been proposed and certain experiments are done to evaluate their performances. Bellon et al. [7] [8] studied various tools based on these techniques to compare them on the basis of six state of the art technique for the tool targeted C and Java. CK Roy et al. [2] [4] provides hypothetical scenarios and used their published properties to compare these

tools. These scenarios are based on the type of clone [3] introduced in code fragments. The results of their experiment prove that certain limitations are associated with each technique. Following Table 2.2 compares various clone detection techniques on the basis of how efficiently they detect particular type of clone.

Table 2.2: Comparison between Existing Clone Detection Techniques.

Type of Comparison	**Portability**	**Type of clones**	**Efficiency**
Text based clone detection	Good	Syntax (Type 1 Clone)	High
Token based clone detection	Average	Syntax (Type1 and Type 2)	Low
Tree based clone detection	Poor	Syntactic(Type1, Type 2 and Type3)	High
Graph based clone detection	Poor	Syntactic and Semantic	High
Metrics based clone detection	Poor	Syntactic(Type1, Type 2 and Type3)	Medium

Chapter 3

Problem Statement

Clone Detection is of great concern for better maintenance and quality of software system. Systems containing code clones are highly susceptible to bugs and defects and become hurdle for better evolution of software system. Hence it is an open area of research from many years and results into various clone detection techniques and tools based on them. But as discussed in literature survey certain limitations are associated with each clone detection technique and tool.

Tools based on text based techniques are applied directly on source code can able to detect only Type I clones whereas tools based on tokens have done lexical analysis on source code and able to detect Type II clones also. But both these techniques can detect only syntactic similar codes and fails if any modification is done within the statements. However some tools use the concept of abstract syntax tree in order find Type III clones but this approach requires complex algorithms and parsers to find similar sub trees. Metrics based tools are rather straight- forward and suitable for large software system but it is not efficient if applied directly on source code. So for better efficiency intermediate stage should be obtained and metrics were found based on them which require high cost. Moreover, metrics contains structural information so able to detect syntactic clones only.

Program Dependence Graph based technique is the only technique which is able to detect code clones both syntactically as well as semantically. But according to study of CK Roy et al. [3] and Bellon et al. [8], this technique can give many false positives and hence show low recall and precision value for positive clones.

Moreover, in literature survey it is also indicated that there are certain hybrid tools which are combination of abstract syntax tree and text based or metric based techniques in order to detect actual clones. But they can detect only syntactic clones.

So following problems are identified in existing work:

i. A novel approach is needed to detect code clones efficiently.

ii. Both syntactic as well semantic clones should be detected by tool.

iii. As clone detection process has many phases, so tool should be automated and light-weighted so that each phase is executed without any additional computational resources.

iv. Many false positive clones are detected by tools which should be removed to get high precision value.

v. There is a need of hybrid approach which uses different techniques other than tree based clone detection technique.

This thesis presents a hybrid approach to detect actual code clones. It merges program dependence graph based technique with metrics based techniques to find code clones for java programs. Firstly PDG is obtained using java byte code, from which adjacency matrix is achieved which carries semantic information of program. The values of adjacency matrix are compared to find potential clones.

After getting potential clones, various object oriented and control metrics are calculated to verify them as actual clones.

A clone detector tool called HCDectector is proposed to find actual clones for java programs by using hybrid approach. This is an automated tool with user friendly interface which displays obtain PDG, adjacency matrix , calculated metrics and gives result whether tested programs are actual clones or not.

Chapter 4

Proposed Work and Implementation

Code clones are considered as threat to the maintenance and correctness of software systems if they are not consistently managed [38]. It is estimated that 90% of total software cost spent on the maintenance problems [5]. Refactoring and other reengineering activities are used to remove them but not efficient for all types of clones and it requires high cost. Hence various clone detection techniques and tools are proposed in literature in order to detect clones but certain limitations are associated with them. Moreover it not feasible to track code clones manually. So clone detection is an open research area and there is a need of an automated tool which can detect clones efficiently.

The proposed work presents an automated clone detection tool for java programs. This tool is a hybrid approach based tool which merges program dependence graph based technique and metrics based technique to detect code clones efficiently. The following sections describe the essential requirements to create such tool, proposed methodology and implementation of work.

4.1 Design of Solution

To design a system which is able to detect all four types of clones efficiently on the basis of semantic information of a program, a program dependence graph which depicts flow of data and control between the statements of a program should be obtained. Hence, on the basis of this PDG potential clones can be detected. After finding potential clones, it is essential to check whether they are actual clones or not, this can be done with the help of metrics. If potential clones yields same metrics value then they can be actual clone. So, proposed work is hybrid approach which requires these two as essential elements:

4.1.1 Program Dependence Graph

Program dependence Graph is a directed graph [32] which determines two types of dependency that exists between the statements of the source code. These two dependencies are Control Dependency and Data dependency. In this graph we represent

statement and predicates as nodes and relation between them is represented by edges [40].

Control Dependency: It exists between two statements only when first statement is conditional predicate and execution of second statement depends on the result of the first statement [33]. This is illustrated by following code:

```
If (a<b)  //statement 1
then
 c = b -  a ; //
statement 2
else
c= a- b ; //statement 3
```

In the above program execution, statement 2 and statement 3 depends on the result of predicate expression at statement 1. Hence control dependency exists between them.

Data Dependency: It exists between two statements only when first statement contains the definition of variable and second statement uses the variable without redefinition of the variable [40]. This can be explained with the help of following code:

```
int a = 2; //statement 1
b=a+c;     //statement 2
a = p;    //statement 3
d = a + c; //statement 4
```

In the above program statement 1 and statement 2 are data dependent while statement 3 is not because it contains the redefinition of the variable and statement 4 is dependent on statement 3 only.

Example 1: These dependency are shown with the help of following program [33]

```
1 int func(int i, int j) {
2 int k = 10;
3 while (i < k) {
4 i++;
5 }
6 j = 2 * k;
7 printf("i=%d, j=%d\n", i, j);
8 return k;
9 }
```

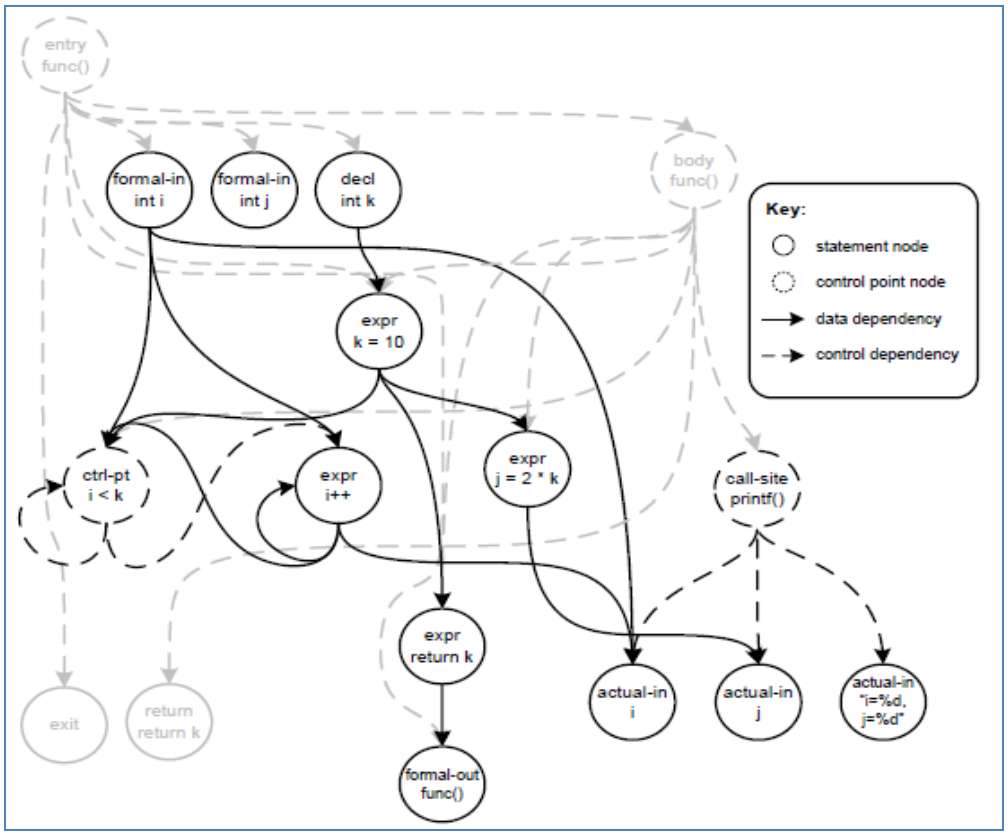


Figure 4.1: Program Dependence Graph of example 1 [33].

4.1.2 Metrics

There are certain metrics of a program which yields same value when any similarity exists between the code fragments. These metrics can be class metrics, layout metrics, method metrics and control metrics [27]. There are certain other metrics like

Kontogiannis et al. [31] determines number of functions called (fanout), ratio of input/output variables to the fanout, McCabe cyclomatic complexity, Modified Albrecht's function point metric, Modified Henry-Kafura's information flow quality metric to determine whether code fragments are clones or not. For program to be clone there yield values should be same.

4.2 Proposed Methodology

Our proposed tool focuses on the semantic information carried in PDG and applies comparison operation on this to find potential clones. After detection of potential clone it is necessary to verify whether they are actual clones or false positives [2]. To solve this purpose various metrics are calculated and comparison of yield values has been made. Hence, this tool goes through various phases during its clone detection life cycle.

4.2.1 Adaption Phase

This is the first phase which determines whether inputted files are in .class format or not. As proposed system is only for java programs and works on java byte code, therefore, input files should be java file with .class extension. Java .class file contains compiled byte code of particular file and it transforms the code into unified format by removing all syntactic dissimilarity that exists in the program. For instance, while loop and for loop have different syntax but performs similar functionality. This structural difference does not appear in java byte code which is beneficial in order to find semantic clones.

Java Byte Code: Java byte code is a complied code (low level language code) of program written by programmer in high level language. It is generated in JDK environment by using Javadoc compiler. It is consider as an intermediate representation of source code which makes java programs independent of any platform. The main purpose of using java byte code is that API which we used for creation of PDG and calculation of metrics works only on Java byte code. Moreover, it helps to find semantic clones by removing syntactic dissimilarities.

Figure 4.2 represents the java byte code of program which finds largest among three numbers. This code is platform independent and can run in any environment.

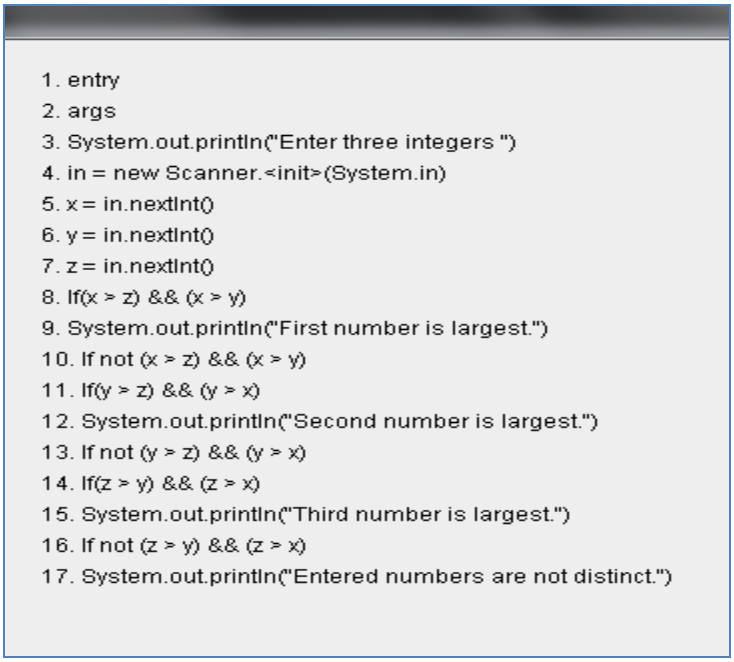


Figure 4.2: Java Byte Code of Program to Find Largest Number.

4.2.2 Transformation Phase

In clone detection process, the comparison algorithms are mostly applied on the intermediate stage except text based detection process. The extraction of intermediate stage from the source code is done in this phase. This intermediate stage can be tokens, trees and graphs based on the clone detection technique applied [2]. As program dependence graph based technique is used to detect code clone, so PDGs are obtained from source code during this phase of proposed system. To get PDG, Java System Dependence Graph API is used which is the only Java API available to perform this function. After extraction of PDG, analysis on control dependence and data dependence nodes are made and stored in the form of adjacency matrices of size V X V (V is the number of nodes in PDG). If one node is data dependent on other then this relation is represented by 1, control dependence is represented by 2 and independent nodes are represented by 0.

Figure 4.3 shows PDG of program shown in Figure 4.2 which finds largest number among three numbers. Here, black line represents the control dependency while red line represents the data dependency among the nodes.

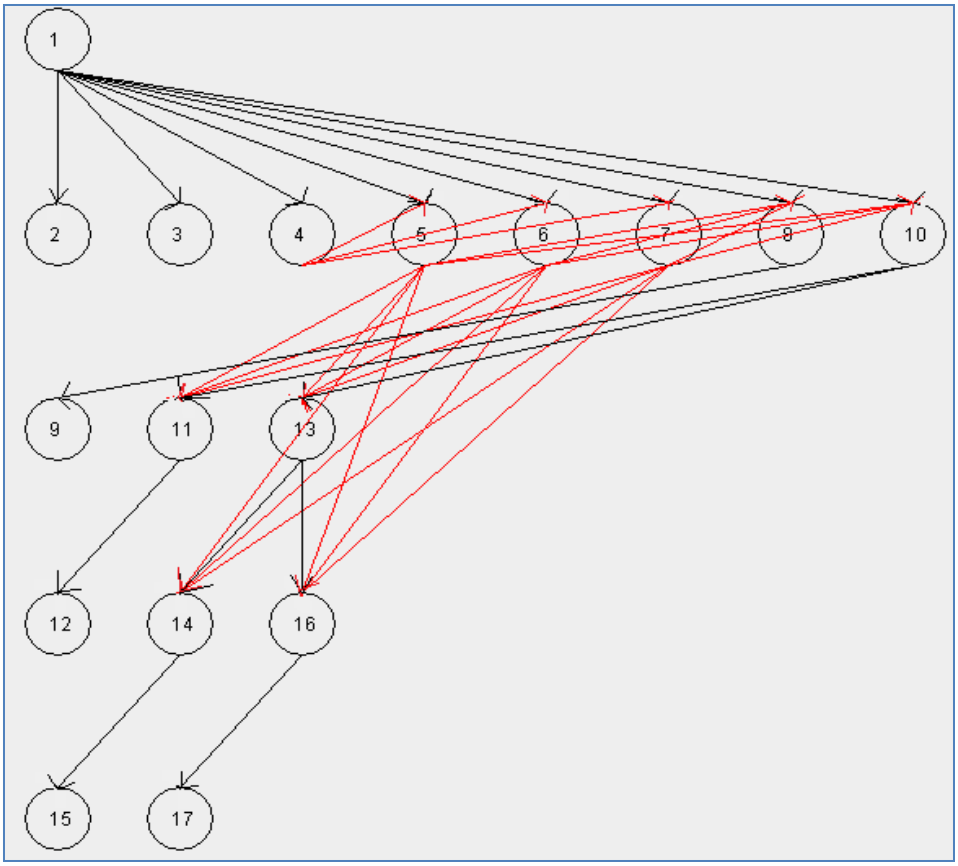
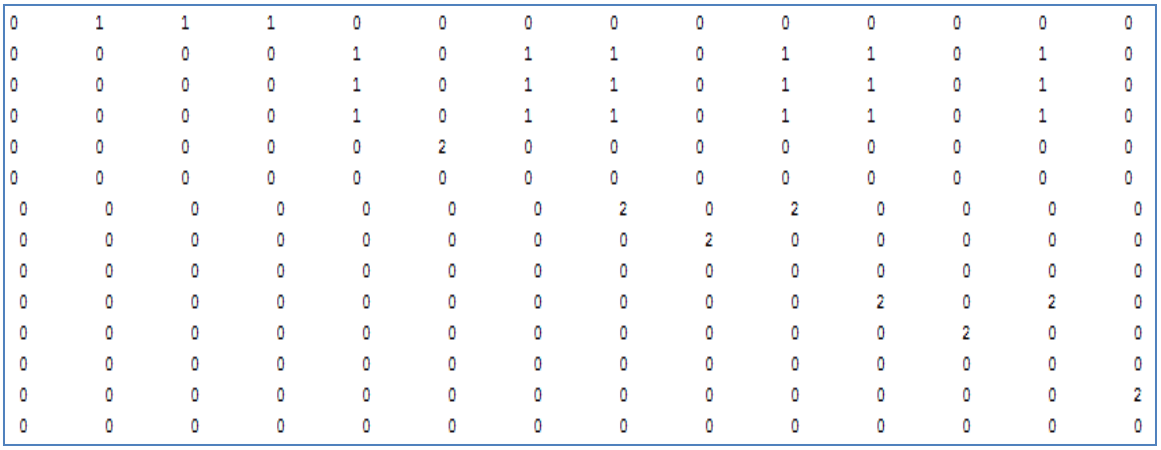


Figure 4.3: PDG of Program Find the Largest among Three Numbers.

4.2.3 Normalization Phase

Normalization phase is used to remove irrelevant differences exist in code fragments like whitespaces, comments and layout [3]. But program dependence graphs are independent of these changes so it is not required to remove these differences from code fragments. However PDG is sensitive to addition, deletion or modification in any statement and if such conditions does not contribute in any data and control flow of program then they should be removed to reduce number of comparison and to detect clones more accurately. Due to these reasons, there is a need to further modify these extracted matrices and obtained filtered matrices. Hence, we remove those nodes which are control and data independent so that information about only semantically similar nodes should retain in matrix which helps us to detect clones even when any data or control independent statement is added or deleted. Figure 4.4 represents filtered adjacency matrix of an above program in which control independent and data independent nodes are removed.



0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	0	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	2	0	2	0	0	0	0
0	0	0	0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	2	0	2	0
0	0	0	0	0	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.4: Filtered Adjacency matrix of the program 1 represents data dependency 2 represents Control dependency.

4.2.4 Comparison Phase

After normalization, filtered matrices are obtained and fed into comparison phase where it determines whether any similarity holds between both code fragments or not. For this purpose, comparison algorithm compares the values of both matrices to find corresponding node to node dependence between programs i.e. if one node is either control, data or independent of other nodes in first matrix then corresponding node should contain same values with respect to other nodes in second matrix. If this similarity exists then they are potential clones otherwise not. Hence comparison algorithm is used to check data and control flow rather than any textual similarity. Therefore, if two program exhibits similar control and data flow then they are consider to be potential clones.

4.2.5 Metrics Computation Phase

Now after getting code fragments as potential clone, it is necessary to verify whether they are actual clones or not which can be done by calculating metrics and then making comparison on yield values. For this purpose, proposed tool calculates various control flow metrics [2] [31] and object oriented metrics at class and function level for java programs [40]. To calculate object oriented metrics Java reflection API is used whereas control flow metrics are calculated with the help of program dependence graph (PDG). Table 4.1 show various metrics used in system to verify potential clones.

Table 4.1: Metrics Used in System.

Metric Type	Abbreviation	Description
Control Flow Metrics	Complexity	McCabe cyclomatic complexity
	Cnodes	Number of control nodes
	Dnodes	Number of data nodes
	Edcounts	Number of edges
Class Metrics	Fanout	Number of methods called
	Tcount	Total variable
	pubV	Number of public variable
	protected	Number of protected variable
	Private	Number of private variable
Function metrics	Fname	Function name
	Ptype	Type of parameter passed
	nParameter	Total number of parameter
	Rtype	Return type

4.3 Implementation

This tool requires two .class file as input and finds similar codes in both files with the help of PDG and above mentioned metrics. A clone detection process starts with by giving right input (i.e. Java .class file), if wrong input is given; this process ends automatically at this point. When byte code is given as input to system then PDG is generated for programs. This is obtained by exporting PDG generation package from Java system dependence graph API. The other functions of this API are also used to get number of nodes in graph and to generate adjacency matrix. In addition to this several supporting functions are used to get required functionality of filtering matrices, finding potential clones, calculating metrics and verify them as actual clones or not. This functionality should be achieved in sequence and which is depicted in following flow diagram in Figure 4.5.

4.3.1 Flow Diagram of Proposed Work

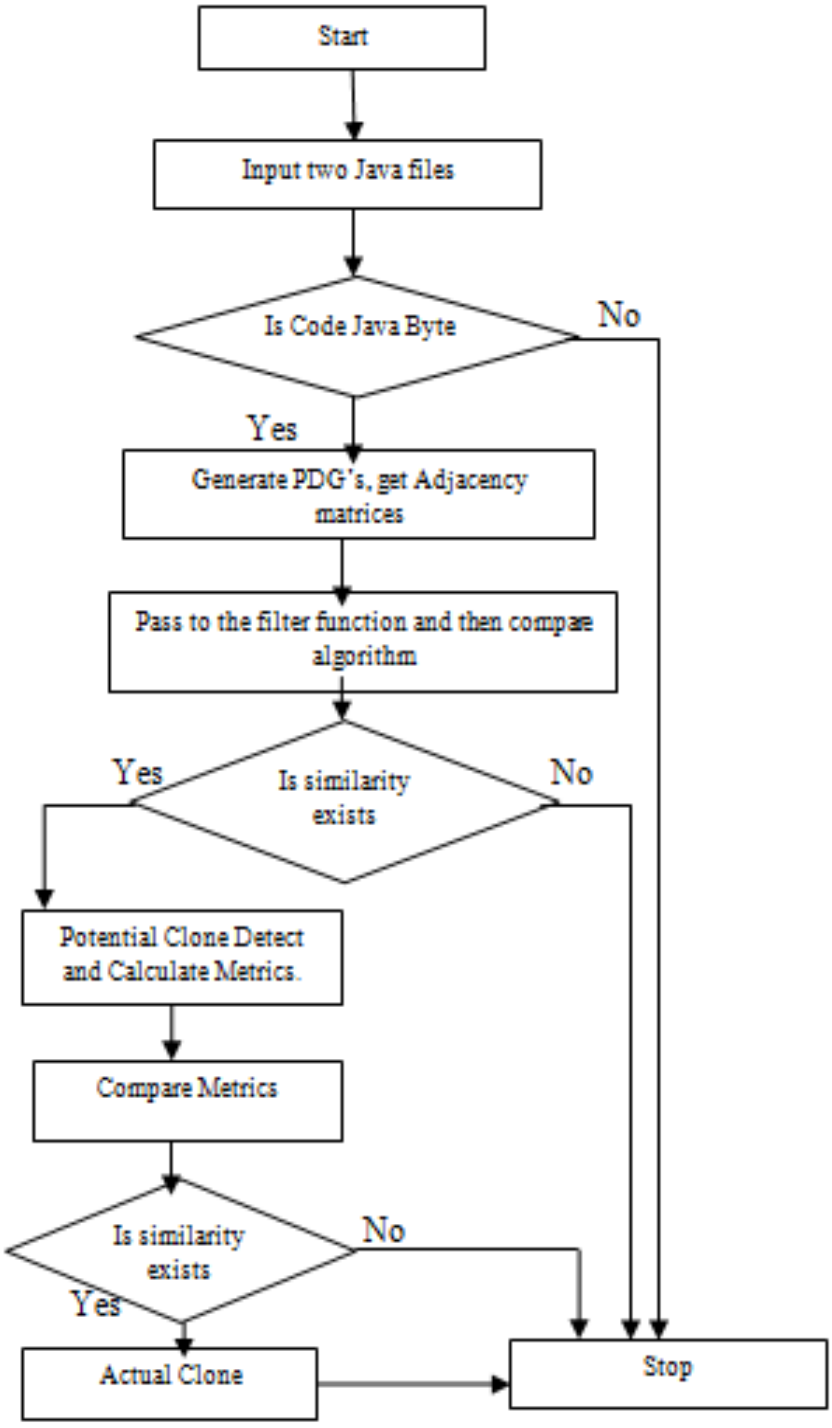


Figure 4.5: Flow Diagram of Proposed System.

4.3.2 Architecture of Proposed Work

The proposed tool uses both semantic as well as syntactic technique to find code clones. Figure 4.6 represents the architecture of proposed system.

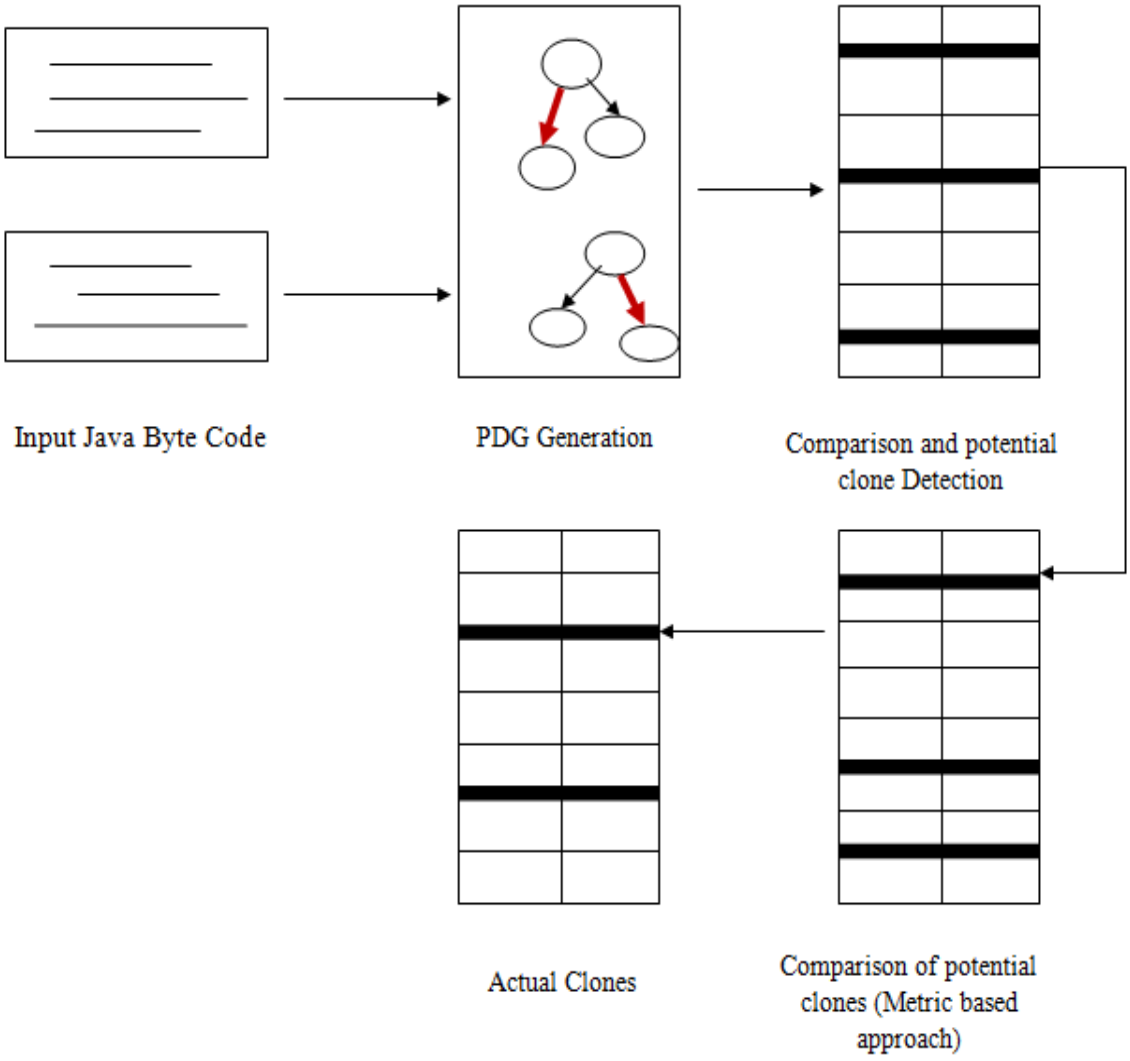


Figure 4.6: Architecture of Proposed System.

4.3.3 Algorithms Used

There are many supporting functions used in order to get required functionality. After entering java .class files, it is required to generate program dependence graph and adjacency matrix of both files. Following algorithms are used to achieve PDG's and their corresponding Adjacency Matrices.

Algorithm 1: Create_pdg (file1, file2)

Input: Two java .class files

Output: PDG and Adjacency matrix

// Following API should be imported in order to create graph.

com.graph.pdg.ProceduralDependenceGraphMatrix;

1. Use ProceduralDependenceGraphMatrix.getPDGByMethodSignature (absolute path of file) to create PDG
2. int i = lvPDG.countTotalNumberOfNode()
3. Print all control and data dependency.

First function which is used after creation of program dependence graph is filter function which removes all the data and control independent statements which do not affect the flow of program. In this function if value of both row and column corresponding to particular node is zero then this node is considering as independent node and should be removed from matrix as it does not affect the control and data flow of program.

Algorithm 2: Filter (adjacent_matrix)

Input: Adjacency Matrix

Output: Filtered Adjacency Matrix

```
1.  for all nodes  in adjacent_matrix repeat step 2 to 6
2.      row = column = node
3.      if ( adjacent_matrix[row] == 0 ) do
4.          check ( if adjacent_matrix[column] == 0 ) do
5.                  remove node;
6.      else check for next node;
7.  return f_matrix;
8 .  End
```

Compare function is used to detect potential clones. In this node to node comparison is made as for clones programs similar dependency should exist between programs. This function takes filtered adjacency matrix of both files and finds whether clone relation may exist between them or not.

Algorithm 3. Compare_function (f_matrix1, f_matrix2)

Input: Two filtered adjacency matrix

Output: Finds whether potential clones or not

```
1.  if (f_matrix1.nodes== f_matrix2.nodes)
2.      compare each row and column
3.                  if match found then print->potential clone
4.  else if (f_matrix1.nodes != f_matrix2.nodes)
5.       set min = min (f_matrix1.nodes, f_matrix2.nodes)
6.       for  i-> 0 to min && s->0 to min, repeat
7.            for j-> 0 to min&& t->0 to min, repeat
8.                 if (f_matrix1[i][j] ==f_matrix2[s][t] ) then flag++
 9.                   else j-- or t--  //according to smaller size matrix
10.            if (flag==min) then flag2++;
11.      Continue and check for each row
12.   if (flag2==min) then print potential clones
13.   else find clone_ratio.
14. End
```

Clone ratio can be defined as percentage of nodes matched. If all the nodes of code fragments are not matched then it tells how much percent of total code is considered as code clones. After finding potential clones, next step is to prove them actual clones. Hence, various metrics for these potential clones are calculated with the help following Count_Metric algorithm:

Algorithm 4. Count_Metric (Class_Name,adjacent_matrix)

Input: Class name and generated adjacency matrix

Output:Array contain metrics values

```
1. Traverse adjacency_matrix
2.          if (adjacency_matrix[row][column]==2) then Cnodes++ ; ednodes ++;
            if (adjacency_matrix[row][column]==1) then Dnodes++;  ednodes ++;
3.          Find complexity;
```

4. Use Java reflection API to get method name & repeat step 4.1 to 4.3 upto(i< getdeclaredmethod.length)
    4.1. Use method.getReturntype to get return type
    4.2. Use method.getParamterType to find type of parameter passed
    4.3. Store method name, parameter type, return type, number of parameters
5. Use Reflection to get the Variable names and their acess modifier with the help of Field class. Repeat step 5.1 (i < fields.length)
    5.1. fieldName <- get the name of field

```
ClassVariableCounter++;
    if (fields[i].getModifiers()==2)
        privateV++;
    if (fields[i].getModifiers()==1)
        pubV++;
    if (fields[i].getModifiers()==4)
        protectedV++;
```

6. End

After collecting metrics of both files their values are compared with help of in built comparison function using Java Array class. If all the metrics values are equal then, they are actual clones otherwise not.

4.4 Working of HCDetector

The working of proposed tool starts with Adaption Phase i.e. by giving two java byte code files as input with the help of user. For this purpose startup page of HCDetector is created by using java frames. To choose files with the help of user, Java FileChooser function is added which allow selecting only .class files. Figure 4.7 shows the first page of proposed system. Input file is selected with the help of File buttons. File1 and File2 Button handle the fileChooser event and display the absolute path of java .class file on java text box. When both input files are given, then Clone Detection button handles the event which passed the file names to the function which generates Program Dependence Graph and adjacency matrix for both programs which are used for detection of potential clones in a program.

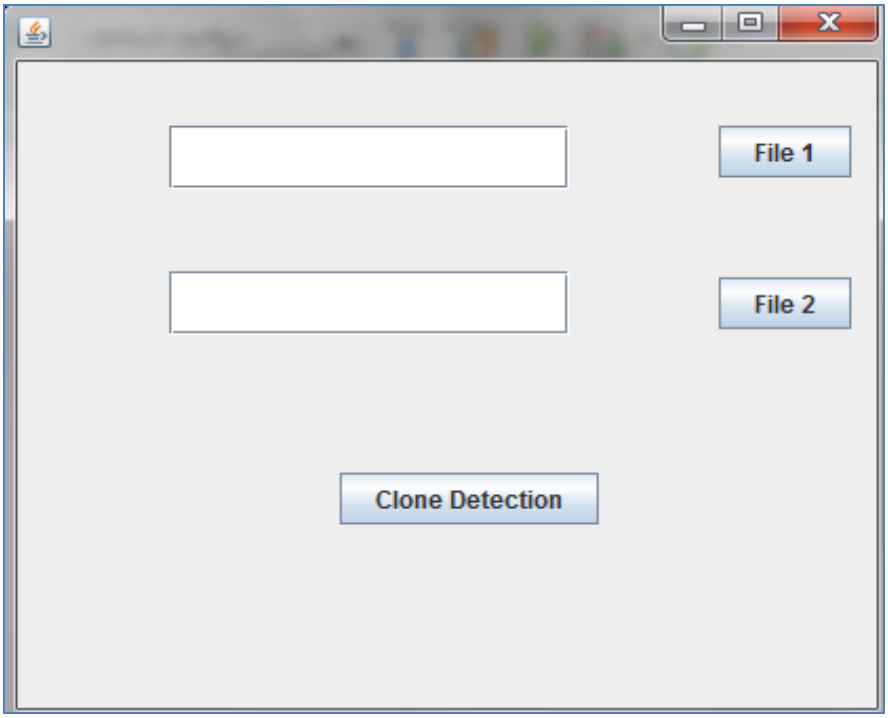


Figure 4.7: Choose File Screen of HCDetector.

When File 1 or File 2 button is clicked by user then open dialogue box is appeared to choose java .class files to find clones in system as shown in Figure 4.8 (i) and (ii)

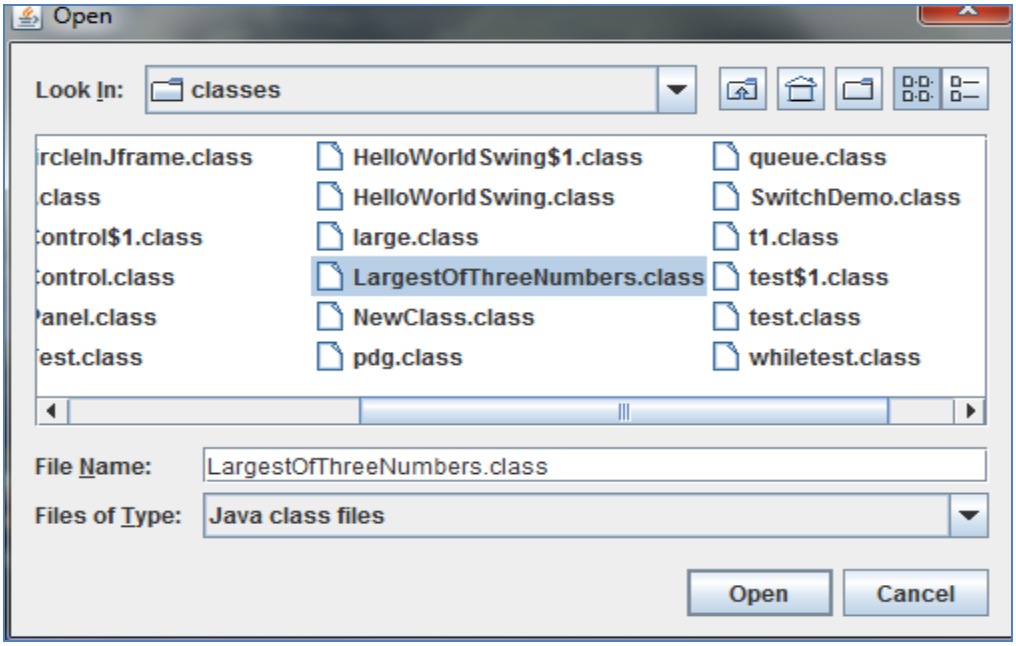


Figure 4.8 (i): File Selection Window for First File.

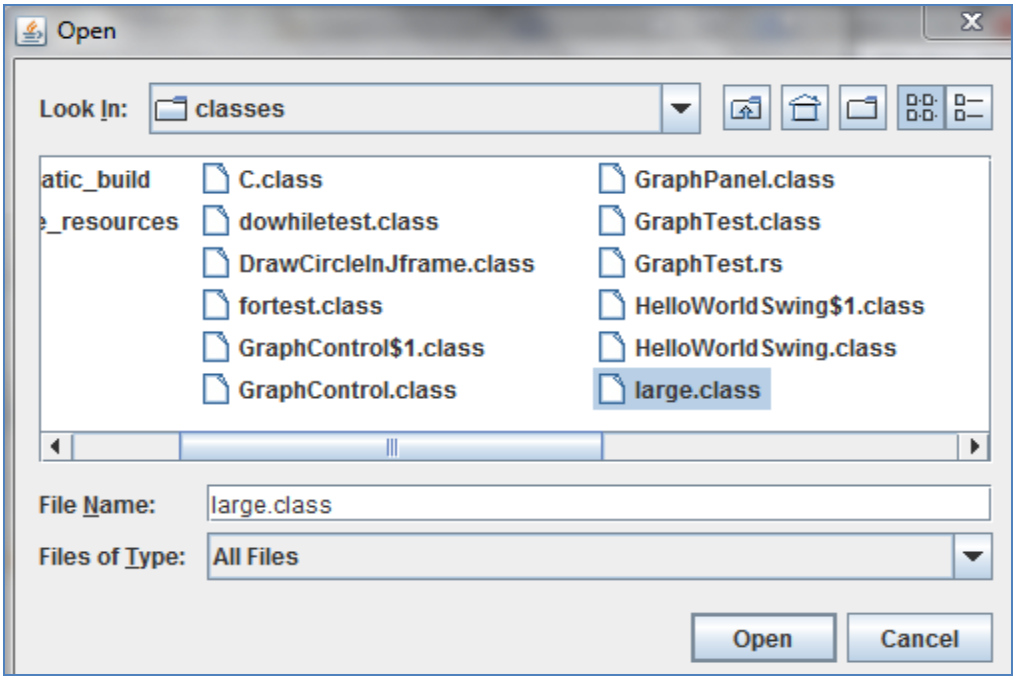


Figure 4.8 (ii): File Selection Window for Second File.

After selecting both files, there absolute path passed for PDG generation when clone detection button is clicked by user as in Figure 4.9.

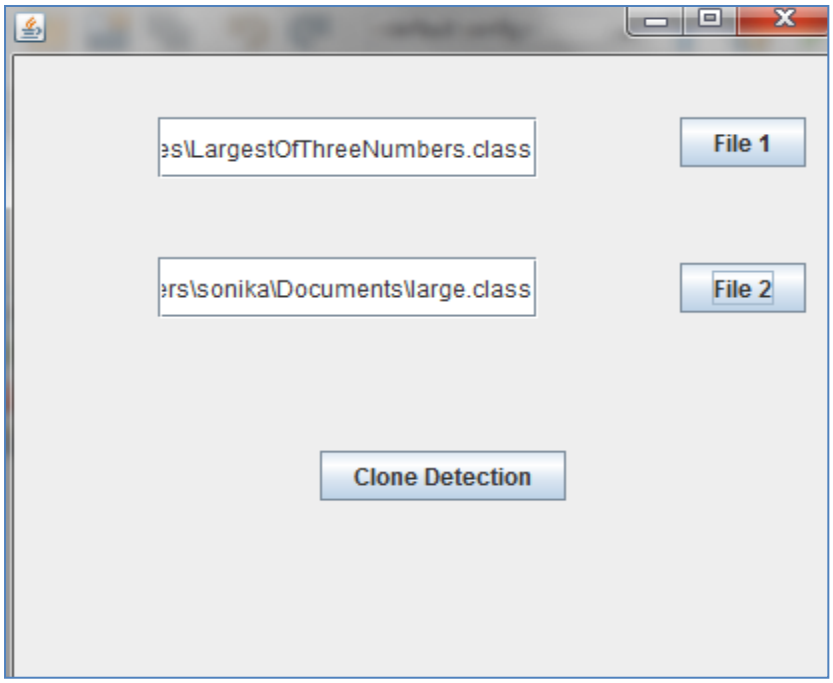


Figure 4.9: Main Window for Clone Detection.

Here we compare two codes large.class and largestofthreeNumbers.class which are of Type 2 i.e. renamed clones with the help of HCDetector. These codes find largest number among three numbers inputted by user. Java byte code for both files is shown in Figure 4.10. This code is conversion of high level language code into an intermediate language code which removes structural dissimilarity exists between programs and makes them in same format.

```
1. entry
2. args
3. System.out.println("Enter three integers ")
4. in = new Scanner.<init>(System.in)
5. a = in.nextInt()
6. b = in.nextInt()
7. c = in.nextInt()
8. If(a > c) && (a > b)
9. System.out.println("First number is largest.")
10. If not (a > c) && (a > b)
11. If(b > c) && (b > a)
12. System.out.println("Second number is largest.")
13. If not (b > c) && (b > a)
14. If(c > b) && (c > a)
15. System.out.println("Third number is largest.")
16. If not (c > b) && (c > a)
17. System.out.println("Entered numbers are not distinct.")
```

```
1. entry
2. args
3. System.out.println("Enter three integers ")
4. in = new Scanner.<init>(System.in)
5. x = in.nextInt()
6. y = in.nextInt()
7. z = in.nextInt()
8. If(x > z) && (x > y)
9. System.out.println("First number is largest.")
10. If not (x > z) && (x > y)
11. If(y > z) && (y > x)
12. System.out.println("Second number is largest.")
13. If not (y > z) && (y > x)
14. If(z > y) && (z > x)
15. System.out.println("Third number is largest.")
16. If not (z > y) && (z > x)
17. System.out.println("Entered numbers are not distinct.")
```

Figure 4.10: Java Byte Code of program to find largest number which is Type II Clone of Each Other.

Now transformation phase is started by generating output in the form of program dependence graph and adjacency matrix for both code fragments. In order to create user friendly interface, Java Swings are used to display PDG where red lines represent data dependency whereas black lines represent control dependency. Figure 4.11 represents the PDG of large.class and figure 4.12 represents PDG of largestofthethree.class

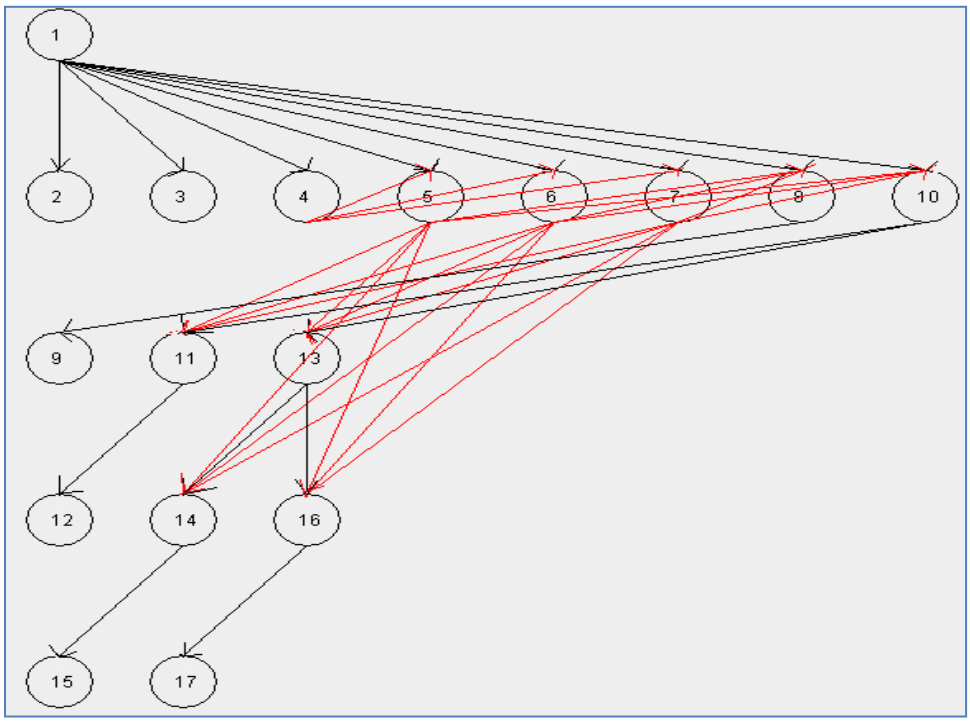


Figure 4.11: Program Dependence Graph of large.class.

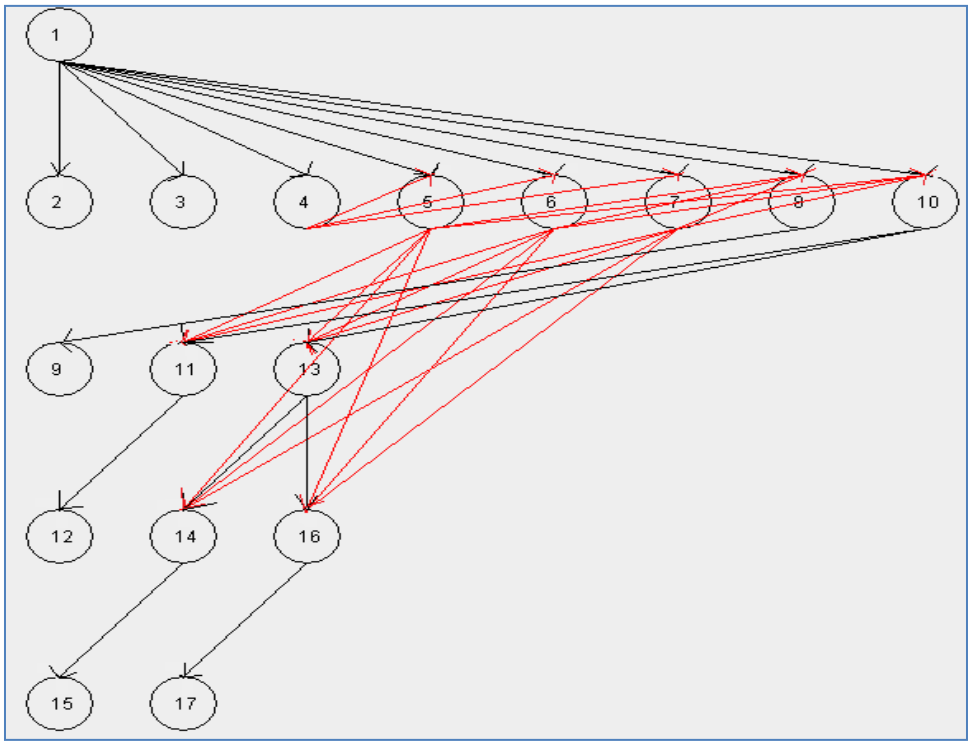
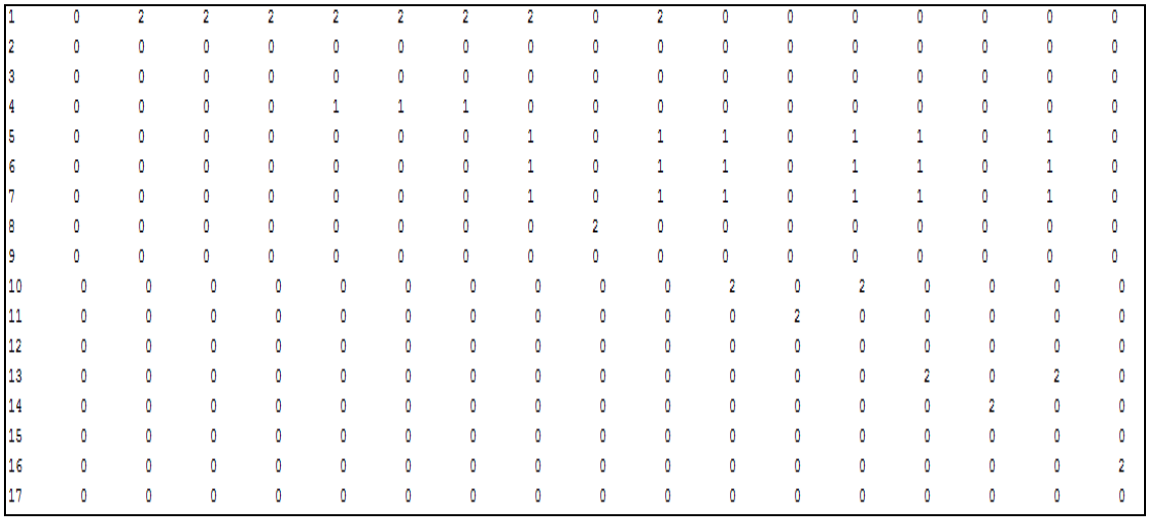


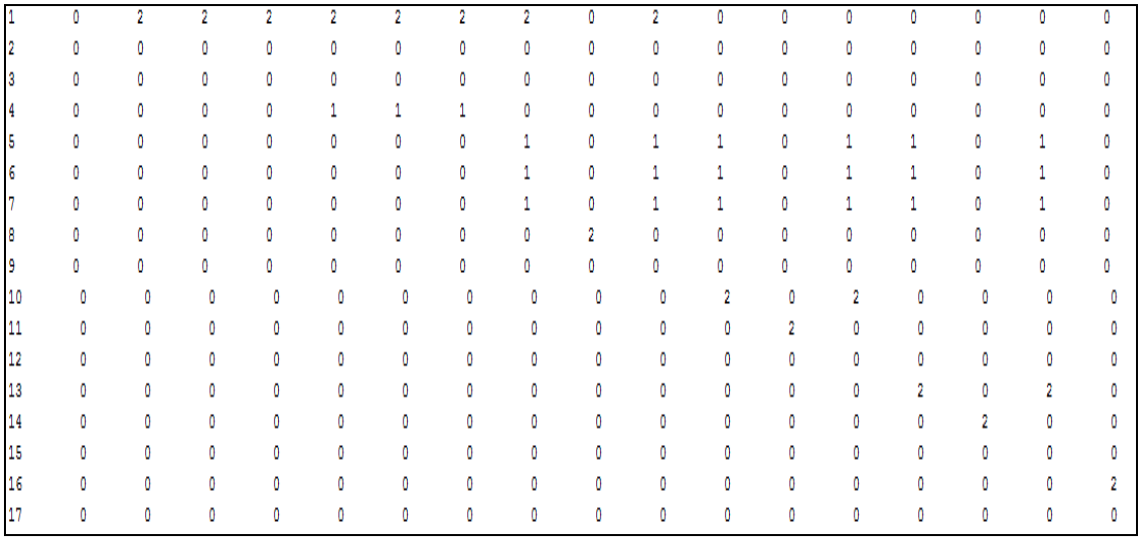
Figure 4.12: Program Dependence Graph of largestofthreeNumbers.class.

Figure 4.13 represents adjacency matrix for both programs whose PDG's are shown in Figure 4.11 and Figure 4.12



1	0	2	2	2	2	2	2	2	0	2	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
6	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
7	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
8	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	2	0	2	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

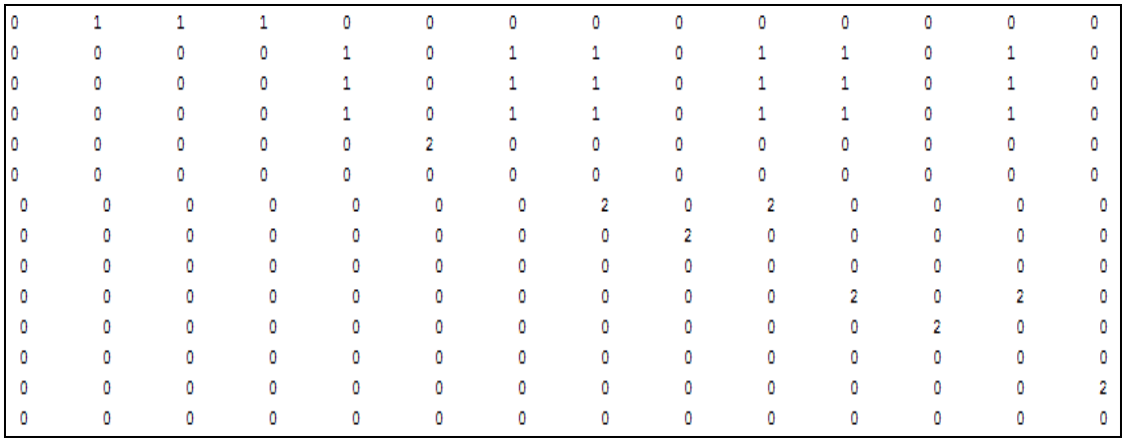
Figure 4.13 (i): Adjacency Matrix of File large.class.



1	0	2	2	2	2	2	2	2	0	2	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
6	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
7	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1	0
8	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	2	0	2	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

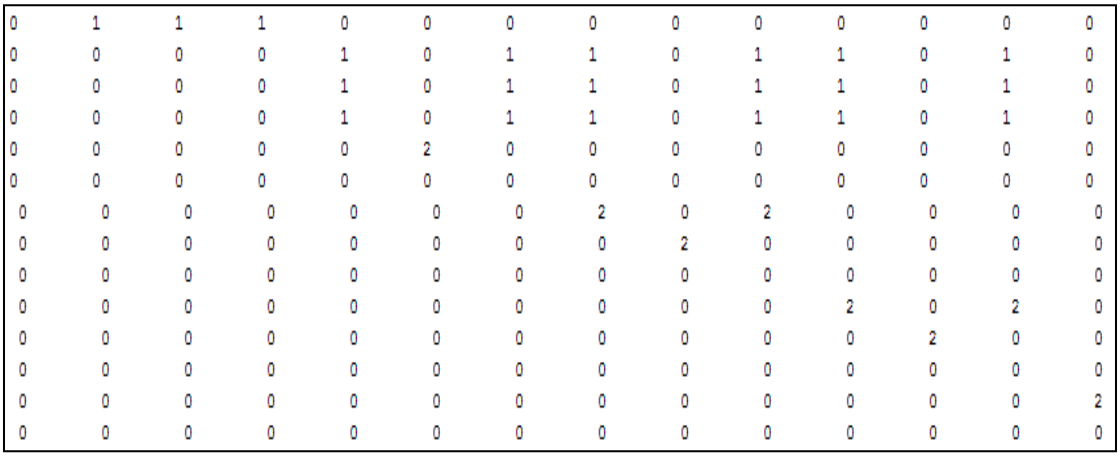
Figure 4.13 (ii): Adjacency Matrix of File largestofThreeNumbers.class.

These obtained adjacency matrices are filtered to remove independent nodes during normalization phase. Figure 4.14 shows the filtered adjacency matrices of both tested programs. Then this generated filtered adjacency matrices are compared with the help of compare function to show whether they are potential clones or not. This is called comparison phase of HCDetector.



0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	0	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	2	0	2	0	0	0	0
0	0	0	0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	2	0	2	0
0	0	0	0	0	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0

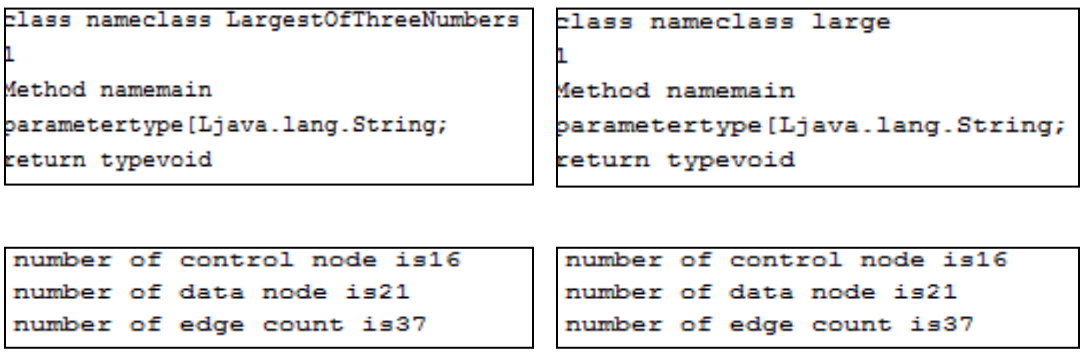
Figure 4.14 (i): Filtered Adjacency Matrix of Program large.class.



0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	0	0	0	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	2	0	2	0	0	0	0
0	0	0	0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	2	0	2	0
0	0	0	0	0	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.14 (ii): Filtered Adjacency Matrix of Program largestofThreeNumbers.class.

Now various object oriented metrics and control metrics are calculated in order to prove them as actual clones.



```
class nameclass LargestOfThreeNumbers
1
Method namemain
parametertype[Ljava.lang.String;
return typevoid
```

```
class nameclass large
1
Method namemain
parametertype[Ljava.lang.String;
return typevoid
```

```
number of control node is16
number of data node is21
number of edge count is37
```

```
number of control node is16
number of data node is21
number of edge count is37
```

Figure 4.15: Object Oriented and Control Metrics for both Tested Program.

These metric shown in Figure 4.15 are compared to prove them as actual clones. Hence in this way all the phases of proposed tool are followed by files. The Figure 4.16 describes phase by phase implementation of both files in order to detect actual clones.

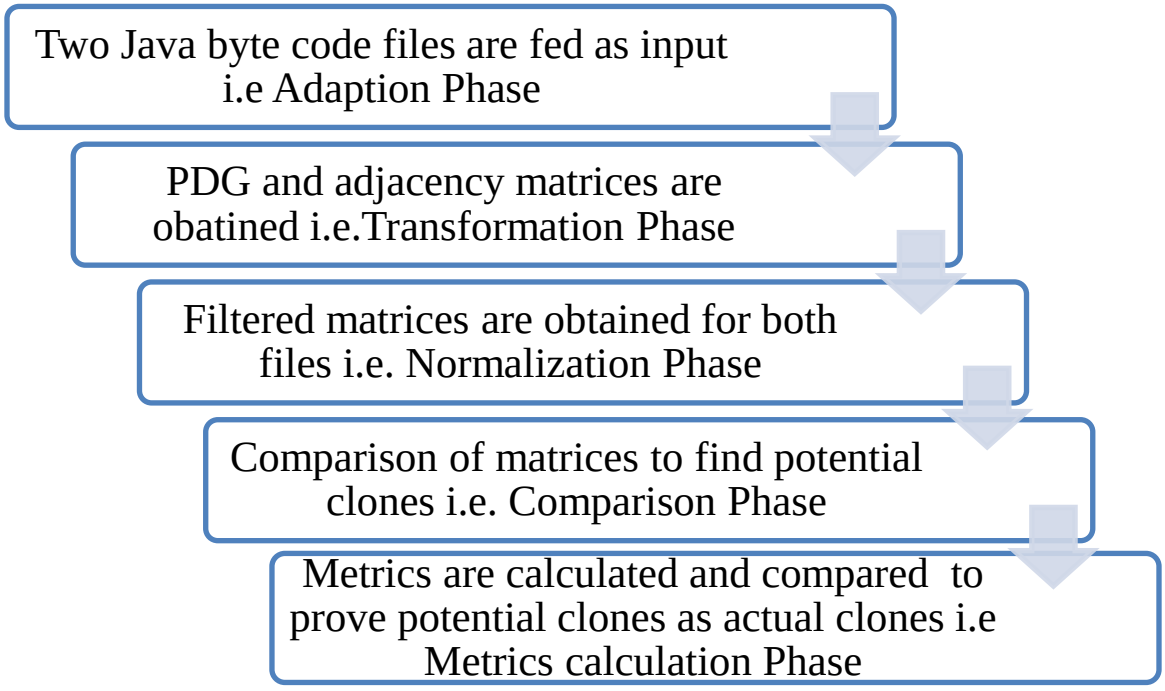


Figure 4.16: Phases Followed by Tested Program for Clone Detection.

Chapter 5

Results and Discussion

The tool proposed here is able to detect all types of clone efficiently. It does comparison on the basis of semantic similarity due to this change in layout, comment and identifier renaming does not affect the detection process. So this tool is able to detect Type I clone and Type II clone efficiently. Moreover, they are again verified by comparing obtained metrics. For Type III clones, insertion and deletion made on same statement that does not affect control and data dependency exist between statements can be easily detectable. As comparison is made on the basis of control and data relation that remain same in modified version of same program. However, when any data or control independent statement is added then it is already removed by filter function so system is able to detect Type III clones efficiently.

Type III clone detection can be illustrated with the help of following example: Two java .class files i.e. these files contains byte code of program, are entered as input to proposed system. Figure 5.1 and Figure 5.2 shows the byte code of file add.class and t1.class respectively. Here both files differ in position of control independent statement. Addition and reordering of this statement does not affect the flow of program and consider as type III clone.

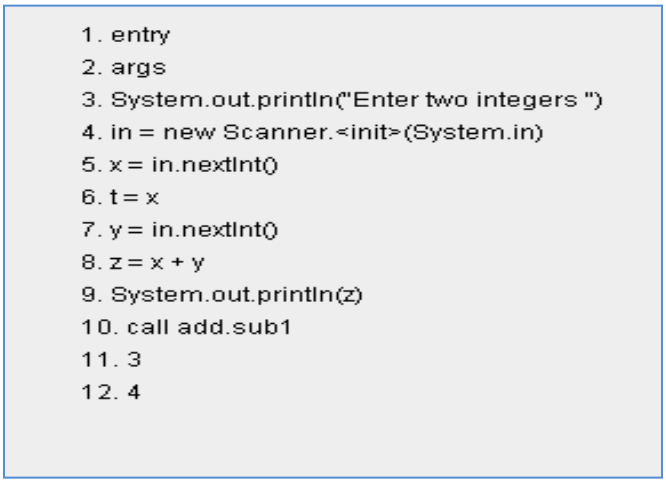


Figure 5.1: Java Byte Code of File add.class.

```
1. entry
2. args
3. System.out.println("Enter two integers ")
4. in = new Scanner.<init>(System.in)
5. x = in.nextInt()
6. y = in.nextInt()
7. z = x + y
8. t = x
9. System.out.println(z)
10. call t1.sub
11. 5
12. 2
```

Figure 5.2: Java Byte Code of File t1.class.

Now PDG of both files should be obtained with the help of function used to generate them. Figure 5.3 and Figure 5.4 represents PDG of file add.class and t1.class respectively. In PDG black lines represent control dependency and red lines represent data dependency exist among nodes.

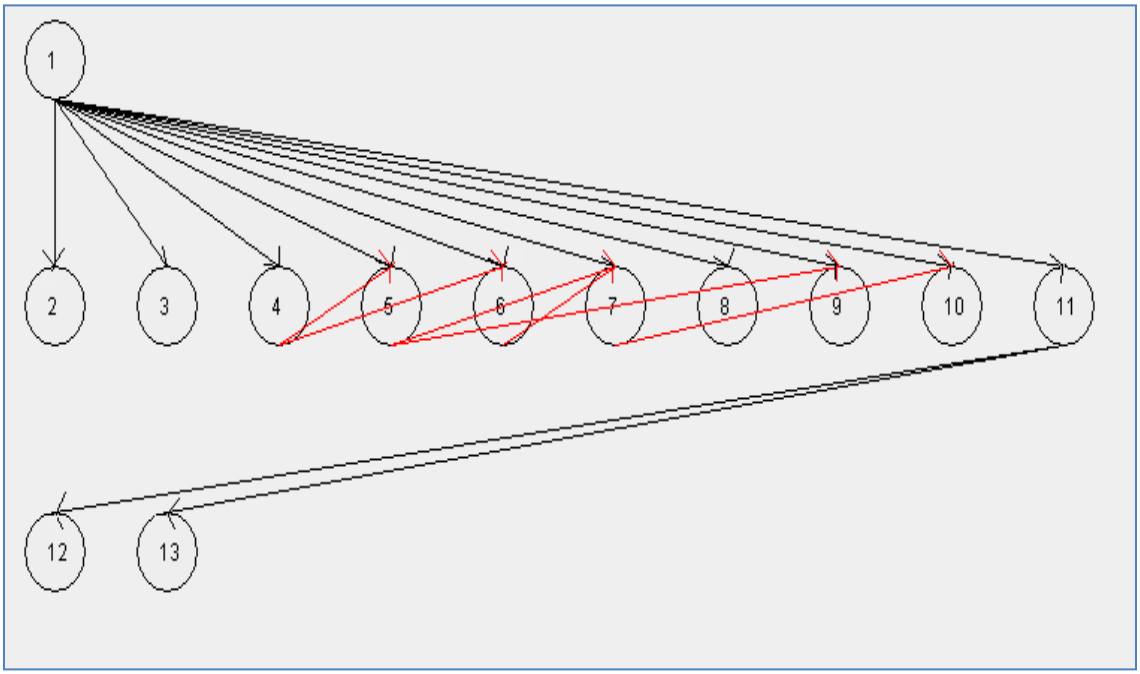


Figure 5.3: Program Dependence Graph of add.class.

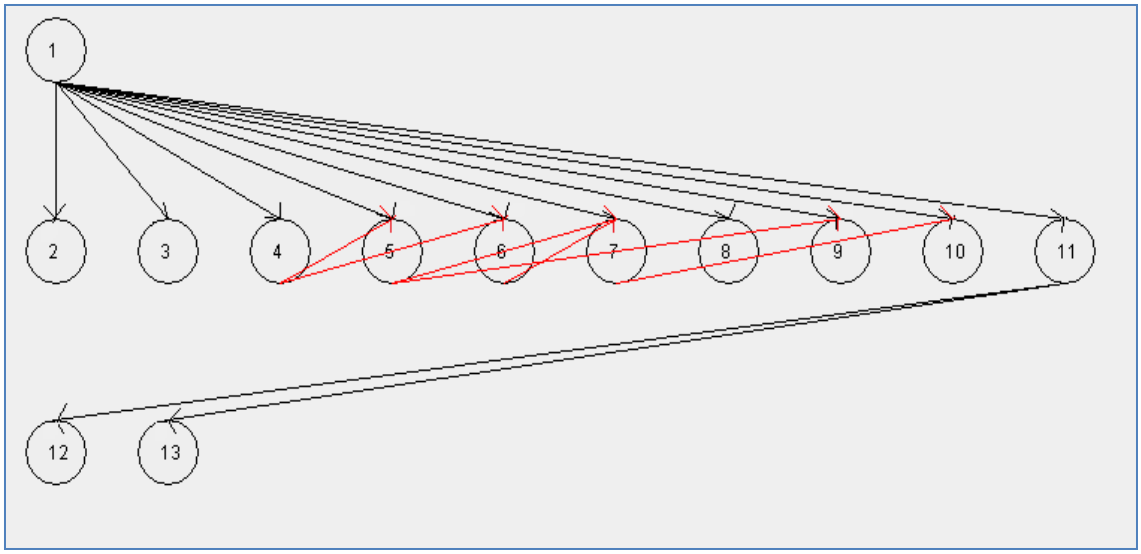


Figure 5.4: Program Dependence Graph of t1.class.

After obtaining PDG, adjacency matrix is obtained which represents data dependency with 1, control dependency with 2 and independent nodes with 0. Figure 5.4 and Figure 5.5 represents adjacency matrix of program add.class and t1.class.

node	1	2	3	4	5	6	7	8	9	10	11	12
1	0	2	2	2	2	2	2	2	2	2	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	2	2
11	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.5: Adjacency Matrix of File add.class.

node	1	2	3	4	5	6	7	8	9	10	11	12
1	0	2	2	2	2	2	2	2	2	2	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	2	2
11	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.6: Adjacency Matrix of File t1.class.

Now adjacency matrices are compared with the help of comparison function and after finding them as potential clones, various control and object oriented metric values are calculated. Figure 5.7 shows value of various control metrics and Figure 5.8 shows value of various object oriented metrics calculated from both file.

```
number of control node is11
number of data node is6
number of edge count is17
complexity is7
```

```
number of control node is11
number of data node is6
number of edge count is17
complexity is7
```

Figure 5.7: (i) Control Metrics of file add.class (ii) Control Metrics of file t1.class (Type III Clone).

```
class nameclass add
2
Method namesub1
parametertypeint
parametertypeint
return typevoid
Method namemain
parametertype[Ljava.lang.String;
return typevoid
field namepublic int add.u
field namepublic int add.v
field nameprotected int add.h
field nameprivate int add.i
```

```
class nameclass t1
2
Method namemain
parametertype[Ljava.lang.String;
return typevoid
Method namesub
parametertypeint
parametertypeint
return typevoid
field namepublic int t1.u
field namepublic int t1.v
field nameprotected int t1.h
field nameprivate int t1.i
```

Figure 5.8: (i): Object Oriented Metrics of file add.class (ii) Object Oriented Metrics of file t1.class (Type III Clone).

These calculated values are compared with the help of comparison function. Table 5.1 shows value of control metrics which are compared in order to prove potential clones as an actual clones.

Table 5.1: Control Metrics Value for Tested Program (Type III Clone).

Name of Program	Control Nodes	Data nodes	Edge Count	Complexity
Add.class	11	6	17	7
T1.class	11	6	17	7

After comparing control metrics, object oriented metrics are calculated with the help of Java reflection API and compared their yield values to find similarity. Table 5.2 shows various metrics calculated at class level while Table 5.3 shows various metrics calculated at function level.

Table 5.2: Class Metrics Value for Tested Program (Type III Clone).

Name of program	Fan out	Total V	PublicV	PrivateV	ProtectedV
Add.class	2	4	2	1	1
T1.class	2	4	2	1	1

Table 5.3: Function Metrics Value for Tested Program (Type III Clone).

Name of program	Name of method	No. of Parameter	Type parameter	Return Type
Add.class	Main	0	Java.lang.string	Void
	Sub	2	int, int	Void
T1.class	Main	0	Java.lang.string	Void
	Sub	2	int, int	Void

From the generated metrics it is clear that both are actual clones even additional statement is added at different positions. Hence it supports reordering of any data independent and control independent statements.

Moreover code is working on java byte code so it removes semantic dissimilarity of code

and able to detect Type IV clones. This can be illustrated with the help of following example. Here are two code fragments to find factorial of a number one is by using for loop and second by using while loop.

```java
public class whiletest {

   public static void main(String args[]) {
      int x;
      int y = 5;
      int fact=1;
      x=1;

      while( x <= y ) {
          fact=fact*x;

         x++;
      }
       System.out.print("value of fact : " + fact );
   }
}
```

```java
public class fortest {

   public static void main(String args[]) {
       int x;
       int y=5;
       int fact=1;

      for(x=1 ; x <= y; x = x+1) {
        fact=fact*x;
        // System.out.print("\n");
      }
        System.out.print("value of fact : " + fact );
   }
}
```

Fig 5.9: Source Code to find factorial of number (i) using for loop and (ii) using while loop (Type IV Clone).

These codes are structurally different but this dissimilarity removes in java byte code. When these files are given as input to system PDG is obtained for both files. Figure 5.10 represent PDG of file whiletest and figure 5.11 represents PDG of file fortest.

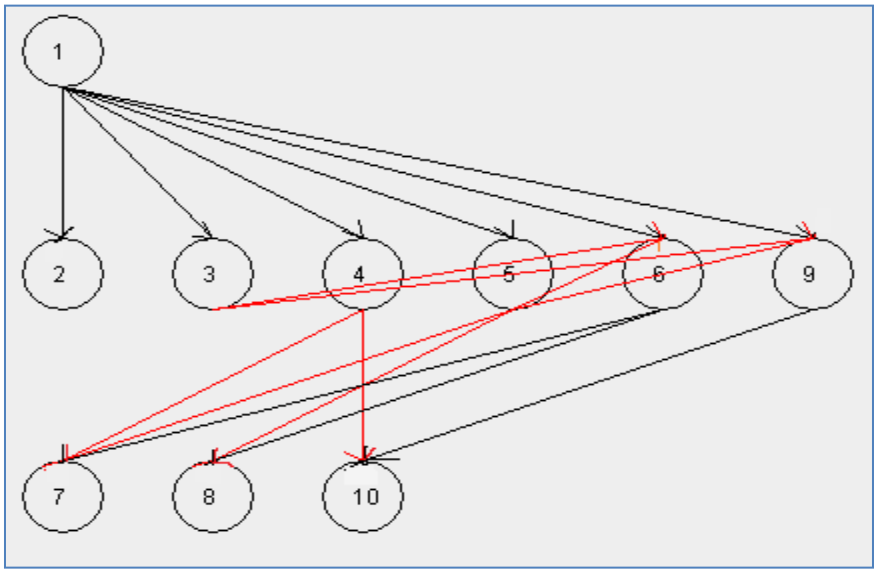


Figure 5.10: Program Dependence Graph of whiletest.class.

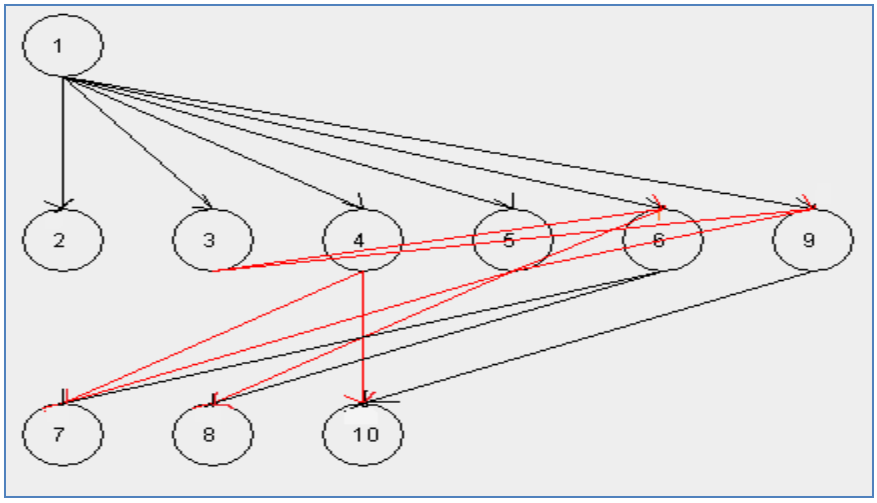


Figure 5.11: Program Dependence Graph of fortest.class.

Their generated matrices are shown in figure 5.12 from where it is clear that both are potential clones.

node	1	2	3	4	5	6	7	8	9	10
1	0	2	2	2	2	2	0	0	2	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	1	0
4	0	0	0	0	0	0	1	0	0	1
5	0	0	0	0	0	1	1	1	1	0
6	0	0	0	0	0	0	2	2	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	2
10	0	0	0	0	0	0	0	0	0	0

node	1	2	3	4	5	6	7	8	9	10
1	0	2	2	2	2	2	0	0	2	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	1	0
4	0	0	0	0	0	0	1	0	0	1
5	0	0	0	0	0	1	1	1	1	0
6	0	0	0	0	0	0	2	2	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	2
10	0	0	0	0	0	0	0	0	0	0

Figure 5.12: Adjacency Matrix of whiletest.class and fortest. class.

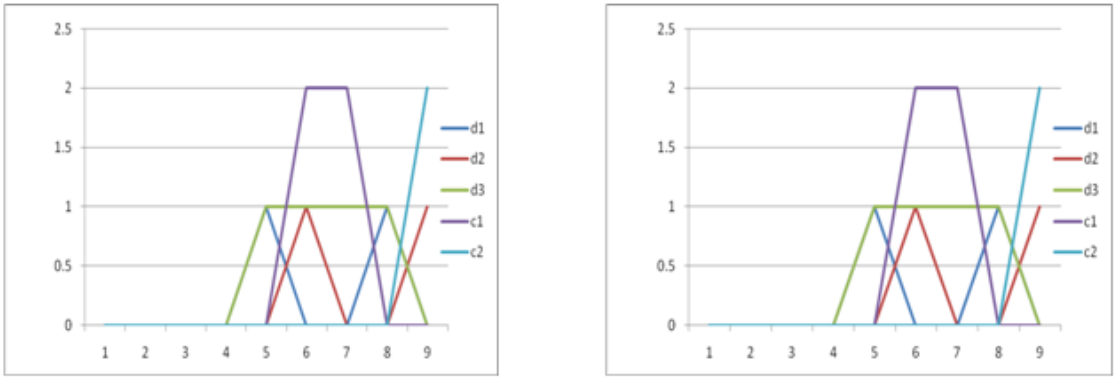


Figure 5.13: Line Graph Representation of Control and Data Dependencies of Program to Find Factorial of Number (i) using while loop and (ii) using for loop.

Figure 5.13 represents graph which shows control and data dependencies exist within statements of programs. Here X-axis represents nodes in PDG while Y- axis represent control and data dependency. However c1, c2, d1, d2, d3 represents effective control and data dependency after filtering independent nodes. From both the generated matrices and graph, it is clear that both are potential code clones.

In order to prove that both are actual clones figure 5.14 represents various control metrics and figure 5.15 represents object oriented metrics.

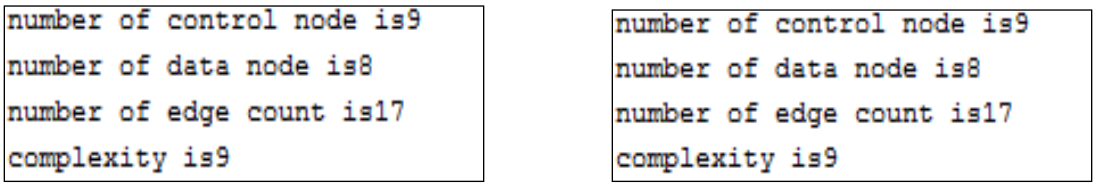


Figure 5.14: Control Metrics of whiletest and fortest.

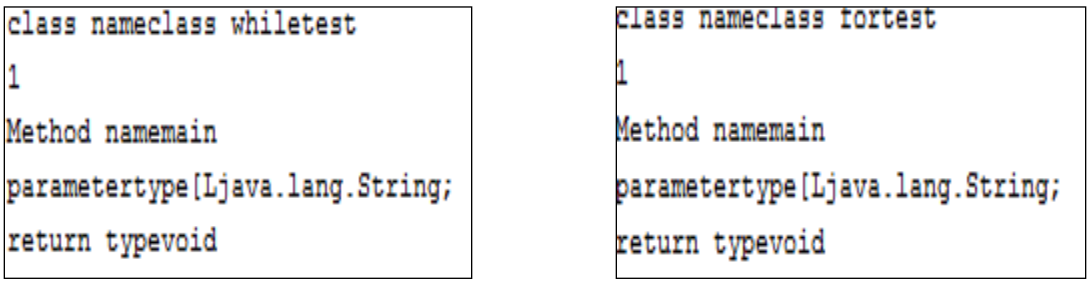


Figure 5.15: Object Oriented Metrics of whiletest and fortest.

These metrics are compared to find actual clones. Following tables represents control and object oriented metrics for tested programs.

Table 5.4: Control Metrics Value for Tested Program (Type IV Clone).

Name of Program	**Control Nodes**	**Data nodes**	**Edge Count**	**Complexity**
whiletest.class	9	8	17	9
fortest.class	9	8	17	9

Table 5.5: Class Metrics Value for Tested Program (Type IV Clone).

Name of program	**Fan out**	**Total V**	**PublicV**	**PrivateV**	**ProtectedV**
whiletest.class	1	0	0	0	0
fortest.class	1	0	0	0	0

Table 5.6: Function Metrics Value for Tested Program (Type IV Clone).

Name of program	**Name of method**	**No. of Parameter**	**Type parameter**	**Return Type**
Add.class	Main	0	Java.lang.string	Void
T1.class	Main	0	Java.lang.string	Void

From above tables it is clearly proved that both are actual clones. Hence this tool is capable of detecting semantic similar codes. However it does not support reordering of control and data dependent statements hence tool shows below average results when such type of clones present in program. Table 5.7 shows tool efficiency in terms of its capability to detect clones.

Table 5.7: Efficiency of Proposed Tool.

Clone Type	**Efficiency**
Type I	Good
Type II	Good
Type III	Average
Type IV	Below Average

Chapter 6

Conclusions and Future Scope

6.1 Conclusions

- The proposed tool is a hybrid approach tool which combines program dependence graph based clone detection technique with metrics based technique.
- Program dependence graph technique is used to find potential clones in system while metrics based technique is used to verify them as actual clones
- As PDG carries semantic information of system, hence proposed tool is able to detect both syntactic as well as semantic similar code clones.
- The proposed tool finds code clones only for programs written in Java language.
- This tool goes through five phases during its clone detection life cycle.
- Java byte code is given as input to the system as it removes all structural dissimilarities that exists in system and converts code fragments into unified code format.
- PDG is obtained with the help of Java System Dependence Graph API which is displayed in java frame with the help of Java Swings.
- Adjacency matrix is achieved with the help of Java System Dependence Graph API where data dependency among nodes represented by 1, control dependency by 2 and independent nodes by 0.
- These adjacency matrices are filtered to remove independent nodes and node by node comparison is made to prove them potential clones.
- Various object oriented metrics at class level and function level are calculated with the help of reflection API
- Various control metrics are calculated with the help of obtained PDG.
- The proposed tool compares these metrics values to find whether potential clones are actual clones or not.

6.2 Future Scope

- This approach is implemented only for java programs. In future it can be adapted for other languages like C++, C# etc. so that it become language independent.
- More metrics can be calculated with it in order to get more clear results.
- Efficiency of tool can be improved for type IV clone where reordering of control and data dependent statement is associated
- Calculated metrics can also used to rank code clones for efficient clone management.
- This tool can be further enhanced by using clone removal techniques after detecting actual clones.

.

References

[1] D. Rattan, Rajesh Bhatia, and Maninder Singh, "Software clone detection: A systematic review," Information and Software Technology, vol. 55, no. 7, 2013, pp 1165-1199.

[2] C. K. Roy and James R. Cordy, "A Survey on Software Clone Detection Research," Technical Report No. 2007-541, School of Computing Queen's University at Kingston Ontario, Canada, September 26, 2007.

[3] C. K. Roy, James R. Cordy, and Rainer Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," in Science of Computer Programming, May 2009, pp. 470-495.

[4] C. K. Roy, M. F. Zibran, R. Koschke, "The Vision of Software Clone Management: Past, Present, and Future (Keynote Paper)," in IEEE Conference of Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Software Evolution Week, 2014, pp.18-33.

[5] J Brooks, P Frederick," The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering", Pearson Education, 1995.

[6] C. J. Kapser, M. W. Godfrey, "Cloning considered harmful considered harmful: patterns of cloning in software," Empirical Software Engineering vol.13, No. 6, 2008, pp. 645-692.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and evaluation of clone detection tools," in IEEE Transactions on Software Engineering vol. 33, no. 9, 2007 pp. 577-591.

[8] S. Bellon, and R. Koschke, "Detection of software clones—tool comparison experiment," In International workshop on source code analysis and manipulation, Montreal, 2002.

[9] Manishankar Mondal, Md Saidur Rahman, Ripon K. Saha, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider, "An empirical study of the impacts of clones in software maintenance," In 19th International Conference on Program Comprehension (ICPC), 2011, pp. 242-245.

[10] C. Roy and J. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," In 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181.

[11] Rahman Foyzur, Christian Bird, and Premkumar Devanbu, "Clones: What is that smell?," In Empirical Software Engineering vol. 17, no. 4-5, 2012, pp. 503-530.

[12] Jiang, Zhen Ming, and Ahmed E. Hassan, "A framework for studying clones in large software systems," In Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2007, pp. 203-212.

[13] Koschke, Rainer, "Survey of research on software clones," In Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings, 2007.

[14] Krinke, Jens, "A study of consistent and inconsistent changes to code clones," In 14th Working Conference on Reverse Engineering (WCRE 2007), 2007, pp. 170-178.

[15] Yue JIA, "Clone Detection Using Dependence Analysis and Lexical Analysis," PhD Thesis, King's College London, 2007.

[16] J. Johnson, "Identifying redundancy in source code using fingerprints," in: Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 1993, pp. 171-183.

[17] J. Johnson, "Visualizing textual redundancy in legacy source," in: Proceedings of Conference of the Centre for Advanced Studies on Collaborative research, (CASCON), 1994, pp. 171-183.

[18] Seunghak Lee and Jeong Iryoung, "SDD: high performance code clone detection system for large scale source code," In Companion to the 20th annual ACM

SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, 2005, pp. 140-141.

[19] 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 *(ICSM'99)*, September 1999, pp. 109–118.

[20] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multi-linguistic token- based code clone detection system for large scale source code," in IEEE Transactions on Software Engineering, 2002, pp. 654-670.

[21] Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," Software Engineering, IEEE Transactions, vol. 32, March 2006, pp. 176-192.

[22] Baker, Brenda S, "On finding duplication and near-duplication in large software systems," In Proceedings of 2nd Working Conference on Reverse Engineering, IEEE, 1995, pp. 86-95.

[23] I. D. Baxter, A. Yahin, L. Moura, M. SantAnna, L. Bier, " Clone detection using abstract syntax trees," in Proceedings of the 14th International Conference on Software Maintenance (ICSM "98), Bethesda, Maryland, USA, 1998, pp. 368-378.

[24] Evans William S., Christopher W. Fraser, and Fei Ma, "Clone detection via structural abstraction," In Software Quality Journal, vol.17, no. 4, 2009, pp. 309-330.

[25] Jiang, Lingxiao, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. "Deckard: Scalable and accurate tree-based detection of code clones," In Proceedings of the 29th international conference on Software Engineering, Minneapolis, MN, USA, 2007, pp. 96-105.

[26] Johnson, J. Howard, "Identifying redundancy in source code using fingerprints," In Proceedings of the conference of the Centre for Advanced Studies on

Collaborative research: software engineering- IBM Press, vol. 1, 1993, pp. 171-183.

[27] J. Mayrand , Claude Leblanc, and Ettore M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," In Proceedings of International Conference on Software Maintenance, IEEE , 1996, pp. 244-253.

[28] R. Koschke, Raimar Falke, and Pierre Frenzel, "Clone detection using abstract syntax suffix trees," In 13th Working Conference on Reverse Engineering (WCRE'06), IEEE, 2006, pp. 253-262.

[29] Leitao Antonio Menezes, "Detection of redundant code using R 2 D 2," software quality journal, vol. 12, no. 4, 2004, pp. 361-382.

[30] N. Davey, Paul Barson, Simon Field, Ray Frank, and D. Tansley, "The development of a software clone detector," In International Journal of Applied Software Technology, vol.1, no. 3-6, 1995, pp. 219-236.

[31] A. Kostas Kontogiannis, Renator DeMori, Ettore Merlo, M. Galler, and Morris Bernstein, "Pattern matching for clone and concept detection," In Reverse engineering, Springer US, 1996, pp. 77-108.

[32] R. Komondoor, S. Horwitz, "Using slicing to identify duplication in source code," in Proceedings of the 8th International Symposium on Static Analysis (SAS' 01), Vol. LNCS 2126, Paris, France, 2001, pp. 40-56.

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

[34] Gabel, Mark, Lingxiao Jiang, and Zhendong Su, "Scalable detection of semantic clones," In 30th International Conference on Software Engineering, (ICSE'08), ACM/IEEE, 2008, pp. 321-330.

[35] C. Liu, C. Chen, J. Han, P. S. Yu, "GPLAG: Detection of Software Plagiarism by

Program Dependence Graph Analysis," In Conference on Knowledge Discovery and Data Mining, 2006, pp. 872-881.

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

[37] J. F. Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Lague, "Extending software quality assessment techniques to java systems," In Seventh International Workshop on Program Comprehension, IEEE, 1999, pp. 49-56.

[38] E. Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner, "Do code clones matter?," In Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 485-495.

[39] Keisuke Hotta , Yoshiki Higo, and Shinji Kusumoto, "Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph," In 16th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2012, pp. 53-62.

[40] J. Ferrante , K.J. Ottenstien , J.D. Warren, "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems,NY vol. 9, no. 3, July 1987, pp. 319-249.

Publications

Published/Accepted

[1] Surbhi Sonika and Rajkumar Tekchandani, "A Hybrid Approach to Detect Code Clones", In Proceedings of International Conference on Emerging Research in Computing, Information, Communication and Applications (ERCICA), India, August 2014.

[2] Surbhi Sonika and Rajkumar Tekchandani, "A Systematic Review and Comparison of Various Program Dependence Graph Based Clone Detection Tool", In Proceedings of Global Summit on Computer & Information Technology (GSCIT), IEEE, Tunisia, 2014.

Communicated

[1] Surbhi Sonika and Rajumar Tekchandani, "Program Dependence Graph Based Potential Clone Detection Tool", In Proceedings of Seventh International Conference on Contemporary Computing (IC3), India, 2014.