Enhancing Program Dependency Graph Based Clone Detection Using Approximate Subgraph Matching

A DISSERTATION
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR
THE AWARD OF THE DEGREE
OF

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY
C.M. Kamalpriya
Roll No. 14203020

UNDER THE SUPERVISION OF
Dr. Paramvir Singh
(Assistant Professor)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DR. B. R. AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY
JALANDHAR – 144011, PUNJAB (INDIA)
JULY, 2016
CANDIDATE’S DECLARATION

I hereby certify that the work, which is being presented in the dissertation, entitled “Enhancing Program Dependency Graph Based Clone Detection Using Approximate Subgraph Matching” by “C.M. Kamalpriya” in partial fulfillment of requirements for the award of degree of M.Tech. (Computer Science and Engineering) submitted to the Department of Computer Science and Engineering of Dr. B R Ambedkar National Institute of Technology, Jalandhar, is an authentic record of my own work carried out during a period from August, 2015 to July, 2016 under the supervision of Dr. Paramvir Singh, Assistant Professor. The matter presented in this dissertation has not been submitted by me in any other University/Institute for the award of any degree.

C.M. Kamalpriya
Roll No. 14203020

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

Dr. Paramvir Singh (Supervisor)
Assistant Professor
Department of Computer Science & Engineering
Dr. B. R. Ambedkar NIT, Jalandhar

The M.Tech (Dissertation) Viva-Voce examination of C.M. Kamalpriya, Roll No. 14203020, has been held on ____________ and accepted.

(Signature)  (Signature)  (Signature)
External Examiner  Supervisor  Head of Department
ACKNOWLEDGEMENTS

I would like to hereby express my sincere gratitude to my supervisor Dr. Paramvir Singh, Assistant Professor, Department of Computer Science and Engineering, DR B. R. Ambedkar National Institute of Technology, Jalandhar for the guidance, support and continuous engagement through the learning process of this master thesis. He motivated and encouraged me throughout this work. Without his encouragement and guidance this project would not have materialized. I consider myself extremely fortunate to have a chance to work under his supervision. In spite of his busy schedule, he was always approachable and took his time off to guide me and gave appropriate advice.

I also wish to thank whole heartedly all the faculty members of the Department of Computer Science and Engineering for the invaluable knowledge they have imparted on me. I also extend my thanks to the technical and administrative staff of the department for maintaining an excellent working environment.

I would like to thank my parents and all my family members for their continuous support and blessings throughout the entire process. I would also like to thank all my batch mates for the useful discussions, constant support and encouragement during whole period of the work.

Last but not the least, I would like to thank the almighty GOD for His blessings and for giving me enough strength to accomplish this phase of life.

C.M. Kamalpriya
Software Maintenance is one of the paramount processes in Software Engineering. It accounts for over sixty percent of the total effort and cost expended in the overall software engineering process. The main goal of software maintenance is to preserve the value of the developed software over time. During software development, programmers often reuse existing code to build new code. Replicated code fragment that is an exact copy or a modified version of an existing code fragment is called a code clone. Replicating existing code helps in quick software development and enhancement for changed user requirements. So, the occurrence of code clones is inevitable in the development of large software systems. But, cloning occurrences lead to higher cost and effort for software maintenance, increased probability of bug propagation, sloppier system design and increase in system size. Clone detection and removal are therefore, fundamental to efficient and effective software development and maintenance process. There are different types of code clones based on the amount and kind of replication performed.

Various tools and techniques have been developed to detect clones from software systems. Each clone detection tool or technique specializes in detection of one or more type of clones. Program Dependency Graph (PDG) based clone detection techniques have a key advantage over other techniques, that they are capable of detecting non-contiguous code clones. This work proposes further enhancement to PDG-based detection in order to identify all possible clone relations from the obtained clone results by applying Approximate Subgraph Matching (ASM). The results of the proposed technique were obtained on three subject software systems. The obtained results are composed of many new subsumed clone relations and exact and approximate clone relations derived from the clone pair results of PDG-based technique. These results indicate that using the proposed approach, a large number of new clone relations can be identified from the clone pairs obtained by PDG-based detection. The results have been manually validated for each subject system. This work also presents a novel approach using ASM to identify different node-to-node mappings between code fragments of each detected clone relation and proposes a new ASM-based distance measure to quantify their similarity.
# TABLE OF CONTENTS

CANDIDATE’S DECLARATION i  
ACKNOWLEDGEMENTS ii  
ABSTRACT iii  
LIST OF PUBLICATIONS vii  
LIST OF FIGURES viii  
LIST OF TABLES x  
LIST OF ABBREVIATIONS xi

## CHAPTER 1

INTRODUCTION

1.1 Basic Types of Clones 1  
1.2 Causes of Software Cloning 2  
1.3 Advantages of Software Clones 2  
1.4 Disadvantages of Software Clones 3  
1.5 Clone Detection Process 3  
1.5.1 Intermediate Source Code Representation 5  
1.6 An Overview of Clone Detection Tools and Techniques 6  
1.7 Overview of Program Dependency Graph (PDG) Based Clone Detection 8  
1.8 Example Study of Approximate Subgraph Matching (ASM) Algorithm 8  
1.9 Research Motivation 10  
1.10 Research Objectives 11  
1.11 Thesis Organization 12
CHAPTER 5
RESULTS AND DISCUSSION

5.1 Clone Relations Obtained in Subject Systems 46
5.2 Comparison of Results with Scorpio 51
5.3 Validation of Obtained Results 52
  5.3.1 Results on Test System of Scorpio Tool 53
  5.3.2 Results on EIRC 59
  5.3.3 Results on Eclipse-ant System 61
5.4 Results Summary 63
5.5 Chapter Summary 63

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Conclusions 64
6.2 Future Work 65

References 67
LIST OF PUBLICATIONS

LIST OF FIGURES

Figure 1.1  Clone Detection Process  5
Figure 1.2  Approximate Subgraph Matching \((G_1, G_2)\)  9
Figure 4.1  Snapshot of Eclipse Platform  34
Figure 4.2  Scorpio – Invoke Slicing on Similar Node Pair  36
Figure 4.3  Scorpio – Perform Slicing on Input Node Pair  37
Figure 4.4  Output Snapshot of Scorpio Tool  38
Figure 4.5  Cloned Code Fragments Detected by Scorpio from its Test System  39
Figure 4.6  Snapshot of Output Clone Pair File Generated by Scorpio for its Test System  39
Figure 4.7  Example of Scorpio Generated PDG  40
Figure 4.8  Flow Diagram of Proposed Clone Detection Method  43
Figure 4.9  Representation of Output Format  44
Figure 5.1  Graphical Representation of Results on Test System of Scorpio Tool  48
Figure 5.2  Graphical Representation of Results on EIRC System  50
Figure 5.3  Graphical Representation of Results on Eclipse-ant System  51
Figure 5.4  Graphical Representation of Comparison of Results With Scorpio  52
Figure 5.5  Output Snapshot 1 – ASM Relation from Test System of Scorpio (distance < 0.5)  53
Figure 5.6  Cloned Code Fragments for Output Snapshot 1  55
Figure 5.7  Output Snapshot 2 – Clone Relation from Test System of Scorpio (distance = 0.0)  
Figure 5.8  Cloned Code Fragments for Output Snapshot 2  
Figure 5.9  Output Snapshot 3 – ASM Relation from Test System of Scorpio (distance > 0.5)  
Figure 5.10  Cloned Code Fragments for Output Snapshot 3  
Figure 5.11  Output Snapshot 4 - Clone Relation from EIRC System (distance = 0.0)  
Figure 5.12  Cloned Code Fragments for Output Snapshot 4  
Figure 5.13  Output Snapshot 5 - Clone Relation from Eclipse-ant System (distance = 0.0)  
Figure 5.14  Cloned Code Fragments for Output Snapshot 5
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Details</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Injective Matches ($G_1,G_2$)</td>
<td>10</td>
</tr>
<tr>
<td>Table 1.2</td>
<td>Candidate Matchings ($G_1,G_2$)</td>
<td>10</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Subject Systems</td>
<td>42</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Number of Clone References and Number of Clones Detected by Scorpio</td>
<td>43</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Results on Test System of Scorpio Tool</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Results on EIRC System</td>
<td>49</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Results on Eclipse-ant System</td>
<td>50</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Comparison of Results with Scorpio tool</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

AST       Abstract Syntax Tree
PDG       Program Dependency Graph
ESM       Exact Subgraph Matching
ASM       Approximate Subgraph Matching
EIRC      Eteria Internet Relay Chat (Eight IRC)
CHAPTER 1
INTRODUCTION

Software Maintenance is a very essential process in Software Engineering. There are four different types of software maintenance activities: corrective maintenance, adaptive maintenance, perfective maintenance or enhancement and preventive maintenance or reengineering [1]. In all types of maintenance activities, modifications need to be performed in the software code. For speeding up the software development process, sometimes already developed code is reused to build new code. A replicated code fragment that is an exact copy or modified version of an existing code fragment is called a code clone. The occurrence of code clones hinders maintenance activities and is a major reason behind introduction of bugs into the system. Code cloning is the process of replicating existing code to develop new code.

Code cloning facilitates quick software development and modification as per change requirements, enables code reuse in case of programming language limitations and minimizes the overhead of procedural calls. But, it also leads to higher maintenance cost and effort, increased probability of bug propagation, a sloppier system design, lesser system understanding, and larger system size.

During software testing and maintenance phase, any modifications required in the original code fragment have to be replicated in the copied code fragments also. Experiments by Baker suggest that 20-30% of large software systems contain cloned code [6]. So, code clone detection is fundamental to better system design and understanding and cost-cum-effort effective software maintenance.

1.1 Basic Types of Clones

There are four major types of clones: Type-1 (exact clones), Type 2 (renamed / parameterized clones), Type 3 (near miss clones) and Type 4 (semantic clones).

- **Type 1 (exact clones):** These are exactly identical program fragments except for differences in white spaces, layout and comments.

- **Type 2 (renamed/parameterized clones):** These are syntactically/structurally similar code fragments except for variations in identifiers, literals, types, white spaces, layout and comments.
• **Type 3 (near miss clones):** Program fragments that are copied and later modified come in this category. Modifications performed after replication include statement insertions, deletions and changing of identifiers, literals, types, whitespaces, layout and comments.

• **Type 4 (semantic clones):** Semantically/functionally similar program fragments that are not instances of textually copied code come under this clone type.

### 1.2 Causes of Software Cloning

Software cloning occurs because of the following reasons [42, 44]:

• **Limitations of programmers’ skills and time constraints of project completion:** Limitation in software programmers’ skill and design understanding is a major reason behind code cloning. Moreover, the hard time constraints and strict project schedule often leads developers to copy existing code and modify the copied code to design new code or to fulfill new requirements.

• **Lack of knowledge of the programming language used:** Any software programmer generally does not have knowledge and practice in every programming language. So, replicating already developed code is the immediate alternative rather than learning the programming language that is used for the system.

• **Difficulty in understanding large and complex projects:** Software projects are at times too large and complicated to understand. This often encourages new programmers to replicate and use already developed code.

• **Fear that new code may lead to new errors and bugs:** Since, existing code has already been tested and used, it is easy to replicate and modify it for performing new functionality, rather than developing entirely new code for every new requirement.

• **Delaying the restructuring of code:** Many a times due to time constraints of project completion, steps such as code restructuring, refactoring and code abstraction are skipped. This often leads to the formation of clones.

### 1.3 Advantages of Software Clones

Replication of existing code fragments has the following advantages:

• **Fast method to address change requirements:** In case of immediate user requirement to change software functionality, reusing existing code is a quick method to build new code.
• **Supports quick enhancement of existing functionality:** In case the programming language lacks reuse and abstraction, then cloning existing code is an easy alternative.

• **Avoid the overhead of procedural calls:** Copy code for similar functionality avoids the use of procedural calls and hence minimizes requirement of memory and processing resources for handling subroutine calls and returns.

1.4 **Disadvantages of Software Clones**

Code Clones lead to various issues in software maintenance and updation in case of changing requirements. Some of the disadvantages are stated as follows:

• **Increased Software Maintenance Cost:** Presence of code clones in software increases post implementation maintenance (preventive and adaptive) effort.

• **Increased Probability of Bug Propagation:** If a code fragment contains a bug and that fragment is replicated somewhere else in the system, then the replicated clone will also contain the bug.

• **Poor Design:** Code cloning leads to lesser use of design practices such as refactoring and inheritance resulting in poor design.

• **Bad Impact on System Understanding / Improvement:** Generally, the people who code a software system are not the ones who maintain it. So, copy-pasting of code complicates the design and reduces system understanding thereby hindering scope for improvements or modifications

• **Increased Resource Requirements:** Code cloning increases size of software system. This leads to more compilation / execution time and space requirements.

So, it can be observed that code clones have serious disadvantages and hence need to be detected and eliminated.

1.5 **Clone Detection Process**

A clone detector should compare every code fragment with other code fragment but this is computationally expensive. Hence, several measures are used to reduce the domain of comparison before actually performing comparing code fragments. After identifying potential cloned fragments, further analysis is required to detect actual clones. The basic steps in a clone detection process are shown in Figure 1.1.
• **Preprocessing:** In this first phase, the source code is partitioned and the domain of the comparison is determined. Three main steps are performed in this phase:
  - Source code unimportant to comparison is filtered out in this phase.
  - Remaining source code is partitioned into a set of disjoint fragments that may be in direct clone relation with each other.
  - Determine comparison units / granularity: Source units are further partitioned into smaller units depending on the comparison technique used by the tool.

• **Transformation:** If the comparison technique is not textual, the source code of the comparison unit is transformed to an appropriate intermediate representation for comparison.
  - Extraction: It transforms source code to the form suitable as input to the comparison algorithm. It involves the following steps:
    - Tokenization: Each line of the source is divided into tokens according to the lexical rules of the programming language. The tokens of lines or files form the token sequences to be compared. All whitespaces and comments between tokens are removed from the token sequences. CCFinder [21] and Dup [59] apply such tokenization on source code.
    - Parsing: In syntactic approaches, the source code is parsed to build parse tree or abstract syntax tree (AST) and the comparison algorithm searches similar subtrees. Metrics-based approaches use metrics of subtrees in parse tree of the source.
    - Control and Data Flow Analysis: Semantics-aware approaches generate PDGs from the source code. Some metrics-based approaches use PDG subgraphs to calculate data and control flow metrics.
  - Normalization: It is an optional step performed to eliminate superficial differences like differences in whitespace, commenting, formatting or identifier names. In addition, pretty printing and structural transformations may also be applied.

• **Match Detection:** The transformed comparison units are compared with each other to find matches. The output of match detection is a list of matches in transformed code which is represented to form a set of candidate clone pairs.

• **Formatting:** Source coordinates of each clone pair obtained from the comparison phase are mapped to their locations in the original source files.
- **Post-processing (Filtering):** Clones are filtered by using manual analysis to filter false positives and applying heuristics such as length, diversity, frequency of clones etc.

- **Aggregation:** Clones may be aggregated into clone classes by performing subsequent analysis.

![Figure 1.1: Clone Detection Process](image_url)

1.5.1 **Intermediate Source Code Representation**

First, the source code is pre-processed to remove any uninteresting parts (e.g. comments and blank lines). Then suitable transformation techniques are applied to the pre-processed code to
obtain an intermediate source code representation. Intermediate representation is used to extract important information based upon which comparison is done. Abstract Syntax Trees (ASTs) or parse trees, regularized tokens and Program Dependence Graphs (PDGs) are the most frequently used intermediate representations. Clone granularity defines the boundary of comparison to be performed. It is either fixed, e.g. function, class, etc. or free, e.g. number of statements. Different clone granularity levels apply to different intermediate representations. Selection of a match detection algorithm is a prominent issue in clone detection process. After a suitable source code representation and granularity level is decided, an appropriate match detection technique (algorithm) is applied to the units of source code representation. The output is a list of matches with respect to the transformed code.

1.6 An Overview of Clone Detection Tools and Techniques

After preprocessing the source code, it is converted into an intermediate representation. Intermediate representation extracts useful information from code based upon which comparison is performed. Abstract Syntax Trees (ASTs) or parse trees, regularized tokens and Program Dependence Graphs (PDGs) are some common intermediate representations. Clone granularity defines the boundary of comparison. It can be fixed, e.g. function, class, etc.; or free, e.g. number of statements. After a suitable source code representation and granularity level is decided, an appropriate match detection technique is applied. The output is a list of matches with respect to the transformed code. The common clone detection techniques are categorized as follows:

- **Text Based Clone Detection:** Textual clone detection techniques use little or no transformation / normalization on the source code before performing actual comparison. In most cases raw source code is used directly in the clone detection process. Two code fragments are compared with each other in the form of text / strings / lexemes and the similar code fragments identifies are reported as code clones.

- **Token Based Clone Detection:** In token-based techniques, tokens are extracted from source code by lexical analysis. Each set of tokens (at a chosen granularity level) is formed into a sequence. Token sequences are then fed into a suffix tree or suffix array based token-by-token comparison is done to detect clones. These are efficient in detection of Type 2 (parameterized clones). Lexical approaches are more robust to minor changes in code, such as formatting, spacing and renaming, than textual techniques.
- **Tree based Clone Detection**: Source code is transformed into an intermediate representation such as abstract syntax tree or parse tree from which clones are identified and mapped back on to the source code. Variable names, literal values and other leaves (tokens) in the source code are abstracted in the tree representation. This allows for more sophisticated detection of clones. They are effective in identifying Type 3 clones (involving insertions and deletions of code fragments). The limitation of these techniques is that they are highly time consuming.

- **Graph Based Clone Detection**: In these techniques, program dependency graph (PDG) is used as an intermediate source code representation. A program dependency graph stores information of the program structure and the data flow within a program. In a PDG, a node represents program element (statement) and an edge represents data or control dependency. Similar subgraphs are identified from the constructed PDG. These similar subgraphs are mapped back onto the original code to detect clones.

- **Model Based Clone Detection**: Model driven software development has turned to be an emerging area. Large models are developed using UML, Matlab/Simulink and domain specific modeling languages. The presence of duplicated sub structures in different types of models leads to occurrence of code clones. Model based clone detection techniques involve making use of models that are used to design or develop code to detect code clones.

- **Metrics Based Clone Detection**: Metrics-based techniques collect a number of metrics from code fragments and then compare metric vectors rather than code or ASTs directly. Metrics in the form of characteristic vectors are generated from subtrees in the abstract syntax tree of the source code and similarity of subtrees is identified on the basis of metric similarity.

- **Hybrid Clone Detection**: Such techniques are a combination of 2-3 above techniques to perform clone detection. These techniques use a combination of syntactic and semantic characteristics to detect code clones.

The clone detection results obtained varies from one tool / technique to another because the underlying definition as to how each tool / technique defines a code clone varies altogether. Many of the tools and techniques are limited to identifying only some types of clones.
1.7 Overview of Program Dependency Graph (PDG) Based Clone Detection

The proposed algorithm identifies new clone relations from the clone pair results of PDG-based detection. This section briefly discusses PDG-based clone detection.

In graph-based clone detection, Program Dependency Graph (PDG) is used as an intermediate source code representation. PDG is a directed attributed graph that represents dependencies between program elements (statements) of the source code. A node in a PDG represents a program element of the source code and an edge represents a dependency between two nodes. In conventional PDGs, there are two types of dependencies between nodes:

- **Control Dependency**: It exists from program statements \( s_1 \) to \( s_2 \) iff:
  a) \( s_1 \) is a conditional predicate (for e.g. if statement, switch statement) and
  b) the execution of \( s_2 \) depends upon the result of \( s_1 \)

- **Data Dependency**: It exists from program statement \( s_1 \) to \( s_2 \) iff:
  a) \( s_1 \) defines \( v \)
  b) \( s_2 \) references \( v \)
  c) there exists at least one execution path from \( s_1 \) to \( s_2 \) without any redefinition of \( v \)

PDG-based techniques are capable of detecting non-contiguous code clones as compared to other clone detection techniques. But, they have lower performance in detecting contiguous code clones. The Scorpio tool [11] developed by Higo and Kusumoto specializes conventional PDGs by introducing execution next-links in addition to control and data dependencies [2]. Execution next-links denote the execution order of program statements. The tool has therefore, better performance in detecting contiguous code clones.

1.8 Example Study of Approximate Subgraph Matching (ASM) Algorithm

The proposed algorithm identifies new clone relations from PDG-based detection results by using Approximate Subgraph Matching (ASM). This section discusses the basic concepts of ASM.

Consider the pair of graphs \( G_1 \) and \( G_2 \), shown in Figure 1.2, to be tested for ASM matching. The inputs provided to ASM are: \( \{G_1, G_2, distance\text{Threshold}, distance\text{Weights}\} \), where \( distance\text{Threshold} \) is the threshold of subgraph matching distance and \( distance\text{Weights} \) is an array consisting of three values: \( structure\text{Weight} \) (weight of the structural subgraph distance),
labelWeight (weight of the label subgraph distance) and directionalityWeight (weight of the directionality subgraph distance), which specify the approximation allowed in matching [10].

![Graph G1 and Graph G2](image)

**Figure 1.2: Approximate Subgraph Matching(G₁, G₂)**

As is clear from Figure 1.2, there are two approximate matching occurrences of G₁ in G₂. The first instance which is an exact match is marked by dark solid connecting edges. The second instance is an approximate match and is marked by dashed connecting edges. It has variations with respect to G₁ in the edge directionality of its constituent edge labelled C.

Figure 1.2 uses the numbering notation to label graph vertices and alphabetic notation to label edges. G₁ is the graph whose approximate subgraph matching instances are to be identified from G₂. Vertex 1 is assigned to be the start vertex of G₁ to perform ASM matching. Start vertex is not assigned for G₂, since more than one matching instance of G₁ could be present in G₂. ASM performs the following steps:

i. It computes shortest distance paths and corresponding shortest distance values for each pair of vertices in the subgraph (G₁) by applying Dijkstra Algorithm [61]. The shortest distance information is used in ASM to compute differences in the structure, vertex labels and edge directionality between the input graphs [10].

ii. As shown in Table 1.1, a map called Injective Matches is constructed which maps each vertex in G₁ to all possible matching vertices in G₂.

iii. All combinations of node mappings are obtained from Injective Matches from which a list of matching vertex maps called Candidate Matchings is constructed as shown in Table 1.2.
iv. Subgraph matching distance function \( subgraphDistance(G_1, G_2) \) is computed for each map of Candidate Matchings list [10]. If the computed distance value satisfies Equation (1.1), then map is considered to be an approximate subgraph matching occurrence of \( G_1 \) in \( G_2 \).

\[
0 \leq subgraphDistance(G_1, G_2) \leq distanceThreshold
\]  

(1.1)

v. Similarly, all matching occurrences and their corresponding distance values are identified.

Table 1.1: Injective Matches \((G_1, G_2)\)

<table>
<thead>
<tr>
<th>( G_1 ) Vertex</th>
<th>Matching ( G_2 ) Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2, 2'</td>
</tr>
<tr>
<td>3</td>
<td>3, 3'</td>
</tr>
</tbody>
</table>

ASM can generate more than one matching occurrence between a given pair of graphs. It is possible that more than one vertex-to-vertex mapping are found at the same subgraph matching distance.

Table 1.2: Candidate Matchings \((G_1, G_2)\)

<table>
<thead>
<tr>
<th>Map#</th>
<th>( G_1 ) Vertex</th>
<th>Matching ( G_2 ) Vertex</th>
<th>Map#</th>
<th>( G_1 ) Vertex</th>
<th>Matching ( G_2 ) Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>1</td>
<td>1</td>
<td>3)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td>2'</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td>3'</td>
</tr>
<tr>
<td>2)</td>
<td>1</td>
<td>1</td>
<td>4)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2'</td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td>3'</td>
</tr>
</tbody>
</table>

1.9 Research Motivation

In contrast to other techniques, Program Dependency Graph (PDG) based techniques are capable of detecting Type 3 clones especially non-contiguous code clones (cloned code fragments in which the matching elements are not contiguously placed in the source code). Komondoor and Horwitz presented a PDG-based detection technique using program slicing to identify isomorphic subgraphs from PDGs which represent clones [3]. So, it may be observed that identifying subgraph isomorphism and subgraph matching has a fundamental role in PDG-based detection.
For biomedical text mining, graphs are often used to represent data involving structures and sequences of proteins and genes etc. Many algorithms and techniques based on exact and approximate subgraph matching (ESM and ASM) have been developed for mining relations and events from biomedical data [8, 9 and 10]. It is possible to apply these techniques to perform subgraph matching in PDG-based clone detection.

Higo and Kusumoto proposed enhancements to increase the performance of PDG-based techniques in detection of contiguous code clones and to optimize the runtime [2]. They proposed certain specializations to conventional PDGs: the introduction of execution-next links to enable detection of contiguous code clones and the merging of directly connected equivalence nodes to control computational cost of clone detection, hence optimizing the run time. They presented detection heuristics to improve the recall factor: two-way slicing to detect clones that are not detected by only forward or backward slicing, applying size threshold for equivalence classes of PDG nodes to reduce the number of slicing points, and specifying a minimum size threshold for detected code clones. They implemented these enhancements in the Scorpio tool [11], a Java implementation of their proposed method which achieves a higher recall than earlier PDG-based techniques. The tool has been tested on Java projects.

This work proposes an algorithm to further enhance PDG-based detection by identifying new approximate clone relations from the obtained clone pair results. In this experimental work, the algorithm was tested on three subject systems- the test system used for Scorpio tool [14], the EIRC system [16] and the Eclipse-ant system [17] (of Bellon’s benchmark [15]). The algorithm was applied on the clone pair results obtained by Scorpio tool for the respective test systems. The cloned code fragments obtained from the subject systems have been manually validated. The proposed algorithm can be applied to the clone pairs detected from any software system using any PDG-based detection tool.

1.10 Research Objectives

This work presents a method to identify new approximate clone relations from the clone pair results of PDG-based detection, thereby enhancing PDG-based techniques. The objectives of this research work are:

**Objective 1:** To propose a novel approach that detects new clone relations from clone pair results of PDG-based detection by applying approximate subgraph matching.
Objective 2: To devise a novel ASM-based node-to-node mapping mechanism and a distance measure to quantify the similarity among the code fragments of each detected clone relation.

1.11 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents a literature review on the field of clone detection. It provides a detailed discussion of the various tools and techniques developed in each type of clone detection. It also discusses the subgraph matching techniques studied for this work.
- Chapter 3 discusses the generic methodology of the proposed approach. It gives an overview of the proposed algorithm and then discusses the algorithm in full detail.
- Chapter 4 discusses the experimental design and methodology – details of the platform and computing environment, tools used, subject systems, output format and output validation method followed.
- Chapter 5 discusses the experimental results. It compares the results obtained with that of Scorpio tool. It analyses the results and manually validates the clones detected using the proposed approach.
- Chapter 6 presents the conclusions, important applications and future scope of this thesis work.
- References section lists all references used for this work.
CHAPTER 2

RELATED WORK

This chapter discusses the related work in the field of clone detection. It describes the important literature reviews and surveys of the field. It elaborates on the various tools and techniques developed in each clone detection type. It discusses the related work in PDG-based clone detection in more detail. The chapter also presents the important subgraph matching techniques studied for this thesis work.

2.1 Related Surveys

Roy and Cordy [43] performed a comprehensive survey of the capabilities of all clone detection techniques available. Their work first presents a survey of techniques. Next, it evaluates the techniques on the basis of hypothetically created editing scenarios for different clone types. It analyzes the potential of each technique in accurately detecting clones created by the studied scenarios.

Morshed et al. [44] conducted a literature review of code clone analysis to improve software maintenance process. They present a study of the several methods proposed by researches to enhance code clone maintenance process. The work highlights the different perceptions and ideologies of researchers regarding code clones, their impacts and detection techniques on the basis of case studies. The review also elaborates on the pros and cons of code clones.

Mubarak-Ali et al. [45] systematically reviewed different code clone prevention approaches. Their review discusses different scenarios that lead to code clones. It summarizes the existing literatures on code clone in the context of clone prevention. It also discusses the strengths and limitations of each clone prevention approach.

Patil et al. [46] presented a systematic review of the literature on clone evolution. It discusses the works of 20 papers and concludes that though many benchmarked studies have been conducted on clone evolution, many disparities exist in their findings for e.g. regarding lifetimes of clone lineages and the persistence with which clones are modified during software evolution.

The most recent literature review of code clone detection by Rattan et al. [42] gives a detailed description and analysis of all clone detection tools and techniques. It presents an empirical
evaluation of all the tools / techniques presented. The review discusses all types of clones, intermediate source code representations and match detection techniques. It also describes the emerging techniques such as semantic clone detection and model based clone detection.

2.2 Clone Detection Tools and Techniques

There are seven types of clone detection techniques as follows:

2.2.1 Text Based Clone Detection

In text-based techniques, two code fragments are compared with each other in the form of text / strings / lexemes and similar pairs are identified as clone pairs.

- Johnson [18] applied a finger printing technique for comparison of source code. The experiment was done on a 40 megabyte source tree representing two releases of gcc compiler. The proposed approach detects similarities in source code which occur due to software cloning, movement of code between releases and code preprocessing.

- Ducasse et al. [19] developed a lightweight approach that uses simple string matching to detect a large amount of code duplication – no need of parsing unlike previous approaches. They implemented the technique as a language independent tool Duploc. The tool was tested on software systems of different sizes, built in four different programming languages.

- Roy and Cordy [7] presented a language specific parser-based lightweight clone detection technique NICAD which detects near-miss clones with high precision and recall based on an efficient line comparison technique. It uses agile parsing to enable user specified flexible pretty printing that removes noise and standardize formatting. It performs efficient flexible extraction of potential clones to be compared using island grammars.

- Wettel and Marinescu [47] proposed a technique which detects duplication chains consisting of a number of smaller sized exact clones. It uses an automated approach to recover duplication blocks, by composing small isolated duplicated fragments into larger and more relevant duplication chains. Its efficiency and scalability has been validated on several open source systems. DuDe is the supporting tool developed by them.

- Lee and Jeong [48] proposed a technique useful for detecting clones in large size systems. The technique generates index and inverted index for code fragments and their positions.
Then an n-neighbor distance algorithm is used to detect similar code fragments. They implemented a supporting tool named SDD (Similar Data Detection).

- Harukizaemon [20] developed the Simian clone detection tool. It is capable of detecting clones in different programming languages – it treats source code file as a plain text file if the programming language used is not recognized. The tool has been tested on a number of systems built in different programming languages. The tool delivers good performance for systems as large as 1.2 million lines of code.

2.2.2 Token Based Clone Detection

In token-based techniques, tokens are extracted from source code by performing lexical analysis. Each set of tokens is formed into a sequence. Token sequences are then fed into a suffix tree and suffix tree or suffix array based token-by-token comparison is performed to detect clones.

- Baker [59] developed a detection approach in which token sequences are fed into a suffix tree. The approach abstracts concrete values of identifiers and literals while maintaining their occurrence order; hence it detects more parameterized clones than text based techniques. It was implemented as the Dup tool.

- Kamiya et al. [21] applied suffix tree matching algorithm to find cloned subsequences. The algorithm was tested on open source systems. It also finds metrics that characterize the given subject system. Later, researchers developed D-CCFinder for large software systems and CCFinderX to focus on detection of Type-2 clones. The algorithm was implemented as the CCFinder tool.

- Li et al. [22] used data mining techniques to efficiently detect duplicated code in large software systems like operating systems, and detects copy-paste related bugs. It uses frequent itemset mining to detect bugs that occur due to presence of clones in a system. The tool CP-Miner was developed in support of the proposed approach.

- Sasaki et al. [49] designed a token based clone detection tool FCFinder that uses hashing to detect file clones (files which are copy-pasted across projects). The study found that 68% of FreeBSD Ports collection to be file clones. It is efficient in detecting file clones from many similar or related projects that are systematically introduced from base resources.
2.2.3 Tree based Clone Detection

Source code is transformed abstract syntax tree or parse tree from which clones are identified and mapped back on to the source code. They are effective in identifying Type 3 clones.

- Yung et al. [23] presented a simplified approach to parameterized clone detection using abstract syntax trees. The approach focuses on those cloned code fragments that can be converted to procedural calls. So, it becomes possible to find clones that are not detected by other techniques. The work proposes direct transformation of detected clones into procedural calls.

- Koschke et al. [24] proposed clone detection using abstract syntax suffix trees. The technique uses suffix trees to find clones in abstract syntax trees. The method is capable of detecting syntactic clones in linear time and space. It has been empirically compared with other techniques for performance comparison and has been validated on Bellon’s benchmark [15] for clone detection.

- Baxter [25] proposed a method that detects exact and near miss clones using hashing and dynamic programming. It uses a variant of well-known compiler method to detect common sub-expressions. The method finds exact tree matches. A number of adjustments are required to identify equivalent statement sequences, commutative operands and nearly exact matches. The method was implemented as the CloneDR tool.

- Jiang [26] proposed a method that builds abstract syntax tree from the source code and computes numerical vectors on the subtrees in the Euclidean space $\mathbb{R}^5$. It performs clustering of vectors to identify corresponding similar subtrees that can be mapped back on to the source code. Deckard tool was implemented in support of the proposed method.

- Nguyen [51, 52] proposed an incremental AST based detection technique. It constructs characteristic vectors from AST subtrees and uses locality sensitive hashing to detect clones. The technique was implemented as the ClemanX tool. The empirical evaluation of the tool on real-world software projects shows that it is very efficient, complete and precise. It is capable of working incrementally when the code changes.

- Gitchell and Tran [50] developed an approach which converts source programs to parse trees. The tool views parse trees as strings and applied longest common subsequence algorithm and dynamic programming to assess code similarity. Sim tool was developed in support of the proposed approach.
2.2.4 Graph Based Clone Detection

This subsection discusses the important graph based clone detection techniques that use program dependence graphs as source code representation in combination with some other clone detection approach to detect clones from a system.

- Jia et al. [5] proposed an approach to detect both contiguous and non-contiguous code clones. They applied modified string suffix algorithm in the first phase to detect Basic Clone Pairs (BCPs). Suffix is generated using token values from five contiguous source statements. Any two statements having same suffix form a BCP. Also, the data and control dependence information between statements is gathered in this phase. In the second phase, BCPs are enlarged to get Type-1 and Type-2 clone pairs of maximal size by using extend-backwards and extend-forwards functions and Type-3 clone pairs are detected by applying extend-control-back, extend-control-forward, extend-data-back, and extend-data-forward functions on the identified Type-1 and Type-2 clones.

- Gabel et al. [60] presented the first algorithm for scalable detection of semantic code clones. The algorithm first applies the Deckard tool which generates characteristic subtree vectors for all significant nodes in the abstract syntax tree of the source code and merges subtree vectors by moving sliding window along adjacent subtrees. Next, for each procedure, a finite set of significant PDG subgraphs that represent semantic clone candidates is identified. For every subgraph, an AST forest is constructed corresponding to which a complete set of characteristic vectors is computed using Deckard’s sliding window for merging vectors. Locality Sensitive Hashing (LSH) is applied on the vectors of generated AST forests to identify clones on the basis of vector clustering.

- Krinke [4] proposed the use of fine-grained PDGs as intermediate representation of source code. Fine-grained PDG is a hybrid of AST and traditional PDG. Since in fine-grained PDG, nodes are mapped one-to-one onto nodes of AST, the number of nodes is large. Maximal similar k-limited subgraphs are identified using induction by considering every pair of matching predicate nodes as start nodes. They have implemented this k-limiting runtime optimizing approach in the Duplix clone detection tool.
2.2.5 Model Based Clone Detection

Clones are detected from a given subject system by identifying similar substructures in software models like UML models.

- Liu et al. [37] proposed a method to detect duplication in sequence diagrams by converting a 2-dimensional sequence diagram to a 1-dimensional array. Then, the obtained 1-dimensional array is used to build suffix tree. Common prefixes are identified from the suffix tree in the form of reusable sequence diagram. These common prefixes are the obtained refactorable candidates. The study confirms that 14% duplication is present in sequence diagrams of sample industrial projects.

- The automatic detection of clones in models results in identification of potential domain specific library elements. Deissenboeck et al. [38] used ConQAT as an integrated framework to detect clones in Simulink/Matlab models. The tool CloneDetective, which is part of the ConQAT framework, represents the model as a normalized multigraph where labels are assigned to relevant blocks. Similarity between blocks is identified on the basis of a heuristic which performs a depth first search looking for matched pairs. After the detection process, clones are clustered based on set of nodes using union function.

- Pham et al. [39] presented two algorithms - escan and ascan to detect clones in models. These were implemented in the tool ModelCD. First, the model was pre-processed to form a parsed, labeled directed graph. escan was used to detect exact matching by using an advanced graph matching technique known as canonical labeling and ascan was used to detect approximate matching by counting vector of sequence of nodes and edges’ labels. This is an incremental technique because it generates candidate cloned sub graphs. Their tool ModelCD was compared with CloneDetective (which is included in the ConQAT framework) [38].

- Hummel et al. [40] presented an incremental algorithm for model based clone detection. Firstly, a Simulink/Matlab model is pre-processed by flattening the model into a directed multigraph. Then, the relevant edges and blocks are labeled. A clone index is created for all subgraphs of the same size. Canonical labels of all subgraphs in the clone index are computed and similar labels are hashed. Clone retrieval and index update are integrated for fast retrieval of clones. It has not been tested on large models.
Related Work

- Storrle [41] pioneered clone detection in all types of UML domain models. The proposed technique was based on model querying. XMI files are generated from UML domain models by using any of the UML case tools. These files are then transformed into Prolog files. A small model is input in the query and model matching is used to generate the output. The tool is can compare models from diverse sources including a variety of UML versions. The tool is naive and is still to be tested for different subject systems.

2.2.6 Metrics Based Clone Detection

Metrics in the form of characteristic vectors are generated from subtrees in the abstract syntax tree of the source code and similar subtrees are detected on the basis of metric similarity.

- Mayrand et al. [29] compared metrics obtained from an AST to detect clones. This clone proposed technique uses 21 function metric organized into four points of comparison. The metrics are computed from names, layout, expressions and control flow of functions. Each point of comparison is used to compare functions and determine their cloning level. The technique defines an ordinal scale of eight cloning levels. The levels range from an exact copy to distinct functions. CLAN tool was developed in support of the technique.

- Kontogiannis et al. [30] proposed detection of code clones using metrics extracted from an AST representation of source code. Dynamic programming is applied on source code lines using minimum edit distance to perform match detection.

- Perumal et al. [53] used metrics and fingerprinting technique to obtain clones in source code. They proposed a new technique to detect similar code blocks and quantify their similarity. Their technique can be used to find clone clusters and sets of code blocks within a user-supplied similarity. It detects similar clones using metrics for type 1, type 2 of clones.

- Kodhai et al. [54] proposed a metric-based approach combined with the textual comparison of source code for detecting functional clones in C source code. They have formulated various metrics and the values obtained were utilized during detection process. Compared to the other approaches, the method is less complex and performs clone detection more accurately and efficiently.

- Dagenais et al. [55] examined clone occurrences in a large set of object oriented software libraries and program. The work studies occurrence of clones in a large set of object
oriented software libraries and programs (totaling six lakh lines of code), and discusses the factors affecting the clone detection accuracy, and their frequency of occurrence.

- Patenaude et al. [56] proposed a method to detect metrics from source code on five themes namely- classes, coupling, methods, hierarchical structure and clones. Their method reports clone occurrences in medium and large Java software systems. The clone identification process uses Java metrics to determine similarity between methods in a given subject system.

- Yuan et al. [57] proposed the count matrix based technique CMCD that detects clones by creating count matrices that contain occurrence frequencies of variables in the code. Detecting similar code fragments using count matrices helps to detect many difficult clones.

### 2.2.7 Hybrid Clone Detection

Such techniques are a combination of 2-3 above techniques to perform clone detection.

- Cordy and Grant [31] use Independent Component Analysis (ICA) to analyze vector representation of software methods. Singular value decomposition is applied on method token matrix and then ICA is applied that points in the vector space corresponding to the input data. Similarity is detected based upon distance between the vectors.

- Chilowicz et al. [32] developed a technique for the detection of function clones in source code represented as call graph using suffix array and metrics. They have designed a new algorithm to detect similarity in source codes. The algorithm focuses on obfuscation (deliberate creation of complex code) with inlining and outlining of functions. The method is efficient against insertions, deletions and permutations of instruction blocks. It applies code factorization and uses adapted pattern matching algorithms.

- Zibran and Roy [58] proposed the k-difference hybrid suffix tree algorithm for the detection of exact as well as near-miss clones. They developed an IDE plugin that implements the proposed algorithm.

### 2.3 Program Dependency Graph (PDG) Based Clone Detection

This work enhances the results of PDG-based clone detection. So, in this section we discuss the related works in the area of PDG-based detection.
• Komondoor and Horwitz [3] applied program slicing on PDGs of procedures in a software system to identify isomorphic subgraphs that represent code clones. The nodes in the PDG represent program statements and predicates while edges represent data and control dependencies between the statements. Starting with each pair of matching nodes in two PDGs, two isomorphic subgraphs are identified by slicing backward in lock step to reach matching predecessors. Forward slicing is performed in case of matching predicate nodes, so as to reach matching control-dependent successors. So, either forward or backward slicing is performed depending upon the type of matching nodes. Their method can detect non-contiguous code clones, intertwined code clones and code clones in which matching program statements have been reordered. They have tested their clone detection tool PDG-DUP on open-source software systems in C language.

• To overcome the limitations of PDG-based techniques such as lower performance in detection of contiguous code clones and large running time, Higo and Kusumoto proposed specializations to conventional PDGs and some heuristics to enhance the clone detection process [2]. Their technique implemented in the Scorpio tool [11], improves the recall of PDG-based detection.

• Incremental code clone detection proposed by Higo and Kusumoto [27] applies clone detection in large software systems. This is inspired by incremental clone detection in line or token based techniques. Clone pairs are identified on the basis of edge level matching and a recursive algorithm identifies clones by analyzing only updated files and not all source files.

• Ammar Hamid [28] performed changes to Komondoor and Horwitz’ experiment [3] - detecting clones only in reachable procedures (to enhance accuracy) and uses only backward slicing (forward slicing used only to define refactoring strategy and not to detect clones).

2.4 Subgraph Matching Techniques

The following subgraph matching techniques have been studied for this work:

• Liu et al. [8] proposed the Exact Subgraph Matching (ESM) algorithm for dependency graphs using backtracking approach. The algorithm has been implemented in Java. It has
been successfully applied for the extraction of relations and events from biomedical data such as detection of protein-residue associations and protein-protein interactions.

- Tian et al. [9] developed Substructure index-based Approximate Graph Alignment (SAGA) algorithm that identifies approximate graph matching by relaxing graph differences such as node gaps, node mismatches and structural differences. It has been implemented in C++.

- Liu et al. [10] also later proposed Approximate Subgraph Matching (ASM) algorithm for dependency graphs. It takes two graph inputs, and considering the smaller one as subgraph and the larger one as graph, it identifies all approximate mapping instances of subgraph in the graph. It allows approximate matching on the basis of input subgraph distance threshold and the weights defined respectively for the differences in labels, structure and directionality between the subgraph and its matching instance in the graph.

- Tian and Patel [33] proposed TALE, a technique that performs approximate matching of large graph queries in large graph datasets. In this technique, graph structural information is put into a hybrid index structure. Matching is done starting with the important nodes of query graph (important in terms of degree centrality) and then the matching is extended to other nodes in the importance order.

- Zhang et al. [34] presented GADDI technique that finds matches of query graph in a large graph using structure distance based approach. The technique is optimized by the use of a dynamic matching scheme to minimize redundant calculations. A number of real and synthetic data sets have been used to evaluate the efficiency and scalability of the proposed technique.

- Han et al. [35] proposed a technique TurboISO that finds all matches of query graph in a data graph using two novel concepts: candidate region exploration and combine-and-permute strategy. Extensive experiments conducted with many real datasets show that TurboISO significantly outperforms all competitors by up to several orders of magnitude.

- Mendivelso [36] proposed a new approach to isomorphism in attributed graphs. The technique finds all matches of pattern graph in attributed data graph. It applies pattern graph linearization and performs matching of the linearized pattern graph with the data graph.
2.5 Chapter Summary

This chapter presents a detailed literature review of various works done in software clone detection and subgraph matching. Section 2.1 discusses popular literature surveys conducted in the field of clone detection. Section 2.2 explains various tools and techniques developed in each clone detection type. Further, Section 2.3 presents related work in PDG-based clone detection. Section 2.4 discusses the subgraph matching techniques surveyed for this study.
CHAPTER 3
PROPOSED APPROACH

This chapter presents the proposed approach and algorithm designed for identifying new clone relations from Program Dependency Graph (PDG) based clone detection results using approximate subgraph matching technique. The algorithm can be applied to the clone pairs obtained from a given subject system by applying any PDG-based detection tool (Scorpio tool [11] was used for this experimental work).

The first subsection of this chapter presents an outline of the main steps of the proposed algorithm. Next, the algorithm is discussed in complete detail. The last subsection discusses Approximate Subgraph Matching (ASM) algorithm [10] and the modifications performed in ASM source code [12] for this experimental work.

3.1 Algorithm Steps Outlined

The proposed algorithm (detailed in Section 3.2) obtains all code fragment combinations from the clone pairs detected by from a given subject system by PDG-based detection. For each combination, it constructs a pair of directed graphs and applies (ASM) [10] to identify approximate subgraph matching instances in the generated graph pair. On the basis of ASM results for all the code fragment combinations, the algorithm detects new clone relations. It also defines a novel ASM-based distance measure to represent code similarity in each detected clone relation.

In the experimental work, the initial clone pair results for a subject system were obtained by applying the Scorpio tool [11]. The details of Scorpio tool have been discussed in detail in the next chapter. Each clone pair detected by Scorpio represents a pair of code fragments termed as left and right code fragments which contain matching program elements (and corresponding matching PDG nodes).

The proposed algorithm (detailed in Section 3.2) works as follows:

- In lines 1 – 16 of the algorithm, a list Combinations is generated. It contains all initial clone pairs obtained by PDG-based detection. This has been done because clone pairs that are already identified by PDG-based detection need not be tested for ASM.
- In lines 17 – 93, the following steps are performed on each code fragment combination:
1) Obtain absolute path of the source file of left code fragment in path1.
2) Create an array list of PDG nodes nodesG1 for the left code fragment and compute the list size sizeG1.
3) If there exists any method entry type PDG node in the list nodesG1, then set flag1 and store label of the node in startNodeTextG1.
4) Create an array list of PDG edges edgesG1 which connect the nodes in list nodesG1.
5) Construct directed graph G1 composed of vertices and edges for the nodes in list nodesG1 and edges in list edgesG1 respectively. Add vertices and edges to G1 by performing the following steps:
   a) For each PDG node n, compute hash value nhash on the basis of its core type (or program element type) using the same method as used in the Scorpio tool. Add vertex v(nhash, getText(n)) to directed graph G1.
   b) For each PDG edge e, identify source vertex src and destination vertex dest from G1 corresponding to the source and destination PDG nodes of e respectively. Next compute hash value of the edge in ehash on the basis of its edge type. Add edge e(ehash, src, dest) to G1.
6) Repeat above steps 1) -5) to generate path2, nodesG2, startNodeTextG2, flag2, edgesG2, G2
7) Find if code fragments in the combination are the same (A – A type). Next, check if the combination has already been tested. In any such case, there is no need to test the combination again. Else, add it to Combinations list.
8) If the combination is added to Combinations list and the vertex counts (denoted as VC) of graphs G1, G2 satisfy Equation (3.1) then proceed further (input parameter p is specified in Section 3.2), else skip to next code fragment combination i.e. step 1) above.
   \[ |VC(G_1) - VC(G_2)| \leq \left\lfloor \left( \frac{p}{100} \right) \times \max(VC(G_1), VC(G_2)) \right\rfloor \] (3.1)
9) If the vertex count of G1 is less than or equal to that of G2 (G1 is to be considered as subgraph and its matching instances are to be found in G2), then perform the following steps on (G1,G2), else perform the steps on (G2,G1):
   a) If either flag1 or flag2 are not set (i.e. either of the lists nodesG1 or nodesG2 do not contain PDG method entry type node) then, assign startVertex randomly from
vertices in $G_1$ and put all the vertices of $G_2$ into the list $startVertices$. This is done to obtain all possible matching instances.

b) Else, find the vertex having same label as the string $startNodeTextG1$ and assign it as $startVertex$. Next, find all vertices labeled same as the string $startNodeTextG2$ and put them into the list $startVertices$.

c) Perform ASM matching to find matching instances of subgraph $G_1$ in graph $G_2$.

d) Store distance values of all identified matching instances in list $distanceList$ and find minimum distance value from them.

e) If atleast one matching instance is obtained, then the combination has an ASM relation. If there are more than one matching instances and if minimum distance matching (vertex mapping) occurs at 0.0 value, then the ASM relation is a clone relation.

➢ In lines 94 – 95 of the algorithm, all identified ASM relations are displayed and in the lines 96 – 97 all identified clone relations are displayed.

3.2 Proposed Algorithm

This section presents the proposed algorithm in complete detail. First, the input and output of the algorithm have been discussed. The output format has also been explained. Next the detailed steps of the algorithm have been presented.

Input

clonepairArray is the array of clone pairs detected by PDG-based detection tool from input dataset, $N$ is the length of clonepairArray, $distanceThreshold$ is the threshold of subgraph distance, $distanceWeights$ is an array consisting of three values: structureWeight (weight of the structural subgraph distance), labelWeight (weight of the label subgraph distance), directionalityWeight (weight of the directionality subgraph distance), p is the percentage value which defines which clone pairs are processed for approximate matching

Output

ASMRelations (a list of ASMRelation entries),

CloneRelations (a list of CloneRelation entries) – CloneRelation is an ASMRelation having minimum distance mapping at 0.0
**Output Format for ASMRelation or CloneRelation entry**

\{path1, firstLine1, lastLine1, path2, firstLine2, lastLine2\} where:

- \(path1, path2\): are the absolute paths of source files which contain Clone Relation
- \(firstLine1, lastLine1\): represent start and end line no. of the related clone in file \(path1\)
- \(firstLine2, lastLine2\): represent start and end line no. of the related clone in file \(path2\)

**Steps**

1. // add all existing input clone pairs into Combinations list
2. for index \(i\) from 0 to \((N-1)\) do
3.  // obtain information of left code fragment of clone pair at index \(i\)
4.  \(pathA = getPathA(clonpairArray[i])\) // absolute path of source file
5.  \(codeA = getLeftCodeFragment(clonpairArray[i])\)
6.  \(elementsA = getElements(codeA)\) // program elements in the code fragment
7.  \(firstA = getFirstElement(elementsA)\)
8.  \(firstLineA = getFirstLine(firstA)\)
9.  \(lastA = getLastElement(elementsA)\)
10. \(lastLineA = getLastLine(lastA)\)
11. // obtain information of right code fragment in clone pair at index \(i\)
12. \(pathB = getPathB(clonpairArray[i])\)
13. \(codeB = getRightCodeFragment(clonpairArray[i])\)
14. Repeat steps 6 – 10 to compute \(elementsB, firstB, firstLineB, lastB, lastLineB\)
15. \(C1 = createCombination(pathA, firstLineA, lastLineA, pathB, firstLineB, lastLineB)\)
16. addCombination(Combinations, C1)
17. // process each code fragment combination in clonpairArray and apply ASM
18. for index \(j\) from 0 to \((N – 2)\) do
19.  for index \(k\) from 0 to \((N – 1)\) do
20.   // Perform steps 21 – 48 for left code fragment of clone pair at index \(j\) to generate path1, \(nodesG1, startNodeTextG1, flag1, edgesG1, G1\)
21. \(path1 = getPathA(clonpairArray[j])\)
22. \(nodesG1 = getLeftNodes(clonpairArray[j])\)
23. \(sizeG1 = size(nodesG1)\)
24. \(flag1 = 0\)
25. for index \(m\) from 0 to \(sizeG1\) do
Proposed Approach

26. if (isPDGMethodEnterTypeNode(nodesG1[m])) then
27.     startNodeTextG1 = getText(nodesG1[m])
28.     flag1 = 1
29.     break
30.     edgesG1 = φ
31. for each node n in nodesG1
32.     for each edge e in getForwardEdges(n)
33.         if ((fromNode(e) ∈ nodesG1) and (toNode(e) ∈ nodesG1)) then
34.             edgesG1 = edgesG1 ∪ e
35. G1 = createDirectedGraph()
36. for each node n in nodesG1
37.     nhash = computeNodeHash(core(n))
38.     v = createVertex(nhash, getText(n))
39.     addVertex(G1, v)
40. for each edge e in edgesG1
41.     for each vertex v in getVertices(G1)
42.         if (getLabel(v) = getText(fromNode(e))) then
43.             src = v
44.         break
45.     Repeat steps 41 – 44 to identify dest vertex from G1
46. ehash = computeEdgeHash(type(e))
47. e = createEdge(src, ehash, dest)
48. addEdge(G1, e, src, dest)
49. Repeat steps 21 – 48 for left code fragment of clone pair at index k to generate path2, nodesG2, startNodeTextG2, flag2, edgesG2, G2
50. // Perform steps 51 – 88 on (G1, flag1, startNodeTextG1, G2, flag2, startNodeTextG2)
51. Allocate new memory to variables startVertex, startVertices
52. Perform steps 5 – 10 to compute code1, elements1, first1, firstLine1, last1, lastLine1 for left code fragment of clone pair at index j
53. Perform steps 5 – 10 to compute code2, elements2, first2, firstLine2, last2, lastLine2 for left code fragment of clone pair at index k
54. toTest = 1
55. if((path1 = path2) && (firstLine1 = firstLine2) && (lastLine1 = lastLine2))
56.     toTest = 0  // A – A type combination
Proposed Approach

57. \[ C_2 = \text{createCombination}(\text{path1, firstLine1, lastLine1, path2, firstLine2, lastLine2}) \]
58. \[ \text{int exists = 0} \]
59. for each combination \( C \) in Combinations
60. \[ \text{if}(C_2 = C) \quad \text{// compare every information in the combination} \]
61. \[ \text{exists = 1} \quad \text{// already processed combination} \]
62. \[ \text{if (toTest = 1 && exists = 0)} \]
63. \[ \text{addCombination(Combinations, C2)} \]
64. \[ \text{count1} = \text{getVertexCount}(G_1) \]
65. \[ \text{count2} = \text{getVertexCount}(G_2) \]
66. \[ \text{if (abs(count1 - count2) \leq \left( \frac{p}{100} \right) \times \max(count1, count2) \) && (toTest = 1) and (exists = 0)} \]
67. \[ \text{if (count1 \leq count2)} \]
68. \[ \text{// Perform steps 69 – 80 on (G1, G2) if G1 has smaller (or equal) vertex count compared to G2, else perform steps 69 – 80 on (G2, G1) \]
69. \[ \text{if (flag1 = 0 or flag2 = 0)} \]
70. \[ \text{startVertex} = \text{getRandomVertex(getVertices(G1))} \]
71. \[ \text{startVertices} = \text{getVertices(G2)} \]
72. else
73. for each vertex \( v \) in getVertices(G1)
74. \[ \text{if (getLabel(v) = startNodeTextG1)} \]
75. \[ \text{startVertex} = v \]
76. \[ \text{break} \]
77. for each vertex \( v \) in getVertices(G2)
78. \[ \text{if (getLabel(v) = startNodeTextG2)} \]
79. \[ \text{startVertices} = \text{startVertices} \cup v \]
80. \[ \text{asm1} = \text{createASM}(G_1, \text{startVertex}, G_2, \text{startVertices, distanceThreshold, distanceWeights}) \]
81. \[ \text{// store all mappings between G1, G2 along with their distances} \]
82. \[ \text{matching1} = \text{performMatching(asm1)} \]
83. \[ \text{distanceList} = \text{getDistances(matching1)} \]
84. \[ \text{minDistance} = \text{findMinimum(distanceList)} \]
85. \[ \text{if (matching1 is non-empty)} \quad \text{// approximate matching exists} \]
86. \[ \text{addCombination(ASMRelations, C2)} \]
87. \[ \text{if (matching1 is non-empty && minDistance = 0.0)} \quad \text{// matching at 0.0 distance} \]
Proposed Approach

3.3 Approximate Subgraph Matching (ASM) Algorithm and Modifications Performed in ASM Source Code

The ASM algorithm [10] finds if an event rule graph is approximately isomorphic to a subgraph of a sentence graph within the specified condition of subgraph matching distance threshold. In this section we briefly explain the main ASM algorithm. The subroutines used in this algorithm can be referred from [10]. As explained in the working example of Section 1.8, the algorithm computesInjective Matches map IM and Candidate Matchings CMS for the given subgraph-graph pair. It outputs the set MR composed of those candidate matchings in CMS which satisfy Equation (1.1) on the subgraph-graph pair.

Input
Dependency graph of a testing sentence $s$, $G_s = (V_s, E_s)$ where $V$ is the set of nodes and $E$ is the set of edges of the graph; a finite set of biological event rules $R = \{ r_1, r_2, ..., r_i, ... \}$, where $r_i = (e_i, G_{r_i})$. $G_{r_i} = (V_{r_i}, E_{r_i})$ is the dependency graph of $r_i$; a given subgraph distance threshold $t$

Output
$MR$: a set of biological event rules from $R$ matched with $s$ together with the injective mapping

Main algorithm
1. $MR \leftarrow \emptyset$
2. for all $r_i \in R$ do
3. $st_{r_i} \leftarrow \text{startNode}(G_{r_i})$ // startNode() finds the start node $st_{r_i}$ of the rule graph $G_{r_i}$
4. \( ST_s \leftarrow \{ s_t s_1 , s_t s_2 , \ldots , s_t s_j , \ldots \} \) \( ST_s \) : the set of start nodes of the sentence graph \( G_s \)

5. for all \( s_t s_j \in ST_s \) do

6. if matchNode(\( s_t r_i \), \( s_t s_j \)) returns FALSE then

7. // matchNode() checks if an injective match exists between two nodes

8. go to Line 5

9. else

10. \( IM \leftarrow (s_t r_i , s_t s_j ) \) \( IM \) : record injective matches between nodes in \( G_{r_i} \) and \( G_s \)

11. for all remaining nodes \( v_r \in G_{r_i} \) do

12. \( IM_{v_r} \leftarrow \phi \) \( IM_{v_r} \) : record injective matches between \( v_r \) and nodes in \( G_s \)

13. for all remaining nodes \( v_s \in G_s \) do

14. if matchNode(\( v_r , v_s \)) returns TRUE then

15. // matchNode() assesses if two nodes can be matched using node features

16. \( IM_{v_r} \leftarrow IM_{v_r} \cup \{ v_s \} \)

17. if \( IM_{v_r} \) is empty then

18. go to Line 5

19. \( IM \leftarrow IM \cup IM_{v_r} \)

20. \( CMS \leftarrow \phi \) \( CMS \) : record candidate node matching schemes in \( G_s \)

21. \( cms \leftarrow \phi \) \( cms \) : a candidate node matching scheme

22. \( CMS \leftarrow \text{combinMatching}(IM, CMS, cms) \)

23. // combinMatching() recursively generates all candidate node matching schemes in \( G_s \)

24. for all candidate matching \( cms_i \in CMS \) do

25. if subgraphDistance(\( cms_i , G_{r_i} , G_s \)) \leq t \) then

26. \( MR \leftarrow MR \cup \{ r_i \ with \ cms_i \} \)

27. return \( MR \)

**Modifications Performed in ASM Source Code**

For the convenience of applying ASM algorithm [10] in the proposed method, ASM source code [12] was modified as follows:

- The ASM input schema was modified to \((\text{subgraph}, \text{subgraphStartVertex}, \text{graph}, \text{graphStartVertices}, \text{subgraphDistanceThreshold}, \text{subgraphWeights})\).
• A function performMatching( ) was added. It is used in line 82 of the algorithm. It performs the following steps:
  a) Invokes computeSubgraphPairwiseShortestDistanceAndPaths( ) function on the subgraph parameter
  b) Generates ASM matching between subgraph and graph by invoking the getApproximateSubgraphMatchingMatches( ).

• The function isSubgraphSmaller( ) is eliminated since, the proposed algorithm itself checks which of the graphs on which ASM is to be performed has smaller (or equal) vertex count and considers it as the subgraph and the other one as graph.

• Vertex type is modified to have only two data variables - token and label where token is the hash value of the core type of the node for which the vertex is constructed and label is the corresponding node label.

• The matchNodeContent( ) function is modified to compare input nodes on the basis of their respective token values.

3.4 Chapter Summary
This chapter presents a detailed description of the proposed approach. Section 3.1 gives a brief overview of the proposed algorithm. Section 3.2 elaborately discusses the algorithm. Section 3.3 explains the ASM algorithm [10] and modifications made to ASM source code [12] for implementing the proposed method.
CHAPTER 4
EXPERIMENTAL DESIGN AND METHODOLOGY

This chapter discusses the experimental design and methodology of the implemented work. It first presents details of the platform and computing environment in which the implementation has been performed. It explains the details of Scorpio tool that has been used for obtaining input to the proposed algorithm. It also provides details of the subject systems on which the proposed approach has been tested. It explains the generic methodology of implementation of the proposed approach. Further, the chapter explains the output format of the results obtained in this work. It also briefly discusses the output validation method used.

4.1 Platform and Computing Environment

This section presents details of the platform and computing environment in which the implementation has been performed - such as development platform, programming language used and computer processor details.

4.1.1 Eclipse Platform

The implementation of the proposed algorithm was done on Eclipse Integrated Development Environment (IDE). Eclipse is the most popularly used IDE for Java programming. It contains a base workspace and an extensible plugin system to customize the computing environment. Eclipse is written mostly in Java and is majorly used for developing Java applications and projects, but it may also be used to develop applications in other programming languages with the help of plugins. In this experimental work, Eclipse version 4.3 has been used. Figure 4.1 shows a snapshot of the Eclipse platform.

4.1.2 Java Programming Language

Java is a high-level programming language developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. Scorpio tool [11] has been developed in Java. Since, the proposed algorithm obtains input from Scorpio’s results; the implementation of this work was also done in Java. Java Runtime Environment (JRE) version 1.6 was used for the implementation.
4.1.3 Computing Environment

The implementation of this work was done on a personal computer with an Intel Core i3 processor (processing speed of 1.80 GHz) and RAM space of 4 GB. The system has Windows 8.1 operating system (64-bit version) installed in it.

4.2 Scorpio Tool

In the experimental work, the initial clone pair results for a subject system were obtained by applying the Scorpio tool [11]. This section explains the working of Scorpio tool and its flowchart representation. It also explains the output format of Scorpio with an example. A visualization example of Program Dependency Graph (PDG) is also presented.

4.2.1 Working of Scorpio

The Scorpio tool works as follows:

- PDGs are constructed for all methods in all files of the input subject system. An array of PDG pairs is constructed from the PDGs.
- Create an initially empty output set ClonePairs.
For each PDG pair – (pdgA, pdgB):
  a. Find paths pathA, pathB respectively.
  b. Compute mapPDGNodetoPDGNodes, mapPDGEdgeToPDGEdges by using hash values.
  c. All edges with same hash value are grouped together to form sorted set sortedPDGEdges.
  d. An empty sorted set checkedNodePairs is constructed.
  e. For each array of PDG edges in sortedPDGEdges:
     i. For each pair of edges – (edgeA, edgeB) in sortedPDGEdges perform slicing on (edgeA_fromNode, edgeB_fromNode, pathA, pathB) to obtain ClonePair and add the obtained ClonePair to output set ClonePairs.
     ii. Perform Slicing on Input Pair of Nodes:
         ▪ Create empty sorted sets checkedNodesA, checkedNodesB.
         ▪ If the pair of nodes is already processed, then return ClonePair (pathA, pathB), else:
             ▪ Add nodeA to checkedNodesA and nodeB to checkedNodesB.
             ▪ Obtain forward and backward edges of both nodeA and nodeB.
             ▪ Perform slicing recursively to enlarge the clonepair in forward and backward direction (to get maximal clones).

4.2.2 Flowcharts of Scorpio’s Working
According to the steps explained, the constructed PDGs are obtained as input and for each PDG pair slicing is invoked on every similar node pair (see Figure 4.2). Next, actual slicing operation is performed on the input pair of nodes (see Figure 4.3).

4.2.3 Scorpio Output
Figure 4.4 is an output snapshot of the Scorpio tool. The tool generates PDGs of the methods contained in program files of the input subject system. It calculates hash values of the nodes and edges in each PDG. It detects clone pairs [2] and writes them to an output file according to its predefined output format (explained further in this section). The total elapsed time for clone pair detection is also displayed by the tool. The number of comparisons performed during forward and backward slicing operations is also displayed.
Start

Create Set: **ClonePairs**

Input: array **PDGs**, array **PDG_Pairs**, map**PDGtoPDGNodes**, map**PDGtoPDGEddges**, **Size_Threshold**

For each PDG_Pair in **PDG_Pairs**:

- pdgA = pair.left
- pdgB = pair.right
- pathA = get method unit path of pdgA
- pathB = get method unit path of pdgB

Using **mapPDGtoPDGNodes**, **mapPDGtoPDGEddges**
compute **mapPDGNodesToPDGNodes**, **mapPDGEddgesToPDGEdges**

Create sorted set **sortedPDGEddges** on basis of edge hash values

Create empty sorted set of node pairs **checkedNodePairs**

For each array of PDG edges in **sortedPDGEddges**
For each pair of edges in current array edgeA, edgeB

1. Create node pair: (edgeA_fromNode, edgeB_fromNode)
2. Create Slicing on (pathA, pathB, edgeA_fromNode, edgeB_fromNode, mapPDGNodesToPDGNodes, mapPDGEddgesToPDGEdges, checkedNodePairs)
3. Create ClonePair by invoking PerformSlicing on Slicing
4. If ClonePair size > **Size_Threshold**, add ClonePair to **ClonePairs**

More elements

Output: **ClonePairs**

End

**Figure 4.2: Scorpio - Invoke Slicing on Similar Node Pair**
a) **Scorpio - Output Clone Pair Format**

Scorpio detects clone pairs in the format: \( f_1, s_1, e_1, f_2, s_2, e_2, g_1, g_2 \) where:

- \( f_1, f_2 \) denote the absolute paths of the two files from which clone pair has been detected.
- \( s_1, e_1 \) denote the start and end line number of cloned fragment in file \( f_1 \).
- \( s_2, e_2 \) denote the start and end line number of cloned fragment in file \( f_2 \).
- \( g_1 \) indicates the set of line numbers in file \( f_1 \) that is not included in the detected clone pair.
- \( g_2 \) indicates the set of line numbers in file \( f_2 \) that is not included in the detected clone pair.
An example of cloned code fragments identified by the tool from one of the subject systems – test system of Scorpio [14] is shown in Figure 4.5. The clone pair detected by Scorpio for the cloned code fragments is:

{test/test017/AnnotationWriter.java, 244, 248, test/test017/AnnotationWriter.java, 261, 271, −, {264, 266, 268, 269, 270}}.

According to the clone pair format used by Scorpio, it is clear that the code fragment 244 – 248 in file AnnotationWriter has a cloned code fragment in lines 261 – 271 of the same file. As it can be seen in Figure 4.5, there are no clone gaps in the former code fragment, whereas lines {264, 266, 268, 269, 270} of the latter code fragment indicate clone gap.

b) Scorpio - Output Clone Pair File

Figure 4.6 shows a snapshot of the output clone pair file generated by Scorpio. The clone pairs are generated in the predefined output format discussed earlier in this section. The snapshot shows a set of five clone pairs detected by Scorpio from its test system [14]. Each clone pair contains information of the identified cloned code fragments.
Cloned Code Fragment 1

```java
243 int getSize() {
244     int size = 0;
245     AnnotationWriter aw = this;
246     while (aw != null) {
247         size += aw.bv.length;
248         aw = aw.next;
249     }
250     return size;
251 }
```

Cloned Code Fragment 2

```java
260 void put(final ByteVector out) {
261     int n = 0;
262     int size = 2;
263     AnnotationWriter aw = this;
264     AnnotationWriter last = null;
265     while (aw != null) {
266         ++n;
267         size += aw.bv.length;
268         aw.visitEnd(); // in case user forgot to call visitEnd
269         aw.prev = last;
270         last = aw;
271         aw = aw.next;
272     }
```

Figure 4.5: Cloned Code Fragments Detected by Scorpio from its Test System

test\test017\AnnotationWriter.java 152 154 test\test017\AnnotationWriter.java 164 166 - -
test\test017\AnnotationWriter.java 158 160 test\test017\AnnotationWriter.java 164 166 - -
test\test017\AnnotationWriter.java 193 197 test\test017\AnnotationWriter.java 203 207 194 -
test\test017\AnnotationWriter.java 203 211 test\test017\AnnotationWriter.java 215 222 204,208,209,210 219,220,221

test\test017\AnnotationWriter.java 244 248 test\test017\AnnotationWriter.java 261 271 -
264,266,268,269,270

Figure 4.6: Snapshot of Output Clone Pair File Generated by Scorpio for its Test System
4.2.4 PDG Visualization of Scorpio Tool

Scorpio tool generates Program Dependence Graphs (PDGs) for the methods in each program file. Figure 4.7 is a visualization example of the generated PDGs. Figure 4.7(a) is a code fragment for which Scorpio generated PDG is shown in Figure 4.7(b). Nodes are shown in elliptical shape and are associated with respective program line numbers of the code in Figure 4.7(a). Method entry type PDG node is demarcated with a diamond shaped node and PDG exit node is represented by dark solid lined node. Scorpio uses colored representation to distinguish these nodes [11]. Data Dependence edge is represented by a simple solid line associated with its data variable. Control Dependence edge is shown by a dotted line along with boolean value true or false. Execution Dependence edge is shown by a dark solid line between the connecting nodes.

```
1 package yoshikihigo.tinypdg.test.test001;
2
3 public class Test001 {
4    5       int method() {
6           int a = 10;
7           int b = 20;
8           int c = a + b;
9           return c;
10        }
11    }
```

Figure 4.7: Example of Scorpio Generated PDG
4.3 Subject Systems

Three subject systems were selected for the experimental work and analysis, the details of which are shown in Table 4.1. The proposed algorithm was applied on the clone pair results obtained from each subject system by Scorpio tool [11].

The Test System of Scorpio tool was selected as a subject system because for this experimental work, the clone pair results of Scorpio detection were used as the input for the proposed algorithm. So, it was very useful to test the algorithm on this system first. EIRC (Eight IRC) is the Eteria Internet Relay Chat software built in Java for Windows System. Eclipse-ant is a widely used Java tool for software and web development. It is a constituent subject system of Bellon’s benchmark [15]. This selection of subject systems made it possible to test the algorithm on systems varying from small to large size.

Table 4.1: Subject Systems

<table>
<thead>
<tr>
<th>Software</th>
<th>Short Name</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test System of Scorpio [14]</td>
<td>-</td>
<td>5836</td>
</tr>
<tr>
<td>Eight IRC – 1.0.3 [16]</td>
<td>EIRC</td>
<td>14360</td>
</tr>
<tr>
<td>Eclipse-ant [17]</td>
<td>Eclipse</td>
<td>34744</td>
</tr>
</tbody>
</table>

In Table 4.2, the number of clone references of each subject system (clones that should be detected as per Bellon et al. [15]) and the number of clone pairs detected by Scorpio [11] have been enumerated. The number of clone references is available only for Eclipse system because – it is a part of Bellon’s benchmark.

Table 4.2: Number of Clone References and Number of Clones Detected by Scorpio

<table>
<thead>
<tr>
<th>Software</th>
<th>Number of Clone References</th>
<th>Number of Clones Detected by Scorpio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test System of Scorpio</td>
<td>-</td>
<td>141</td>
</tr>
<tr>
<td>EIRC</td>
<td>-</td>
<td>301</td>
</tr>
<tr>
<td>Eclipse</td>
<td>30</td>
<td>653</td>
</tr>
</tbody>
</table>
4.4 Generic Methodology

The proposed clone detection method has been summarized in the flow diagram of Figure 4.8. First, the Scorpio tool was applied on a particular subject system (composed of all its program files). Scorpio generated its clone pair results. Then, the proposed algorithm was applied on these clone pairs obtained from Scorpio [11]. The algorithm generates all possible code fragment combinations from the input clone pairs of Scorpio. It applies ASM on each code fragment combination. All ASM relations are detected from the code fragment combinations. The ASM relations are filtered to find those relations which have 0.0 distance value. These ASM relations are the clone relations detected by the proposed method from the given subject system.

4.5 Output Format

This section defines the output format of the results obtained in this implementation. The output format is shown in Figure 4.9. The first line contains information of sizes (in terms of vertex count) of the graphs generated for the code fragment combination in a detected clone relation. The smaller graph (size $a$) is the subgraph whose ASM matching instances are found within the larger graph (size $b$). Next, the output displays the number of identified subgraph matching occurrences (say $m$). For each matching occurrence, it displays the ASM computed matching distance $d_i$, matching occurrence index number $i$ and the identified node-to-node mapping in following format.

**Node-to-Node Mapping Format**

Node$_1^S$ -> Node$_1^G$

Node$_2^S$ -> Node$_2^G$  ... Similarly, ($a$ - $l$) mapped node pairs are displayed.

This mapping shows that according to ASM computations, Node$_1$ of the subgraph has been mapped to Node$_1$ of the graph. Similarly, Node$_2$ of the subgraph has been mapped to Node$_2$ of the graph. Similarly, ($a$-1) mapped node pairs are displayed since each node in the subgraph is mapped to at least one node in the graph. Each node label Node$_{1S}$, Node$_{1G}$, Node$_{2S}$, Node$_{2G}$ etc. is also associated with line number of the associated program statement in source code.

Next the output shows the minimum distance value of all mappings generated for the graph pair. On the basis of this minimum distance value, the minimum distance mapping can be observed.
Subject System (Source Code) 

Composed of Program Files

Scorpio

Clone Pairs

Proposed Algorithm Implemented in Java

Generate

All Code Fragment Combinations

Approximate Subgraph Matching (ASM)

ASM Relations

Identify Relations having ASM Distance = 0.0

New Clone Relations

Output

Figure 4.8: Flow Diagram of Proposed Clone Detection Method
Further, the clone relation between the cloned code fragments present in program files - file1 and file2 is displayed in the format: $f_1$, $s_1$, $e_1$, $f_2$, $s_2$, $e_2$ where:

- $f_1$: denotes the absolute path of file1
- $s_1$: denotes the start line number of cloned code fragment in file1
- $e_1$: denotes the end line number of cloned code fragment in file1
- $f_2$: denotes the absolute path of file2
- $s_2$: denotes the start line number of cloned code fragment in file2
- $e_2$: denotes the end line number of cloned code fragment in file2
4.6 Output Validation Method

We have performed validation of the obtained clone relations by carefully examining the code fragments corresponding to each clone relation. For each output clone relation, we observed whether actual similarity exists between the code fragments that are identified. The results obtained indicate that a considerable level of code similarity exists in the detected clone relations (elaborated in Section 5.3).

4.7 Chapter Summary

This chapter discusses the experimental design and methodology followed for implementation of the proposed method. Section 4.1 discusses details of the platform and computing environment used. Section 4.2 explains the working of Scorpio tool which was used for obtaining input for the proposed algorithm. Section 4.3 provides details of the subject systems on which the experiment was performed. Section 4.4 discusses the generic methodology of implementation of the proposed approach. The output format of the obtained results has been explained in Section 4.5. The validation method employed to verify the obtained results has been discussed in Section 4.6.
CHAPTER 5
RESULTS AND DISCUSSION

This chapter presents the results and analysis of our proposed clone detection method. It elaborates the results for each subject system on which the proposed method has been tested. It also presents graphical representation of the results obtained. Further, it compares the results with the clone detection results of Scorpio tool. The chapter also analyzes and validates the clone relations detected from each subject system.

5.1 Clone Relations Obtained in Subject Systems

The proposed algorithm was implemented in Java. Results were obtained for each subject software system specified in Table 4.1. The results for the subject systems have been recorded in Tables 5.1, 5.2, and 5.3 respectively.

The input parameters of the algorithm distanceThreshold and distanceWeights are user-defined. For biomedical text mining, approximate subgraph matching instances for a given subgraph - graph pair are found at distanceThreshold value as high as 10. The distanceWeights array is set as per application requirement [10]. But for clone detection purpose, using a smaller distanceThreshold is recommended to allow only minimal variations in each subgraph - graph pair generated for every possible code fragment combination that is to be tested for ASM matching. Also, assigning equal weight values to elements in distanceWeights array is preferable. The results were computed for the values distanceThreshold = 1 and distanceWeights = {1,1,1}.

The p% specified in Tables 5.1, 5.2 and 5.3 is a user-defined input parameter to the algorithm. As defined in Equation (3.1) , a graph pair constructed for a code fragment combination is tested for ASM matching iff the absolute difference in the vertex counts of the graph pair is less than or equal to p% of the larger graph size. So, p% is a measure that defines what type of subgraph - graphs pairs we intend to find clone relations from.

If we keep p% small, for e.g. 10%, then only the code fragment combinations for which subgraph - graph pair have comparable sizes (size difference ≤ 10%) are tested for ASM matching, whereas the combinations for which subgraph – graph pair has larger difference in vertex count are not. As we increase p%, the combinations with higher variation in graph sizes are also tested for ASM matching. When p% is as high as 100%, all combinations are tested for ASM matching.
Increasing p\% automatically increases the number of detected subsumed clone relations and hence, increases the total no. of clone relations identified. So, it can be noted that detection results at a higher p\% are inclusive of the results obtained at a lower p\%.

The selection of an optimum p\% value such as 40 or 50 \% is recommended. This is suggested so that maximum clone relations are identified, but only those code fragment combinations for which the constructed graphs are of comparable sizes (vertex count of the smaller graph is atleast half that of the bigger graph) are evaluated for obtaining clone relations.

The proposed algorithm generates all code fragment combinations from the input clone detection results of PDG-based detection. In this experiment, clone detection result of Scorpio tool [11] was used as the input dataset for every subject system. The algorithm identifies ASM-based clone relations from the code fragment combinations whose corresponding graph pair satisfy p\% condition in Equation (3.1). For every such combination, ASM matching is done. If any matching occurrence exists between the graphs constructed for the combination, then ASM generates all possible node-to-node mappings between the constructed graphs and calculates the subgraph distance function [10] for each mapping. The subgraph distance value satisfies the condition specified in Equation (1.1). There can be more than one node-to-node mapping at the same distance value. Since distanceThreshold was set to 1 for this experiment, it can be observed from the obtained results that every node-to-node mapping generated for a clone relation has distance value in the range [0.0 – 1.0].

For every code fragment combination having ASM matching occurrence in the corresponding graph pair, the proposed algorithm identifies the node-to-node mapping having minimum distance value. The combinations having minimum distance mapping at 0.0 value are the new approximate clone relations detected by the algorithm.

The total number of ASM relations and the total number of clone relations identified by the proposed method for each subject system have been recorded in Tables 5.1, 5.2 and 5.3. The results have been enumerated for different p\% values.

Table 5.1 shows the results for one of the subject systems – the test system used in Scorpio tool. Results at varying p\% values – 10, 30, 40, 50 and 100\% have been enumerated. It can be observed that for any given p\% value, a large percentage of ASM relations yield clone relations. This is because for this experiment distanceThreshold is set to 1 because of which majorly those
ASM relations are detected for which subgraph - graph pair has limited variations in terms of structure, vertex labels and edge directionality. Such ASM relations generally have at least one 0.0 distance mapping and are therefore, detected as clone relations. The results of Table 5.1 have been graphically represented in Figure 5.1. It can be observed that the number of ASM relations and number of clone relations both increase with increase in p%. Initially, as we increase p% both the number counts increase rapidly. But as we increase p% beyond 50%, only a small increase occurs in the number counts. This is because clone occurrences are more probable for code fragments with comparable vertex counts than those with large difference in vertex counts.

Table 5.1: Results on Test System of Scorpio Tool

<table>
<thead>
<tr>
<th>p %</th>
<th>Number of ASM and Clone Relations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of ASM Relations</td>
<td>Number of Clone Relations (min. ASM distance = 0.0)</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>43</td>
</tr>
<tr>
<td>30</td>
<td>158</td>
<td>126</td>
</tr>
<tr>
<td>40</td>
<td>164</td>
<td>130</td>
</tr>
<tr>
<td>50</td>
<td>284</td>
<td>246</td>
</tr>
<tr>
<td>100</td>
<td>296</td>
<td>252</td>
</tr>
</tbody>
</table>

Figure 5.1: Graphical Representation of Results on Test System of Scorpio Tool
It can also be observed that as we increase p% from 30% to 40%, there is only a small increase in the number counts. This is so because, the characteristics of the clone pair dataset obtained from the test system by Scorpio detection are such that, there is only a small increase in the code fragment combinations tested for ASM matching as we increase p% from 30 to 40%. In contrast to this, a steep increase is observed in the number counts as we increase p% from 40 to 50%, again due to the nature of the input clone pair dataset. Such observations may vary from one subject system to another.

The algorithm was also tested on subject systems EIRC and Eclipse-ant. The results obtained have been recorded in Tables 5.2 and 5.3 respectively. These results have been summarized in the graphs of Figures 5.2 and 5.3 respectively. These software systems are large in terms of lines of code. The clone pairs detected for each of them by Scorpio tool are also large in number. Hence, for each of these subject systems, the input dataset for the algorithm is large. At p = 10%, we obtain less number of ASM as well as clone relations as per Equation (3.1). For p = 30%, both the number counts show a large increase compared to p = 10%. When p is increased from 30 to 40%, both the number counts increase at a normal rate. Similar trend is observed when p is changed from 40 to 50%. At p = 100%, all possible new approximate clone relations are identified and hence, both the number counts are maximum.

<table>
<thead>
<tr>
<th>p %</th>
<th>Number of ASM Relations</th>
<th>Number of Clone Relations (min. ASM distance = 0.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>69</td>
<td>25</td>
</tr>
<tr>
<td>30</td>
<td>251</td>
<td>117</td>
</tr>
<tr>
<td>40</td>
<td>324</td>
<td>152</td>
</tr>
<tr>
<td>50</td>
<td>413</td>
<td>199</td>
</tr>
<tr>
<td>100</td>
<td>617</td>
<td>263</td>
</tr>
</tbody>
</table>

Table 5.2: Results on EIRC System

It can be observed from Tables 5.1, 5.2 and 5.3 that given a particular p%, the number of ASM and clone relations generally increase with the increase in size of subject system (few exceptions for subject systems EIRC and Eclipse-ant at p = 10, 30 and 50%). This is because, as the system
size increases, the probability of clone occurrence also increases. But, this observation is not always true because the occurrence of clones largely depends on the methods and techniques adopted by software developers during system development.

![Graphical Representation of Results on EIRC System](image)

**Figure 5.2: Graphical Representation of Results on EIRC System**

### Table 5.3: Results on Eclipse-ant System

<table>
<thead>
<tr>
<th>p %</th>
<th>Number of ASM and Clone Relations</th>
<th>Number of ASM Relations</th>
<th>Number of Clone Relations (min. ASM distance = 0.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>156</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>790</td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1063</td>
<td>591</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1216</td>
<td>688</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1421</td>
<td>762</td>
<td></td>
</tr>
</tbody>
</table>
5.2 Comparison of Results with Scorpio

The experimental results of the proposed algorithm have been compared with the results of Scorpio tool. Table 5.4 shows this comparative analysis. The comparison has been done for results obtained from each subject system for the value of parameter $p = 50\%$ (optimum value).

<table>
<thead>
<tr>
<th>Subject System</th>
<th>No. of Clones Detected by Scorpio</th>
<th>Results of Proposed Algorithm at $p = 50%$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of ASM relations</td>
<td>No. of Clone Relations</td>
</tr>
<tr>
<td>Test System of Scorpio</td>
<td>141</td>
<td>284</td>
</tr>
<tr>
<td>EIRC</td>
<td>301</td>
<td>413</td>
</tr>
<tr>
<td>Eclipse-ant</td>
<td>653</td>
<td>1216</td>
</tr>
</tbody>
</table>

It can be observed from Table 5.4 that for each subject system a large no. of ASM relations and clone relations have been identified from the clone pairs detected by Scorpio. The clone relations identified by the proposed method include some clones detected by Scorpio, some subsumed clone relations derived from Scorpio’s results and some new clone relations found on the basis of approximate subgraph matching.
Also, the number of clone relations detected for a given subject system is higher than the number of clones detected by Scorpio. There is an exception in the case of EIRC system, where the number of clone relations obtained is lesser than number of clones detected by Scorpio. But, the ASM relations identified is still higher in number compared to Scorpio results. This means that a higher p% value will yield more clone relations for EIRC. In general, p = 40 or 50% is recommended for optimum results and meaningful clone relations (actual similarity). These observations are also highlighted in Figure 5.4 which is a graphical representation of the comparison of results with Scorpio tool.

5.3 Validation of Obtained Results

This work proposes a novel approach to detect approximate clone relations from PDG-based clone detection results. The results of the proposed approach have not been evaluated using precision or recall factor which are generally used in evaluation of a new clone detection method. Literature survey in clone detection indicates that precision and recall factor of a clone detection method are generally computed with respect to Bellon’s clone references [15] (clones that should be detected). But the results obtained according to the proposed approach include many subsumed clone relations derived from Scorpio’s output and many approximate clone relations detected on the basis of Approximate Subgraph Matching (ASM). Hence, it is not possible to find
precision and recall with respect to Bellon’s references that is composed of exact clones. To study the accuracy of the proposed algorithm, the clones obtained from the subject systems have been manually validated. This section further discusses some example results obtained from each subject system.

5.3.1 Results on Test System of Scorpio tool

For this subject system, example ASM relations at three distance values are selected for discussing the results: matching distance < 0.5; matching distance = 0.0 (ASM Relation is a Clone Relation); and matching distance > 0.5

a) Subgraph Matching Distance < 0.5

Figure 5.5 shows an output snapshot of an ASM relation detected from Test System of Scorpio Tool at subgraph matching distance less than 0.5.

```
The size of the subgraph: 7
size of the graph: 11

1 matchings generated

matching distance: 0.36231884057971014

matching : 0

Nodes matched:
u += 6 + readInt(u + 4); <847> → u += 6 + readInt(u + 4); <1140>
i > 0 <803> → i > 0 <1062>
String attrName = readUTF8(u + 2,c); <804> → String attrName = readUTF8(u + 2,c);
<1063>
"Code".equals(attrName) <807> → "LocalVariableTable".equals(attrName) <1064>
int i = readUnsignedShort(u) <803> → int i = readUnsignedShort(u) <1041>
Enter <770...945> → Enter <947...1410>
--i <803> → --i <1062>

min distance: 0.36231884057971014

Writing....
C:\Users\my\git\TinyPDG1\TinyPDG\test\test018\ClassReader.java
803
847

C:\Users\my\git\TinyPDG1\TinyPDG\test\test018\ClassReader.java
962
1140
```

Figure 5.5: Output Snapshot 1 – ASM Relation from Test System of Scorpio (distance < 0.5)
In the output snapshot of Figure 5.5, the following ASM relation has been identified (according to Output Format of Section 4.5):

\{ test\test018\ClassReader.java, 803, 847, test\test018\ClassReader.java, 962, 1140 \}

Node-to-Node Mapping

The graphs constructed by the algorithm for this combination have node count sizes of 7 and 11 nodes respectively. Only one node-to-node mapping is found at a distance value of 0.3623. The node-to-node mapping (in the format specified in Section 4.5) is:

1. \( u += 6 + \text{readInt}(u + 4) <847> \rightarrow u += 6 + \text{readInt}(u + 4) <1140> \)
2. \( i > 0 <803> \rightarrow i > 0 <1062> \)
3. \( \text{String attrName = readUTF8}(u + 2,c); <804> \rightarrow \text{String attrName = readUTF8}(u + 2,c); <1063> \)
4. \( "\text{Code}".equals(attrName) <807> \rightarrow "\text{LocalVariableTable}".equals(attrName) <1064> \)
5. \( \text{int i = readUnsignedShort}(u) <803> \rightarrow \text{int i = readUnsignedShort}(u) <1041> \)
6. \( \text{Enter <770...945> \rightarrow Enter <947...1410>} \)
7. \( --i <803> \rightarrow --i <1062> \)

It can be seen from the mapping generated, that exact matches are not obtained for all nodes. Approximation exists but to a limited extent and hence, the matching distance is less than 0.5. This can be verified from the code fragments shown in Figure 5.6.

b) Subgraph Matching Distance = 0.0 (ASM Relation is a Clone Relation)

Figure 5.7 is an output snapshot of an ASM relation detected from Test System of Scorpio Tool at subgraph matching distance equal to 0.0 (hence ASM relation is a clone relation). In this snapshot, the following ASM relation has been identified:

\{ test\test018\ClassReader.java,720, 750, test\test018\ClassReader.java, 592, 637 \}

Node-to-Node Mapping:

The graphs constructed by the algorithm for this combination have node count sizes of 6 and 7 nodes respectively. Only one node-to-node mapping is found at a distance value of 0.0. The node-to-node mapping (in the format specified in Section 4.5) is:

1. \( i > 0 <750> \rightarrow i > 0 <637> \)
2. \( \text{ANNOTATIONS} && i\text{anns} != 0 <749> \rightarrow \text{ANNOTATIONS} && i\text{anns} != 0 <636> \)
3. \( --i <750> \rightarrow --i <637> \)
4. \[\text{int } i = \text{readUnsignedShort}(ianns) \implies ianns + 2 \quad <750> \rightarrow \text{int } i = \text{readUnsignedShort}(ianns) \implies ianns + 2 \quad <637>\]

5. \[\text{ANNOTATIONS && "RuntimeInvisibleAnnotations".equals(attrName) } \quad <720...721> \rightarrow \text{ANNOTATIONS && "RuntimeInvisibleAnnotations".equals(attrName) } \quad <592...593>\]

6. \[\text{ianns} = u + 8; \quad <722> \rightarrow \text{ianns} = u + 8; \quad <594>\]

---

File 1: test\test018\ClassReader.java, Code Lines 803-847

```java
for (int i = readUnsignedShort(u); i > 0; --i) {
    String attrName = readUTF8(u + 2, c);
    ...
    if ("Code".equals(attrName)) {
        if ((context.flags & SKIP_CODE) == 0) {
            code = u + 8;
        }
    } else if ...
    ...
    else if (ANNOTATIONS && "RuntimeInvisibleParameterAnnotations".equals(attrName)) {
       impanns = u + 8;
    } else {
        ...
    }
}
```

847 \[u += 6 + \text{readInt}(u + 4);\]

---

File 2: test\test018\ClassReader.java, Code Lines 962-1140

```java
for (int i = readUnsignedShort(u); i > 0; --i) {
    ...
} u += 2;
...
for (int i = readUnsignedShort(u); i > 0; --i) {
    String attrName = readUTF8(u + 2, c);
    if ("LocalVariableTable".equals(attrName)) {
        ...
    } else if ("LocalVariableTypeTable".equals(attrName)) {
        ...
    } else {
        ...
    }
} u += 6 + \text{readInt}(u + 4);"

---

Figure 5.6: Cloned Code Fragments for Output Snapshot 1
The size of the subgraph: 6
size of the graph: 7

1 matchings generated

matching distance: 0.0

matching : 0
Nodes matched:
i > 0 <750> -> i > 0 <637>
ANNOTATIONS && ianns != 0 <749> -> ANNOTATIONS && ianns != 0 <636>
--i <750> -> --i <637>

int i = readUnsignedShort(ianns)v = ianns + 2 <750> -> int i =
readUnsignedShort(ianns)v = ianns + 2 <637>
ANNOTATIONS && "RuntimeInvisibleAnnotations".equals(attrName) <720...721> -> 
ANNOTATIONS && "RuntimeInvisibleAnnotations".equals(attrName) <592...593>

ianns = u + 8; <722> -> ianns = u + 8; <594>

min distance: 0.0

Writing....
C:\Users\my\git\TinyPDG1\TinyPDG\test\test018\ClassReader.java
720
750

C:\Users\my\git\TinyPDG1\TinyPDG\test\test018\ClassReader.java
592
637

**Figure 5.7: Output Snapshot 2 - Clone Relation from Test System of Scorpio (distance = 0.0)**

As it can be seen from the mapping generated, that exact matches are obtained for all nodes. Hence, the matching distance is exactly 0.0. This can also be verified from code fragments given in Figure 5.8.

c) Subgraph Matching Distance > 0.5

Figure 5.9 is a snapshot of an ASM relation detected from Test System of Scorpio Tool at subgraph matching distance greater than 0.5. The following ASM relation has been identified in the snapshot:

{ test\test018\ClassReader.java, 397, 401, test\test018\ClassReader.java, 964, 1140 }
Node-to-Node Mapping

The graphs constructed by the algorithm for this combination have node count sizes of 5 and 11 nodes respectively. Six node-to-node mappings have been found at the following distance values. These mappings (in the format specified in Section 4.5) are:

1) 0.7058823529411764 (minimum distance)
2) 0.8823529411764706
3) 0.8857142857142857
4) 0.9428571428571428
5) 0.945945945945946
6) 1.0

In the output snapshot of Figure 5.9, the following ASM relation has been identified:

{ test\test018\ClassReader.java, 397, 401, test\test018\ClassReader.java, 964, 1140 }

Minimum Distance Mapping

1. \( v += 2; \) <399> \( \rightarrow u += 8; \) <964>
2. \( \text{int } k = \text{readUnsignedShort}(v + 2) \Rightarrow \text{int } j = \text{readUnsignedShort}(u + 8) \)

3. \( k > 0 \Rightarrow i > 0 \)

4. \( v += 4; \Rightarrow u += 2; \)

5. \( --k \Rightarrow --i \)

As it can be seen from the mapping generated, that there is a large amount of approximation. Hence, the minimum distance mapping is obtained at distance > 0.5. This can be verified from the source code fragments given in Figure 5.10.

```
The size of the subgraph: 5
size of the graph: 11
6 matchings generated
matching distance: 0.7058823529411764
matching : 0  // minimum distance mapping
Nodes matched:
v += 2; <399> -> u += 8; <964>
int k = readUnsignedShort(v + 2) <397> -> int j = readUnsignedShort(u + 8)v = u <1067>
k > 0 <397> -> i > 0 <1041>
v += 4; <401> -> u += 2; <1049>
--k <397> -> --i <1041>

... // 6 matchings generated indexed from 0 to 5
(matching distance: 1.0)

matching : 5
Nodes matched:
v += 2; <399> -> u += 2; <1049>
int k = readUnsignedShort(v + 2) <397> -> int j = readUnsignedShort(u + 8)v = u <1067>
k > 0 <397> -> i > 0 <1041>
v += 4; <401> -> u += 8; <964>
--k <397> -> --i <1041>

min distance: 0.7058823529411764
Writing....
```

Figure 5.9: Output Snapshot 3 - ASM Relation from Test System of Scorpio (distance > 0.5)
5.3.2 Results on EIRC System

For this subject system, a clone relation (at subgraph matching distance = 0.0) is shown in Figure 5.11. In the snapshot, following ASM relation has been identified:

{EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java, 252, 255,
EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java, 281, 286}

The source codes for the identified fragments have been shown in Figure 5.12.
The size of the subgraph: 6
size of the graph: 8

1 matchings generated

matching distance: 0.0

matching : 0

Nodes matched:
Enter <251...258> -> Enter <281...289>  
Component[] comps = getComponents(); <252> -> Component[] comps = getComponents(); <284>  
NickItem item = (NickItem)comps[i]; <255> -> NickItem ni = (NickItem)comps[i]; <286>  
i++ <254> -> i++ <285>  
i < comps.length <254> -> i < comps.length <285>  
int i = 0 <254> -> int i = 0 <285>  

min distance: 0.0

Writing....
C:\Users\my\git\TinyPDG\TinyPDG\EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java  
252  
255  
C:\Users\my\git\TinyPDG\TinyPDG\EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java  
281  
286

**Figure 5.11: Output Snapshot 4 - Clone Relation from EIRC System (distance = 0.0)**

**File 1:** EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java, **Code Lines** 252-255

```java
252 Component [] comps = getComponents();
253
254     for (int i = 0; i < comps.length; i++) {
255     NickItem item = (NickItem) comps[i];

```

**File 2:** EIRC-1.0.3\src\main\ar\com\jkohen\awt\NickList.java, **Code Lines** 281-286

```java
281 public void setTextBackground(Color c) {
282     this.textbg = c;
283
284     Component [] comps = getComponents();
285     for (int i = 0; i < comps.length; i++) {
286     NickItem ni = (NickItem) comps[i];

```

**Figure 5.12: Cloned Code Fragments for Output Snapshot 4**
5.3.3 Results on Eclipse-ant System

For this subject system, a clone relation (at subgraph matching distance = 0.0) is as follows:

The size of the subgraph: 6
size of the graph: 7

1 matchings generated

matching distance: 0.0

matching : 0

Nodes matched:
patIdxEnd == 0 <434> -> end == 0 <265>
int i = 0 <417> -> int i = 0 <251>
int patIdxStart = 0; <411> -> int paramDepth = 0; <247>
int start = 0; <248>
int end = 0; <249>

i++ <417> -> i++ <251>
Enter <395...508> -> Enter <246...267>
patIdxStart++; <444>
strIdxStart++; <445>
-> paramDepth++; <254>

min distance: 0.0

Writing....
C:\Users\my\git\TinyPDG1\TinyPDG\eclipse-ant\src\ant\DirectoryScanner.java
411
445

C:\Users\my\git\TinyPDG1\TinyPDG\eclipse-ant\src\mail\MailMessage.java
246
265

Figure 5.13: Output Snapshot 5 – Clone Relation from Eclipse-ant System (distance = 0.0)

In the output snapshot of Figure 5.13, clone relation has been identified between the following code fragments:

{eclipse-ant\src\ant\DirectoryScanner.java, 411, 445,
eclipse-ant\src\mail\MailMessage.java, 246, 265}
The source codes of these cloned fragments have been shown in Figure 5.14.

**Figure 5.14: Cloned Code Fragments for Output Snapshot 5**
5.4 Results Summary

The results obtained clearly indicate that for each subject system, a large no. of ASM relations and clone relations have been identified from the clone pairs detected by Scorpio. The clone relations identified in the results include some clones detected by Scorpio, some subsumed clone relations derived from Scorpio’s results and some new clone relations found on the basis of approximate subgraph matching technique.

5.5 Chapter Summary

This chapter presents the clone detection results obtained in this work. Section 5.1 elaborately discusses the results obtained for each subject system. Section 5.2 compares the results with the clone detection results of Scorpio tool. Section 5.3 presents a detailed analysis of the obtained results and the validation method used.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the conclusions drawn from this thesis work. It discusses the key applications of the proposed approach for clone detection. It also highlights the future directions in which this work can be taken forward for further development in the field of software clone detection and especially for further enhancement in PDG-based clone detection.

6.1 Conclusions

This research work proposes a novel algorithm to detect new approximate clone relations from the clone pair results of Program Dependency Graph (PDG) based detection. The algorithm uses Approximate Subgraph Matching (ASM) technique [10] to detect new clone relations from current state of the art PDG-based clone detection [2]. For the experimental work, input clone pairs for each subject system were obtained using the Scorpio tool. The proposed algorithm was tested on three subject systems – test system of the Scorpio tool [14], EIRC system [16] and Eclipse-ant system [17] (of Bellon’s benchmark [15]). The obtained results have been manually validated for each subject system. Also, the results have been elaborately analyzed for each subject system. For every detected clone relation, all possible node-to-node mappings (between the graphs constructed corresponding to the relation) and their respective ASM-based distance values are computed. The algorithm determines the minimum distance mapping for each clone relation. This minimum distance measure can be used to quantify the similarity between the identified clone relations. This work proposes a new approach to enhance results of the current state of the art PDG-based detection technique.

The key applications of this research work are:

- The proposed technique can be applied to large software projects and even combinations of multiple projects to detect approximate clone relations. Also, by observing similarity in the type of detected clone relations and identifying occurrences of transitive relations, clone sets can be constructed for a given project(s). All similar approximate clone relations can be classified into the same clone set. Such clone sets will be much larger than the clone sets defined in earlier clone detection techniques, because of the inclusion of approximate clone
relations. It is possible to define a common procedure for all the code fragments in a clone set and replace each code fragment with a procedural call invoked on appropriate parameters. The proposed method can be used for data mining information regarding similar and most common types and thereby enhance development maintenance of software projects.

- For each identified clone relation, the technique enumerates different node-to-node mappings (based on graph pair construction) and their respective approximate subgraph matching distances. This is useful to analyze how the clones would have originally been created i.e. what type of modifications would have been made after a code fragment was copy-pasted.

- The ASM – based computation of the minimum distance mapping can be used to quantify the similarity in each clone relation that is detected.

- Distance measure generated by the algorithm can be used to distinguish clones that are that can be managed within the system from those that are more prone to maintenance issues and need to be removed. Approximate clone relations can be also used to identify alert areas in a given project(s), so that code fragments that are nearly close to being termed as clones can be attended to, during project maintenance.

6.2 Future Work

In future, this work can be taken further in the following directions to further enhance and develop the field of PDG-based clone detection:

- The implementation of the proposed approach can be tested using other approximate subgraph (or graph) matching techniques and algorithms. The work should be taken further to perform elaborate analysis and comparison of the performance of the algorithm for all the techniques and algorithms. This analysis should be verified based on the efficiency and accuracy of each technique (or algorithm).

- The clone distance calculation method of the proposed approach which is based on approximate subgraph matching, can be evaluated in comparison to other clone identification and distance calculation measures such as Euclidean similarity, Cosine similarity, Jaccard distance etc.
• The proposed approach can be validated using precision and accuracy parameters which are generally used for evaluation of clone detection methods.

• Subject software systems can be processed using the proposed clone detection approach and the identified clones should be suitably handled and removed. The actual cost benefit of the resultant system should be evaluated and compared with similar processing using other clone detection approaches.

• The algorithm can be tested on combinations of larger software systems to detect and analyze approximate clone relations for enhanced software development and maintenance.
REFERENCES


[16] Subject System: EIRC
<https://sourceforge.net/projects/eirc/?source=typ_redirect – eirc1.0.3>


