Software Clone Detection Using Cosine Distance Similarity

A Dissertation

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

MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

Submitted by

Chavi Ralhan
(Roll No. 11203034)

Under the Supervision of

Dr. Paramvir Singh
(Assistant Professor)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
DR B R AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY
JALANDHAR – 144011, PUNJAB (INDIA)

July, 2015
CANDIDATE DECLARATION

I hereby certify that the work, which is being presented in the thesis, entitled “Software Clone Detection Using Cosine Distance Similarity” by “Chavi Ralhan” in partial fulfillment of requirements for the award of degree of M.Tech.(Computer Science and Engineering) submitted in the Department of Computer Science and Engineering at Dr B.R. Ambedkar National Institute of Technology, Jalandhar is an authentic record of my own work carried out during a period from January 2014 to July, 2015 under the supervision of Dr. Paramvir Singh. The matter presented in this dissertation has not been submitted by me in any other University/Institute for the award of any degree.

Chavi Ralhan
Roll No: 11203034

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. (Thesis) Viva-Voce examination of Chavi Ralhan, Roll No. 11203034, has been held on ___________ and accepted.

Supervisor

Signature of External Examiner

Head of the Department
Department of Computer Science & Engineering
Dr B.R. Ambedkar NIT, Jalandhar
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards my supervisor Dr. Paramvir Singh, Assistant Professor, Department of Computer Science and Engineering, Dr B R Ambedkar National Institute of Technology, Jalandhar for his invaluable guidance. It would have never been possible for me to take this work to completion without his support. It has been a very enlightening and enjoyable experience to work under his supervision. His advice on both research and career has been priceless.

I thank him for encouraging my research and for allowing me to grow as a research scientist.

I would like to thank whole heartedly all the faculty members of the Department of Computer Science and Engineering for the invaluable comprehension they have imparted on me and for teaching the principles in most exciting and enjoyable way. I also extend my thanks to the technical staff of the department for maintaining an excellent working facility.

Special thanks to my husband Aseem Kumar for being continuous support throughout the entire process, both by keeping me harmonious and helping me putting me pieces together.

I want to thank my Parents and siblings also for supporting and taking me to this stage in life, it was their blessings which always gave me courage to face all challenges and made my path easier.

Last but not the least I would like to thank almighty GOD for giving me enough strength and lifting me uphill this phase of life and I conclude my gratitude by saying the famous quote of Sir Isaac Newton:

“If I have seen further, it is by standing on the shoulders of giants. “

Chavi Ralhan
ABSTRACT

The critical problem in the development of software is existence of duplicate code that can affect the maintainability of the software. The main goal of the clone detection technique is to identify the parts of the software code which are identical. In the past the text based and token based techniques reflected identical code fragments. However they were not considered reliable method because of their inability to find out syntactic differences between programs. The syntactic difference can be efficiently evaluated using abstract syntax tree.

Syntax based clone detection has been found to be useful in detecting duplicate code. There are many ways to find similarity between two programs. The name of the techniques is characteristic vector clustering, metric based vector comparison independent component analysis to analyze method vector and hybrid approach applied on variable size vector for finding similarity. The similarity measure is the distance between two vectors in the above mentioned techniques. This work aims at providing robust and accurate similarity index for detecting software clones. The performance parameters considered are line of codes detected as clones and range of value obtained from cosine distance. The operations used for the required comparison are tree construction of the program code, characteristic vector extraction from the trees and evaluation of the characteristic vector. The results imply that cosine distance similarity is more efficient than random distance measuring function.
# TABLE OF CONTENTS

CANDIDATE DECLARATION i
ACKNOWLEDGEMENTS ii
ABSTRACT iii
TABLE OF CONTENTS iv
LIST OF FIGURES vii
LIST OF TABLES viii

## CHAPTER 1

### INTRODUCTION

1.1 Software Clone Detection Overview 1
1.2 Types of Software Clone 2
1.3 Reasons for Software Cloning 2
1.4 Advantages of Clones 3
1.5 Disadvantages of Clones 3
1.6 Clone Detection Techniques 4
   1.6.1 Textual Based 4
   1.6.2 Token Based or Lexical Technique 4
   1.6.3 Tree Based Clone Detection Technique 6
   1.6.4 Graph Based Clone Detection Technique 6
   1.6.5 Metrics-Based Technique 6
   1.6.6 Hybrid Clone Detection Technique 6
1.7 Clone Detection Process 7
   1.7.1 Intermediate Code Representation 9
   1.7.2 Match Detection Algorithm 9
CHAPTER 2
LITERATURE REVIEW

2.1 Software Cloning
2.2 Software Detection In Vector Based Clone Detection
2.3 Clone Terminology and Clone Class family
2.4 Source Representation and Similarity Function Selected For Clone Detection
2.5 Chapter Summary

CHAPTER 3
METHODOLOGY AND IMPLEMENTATION

3.1 Architecture of Cosine Distance Based Clone Detection
3.2 Generation of Vector in Clone Detection Technique Using Cosine Distance
   3.2.1 Vector for Structural Representation of the Program
   3.2.2 Merging of Vector for Detecting Codes With Larger Context
   3.2.3 Vector Comparison for Similarity
   3.2.4. Vector Similarity Method for Clone Detection
   3.2.5 Methods for Calculating Distance Between Two Vectors
   3.2.6 Algorithm for Generating the Characteristic Vector
   3.2.7 Algorithm For Vector Generation of Tree Forest
   3.2.8 Vector Similarity
3.3 Methodology
3.3.1 Tools Used 32
  3.3.1.1 Deckard 32
  3.3.1.2 Microsoft Excel 33
3.4 Clone Detection Using Cosine Similarity 34
  3.4.1 Cosine Distance Function 35
3.5 Chapter Summary 35

**CHAPTER 4**
RESULTS AND ANALYSIS

4.1 Experimental Results 36
4.2 Results Comparison of Cosine Similarity with Deckard Tool 40
4.3 Analysis of Nature of Cosine Values 41
4.4 Results Summary 41
4.5 Chapter Summary 42

**CHAPTER 5**
CONCLUSIONS AND FUTURE SCOPE

5.1 Conclusions 43
5.2 Future Work 44

References 45
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Reasons of Software Cloning</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Clone Detection Process</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Cosine Distance Based Clone Detection</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Parse Tree with Generated Characteristic Vectors</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Work Flow of Clone Detection Technique Using Cosine Distance</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Commands Used In Deckard Tool</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Clone Cluster by Deckard Tool</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Characteristic Vector in .CSV format</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Cosine Distance Value for Vector Comparison</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Similar Vectors Extracted From Cosine Distance Program</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>LOC Detection Using Cosine Distance Program</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of Results</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Range of Values of Cosine Distance For Vector Comparison</td>
<td>41</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>LOC Calculated From Deckard</td>
<td>37</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>LOC Calculated Using Cosine Distance</td>
<td>40</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 Software Cloning Overview

Software clone, clearly its name explains everything related to the concept. Copying of the code and pasting them without any modification is the technique is known as cloning. And this technique when applied in software codes then it is called software cloning.

In software development the practice of copy and paste is very common. This led to availability of large number of similar code fragment, in soft-ware systems [1, 2]. These similar codes are termed as code clones. There are numerous studies which suggesting the presence of clones in many software systems [3]. In recent year clone detection has emerged as an important research area in the field of software understanding, evolution and maintenance [4,5]. Clone detection is defined as finding similarity in software systems. The measure of similarity can differ based on the structure of the program under study. Many studies have revealed presence of clone up to 7-23% [4]. Hence code cloning is an open research area to be explored.

The Software Clone is exactly related to developing the duplicate software. In software development copy and paste of code is quite a frequent activity in today’s scenario. The duplicated code is known as a software clone and the activity is known as code cloning. Software clones present in the code may lead to propagation of bugs and serious maintenance issues in software. As an example when we running any of the program and we find some error in that at that point we just correct the error at one point but the code require the correction in all the replicated frames of the program. Now, this is the important aspect that we have to identify the related segments throughout the program code. And these kinds of methods demands for the High maintenance cost of the project, for this very reason software clone detection has become as an active field of research. Due to availability of many distinct programming paradigm and languages led to number of clone variants and software clone detection techniques.
1.2 Types of Software Clones

In software cloning the standards in terms of nomenclature are missing since as there exists a number of languages and programming methodologies hence this led to different taxonomies by different researchers. We are listing here different types of clones [1].

**Type-1 (exact clones):** The Program segments which are exactly identical with other segments except for variations in comments and white space.

**Type-2 (renamed/parameterized clones):** In this type of we are looking for syntactically similar structures in given piece of code except for any changes identifier data types, layout of the program and comments, identifiers.

**Type-3 (near miss clones):** This type of clone can be defined as program segments that have been introduced in the software by means of activities like insertions or deletions of statements , changes in identifier name , literals, date types and layouts of the program.

**Type-4 (semantic clones):** The program segments are textually not similar however the functionality of the code is giving the same result.

**Structural clones:** These interrelated classes which have identical patterns during the design and analysis phase of software development. They identify design level similarities [8].This type of clone detection helps in the maintenance of the software

**Function clones:** The clones which are identified at function/method or procedure level only are defined as function clones..

**Model based clones:** In languages where code is replaced by graphical language the detection of duplicate graphical models are termed model based clones [9].

1.3 Reasons for Software Cloning

The practice of code duplication is considered as bad for activity for maintenance phase however the programmers prefers this because of many reasons. The reasons are mentioned below:

- **Programmer’s limitation and time constraints:** The development of software does not take place under ideal circumstances. Due to limited skills set of the programmer and time constraints caused inclination towards copying/pasting/editing.
• **Complexity of the system:** The large systems are very complex and difficult to understand. The programmers are bound to copy existing functionality and logic in the programs.

• **Language limitations:** Kim *et al.* [10] conducted a rigorous study on causes of copying and pasting of code by programmers. The conclusion of the study was due to limitations in programming languages this activity happens. Moreover many do not have necessary support for code reusability which leads to code cloning.

• **Phobia of fresh code:** It is easy to copy the code rather than implementing new ideas in existing software. The Programmers have the fear that new code may result in difficult and lengthy software development life cycle. Furthermore new code may lead to introduction of new bugs.

• **Lack of restructuring:** Because of time constraint the programmers tend to delay activities refactoring, abstraction etc. Hence increasing the subsequent maintenance costs.

### 1.4 Advantages of Clones

The introduction of clones in software is deliberately done by software developers a study was conducted on this issue by Kapser and Godfrey [11, 12]. Some of the advantages of introduction of clones in software are mentioned below:

• Changing requirements can be accommodated easily and in a fast manner.

• The use of templates is encouraged in some programming languages thus making the task easier for the programmer.

• Lacks of reusability and abstraction in a programming language can be easily fulfilled with the help of code clone

• For making the system efficient the overhead of procedure calls can be compensated using code duplication.

### 1.5 Disadvantages of Clones

In this subsection we will be introducing few disadvantages of clones
• **Higher maintenance costs:** The activity of copy and paste increases the maintenance cost of the software, two studies [13, 14] confirm the same fact.

• **Bug propagation:** If a bug is present in a code segment and the same code segmented is pasted at different parts of the program, it is quite evident that the same bug will be present in all the parts of the software. Hence increasing the possibility of bug propagation [15, 16].

• **Bad impact on design:** Code duplication does not support the activities like refactoring, inheritance, abstraction etc. [17, 18]. Hence leading to a bad design.

• **Impact on system understanding/improvement/modification:** It is quite common that the person who developed the original system is not the one who is maintaining it.

### 1.6 Clone Detection Techniques

There are many clone detection techniques proposed in the literature. The classification of the techniques can be done in many ways. Generally the categorization of these techniques can be done on the basis of analysis done on the source code. The techniques can be classified under four names and the details of these techniques are given below:

#### 1.6.1 Textual based Technique

Text-based textual techniques use the source code directly for clone detection process. There is no need of any transformation in the source code before comparison [23]. The code fragments defined in text based approach are lexemes/text/strings and the fragments which are similar are defined as code clones [24].

#### 1.6.2 Token Based or Lexical Technique

The basic building block of a software program is tokens, in lexical approaches the source code is tokenized using first phase of compiler design “lexical analysis”. Once the program is transformed into sequence of tokens, this sequence is transformed into subsequence of tokens and matching of subsequence of tokens is done. The original code corresponding to matched subsequence is declared as code clones. Lexical approach is more reliable than textual approach as they are not affected by renaming, spacing and formatting.
Figure 1.1 Reasons of Software Cloning
Token matching is more meaningful than keyword matching. One of the examples of clone detection technique based on token matching is suffix tree or suffix array based comparison [24]. The tool which uses this technique for clone detection is dup [4].

1.6.3 Tree based clone Detection Technique
The source program constructs is a result of parse tree and an efficient way of finding clones is by finding similar sub-trees. The blocks of the program like variable, token, literals can be represented in tree form, therefore it gives an efficient way of detecting the clones [24]. The intermediate representation in this approach are abstract syntax trees and parse trees. As sub tree matching is a complex task hence the techniques suffer from high execution time when the source code is big enough. The output of this approach can be used for refactoring as it produces syntactic units. This technique works well with duplicate code which is produced by insertion or deletion of statements.

1.6.4 Graph Based Clone Detection Technique
In this approach the source program is transformed into program dependency graph. Further partitioning of these graphs are done and the sub graphs obtained are compared for finding the clones [26, 27].

1.6.5 Metrics-Based Technique
In Metrics-based techniques the code fragment is defined by gathering a number of metrics from the code and after that the comparison of metric vectors are done. The calculation of the metrics are done by transforming the code in abstract syntax tree or control flow graph. one of the popular technique uses fingerprinting functions on metrics extracted from syntactic units for finding the clones [23].

1.6.6 Hybrid Clone Detection Technique
Some researchers have introduced a combination of syntactic and semantic properties of a program to detect the clones. Leitao [26] proposed a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques in accordance with specialized comparison functions. The semantic technique used is call graph.
In one of the technique given by Sutton et al. [27] which gives an idea of evolutionary algorithm to identify the duplicate code in software systems. The code fragment defined in this approach is variable size vectors and the clustering was done using longest common subsequence. In another hybrid approach given by Cordy and Grant[28] they combined an existing information retrieval method, defined as independent component analysis (ICA). The software methods were represented as vectors. Initially the singular value decomposition is applied on method token metrics then independent component analysis (ICA) recognizes points in new vector space corresponding to the input data. The measure of similarity is the distance between any two vectors.

1.7 Clone Detection Process

A clone detector must be able to find parts of code of high similarity in a source code of a system and it is a known fact that we cannot judge which code fragment can be repeated hence an efficient detector should compare each code fragment with others. This one to one comparison is very expensive in terms of computation. Therefore many measures are taken to reduce the domain of comparison. After the identification of the clones there can a need for additional technique or tools for retrieving the actual code. Figure 1.2 shows the set of steps that a general clone detector may follow. The steps listed can be a part of clone detection technique however it is not necessary. We are providing the basic steps in a software clone detection process and all the steps are not included in all the techniques. The details of the phases of clone detection process are given below:

- **Pre-processing:** The source code of divided into code fragments and the comparison domain is determined. The main objectives of this phase are:
  1. All the unnecessary source code is deleted,
  2. The remaining source code is divided into set of code fragments
  3. The fragments are further processed in accordance with the comparison technique used.

- **Transformation:** Once the decision of units of comparison is made, the source code has to be transformed to an intermediate representation or in other words extraction of units from the source code need to be done. This process can also be related to extraction in reverse engineering field.
• **Extraction**: We have to transform the source code to a suitable form which can be accepted as input by the comparison algorithm. This method involves production of tokens, parse tree and control and data flow on source code.

• **Normalization**: Normalization is used to remove white space, difference in commenting and formatting and normalizing identifier names. Pretty Printing is also done in this phase in addition to all the structural transformation.

• **Formatting**: The output of the comparison algorithm which is the clone pair list is transformed to a clone pair list that is acceptable by the original source code database. The mapping of coordinates of clone pairs to the original source file is also done.

• **Post processing/filtering**: The ranking of clones or filtering of clones are done in this phase. The method used for filtering can be manual analysis or heuristic approach.

---

**Figure 1.2: Clone Detection Process**
The clone detection process has two important characteristics as mentioned below:

### 1.7.1 Intermediate Code Representation

The source code cannot be directly considered for clone detection. The uninterested parts like comments and blank lines need to be removed. The preprocessed code is sent to transformation units to obtain an intermediate code. The choice of intermediate code representation depends on extracting useful information needed by the comparison technique. The boundary of the comparison is determined by the granularity of the code. It can be defined as free or fixed, free granularity can be defined as number or statements and fixed e.g. function or class. The most frequent intermediate code representation chosen are Parse tree, Abstract syntax tree, graphs, regularized tokens and source code or text.

### 1.7.2 Match Detection Algorithm

Match detection algorithms is the most crucial decision in clone detection process. After choosing the intermediate code representation, an efficient clustering or match detection algorithm is needed for extracting code clones. The most commonly used match detection techniques are matric vector clustering, dynamic programming, and suffix tree based token by token comparison and feature vector clustering.

### 1.8 Motivations

During literature survey, it was found that most of the authors in their work defined a need for devising a technique for detection of clones. A few of the works have been listed out as below:

**Jiang et al.** [32] proposed the categorization of sub trees with numerical vectors termed characteristic vector in the Euclidean space. The matching detection algorithm used is based on a clustering technique which works on Euclidean distance. The vectors in one cluster are considered similar. The implementation of the technique was done as DECKARD tool. The tool was evaluated on various C and Java programs including the Linux kernel and JDK. DECKARD is both scalable, language independent and accurate tool for clone detection.

**Cordy and Grant** [28] they combined an existing information retrieval method, defined as
independent component analysis (ICA). The software methods were represented as vectors. Initially the singular value decomposition is applied on method token metrics then independent component analysis (ICA) recognizes points in new vector space corresponding to the input data. The measure of similarity is the distance between any two vectors.

Yang et al. [30] stated that Text-based techniques are not often reliable because they do not pin point the differences accurately. Since program have a reliable syntactical structure as described by the grammar. Working on the same idea the author described an algorithm that explores the grammar of the program. The author also proposed dynamic programming scheme that points out the difference between two programs.

All these key points mentioned by different authors constituted the motivations for this work.

1.9 Research Objectives

Previous research revealed that despite the research in the field of software cloning, most of them focused on detecting clones using random distance measuring functions between vectors for similarity, and Cosine distance is interpreted as one of the concrete distance measuring function for similarity. Cosine distance as similarity function in clone detection need to be examined on projects which are following the random distance measuring function to check if it gives a better result over other random distance measuring function in terms of detecting clones. Various objectives of this work are as follows:

1. To propose a new clone detection technique based on comparison of characteristic vector using cosine distance.

2. To compare the results of the proposed clone detection technique with existing technique that uses locality sensitive hashing.

1.10 Chapter Summary

Rest of the thesis is structured as follows:

- Chapter 2 presents literature surveyed on software cloning and clone detection techniques. It also provides a summary of work done so far in the related area.
- Chapter 3 discusses the research methodology and tools used for the study
• Chapter 4 presents the results of the experiments along with a brief description about results wherever necessary.
• Chapter 5 concludes with its findings, threats and work for future in software clone detection
CHAPTER 2
LITERATURE REVIEW

This chapter reviews the concept of software cloning and software clone detection in development of software systems. Sections 2.1 and 2.2 presents the literature surveyed on software cloning and software clone detection in software development. Section 2.3 describes the current status of the clone detection terminology and the source representation terminology which are the most important in understanding of any clone detection technique. Finally the chapter ends with a discussion on intermediate code representation and match detection technique used for the analysis in this study.

2.1 Software Cloning

In the field of software development the existence of the copying and pasting of code fragments with or without modification is a regular practice. The process of copy and paste is termed as software cloning. It is a known fact that if bug is detected in one section of the code then the corrective measure will be needed in all the replicated fragment of the code. Taking into consideration the high maintenance cost of software’s, the need of clone detection has grown as a very important research area. As there are a variety of languages and programming paradigms existing hence we have different clone definitions and detection techniques.

Roy et al. [23] discussed in their writing and provide a framework of clone detection tools and techniques in a comparative mode and evaluated them. In their writing the various tools and techniques are classified, compared and evaluated in two different dimensions. Firstly, the classification is done on the basis of set of overlapping attributes. Secondly, the classification is done on editing scenario based on Type-1, Type-2, Type-3 and Type-4 clones. The author has cited examples which can be used to choose an appropriate clone detection tools and techniques. Also a scheme was designed to categorized clone detection techniques and current clone detectors. A qualitative evolutionary picture of current clone detectors is also mentioned.

Further Baker [4] elaborated that the tool dup can be used as a plagiarism detector for
software systems. His work is based on text based clone detection. It was also stated that the code should be replaced by procedures during the reengineering phase of the system. The documentation of the software must include reference to debugged code and copy present in the system. To support his work a sample program was taken for experimentation and it was concluded that the code can shrink by 14% based on exact matches, and 61% based on parameterized matches. The study highlights that 20–30% of large software systems contains duplicate code.

Bellon [1] performed evaluation of six clone detectors based on code extracted from number of C programs. The researchers performed experiment on clone candidate as third party software. The selection of the technique was done in such a manner that it covered all aspects of clone detection. The technique covers the whole spectrum of clone detection from text to program dependency graph including software metrics.

Kamiya et al. [7] proposed a clone detection technique that used intermediate code representation as token extracted from input source code and a match detection algorithm for token comparison. This technique was implemented as a tool called CCFinder (Code Clone Finder), which extracts duplicate code in all set of languages like COBOL, Java, C++. Further the metrics for defining code clones were defined. To prove the utility of CCFinder and matrices numerous case studies were performed on NetBSD, JDK, and FreeBSD.

Li et al. [2] analyzed that existing code detectors failed to perform when the code database is large or it does not give correct results when duplicated code is a result of insertion and deletion hence the author proposed a tool that use data mining technique to identify duplicate code in large software systems. The experiments performed on linux and other software’s proved that time taken this tool is unacceptable range. The parameter used by the author for result validation was size and granularity.

Yang et al. [30] observed that stated that Text-based techniques are not often reliable because they do not pin point the differences accurately. Since program have a reliable syntactical structure as described by the grammar. Working on the same idea the author
described an algorithm that explores the grammar of the program. The author also proposed dynamic programming scheme that points out the difference between two programs.

**Baxter et al.** [31] gave the argument that clone detection methods on abstract syntax tree exist in many tools, however they do no answer how to remove those clones the author proposed the methods to remove clones by mechanical methods producing inline procedures or standard macros. A tool used in this technique on a production software and accurate clone reports were found as published by previous work. The tool uses the same logic as used by compiler for detecting common sub expression. This method detects tree matches however a number of variations are needed. This method also proposed production of more structured code. Its use can also be explored in the field of reverse engineering to discovered domain concepts.

**Jiang et al.** [32] proposed the categorization of sub trees with numerical vectors termed characteristic vector in the Euclidean space. The matching detection algorithm used is based on a clustering technique which works on Euclidean distance. The vectors in one cluster are considered similar. The implementation of the technique was done as DECKARD tool. The tool was evaluated on various C and Java programs including the Linux kernel and JDK. DECKARD is both scalable, language independent and accurate tool for clone detection.

**Lee et al.** [33] presented a tree-pattern-based method for finding the code clones. Duplicate tree-patterns were collected with the help of anti-unification algorithm and exhaustive comparisons were made which were not redundant and finally clustering was done. This method has possibility of finding different types of clones as its saves the syntactic structure of code in tree pattern cluster. This algorithm is very efficient as the same comparisons are not repeated and every possible pair of tree sub trees is compared with each other.

**Whaler et al.** [34] proposed a new approach for the detection of clones in source code. The source code is represented as an abstract syntax tree in XML and the concept of frequent item sets from data mining is applied on the XML code. We can use different kinds of XML languages for different programming languages. This approach is flexible and can be
configured with any language.

Chilowicz et al. [35] presented a simple and scalable architecture based on AST fingerprinting. In this method indexing of abstract syntax tree is done in a database because of the maintenance of database quick detection of clone clusters and retrieval of their abstract syntax tree is possible. Further this method aimed at finding neighboring exact matches so that large approximate matches could be found. The author targeted intra-project copy paste activities and plagiarism.

Horwitz [36] proposed semantic based clone detection method. The intermediate representation chosen for their work was program dependencies graph. The use of program dependency graph helped in the identification of semantic and syntactic difference between two programs. The detection of similarity is done by first partitioning the graph and then sub-string comparison is done. The partitioning is done after observing the behavior of the source code.

Komondoor et al. [6] proposed an approach that took into consideration the program slicing method and program dependence graphs and to identify the clones in the form of sub graphs. This approach is important because of its capability to find clones whose segments do not occur as continuous text in the program, reordered statements and interconnected clones. The clones found by this method are used in extraction phase. These clones are likely to be meaningful computations.

Mayrand et al. [13] presents a technique for the identification of code clone by extracting metrics from the source codes. The metrics extracted are 21 function metrics divided into four points of comparison. The comparison points are used for the comparison of functions and determined the cloning level. The levels are defined eight points of comparisons. The range of the levels is from an exact copy to distinct functions. The evaluation was done on telecommunication monitoring systems. The usefulness of this technique is in maintainability of large software systems.

Patenaude et al. [20] presented a study to qualitatively evaluate the software systems. This
study was done on JAVA software and evidence of clones in that software were reported. Clone present systems study also reported the occurrence of clones in Java software systems. It was concluded that presence of clone increases the size of system and force to invest higher cost for maintenance. The metrics from the source code were classified in five categories like hierarchical structure, coupling, classes, methods, and clones. The metric values were used to determine the similarity. The results presented by the author are gathered from experiments and clone detection techniques, on over 500 Kilo Lines of JAVA program code.

Kontogiannis et al. [37] examined and evaluated the use of metrics extracted from data and control analysis for determining duplicate code fragments. The metrics acted as signatures for a code segment. The purpose of these signatures were to increase the matching speed that can be used to locate instances of code cloning even in the presence of minor modifications for example changes in variable names, and insertion of statements. This approach is based on information retrieval approach. The metrics were determined from the abstract syntax tree representing the source code. Match detection technique used by the author is dynamic programming using minimum edit distance on source code lines.

F. Lanubile et al. [22] examined the scripting code of web applications for clones and the web application chosen for analysis was large enough software systems. The approach was applied to three web applications and the result proved the script function clones can be identified at a faster rate. Also it prevented redundant script clones in web applications.

2.2 Software Detection in Vector Based Clone Detection

There have been numerous studies on the software detection based on vectors. A previous work which took our attention on software detection techniques are as follows:

Jiang et al. [32] proposed the categorization of sub trees with numerical vectors termed characteristic vector in the Euclidean space. The matching algorithm used is a clustering algorithm based on the Euclidean distance metric. Clustering is applied on the vectors and vectors falling in same cluster are considered clones. The implementation of the technique was done as DECKARD tool. The tool was evaluated on Java programs, C program also on
JDK and Linux kernel. DECKARD is both scalable, language independent and accurate tool for clone detection.

**Li and Sun et al.** [29] explored a new approach for clone detection by considering source code as vectors in metric space and the metric values are considered as the coordinates of these vectors. The distance between vectors is calculated and the distance determines the similarity between code segments. This study was considered as scalable and however the verification was not done on different software systems.

**Kodhai et al.** [21] extracted metrics from source code and those metrics were represented the textual form of the source code. The intermediate code used by the author is vector formed by a metrics extracted from source code. The match detection algorithm is based on finding the similarity of the vectors which are further mapped for finding similar code block. It detects type 1, type 2 of clones using metrics.

**Cordy et al.** [28] they combined an existing information retrieval method, defined as independent component analysis (ICA). The software methods were represented as vectors. Initially the singular value decomposition is applied on method token metrics then independent component analysis (ICA) recognizes points in new vector space corresponding to the input data. The measure of similarity is the distance between any two vectors.

**Sutton et al.** [27] which gives an idea of evolutionary algorithm to identify the duplicate code in software systems. The code fragment defined in this approach is variable size vectors and the clustering was done using longest common subsequence.

### 2.3 Clone Terminology and Clone Class family

Clone detection tools return the results which are in form of clone pairs or clone class. The clone pair can be defined as collection of similar code segment. The similarity relation between the cloned fragments is a symmetric, transitive, reflexive relation [25]. A clone relation holds between two code fragments if and only if they are of the same sequence,
where sequences can be referred to character strings with whitespace, character strings without whitespace, tokens, normalized tokens, abstract syntax tree, parse tree, program dependency graph. In the below mentioned section the definition of clone pair and clone class is given:

**Definition 1:** Code Fragment. A code fragment can be defined as is any sequence of code lines. It can be of any granularity, e.g., free or fixed. A CF is defined by its file name and beginning and ending lines numbers. It is generally denoted as a triple \((\text{Code fragment.Filename}, \text{Code fragment.BeginLine}, \text{Code fragment.EndLine})\).

**Definition 2:** Code Clone. A code fragment \(\text{CF}_2\) is a clone of another code fragment \(\text{CF}_1\) if they are similar by some given similarity criteria, that is, \(\text{fn}(\text{CF}_1) = \text{fn}(\text{CF}_2)\) where \(\text{fn}\) is the similarity function. Two fragments that are similar to each other form a clone pair \((\text{CF}_1, \text{CF}_2)\), and when many fragments are similar, they form a clone group.

**Definition 3:** Code Types. There are two main kinds of similarity between code fragments. Fragments can be similar on two categories 1) based on program text and 2) based on the similarity of the function each code fragment is performing. The first type of clone happens due to the act of copy and paste action. We will provide the types of clones based on both the textual (Types 1 to 3)

**Type 1:** The code Fragment which are identical copies of each other. The variations can be in terms of layout, whitespace and comments.

**Type 2:** The code fragments which are syntactically identical. The variations can be in, types, identifiers, whitespace, literals, layout and comments.

**Type 3:** The code fragments which came into existence with modifications, addition or removal of Statements. The variations can be in, types, identifiers, whitespace, literals, layout and comments.

**Type 4:** The code fragments which are functionally equal however their syntact constructs is different. These types of clones are judged on semantic grounds.

The group of all clone classes that belonging to the same domain is called a clone class family [38]. The domain of a clone class is defined as a set of entities from where the source fragments are extracted [38]. The domain representation depends on programming
language or scope of interest. Some common examples are classes, packages, files, and functions.

2.4 Source Representation and Similarity Function Selected For Clone Detection
This section presents the metrics chosen for this thesis work. As the main focus of this work is to propose an efficient clone detection technique, it requires selection of an intermediate code representation and an efficient similarity function (match detection algorithm). We choose characteristic vector as our intermediate code representation of the code and cosine distance similarity function as match detection algorithm.

2.5 Chapter Summary
This chapter provides the description of literature surveyed on software cloning and software clone detection techniques. It also describes the literature surveyed on intermediate representation of programs as vectors. At last it describes the different clone terminology used so far in clone detection techniques.
CHAPTER 3
METHODOLOGY AND IMPLEMENTATION

This chapter provides an overview of the methodology followed to implement the clone detection technique using cosine distance. It provides a description of proposed clone detection technique chosen along with the tools used to carry out the clone detection process. An overview of the cosine distance calculation program used for calculating the distance between two vectors is demonstrated.

3.1 Architecture of Cosine Distance Based Clone Detection

The Figure 3.1 given below shows the detailed architecture of the proposed clone detection technique. (Cosine distance based clone detection).

![Cosine Distance Based Clone Detection Diagram]

We have evaluated the technique on large scale system like Linux kernel. The comparison of this technique is carried out with Deckard a tree based tool for clone detection [32]. The results reveal that the cosine distance clone detection technique detects more clones as compared to Deckard the clone detection tool, it is as scalable as Deckard a tree based clone detection tool. The similarity index extracted from the experiment also reveals that the limit of similarity and dissimilarity of vectors ranges from [1,-1]. The value 1 signifies exact match and -1 signifies directionally opposite vectors (dissimilar vector). Therefore
we can quantify the distance between each vector for future analysis.

3.2 Generation of Vector in Clone Detection Technique Using Cosine Distance

In this section we will demonstrate the main steps of our technique with the help of an example. The research aimed at enhancing the detection of the clones of Deckard tool. The example illustrated here is on the same ground as discussed by the author of Deckard [32]. The program under consideration is an array initialization program.

\[
\begin{align*}
\text{Code Segment-1} & \quad \text{for (Char } j= 0; j < k; j++) \nonumber \\
& \quad a[i] = 0; 
\end{align*}
\]

\[
\begin{align*}
\text{Code Segment-2} & \quad \text{for (Char } j= 0; j < k; j++) \nonumber \\
& \quad b[i] = "";
\end{align*}
\]

It can be reasoned that if the program structure of two code fragment are grammatically same only the lexemes differ than the parse tree of those two code fragments will be same. Further the pairwise comparison of the parse trees with each other will give us list of code fragments which are clones [32]. As Comparison of large number trees and sub trees will make our task more complex. We devised a method for comparison of trees for detecting clones.

The key step in our technique is the choice of reliable tree representation algorithm. As per the literature survey done there are many ways of extracting the structural information of the programs for clone detection. However we concluded that the characteristic vector extracts the structural information of the parse tree. So we introduced characteristic vector in our clone detection technique.
3.2.1 Vector for Structural Representation of the Program

The characteristic vector of a sub-tree is a point \( x_1, \ldots, x_n \) in the Euclidean space, where each \( c_i \) represents the count of occurrences of a specific tree pattern in the sub-tree [32]. Let's consider the Code Segment-1 and Code Segment-2 as mentioned above and all the node kind present in the Code segments are represented as the tree pattern in figure 3.2. For our research we concluded that all the node kinds were not essential for capturing structural information of trees as there are many redundant nodes and their introduction was there to make the grammar of the programming language simple. The tree also has some irrelevant nodes which does not carry any pattern or dimension in the vector generator. The nodes like square brackets and round brackets are irrelevant. For the given example, the ordered dimensions of characteristic vectors are occurrence counts of the relevant nodes: id, lit, assign, increment, array, condition, expression, declare, and for. Hence, the characteristic vector for the sub tree rooted at declaration is \(<1, 1, 0, 0, 0, 0, 1, 0>\) because there is an id node, a lit node, and a declare node.

The characteristics vector of a parse tree is generated by a post order traversal and summing the vector of children with the parents. As an example, the vector for the sub tree rooted at assign \(<2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0>\) is the sum of the vectors for array \(<2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0>\), = \(<0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>\), primary_e \(<0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0>\), and the additional node assign \(<0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0>\).

3.2.2 Merging of Vector for Detecting Codes with Larger Context

All the techniques mentioned in the literature focus on matching those code fragments which have a sub-tree in the parse tree. It is a general observation that the developers copy codes with some larger context. As this activity bring difference in the nearby nodes hence it prevents the parent node to be detected as duplicate code. To solve the above mentioned issue the generation of the characteristics vector we use vector merging, the process of summing the vector of node sequences. To implement the vector merging a sliding window moves serially in the parse tree such that a large code fragments can be represented by the vector. In Figure 2, for example, we merged the vectors for declare and condition to get the vector \(<3, 1, 0, 0, 0, 1, 0, 1, 0>\) for the combined code fragment.
Now the question arises which nodes must be chosen for merging. Only those nodes must be merged which probably will distinguish the cloned code. It is a tendency of developers to copy and paste the expressions hence the merging of expression vectors will be a good choice. In Figure 2, the merge able nodes are the four children of the for statement. It is not mandatory for merge able nodes to be on a same level. The minimum number of tokens considered here are five for the given example. It is important to keep a minimum number of token defined before creating vectors as this will negate the possibility of small irrelevant vectors from crowding our vector space.

3.2.3 Vector Comparison for Similarity
Now we are able to represent our parse tree in the form of the characteristic vector. The next step would be to process these vectors for similarity to extract the code clones. The characteristic vector is given as input to a C++ program which finds the cosine distance among all the vectors generated. After that the pair of vectors having cosine distance 1 (similar vectors) possible clones are extracted. For the given c code fragments having code segment 1 and code segment 2. Characteristic vector value as 4,3,2,1,1,1,1,1 are reported as clones.

3.2.4. Vector Similarity Method for Clone Detection
In this section, we will take a look at tree similarity algorithm and then we will mention the method to evaluate the characteristics vector. At last we will introduce our vector similarity method for clone detection.

We have defined clones as code fragments which are syntactically similar. As we need to check syntactic similarity of the clones hence we need to compare syntax trees for similarity. For finding the similarity between the trees the concept of tree editing distance has been used.

**Definition 3.2.4.1 (Editing Distance)** The *editing distance* of two trees $T_1$ and $T_2$, denoted by $\delta (T_1, T_2)$, is the *minimal sequence* of edit operations (either relabeled a node, insert a node, or delete a node) that transforms $T_1$ to $T_2$. [32]
Definition 3.2.4.2 (Tree Similarity) Two trees $T_1$ and $T_2$ are $\sigma$-similar for a given threshold $\sigma$, if $\delta (T_1, T_2) < \sigma$. [14]

Definition 3.2.4.3 (Clone Pair) Two code fragments $C_1$ and $C_2$ are called a clone pair if their corresponding tree representations $T_1$ and $T_2$ are $\sigma$-similar for a specified $\sigma$ (similarity function)[32].

The very first approach for syntactic code comparison is comparison of the parse tree. In other words the concept of syntactic tree similarity can lead us to finding syntactically similar code fragments. However the practical implementation of tree similarity cannot be done. The reasons for the same are as follows:

1. Calculation of the tree edit distance is very expensive
2. Pairwise comparison will be expensive.

To minimize the cost of tree similarity we have structurally minimized the trees to vectors. These vectors are further compared for similarity. We have used the same vector concept in our work for detecting clones as used by Jiang et al [32].

The choice of characteristic vector carries:

- Syntactic information of the code
- The pairwise vector comparison is less expensive.

Cosine distance similarity (similarity function) a robust similarity index which works on
vectors efficiently can be implemented on characteristic vector [28].

We have concluded that the representation of the parse tree in the form of characteristic vector is valid because of above mentioned reasons. Also the measure of similarity of two trees is its editing distance hence the criteria for vector similarity will be distance between each vector.

3.2.5 Methods for Calculating Distance between Two Vectors

There are numerous methods available for calculating the distance of two vectors. The method given below calculates the distance between two vectors in given $n$-dimensional space:

Let $u_1 = <x_1, \ldots, x_n>$ and $u_2 = <y_1, \ldots, y_n>$ be two $n$-dimensional vectors.

Method-1: Using Hamming Distance: The hamming distance between the two vectors will be calculated as: $\sum_{i=1}^{n} |x_i - y_i|$

Method-2: Using Euclidian Distance: The Euclidian distance between two vectors is another well know distance measuring technique. It can be calculated as: $\sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$

Method-3 Using Cosine Distance: The cosine distance is a measure of similarity between 2 vectors which calculates cosine of the angle between them. It is one of robust and reliable technique for vector comparison [28].

The technique calculated the distance as: $\cos(\theta) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} (x_i)^2} \times \sqrt{\sum_{i=1}^{n} (y_i)^2}}$

The above mentioned distances are easy to calculate and successfully implemented in many applications (for solving near neighbor queries) for similarity measure of vectors.

Based on our literature survey, the code can be represented as vectors in the following two ways:

1. Vector based on metrics extracted from the block of code [20, 37].
2. Characteristic Vector extraction of the parse tree for tree similarity [32].

From the above study we concluded that the vector comparison can extract clones if it carries enough information of the block of code. Hence, we concluded that characteristic vector is a reliable vector conversion technique as it extracts the syntactic information of the parse tree. In the following section we will discuss about the conversion of parse tree
into vectors. Further we will discuss the algorithm for finding the similarity among these vectors for finding clones.

### 3.2.6 Algorithm for Generating the Characteristic Vector

In this section we illustrated the example on characteristic vector. The example was based on a specific case. We will generalize the construction of the characteristic vector. The algorithm in the following section will map a tree to its corresponding vector which will characterize the structure of a given tree [32]. The assumption taken into consideration is that the tree is a binary tree. Initially we have divided the tree into atomic patterns. The atomic patterns capture the structural information of the tree. The structural information is captured by a parameter $q$, which define the height of the pattern.

**Definition 3.2.6.1 (q-Level Atomic Tree Patterns):** A $q$-level atomic pattern is a complete binary tree of height $q$. Given a label set $L$, including the empty label $\epsilon$, there are at most $|L|^{2^{q-1}}$ distinct $q$-level atomic patterns.

**Definition 3.2.6.2 (q-Level Characteristic Vectors)** Given a tree $T$, its $q$-level characteristic vector (denoted by $v_q(T)$) is $b_1, b_2, \ldots, b_{|L|^{2^q} - 1}$, where $b_i$ is the number of occurrences of the $i$th $q$-level atomic pattern in $T$.

In the example given in section 3.2 denoted by code segment1 and code segment2 we have used the meaningful nodes as the 1 level atomic patterns and their corresponding vector as 1 level characteristic vectors.

It can be reasoned that the compression of trees as $q$-level vectors produces an efficient way to find similarity between the trees based on the definition of edit distance between the trees. As proven by the experiments performed on Deckard. Our aim is to use cosine distance between the characteristic vectors to estimate the editing distance of the corresponding trees [39]. The work by Yang *et al.* [32] on computing tree similarity supports our idea of estimating the tree edit distance.
Theorem 3.2.6.3 (Yang et al., Thm. 3.3 [39]) For any trees $T_1$ and $T_2$ with editing distance $\delta(T_1, T_2) = k$, the $l_1$ norm of the $q$-level vectors for $T_1$ and $T_2$, $H(v_1(T_1), v_2(T_2))$, is no more than $(4q - 3)k$.

For any two integer vectors $v_1$ and $v_2$, $H(v_1, v_2) \leq D(v_1, v_2) \leq H(v_1, v_2)$ Thus from the above corollary we can relate the cosine distance between the two $q$ level vectors with the tree editing distance. The algorithm-1 adapted by Jiang et al. [32] demonstrates the creation of $q$ level characteristic vector.

Algorithm 1: $q$-Level Vector Generation

1. function QVG($T$: tree, $C$: configuration) $V$: vectors
2. $V \leftarrow \emptyset$
3. Traverse $T$ in post-order
4. for all node $N$ traversed do
5. $VN \leftarrow \sum_{n \in \text{children}(N)} Vn$
6. if IsRelevant ($N$, $C$) then
7. $id \leftarrow \text{IndexOf}(N, C)$
8. $V_N [id] \leftarrow V_N [id] + 1$
9. end if
10. if IsSignificant($N$, $C$) and
11. ContainsEnoughTokens($V_N$, $C$) then
12. $V \leftarrow V \{V_N\}$
13. end if
14. end for
15. return $V$
16. end function

We can see from Definition 3.2.6.2, Theorem 3.2.6.3, can be transformed to work on tree forests (trees with addition of another root) (a collection of trees). This modification is important for those code fragments that do not produce single sub tree in the parse tree.
3.2.7 Algorithm for Vector Generation of Tree Forest

The discussion carried out in the previous section details that the vector generation takes place in 2 phases one for the sub tree and another for the tree forest. Hence there are two ways of generating the vector:

One for the code fragments which can be represented by single sub tree and another for tree forest. The merging of vectors of sub trees can help us generate the vector for forest. The algorithm-1 shows how to generate the vector for the sub trees. Let’s consider a parse tree \( T \) we are performing a post order traversal of \( T \) to generate the vectors. The Vector for a sub tree is generated by adding the vectors of the vectors of the constituent sub trees. (Line 5 of Algorithm 1). There are certain tree patterns which are irrelevant for software, hence the concept of relevant and irrelevant nodes are introduced (In Figure 3.2 of section 3.2 we have specified the concept of relevant and irrelevant node). If we found a relevant node in a tree (line 6), we need to check for its index in the vector space using Index Of (line 7) and then we need to update the index of the vector. (Line 8).

As per our observation in an application the units which are more likely to be clones are declarations, expressions and statements. Hence the generation of the sub trees happens for only those sub trees rooted at the above mentioned nodes. The users are given the opportunity to select the significant nodes to generate the q-level vectors. (Line 10) For example, if there was no need for generation of vector for the array node array in Figure 3.2 in section 3.2 then we will make this node as insignificant. Also it has been observed that there are code fragments that produce sub tree with very less tokens. In addition, we have to ignore small sub trees that contain too few tokens (increment in Figure 3.2). We have to define a minimum token requirement for generating vectors with relevant values. This can be achieved by applying the Contains Enough Tokens constraint in the Line 11 of Algorithm-1.

The Algorithm-2 adapted by Jiang et al. [32] shows the generation of vectors for adjacent sub-tree forest.
Algorithm 2: Vector Merging for Adjacent Tree Forests

1. function WVG \( (T: \text{tree}, C: \text{configuration}): \text{vector} \)
2. \( ST \leftarrow \text{Serialize} \ (T, C); \ v \leftarrow \text{NULL} \)
3. \( step \leftarrow 0; \ front \leftarrow ST.\text{head} \)
4. \( back \leftarrow \text{NextNode} (ST.\text{head}, C) \)
5. Repeat
6. \( V_{\text{merged}} \leftarrow \sum_{n \in [\text{front}, \text{back}]} V_n \)
7. While \( back \neq ST.\text{tail} \) \&
8. \( \neg \text{ContainsEnoughTokens} (V_{\text{merged}}, C) \) do
9. \( \text{Back} \leftarrow \text{NextNode} (\text{back}, C) \)
10. \( V_{\text{merged}} \leftarrow \sum_{n \in [\text{front}, \text{back}]} V_n \)
11. End while
12. If RightStep \( (step, C) \) then
13. \( v \leftarrow v \cup \{V_{\text{merged}}\} \)
14. End if
15. \( \text{front} \leftarrow \text{NextNode} (\text{front}, C) \)
16. \( step \leftarrow step + 1 \)
17. until \( front = ST.\text{tail} \)
18. return \( v \)
19. end function

The vector for adjacent tree forest is calculated by serializing the parse tree \( T \) in post-order, a sliding window moves along the serialized tree to combine the \( q \)-level vectors for the nodes which are captured by the sliding windows. It is a general observation that we are not interested in all the nodes in the serialized tree. We need to identify certain node kinds (called merge able nodes) in context of clone detection to make the vector represent a larger code. For example, the significant nodes of declare, condition, increment, and expression in Figure 3.2 are defined as merge able nodes. We can choose node kinds as merge able nodes as we switch from one application to another. If both a parent and a child
are mergeable, we do not consider the child in the sliding window for the benefit of selecting larger clones. This is implemented by NextNode function in Algorithm 2 (line 9).

Users can manage the width of the sliding window and how far it encloses the code fragments in each step, i.e. its stride. The ideal stride value should be large as it allows large code fragments to be encoded together as a vector. It increases the probability of detecting large code fragments as clones. Also larger width of the sliding window reduces the possibility of overlapping among tree fragments and reduces the possibility of false clones. The flexibility of varying the stride value leaves us with an option of generating vectors of different sizes and providing a way for finding similar code of different sizes.

### 3.2.8 Vector Similarity

Consider a large set of vector generated for a given application. One to one comparison of vectors are practically impossible for determine the similarity of the vectors. We need to map these vectors on some value which will determine the similarity of these vectors. Instead, we can calculate the cosine distance among them, and then compare the values of this distance to determine the vectors which are similar. As suggested by cordy et al. [28] rather than using a raw distance metric cosine similarity helps the isolation of similarity feature in vector space efficiently.

Cosine similarity is a measure of similarity between two vectors of n-dimensions by finding the cosine of the angle between them, often used to compare documents.

Given two vectors of attributes, A and B, the cosine similarity, \( \theta \), is represented using a dot product and magnitude as:

\[
\cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}
\]

For our clone detection method, the attribute vectors A and B are the Characteristic vectors of the program. Each attribute of the vector is representing the occurrence of nodes (representing the symbol of grammar for the programing construct) that can never be negative. As a result cosine similarity is nonnegative and bounded between [0, 1].

We have adopted the following steps to identify the most similar code fragments in our application:

- Calculate the characteristic vector of the source code for which similar code fragments have to be identified.
• Construct a matrix of the characteristics vector to normalize the vector space (extract cosine distance).
• Analyze all the pairs of vectors having cosine distance
• The similarity index is 1 for exactly similar vector in Cosine similarity hence code clones.
• Find the pair of similar vectors and their exact lines of code from characteristics vector.

3.3 Methodology
This section provides the methodology adopted to implement the clone detection technique using Cosine distance as illustrated by Figure: 3.3

![Diagram of Clone Detection Technique Using Cosine Distance](image)

**Figure 3.3: Work Flow of Clone Detection Technique Using Cosine Distance**

Once the sample code for detecting clones is selected, the next step was to generate characteristic vector for the sample code. To generate the characteristic vector open source tool, Deckard was used. The data is collected for each release version of the selected projects. The characteristics vectors were obtained for each sample project and
characteristic vectors were then exported in Excel files with .csv extension. Microsoft Excel was used to store the n dimensional values of characteristic values for comparison of sub tree values for similarity. The data was in the form of the matrix where every row depicted a vector in n dimensional vector space and every column represented dimension value of the vectors generated. The dimensions are the count of occurrences of terminals and non-terminals in the grammar of the language.

3.3.1 Tools Used

This section presents an overview of tools used for generating the characteristic vectors collection and tools used to carry out to store statistical analysis.

3.3.1.1 Deckard

Deckard is a tool used in our implementation for generating 1-level vectors to capture tree structures of the parse tree of code. It is language independent tool and works on programs in any programming language that has a context-free grammar.

It automatically generates a parse tree builder to build parse trees which is required for generating the vectors. DECKARD takes a YACC grammar and generates a parse tree builder by replacing YACC actions in the grammar’s production rules with tree building mechanisms. The generated parse tree builders also have high tolerance for syntactic errors.

Commands used for the same is illustrated using Figure: 3.4. DECKARD is more applicable than other tree-based clone detection tools, even for languages with incomplete or inaccurate grammars. DECKARD works effectively for both C and Java. In addition, YACC grammars are available for many languages, often with the requisite error recovery to localize syntax problems. Thus, it should be straightforward to port DECKARD to other languages.
3.3.1.2 Microsoft Excel

Microsoft Excel is a spreadsheet application developed by Microsoft for Microsoft Windows and Mac OS [40]. It is a spreadsheet program that allows storing, organizing and analyzing information. It features calculation, graphing tools, pivot tables, and a macro programming language called Visual Basic for Applications. It has been a very widely applied spreadsheet for these platforms, especially since version 5 in 1993, and it has replaced Lotus 1-2-3 as the industry standard for spreadsheets.

It allows Sectioning of data to view its dependencies on various factors for different...
perspectives. It supplies functions to answer statistical, engineering and financial needs. It displays data as line graphs, histograms and charts, and limited three-dimensional graphical display. It has a programming aspect, Visual Basic for Applications, allowing the user to employ a wide variety of numerical methods. Microsoft Excel has the basic features of all spreadsheets, using a grid of cells arranged in numbered rows and columns to organize data manipulations like arithmetic operations, where column represents a variable and each row represents a case.

It has a variety of interactive features allowing user interfaces that can completely hide the spreadsheet from the user, so the spreadsheet presents itself as an application via a custom-designed user interface. Microsoft Excel 2013 included three views: Normal view, Page Layout view and Page Break view. In our analysis, we have used Microsoft Excel 2013 for storing the n-dimensional characteristic vector values. Then the list of similar vectors were backtracked for lines of code carried in it and total count of lines of codes were calculated by using excel values.

3.4 Clone detection using Cosine similarity

Once we have converted the source code into 1-level characteristic vector. Now for detecting clones these characteristic vectors were compared on one to one basis using cosine distance between them. The excel file of vectors produced by Deckard tool in .csv format was given as an input to the C++ cosine similarity program. The output of our program is an excel file with pair of vectors having cosine distance value as 1(exactly similar vectors). These pairs of vectors are clone pairs. After the clone pairs are extracted. Now we need to analyze the unique clone pairs and the lines of codes content in each identified clone depicted by vectors.

Algorithm 3: Cosine Distance Similarity for Characteristic vectors

1. Cosine Distance Similarity (Code Fragments ,M)
2. Code Fragment: Program for which clones are identified
3. M : Line of codes detected as clones
4. **function** CDS (V: Characteristic vectors, d :Cosine Distance ,S_v : Similar Vector Pair)

5. \( S_v \leftarrow \emptyset \), \( d \leftarrow \emptyset \)

6. **Repeat** for every small \( v \in V \)

7. \( d = \text{function Cosine Distance} (v_i \text{ and } v_{i+1}) (v_i \text{ and } v_{i+2}) \ldots \ldots \ldots (v_i \text{ and } v_{i+n}) \)

8. if \(|d| = 1\) then \( S_v \leftarrow v_i \text{ and } v_{i+x} \).

9. **Add** \( S_v \) to the file containing Similar Vector Pair

10. **End if**

11. **End For**

12. return \( S_v \)

13. **End function**

### 3.4.1 Cosine Distance Function

The cosine distance is calculated between two vectors on one to one basis. Let \( v_i \) be \( n \) dimensional vector represented as \( <x_1, \ldots, x_n> \) and \( v_j \) represented as \( <y_1, \ldots, y_n> \) then the cosine of the angle between them is calculated as:

\[
\text{Function Cosine Distance} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} (x_i)^2} \times \sqrt{\sum_{i=1}^{n} (y_i)^2}}
\]

### 3.5 Chapter Summary

This chapter provides the description of architecture of clone detection technique, method of generation of the characteristic vector, tools and methodologies used for generating characteristic vector and applying cosine distance similarity on the vectors.
CHAPTER 4
RESULTS AND ANALYSIS

This chapter provides a detailed description of the results of clone detection technique using cosine distance. We analyze the proposed technique for software clone detection with tool proposed by Jiang et al. The proposed clone detection technique and clone detection with tool proposed by Jiang et al. computed the clones for a large scale system Linux version 4. From the result gathered we analyzed that the technique proposed is better than clustering of vectors by locality sensitive hashing used by Jiang et al in terms of lines of code detected. There are other findings of our research discussed in this chapter.

4.1 Experimental Results

This section summarizes the results obtained from the experimentation carried out on sample projects. Each result is represented in graphical form. In each result the major points identified during analysis are also highlighted. We performed experiment on Linux to validate our technique. The experiments were carried on a total of Seventy C files the most detailed experiment was carried on Linux Kernel version 4. We are mentioning the details of experiment carried on Linux Kernel a large scale system. Firstly the C files of Linux kernel version 4.0 was given as input to the Deckard tool. We choose the most lenient values for the parameters. The values of parameters are as follows:

1. Minimum number of tokens required for clones was set to 50
2. Stride (size of the sliding window) was set to 2
3. Similarity was set to 1.0.

Figure 4.1 Clone Cluster by Deckard Tool
As depicted in Figure 4.1. The Line of code for each cluster is extracted from the output file of Deckard. The cluster file carries a series of information regarding clone clusters. The information is file number start of line, count of line. We are interested in extracting line of code. The line of code was extracted using Microsoft excel. The result mentioned below in Table 4.1 is results of n number of operation applied in Microsoft excel on the data obtained from the above mentioned file.

**TABLE: 4.1. LOC Calculated From Deckard**

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total line of codes from Deckard</td>
<td>62103</td>
</tr>
<tr>
<td>Total thousand line of codes (KLOC) from</td>
<td>62.103</td>
</tr>
<tr>
<td>Deckard</td>
<td></td>
</tr>
</tbody>
</table>

From the file mentioned in Figure 4.1 the count of lines for each clone cluster is extracted by line number from the attribute value line. The outcome of the line of codes were copied on excel file and further computations were applied on the excel file for calculating the total line of code detected by Deckard as clone. Results gathered for clone obtained from cosine similarity.

The same files of Linux Kernel were converted to characteristic vectors to capture the tree structure information of the parse tree for a given code fragment. The vector result is shown in the given figure 4.2. The excel file represents the rows as vectors and the columns are represented by the attribute of the vectors. This file was computed by the program designed for finding the cosine similarity. The output of the cosine similarity program is depicted in Figure 4.3. The findings after running the program are:-

1. The cosine distance between all the vectors on one to one basis are recorded in a excel file as depicted in figure 4.3
2. The pair of vectors for which the cosine distance value is displayed as 1 is gathered as vectors which are similar.
3. The code from where these vectors are extracted is clone for our method. The attribute of the vectors (line of code) is extracted from the file displayed in Figure 4.4 using excel operations.
The vectors which are similar having cosine distance value 1 are extracted from the file mentioned in figure 4.3 as an excel file with vector pairs. The following figure 4.4 shows...
all those pair of similar vector.

<table>
<thead>
<tr>
<th>vector Number</th>
<th>Vector No-2</th>
<th>Values of number of lines of vector number -1</th>
<th>values of number of lines of vector number -2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10045</td>
<td>12067</td>
<td>208</td>
<td>165</td>
<td>373</td>
</tr>
<tr>
<td>10059</td>
<td>12078</td>
<td>243</td>
<td>187</td>
<td>430</td>
</tr>
<tr>
<td>10062</td>
<td>12081</td>
<td>250</td>
<td>194</td>
<td>444</td>
</tr>
<tr>
<td>10080</td>
<td>10084</td>
<td>139</td>
<td>169</td>
<td>308</td>
</tr>
<tr>
<td>10081</td>
<td>10085</td>
<td>138</td>
<td>168</td>
<td>306</td>
</tr>
<tr>
<td>10082</td>
<td>10085</td>
<td>138</td>
<td>168</td>
<td>306</td>
</tr>
<tr>
<td>10408</td>
<td>10510</td>
<td>228</td>
<td>228</td>
<td>456</td>
</tr>
<tr>
<td>10567</td>
<td>11583</td>
<td>362</td>
<td>970</td>
<td>1332</td>
</tr>
<tr>
<td>10712</td>
<td>10713</td>
<td>48</td>
<td>49</td>
<td>97</td>
</tr>
<tr>
<td>11131</td>
<td>11207</td>
<td>150</td>
<td>150</td>
<td>300</td>
</tr>
<tr>
<td>11136</td>
<td>11138</td>
<td>192</td>
<td>215</td>
<td>407</td>
</tr>
<tr>
<td>11140</td>
<td>11141</td>
<td>469</td>
<td>490</td>
<td>959</td>
</tr>
<tr>
<td>11226</td>
<td>11239</td>
<td>181</td>
<td>204</td>
<td>385</td>
</tr>
<tr>
<td>11227</td>
<td>11240</td>
<td>182</td>
<td>205</td>
<td>387</td>
</tr>
<tr>
<td>11228</td>
<td>11241</td>
<td>187</td>
<td>210</td>
<td>397</td>
</tr>
</tbody>
</table>

**Figure 4.4: Similar Vectors Extracted From Cosine Distance Program**

The next step after gathering similar vectors was to extract exact line of code detected as clone by our technique as shown in Figure 4.4, for that we need to back track the vector file produced by Deckard for gathering the line of code from the attribute values. The Vector file shown in figure 4.4 will give us the Line of code of vectors pairs identified as clones. The vector file carries many attributes out of which Line attribute represents

```plaintext
LOC=OFFSET LINE
```

```plaintext
i.e., 116-109=7
```

**Figure 4.5: LOC Detection Using Cosine Distance Program**

as clones. The vector file carries many attributes out of which Line attribute represents
start of the code fragment and OFFSET represents the end line in the fragment. Hence the difference between the start and end line gave us the line of code. These lines of clones were extracted using excel functions and later sum of all of them gave us the total line of code from cosine distance as depicted in Table 4.2.

### TABLE 4.2: LOC Calculated Using Cosine Distance

<table>
<thead>
<tr>
<th>SOURCE</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total line of codes from Cosine Distance</td>
<td>654939</td>
</tr>
<tr>
<td>Total thousand line of codes (KLOC) from Cosine Distance</td>
<td>654.939</td>
</tr>
</tbody>
</table>

#### 4.2 Results Comparison of Cosine Similarity with Deckard Tool

The comparison of our technique with Deckard is graphically displayed in the form of a graph as shown in Figure 4.6.

![Figure 4.6: Comparison of Results](image)

The analysis of the result shows that the count of lines of code obtained from clone detection using cosine similarity is higher than the Deckard tool. The increase in result count is evident because the comparison of the vectors are done on one to one basis as mentioned in Algorithm-3. Also the use of cosine distance similarity gave us the scope to filter out more vectors which were similar.
4.3 Analysis of Nature of Cosine Value

The other findings of our research states that cosine distance similarity is concrete similarity function than other random distance similarity functions. The analysis of values obtained from one to one comparison of vectors using cosine distance gave us range of values which lies between “0” to “1”. “0” depicted dissimilar vectors and “1” depicted exactly similar vectors hence code clones.

![Figure 4.7: Range of Values of Cosine Distance For Vector Comparison](image)

The analysis of values is graphically represented in the Figure 4.6. The figure depicts that the series of vector comparison values range between “0” to “1” therefore quantifying the similarity index within range of values. As the values are discrete and lies in a defined range hence it opens the field of analysis nature of clones lying in defined cosine values.

4.4 Results Summary

This section summarizes the results generated from the experimentation carried out on sample project as mentioned in the previous section. In each point given below the major points identified during analysis are highlighted.
1. The intermediate code (characteristic vector) carried the syntactic information of the code hence the vector comparison is a reliable method for finding the clones.

2. The range of values of similarity in our technique was from [0, 1] i.e. cosine distance between two vectors can range from 0 to 1 whereas the Euclidian distance varies from [1, ∞] this experiment has helped us quantize the similarity values.

3. The vectors in our experiment were compared on one to one basis. Hence the complete information of the characteristic vector was preserved. Therefore reliability of technique was more hence greater lines of code are detected as clones.

Hence we conclude that cosine similarity is a better index for defining similarity between characteristic vector (results indicate the same) rather than the locality sensitive hashing based on Euclidian distance.

4.5 Chapter Summary
This chapter provided the results obtained from the experiment done on Linux Kernel. The details of the analysis carried and how the result was gathered is discussed in this chapter. The chapter also concluded with finding of our experiment.
CHAPTER 5
CONCLUSIONS AND FUTURE WORK

Duplication of the code or copying a code fragment and then reusing it with the help of pasting tools with or without any modifications is a well-known software practice. One of the major disadvantages of such practice is that if a bug is detected in a code fragment, all the other fragments similar to it must be investigated to check the possibility of existence of the same bug in the similar fragments. The above mentioned reason states that clones should at least be detected. The goal for clone coverage is to determine what fraction of a program is duplicated code. Several studies show that about 5% to 20% of software systems can contain duplicated code, which are basically the results of copying existing code fragments and using them by pasting with or without minor modifications.

In our work, we have proposed a clone detection technique based on cosine similarity. We have identified an algorithm to identify the similar sub-trees using characteristics vector and identified the clones. The characterization of the parse tree as vector effectively captures the structural information of the trees and cosine similarity acted as a similarity tool for identifying similar characteristic vectors which are numerical vectors. We have implemented our technique as a C++ program for finding the similarity of the vectors. We have implemented vector generation by extracting the structural information of the trees which is a language independent technique. The evaluation of this technique was done on a large scale system using Linux Kernel.

5.1 Conclusions
Based on the results drawn in the previous chapter, following conclusions have been identified:

1. The conversion of the parse tree into characteristic vector contains sufficient syntactic information of the code.
2. The copy paste activity done by renaming of expressions, identifiers, literals can easily be identified by our technique.
3. The use of characteristic vector reduces the overhead of comparing the sub trees of the code fragment parse tree for finding the clones. Also its use increases the choice of computation method as the vectors are numerical in nature.

4. Instead of using the raw vector comparison distance metrics cosine similarity helps the isolation of similarity feature in vector space efficiently.

5. The one to one mapping of the vectors for comparison proved to be a better comparison technique than locality sensitive hashing based on Euclidian distance.

6. The cosine distance values range from \([0, 1]\) in our comparison. Hence the values are more quantified than Euclidian distance values which ranges from \([1, \infty]\).

5.2 Future Work

Some interesting challenges were identified while carrying out this thesis work, which could be further considered for the future work. This technique can be implemented as a tool for identifying the software clones. As the characteristic vectors are independent in nature, therefore this technique can be extended in a tool which interprets the files of different languages for clones. Also the same technique can be extended in graph based tools. Graphs use nodes for interpreting the semantic structure of the program hence this technique can be extended for interpreting semantic similarities of the programs. The cosine values gathered can be further analyzed for interpreting the nature of clones detected. This technique can be used in application area like pattern discovery, code refactoring.
REFERENCES


Engineering (WCRE’04), Delft University of Technology, November 2004, pages 100–109.


**Online URL**