Detection of Similar Code Fragments Using Adjacency Structures

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

Master of Engineering
in
Software Engineering

Submitted By
Mukesh Kumar
(Roll No. 801031020)

Under the supervision of
Rajkumar Tekchandani
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

Certificate

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

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

(Mukesh Kumar)

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

(Rajkumar Tekchandani)
Assistant Professor
Computer Science and Engineering Department

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

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

Page | i

Acknowledgement

First and foremost, praises and thanks to the God, the Almighty, for his showers of blessings throughout my research work to complete the research successfully.

I am indebted to my supervisor, Rajkumar Tekchandani for his assistance, ideas, and feedbacks during the process in doing this dissertation. Without his guidance and support, this dissertation can not be completed on time.

I am also thankful to Dr. Maninder Singh, Head, Computer Science and Engineering Department for his kind help and cooperation. I express my gratitude to all the staff members of Computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I would like to say thanks to all my friends especially Sahil, Ravinder and Abhishek for their support.

Lastly, I wish to express my sincere gratitude to my family for their encouragement and moral support.

Mukesh Kumar

Abstract

Software clones are identical or similar pieces of code. They often results because of the copy and paste activities as ad-hoc code reuse by programmers. Software clone research is of high relevance for the industry. Many researchers have reported high rates of code cloning in both industrial and open-source systems.

Code clone detection is the common aspect of reuse activity. Copying code fragments and then reuse with or without modifications are common activities in the software development. This type of reuse approach of existing code is called code cloning and the pasted code fragment is called the clone of the original. Existing approaches does not make use of adjacency structures and their properties. This thesis presented an efficient way of finding the code clones using adjacency structures. A directed graph of the source code is developed and the information is parsed into adjacency structure. In-degree and out-degree of a particular directed graph is detected using the properties of adjacency structure. Our insight is to deduce the flow at the nodes of the directed graphs to find the similarity between the nodes of the directed graphs. An algorithm is proposed for detection of similar code fragments using adjacency structures and their properties.

Table of Contents

Figures **Page No.**

Chapter 1
Introduction

Copying code fragments and then reuse by pasting with or without minor modifications or adaptations are the common activities in software development. This type of reuse approach of existing code is called as code cloning and the pasted code fragment is called as clone of the original [1].

Duplicated code is the code that has been produced by copying the existing code. This technique of producing duplicated code is frequently used for reusing purpose. There are many levels at which software can be reused as small code fragments at function level, libraries and entire subsystems.

Under the constraints of time, many developers try to make use of the already implemented and tested software by cloning and then adapting it to meet the needed functionality.

On the basis of classification of clones they are divided into various types. Moreover, they are further classified on the basis of code clone detection techniques.

1.1 Various Reasons for Code Duplication

Code clones do not occur frequently in software systems. There are several factors that influence the developers for using the cloned code in the system. Clone can be introduced by accidents. There exists various reasons for which the clones can be introduced in the source code [2] [3] [4]. The reasons for code duplication are discussed as follows.

1.1.1 Due to Development Strategy

Different reuse and programming approaches are responsible for introduction of clone in software systems.

1.1.1.1 Reuse Approaches:

Reusing code, logic, design and/or an entire system are the prime reasons of code duplication. These are further sub classified as follows.

a) **Reuse Existing Code by Copy/Paste** [4]**:** The basic approach of reuse mechanism in the development process is to reuse existing code by copying and pasting that leads to the code duplication. This is the fastest way of reusing reliable semantic and syntactic constructs. It is also one of the ways of implementing cross-cutting concerns.

b) **Forking:** The term Forking is used by Kapser and Godfrey [5] for reusing the similar solutions so that they will be diverged significantly with the evolution of the system. In case of driver development for a hardware family if a similar hardware family may already have a driver and thus can be reused with slight modifications.

c) **Design, functionalities and logic reuse** [6]**:** If there is already similar solution available then functionalities and logic can be reused. Considering the ports of a subsystem often there is a high similarity between their structure and functionality. Linux kernel device drivers contain large rates of duplication because all the drivers have the same interface and most of them implement a simple and similar logic.

1.1.1.2 Programming Approaches:

Clones can be introduced by the way a system is developed.

a) **Merging of two software systems with similar functionalities:** Often two software systems of similar functionalities are sometimes merged to produce a new one. Such systems may have been developed by different teams and because of the implementations of similar functionalities in both systems clones may emerge.

b) **Generative programming approach for system development:** Code generated from tools which use generative programming approach may produce huge code clones because this approach often use the same template to generate the same or similar logic.

1.1.2 Due to Maintenance Benefits

Clones are also introduced in the systems to obtain several maintenance benefits.

a) **Developing new code is risky:** Cordy [7] reports that clones do occur frequently in financial software as there exists frequent updates/enhancements of the existing system to support similar kinds of new functionalities. Financial products do not change that much from the existing one especially within the same financial institutions. Software organizations often ask developers to reuse the existing code by copying and adapting to the new product requirements. The reason behind this is high risk of software errors in new fragments and because existing code is already well tested.
b) **Clean and understandable software architecture:** Sometimes for the requirement of clean and understandable software architecture clones are intentionally introduced into the system.

1.1.3 Overcoming Underlying Limitations

Clones can be introduced due to the underlying limitations concerning the programming languages of interest and the developers.

1.1.3.1 Limitations of Programming Languages:

Some limitations of the programming languages can lead to code clones especially when the language in use does not have sufficient abstraction mechanisms.

a) **Insufficient reuse mechanism of programming languages** [8]**:** Inheritance, generic types and parameter passing are abstraction mechanisms that some programming languages do not have and consequently the developers are required to repeatedly implement these as idioms. Such repeating activities may create possibly small and potentially frequent clones.
b) **Writing reusable code is error-prone:** Developing reusable code might lead to errors especially for a critical piece of code. Therefore it is preferred to copy the

existing code and then reuse it by pasting with or without modification rather than making reusable code.

1.1.3.2 Programmer's Limitations:

There are also several limitations associated with the programmers due to which clones are introduced in the system.

a) **Large system is difficult to understand:** Programmers force to adapt existing code in case of large systems due to their complex code architecture.
b) **Time assigned to developers is limited:** Generally developers are assigned a specific time limit to finish a certain project or part of it. Due to these time constraints they look for similar existing solutions. These solutions finish up as copy and paste the existing one and adapt to their current needs.
c) **Insufficient Developer's knowledge in problem domain:** It is always not possible that the developers are familiar with the domain of the project in hand and hence they look for existing solutions of similar problems. Once such a solution is found the developer just adapts the existing solution to his/her needs. Lack of knowledge is one of the reasons that make it difficult for the developer to make a new solution.

1.2 Advantages of Code Duplication

The advantage of detection of code duplication is how to improve the quality of the source code by refactoring the cloned code. There are several other benefits and applications of detecting clones. Some of these advantages are

a) **Detects library candidates:** Davey et al. [9] and Burd&Munro [10] have noticed that the system apparently proves its usability when a code fragment copied and reused multiple times in the system. As a result this fragment can be included in a library to announce its reuse potential legitimately.
b) **Provide good program understanding** [11]**:** If the functionality of a cloned fragment is comprehended it is possible to have an overall idea on the other files containing other similar copies of that fragment. In presence of a piece of code for managing

memory it is necessary that all files which contain a copy must implement a data structure with dynamically allocated space.

c) **Detects usage patterns** [11]**:** The functional usage patterns of a code fragment can be discovered if all the cloned fragments of a same source fragment can be detected.

d) **Detects malicious software** [12]**:** When comparing one malicious software family to another it is possible to find the confirmation where parts of one software system match parts of another. So clone detection techniques can be used in finding malicious software.

e) **Detects plagiarism and copyright violation** [12]**:** Finding similar code may also be useful in detecting plagiarism and copyright infringement.

f) **Helps software evolution research:** Using dynamic nature of different clones in different versions of a system the clone detection techniques are successfully used in software evolution research.

g) **Helps in code compaction:** Clone detection techniques can be used for compact device by reducing the source code size.

1.3 Disadvantages of Code Duplication

On the first impression code duplication seems to be a desirable approach to development as it is associated with reuse implementation speed-up and developer care.

a) **Chance of bug propagation is increased** [13]**:** If a code segment is reused by coping and pasting without or with minor adaptations and that code segment contains a bug then the bug of the original segment may remain in all the pasted segments in the system and therefore the probability of bug propagation may increase significantly in the system.

b) **Increased probability of introducing a new bug** [3]**:** In many cases only the structure of the duplicated fragment is reused with the developer's responsibility of adapting the code to the current need. This process can be error prone and may introduce new bugs in the system.

c) **Difficulty in system modification:** If the duplicated code present in the system it is difficult to add new functionalities in the system or to change existing ones because this take additional time to understand the existing cloned implementation.

d) **Increased maintenance cost:** If a cloned code fragment which is reused many times and that code fragment consist of a bug all of its similar counterparts should be investigated for correcting the bug. There is no assurance that this bug has been already eliminated from other similar parts at the time of reusing or during maintenance.

1.4 Organization of the thesis

Chapter 1: This chapter describes about code cloning, various reasons for code cloning, advantages and disadvantages of code cloning.

Chapter 2: This chapter describes types of code clones, the various techniques used for detection of code clones and the process of code clone detection.

Chapter 3: This chapter describes about the gap analysis, objective of the proposed work and methodology adapted.

Chapter 4: This chapter describes about a tool Control Flow Graph Factory which is used to generate Basic Block Graphs, export the information of the graph and then create adjacency structures using exported information and the proposed algorithm.

Chapter 5: This chapter describes the experimental results of the proposed work.

Chapter 6: This chapter describes the conclusion and future scope.

Chapter 2
Literature Survey

This chapter describes types of code clones the various techniques used for detection of code clones and the process of code clone detection.

2.1 Software Code Cloning

Several studies show that about 5% to 20% of software systems can contain duplicated codes which are basically the results of copying existing code fragments and using then by pasting with or without minor modifications.

However in a post-development phase it is difficult to say which fragment is original and which one is copied and therefore fragments of code which are exactly similar to each other are called code clones i.e. instances of duplicated or similar code fragments are called code clones [1].

Clones are often the result of copy-paste activities. Such activities can significantly reduce the programming effort and time as they reuse an existing fragment of code rather than rewriting similar code from scratch. This practice is common in device drivers of operating systems where the algorithms are similar. There are several other factors such as performance enhancement and coding style because of which large systems may contain a significant percentage of duplicated code.

2.2 Types of code Clones

There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text and they can be similar based on their functionality. The types of clones based on both the textual [14] and functional **[**15**]** similarities.

Type-I: Identical code fragments except for variations in whitespace, layout and comments.

Type-II: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-III: Copied fragments with further modifications such as changed, added or removed statements. In addition to these variations in identifiers, literals, types, whitespace, layout and comments.

Type-IV: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

2.2.1 Type I Clones

In Type I clone a copied code fragment is the same as the original. However there might be some variations in whitespace, comments and layouts. Type I is widely known as exact clones. Consider the following code fragment.

```
1. void  sumProd(int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}
8.}
```

Change in Whitespaces:

```
1.void  sumProd(int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}
8.}
```

Here change in white space is shown by the vertical bold black line 2, 3, 4.

2.2.2 Type II Clones

A Type II clone is a code fragment that is the same as the original except for some possible variations about the corresponding names of user-defined identifiers (name of variables, constants, class, methods and so on) types, layout and comments. The reserved words and the sentence structures are essentially the same as the original one. Consider the following code fragment.

```
1. void  sumProd (int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}}
```

A Type II clone for this fragment is as follows:

Renaming of identifiers:

```
1. void id1 (int n) {
2. int  id2=0;
3. int id3=1;
4. for (int i=1; i<=n; i++) {
5. id2=id2 + i;
6. id3 =id3 * i;
7.}
8. }
```

Here renaming of identifiers shows by:

1. sumProd =>**id1**
2. sum => **id2**
3. product => **id3**

2.2.3 Type III Clones

In Type III clones the copied fragment is further modified with statement(s) changed, added and/or deleted. Consider the following code Fragment.

```
1. void  sumProd(int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}
8.}
```

Modification of lines:

```
1. void  sumProd(int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. if  (i % 2 == 0)
6. sum+= i;
7. product = product * i;
8.}
9.}
```

In the above code the modification of line is shows by the 5^{th} and 6^{th} statement of the code fragment which is in bold.

2.2.4 Type IV Clones

Type IV clones are the results of semantic similarity between two or more code fragments. In this type of clones the cloned fragment is not necessarily copied from the original. Two

code fragments may be developed by two different programmers to implement the same kind of logic making the code fragments similar in their functionality.

Consider the following code fragment.

```
1. void  sumProd(int n) {
2. int  sum=0;
3. int product =1;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}
8.}
```

Reordering of statements:

```
1. void  sumProd(int n) {
2. int product =1;
3. int sum=0;
4. for (int i=1; i<=n; i++) {
5. sum=sum + i;
6. product = product * i;
7.}
8.}
```

Here reordering of statement is given by the bold lines 2 and 3.

2.3 Clone Detection Approaches

Based on the level of analysis applied to the source code the techniques can roughly be classified into four main categories: textual, lexical, syntactic and semantic.

2.3.1. Textual Approaches

There are several code clone detection techniques that are based on pure text-based/string based methods.

In this approach the target source program is considered as sequence of lines/strings. Two code fragments are compared with each other to find sequences of same text/strings. Two or more code fragments are found to be similar are termed as clone pair.

Little or no transformation/normalization is performed on the source code before starting the actual comparison and most of the cases the raw source code is directly used in the clone detection process. The following filtering and/or transformation/normalizations are applied on some approaches:

a) **Comments Removal**: Ignores all kinds of comments in the source code depending on the language of interest.
b) **Whitespace Removal**: Removes tabs and new line(s) and other blanks spaces.
c) **Normalization**: Some basic normalization steps can be applied on the source code.

Baker [16] [17] uses a sequence of lines as a representation of source code and detects line-by-line code clones. Therefore she uses a lexer and a line-based string matching algorithm on the tokens of the individual lines. A tool proposed by Baker's is Dup. It removes tabs, whitespace comment and replaces identifiers of functions, variables with a special parameter. It concatenates all lines to be analyzed into a single text line, hashes each line for comparison and extracts a set of pairs of longest matches using a suffix tree algorithm. Dup detects parameterized matches and generates reports on the found matches. It can also generate scatter-plots of found matches.

Johnson [13] proposed another pure text-based approach that is based on fingerprints of the substrings. In this approach signatures calculated per line are compared in order to identify matched substrings.

Karp-Rabin [18] [19] fingerprinting algorithm is used for calculating the fingerprints of the substring of the length n of the text. First a text-to-text transformation is performed on the considered source file for discarding the uninterested characters. Following this the entire text is subdivided to a set of substrings so that every character of the text appears in at least one substring. After that the matching substrings are identified, a further transformation is applied on the raw matches to obtain better results. Instead of applying a set of text-to-text transformations he applies several different transformation scenarios from a combination of basic transformations such as remove all whitespaces, remove all whitespaces except line separators, remove comments, retain only comments and replace each identifier by an

identifier marker. For finding near-miss duplication he attempted to find a normalized/transformed text by removing all whitespace characters except line separators and by replacing each maximal sequence of alphanumeric characters with a single letter 'i'. A line "for (k = 1; k <= n; k + +) {" is replaced by the line "i (i = i; i <=; i + +}" and the line "#defineXDEF234" by "#iii". This kind of transformation produces much more false positives. However, the requirement of keeping at least 50-lines match reduces the huge number of false positives as anticipated.

Marcus [20] apply latent semantic indexing (LSI) [21] to the source text in order to find high level concept clones such as abstract data types (ADTs) in the source code. This information retrieval approach limits its comparison to comments and identifiers, returning two code fragments as potential clones and a cluster of potential clones when there is a high level of similarity between their sets of identifiers and comments.

2.3.2 Token based Approaches

In the token-based detection approach the entire source is parsed to a sequence of tokens. This sequence is then scanned for finding duplicated subsequence of tokens and finally the original code fragments representing the duplicated subsequence are returned as clones. Compared to text-based approaches, a token-based approach is usually more robust against code changes such as formatting and spacing.

Baker [2] [16] [22] proposed a token-based technique that uses a lexer to tokenize the source code and then the tokens of each line is compared based on a suffix-tree based algorithm. Transformation rules are not applied on the token sequences in this token based technique. However, she introduced the tool Dup that performs parameterized matching by a consistent renaming of the identifiers.

Kamiya et al. [23] proposed one of the leading token-based techniques in which first each line of source code is divided into tokens by a lexer and the tokens of all source code are then concatenated into a single token sequence. The token sequence is then transformed i.e. tokens are added, removed or changed based on the transformation rules of the language of interest aiming at regularization of identifiers and identification of structures. After that identifiers related to types, variables and constants are replaced with a special token. This

identifier replacement makes the code fragments with different variable names as clone pairs. A suffix-tree based sub-string matching algorithm is then used to find the similar sub-sequences on the transformed token sequence where the similar sub-sequence pairs are returned as clone pairs. Once the clone pair information is obtained with respect to the token-sequence(s), a mapping is required for obtaining the clone pair information with respect to the original source code.

Cordy [24] et al. proposed token and line-based technique that has been used to detect near-miss clones in HTML web pages. An island grammar is used to identify and extract all similar code using pretty-printing. Extracted fragments are then compared to each other line by line using the Unix diff algorithm [25] to assess similarity. Because syntax is not taken into account, clones found by token-based techniques may overlap different syntactic units.

However, using either pre-processing [24] [26] [27] or post-processing [28], the clones corresponding to syntactic blocks can be found if block delimiters are known [29].

Zhenmin Li et al. [30] proposed token-based technique which uses frequent subsequence data mining approach to find similar sequences of tokenized statements.

H. Basit et al. [31] uses suffix array instead of a suffix tree based on efficient memory handling and provides efficient code clone detection using flexible tokenization using RTF allowing the user to token strings for better clone detection. The user can suppress insignificant token classes that may cause noise in detection and there is an option for equating different token classes to assign the same ID to different types int, short, long, float, double depending on requirements.

2.3.3. Tree based Approaches

In the tree-based approach a program is parsed to a parsed tree or an abstract syntax tree (AST) with a parser of the language of interest. Similar sub trees are then searched in the tree with some tree matching techniques and the corresponding source code of the similar sub trees are returned as clone pairs.

The parse tree or AST contains the complete information about the source code. Although the variable names and literal values of the source are discarded in the tree representation and more sophisticated methods for the detection of clones can be applied.

Yang [32] proposed earlier a similar approach for finding the syntactic differences between two versions of the same programs by generating a variant of parse tree for both the versions and then applying dynamic programming approach for detecting similar sub trees.

Baxter et al. [3] proposed AST-based clone technique in which a compiler generator is used to generate an annotated parse tree. It compares its sub trees by characterization metrics based on a hash function through tree matching. Source codes of similar sub trees are returned as clones. The hash function enables one to do parameterized matching to detect gapped clones and to identify clones of code fragments in which some statements are reordered.

Wahler et al. [33] proposed a method to find exact and parameterized clones at a more abstract level by converting the AST to XML and using the data mining approach to find clones.

Jiang et al. [34] introduced the tool Deckard for detecting similar trees. In their approach certain characteristic vectors are computed to approximate the structure of ASTs in a Euclidean space. Locality sensitive hashing (LSH) is used to cluster similar vectors using the Euclidean distance metric and thus finds corresponding clones.

2.3.4 Metrics based Approaches

Metrics based approaches gather different metrics for code fragments and compare these metrics instead of comparing code directly. There are several clone detection techniques that use various software metrics for detecting the similar code.

First a set of software metrics called fingerprinting functions are calculated for one or more syntactic units such as a class, function, statement and then the metrics values are compared to find clones over these syntactic units. In most cases the source code is parsed to its AST representation for calculating such metrics.

Kontogiannis et al. [35] build an abstract pattern matching tool to identify probable matches using Markov models. It finds the similarity between two programs on the basis of

metrics. He also proposed two approaches of detecting clones. One approach is the direct comparison of the metrics values that classify a code fragment in the granularity of begin-end blocks with the assumption that two code fragments are similar if their corresponding metrics values are approximate. The other uses a dynamic programming technique for comparing begin-end block based on statement-by-statement basis.

Mayrand et al. [36] calculated several metrics for each functional unit of the program. Functional units with the similar metrics values are identified as code clones. Partly similar functional units are not detected as code clones. It uses a representation of the source code named Intermediate Representation Language (IRL) to characterize each function in the source code. Metrics are calculated from names, layout, expression and control flow of functions. A clone is defined as a pair of whole function bodies that have similar metrics values. This approach does not detect copy paste at other granularity such as fragment-based copy/paste which occurs more frequently than function-based copy paste.

Patenaude et al. [37] proposed very similar kinds of method-level metrics such as number of calls from a method, number of statements, McCabe's cyclomatic complexity, number of use-definition of non-local variables and number of local variables that are used for finding similar methods. They define these metrics for Java language and extends the work with Datrix tool.

Di Lucca et al. [38] [39] proposed the metric-based approach that has been applied for finding duplicated web pages for finding clones in web documents. He also proposed an approach for identifying similar static web pages by computing the distance between items in web pages and evaluating their degree of similarity. A string representation is obtained for each of the web pages of a Web Application by replacing each web page control elements with a distinct symbol. For each of the strings the Levenshtein distances are calculated and are used to compare the associated web pages.

2.3.5 PDG based Approaches

Program Dependency Graph (PDG) based approaches [40] [41] [42] go one step further in obtaining a source code representation of high abstraction than other approaches by considering the semantic information of the source code.

PDG [43] contains the control flow and data flow information of a program and hence carries the semantic information. Once a set of PDGs are obtained from a subject program, isomorphic sub graph matching algorithm is applied for finding similar sub graphs which are returned as clones.

Komondoor and Horwitz [40] proposed one of the leading PDG based clone detection tool which finds the isomorphic PDG sub graphs using program slicing.

Krinke [41] uses an iterative approach for detecting maximally similar sub graphs in the PDG.

Gabel et al. [15] proposed another approach that maps PDG sub graphs to related structured syntax and then finds clones using the tool Deckard.

2.3.6 Hybrid Approaches

This approach is developed by merging two or more existing approaches for code clone detection.

Koschke et al. [44] proposed an approach in which abstract syntax tree (AST) nodes are serialized in pre order traversal then a suffix tree is created for these serialized AST nodes and the resulting maximally long AST node sequences are cut according to their syntactic region so that only syntactically closed sequences remain. Instead of comparing the AST nodes their approach compares the tokens of the AST nodes using a suffix tree-based algorithm and therefore this approach can find clones in linear time and space that leads to a significant improvement towards AST-based approaches. A function-level clone detection technique is proposed for the Microsoft's new Phoenix framework using AST and suffix trees [45]. AST nodes are used to generate a suffix tree which allows analysis on the nodes to be performed in linear time and space as of Koschke et al. This approach can find exact matching function clones. Parameterized clones with identifier renaming can also be detected with this approach.

Greenan [46] proposed similar approach is proposed by for finding method level clones on transformed AST using sequence matching algorithm.

Jiang et al. [34] proposed a novel approach of detecting similar trees where certain characteristic vectors are computed to approximate the structural information within ASTs

in the Euclidean space. A Locality Sensitive Hashing (LSH) [47] is then used to cluster similar vectors w.r.t. Euclidean distance metric and thus code clones

2.4 Clone Detection Process

A clone detector must try to find pieces of code of high similarity in the source text. The main problem is that it is not known in advance which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragment. Such a comparison is prohibitively expensive from the computational point of view and thus several measures are used to reduce the domain of comparison before performing the actual comparisons. Even after identifying potentially cloned fragments further analysis and tool support may be required to identify the actual clones. An overall summary of the basic steps in a clone detection process are discussed in this section. This generic overall picture allows us to compare and evaluate clone detection tools with respect to their underlying mechanisms for the individual steps and their level of support for these steps.

2.4.1 Pre-processing

At the beginning of any clone detection approach the source code is partitioned and the domain of the comparison is determined. Three main objectives of this phase are:

a) **Remove uninteresting parts:** All the source code uninteresting to the comparison phase is filtered out in this phase. Partitioning is applied to embedded code to separate different languages. This is especially important if the tool is not language independent. Similarly generated code and sections of source code that are likely to produce many false positives can be removed from the source code before proceeding to the next phase.

b) **Determine source units:** After removing the uninteresting code the remaining source code is partitioned into a set of disjoint fragments called source units. These units are

the largest source fragments that may be involved in direct clone relations with each other.

c) **Determine comparison units/granularity:** Source units may need to be further partitioned into smaller units depending on the comparison technique used by the tool. Source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit. For instance an if control statement can be further partitioned into a conditional expression, then block and else block. The order of comparison units within their corresponding source unit may or may not be important depending on the comparison technique. Source units may themselves be used as comparison units. In a metrics-based tool, metrics values can be computed from source units of any granularity and therefore subdivision of source units is not required in such approaches.

2.4.2 Transformation

Once the units of comparison are determined and if the comparison technique is other than textual, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. This transformation of the source code into an intermediate representation is often called extraction in the reverse engineering community.

Some tools support additional normalizing transformations following extraction in order to detect superficially different clones. These normalizations can vary from very simple normalizations such as removal of whitespace and comments to complex normalizations involving source code transformations. Such normalizations may be done either before or after extraction of the intermediate representation.

2.4.2.1 Extraction

Extraction transforms source code to the form suitable for input to the actual comparison algorithm. Depending on the tool it typically involves one or more of the following steps.

a) **Tokenization:** In case of token-based approaches each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The

tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. CCFinder [23] and Dup [16] are the leading tools that use this kind of tokenization on the source code.

b) **Parsing** [3]**:** In case of syntactic approaches the entire source code is parsed to build a parse tree or abstract syntax tree (AST). The source units to be compared are represented as sub trees of the parse tree or the AST. The comparison algorithms look for similar sub trees to mark as clones. Metrics-based approaches may also use a parse tree representation to find clones based on the metrics for sub trees.

c) **Control and data flow analysis** [40]**:** Semantics aware approaches generate program dependence graphs (PDGs) from the source code. The nodes of a PDG represent the statements of a program and edges represent the control and data dependencies. Source units that need to be compared are represented as sub graphs of these PDGs. Clones are detected on the basis of sub graphs isomorphism. Some metrics-based approaches are applied on PDG sub graphs to calculate data and control flow metrics.

2.4.2.2 Normalization

Normalization is an optional step intended to eliminate superficial differences such as differences in whitespace, commenting, formatting and identifier names.

a) **Removal of whitespace:** Almost all approaches disregard for whitespaces although line-based approaches retain line breaks. Some metrics-based approaches however use formatting and layout as part of their comparison. Davey et al. [9] uses the indentation pattern of pretty-printed source text as one of the features of their attribute vectors. Mayrand et al. [36] used the layout metrics such as the number of non-blank lines.

b) **Removal of comments:** Most approaches remove and ignore comments in the actual comparison. However Marcus and Maletic [20] explicitly used the comments as part of their concept similarity method. Mayrand et al. [36] used the number of comments as one of their metrics.

c) **Normalizing identifiers:** Most approaches apply identifier normalization before comparison in order to identify parametric Type-2 clones. In general all identifiers in the source code are replaced by the same single identifier in such normalizations.

However Baker [16] usesd an order-sensitive indexing scheme to normalize identifiers for detection of consistently renamed Type-2 clones.

d) **Pretty-printing of source code:** Pretty printing is a simple way of reorganizing the source code to a standard form that removes differences in layout and spacing. Pretty printing is normally used in text-based clone detection approaches to find clones that differ only in spacing and layout. Cordy et al. [27] used an island grammar to generate a separate pretty printed text file for each potentially cloned source unit.

e) **Structural transformations:** Other transformations may be applied that actually change the structure of the code. The minor variations of the same syntactic form may be treated as similar. Kamiya et al. [23] remove keywords such as static from C declarations.

2.4.3 Match Detection

The transformed code is then processed into a comparison algorithm, where the transformed comparison units are compared to each other to find matches. Often adjacent similar comparison units are joined to form larger units. For techniques of fixed granularity all the comparison units that belong to the target granularity, clone unit are aggregated. For free granularity techniques aggregation is continued as long as the similarity of the aggregated sequence of comparison units is above a given threshold yielding the longest possible similar sequences.

The output of match detection is a list of matches in the transformed code which is aggregated to form a set of candidate clone pairs. In addition to simple normalized text comparison popular matching algorithms used in clone detection include suffix trees dynamic pattern matching (DPM) [48] and hash-value comparison [3].

Figure 2.1 shows the steps of code clone detection process that follow in general.

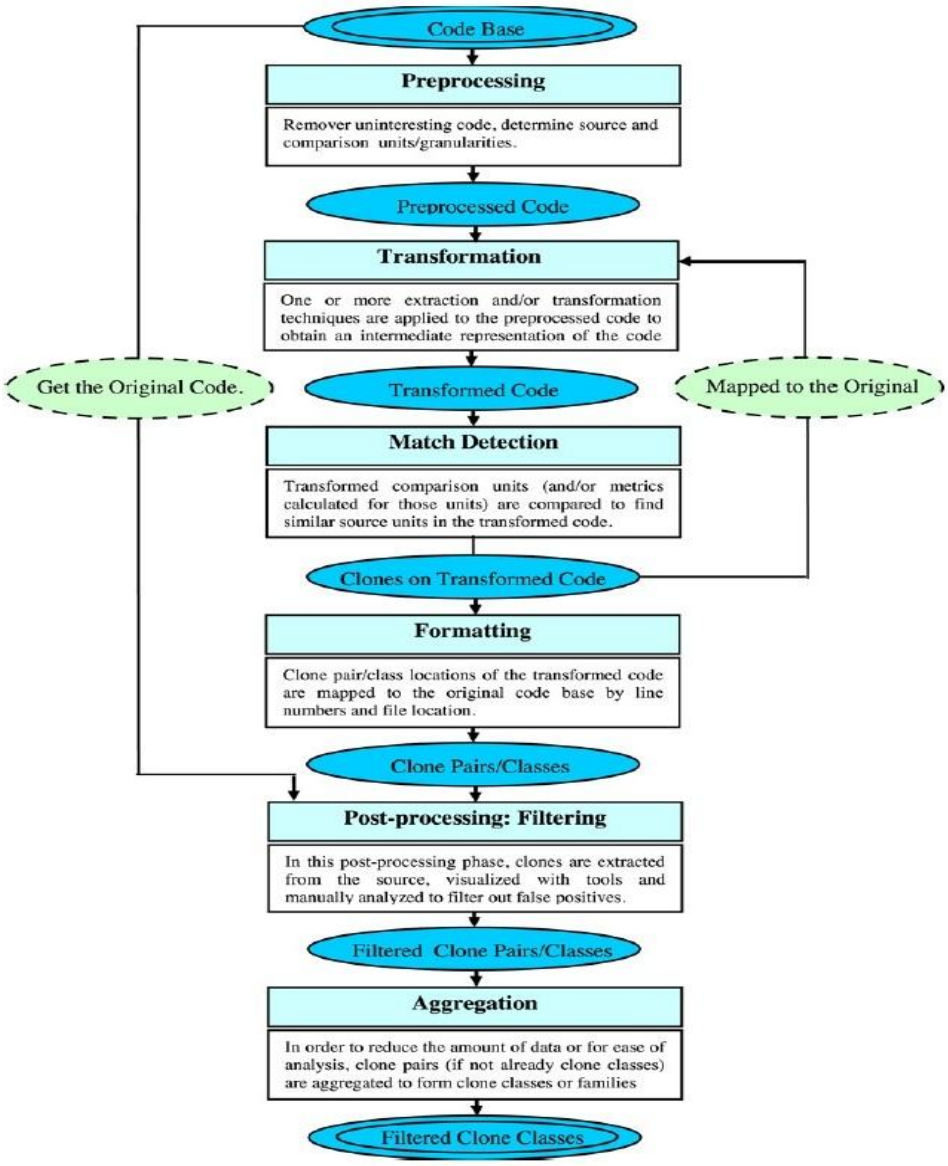


Figure 2.1: Clone Detection Process [1].

2.4.4 Formatting

In this phase the clone pair lists the transformed code obtained by the comparison algorithm is converted to a corresponding clone pair list for the original code base. Source

coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.

2.4.5 Post-processing/Filtering

In this phase clones are ranked or filtered using manual analysis or automated heuristics.

a) **Manual analysis:** After extracting the original source code the clones are subjected to a manual analysis where false positive clones or spurious clones [44] are filtered out by a human expert. Visualization of the cloned source code in a suitable format (e.g. as an HTML web page) can help speed up this manual filtering step.
b) **Automated heuristics** [25]**:** Often heuristics can be defined based on length of clones in order to rank or filter out clone candidates automatically.

2.4.6 Aggregation

While some tools directly identify clone but most of the tools return only clone pairs as the result. In order to reduce the amount of data subsequent analysis is performed and gathers overview statistics to aggregate the clone pairs into classes.

Chapter 3

Problem Statement

3.1 Gap Analysis

Several code cloning techniques are discussed in the literature survey. As per literature survey, various techniques exist for the detection of similar code fragments. Text based techniques are based on textual comparison, token based techniques are based on the comparison of token sequences, tree based techniques are based on finding the similar sub trees, metric based techniques compares the metric values of code fragments and functional based techniques finds the similar functional units.

There exists no work in the literature survey for finding the similar code fragments using adjacency structures and their properties.

3.2 Objective of the proposed work

To propose an algorithm for detection of similar code fragments using adjacency structures and their properties.

3.3 Methodology

a) Till now the adjacency structures have not been explored for finding the similar code fragments.

b) To propose an algorithm for finding the similar code fragments, the source code will be transformed to the directed graphs and their corresponding adjacency structures.

c) Similar flows at the nodes of the corresponding digraph of the source code will be explored using the properties of adjacency structures.

d) Flows at each node of the developed directed graphs of the source files will be compared.

Chapter 4

Implementation

This chapter describes about the tool Control Flow Graph Factory which is used to generate Basic Block Graphs, export the information of the graph, create adjacency structures using exported information and the proposed algorithm.

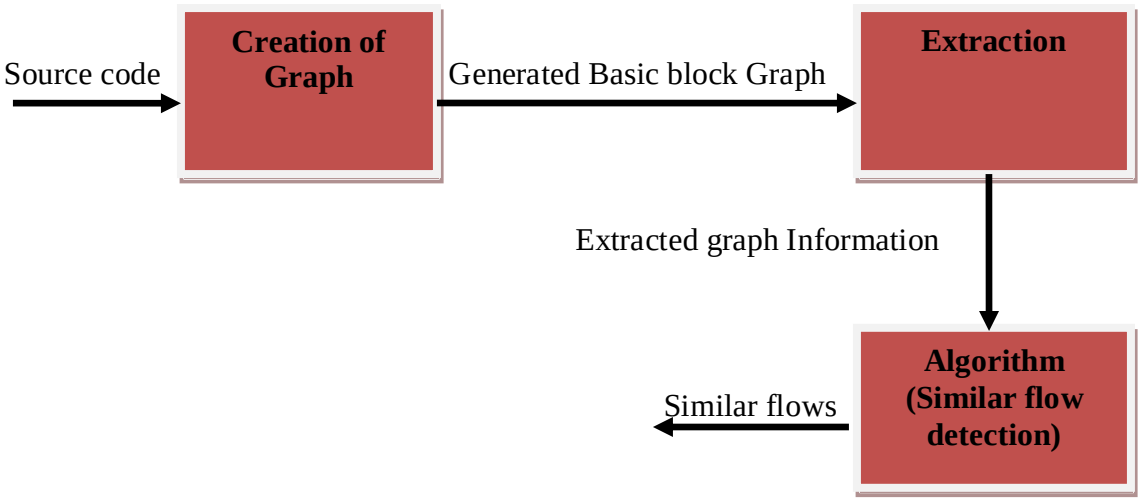


Figure 4.1: Work flow of the proposed technique.

4.1 Steps to generate the similar flows at the nodes of the Directed Graph

Step 1: To generate a Graph (G1) from java code using "While" loop

To generate the graph for the method main select the method in package explorer and open the context menu "Create Control Flow Graph".

```
Package test1;
Public class tes1 {
Public static void main(String[] args) {
int i=0;
while (i<10){
System.out.print ("hello");}}}
```

a) To create Basic Block Graph (G1).

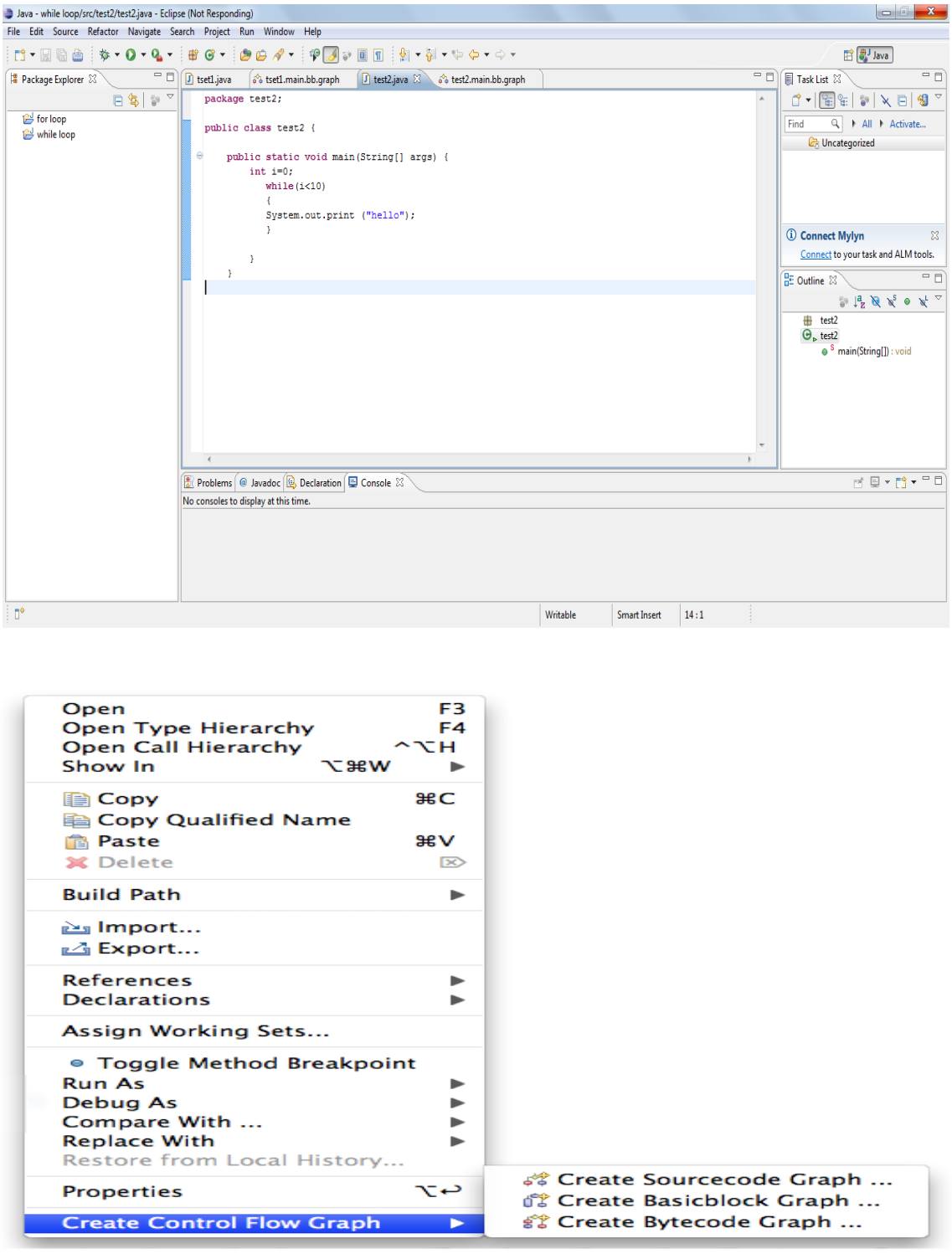


Figure 4.2: To create Basic Block Graph (G1).

b) To save the Graph (G1) in to target folder.

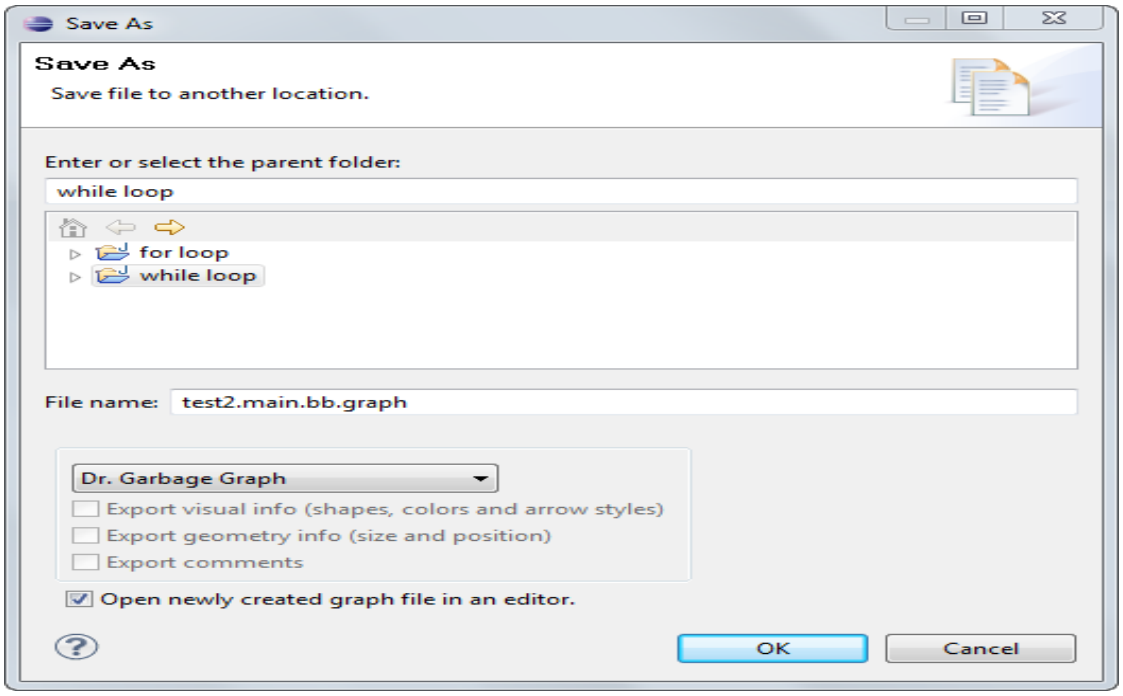


Figure 4.3: Save the Graph (G1) in to target folder.

c) The generated Basic Block Graph (G1).

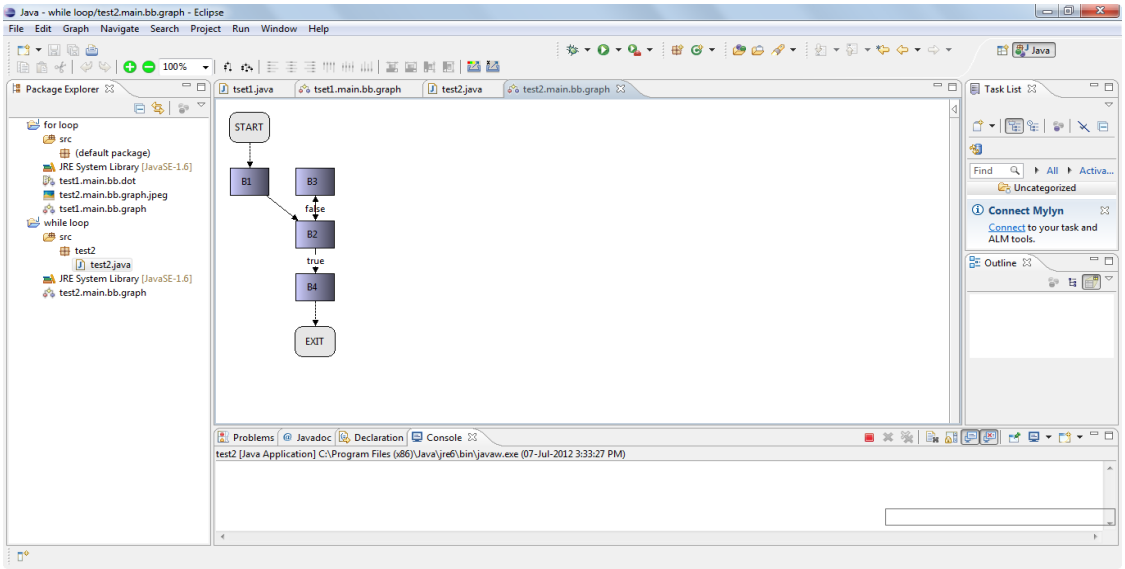


Figure 4.4: Generated Basic Block Graph (G1).

d) Exporting the information of Graph (G1) in Dot format.

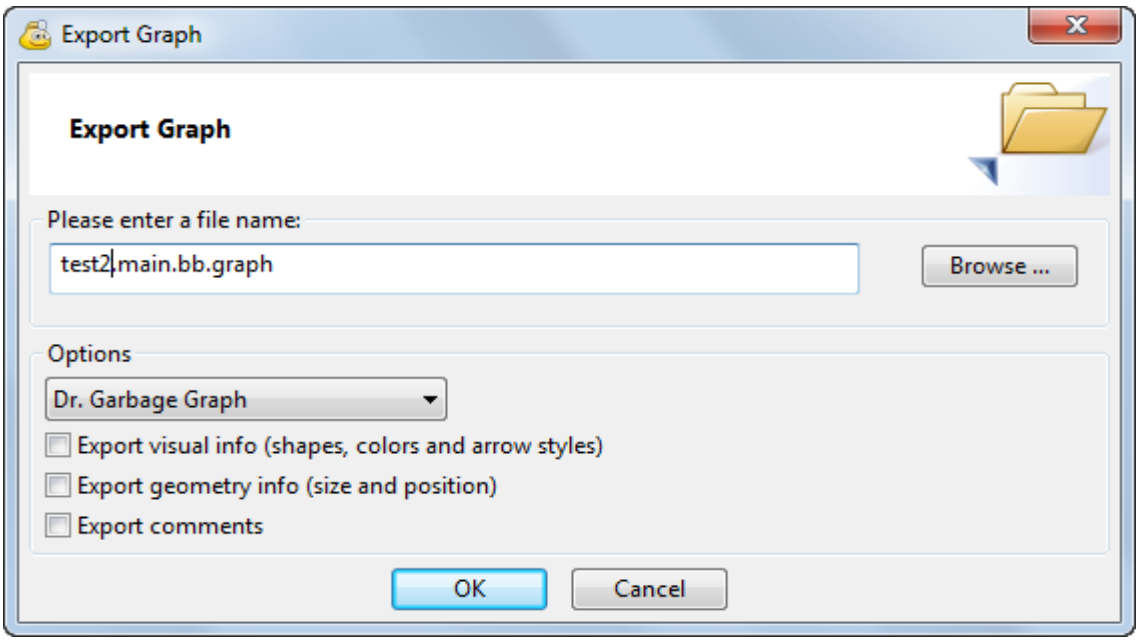


Figure 4.5: Exporting the information of Graph (G1) in Dot format.

e) The Exported information of Graph (G1) in Dot format.

The exported information of graph (G1) in dot format, gives the information about all the vertices and edges which are connected with each other.

```
/* ------------------------------------------------- */
/* Generated by Dr. Garbage Control Flow Graph Factory */
/* http://www.drgarbage.com                          */
/* Version: 3.7.4.201206010948                       */
/* Retrieved on: 2012-07-07 15:51:58.694             */
/* ------------------------------------------------- */
 Graph "test2.main.bb.graph" {
 digraph [label="test2.main.bb.graph"];
 2 [label="B1", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 3 [label="B2", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 4 [label="B3", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
```

5 [label="B4", fixedsize=true, fontsize=12, width=0.48, height=0.36]

6 [label="EXIT", fixedsize=true, fontsize=12, width=0.5, height=0.4]

7 [label="START", fixedsize=true, fontsize=12, width=0.5, height=0.4]

7 -> 2 [label=""]

2 -> 3 [label=""]

4 -> 3 [label=""]

3 -> 4 [label="false"]

3 -> 5 [label="true"]

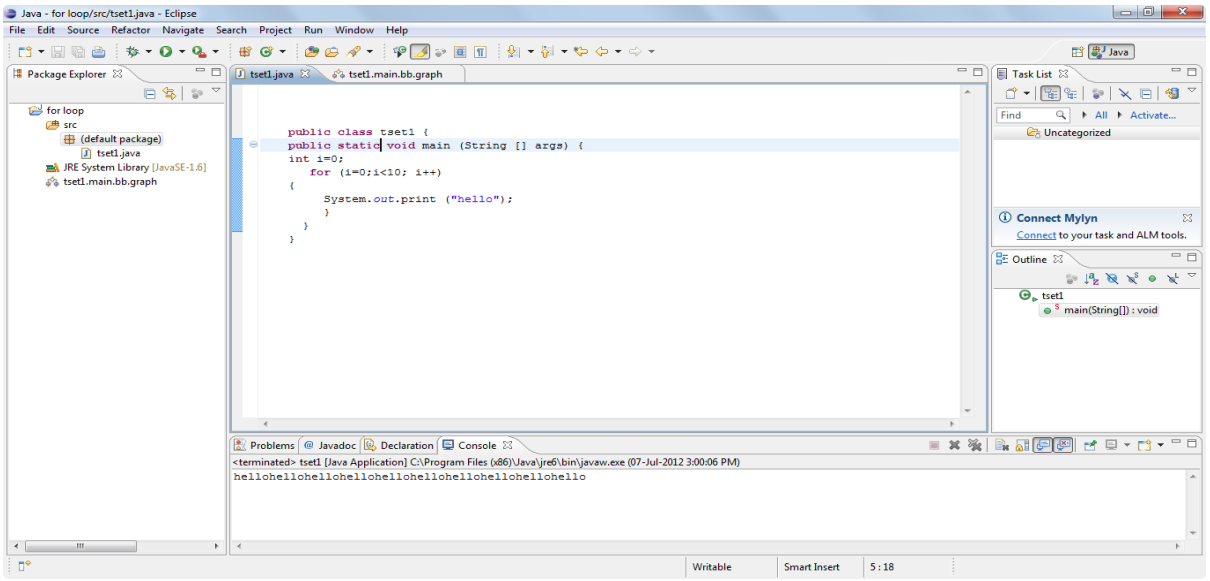
5 -> 6 [label=""]

}

Step 2: To generate a Graph (G2) from java code using "For" loop.

```
Package test2;
Public class tse2 {
Public static void main (String [] args) {
int i=0;
for (i=0;i<10; i++)
{
System.out.print ("hello");}}}
```

a) To create Basic Block Graph (G2).



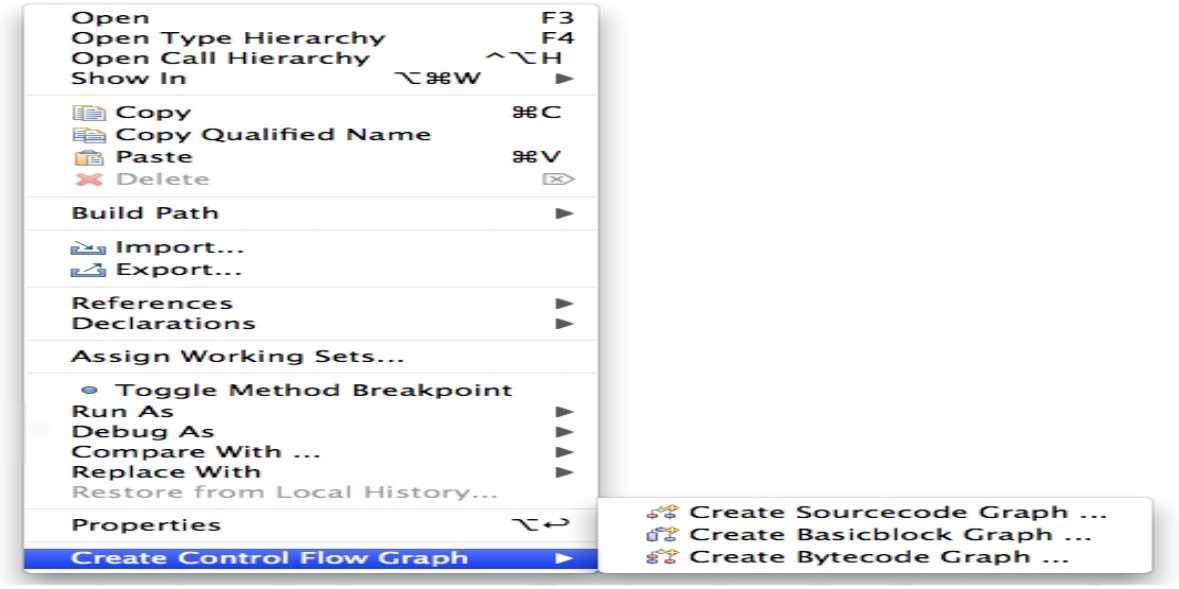


Figure 4.6: To create Basic Block Graph (G2).

b) To save the Graph (G2) in to target folder.

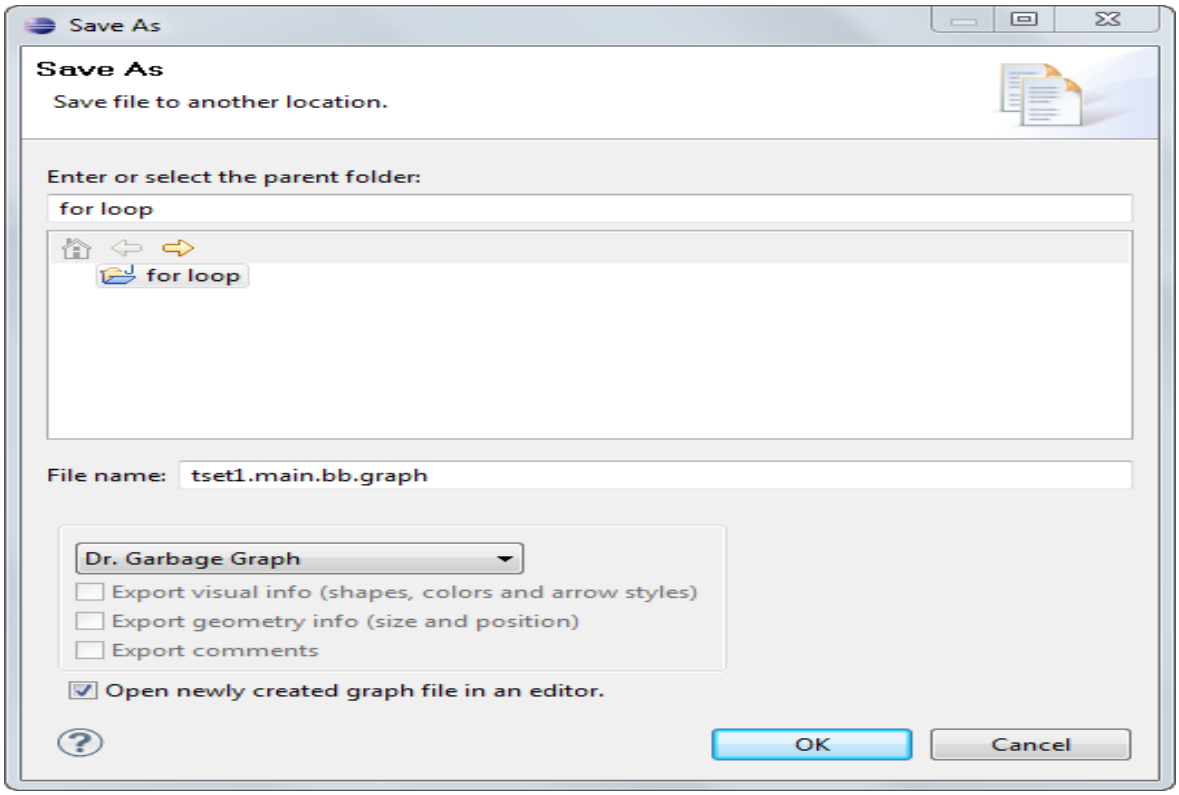


Figure 4.7: Save the Graph (G2) in to target folder.

c) For generated Basic Block Graph (G2).

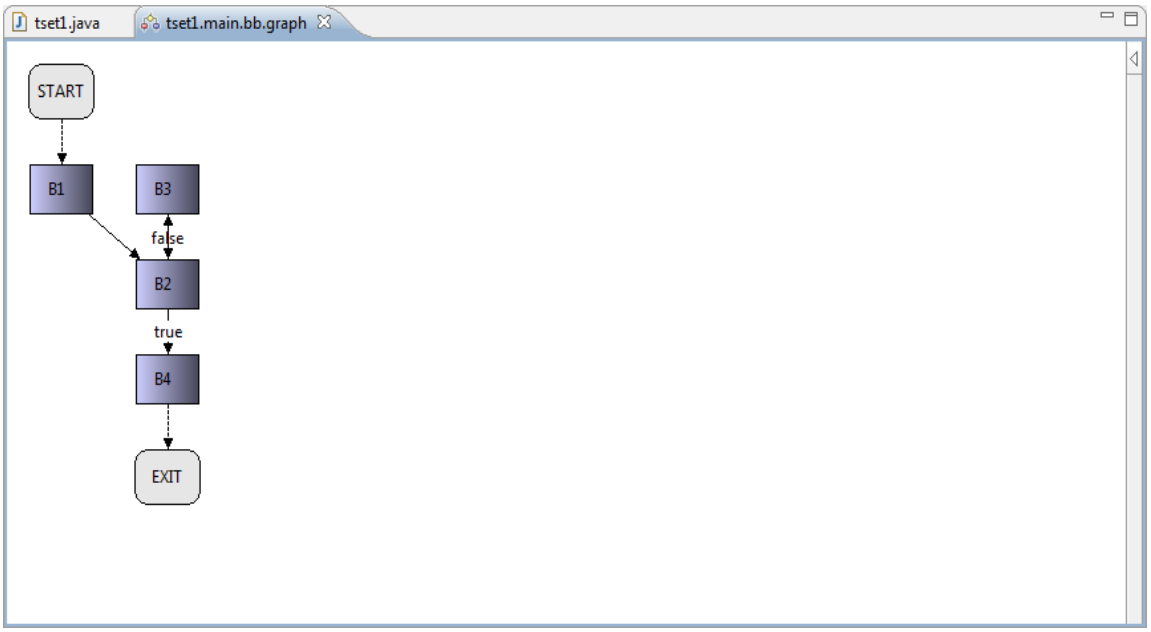


Figure 4.8: Generated Basic Block Graph (G2).

d) Exporting the Graph (G2) in Dot Format.

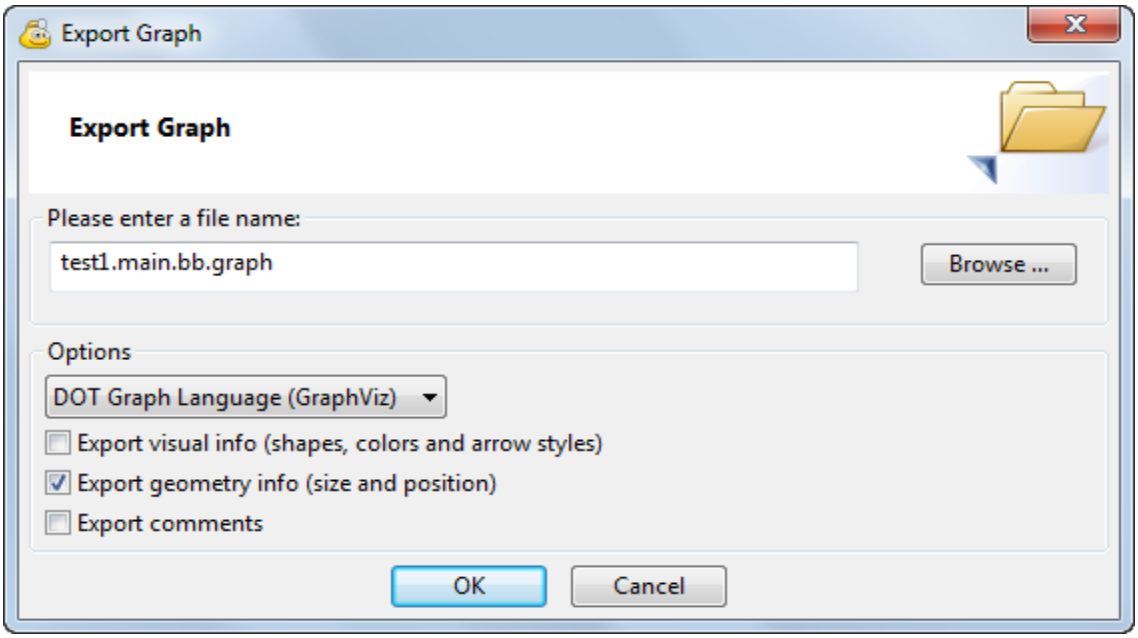


Figure 4.9: Exporting the Graph (G2) in Dot Format.

e) The Exported information of Graph (G2) in Dot format.

The exported information of graph (G2) in dot format, gives the information about all the vertices and edges which are connected with each other.

```
/* ------------------------------------------------- */
/* Generated by Dr. Garbage Control Flow Graph Factory */
/* http://www.drgarbage.com                          */
/* Version: 3.7.4.201206010948                       */
/* Retrieved on: 2012-07-07 15:11:08.078             */
/* ------------------------------------------------- */
 Graph "tset1.main.bb.graph" {
 digraph [label="tset1.main.bb.graph"];
 2 [label="B1", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 3 [label="B2", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 4 [label="B3", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 5 [label="B4", fixedsize=true, fontsize=12, width=0.48, height=0.36 ]
 6 [label="EXIT", fixedsize=true, fontsize=12, width=0.5, height=0.4 ]
 7 [label="START", fixedsize=true, fontsize=12, width=0.5, height=0.4 ]
 7 -> 2 [label="" ]
 2 -> 3 [label="" ]
 4 -> 3 [label="" ]
 3 -> 4 [label="false" ]
 3 -> 5 [label="true" ]
 5 -> 6 [label="" ]
 }
```

Step 3: Edit Graph

The Control Flow Graph Factory editor window provides some editing functions such as edit, copy and delete the graph elements. It also provides order layout to the elements of graph using the layout algorithms or layout the elements manually.

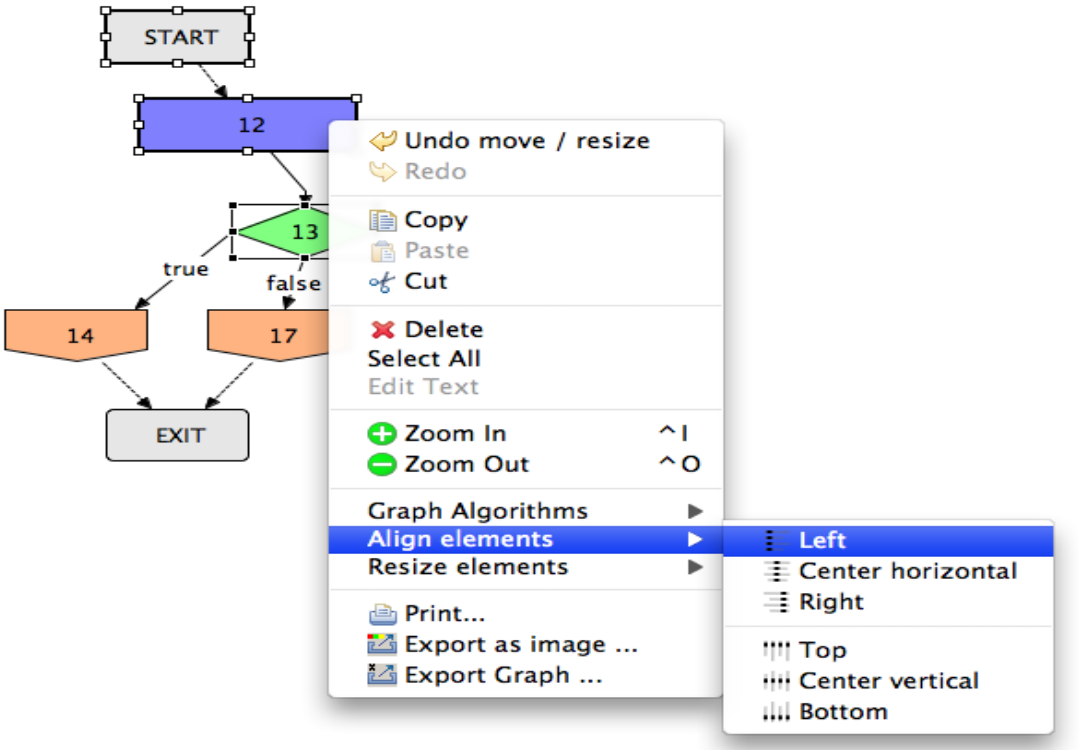


Figure 4.10: Editing of elements (Align elements).

Step 4: Implementation of Step 1 and 2 using Java Source Code.

a) Java Source code that use "While" loop structure.

```
Package test1;
Public class tes1 {
Public static void
main(String[] args) {
        int i=0;
      while(i<10)
      {
      System.out.print
("hello");
      }
      }      }
```

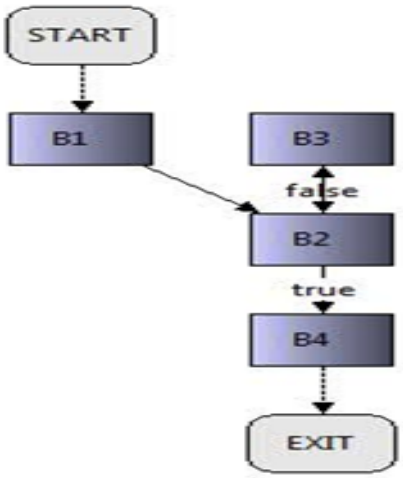


Figure 4.11: Basic Block Graph (G1) for the java source code.

b) Java Source code that use "For" loop structure.

```
Package test2;
Public class tes2 {
Public static void main
(String [] args) {
int i=0;
  for (i=0;i<10; i++)
{
    System.out.print
("hello");
    }
  }
}
```

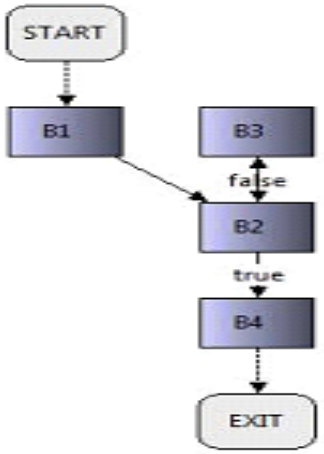


Figure 4.12: Basic Block Graph (G2) for the java source code.

c) The exported information of the Basic Block Graph (G1).

2 [label="B1"]

3 [label="B2"]

4 [label="B3"]

5 [label="B4"]

6 [label="EXIT"]

7 [label="START"]

7 -> 2 [label=""]

2 -> 3 [label=""]

4 -> 3 [label=""]

3 -> 4 [label="false"]

3 -> 5 [label="true"]

5 -> 6 [label=""]

In the following steps the exported information is used to find the out-degree of a graph.

i) An adjacency matrix (A) is created on the basis of exported information.

ii) Take the transpose of an adjacency matrix (A^T).

iii) Diagonal elements of the matrix are retrieved by the multiplication of an adjacency matrix (A) of the Graph G with its transpose (A^T) that provides the information about the out-degree of each node in the directed graph.

d) Adjacency matrix (A) for Basic Block Graph (G1) from the exported information.

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	0	1	0	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	0
7	1	0	0	0	0	0

Figure 4.13: Adjacency Matrix (A) for Basic Block Graph (G1).

This means 6 nodes are presented in the graph and

i) Node 7 is connected with node 2.

ii) Node 2 is connected with node 3.

iii) Node 4 is connected with node 3.

iv) Node 3 is connected with node 4.

v) Node 3 is connected with node 5.

vi) Node 5 is connected with node 6.

e) Transpose of Adjacency matrix (A^T) of Graph (G1).

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	0	0	0	0	0	1
3	1	0	1	0	0	0
4	0	1	0	0	0	0
5	0	1	0	0	0	0
6	0	0	0	1	0	0
7	0	0	0	0	0	0

Figure 4.14: Transpose of Adjacency matrix (A^T) of Graph (G1).

f) Multiplication of Adjacency matrix (A) with transpose of (A) [(A.A^T)] for Graph (G1).

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	**1**	0	1	0	0	0
3	0	**2**	0	0	0	0
4	1	0	**1**	0	0	0
5	0	0	0	**1**	0	0
6	0	0	0	0	**0**	0
7	0	0	0	0	0	**1**

Figure 4.15: Multiplication of Adjacency matrix (A) with transpose of (A) [(A.A^T)] for Graph (G1).

In the following steps the exported information is used to find the in-degree of a graph.

i) An adjacency matrix (A) is created on the basis of exported information.

ii) Take the transpose of an adjacency matrix (A^T).

iii) Diagonal elements of the matrix are retrieved by the multiplication of the transpose of an adjacency matrix (A^T) of Graph G with adjacency matrix (A) that provides the information about the in-degree of each node in the directed graph.

g) Adjacency Matrix (A) for Basic Block Graph (G1).

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	0	1	0	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	0
7	1	0	0	0	0	0

Figure 4.16: Adjacency Matrix (A) for Basic Block Graph (G1).

h) Transpose of Adjacency matrix (A^T) of Graph (G1).

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	0	0	0	0	0	1
3	1	0	1	0	0	0
4	0	1	0	0	0	0
5	0	1	0	0	0	0
6	0	0	0	1	0	0
7	0	0	0	0	0	0

Figure 4.17: Transpose of Adjacency matrix (A^T) of Graph (G1).

i) Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G1).

ROWS →

COLUMNS ↑

	2	3	4	5	6	7
2	**1**	0	0	0	0	0
3	0	**2**	0	0	0	0
4	0	0	**1**	1	0	0
5	0	0	1	**1**	0	0
6	0	0	0	0	**1**	0
7	0	0	0	0	0	**0**

Figure 4.18: Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G1).

The steps described in 4.1 generates the graph, extract the information from the graph, create the adjacency structure using extracted information and find the in degree and out degree at each node of the generated graph of the source code. In 4.2 an algorithm is proposed that implements the steps described in 4.1 and compares the nodes of the graphs on the basis of in degree and out degree.

This is the mathematical approach to find out the similar flow at the nodes of the graphs.

4.2 Proposed algorithm for finding the similar flow at the nodes of Graphs

Input: - 2d Array of strings

Output: - Matrix which contains the out-degree in 1st row and in-degree in 2nd row of a graph.

Algorithm Similar Flow Detection (array of strings)

```
1. for aa←1 to count_of_lines
2.    If text [aa].contains("->")
3.       adj [0] = Source_of_edge
4.       adj [1] = target_of_edge
5.       i=i+1
//Calculating minimum and maximum
6. for aa←1 to l
7.      for j←0 to 1
8.         if (adj[aa][j] > max)
9.            max = adj [aa][i]
10.        if (adj[aa][j] < min)
11.           min = adj[j][k]
//Forming Adjacency
12. for j←i-1 to 0
13.      adjacency [adj[j][0] - min][adj[j][1] - min] = 1
//Forming transpose
14. for j←0 to max-min
15.      for k←0 to max-min
16.         if (adjacency[j][k]==1)
17.            adjacencyT[k][j] =1
//Mutiplying Adjacency and its transpose for out-degree
18. for i1←0 to max-min
```

```
19.     for j←0 to max-min
20.         for k←0 to max-min
21.             out-degree[i1][j]  += adjacency[i1][k] *adjacencyT[k][j]
//Multiplying transpose and Adjacency for in-degree
22. for i1←0 to max-min
23.     for j←0 to max-min
24.         for k←0 to max-min
25.             in-degree[i1][j]  += adjacencyT[i1][k] *adjacency[k][j]
```

// forming a club-up matrix which contains the out-degree in 1st row and in-degree in 2nd row

```
26. for i1← 0 to max – min
27.      clubup[0][i1] = out-degree[i1][i1]
28.      clubup[1][i1] = in-degree[i1][i1]
```

Chapter 5

Experimental Results

The Experimental results are generated using the above algorithm (for finding the similar flow at nodes of Graphs) which takes the exported information of the graph as an input.

a) Adjacency Matrix (A) for Basic Block Graph (G1).

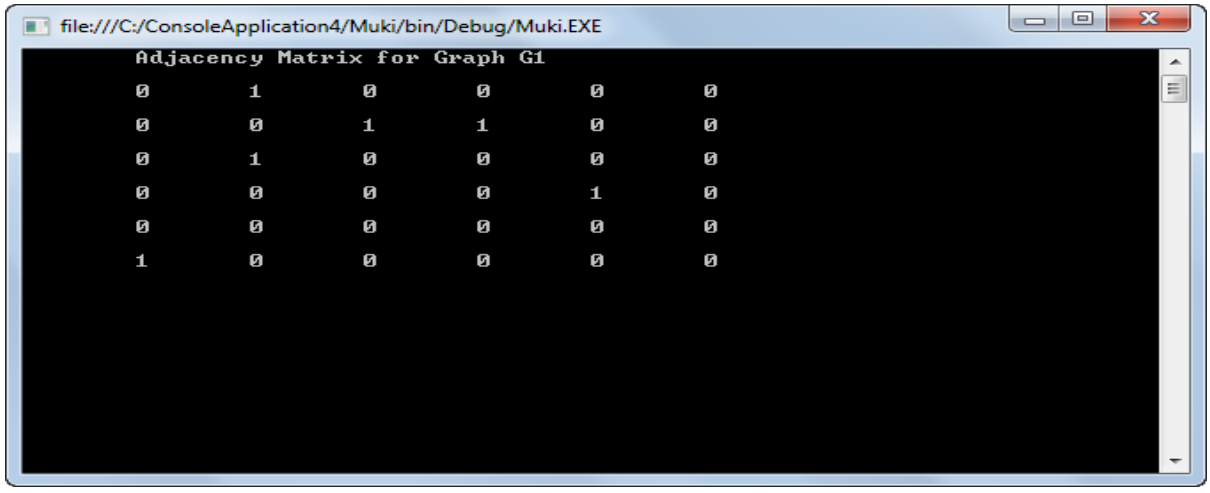


Figure 5.1: Adjacency Matrix (A) for Basic Block Graph (G1).

b) Transpose of Adjacency matrix (A^T) of Graph (G1).

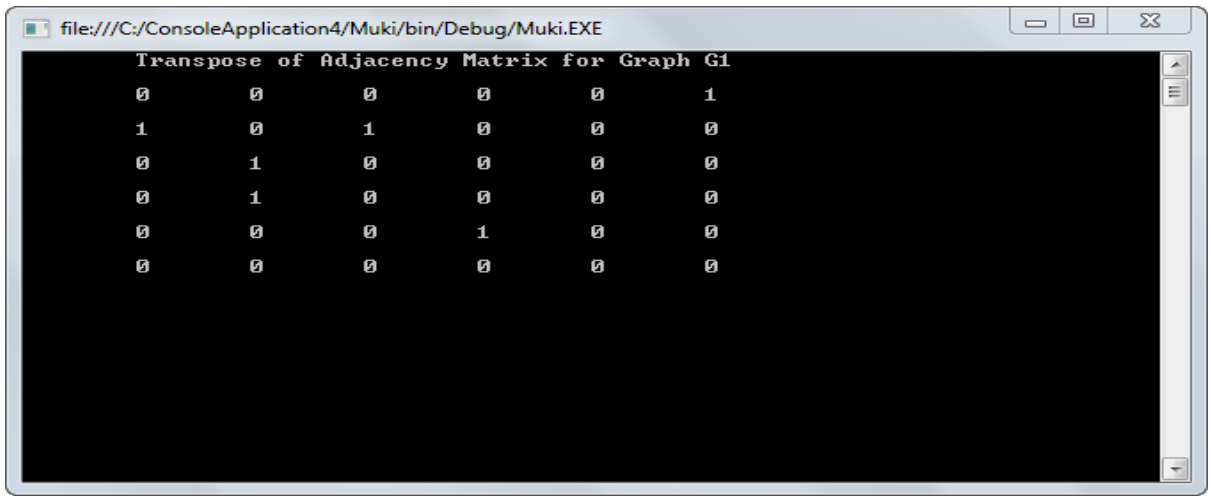


Figure 5.2: Transpose of Adjacency matrix (A^T) of Graph (G1).

c) Multiplication of Adjacency matrix (A) with transpose of (A) [(A.A^T)] for Graph (G1).

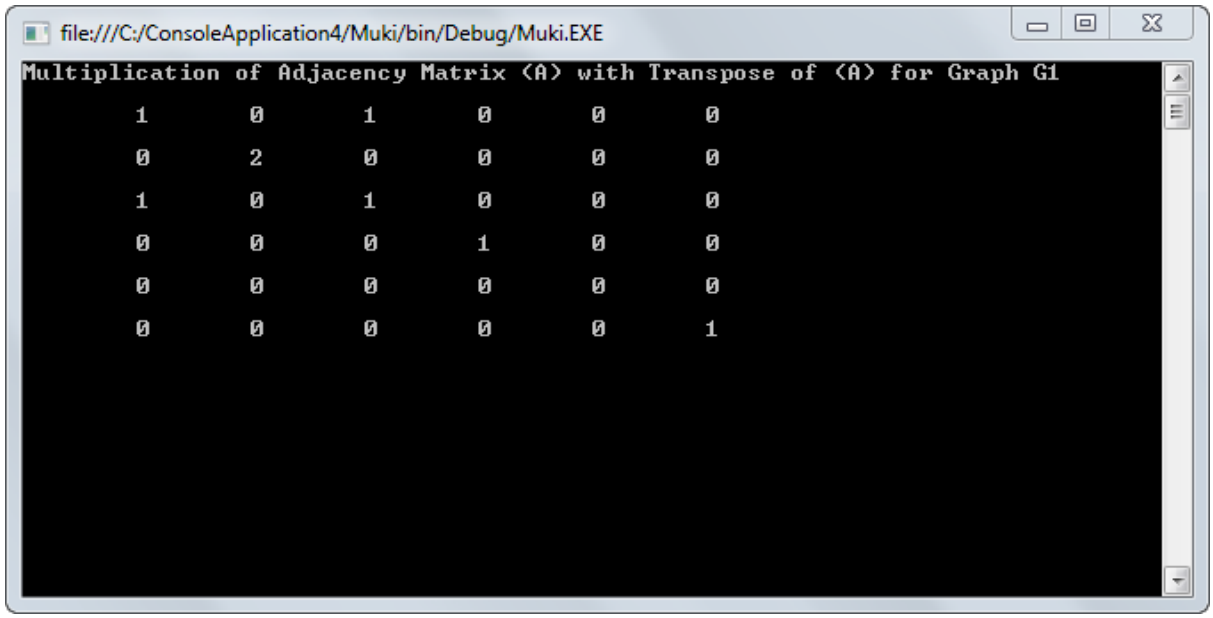


Figure 5.3: Multiplication of Adjacency matrix (A) with transpose of (A) [(A.A^T)] for Graph (G1).

d) Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G1).

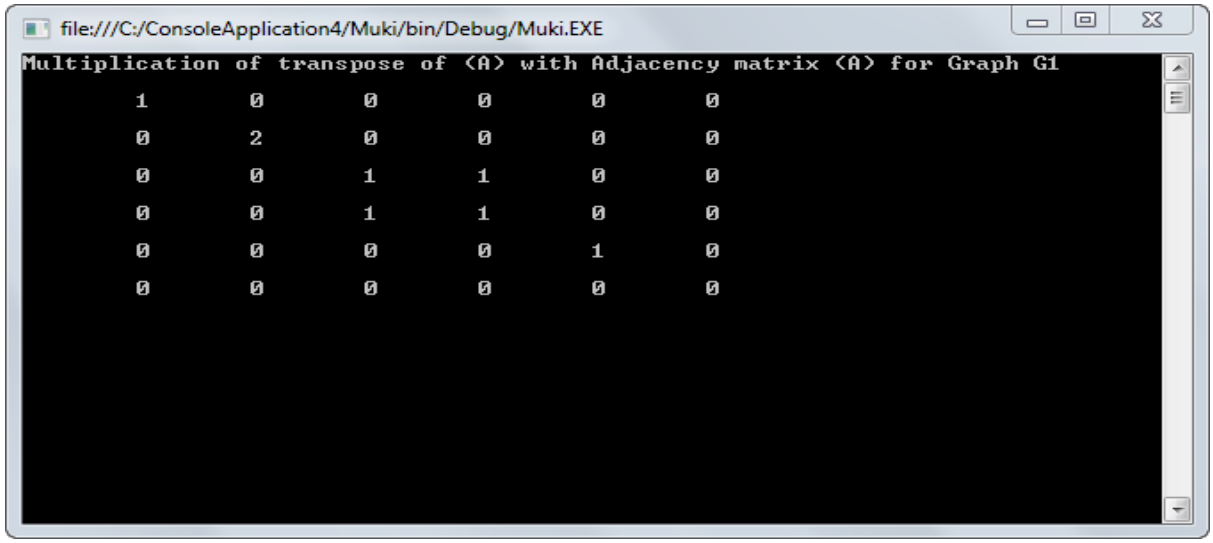


Figure 5.4: Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G1).

e) In degree and Out degree of Basic Block Graph (G1).

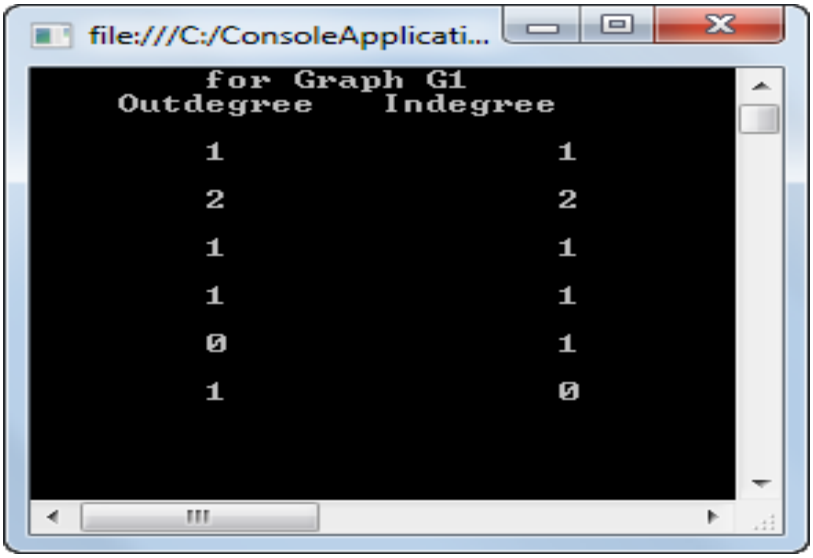


Figure 5.5: In degree and Out degree of Basic Block Graph (G1).

f) Adjacency Matrix (A) for Basic Block Graph (G2).

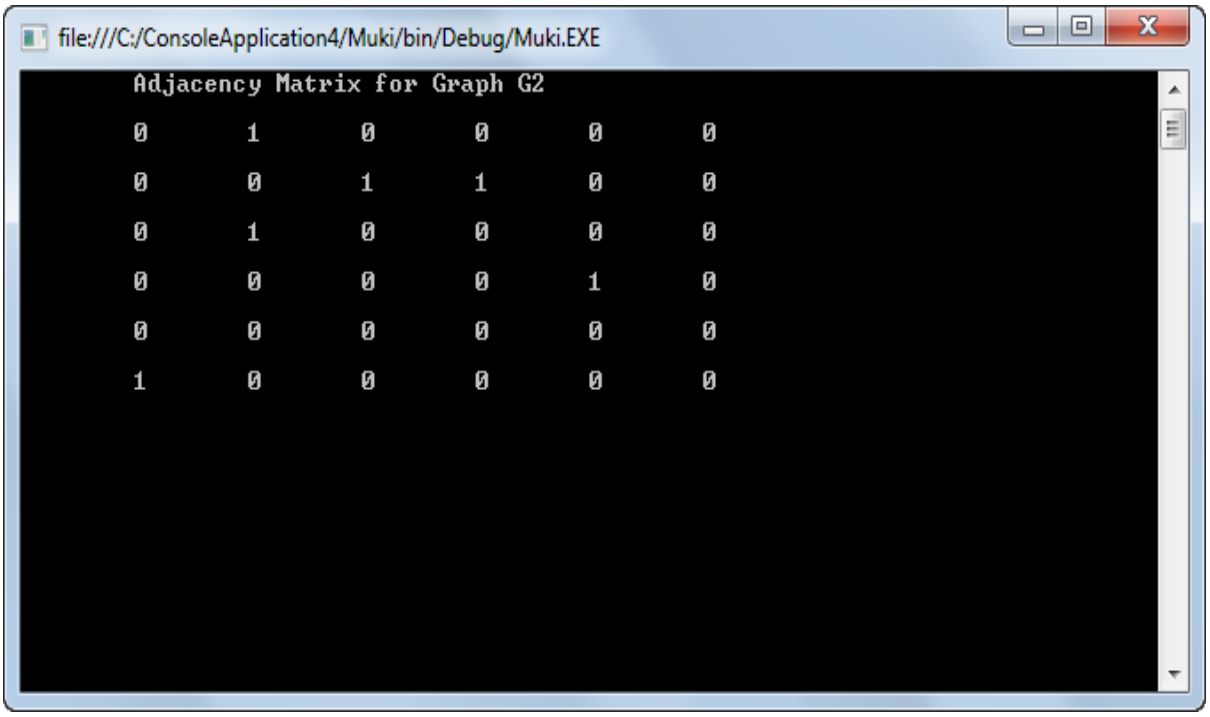


Figure 5.6: Adjacency Matrix (A) for Basic Block Graph (G2).

g) Transpose of Adjacency matrix (A^T) of Graph (G2).

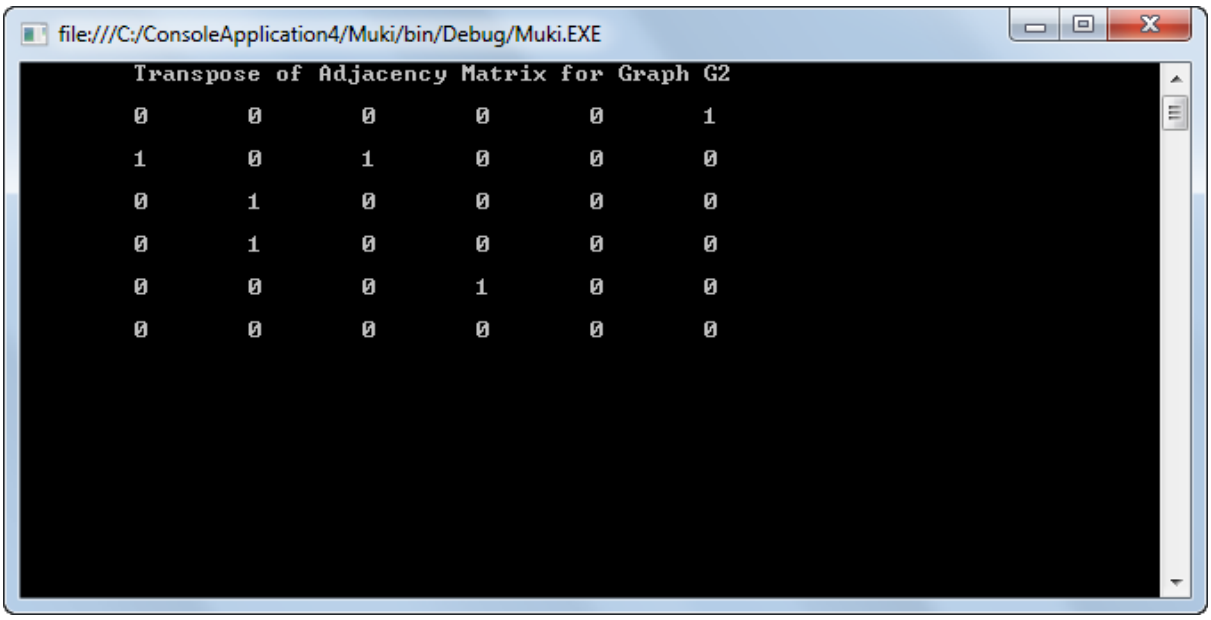


Figure 5.7: Transpose of Adjacency matrix (A^T) of Graph (G2).

h) Multiplication of Adjacency matrix (A) with transpose of (A) [($A.A^T$)] of Graph (G2).

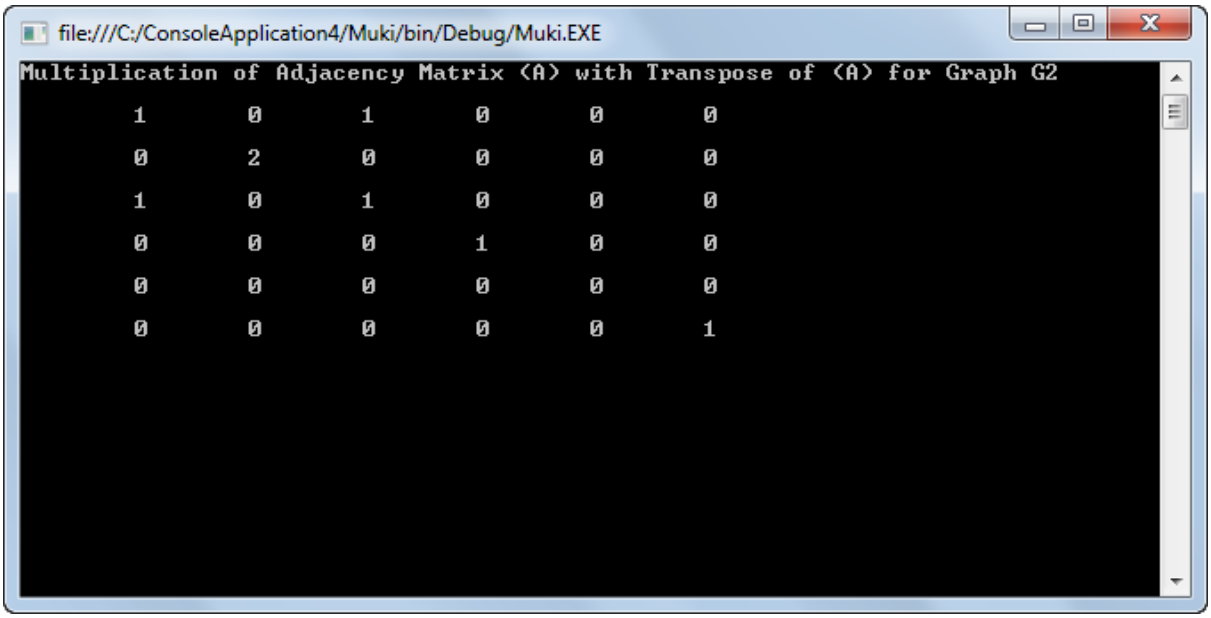


Figure 5.8: Multiplication of Adjacency matrix (A) with transpose of (A) [($A.A^T$)] of Graph (G2).

i) Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G2).

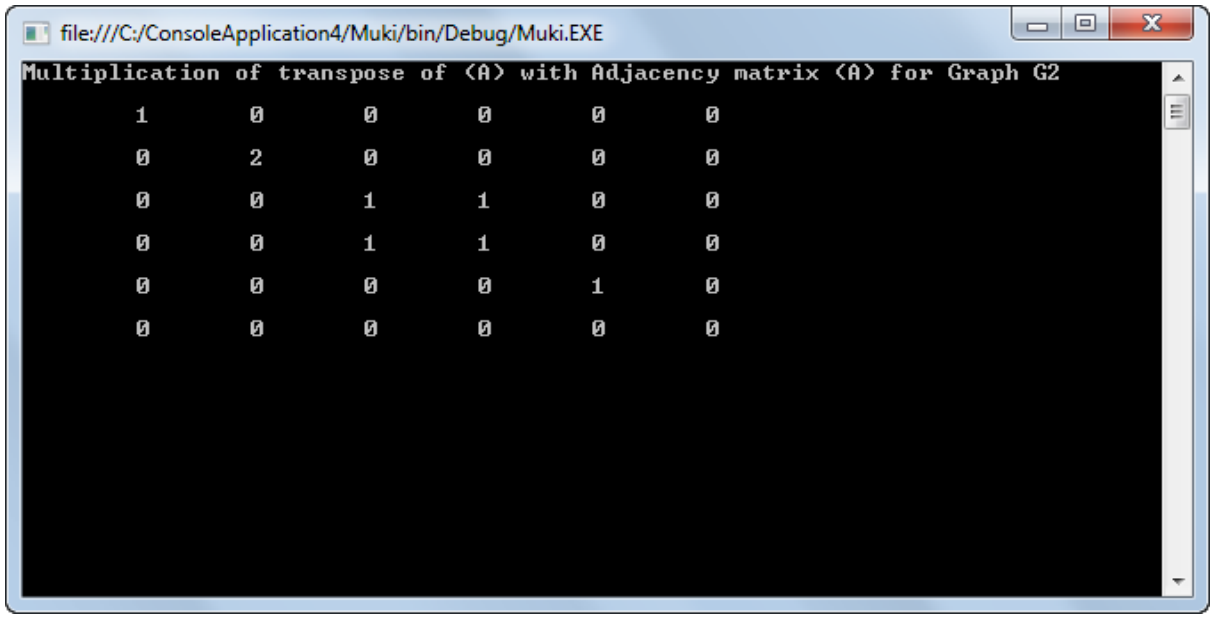


Figure 5.9: Multiplication of transpose of (A) with Adjacency matrix (A) [(A^T.A)] for Graph (G2).

j) In degree and out degree of Basic Block Graph (G2).

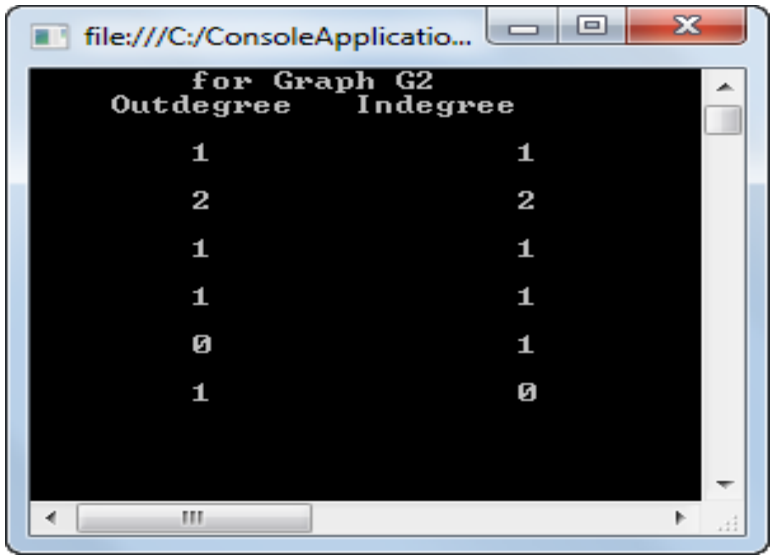


Figure 5.10: In degree and out degree of Basic Block Graph (G2).

k) Mapping similar flow of nodes of Basic Block Graph (G1) with Basic Block Graph (G2).

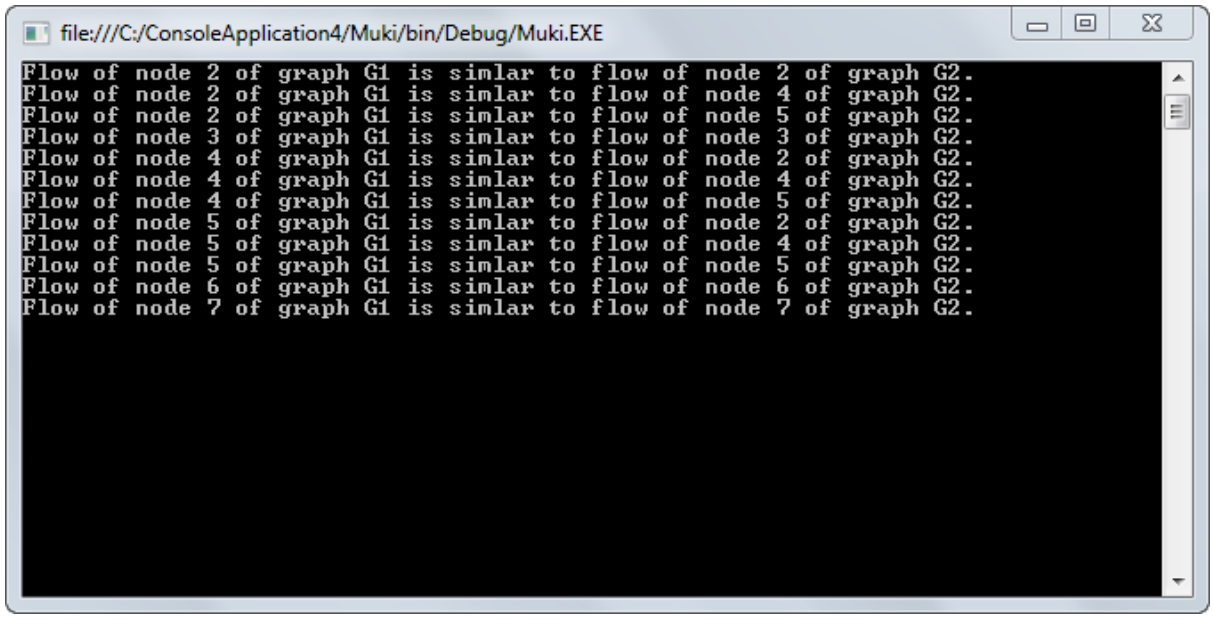


Figure 5.11: Mapping similar flow of nodes of Basic Block Graph (G1) with Basic Block Graph (G2).

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

Code cloning and its important characteristics have been discussed in this work. Several code cloning techniques are discussed in this thesis. This thesis focuses on code clones and detection of similar flows at the nodes of the directed graphs using adjacency structures and their properties. Further an algorithm is proposed and implemented in this work.

6.2 Thesis contribution

a) Code cloning and its research requirements are discussed in this work.

b) In this thesis currently available code cloning techniques like Text based, Token based, Metric based etc are discussed.

c) This work shows that existing approaches do not use the adjacency structures and their properties.

d) An algorithm has been proposed which detects the similar code fragments on the basis of the graphs and adjacency structures.

e) Proposed algorithm is implemented using dot net programming language in this thesis.

6.3 Future Scope

a) The Proposed techniques works only for the java source code files. This work can be extended to other languages.

b) Language Independent features can be incorporated in the proposed technique.

c) Proposed technique can be extended to other control flow graphs that represent the source code file.

References

[1] C.K. Roy, J.R. Cordy, "A survey on software clone detection research", Technical Report: 541, 2007, p. 115.

[2] B. Baker, "On Finding Duplication and Near-Duplication in Large software systems." In Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95), pp. 86-95, Toronto, Ontario, Canada, July 1995.

[3] I. Baxter, A. Yahin, Leonardo Moura, M. Sant Anna, "Clone Detection Using Abstract Syntax Trees." In Proceedings of the 14th International Conference on Software Maintenance (ICSM'98), pp. 368-377, Bethesda, Maryland, November 1998.

[4] M. Kim, L. Bergman, T. Lau, D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL." In Proceedings of 3rd International ACM IEEE Symposium on Empirical Software Engineering (ISESE'04), pp. 83- 92, Redondo Beach, CA, USA, August 2004.

[5] C.Kapser and M.W. Godfrey, "Clones considered harmful" considered harmful. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), pp. 19-28, Benevento, Italy, October 2006.

[6] M.W. Godfrey, D. Svetinovic, and Q. Tu. "Evolution, growth, and cloning in Linux: A case study". In CASCON workshop on Detecting duplicated and near duplicated structures in large software systems: Methods and applications, October 2000.

[7] J.R. Cordy, "Comprehending reality Practical challenges to software maintenance automation." In Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), pp. 196206, Portland, Oregon, USA, May 2003.

[8] H. Basit, D. Rajapakse, S. Jarzabek, "Beyond Templates a Study of Clones in the STL and Some General Implications." In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), pp. 15-21, St. Louis, Missouri, USA, May 2005.

[9] N. Davey, P. Barson, S. Field, R.J. Frank, "The Development of a Software Clone

Detector." International Journal of Applied Software Technology, Vol. 1(3/4):219-236, 1995.

[10] E. Burd, and M. Munro, "Investigating the maintenance implications of the replication of code." In Proceedings of the 13th International Conference on Software Maintenance (ICSM'97), Bari, Italy, September 1997.

[11] M. Rieger, "Effective Clone Detection Without Language Barriers." Ph.D. Thesis, University of Bern, Switzerland, June 2005.

[12] A. Walenstein and A. Lakhotia, "The Software Similarity Problem in Malware Analysis." In Proceedings Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 10 pp., Dagstuhl, Germany, July 2006.

[13] J. Howard Johnson, "Identifying Redundancy in Source Code Using Fingerprints." In Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON'93), pp. 171-183, Toronto, Canada, October 1993.

[14] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and evaluation of clone detection tools." IEEE Transactions on Software Engineering 33 (9) (2007) 577_591.

[15] M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones". In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, 2008, pp. 321_330.

[16] B. S. Baker, "A Program for Identifying Duplicated Code." In Proceedings of Computing Science and Statistics, 24th Symposium on the Interface, Vol. 24:4957, March 1992.

[17] B. S. Baker, "Parameterized diff" In Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA'99), pp. 854-855, Baltimore, Maryland, USA, January 1999.

[18] R. M. Karp, "Combinatorics, complexity and randomness Communications" of the ACM, 29(2):98109, February 1986.

[19] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms." IBM Journal Research and Development, 31(2):249260 March 1987.

[20] A. Marcus, and J. I. Maletic, "Identification of high-level concept clones in source code." In Proceedings of the 16th IEEE International Conference on Automated

Software Engineering (ASE'01), pp. 107-114, San Diego, CA, USA, November 2001.

[21] S.T. Dumais, "Latent Semantic Indexing (LSI) and TREC-2." In Proceedings of the 2nd Text Retrieval Conference (TREC'94), pp. 105-115, Gaithersburg, Maryland, March 1994.

[22] B. S. Baker, "On Finding Duplication in Strings and Software." Journal of Algorithms, 1993.

[23] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code." Transactions on Software Engineering, Vol. 28(7): 654- 670, July 2002.

[24] J.R. Cordy, T.R. Dean, N. Synytskyy, "Practical language-independent detection of near miss clones." In Proceedings of the 14th IBM Centre for Advanced Studies Conference, CASCON 2004, 2004, pp. 29_40.

[25] N. Synytskyy, J.R. Cordy, T.R. Dean, "Resolution of static clones in dynamic web page." In Proceedings of the 5th IEEE International Workshop on Web Site Evolution, WSE 2003, 2003, pp. 49_58.

[26] D. Gitchell, N. Tran, Sim, "A utility for detecting similarity in computer programs" SIGCSE Bulletin 31 (1) (1999) 266_270.

[27] C.K. Roy, J.R. Cordy, "NICAD Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization." In Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, 2008, pp. 172_181.

[28] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, "On software maintenance process improvement based on code clone analysis." In Proceedings of the 4th International Conference on Product Focused Software Process Improvement, PROFES 2002, 2002, pp. 185_197.

[29] L. Moonen, "Generating robust parsers using island grammars." In Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, 2001, pp. 13_22.

[30] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code." IEEE Transactions on Software Engineering 32

(3) (2006) 176_192.

[31] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and S. Jarzabek, "Efficient Token Based Clone Detection with Flexible Tokenization." In Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'07), pp. 513-515, Dubrovnik, Croatia, September 2007.

[32] W. Yang, "Identifying syntactic differences between two programs." In Software Practice and Experience, 21(7):739755, July 1991.

[33] V. Wahler, D. Seipel, J. Gudenberg, G. Fischer, "Clone detection in source code by frequent itemset techniques." In Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, SCAM 2004, 2004, pp. 128_135.

[34] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones." In Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, 2007, pp. 96_105.

[35] K. Kontogiannis, M. Galler, and R. DeMori, "Detecting code similarity using patterns." In Working Notes of 3rd Workshop on AI and Software Engineering, 6pp., Montreal, Canada, August 1995.

[36] J. Mayrand, C. Leblanc, E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics." In Proceedings of the 12th International Conference on Software Maintenance (ICSM'96), pp. 244-253, Monterey, CA, USA, November 1996.

[37] J.F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, "Extending software quality assessment techniques to java systems." In Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99), pp. 4956, Pittsburgh, PA, USA, May 1999.

[38] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino and P. Granato, "Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages." In Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS'99), pp. 107-113, Florence, Italy, November 2001.

[39] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino, "An approach to identify

duplicated web pages." In Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02), pp. 481486, Oxford, England, August 2002.

[40] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code." In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[41] J. Krinke, "Identifying Similar Code with Program Dependence Graphs." In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), pp. 301 309, Stuttgart, Germany, October 2001.

[42] C. Liu, C. Chen, J. Han and P. S. Yu. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis." In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06), pp. 872-881, Philadelphia, USA, August 2006.

[43] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization." ACM Trans. Program. Lang. Syst., 9(3):319349, 1987.

[44] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees." In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), pp. 253-262, Benevento, Italy, October 2006.

[45] R. Tairas, J. Gray, "Phoenix-Based Clone Detection Using Suffix Trees." In Proceedings of the 44th annual Southeast regional conference (ACM-SE'06), pp. 679- 684, Melbourne, Florida, USA, March 2006.

[46] K. Greenan, "Method-Level Code Clone Detection on Transformed Abstract Syntax Trees using Sequence Matching Algorithms." Student Report, University of California - Santa Cruz, Winter 2005.

[47] M. Datar, N. Immorlica, P. Indyk, and V. S.Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions." In Proceedings of the 20th annual symposium on Computational geometry (SoGG'04), pp. 253-262, Brooklyn, New York, USA, June 2004.

[48] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, M. Bernstein, "Pattern matching for clone and concept detection." Journal of Automated Software Engineering 3 (1_2) (1996) 77_108.

List of Publications

1. Mukesh Kumar, Rajkumar Tekchandani, "Code Clone Detection Using Graphs and Adjacency Structures", 2nd international conference on communication, computing and security ICCCS 2012 (Communicated).