Graph Theoretical Algorithms For Structural Comparison Of Java Source And Byte Code

---

Submitted By

Artem Garishin

FB2: Faculty of Computer Science and Engineering

This thesis presented for the degree of
Master of Science
in the

High Integrity Systems

Research Supervisor: Prof. Dr. Sergej Alekseev

Co-Supervisor: Prof. Dr. Matthias Wagner

November 2014

Legal Declaration

I declare that this thesis document is completely my own work and all used references have been clearly cited. I have not submitted this assignment in the context of an examination to any other examination board or person.

Signature: _____

Location, Date: _____

Abstract

Goal of this work is research for the most optimal ways to compare similar and at the same time different fragments of code. So far there are two techniques of code comparison: textual-based and structural-based. Textual compare can be to some extent insufficient to analyse difference or to find a code similarity between given pieces of code. For that reason, a structural comparison is being investigated and improved in regards with clone detection, plagiarism and code similarity.

Acknowledgments

I would like to take this time to thank Frankfurt University of Applied Sciences for all of the resources which they provided me in order to pursuing my master study in computer science and make this thesis possible.

I would like to express my sincere gratitude to Prof. Dr. Sergej Alekseev and Prof. Dr. Matthias Wagner for their patient guidance, encouragement and advice which they provided me throughout this thesis work.

Contents

List of Figures

List of Tables

Abbreviations

AST	**A**bstract **S**yntax **T**ree
BCV	**B**yte **C**ode **V**isualizer
BFS	**B**readth **F**irst **S**earch
BUMC	**B**ottom **U**p **M**aximum **C**ommon
CDS	**C**ontrol **D**ependency **S**ubgraph
CFGF	**C**ontrol **F**low **G**raph **F**actory
DDS	**D**ata **D**ependency **S**ubgraph
MBWM	**M**aximum **B**ipartite **W**eighted **M**atching
PDG	**P**rogram **D**ependecny **G**raph
SCV	**S**ource **C**ode **V**isualizer
TDMC	**T**op **D**own **M**aximum **C**ommon
TC	**T**ext **C**ompare

Chapter. 1

Introduction

Code maintenance is one of the most important component in computer project development. It is obvious, that the whole development process is based on re-usage and adjusting of existing code fragments. But practically, it is observed that mostly the code duplication and further appliance are not proper used. Large software projects are developed by many people for years and at some point there are problems in applications since code was not proper supported. These issues like the code duplication and then following re-usage by copy-pasting with or without any modifications is a well known issue in computer software maintenance.
These problems of disordered code use originally come from tendency directly copy fragments of code that implement something similar for personal needs and by performing small correction. The code is adapted to the new application and the code is already reused. Secondly, to copy a code fragment (which may be already tested for functional purposes) is easier and faster than to implement the code from scratch.
Many studies say that about 5 to 20 percent of a computer software systems can contain a duplicated code fragments. One of the major disadvantages of such duplicated code fragments is that in case of bug detection in a code fragment itself, all the other fragments similar to it should be investigated to check the possible existence of the same bug in the similar fragments. Another problem is the code redundancy, particularly unknown pieces of reusable code are located in different places that makes code uncomprehending for further development.
One more not less important problems nowadays is plagiarism detection. Most occasions of plagiarism are found in documents are typically articles, reports or essays. Nonetheless, plagiarism can be found in any field, including scientific papers, art designs and source code. In this work, objectively is about source code

duplication and their methods to find out similar or same fragments of source code. Java is one of the most well known programming languages and there are many open-source projects available in open network. Indeed, the majority of them are developed not from scratch and there are many methods and functions that have been created by copy-pasting.
Clone detection and following re-factoring of the duplicated code is theme for discussion in software maintenance. Although practically restructuring code without changing its external behaviour of certain clones are not desirable and there are a risks when these duplication have been removed. However, it is widely agreed that clones should at least be detected and somehow taken into consideration.
Modern methods to compare of code fragments are used to analyse codes changing, to explore the development process and to optimize the implementation. Basically, in current tools or plug-ins only text compare methods are used, for example embedded text-to-text compare feature. In several cases that is not full sufficient to define clones or moreover to comprehend where are clones, and how they organized. Sometimes another techniques can be very helpful for such purposes. One of them is a structural code compare, based on building a tree structures, and methods to compare any similar or same structures.
Portions of the source code for some proprietary software are leaked to the Internet. This could be evidence of the frequently made assertion by proponents of open source that it is difficult to keep source code in secret. The most remarkable examples of code clones have open source projects as MySQL and PostgreSQL database management systems. The same way open-source operation systems Linux and FreeBSD have been implemented. The table 1 presented from the research of clone detection tool in [15] shows the figures of code clones these projects.

Project	**Number of segments**	**Copy-pasted %**
Linux	198,605	23.0%
FreeBSD	153,230	20.4%
Apache	6,196	17.7%
PostgreSQL	16,662	22.2%

Table 1.1: The table shows

As presented in the table, in Linux and FreeBSD, there are around 200,000 and 150,000 copy-pasted segments without any modifications. This percentage composes circa 20% of the whole written, here and there copy-pasted code. In other words, the quarter code of these four software products is practically identical.

Chapter. 2

Description of problem

In this chapter an issue of actual topic in the master work is explained. As usual, a comparison of two code fragments considers a selection of classes, functions or methods to be in between compared. Thereby, a compare can be counted as examinations of two pieces of code, in the most convenient way - methods or functions. They can have a similar implementation or alike syntax at first sight, however these two pieces of code are different. As well, the pieces of code can be completely varied and therefore this variety needs to be investigated.
There are many purposes to compare a code, to find out a similarity or determine a difference between them. One of the option is to search for plagiarism in case code can be taken from external source and a variables have been changed. This case called code clone, by reason that has identical implementation, i.e same logic. In addition to the general plagiarism detection it can be improved to look out a similar code in big projects. This is needed to search out for an equal code chunks and by that reduce superfluity.

2.1 Insufficiency of textual comparison approach

Possible result of this thesis is development of concept or and idea to find out code difference using graph theory. To get started, the introduction why textual comparison is not sufficient, a small example demonstrates, what kind of result delivers normal text compare.

```
    public void method1(){
    if(i > 1) i++;
}

public void method2(){
    if(i > 1)
    i++;
}
```

In this example there are two pieces of code presented methods `method1` and `method2` in Eclipse compare editor. One enter symbol after line `if(i > 1)` takes place in `method2`. Therefore, the result of the codes looks different. Even a calculation of statistic how much code is different, the a preliminary answer two rows is wrong and not precise. Using simple text-based approach of compare, only these gap will be found, however this difference does not play any role regards to application logic. The suspicious piece of code can be written in one string and textual compare editor is not able to deliver proper results.

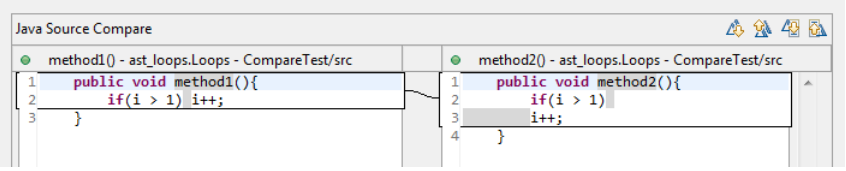


Figure 2.1: The simplest example of text-to-text comparison indicating that text compare sometimes is not enough to investigate code difference, however there are no changes regards the application logic

2.2 Structural graph comparison is NP-complete problem

Since the code is a clear text in abstract form, than code difference can be introduced as a composition of two compared objects and highlighted imposed snippets on both texts. This trend of comparison called textual compare when strings of code are collated to each other and mismatches are revealed. Such way to discover difference has drawbacks. The chapter Existing Methods of Code Comparison, section Text-based techniques 3.2 discloses their shortages. Another way is approach using graph theory, or by another words the structural comparison.
The structural comparison stands for comparing data graphs structures. There are some techniques how to create a graph from Java source code. In order to find some structural similarities this two created graphs must be compared. For this purpose there are existing algorithms to figure out graph isomorphism. But unfortunately, their execution time is exponential complex. However, trees data structures can be compared in polynomial time, that is not reachable for graphs.

The graphs can contain loop inside, i.e. an edges that connects a vertex to itself. Basically, these types of graphs as program flowchart or program dependency graphs are circular graphs.
If two graphs are compared with each other to figure out graph isomorphism then it is NP-problem. Graph isomorphism between two graphs G_1 and G_2 is a bijection between the vertex sets $f : V(G_1) \to V(G_2)$ such that
$\forall v, w \in V(G_1)$: $\{v, w\} \in E_1 \iff \{f(v), f(w)\} \in E_2$.
In the complex theory the worst case running time of all known algorithms is of exponential order, and just for certain special types of graphs, polynomial-time algorithms have been devised [7]. Maximum common sub-graph isomorphism is an optimization problem that is known to be NP-hard. The formal description of the problem is as follows: there are two input graphs, respectively the maximum common sub-graph isomorphism MCSGI(G_1, G_2):

- Input: Two graphs G_1 and G_2.
- Question: What is the largest sub-graph of G_1 isomorphic to a sub-graph of G_2 can be found?

The associated decision problem, i.e., given G_1, G_2 and an integer k, deciding whether G_1 contains a sub-graph of at least k edges isomorphic to a sub-graph of G_2 is NP-complete [7].This type of graph comparison is very expensive from a computational point of view and thus an action must be taken into account to reduce the domain of comparison before performing the actual comparison. To reduce this problem, luckily, a tree comparison is able to executed in polynomial time. Moreover there are some existing algorithms to investigate trees for isomorphism that presented in this paper.

Thus, there are no deterministic algorithms to compare graphs because of loops inside. In this case, the input code can be transformed into flowchart graph firstly, after the graph creation, it must be converted into tree, using simple techniques for example removing back edges. The back edges in the input graph are edges, which point from a node to one of its ancestors. Another approach is usage of AST trees obtained from source code. In fact, none of extra operations like removing of back edges required that can beneficial to use AST trees.
Under those circumstances the following techniques are researched and presented in the paper. After these prerequisites, a some questions can be inquired:

1. How to optimize the search of code for similarity?
2. How to compare these trees to get reasonable results?

3. How to reference code pieces and nodes, respectively how to put the code difference by the most elegant way?

Regards to the first question, the concept of idea is described in chapter **Techniques to normalize AST trees** 6. The second question comprehends existing algorithm and their combination and improvements in chapter **Maximum common subtree isomorphism algorithms** 4. The last issue is about how to lead back the result of the code and is stated in chapter.

Chapter. 3

Existing comparison methods for plagiarism detection

3.1 Plagiarism detection methods

Task of plagiarism detection is an identification of text's similarity. Thus, a research of existing methods is useful for this work regards code's comparison.

> Detection of plagiarism can be either manual or software-assisted. Manual detection requires substantial effort and excellent memory, and is impractical in cases where too many documents must be compared, or original documents are not available for comparison. Software-assisted detection allows vast collections of documents to be compared to each other, making successful detection much more likely. [5].

Mostly the task of plagiarism detection is considered for many fields, like text documents, software and source code. In this chapter source code plagiarism is being reviewed. According to the article of Chanchal Kumar Roy and James R. Cordy [6], source-code similarity detection algorithms can be classified as following:

- Text-based Techniques
- Token-based Techniques
- Tree-based Techniques
- PDG-based Techniques
- Metrics-based Techniques

According to this research the above methods are described in this chapter. The following conclusions takes place after every method to overview pros and cons of these methods and gain a profit from them to improve existing algorithms.

3.2 Text-based Techniques

Text based techniques are widely used to compared pieces of text. For this purpose they can be valuable to compare java source code but sometimes insufficient. This section shortly explains how it functions and derived advantages and disadvantages from chapter experiments 5 characterized. Used technique in text-to-text comparison applies so-called tree suffix algorithm, designed to compare strings. General idea of this suffix tree comparison algorithm: the code is split into strings, afterwards from these strings a suffix tree is being built, then using sub-suffix algorithm performs a search for substring from one code fragment to another. Building such kind of tree allows to find a sub-string in given string within $O(m)$ complexity, where m is the length of the sub-string (but with initial $O(n)$ time required to build the suffix tree for the string, where n is input string).

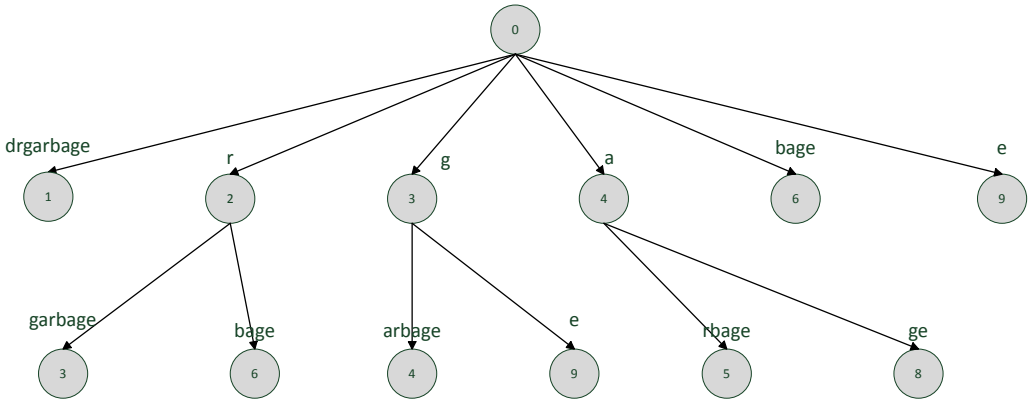


Figure 3.1: Example of suffix-tree data structure decomposed on word **drgarbage**. The number inside each node indicates index of first letter. The text under each node means a sub-string depending on parent node.

The figure 3.1 shows the built suffix tree based on word **drgarbage**. The tree has been created by so proficient way, that sub-string search is minimized. Without such tree structure the worst case of search considers to iterate all possible combinations, that is $O(n^2)$. For example, to find a whether the word *garbage* enters in the $drgarbage$, the algorithm looks first for the first letter of search string - g, it was found, then there only 2 possible combinations to consider, since g has two child-nodes. The rest of search words is sliced till *arbage*. If there is a leaf-node with such content then the word is found. For instance, *garbaga* cannot be found since there are no leaf node with the matched content.
Thus the algorithm is being applied for normalized text after steps above. The text comparison splits each i line of code and compare the string with i line of code of other fragment. From the first string the suffix tree is built. The second line of code as a search-string is interpreted. The suffix tree is traversed looking for search

string occurrence. From performed experiments from chapter of experiments 5 the next conclusion can be derived.
Advantages of text-to-text comparison using tool Eclipse compare editor:

- The difference direct in text represented.
- Every small mismatch is noticed and highlighted.
- High performance due to optimal formed suffix tree, that the match of search string is for $O(m)$ complexity executed, where m is length of searched substring.

Disadvantages of text-to-text comparison using tool Eclipse compare editor:

- If lines of the same command are different, comparison says it is mismatch.
- Changing the order of command or variable's names bring almost full mismatch.
- If sequence of commands and names of variable have been changed or modified then text highlight result is too jumbled.

In order to avoid line's mess, the graph theory can be very helpful to make an application logic more independent from text, therefore to improve quality of text difference representation. The paragraph 8 interprets an idea how AST tree assists to specify precisely match/mismatch of command. This kind of algorithm is a beneficial part to compare clear texts, that do not have logic.

3.3 Token-based Techniques

Using this technique the input code is firstly prepared with lexical analyzer. The entire code is lexed, parsed and transformed into sequence of tokens. This principle is advanced text-to-text comparison concept with steps before text comparison starts the preparation of code includes following steps:

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

One of the leading token-based techniques is CCFinder from Kamiya[10]. Firstly, each line of source files is divided into tokens by a lexer and the tokens of all source files are then concatenated into a single token sequence. The token sequence is then transformed, 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 each identifier related to types, variables, and constants is replaced with a special token[10]. This identifier replacement makes code fragments with different variable names clone pairs [10]. A suffix-tree based sub-string matching algorithm is then used to the similar sub-sequences on the transformed token sequence where the similar sub-sequence pairs are returned as clone pairs/clone classes[10]. When the clone pair class is obtained according to the token-sequence, the original code must be mapped regards to clone pair/clone class information.

1. **Lexical analysis** - each line of code is divided into tokens depending on programming language. The tokens are parsed via lexical analyzer which performs formatting of tokens, the rules described above reconstructing the source code is.

2. **Transformation** - the process includes two sub-processes. During the sub-processes the meta information regards mapping to original source code is kept. Thus the original code is restructured without loss of link-identifiers.

   2.1. **Transformation according to determined rules** - the token sequence is reorganized pursuant to some rules. One of them is conversion of compound block. For example the code: `if(i == 1) i = i + 1;` is being transformed into: `if(i == 1) {i = i + 1;}` covered within brackets. All other rules can be programming language dependent.

   2.2. **Parameter replacement** - each identifier related to types, variables and constant is replaced with special token[10].

3. **Match detection** - the token sequences are searched for equivalent pairs and marked as clones with suffix-tree algorithm [3.2]. Each clone is divided by indices in four parts: LeftBegin, LeftEnd, RightBegin, RightEnd. These metrics are indices of leading clone for mapping for according positions in the following clone.

4. **Formatting** - the clones in original code are highlighted according results from previous step.

These results followed by [10] provide confirmatory evidence that native text comparison [3.2] can be significantly improved. Simple based normalization of code structure, namely transformation in tokens, brings better results as compare to native text comparison [3.2].

3.4 Tree-based Techniques

The research from Chanchal Roy [6] offers an alternative to figure out differences using trees derived from logic of code itself. The complex trees are formed from source code and parsed to find mismatched nodes. The most remarkable distinction between text-to-text compare and trees is logic of tree building. In other words, the text compare assumes also a suffix-tree creation, however this technique is based on input strings, i.e the structure of the tree depends on letters and sub-string 3.2. The alternative approach is tree creation directly from source code, based on programming language. It encompasses building a branch of trees not by one string line, but by a combination of programming command structures. Such logic of tree creation called Abstract Syntax Tree (in short, AST). The abstract syntax tree analysis is more accurate than a line by line analysis or programming language token based approach 3.3 due to the fact that it builds the abstract syntax tree.

The proposed clone detection solution from [11] can be organized into few steps:

1. Parsing via source code and create non-optimized AST tree.
2. Each sub-tree of AST tree is hashed and grouped by in different buckets based on their hash value.
3. Apply the clone detection algorithm that three sub-algorithms:
   - 3.1. Basic Algorithm finds sub-tree clones compares every sub-tree with another sub-tree for equality. The algorithm uses parsing both AST trees finding out clones with exact equality and excludes the possibility to find near-miss clones. The complexity of algorithm offered from Baxter [12] is $O(N^3)$. This can be reduced using hash-value in node parents of sub-tree. Thus each parent node keeps a hashed value about sub-tree itself. For example, a parent node contains following tree structure: `for(int i = 1; i <= 10; i++ ){ b = b + 1; }` that represents a complex sub-tree. The hash-value of structure is `f4caf1dd3e33c53d971f0e18f72249e0`. During tree traverse the hash values of corresponding nodes are compared. If these hash values are same there is no need to check the whole sub-tree. Otherwise sub search is being further proceed. This technique is used to optimize search of equal sub-structures between AST trees. In most cases it contributes up to 30% performance, since searched projects keep clones.
   - 3.2. Sequence Detection - is a part of clone detection the algorithm that handles the detection of code clones within sequences of statements or declarations [11]. When parsing a source code fragment, all sequences are

stored in a sequence list for future usage. The first step named sub-sequence algorithm, takes care of detecting code clones inside each sequence, and verifies if two sub-sequences of the same sequence are clones.

3.3. Generalization algorithm - the algorithm takes care about detection near-miss clones. It verifies whether the parent nodes of detected clones belong to near-miss clones.

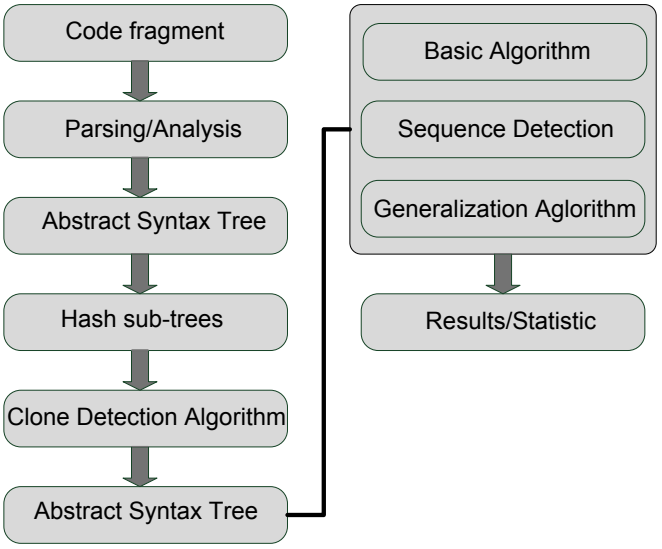


Figure 3.2: The flowchart demonstrates general steps of code comparison with AST trees. Once the Abstract Syntax Tree has been created, a sequence of algorithms is applied.

In summary, tree-based approach is the most flexible tool to compare source code because only this fragment of code has a syntax and structure. Consequently a flexible tree can be built based on syntax itself.
Advantages of Abstract syntax tree comparison:

- Possibility to investigate for clones in sub-expressions on high abstract level
- Semantic analysis during tree assembling
- Independence form comments, spacing, or other non-semantic changes

Disadvantages of Abstract syntax tree comparison:

- Execution time, comparing sequences of trees the computational process is $O(N^4)$. But due to hashing of buckets it can be significantly reduced.
- Complexity of algorithms to parse trees and figure out clones.
- So far this algorithm have been used only to identify clones and add/remove sequences of clone to optimize projects. For small examples of code there are yet an algorithm to represent clones textually. This concept is described in section **Text comparison improvement with AST trees** 8.

- The level of abstraction is not so high to handle application process of variables. For example, swapping of command changes the final result, however AST does not care about variable data flow.

To conclude this way of compare, as described above, it allows to differ even small clones between two pieces of code. This technique is not suitable for pure text since text has no logic and structure. AST tree based technique is not fit with Java Byte code since Java byte-code has not so much complex structure to form an AST tree.

3.5 PDG-based Techniques

PDG based techniques for clone detection is most suitable to detect for non-contiguous code clones while other clone detection technique as AST in section 3.4, text and token methods are not. The most remarkable difference of other approaches is logical content of graph, therefore the dependencies among variables and methods. This method sets an effective testing to discover and locate the redundant functional modules and the unreachable paths based on dependency relationship.
A program dependency graph (in short, PDG) is a directed graph $G = (V, E)$ contains the control flow and data flow and represents the dependencies between program elements (statements). Respectively a PDG nodes V_i is a program element and a PDG edge E_i indicates a dependency between two nodes. PDG is considered as a combination of two different layer sub-graphs: a data dependence sub-graph (in short, DDS) and a control dependence sub-graph (in short, CDS). After PDG graph creation, a matching algorithm is applied to find a similar sub-graphs which are returned as clones.

```
1  private static String sample(){
2      String text = null;
3      int i = 0;
4      while(i < 10){
5         i++;
6         text.compareTo("my text");
7         text.trim();
8    }
9    return text;
10 }
```

Labels attached to the nodes mean the lines where their elements located in the source code. The node labeled $< 1 >$ is the enter node of the PDG graph. This example example demonstrates data dependencies between nodes using variables. Node $< 2 >$ has initialization of text variable `text`, this variable is used in lines

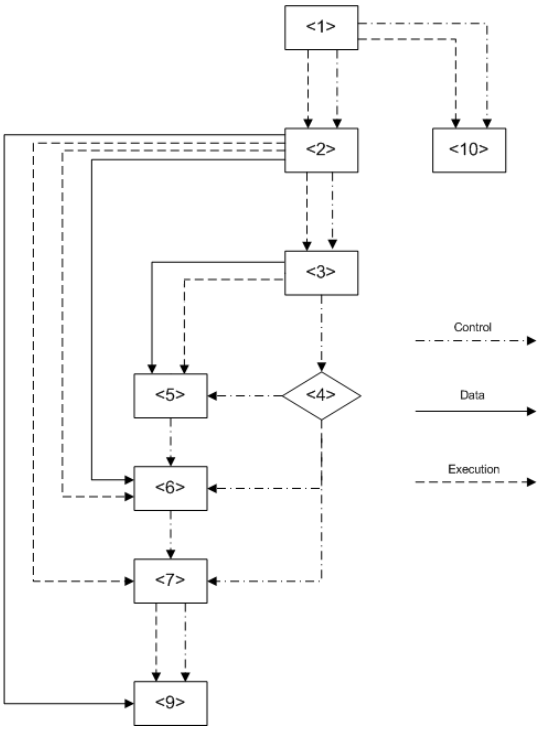


Figure 3.3: Program Dependency Graph derived from Java source code

6 and 7, therefore there are edges from node $< 2 >$ to nodes $< 6 >$ and $< 7 >$ respectfully. The dashed line from $< 2 >$ to $< 6 >$ and $< 7 >$ indicates that some methods on this variable used. Moreover the node $< 6 >$ has data edge because of input parameter of method `compareTo()`. The loop $< 4 >$ keeps under execution all actions inside.
The algorithm for clone detection offered from [13] is based on Komondoor's Method[14]:

1. All nodes are hashed according to their content, similar as in AST tree compare 3.4. Nodes with the same hash value are considered as equivalent class. The hashing process is independent from name of variables, thus only syntactically identical program elements have the same hash value.

2. The second step the pair of root nodes of all equivalence classes is identified. In other words roots (r_1, r_2) of similar sub-graphs are explored. If the predecessors (p_1, p_2) have the same hash value, the **slicing operation** is executed and the predecessors are added into list of pair of slices. There are some cases when these pairs are not added in the list:

   (a) Predecessors (p_1, p_2) have different hash values

   (b) Predecessors (p_1, p_2) have the same hash value but $p_1(p_2)$ already in the slice list $r_1(r_2)$. This technique provides an avoiding of endless loop.

(c) Predecessors (p_1, p_2) have the same hash value but $p_1(p_2)$ already in the slice list $p_2(p_1)$. This technique ensures to prevent two slices from sharing the same node.

3. If a clone pair of statements (s_1, s_2) is subsumed to another clone pair (s'_1, s'_2) such that the following intersection takes place $s_1 \subseteq s'_1 \cap s_2 \subseteq s'_2$. This intersection is removed from the set of detected clone pairs. This removal must not affect on application logic since these clone pairs are not useful.

4. The clone set is created from clone pairs as union of clone pairs. For example (s_2, s_4) and (s_2, s_5) generate a set (s_2, s_4, s_5).

A program slicing is the union or association of the set of programs statements, that affects the values at some point of interest, referred to as a slicing criterion. Initially it is used for debugging purposes to trace the whole process of concrete data structure. This procedure is used in the algorithms described above in step 2. The example reveals slicing operation under function `mySlice()`:

```
public void mySlice(){
     boolean detected = false;
     int amount = 0;
     String listOfClones = null;
     while(amount < 10){
        amount ++;
        listOfClones += "clone number: " + amount;
     }
     System.out.println(listOfClones);
     System.out.println(amount);
  }
```

The criterion of slicing is variable `listOfClones` and therefore the new sliced instance is code:

```
public void mySlice(){
     int amount = 0;
     String listOfClones = null;
     while(amount < 10){
        amount ++;
        listOfClones += "clone number: " + amount;
     }
     System.out.println(listOfClones);
  }
```

The algorithm was offered from Komondoor [14] in his research "Semantics-preserving procedure extraction". The noticeable feature of this whole technique is that the clones in code are searched in the one code fragment itself. This technique of clone detection based on program dependency graphs offered by Yoshiki

Higo [13] is mainly directed to find clones in the input code of course depending of programming language. Thus there is no second input code chunk can be compared. The idea is code redundancy and optimization of execution in relative large projects when the pasted code is incorrectly changed or forgotten to be changed. The appreciable privilege of such kind of graphs is obvious. The logical relationship among variables and methods are established in the graph that makes possible to comprehend the information flow control and usability of statements. Unlike the previous ways to find clone like text-to-text 3.2 or AST trees 3.4 are not able to keep statements of elements of statements that are not consecutively located on the source code. However it can be improved with AST tree comparison when same statements are split with regards to their locations in section Text Comparison improvement with AST trees 8.

There are some weaknesses of existing PDG based detection methods. Compared to other detection techniques, the first weakness is that PDG-based detection has lower performance for the detection of contiguous code clones [13]. This is because consecutive program elements in the source code do not have necessarily data dependency or control dependency. On the other hand, line- or token-based detection methods do not consider such dependencies but rather compare program elements textually, so that these methods are productive at detecting contiguous code clones. Secondly the PDG-based detection has a high computational complexity. The number of nodes used as slice point is significantly large and every slice performed operation is a non-trivial task. Since the similar sub-graph needs to be identified it entails NP-complete problem because graph isomorphism is NP-complete problem.

Chapter. 4

Maximum common subtree isomorphism algorithms

This chapter is concerned with the issue of an important generalization of tree isomorphism, mostly known as Maximum Common Sub-tree isomorphism. The goal of these algorithms is finding a largest common sub-tree between two trees. It plays a major role either in scientific fields or in fundamental problems. The trees can be searched for most common sub-tree from top to down, correspondingly form the head of tree till leaves, or from bottom to up, that means the search for largest sub-tree starts from leaves upwards. The algorithms are provided by Gabriel Valiente [3]. In his book *Algorithms on Trees and Graphs* he presented detailed information about Top Down Max Common Sub-tree and Bottom Up Max Common Sub-tree isomorphism. The algorithms have been implemented in dr. Garbage tools plugin for Eclipse integrated development environment.

Further research in this area may include not only the implementation of algorithms but also their application field. On this grounds, that the algorithms can be very helpful for tree similarity investigation. The current chapter includes an explanation of how these two algorithms have implemented in the project, about auxiliary algorithms and how they can be helpful for the structural code comparison and similarity investigation.

4.1 Top-down maximum common sub-tree isomorphism algorithm

There are two types of top-down maximum common sub-tree isomorphism algorithms. One of them finds the largest common ordered sub-tree T between T_1 and T_2 such can found in both trees, by that when the sequence of edges from parent

node does make a sense. On the contrary, the second type is the same top-down algorithm that takes into account the order of edges of parent node, during the algorithm execution.

A top-down common sub-tree of two unordered trees T_1 and T_2 is an unordered tree T such that there are top-down unordered sub-tree isomorphisms of T into T_1 and into T_2. A maximal top down common sub-tree of two unordered trees T_1 and T_2 is a top-down common sub-tree of T_1 and T_2 which is not a proper sub-tree of any other top-down common sub-tree of T_1 and T_2. A top-down of two unordered trees T_1 and T_2 is a top-down common sub-tree of T_1 and T_2 with the largest number of nodes [3].

Definition 3.1. *A **top-down common sub-tree** of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M), where $X_1 = (W_1, S_1)$ is a top down unordered subtree of T_2 and $M \subseteq W_1 \times W_2$ is an ordered tree isomorphism of X_1 to X_2. A top-down common sub-tree (X_1, X_1, M) of T_1 to T_2 is **maximal** if there is no top-down common sub-tree of (X_1', X_2', M') of T_1 to T_2 such that X_1 is a proper top-down common sub-tree of X_1' and X_2' is a proper top-down sub-tree of X_2', and it is **maximum** if there is no top-down common sub-tree (X_1', X_2', M') of T_1 to T_2 with the size$[X_1] <$ size$[X_1'][3]$.*

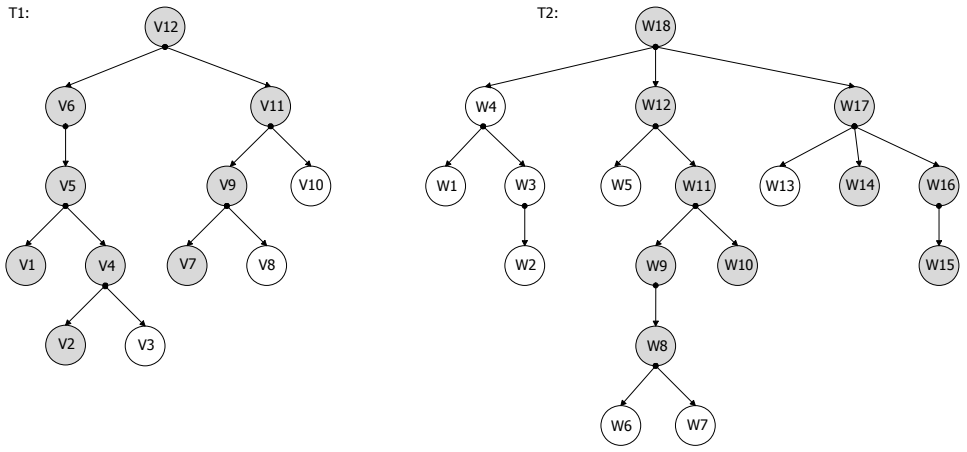
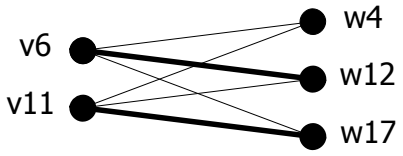


Figure 4.1: Top-down maximum common ordered sub-tree of two unordered trees T1 and T2. Nodes are numbered according to the order in which they are visiting during a post order traversal. The gray highlighted nodes are shaped maximum common sub-tree starting from the root [3].

The figure 4.1 demonstrates a partial injection $M \subseteq W_1 \times W_2$ among trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ where $M = \{(v1, w10), (v2, w8), (v4, w9),$ $(v5, w11), (v6, w12), (v7, w15), (v9, w16), (v10, w14), (v11, w17), (v12, w18)\}$ is unordered top-down maximum common sub-tree isomorphism T_1 and T_2 [3].

Important to realize that the injection $M \subseteq W_1 \times W_2$ can contain different pairs of nodes, as a result there are is only one unique maximum common sub-tree available. Nevertheless the number of pairs of nodes is constant and always



	w4	w12	w17
v6	3	5	3
v11	4	5	4

Figure 4.2: The solution MBM of bipartite graph brings $5+4=9$ weight from previous solutions

maximal.

If nodes v is leaf T_1 and w is leaf of T_2 accordingly mapped to each other, then the maximum common sub-tree gains size 1. Let suppose that p is number of children of T_1 and q is number of children of T_2 respectively. Consequently $v_1, ..., v_p$ and $w_1, ..., w_q$ are children of v and w[3]. Solving the algorithm needs to build a bipartite graph $G = (\{v_1, ..., v_p, w_1, ..., w_q\}, E)$ on $p+q$ vertices, with edge $v_i, w_i \in E$ if and only if the size of maximum common sub-tree of the sub-tree T_1 rooted at node v_i and the sub-tree of T_2 rooted at node w_i is non zero[3]. The bipartite graphs are built recursively the algorithm in one of trees T_1 or T_2 a leaf reaches. Let the leaf node of v_i of tree T_1 is found, then a bipartite graph can be formed with node w_i from T_2 at the same hierarchy level. Using the algorithm of *Maximum Bipartite Weighted Matching* the corresponding parent nodes w_{i+1} and v_{i+1} gains weight one plus *Maximum Bipartite Weighted Matching*(in short, MBM)of current bipartite graph.

	w2
v1	1
v4	1

Figure 4.3: The v_1 and v_4 are leaf nodes of the left branch of T_1, and leaf node w_2 of T_2. The edge (v_1, v_4) is selected with *MBM* algorithm.

As an example can be taken from figure 4.2. As said, the nodes are being traversed until leaf node in the trees T_1 or T_2 is found. Let consider the left branch of T_1, namely leaves v_1 and v_4 and respectively the left branch of T_2, namely leaf w_2. A created bipartite graph is depicted at figure 4.3. The corresponding table shows weights of edges. Since these two are leaves they equal one. With algorithms *MBM* the edge v_1 and w_2 is being selected. Once the selection is done, the algorithm take their parent nodes, accordingly v_5, w_3 and v_3. The appropriate bipartite graph is constructed. The edges v_5 and w_3 have weight equal to one

Figure 4.4:

	w1	w3
v5	1	2

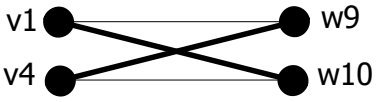
Figure 4.5: The edge (v_5, v_3) gains weight equal to two from previous solution4.3, due to the parent node

initially in table 4.5, but since from previous solution 4.3 the node v_1 has a parent v_5, the edge (v_5, w_3) gets weight equal to two. Likewise the edge (v_6, w_4) in table 4.2 has a value three based on previous result.

By the same way all entries in table 4.2 have been built. Passing through the tree T_1 at another left branch until leaves v_2 and w_3 the similar bipartite graph can be formed 4.6. As said before, the current edges get same weight equal to one. The edge (v_2, w_8) is selected regards the *MBM* algorithm. Under those circumstances the next bipartite graph can be built 4.7. Based on previous solution the edge (v_4, w_9) obtains weight equal to two.

	w8
v2	1
v3	1

Figure 4.6: Starting from leaves select of both trees select edges with maximum weight. According to the algorithm the connection (v_2, w_8) has been selected.



	w9	w10
v1	1	1
v4	2	1

Figure 4.7: Since previous decision was (v_2, w_8), and they respectively are parents of (v_4, w_9) its weight of edge gains one plus the decision equal to one.

	w5	w11
v5	1	4

Figure 4.8: The the sum maximum matched edges from 4.7 equal to 3, in the same manner the edge (v_5, w_{11}) gains $3 + 1 = 4$ weight

Having a look at table in 4.2 in cell (v_6, w_{12}) where the value is 5. This has been formed due to the solution from table 4.8, because nodes (v_6, w_{12}) are direct parents of (v_5, w_{11}). likewise all other leaves in both trees are investigated and on-fly the tables with weight are filled out. The table in 4.2 is completed by the same way as described above. At the end the maximum bipartite weighted matching algorithm selects the edges (v_6, w_{12}) and (v_{11}, w_{17}) because they bring the maximum weight of the bipartite graph. In this case two branches in 4.1 are picked and the corresponding nodes are grey highlighted forming the maximum common sub-tree from the bottom.
Description of the operating principle of algorithm is represented below:

```
list<node, node> topDownMaxCommonSubtreeIsomorphism(const TREE T1, const TREE T2){
node r1 = root(T1);
node r2 = root(T2);
traverse(node r1, node r2);
M = list<node, node>
```

```
reconstruct(r1, M);
}
```

This method goes recursively simultaneously through T1 and T2 till leaves of them are founded.

```
void traverse(node v, node w){
int p = v.getNumberOfChildren();
int q = w.getNumberOfChildren();
if(p == 0 or q == 0) return 1;

Array matrix = new Array[p][q];

for(i = 0; i < p; i++){
   for(j = 0; j < q; j++){
      node vChild = v.getChild();
      node wChild = w.getChild();
      matrix[i][j] = <vChild, wChild, -1>
   }
}

int res = 1;
if(p != 0 and q != 0){
bipartiteGraph bg = createBipartiteGraphFromMatrixEntry(matrix);
if (bg.numberOfEdges == 0) return 0;

list<edge> L = MaxWeightedBipartiteMatching(bg);
forall(e, L){
result += e.getCounter();
}
matching(list<edge> L);
return res;
}
}
```

This method reconstructs the TDMC unordered sub-tree isomorphism mapping.

```
reconstruct(node r1, list<node, node> M){
M[r1] = r2;
list<node> L;
preorder-tree-traversal(L, T1);
forall(node v, L){
   forall(node w, B[v]){
   if(M[T1.parent(v)] == T1.parent(w)){
      M[v] = w;
      break;
   }
}
}
```

Puts into list B matched edges for further reconstruction

```
matching(list<edge> L){
   forall(e, L){
   B[r1].insert(r2);
   }
}
```

Overall, this complex algorithm is very useful tool to find an abstract similarity between tree structures. Application fields can be very various: chemistry, biology, computer vision and pattern recognition.

4.2 Bottom-Up maximum common sub-tree isomorphism algorithm

There are two types of bottom-up maximum common sub-tree isomorphism algorithms. One of them finds the largest common ordered sub-tree T between T_1 and T_2 such can found in both trees, by that when the sequence of edges from parent node does make a sense. On the contrary, the second type is the same top-down algorithm that takes into account the order of edges of parent node, during the algorithm execution.

A bottom-up common sub-tree of two unordered trees T_1 and T_2 is an unordered tree T such that there are top-down unordered sub-tree isomorphisms of T into T_1 and into T_2. A maximal bottom-up common sub-tree of two unordered trees T_1 and T_2 is a bottom-up common sub-tree of T_1 and T_2 which is not a proper sub-tree of any other bottom-up common sub-tree of T_1 and T_2. A bottom-up of two unordered trees T_1 and T_2 is a bottom-up common sub-tree of T_1 and T_2 with the largest number of nodes [3].

Definition 3.1. *A **bottom-up common sub-tree** of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M), where $X_1 = (W_1, S_1)$ is a bottom-up unordered subtree of T_2 and $M \subseteq W_1 \times W_2$ is an ordered tree isomorphism of X_1 to X_2. A bottom-up common sub-tree (X_1, X_1, M) of T_1 to T_2 is **maximal** if there is no bottom-up common sub-tree of (X_1', X_2', M') of T_1 to T_2 such that X_1 is a proper bottom-up common sub-tree of X_1' and X_2' is a proper bottom-up sub-tree of X_2', and it is **maximum** if there is no bottom-up common sub-tree (X_1', X_2', M') of T_1 to T_2 with the $size[X_1] < size[X_1']$[3].*

The figure 4.9 demonstrates a partial injection $M \subseteq W_1 \times W_2$ among trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ where $M = \{(v1, w10), (v2, w6), (v3, w7), (v4, w8), (v5, w9), (v6, w11), (v7, w5), (v8, w12)\}$ is unordered bottom-up maximum common sub-tree isomorphism T_1 and T_2 [3].

The problem of finding a maximum common sub-tree isomorphism between two

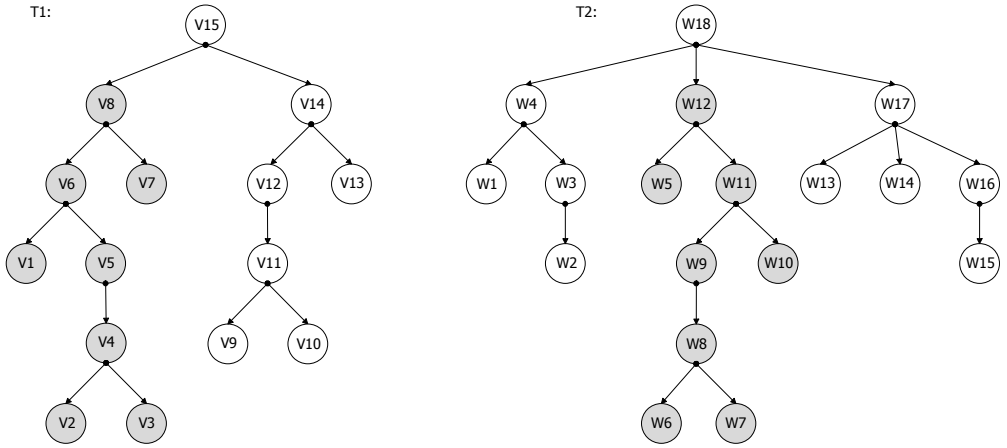


Figure 4.9: Bottom Up maximum common ordered sub-tree of two unordered trees T1 and T2. Nodes are numbered according to the order in which they are visiting during a post order traversal. The gray highlighted nodes are shaped maximum common sub-tree starting from the leaves [3].

trees T_1 and T_2, where T_1 has n_1 nodes and T_2 has n_2 respectively can be reduced to the problem of partitioning $V_1 \bigcup V_2$ into equivalent classes of bottom-up sub-tree isomorphism. Two nodes are equivalent if and only if the bottom-up unordered sub-tree rooted at them are isomorphic.

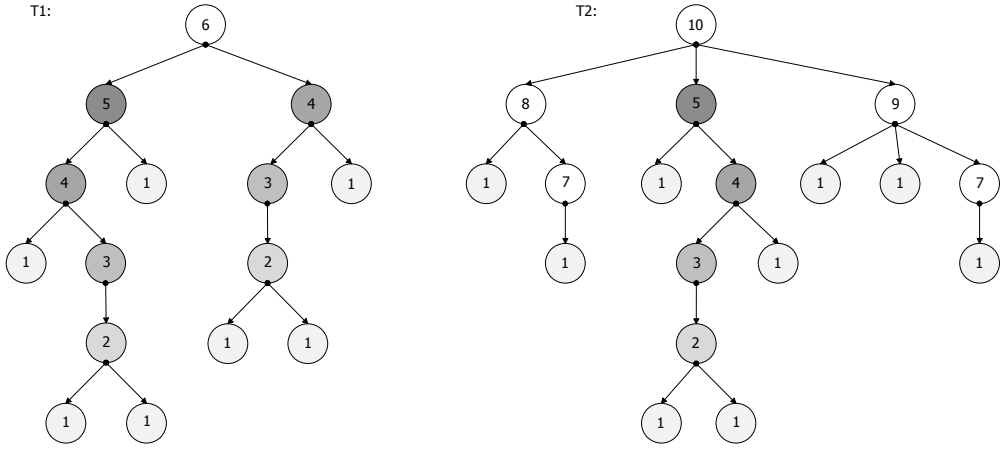


Figure 4.10: Bottom Up maximum common equivalence classes reflected to the figure 4.9. The node are numbered according to the equivalence class to which they belong and the equivalence classes are shown highlighted in the different shades of gray[3].

The algorithm start with a partition a tree into bottom-up equivalence classes. Let the number of known equivalence classes be initially equal to 1, corresponding to the equivalence class of all leaves in the trees. For all nodes v of T_1 and T_2 in post-order, set the equivalence class of V to 1 if node v is a leaf. Otherwise, look up in the dictionary the ordered list of equivalent classes to which the children of node v belong. If the ordered list (key) is found in the dictionary then set the equivalence class off node v to the value (element) found. Otherwise, increment by

one the number of known equivalence classes, insert the ordered list together with the number of known equivalence classes in the dictionary, and set the equivalence class of node v to the number of known equivalence classes.

Chapter. 5

Experimental analysis between structural and textual methods of code comparison

5.1 Introduction in experiments

This chapter is dedicated to comprehend the methods of textual and structural compare approach. As a result of these experiments an improved concept of structural approach compare is created. For the better understanding of probable improvements of code comparison using existing algorithms, some possible ideas of implementation and an amount of experiments is required. In order to build a proper tool, or at least a concept, in Eclipse plugins at Dr. Garbage ® some hand experiments in code should be fulfilled.
There are many combination how code can be smartly changed, then existing methods of compare sometimes are not able to identify code plagiarism. All these test cases are performed in Eclipse IDE [16] and divided into sections where mostly text component of is changed which sometimes does not influence on the application logic. It assist to figure out why textual comparison is insufficient of careful code investigation. There are two the most well-known methods how to build a graph from source code These analysis steps are fulfilled in this chapter:

1. Research on Flowchart graphs derived from java source code, using existing methods for maximum common tree isomorphism from chapter 4 and embedded Eclipse compare editor:

2. Research on abstract syntax tree derived from java source code, using existing methods for maximum common tree isomorphism from chapter 4 and embedded Eclipse compare editor:

3. Research on java byte code comparison using graph theory.

All test sets are investigated under java methods and functions, in other words two functions with similar application logic. In some cases the demonstrative result of comparison is not important, but the statistical data are substantial. The analysis phase is carried out under small snippets of code, thereby the real difference, application logic and code similarity, is observed in details by the experimenter.
Playing around with the patterns of code changing variables, names, sequences of commands, adding loops or conditions and apply on it methods of textual comparison. This comparison is already implemented in Eclipse IDE [16], with so-called command "compare with each other by member" that calls text compare editor. This type of comparison provides a pop-up window, where two pieces of code are compared, particularly line by line.
In parallel, a control flowchart graphs or AST trees from the functions are being created and compared using implemented algorithms top-down maximum common and bottom-up maximum common(in short, TDMC and BUMC). The further task is to figure out the difference/similarity from graphical visual comparison. Consequently these both results must be matched and recorded for succeeding research. Take into account that textual compare is guided to investigate precise difference in code, whereas TDMC and BUMC are concerned to find out tree isomorphism, in particular similarity. Statements of code are nodes in compared trees depends on tree organization. When two fragments of code are compared, text editor highlights an amount statements and amount of isomorphic nodes are highlighted in trees. This can be computed into real figures and quantitative compared to see which can identify the distinction. Statements can be a name of variable, sequence of commands or assignment of variable. It completely depends on tree organization. For example, `if(people.count() > = 10)` for the flowchart it represents one a decision node, rhombus formed and consequence branches depend on condition itself. The same chunk of code in AST represents a branch with root node the while expression and sub branches, for instance `people` and `10`.
Let the set S_1 is all statements marked in text compare editor and the set S_2 is amount of all mismatched nodes after the algorithm execution. The derived results from experiments can be as listed:

1. Text compare and TDMC/BUMC find the same differences $S_1 = S_2$. Thus, textual found statement are not equal to their according non-isomorphic nodes.

2. Text compare and TDMC/BUMC deliver similar difference $S_1 \cap S_2 \neq \emptyset$. It means in some cases the algorithms are able to detect same mismatches and sometimes not.

3. Text compare and TDMC/BUMC provide full difference $S_1 \cap S_2 = \emptyset$, conse-

quently the intersection of all marked statements is an empty set.

5.2 Experiments on Java source code flowcharts

A flowchart is a type of diagram that represents an algorithm, work-flow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various field s[4].

Dr. Garbage tools[1] provides a solution how to represent sequential flowchart alongside to Java source code (see figure 5.1).

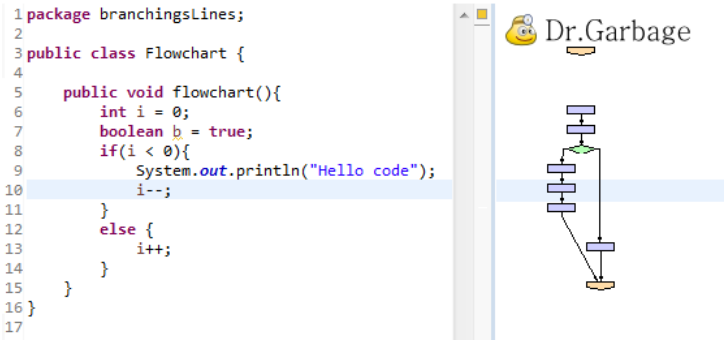


Figure 5.1: Example of source code visualizer

A depicted flowchart can be easily extracted into control flow graph. For another similar function, it can be transformed into next control flow graph. Afterwards these two graphs are being compared using existing TDMC[4.1] and BUMC[4.2] algorithms.

Unfortunately these two algorithms are applicable only for tree data structures. For this reason this problem can be reduced, removing minimum number of edges to get a spanning tree. Thus the the edges in the input graphs are reduced by Spanning Tree Algorithm[6.1]. The removed edges are red highlighted, hence this for this structure TDMC and BUMC[4] can be easily applied.

To conduct an experiments a sequence of actions and following statistic are needed. After conducted experiments, taking into account the derived statistic, and outgoing conclusion takes place. The steps are carried through sequence of action:

- Write two similar java functions in Eclipse IDE
- Apply for them text-to-text comparison with Eclipse compare editor
- Declare the statistic, respectively how many statements are different
- Create a source-code graph for both function

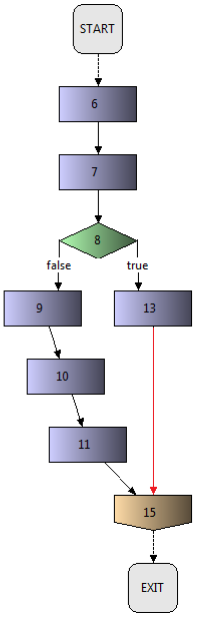


Figure 5.2: Extracted control flow graph from Java source code

- Apply TDMC and BUMC algorithms to get structural difference
- Declare the statistic, respectively how many nodes as the result are different

In this section all experiments have been executed by hand using plug-in tool **Graph Comparison** in Eclipse from dr. Garbage project. Before starting, a several rules how to evaluate code difference in text and in graph must be established. It is one of the crucial moment, because followed data statistic are used in further tool development. For the estimation of structural difference there are criteria listed:

1. Each java statement is considered as a simple node
2. Changing a conditions of block on the contrary issues 100% difference of application logic, however the structure stays unchangeable.
3. Percentage of difference in textual compare is computed, thereafter it roughly equals to division of number of marked to unmarked statements. The same procedure of average similarity proportion is calculated after tree comparison. Since there are two algorithms looking for isomorphic nodes, the statistic is taken from the most suitable one, which specially brings the largest number of matched nodes.

Mostly all calculations are performed roughly, because there are many criterias how to evaluate the logic and structure difference. Nonetheless, from this perspective these rules are enough to examine graph's similarity and make a conclusions.

For the text difference found via Eclipse tool a criteria to evaluate can be added:

- Percentage is computed by number of
- If in one line of code only one symbol has been covered as found, then it is division of one to amount of symbols in this line.

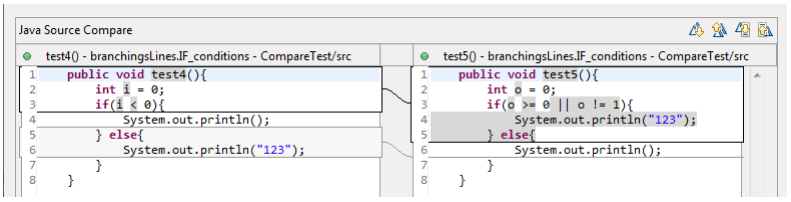


Figure 5.3: Two pieces of code are being compared with Eclipse Text Comparison

And after generation of two graphs, these both are compared(see the figure 5.4) using existing TDMC and BUMC[4] algorithms.

Compare experiments			Text -to-text compared	Graph compared
Test id	Name of functions	Real code difference %	Layout difference % found	TDMC&BUMC similarity % found
1	t1() and t2()	50	100	100
2	t1() and t3()	50	100	75
3	t1() and t4()	50	100	100
4	t1() and t2()	50	100	80
5	t1() and t5()	50	100	100
6	t6() and t7()	33	33	100
7	ti1() and ti2()	10	100	0
8	ti2() and ti3()	16	100	100
9	ti3() and ti4()	25	100	66
10	ti4() and ti5()	50	100	0
11	ti6() and ti7()	90	100	42
Overall results:			93.9 %	69%

Table 5.1: The table demonstrates results of Java Source comparison using text-to-text compare method and application of algorithms to their source code graphs

The table 5.1 shows results of java source code experiments. Existing algorithms [4] and Eclipse text-to-text comparison have been used to reveal the best approach of code comparison. As it said above, the difference code can be figured out either structurally or simple text comparison. Based on results in the table 5.1 the apparent conclusion are composed:

- If the application logic is totally different (For example the condition **if(people.count() > 10)**) then Eclipse compare editor finds the difference, thus condition itself is highlighted. The graphical comparison with TDMC&BUMC are not able to see this distinction. It is obvious, since graph theory in this case is able to find how similar structure of code fragments. The conducted example 5.4 above can testify this conclusion. In this way, graph theory is not applicable to differentiate logic of application.

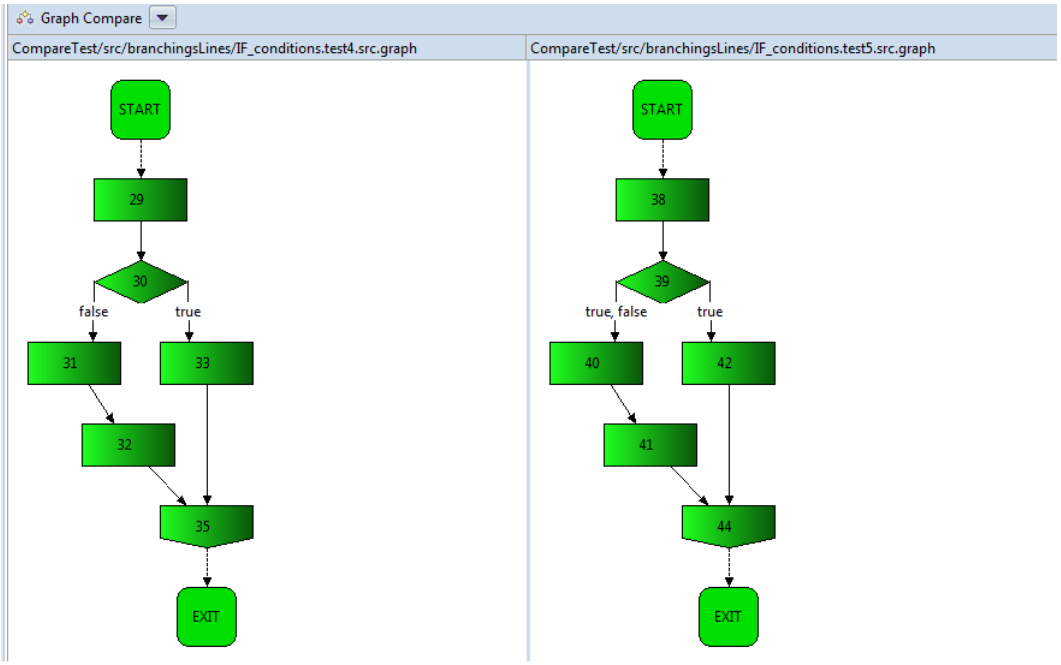


Figure 5.4: Compared source code graphs using TDMC algorithm

- In opposite said above, the graphical compare is quite useful tool to investigate a code structure. For example, if another third party person has changed local variables, the structure remains same, thus TDMC& BUMC (especially TDMC) find high level of similarity. This approach is enough to build a new concept allowing to investigate the code similarity more precisely.
- The graph is totally bound to lines of codes. If one brace is shifted, then it is considered one more block in the graph(Dr. Garbage Source Code Visualizer [1] generates extra node for each code operator). Hence, TDMC is not able to find following branch where there an extra node and generated Java source code graph is not optimized for comparison.
- Text-to-text compare is enough to investigate a text difference because this tool finds every different sub-string in code line. But as stated above, changing variables, sequence of operators or even production same loops with different operators the text-to-text compare find too much unmatched strings. Eventually the result of text compare looks like a disorder with same and unmatched sub-strings.

5.3 Experiments using Abstract Syntax Tree graphs

In this section the abstract syntax trees are generated from java source code are used to be compared. In contrast to flowcharts, the AST trees do not contain cycles, therefore it is not necessary to remove any back edges to get a tree struc-

tures. The most notable advantage of AST trees creation is inflexible structure and proper ordered depths nesting of vertices according to the code. For example, loop in code has of sub-statements, the same way the nesting of branch is organized.

Test id	Case of mismatch	Text difference	Structural difference	Result
1	redundant statement added	found	found	both techniques are able to notice the difference visually
2	decision statement changed	found	not found but structure is same delivered	text compare highlights changed condition
3	statements have been swapped	found but is incorrect presented	found	swapped statements can be easily identified
4	decision statement changed; parameters changed	found	not found but structure is same delivered	structure of code is absolute identical
5	names of variables changed	found	not found	structure of code is absolute identical, i.e. 100% code similarity

Table 5.2: Results showing the difference of comparison between combination of AST trees and existing algorithms and traditional textual compare

The table 5.3 demonstrates intermediary results between structural and textual comparison. Primarily can be said that the sub-tree algorithms are very suitable to define similarity if code fragments have a complementary structure. Thus, the text compare is beneficial tool to identify precise difference in:

1. Small code fragments, for the reason that large compare results bring quite complicated view.
2. Code fragments with very similar code construction, where there are insignificant changes, for instance conditions in decision statement, or another style of increment writing.

After analysis part on AST trees and applied algorithms, the next assertions can be expressed:

1. The algorithms are supportive to identify the percentage of plagiarism.
2. Applicable for large pieces of code, if statistical details required. Specifically, if it is necessary to know how alike compared objects.

Based on these outcomes, some improvements are described in this work. Particularly, the technique that allows to built more similar AST trees whereas the different style of programming performing same application logic. The chapter **Techniques to normalize AST improving structural comparison** 7 reports

the actual solution. In addition, the chapter **Text comparison improvement with AST trees** 8 discloses another approach of code text representation.

5.4 Experiments on JavaByte Code

In this section an investigation regards java byte-code comparison is expound. Unlike Java source code, the corresponding byte code has practically no application logic. In spite of this the topic must be researched for the clone detection.
Citation from the IBM **developerWorks** journal, "Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembly helps the C or C++ programmer" [17].

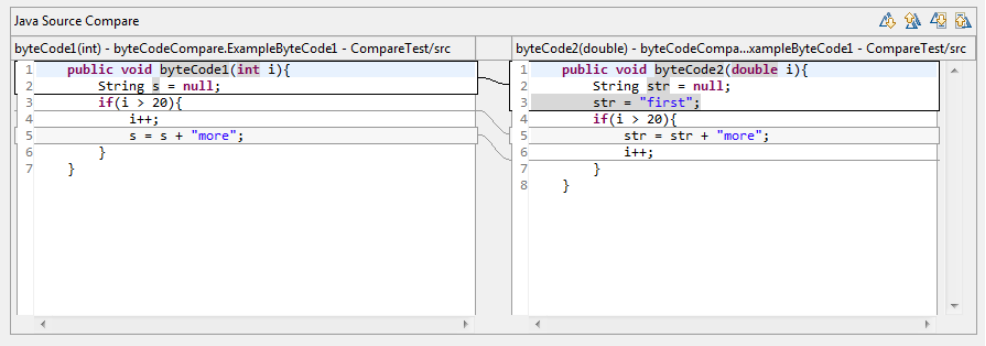


Figure 5.5: Java byte code compared using compare editor in Eclipse environment

Assignment of variable in java byte code includes java byte code address and according type suffix/prefix. The complexity of java byte code compare with graph theory is that the code has no structured logic like java source code. However, there are some clues hot to find some code clones between two java compiled files. In this case, the comparison must be performed within one complied .class file, not a methods as it described above in java source code.

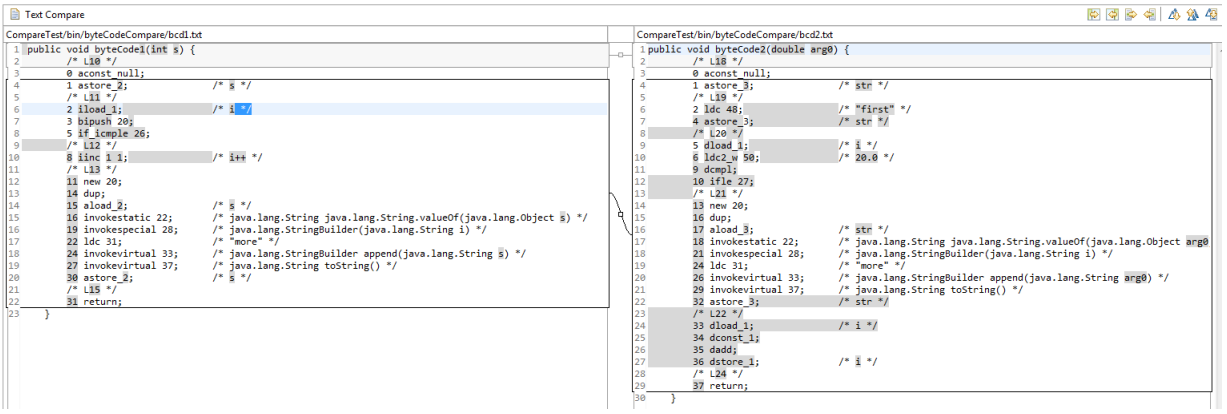


Figure 5.6: Java source code compared with editor in Eclipse environment

To detect clones or find similarity between two complied java function using graph creation is not suitable technique for this. It may be an idea to create a bipartite graph, where partition A is all nodes with corresponding instructions and partition B is formed by the same way. The relations among the node are a Cartesian product. Then, an algorithm that searches for same instructions that exist in both partitions. This task, can be organized with collection and theory of sets.

```
/* L14 */
0 bipush 15;
2 istore_1;             /* people */
/* L15 */
3 getstatic 19;         /* java.lang.System.out */
6 ldc 25;               /* "jf" */
8 invokevirtual 27;     /* void println(java.lang.String s) */
/* L16 */
11 iload_1;             /* people */
12 invokestatic 33;     /* java.lang.String java.lang.Integer.toString(int s) */
15 astore_2;            /* s */
/* L17 */
16 return;
```

Assignment of variables can be found with association of opcodes, for example: `aload_0` or `istore_2`. This is relation to the type of the data that the instruction operates. The prefix a means that the opcode is manipulating an object reference and the prefix i means the opcode is manipulating an integer. The list L_1 represents the first collection of instructions from code a, and L_2 has instruction from code b. Same the instructions can be found in the parsing of both lists. If element E_i contains `istore_1`, then the same list L_1 is searched for any instructions operate with #1 index. This reminds a slicing operation described in chapter **Existing comparison methods for plagiarism detection** PDG 3.5. All these instruction with index 1 are encompassed into set S_i. The maximum intersection set S_i' of instructions is searched in the list L_2 that have same index, that bijection function f exists that $f : S_i \to S_i'$. In other words, a subset with maximum number of instruction in L_1 is searched, that similar to S_i.

Chapter. 6

Graph transformation algorithms

6.1 Introduction to the graph transformation

TODO: explain how graph is generated, which libraries are used, plugins

Chapter. 7

Techniques to normalize AST improving structural comparison

Researches show that many big development projects have duplicated code, which is generally result of copying and pasting existing pieces of code. Moreover the code can be completely redone, changing name of variables, in some cases replace lines of code. Nevertheless, the most sophisticated part comes when a same command can be in the code substituted. It is known, that from programming perspectives, for instance, a loop can be differently created, however application logic stays unchanged. There three fundamental ways to reproduce the iteration statements, namely: *pre - condition*, *post - conditions* and so-called *for-loop*. The function can be reworked by so delicate way, that none of any text-to-text compare finds similarity. Mostly, only controversy of both fragments in compare editor will be displayed. However, the functions bring the same application logic. This code clones can be found almost everywhere, especially in free-published software, the most notable example is clone pair between FreeBSD and Linux, that code listing 7.1 presents.

In the following example 7.1 these functions are similar and have same application logic. If these functions are compared with text compare, then definitely one line with increment of variable `frameGroupLine` is highlighted.

```
1  public void fragment1(){
2        int frameGroupLine = 10;
3        for(int Cnt = 1; Cnt < frameGroupLine; Cnt =+ 2)
4        {
5           if(Cnt*4 != 2){
6              frameGroupLine++;
7           }
8        }
9     }
10
11 public void fragment2(){
12       int frameGroupLine = 10;
13       for(int Counter = 1; Counter < frameGroupLine; Counter =+ 2)
14       {
15          if(Counter*4 != 2){
16             frameGroupLine = frameGroupLine + 1;
17             }
18       }
19    }
```

Listing 7.1: Clone pair between FreeBSD and Linux

In the chapter **Existing comparison methods for plagiarism detection**, namely in token-based technique there are some processes to prepare input string for optimized graph creation. Therefore, the Sub-tree suffix algorithm 3.3 is able to bring better results. For example, Karp-Rabin fingerprints algorithm reviewed in [6] is used for calculating the fingerprints of all length n sub-strings of a text. First, a text-to-text transformation is performed on the considered source for discarding the uninterested characters.

Afterwards, the entire text is subdivided to a set of sub-strings so that every character of the text appears in at least one substring [6]. After that the matching sub-strings are identified.

In that stage, a further transformation is applied on the raw matches to obtain better results. Instead of applying a set of text transformations, he applies several different transformation scenarios from a combination of basic transformations such as for identifying near-miss duplication he attempted to find a normalized/-transformed text by removing all white space characters except line separators and by replacing each maximal sequence of alphanumeric characters with a single letter `i`. For example, a line `for(k = 1; k <= n; k + +) {` is replaced by the line `i(i = i; i < =; i++)`. This kind of transformation gives too more false positives. But, this idea can be used in structural similarity comparison. Since the TDMC algorithm 4.1 looks the largest sub-tree, it means the largest sub-tree is kind of common part of two derived AST trees from code fragments.
Consider the line number 16 from the listing 7.1. If this line `frameGroupLine =`

`frameGroupLine + 1;` will be converted into one format of sub-tree, that indicates the same as `frameGroupLine++;`. As the result, this another way created sub-tree can be found with TDMC or BUMC algorithms. Consequently it brings more covered nodes that signalize more similarity, exactly that improves the statistic of covered nodes.

```
1 public void normalizedFragment(){
2    int frameGroupLine = 10;
3    for(int Counter = 1; Counter < frameGroupLine; Counter =+ 2)
4    {
5       if(Counter*4 != 2){
6          frameTeamLine++;
7       }
8    }
9 }
```

Listing 7.2: Normalized function `test3()` concerning variable `frameGroupLine`

In the function `fragment2()` one the increment of variable `frameTeamLine` is differently written. From text-to-text compare the functions are different at this point. If the text code similarity will be calculated as mismatch in this place, then these two fragments are not same (probably 90% similarity). Using Abstract Syntax Trees Optimization, this types of structure can be converted in the same sub-tree of whole AST tree. Thereby, these two different text structures are represented as same sub-tree.

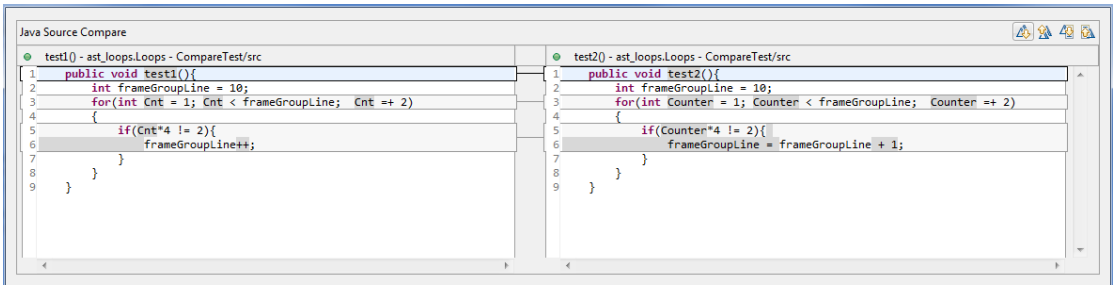


Figure 7.1: Example of standard text-to-text comparison of Java code

On figure 7.1 the example demonstrates how these two functions are compared using Eclipse Text comparison window. After creation and comparison these two AST trees, it can be easily seen that sub-trees (the increment) are different.

From the figure 7.2 TDMC algorithm finds incomplete code similarity since not all nodes have been covered. From statistical point of view using simple math, the percentage of similarity is figured out: 37 nodes in the `test2()` and 3 of them are not covered. Thus, the calculation indicates $\left(1 - \left(\frac{3}{37}\right)\right) \cdot 100 = 91\%$ code similarity according to AST trees and applied TDMC algorithm.

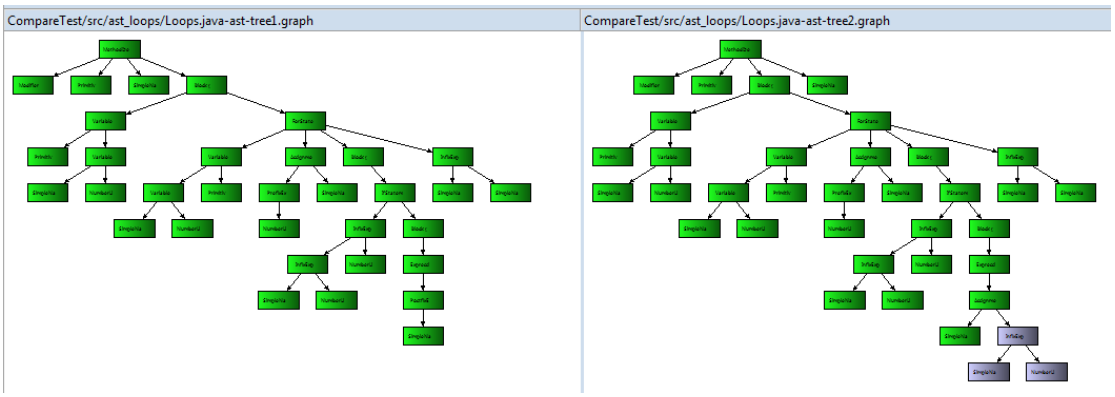


Figure 7.2: Graph comparison of function fragment1() and fragment1() using TDMC algorithm 4.1

This mismatch can be optimized during AST tree production. These two lines of code `frameGroupLine++;` and `frameGroupLine = frameGroupLine + 1;` must be built as a same sub-tree, accordingly same structure and same number of nodes. Using this simple replacement it allows to built a similar AST trees, when logic is same but the source code text is different. Consequently the converted AST trees to some extent are independent from source code and can be compared to explicit the difference.

The example described above is relatively small, but exact showing how this logic can be circumvented. Also in pre-processing of AST tree can be included all possible java operators. This process called code normalization, and can be prepared before the AST tree will have been built.

Operator	input	output
postfix	`i++`	`i = i + 1`
	`i--`	`i = i - 1`
prefix	`++i`	`i = i + 1`
	`--i`	`i = i - 1`
assignments	`i *= n`	`i = i * n`
	`i /= n`	`i = i / n`
	`i %= n`	`i = i % n`
	`i -= n`	`i = i - n`
	`i += n`	`i = i + n`
	`i =+ n`	`i = i + n`
	`i =- n`	`i = i - n`

Table 7.1: The table demonstrates a proper string transformation for AST tree optimization. In event of occurrence of input operator, the original can be substituted before AST tree have been built.

The table 7.1 presents possible java string transformation depending on the input command. Thus the fragment of code will be better prepared for AST tree reformation. The trees of both both comparing functions will more similar that TDMC algorithm finds more similarity. Using statistical data to consider how

function similar to each other the formula can be derived. Let NC is subset all of unmatched nodes then
$N \subseteq T_1 : n \in NC$ and $M \subseteq T_2 : m \in NC$
The trees are combined as set of vertexes and edges, thereby $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ respectfully, the the probability that two functions have same structure takes place:

$$similarity(\%) = \left(1 - \left(\frac{max(|N|, |M|)}{max(|V_1|, |V_2|)}\right)\right) \cdot 100$$

where $|V_1|$ and $|V_2|$ are cardinalities of their subsets. In other words the formula can be described as ratio of maximum number of unmatched nodes between T_1 and T_2 to maximum number of nodes in both trees T_1 and T_2.

Overall, such combination of tree normalization and applied maximum subtree algorithms has an initial stage of the structural comparison. The beneficial point is an additional parser, that restructures AST tree according to some rules. It allows to get more precise statistical results. The further development in this field can be dedicated to an extension of existing regulations. Particularity, loop implementation or restructure of tree among `switch` and `if` statements, that implemented as same control flow with different commands.

Chapter. 8

Text comparison improvement with AST trees

In the chapter about existing comparison methods, section section text based techniques 3.2 is written about disadvantages of text-to-text compare, namely changing the sequence of statements which do not influence on application logic. However in this case, comparison of text-to-text is sensitive and thereby its representation of difference is redundant.

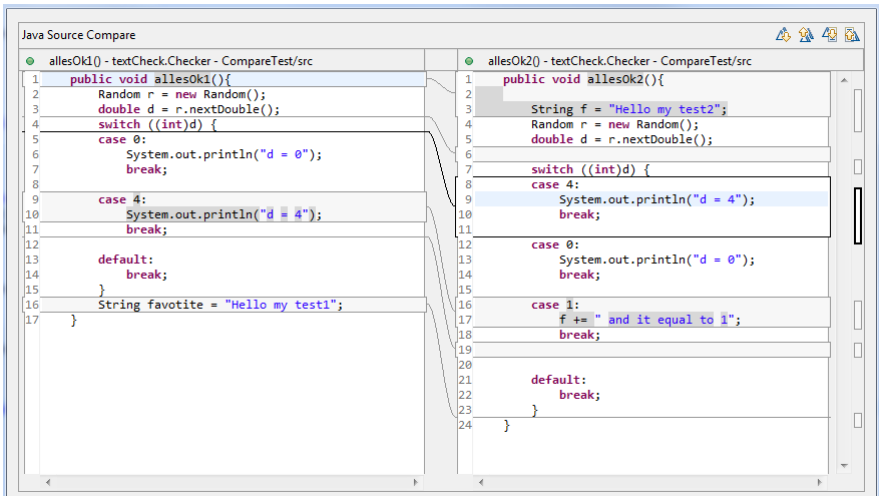


Figure 8.1: Example of standard text-to-text comparison of Java code

This example 8.1 demonstrates how text compare is not available to notice the string difference in case some statements have been simply replaced. Namely the operators: `case 0:` and `case 4:` from figure 8.1 have been replaced but text compare find this difference.

Moreover, when this code mostly changed, namely the names of variables, line position in code, thereby not impacting on the application logic of function then it is much more difficult to clarify the code difference. Thus, comparing two

function using text-to-text compare, it can be that the final result is totally mixed, however these functions are almost identical. Comparing of relatively large code fragments which are consecutively swapped according to time line makes text-to-text comparison not so much effective, especially on the back code compare representation. The inefficiency consists in complexity of imposed comparison, in some cases it is not readable.

Why text-to-text cannot find these obvious differences? This algorithm for string search is described in chapter 3.2. Using the Suffix sub-tree algorithm, according to research of Roy, Chanchal Kumar [6], there are some disadvantages that can interpret the issue described above on picture 8.1 where some sub-strings are not found described in following citation:

> This tools does not support exploration and navigation through the duplicated code. Detection accuracy is low e.g., cannot detect code clones written in different coding styles. For example "{" position of if-statement or while-statement. Cannot detect code clones using different variable names, e.g. to identify the same logic code as code clones even if variable names are different[6].

Hypothesized that the Suffix sub-tree algorithm is more negatively related to proper code compare in case if clone detection is being searched. This can be improved with help of tree based algorithms techniques;

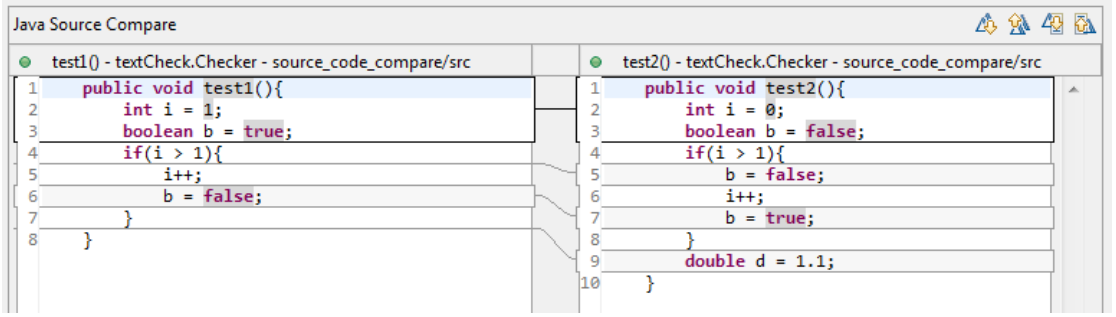


Figure 8.2: Example of text-to-text demonstrating the code difference using the Suffix sub-tree algorithm

As prerequisites take the following example depicted on the picture 8.2. Considering the example very carefully, line by line, it is easy to get an idea how these two peaces of code are organized. In the line #2, the obvious mismatch variable `i = 1` to compared with `i = 0` and `false` to `true` are gray highlighted in the third line.

Up next, comes the `if` operator, where there are two and three blocks respectfully. They make almost the same logic however the sequence of statements is changed. The last difference on the line #11: the function `test2()` contains extra command

`b = true;`. At the end of function `test2()` there is an additional operator. Notice that, the example is relatively small and has not some much different commands and operators and moreover, when sequence of statements is shifted, swapped or changed. It can be that two functions are so complex modificated, thereby the text compare yields a mix of various lines.
To improve this, a code can be transformed into Abstract Syntax Tree(in short, AST) and both tree are traversed synchronously checking matched nodes. The biggest advantage of methods based on trees that AST trees are built independently of sequence of commands. The difference in this trees is unordered range of nodes. It means that these AST trees are taken into consideration as unordered trees, where ordering is not specified for the children of each vertex. Swapped statements can be limited by one block, in other words the higher rank command represents a parent-node and within this block there swapped sub-command that are descendant-nodes. The sequence of descendant-nodes is arbitrary, since AST trees are also built as unordered.

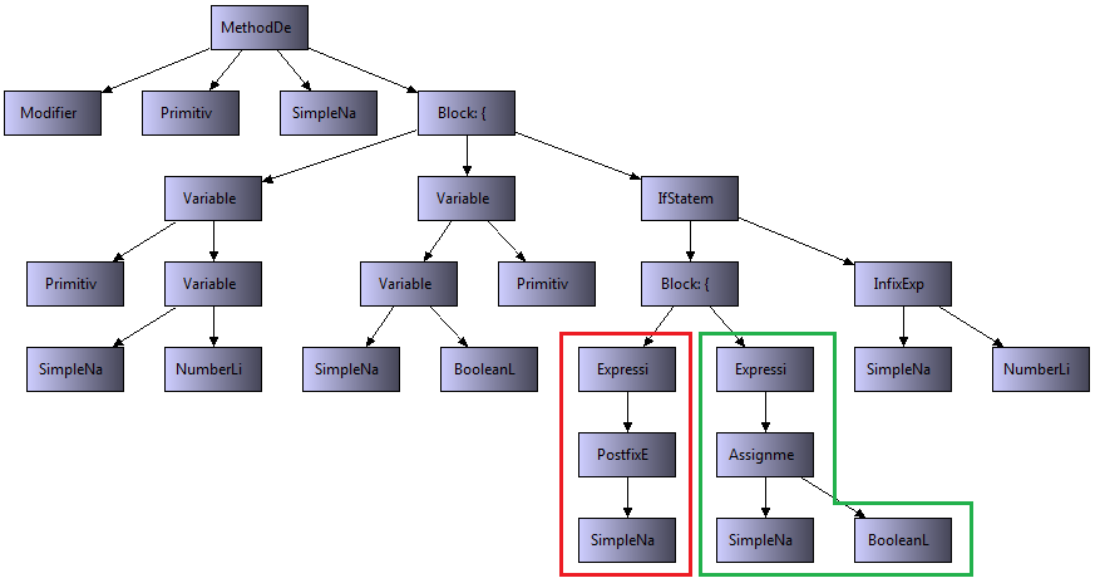


Figure 8.3: Abstract Syntax Graph of function `test1()`.

Example 8.2 and derived AST trees 8.3 and 8.4 indicates similarity, difference and redundancy:

1. Red and green framed branches are equal: `i++` and `b = false` , therefore the nodes in this branch must be marked as *matched*, in other words same without difference.

2. The yellow frame marks out the statement `b = true;` line, that is redundant in `test1()` in this branch, specifically in `if(i > 1){` condition. Hence, only

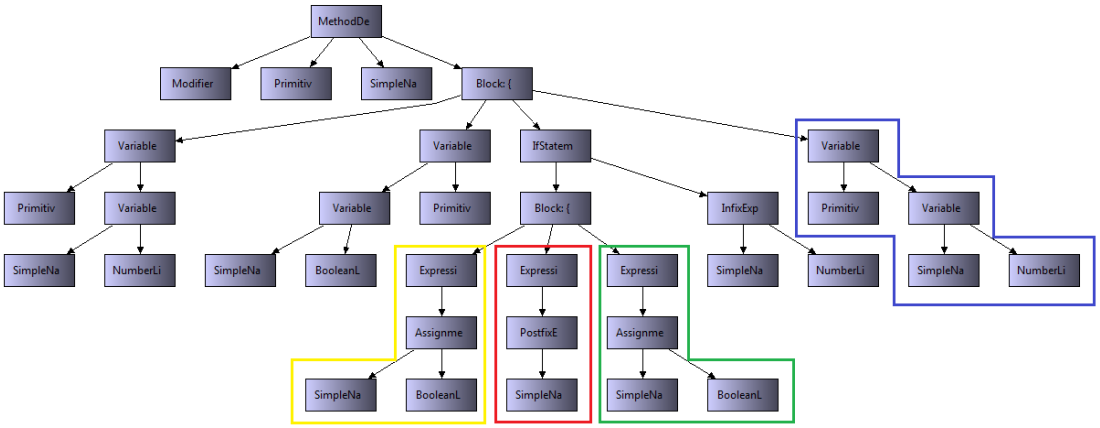


Figure 8.4: Abstract Syntax Graph of function `test2()`.

this statement must be highlighted in text compare windows as *mismatched*.

3. The blue frame in picture 8.4 denotes a new command line, namely `double d = 1.1;`. This statement is also redundant and comes from another nesting level and must be marked as possible mismatch. In Cartesian product this parent node is not matched, hence the whole branch starting from the parent is marked as *mismatched*.

The picture 8.3 shows how AST tree from function `test1()` is being built from the source code. As said above, the presented AST tree 8.3 is unordered, thus the replaced lines of code and the nested operators and independent from each other. The trees traversal enables to identify the concrete difference in source code, independent how the code is organized. The evident mismatch must be referred to the source code. The idea is to keep the current command or statement in the AST tree node, i.e. to keep inside a meta-information: a current text position of command or statement in the corresponding node.

These coloured framed branches are visible and signalize a discrepancy between two AST trees. Apparently that, the same discrepancy makes sense to be proper delivered into text editor, specifically into text-to-text comparator. For this reason a modified traversal algorithm is required. The Breadth-first search traverse algorithm is performed on two AST trees derived from *test*1() and *test*2() methods. At same node depth i-level a Cartesian Product from one parent-node among nodes is built (a bipartite graph, almost identical strategy has been used in existed algorithm TDMC 4.1)to search for different, missed or redundant nodes and mark them accordingly for further text representation. However, in this bipartite graph the content is compared. For the algorithm optimization technique all descendant-nodes, respectfully branch and its content is being hashed. This procedure is done during AST tree creation.

Taking all sketches into consideration the following algorithms steps can be derived:

1. AST tree creation from the source code. This AST tree is unlike to original because of content of nodes. This node meta-information is extended and recorded during tree forming. Each node keeps the following information:

   1.1. Content - exact statement, operator or command in text format. For example: `if (i != 1)`

   1.2. Hashed value sub-branch - during the tree creation each sub-tree content is hashed. For example, the root node of statement:

   ```
   if (i != 1){
    i = i + 1;
   }
   ```

   has a for-example MD5 hash-value of `i = i + 1;`
   that equals to `05b200b7f0b9b4bdf3f4a1b1d8aa041c`. This allows to decrease the complexity of algorithm to search sub-strings. If large sub-functions are same, comparing their hash-values gains a lot of performance. In the best case the complexity can be $O(1)$, instead of searching out all sub-nodes.

   1.3. Global text positions - precisely the number of line of the node element and start and end point coordinates within this line. It needs for the proper text difference representation on text-compare window.

   1.4. A variable *matched* that keeps information about whether this statement exists in both functions and must not be highlighted.

   1.5. Any other meta-information for statistical data to be extended.

2. Once AST tree is built, the Bread-first search (in short, BFS) is applied on the trees. The traverse is executed simultaneously comparing i-nested level of each tree. Firstly the content of nodes is compared, once the content is matched textually, up next the hashed-values are compared. Since hashed value are unique identifiers there are two scenarios:

   2.1. Hash-values are same - in this case the whole branch is marked as *matched*. It says that on this nested level there are same statement in both compared statements.

   2.2. Hash-values are different - the BFS is being continued further and step 2. is repeated to investigate the leaves where difference is placed since the hash-values are not matched.

3. Once BFS is finished the procedure to deliver result back to text is started. This is second traversal that runs over all unmatched nodes and distributes difference on the compared text. Nodes not marked as *matched* are represent the code redundancy. In most cases, the highlighted elements are explicit redundant with regards of executed comparison, in other words the compare fragments have not exact same highlighted statements.

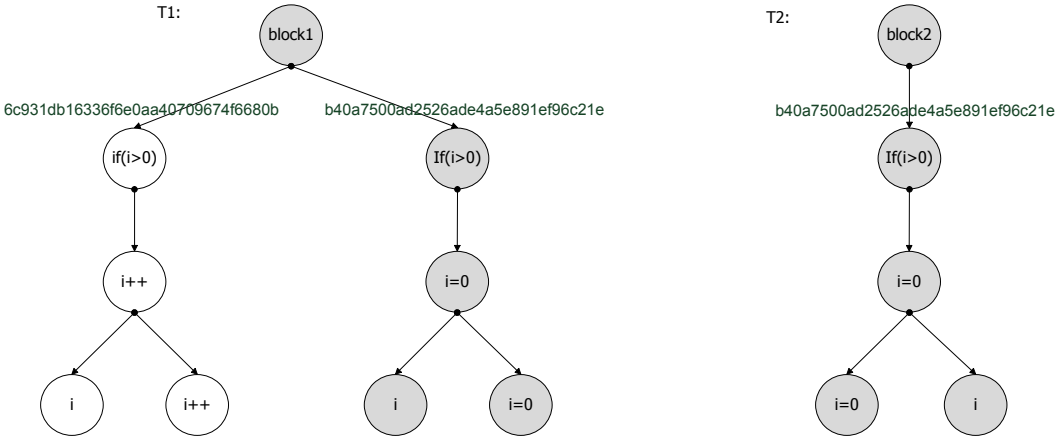


Figure 8.5: Example of hashed branch content assists to optimize the algorithm search for clones. The more similar compared fragments of the faster the algorithms execution.

The figure 8.5 shows two AST tree compared during tree-traversal. Firstly, the content of 2-nested level is compared and both have `if(i > 0)` that indicated that nodes are same. In the described second step of the algorithm the hash-values containing the whole content of sub-tree are compared. The left branch of T_1 is unique to the main branch T_2 and consequently these both are marked *matched*. Obviously a question appears, why firstly not to compare by the hash-value since it is unique, avoiding compare of the content. It does not make a sense because the parent-node has content with sub-elements and the difference is takes place there. The algorithm seeks out for a concrete mismatch, and there is no sense to stop the BFS in hash-values are different whereas content is same. To be precised, the difference is placed in the descendant-nodes that must further investigated.

The complexity of algorithms in the worst case can be $O(3 * |V|)$, where V number of created AST nodes, and three traversal are required: creation of the tree, search for clones and representation in text compare window. The hashing technique simplifies the complexity roughly 20% since large computer systems have duplicated code.

Disadvantages of this approach are restriction of the search space. The AST tree represents a non-flexible data structure which does not depend on real logic of application as it described about Program Dependency Graphs 3.5. The sequence of statements is limited by one parent node located on i-nested level. It can be

possible when there is the same block of code statements where the duplicated fragment is placed $i + 1$-nested, i.e. under another code operator. Nevertheless, the application logic of code fragment is not different. Further research must be dedicated to above described problem. The idea can be taken from PDG-methods 3.5. To trace the variables or to slice under required condition are basics to detect clones located in different branches.

Statistical data interpretation is a crucial moment after fragments have been compared. The statistics can interpret in the first row how similar were the code fragments.

$$similarity(\%) = \left(1 - \left(\frac{max(|T_1'|, |T_2'|)}{max(|T_1|, |T_2|)}\right)\right) \cdot 100$$

Where: $T_1' \subseteq$\{all mismatched nodes in T_1 \} and
$T_2' \subseteq$\{all mismatched nodes in T_2 \}
The statistical data play a big role in code similarity investigation. After execution of comparison a collection of derived figures must be provided:

- Statistical review of changed command structures. Particularly, this is information shows proportion of `if`-conditions have been changed for instance. In additional, a command statements structure redundancy, for example quantity of extra `for` loop in both compared code chunks.
- Code redundancy - in other words how much code redundancy in both code fragments.

Overall, the idea how find similar or same disordered pieces of code with AST tree is quite argumentative. At least, using AST is improvement of standard the Suffix-tree based algorithm, that compares line by line not taking into consideration the application logic. On one hand, text-to-tex comparison is is useful to detect swapped statements, despite the fact that the logic of application is not disturbed. However, if in the similar code has another small changes, like names of variables or various index increments statements, then text-compare provides rather sophisticated and hard read results. Oh the other hand, if this sensitive is code difference is really necessary, it is reasonable to use standard text comparison.

Chapter. 9

Conclusion

The discussions regards plagiarism detection and successive avoiding and maintenance of it are not a innovation today. However nowadays, in software maintenance this theme is object of current interest. As said in the introduction, the real code development process can not go without reusable code, that is not pragmatical in usage. It implies that misuse of copying-pasting of existing code fragments is common situation in the development process. In length of time, the code handling becomes progressively harder, because of software clones. Another problem is comparison code fractions for similarity. There is a necessarily when fragments of code need to be compared to figure out precisely and visually where this difference come from. Sometimes, when code fragments are too large to seek out each difference a statistical data representation is sufficient proof whether codes are similar or different with probability.

The process analysis phases in chapter Experimental analysis between structural and textual methods of code comparison reports about ways of comparison using tree-based approach and in addition these techniques are compared with textual method. In this process, the existing two algorithms, namely Top-down and Bottom-up maximum common sub-tree isomorphism have been involved to define a a structural similarity. It turns out, that the larger common sub-tree, in most cases sub-tree found from top to down, more similar their code structures. From this experience has been derived that AST trees are more suitable to compare code for similarity and also partly to investigate exact difference.

Furthermore the existing well-know techniques of clone detection and compare have been mentioned. From experimental part it is become known that text-based approach is insufficient to detect precise difference in code. This is justified by the fact that code has its own logic, and changing text textual components

of it, the logic stays same. Nonetheless, textual does not provide sufficient facts that code implements same logic, as result that is code plagiarism. Clear code text is non-flexible structure for text-based algorithms, therefore the text-to-text compare methods have an independent gap between logic and text.

This work describes an interpretation of algorithms which finds the largest sub-trees between input two trees taken from Gabriel Valiente [3]. The algorithms have been implemented and adjusted to the project dr. Garbage - is a set of open source plug-ins for control flow analysis of java programs. These algorithms are useful not only in software area but in chemical and biological branches, because essences learnt there can be defined as tree structures.

The chapters Techniques to normalize AST and Text comparison improvements announce an ideas to enhance quality of code search with actual method of clone detection. The first improvement report about tree optimization during it form from code, by that makes compared structured uniformed. The second is about technique to improve textual compare, including of application logic, with tree-based method.

Each detection technique has its own relative advantages and disadvantages, and no technique is superior to any of the other techniques in all aspects.

Bibliography

[1] The Dr. Garbage Tools Project® 2014, Sergej Alekseev, Peter Palaga and Sebastian Reschke, URL: `http://www.drgarbage.com`

[2] Sergej Alekseev. *Graph theoretical algorithms for control flow graph comparison*, 2013.

[3] Gabriel Valiente,*Algorithms on Trees and Graphs*, Berlin: Springer-Verlag, 2002.

[4] Free content Internet encyclopedia - Wikipedia: Flowcharts, URL: `https://en.wikipedia.org/wiki/Flowchart`

[5] Free content Internet encyclopedia - Wikipedia: Plagiarism detection, URL: `http://en.wikipedia.org/wiki/Plagiarism_detection`

[6] Roy, Chanchal Kumar; Cordy, James R. (September 26, 2007). *"A Survey on Software Clone Detection Research"*. School of Computing, Queen's University, Canada.

[7] Koebler Johannes; Schoening, Uwe. (July 29, 1991). *"GRAPH ISOMORPHISM IS LOW FOR PP"*. Theoretische Informatik, Universitaet Ulm

[8] Brenda S. Baker *"A Program for Identifying Duplicated Code"*. AT&T Bell Laboratories, Murray Hill, New Jersey

[9] Beat Fluri, Student Member, IEEE, Michael Wuersch, Student Member, IEEE, Martin Pinzger, Member, IEEE, and Harald C. Gall, Member, IEEE *"Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction"*. Department of Informatics, University of Zurich, Zurich

[10] Toshihiro Kamiya, Member, IEEE, Shinji Kusumoto, Member, IEEE, and Katsuro Inoue, Member, IEEE *"CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code"*. July 2002

[11] Flavius-Mihai Lazar *"Clone detection algorithm based on the Abstract Syntax Tree approach"*. Politehnical University of Timisoara, Timisoara, Romania, 9th IEEE International Symposium on Applied Computational Intelligence and Informatics May 15-17, 2014 Timişoara, Romania

[12] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, Lorraine Bier *"Clone Detection Using Abstract Syntax Trees"*. Semantic Designs 12636 Research Blvd. Suite C214, Austin Texas

[13] Yoshiki Higo, Shinji Kusumoto *"Code Clone Detection on Specialized PDGs with Heuristics"*.2011 15th European Conference on Software Maintenance and Reengineering - Graduate School of Information Science and Technology, Osaka University, Suita, Osaka, Japan

[14] R. Komondoor and S. Horwitz *"Effective, Automatic Procedure Extraction"*.in Proc. of the 27th ACM SIGPLANSIGACT on Principles of Programming Languages, Jan. 2000, pp. 155–169.

[15] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou *"CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code"*.IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 32, NO. March 2006

[16] Eclipse documentation, URL: `http://help.eclipse.org/luna/index.jsp` last visit:

[17] Understanding bytecode makes you a better programmer, URL: `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/` last visit:

[18] Sample Author, NAME, (Berlin: Springer-Verlag, 2002).