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

Submitted By

**Artem Garishin**

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

**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

TODO:

This should be a 1-page (maximum) summary of your work. What environment for development has been used, experimental results An abstract is a summary in your own words of the Thesis It is not evaluative and must not include your personal opinions. The purpose of an abstract is to give a reader sufficient information for him or her to decide whether it would be worthwhile reading the entire article or book. An abstract should aim at giving as much information as possible in as few words as possible.

Goal of this work is to search out the most optimal ways to compare different pieces of code. So far there are two techniques for code comparison: a normal text comparison and visual compare. Normal text-compare can be not sufficient to analyse two pieces of code, or to find a similarity between them. For that reason a structural/graph compare opens a doors to discover more possibilities of comparison.

# 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 reusage and adjusting of existing code fragment. But practically, it is observed that mostly the code duplication and further appliance is not proper used. Large software projects are developed by many people for years and at some point there are problems of applications since code was not proper supported. These strategies like the code duplication and then following reuse 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 to personal needs and by performing small correction. The code is adapted to the new program 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 duplicated code. 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 code redundancy, particularly unknown pieces of reusable code are located in different places that makes code uncomprehending for further development.
Another not less important problems nowadays is plagiarism detection. Most occasion 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 duplica-

tion and their methods 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 refactoring of the duplicated code is theme for discussion in software maintenance. Although practically restructuring code without changing its external behavior of certain clones are not desirable and there are a risks when these duplication 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 analyze codes changing, to explore 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 operation systems Linux and FreeBSD have been implemented. The table presented from research [15] shows the figures of code clones these projects.

| **Project** | **Number of segments** | **Copy-pasted %** |
|---|---|---|
| Linux | 198.55 | 23.0% |
| FreeBSD | 153.23 | 20.4% |
| Apache | 6.19 | 17.7% |
| PostgreSQL | 16.6 | 22.2% |

Table 1.1: The table shows

As shown in this table, in Linux and FreeBSD, there are more than 100,000 and 120,000 copy-pasted segments without any which account for about 15 percent of the source code.

# Chapter. 2

## Description of problem

In this chapter an issue of the master work is being explained. As usual a compare of two code fragments consider comparison of classes, functions or methods. Thereby a compare can be counted as examinations of two pieces of code, in the best case a methods or functions. They can have a similar implementation or alike syntax, however these two pieces of code are different.

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 a code can be taken from external source and a variables have been changed. In addition to general search can be improved to look out a similar code in big projects.
REDO: Duplicated code occurs frequently in real programs,as indicated by the results of several studies
Structural compare stands for comparing two graphs. There are many possibilities how to create a graph from java source. 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. For example the maximum common sub-tree isomorphism algorithms. The purpose of them to seek out the largest common sub-tree between two trees. These algorithms are used not only to investigate code difference but also they are a fundamental problem with multiplicity of applications in nature sciences and engineering. But unfortunately it is possible only for trees but not for graphs. If two graphs are being isomorphic compared with each other, then it is an NP - complete problem and takes much times and efforts to be done.

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, and some action must be taken into account to reduce the domain of comparison before performing the actual comparison. Therefore this problem must to be reduced. Luckily, a tree comparison is able to executed in polynomial time and moreover there are some existing algorithms to compare trees.

Thus there are no deterministic algorithms to compare them because of loops in graphs. In this case the input code can be transformed into graph firstly, after the graph creation, it must be converted into tree, using simple techniques removing back edges. The back edges in the input graph are edges, which point from a node to one of its ancestors. Under those circumstances the following techniques are searched and deployed in the paper. After all a few questions can be asked:

1. How to transform code into tree by optimal way?
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?

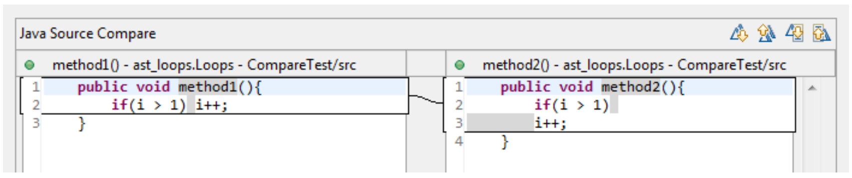
Regards to the first question, the concept of idea is described in chapter **Graphs Transformation** 5. The second question comprehends existing algorithm and their combination and improvements in chapter **Existing algorithms** 3. The very last issue is about how to lead back the result of the code and is stated in chapter.

Possible result of this thesis is development of concept to find out code difference using graph theory, in the best case a tool in Project Dr. Garbage [1]. Can be implemented that highlights similarity/difference of input code snippet and represented respective graph.

To get started with a small example demonstrating, what kind of result gives text compare (text-to-text compare):

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

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

In this example, two pieces of code there one enter symbol after line (i ¿ 1). The result therefore the codes look different. Using simple text compare approach, only these gap will be found, however this difference does not play any role regards business-logic. In Abstract Syntax View, these two graphs will be same, and no discrepancy will have been discovered.


Java Source Compare

method1() - ast_loops.Loops - CompareTest/src

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

method2() - ast_loops.Loops - CompareTest/src

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

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

Since this article includes comparison not only of source code, but also Java byte code, where there is no syntax. Basically to have a look at byte code example, there are no bounds to hold a functions or methods. Based on this, control flow graph can be derived from byte code, that represents a graph, but not a spanning tree. Every node has a reference to byte code address.

To investigate code comparison, two algorithms of structural compare are required, in fact top-down maximum common sub-tree and bottom-up maximum common sub-tree algorithms. To make a contribution into development of structured code compare, the following tasks should be explored:

1. The existing algorithms must be investigated (The text-compare method is not sufficient to find a similarity in code)

2. The algorithms for the structured compare(Abstract syntax trees, Control flow graphs) must be explored

3. New methods and algorithms find a place to tried out. A prototypes of combination text-compare and structure-compare can be implemented.

4. Experimental results of compare must be derived.

# Chapter. 3

# Tree 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 for plugin 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 code comparison.

## 3.1 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].*

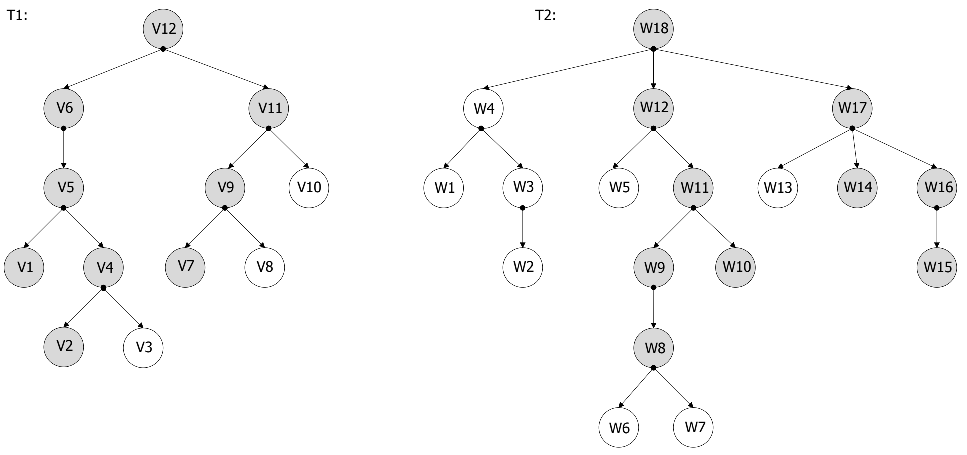
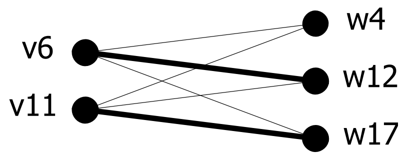


Figure 3.1: Top-down maximum common ordered sub-tree of two unordered trees T1 and T2. Nodes are numbered according to the order in which they are visiting during a post order traversal. The gray highlighted nodes are shaped maximum common sub-tree starting from the root [3].

The figure 3.1 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 maximal.

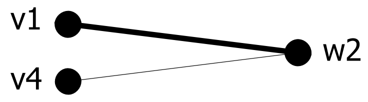
If nodes $v$ is leaf $T_1$ and $w$ is leaf of $T_2$ accordingly mapped to each other,



| | w4 | w12 | w17 |
|---|---|---|---|
| v6 | 3 | 5 | 3 |
| v11 | 4 | 5 | 4 |

Figure 3.2: The solution MBM of bipartite graph brings $5+4=9$ weight from previous solutions

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 3.3: 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 3.2. 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 3.3. 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 initially in table 3.5, but since from previous solution 3.3 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

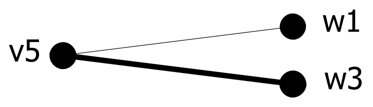


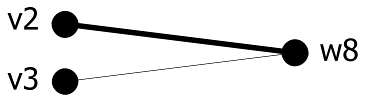
Figure 3.4:

| | w1 | w3 |
|---|---|---|
| v5 | 1 | 2 |

Figure 3.5: The edge $(v_5, v_3)$ gains weight equal to two from previous solution3.3, due to the parent node

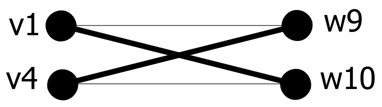
3.2 has a value three based on previous result.

By the same way all entries in table 3.2 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 3.6. 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 3.7. Based on previous solution the edge $(v_4, w_9)$ obtains weight equal to two.



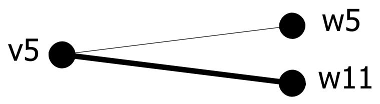
|  | w8 |
|---|---|
| v2 | 1 |
| v3 | 1 |

Figure 3.6: 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 3.7: 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 3.8: The the sum maximum matched edges from 3.7 equal to 3, in the same manner the edge $(v_5, w_{11})$ gains 3 + 1 = 4 weight

Having a look at table in 3.2 in cell $(v_6, w_{12})$ where the value is 5. This has been formed due to the solution from table 3.8, 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 3.2 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 3.1 are picked and the corresponding nodes are grey highlighted forming the maximum common sub-tree from the bottom. Pseudo code 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 top-down max common 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);
}
}
```

## 3.2 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 3.9 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 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.

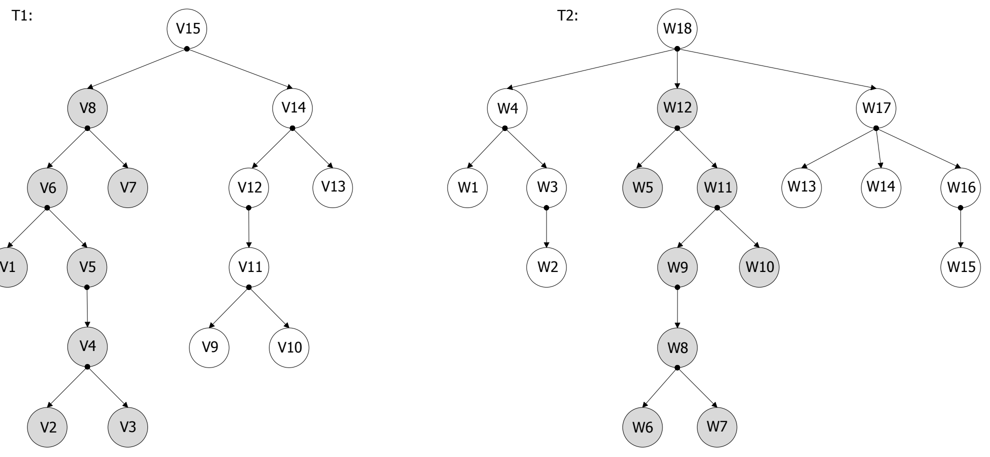


Figure 3.9: Bottom Up maximum common ordered sub-tree of two unordered trees T1 and T2. Nodes are numbered according to the order in which they are visiting during a post order traversal. The gray highlighted nodes are shaped maximum common sub-tree starting from the leaves [3].

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.

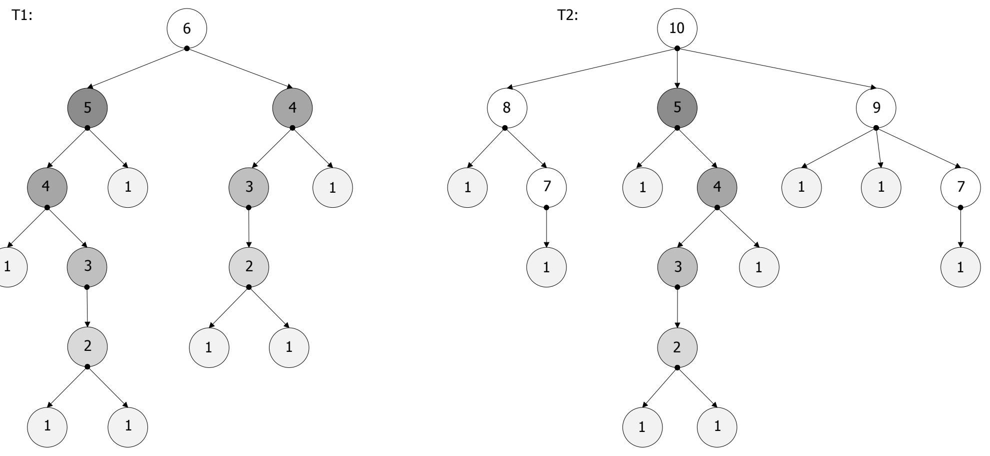


Figure 3.10: Bottom Up maximum common equivalence classes reflected to the figure 3.9. The node are numbered according to the equivalence class to which they belong and the equivalence classes are shown highlighted in the different shades of gray[3].

# Chapter. 4

# Code compare experiments

## 4.1 Introduction to experiments

For better understanding of possible ”code compare” concept, possible ideas of implementation and following development an amount of experiments are required. In order to build a proper tool or at least a concept, in Eclipse plugins at Dr. Garbage Community® some hand experiments in code should be fulfilled.

All these test cases are performed in Eclipse IDE [16] and divided into blocks. These steps can be approached by following:

1. Research on Java source code using existing methods to compare:
    (a) Normal text compare
    (b) Spanning trees transformed from control flow graphs
2. Research on Java source code using existing methods to compare:
    (a) Normal text compare
    (b) Abstract syntax trees
3. Research on Java byte code using existing methods to compare:
    (a) Normal text compare
    (b) Control flow graphs

All test set are investigated under Java methods and functions. Playing around with the patterns of code changing variables, names, sequences of commands, adding loops or conditions and apply the simple ”text to text” compare. This comparison is already implemented in Eclipse IDE [16], so-called command "compare with each other by member". This type of comparison provides a pop-up window, where two pieces of code are compared, line by line.

In parallel a control flow graphs or source graphs from the functions are being created and compared using implemented algorithms Top-Down and Bottom-Up(following called: TD& BU). 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.

The derived results from can be as follows:

1. Text compare and TD & BU have same difference
2. Text compare and TD & BU give similar difference
3. Text compare and TD & BU five full difference

Hence, as it was declared in section description of problem, based on these results can be decided what kind of tool in Dr. Garbage Eclipse plug-ins can be built.In case similar or full difference results, a combination of both methods can be used for optimal comparison.

Small example can be demonstrated: there are two functions that look very similar but nevertheless they have different number of string and different functionality. The abstract results can be following:

1. Text compare shows that strings 1 and 5 are different
2. Graph compare shows that string 7 is different

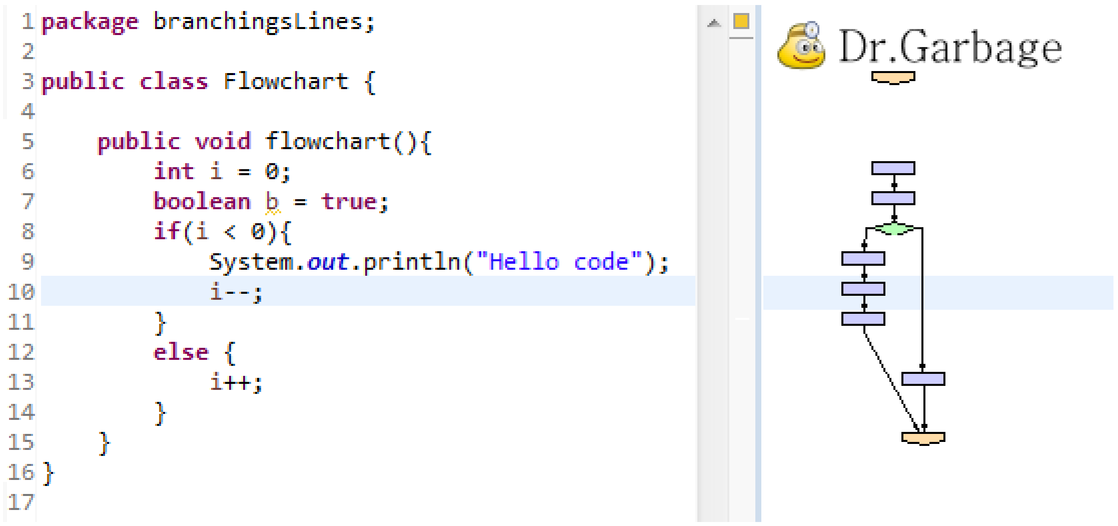
Conclusion: a combination of two methods can explicit that strings 1,5 and 7 are distinguished and much more distinction has been found. Thus it provides an optimal way of investigation.

## 4.2 Experiments on Java source code Flowcharts

A flowchart is a type of diagram that represents an algorithm, workflow 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 fields[4].

Dr. Garbage tools[1] provides a solution how to represent sequential flowchart alongside to Java source code(see figure 4.1).

A depicted flowchart can be easily extracted into control flow graph (see figure 4.2). If there is another similar function, it can be transformed into next control



```java
package branchingsLines;

public class Flowchart {

    public void flowchart(){
        int i = 0;
        boolean b = true;
        if(i < 0){
            System.out.println("Hello code");
            i--;
        }
        else {
            i++;
        }
    }
}
```

Figure 4.1: Example of source code visualizer

flow graph. These two graphs are being compared using existing TDMC[3.1] and BUMC[3.2] algorithms.

Unfortunately these two algorithms are applicable only for tree 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[5.1]. The removed edges are red highlighted, hence this for this structure TDMC and BUMC[3] 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, a conclusion takes place. The steps are carried through sequence of action:

- Write two similar functions in Eclipse IDE
- Apply for them text-to-text comparison
- Declare the statistic, respectively how many lines are different
- Create a source-code graph for both
- Apply TD & BU algorithms to get structural difference
- Declare the statistic, respectively how many nodes 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:

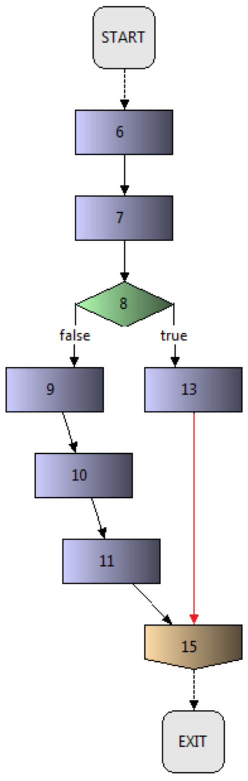


Figure 4.2: Extracted control flow graph from Java source code

- Each java operator is considered as simple node
- Changing a conditions of block on the contrary issues 100% difference of business logic, however the structure stays unchangeable.
- Availability of extra variables in second piece of code is calculated by division of number of extra variables to amount of all variables.

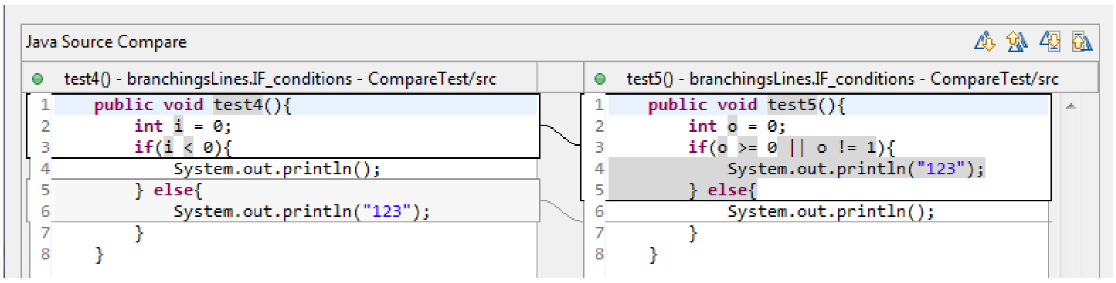
Mostly all calculations are performed by roughly, because there are many criteria how to evaluate the logic and structure difference. But from this perspective these rules are enough to examine graph's similarity.

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.

And after generation of two graphs, these both are compared(see the figure 4.4) using existing TDMC and BUMC[3] algorithms.

The table 4.1 shows results of java source code experiments. Existing algorithms [3] 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



Java Source Compare

test4() - branchingsLines.IF_conditions - CompareTest/src

```java
    public void test4(){
        int i = 0;
        if(i < 0){
            System.out.println();
        } else{
            System.out.println("123");
        }
    }
```

test5() - branchingsLines.IF_conditions - CompareTest/src

```java
    public void test5(){
        int o = 0;
        if(o >= 0 || o != 1){
            System.out.println("123");
        } else{
            System.out.println();
        }
    }
```

Figure 4.3: Two pieces of code are being compared with Eclipse Text Comparison

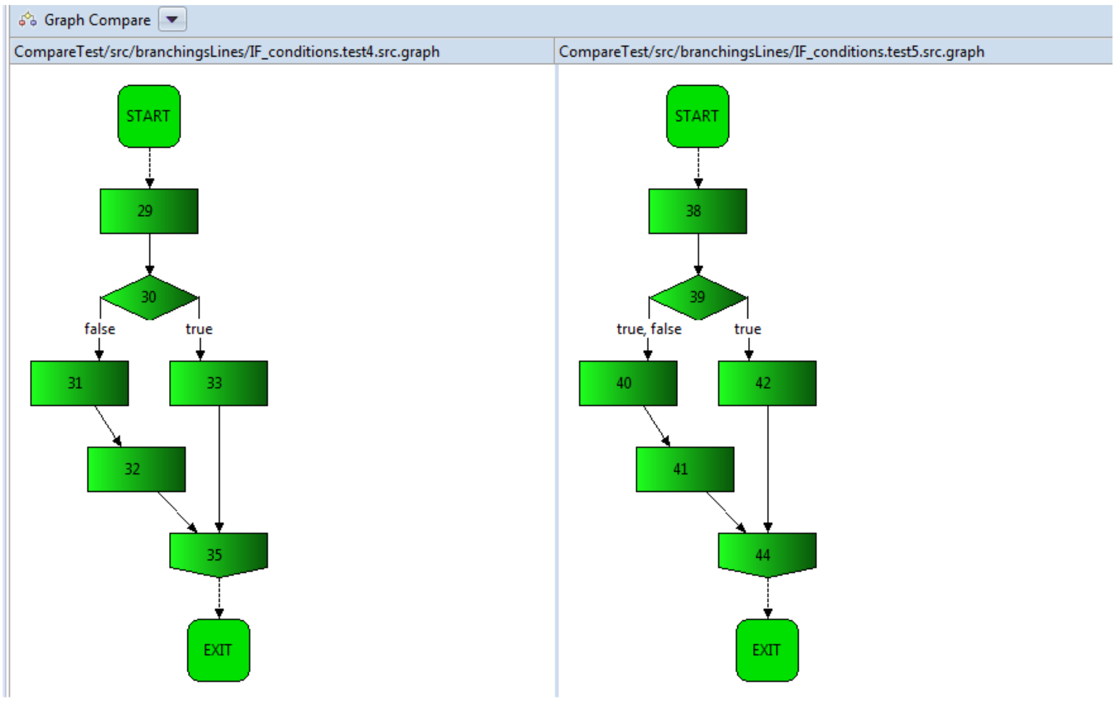

Graph Compare
CompareTest/src/branchingsLines/IF_conditions.test4.src.graph
CompareTest/src/branchingsLines/IF_conditions.test5.src.graph
START
29
30
false
true
31
33
32
35
EXIT
START
38
39
true, false
true
40
42
41
44
EXIT

Figure 4.4: Compared source code graphs using TDMC algorithm

| Compare experiments | | | Text -to-text compared | Graph compared |
|---|---|---|---|---|
| Test id | Name of functions | Real code difference % | Layout difference % found | TD&BU 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 4.1: The table demonstrates results of Java Source comparison using text-to-text compare method and application of algorithms to their source code graphs

either structurally or simple text comparison. Based on the table 4.1 the apparent conclusion are composed:

- If the application logic is totally different (For example **if conditions**) then Eclipse Text Compare finds the difference, thus condition itself is highlighted. The graphical comparison with TD& BU 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 4.4 above can testify this conclusion. In this way graph theory is not applicable to differentiate logic of application.

- In opposite said above, the graphical compare is quite useful instrument 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. Unfortunately this approach is not enough to build a new concept allowing to investigate the difference more precisely.

- The graph is totally bound to lines of codes. If one brace is shifted, then it's 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.

## 4.3 Experiments using Abstract Syntax Tree graphs

In this section the abstract syntax trees are generated from Java source code using Dr. Garbage plugins [1]. The most notable advantage of building AST trees is a direct converting Java source code into AST tree, thereby avoiding graphs with cycles. Thus there is no need to delete back edges(see Spanning tree algorithms).

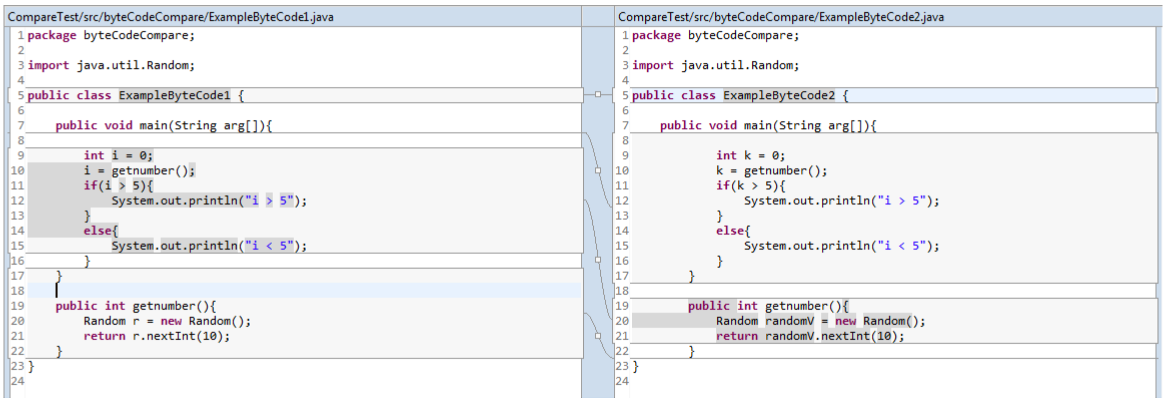
TODO: short explanation about AST how it looks like and we need them;

In this section all experiments have been executed by hand using plug-in tool "Graph Comparison" in Eclipse from dr. Garbage project [1]. Before starting, a several rules how to evaluate code difference in text and in graph must be established. It one of the crucial moment, because followed data statistic are used in further tool development.

TODO: after this compares write a conclusion what is better to compare

## 4.4 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.



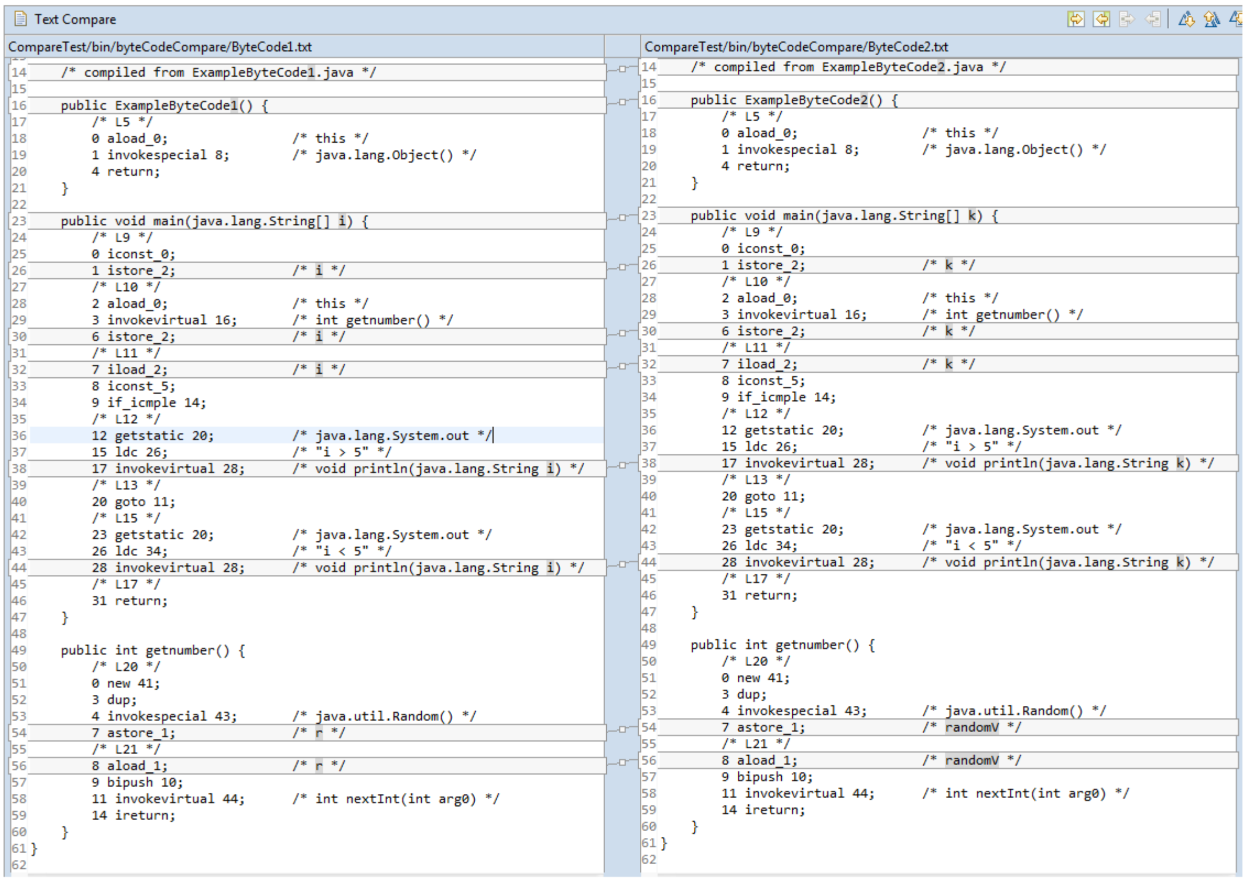
CompareTest/src/byteCodeCompare/ExampleByteCode1.java

```java
package byteCodeCompare;

import java.util.Random;

public class ExampleByteCode1 {

    public void main(String arg[]){

        int i = 0;
        i = getnumber();
        if(i > 5){
            System.out.println("i > 5");
        }
        else{
            System.out.println("i < 5");
        }
    }

    public int getnumber(){
        Random r = new Random();
        return r.nextInt(10);
    }
}
```

CompareTest/src/byteCodeCompare/ExampleByteCode2.java

```java
package byteCodeCompare;

import java.util.Random;

public class ExampleByteCode2 {

    public void main(String arg[]){

        int k = 0;
        k = getnumber();
        if(k > 5){
            System.out.println("i > 5");
        }
        else{
            System.out.println("i < 5");
        }
    }

    public int getnumber(){
        Random randomV = new Random();
        return randomV.nextInt(10);
    }
}
```

Figure 4.5: Java source code compared using text-to-text



CompareTest/bin/byteCodeCompare/ByteCode1.txt

```
    /* compiled from ExampleByteCode1.java */

    public ExampleByteCode1() {
        /* L5 */
        0 aload_0;                /* this */
        1 invokespecial 8;        /* java.lang.Object() */
        4 return;
    }

    public void main(java.lang.String[] i) {
        /* L9 */
        0 iconst_0;
        1 istore_2;               /* i */
        /* L10 */
        2 aload_0;                /* this */
        3 invokevirtual 16;       /* int getnumber() */
        6 istore_2;               /* i */
        /* L11 */
        7 iload_2;                /* i */
        8 iconst_5;
        9 if_icmple 14;
        /* L12 */
        12 getstatic 20;          /* java.lang.System.out */
        15 ldc 26;                /* "i > 5" */
        17 invokevirtual 28;      /* void println(java.lang.String i) */
        /* L13 */
        20 goto 11;
        /* L15 */
        23 getstatic 20;          /* java.lang.System.out */
        26 ldc 34;                /* "i < 5" */
        28 invokevirtual 28;      /* void println(java.lang.String i) */
        /* L17 */
        31 return;
    }

    public int getnumber() {
        /* L20 */
        0 new 41;
        3 dup;
        4 invokespecial 43;       /* java.util.Random() */
        7 astore_1;               /* r */
        /* L21 */
        8 aload_1;                /* r */
        9 bipush 10;
        11 invokevirtual 44;      /* int nextInt(int arg0) */
        14 ireturn;
    }
}
```

CompareTest/bin/byteCodeCompare/ByteCode2.txt

```
    /* compiled from ExampleByteCode2.java */

    public ExampleByteCode2() {
        /* L5 */
        0 aload_0;                /* this */
        1 invokespecial 8;        /* java.lang.Object() */
        4 return;
    }

    public void main(java.lang.String[] k) {
        /* L9 */
        0 iconst_0;
        1 istore_2;               /* k */
        /* L10 */
        2 aload_0;                /* this */
        3 invokevirtual 16;       /* int getnumber() */
        6 istore_2;               /* k */
        /* L11 */
        7 iload_2;                /* k */
        8 iconst_5;
        9 if_icmple 14;
        /* L12 */
        12 getstatic 20;          /* java.lang.System.out */
        15 ldc 26;                /* "i > 5" */
        17 invokevirtual 28;      /* void println(java.lang.String k) */
        /* L13 */
        20 goto 11;
        /* L15 */
        23 getstatic 20;          /* java.lang.System.out */
        26 ldc 34;                /* "i < 5" */
        28 invokevirtual 28;      /* void println(java.lang.String k) */
        /* L17 */
        31 return;
    }

    public int getnumber() {
        /* L20 */
        0 new 41;
        3 dup;
        4 invokespecial 43;       /* java.util.Random() */
        7 astore_1;               /* randomV */
        /* L21 */
        8 aload_1;                /* randomV */
        9 bipush 10;
        11 invokevirtual 44;      /* int nextInt(int arg0) */
        14 ireturn;
    }
}
```

Figure 4.6: Java byte code compared using text-to-text compare in Eclipse environment

# Chapter. 5

# Graph transformation algorithms

## 5.1 Introduction to the graph transformation

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

## 5.2 Techniques to normalize AST improving structural comparison

Researches show that many big development projects have duplicate 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. But the most sophisticated part comes when a similar command can be in code substituted. It is known, that from programming perspective, for instance, a loop can be differently organized. 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. However the functions execute the same application logic. And this can be found almost everywhere, the most notable example is clone pair between FreeBSD and Linux.

TODO: write that in for trees this transformation can be applied:
Karp-Rabin fngerprinting algorithm is used for calculating the fingerprints of all length n substrings of a text. First, a text-to-text transformation is performed on the con- sidered source le for discarding the uninterested characters. Following this the entire text is subdivided to a set of substrings so that every character of the text appears in at least one substring. After that the matching substrings are identifed. In that stage, a further transformation is applied on the raw matches

to obtain better results. Instead of applying a set of text-to-text transformations, he applies several different transformation scenarios from a combination of basic transformations such as For ifnding near-miss duplication he attempted to find a normalized/transformed text by removing all whitespace characters except line separators and by replacing each maximal sequence of alphanumeric characters with a single letter 'i'. For example, a line for(k = 1; k ¡= n; k + +)f" is replaced by the line i(i = i; i ¡=; i + +g" and the line hash defineXDEF234" by hash iii". TODO:

TODO: write from what this idea comes from: section token based technique: normalization Further research can also be carried out upon cantilever structures or continuous structures.

In the following example 5.2 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.

```
public void test1(){
      int frameGroupLine = 10;
      for(int Cnt = 1; Cnt < frameGroupLine; Cnt =+ 2)
      {
         if(Cnt*4 != 2){
            frameGroupLine++;
         }
      }
   }

public void test2(){
      int frameGroupLine = 10;
      for(int Counter = 1; Counter < frameGroupLine; Counter =+ 2)
      {
         if(Counter*4 != 2){
            frameGroupLine = frameGroupLine + 1;
         }
      }
   }
```

Listing 5.1: Clone pair between FreeBSD and Linux

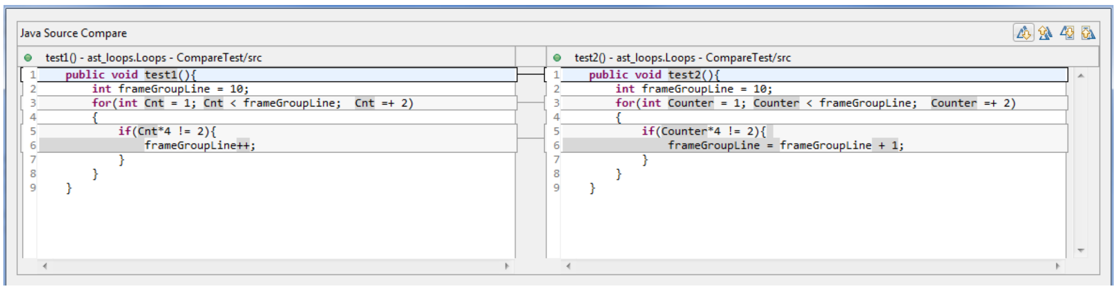
If this line `frameGroupLine = frameGroupLine + 1;` will be converted into one format of sub-tree, that indicates the same as `frameGroupLine++;`. Thus this sub-tree can be found with TDMC or BUMC algorithms. Consequently it brings more covered nodes that signalize more similarity.

```
   public void test3(){
      int frameGroupLine = 10;
      for(int Counter = 1; Counter < frameGroupLine; Counter =+ 2)
      {
         if(Counter*4 != 2){
            frameTeamLine++;
         }
      }
   }
```

Listing 5.2: Normalized function `test3()` concerning variable `frameGroupLine`

In the function `test2()` 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, 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.

On figure 5.1 the example demonstrated how these two functions are compared using Eclipse Text comparison window. After creation and comparison these two



Java Source Compare

test1() - ast_loops.Loops - CompareTest/src

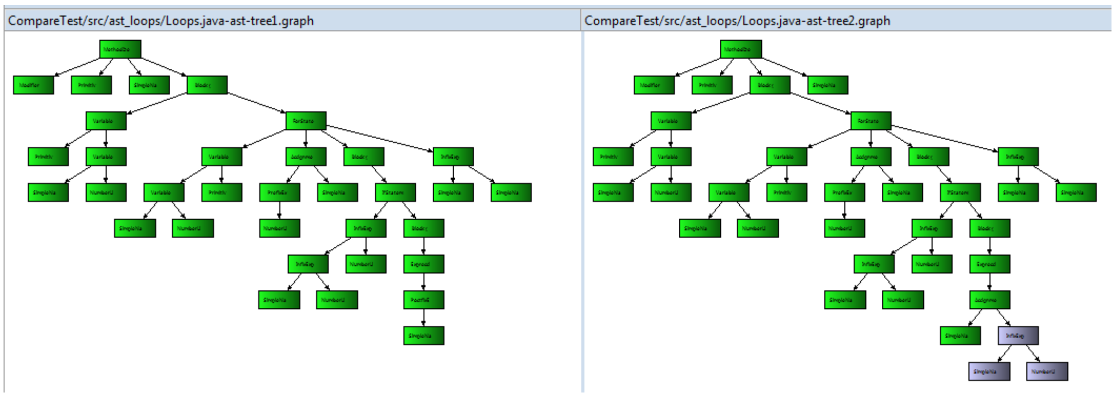
```
public void test1(){
    int frameGroupLine = 10;
    for(int Cnt = 1; Cnt < frameGroupLine;  Cnt =+ 2)
    {
        if(Cnt*4 != 2){
            frameGroupLine++;
        }
    }
}
```

test2() - ast_loops.Loops - CompareTest/src

```
public void test2(){
    int frameGroupLine = 10;
    for(int Counter = 1; Counter < frameGroupLine;  Counter =+ 2)
    {
        if(Counter*4 != 2){
            frameGroupLine = frameGroupLine + 1;
        }
    }
}
```

Figure 5.1: Example of standard text-to-text comparison of Java code

AST trees, it can be easily seen that sub-trees (the increment) are different.



CompareTest/src/ast_loops/Loops.java-ast-tree1.graph
CompareTest/src/ast_loops/Loops.java-ast-tree2.graph

Figure 5.2: Graph comparison of function test1() and test2() using TDMC algorithm 3.1

From the figure 5.2 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.

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.

The table 5.1 presents possible java string transformation depending on the

| 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 5.1: The table demonstrates 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.

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 $T_2$.

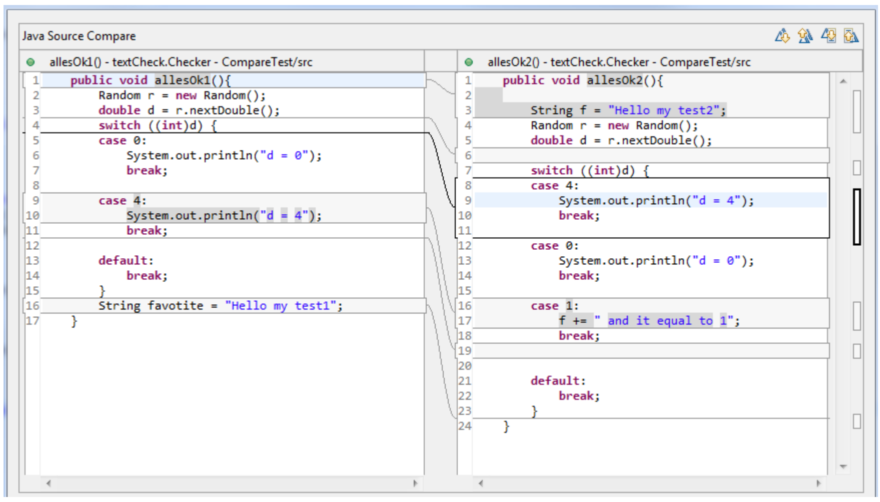
TODO: derive how it is possible to built same types of loops, (IDEA read about tockens techniques)

## 5.3 Convert graph to tree

TODO: decribe here how to remove edges in order to get spanning tree, DO WE NEED IT HERE??

## 5.4 Text comparison improvement with AST trees

In the chapter about existing comparison methods, section section text based techniques 6.2 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.


Java Source Compare
allesOk1() - textCheck.Checker - CompareTest/src
allesOk2() - textCheck.Checker - CompareTest/src

```java
    public void allesOk1(){
        Random r = new Random();
        double d = r.nextDouble();
        switch ((int)d) {
        case 0:
            System.out.println("d = 0");
            break;

        case 4:
            System.out.println("d = 4");
            break;

        default:
            break;
        }
        String favotite = "Hello my test1";
    }
```

```java
    public void allesOk2(){

        String f = "Hello my test2";
        Random r = new Random();
        double d = r.nextDouble();

        switch ((int)d) {
        case 4:
            System.out.println("d = 4");
            break;

        case 0:
            System.out.println("d = 0");
            break;

        case 1:
            f += " and it equal to 1";
            break;


        default:
            break;
        }
    }
```

Figure 5.3: Example of standard text-to-text comparison of Java code

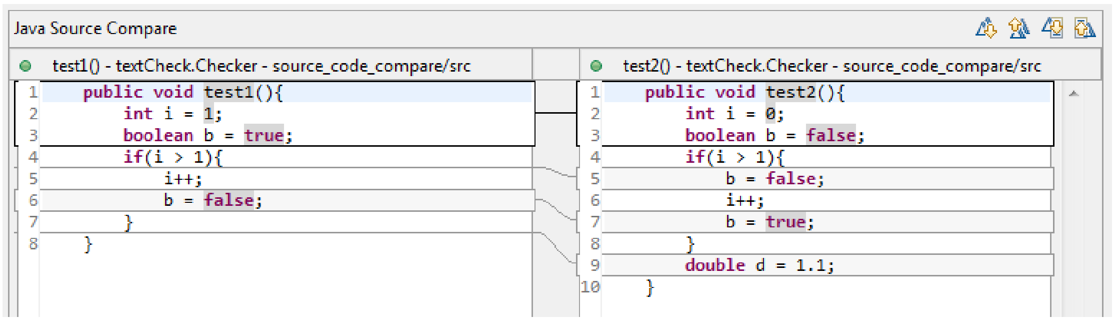
This example 5.3 demonstrates how text compare is not available to notice the string difference in case some statements have been simply replaced. Namely the operators: `case 0:` and `case 4:` from figure 5.3 have been replaced but text compare find this difference.

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 6.2. 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 5.3 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;



Java Source Compare

test1() - textCheck.Checker - source_code_compare/src

```
1     public void test1(){
2         int i = 1;
3         boolean b = true;
4         if(i > 1){
5             i++;
6             b = false;
7         }
8     }
```

test2() - textCheck.Checker - source_code_compare/src

```
1     public void test2(){
2         int i = 0;
3         boolean b = false;
4         if(i > 1){
5             b = false;
6             i++;
7             b = true;
8         }
9         double d = 1.1;
10    }
```

Figure 5.4: 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 5.4. 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.

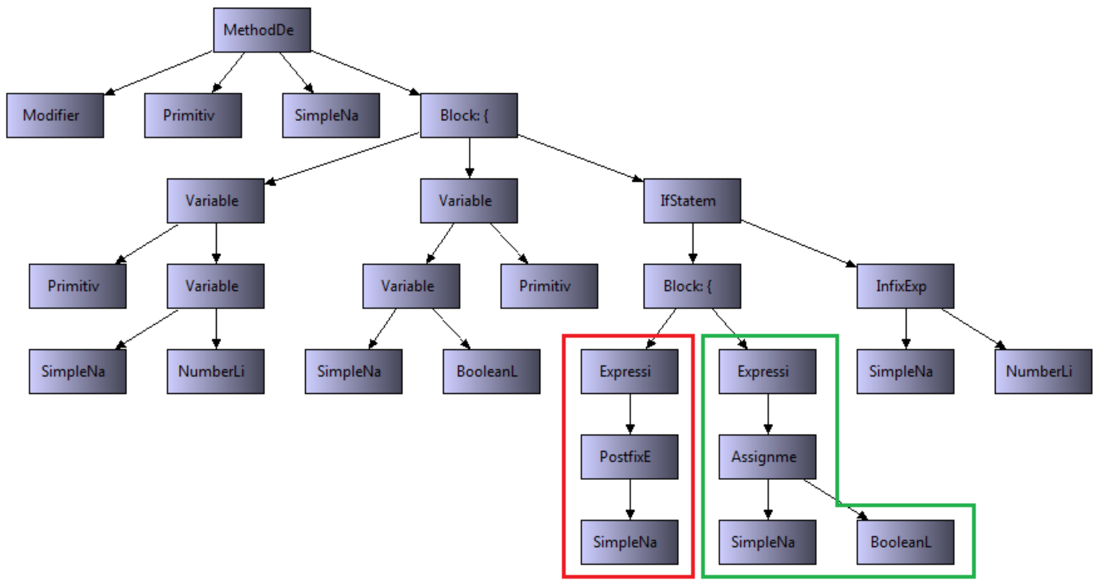


Figure 5.5: Abstract Syntax Graph of function `test1()`.

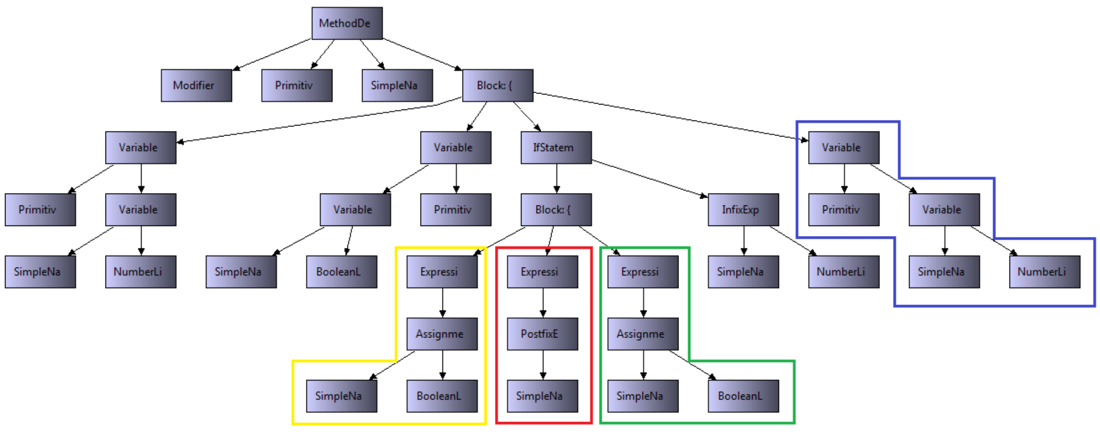


Figure 5.6: Abstract Syntax Graph of function `test2()`.

Example 5.4 and derived AST trees 5.5 and 5.6 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 this statement must be highlighted in text compare windows as *mismatched*.

3. The blue frame in picture 5.6 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 5.5 shows how AST tree from function `test1()` is being built from the source code. As said above, the presented AST tree 5.5 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 $test1()$ and $test2()$ 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 3.1)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.

The figure 5.7 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

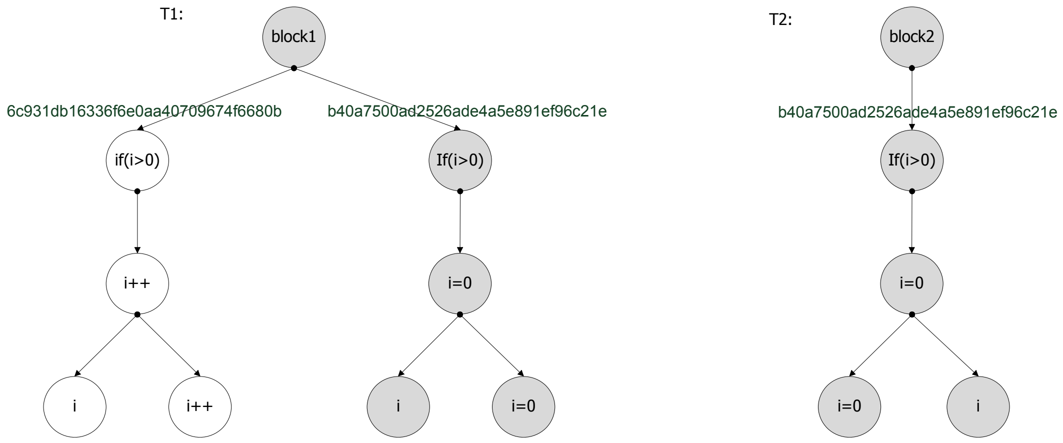

T1:
block1
6c931db16336f6e0aa40709674f6680b
b40a7500ad2526ade4a5e891ef96c21e
if(i>0)
If(i>0)
i++
i=0
i
i++
i
i=0
T2:
block2
b40a7500ad2526ade4a5e891ef96c21e
If(i>0)
i=0
i=0
i

Figure 5.7: Example of hashed branch content assists to optimize the algorithm search for clones. The more similar compared fragments of the faster the algorithms execution.

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 6.5. 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 6.5. 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. 6

# Existing Comparison methods

## 6.1 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.

> Plagiarism detection is the process of locating instances of plagiarism within a work or document. The widespread use of computers and the advent of the Internet has made it easier to plagiarize the work of others. Most cases of plagiarism are found in academia, where documents are typically essays or reports. However, plagiarism can be found in virtually any field, including scientific papers, art designs, and source code [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.

## 6.2 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. This section shortly explains how it functions and derived advantages and disadvantages from chapter experiments 4. 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 splitted into strings, 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).
TODO: suffix tree example 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 4 the next conclusion can be derived.
Advantages of text-to-text comparison:

- 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:

- 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 application logic more independent from text, therefore to improve quality of text difference representation. The paragraph 5.4 interprets an idea how AST tree assists to specify precisely match/mismatch of command .

## 6.3 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 [6.2]. 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 [6.2] can be significantly improved. Simple based normalization of code structure, namely transformation in tokens, brings better results as compare to native text comparison [6.2].

## 6.4 Tree-based Techniques

The research from Chanchal Roy [6] offers an alternative to figure out differences using trees. 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-string6.2. 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 6.3 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 subsequence 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.

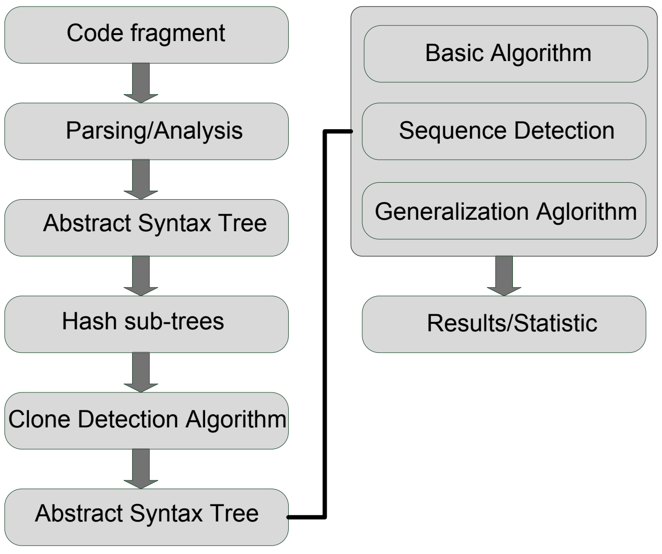


Figure 6.1: The flowchart demonstrates general steps of code comparison with AST trees. Once the Abstract Syntax tree has been created, a sequence of algorithms is applied.

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 highest 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** 5.4.

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.

## 6.5 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 6.4, 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.

```
private static String sample(){
    String text = null;
    int i = 0;
    while(i < 10){
```

```
5           i++;
6           text.compareTo("my text");
7           text.trim();
8       }
9       return text;
10  }
```

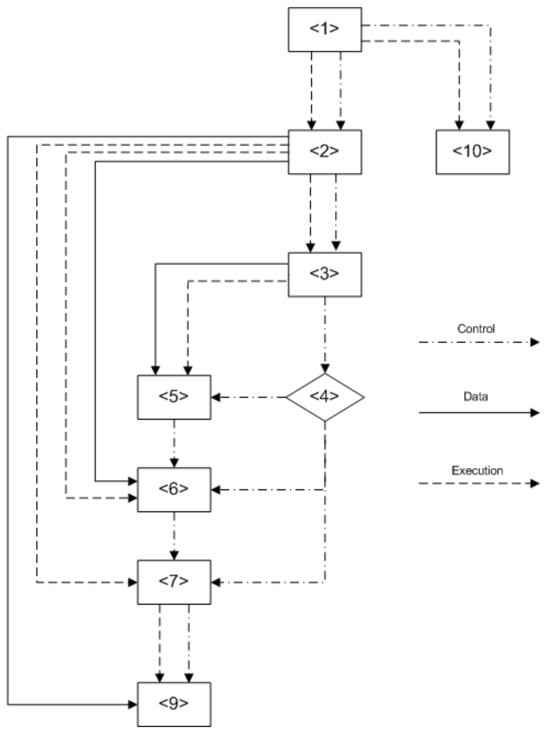


Figure 6.2: Program Dependency Graph derived from Java source code

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 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 6.4. 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()`:

```java
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:

```java
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 6.2 or AST trees 6.4 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 5.4.
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. 7

## Conclusion

based on experimental results and own opinion, write here what results were derived From time to time write here combined conclusions or improvements 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`

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