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

September 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

TDMC	**T**op **D**own **M**ax **C**ommon
BUMC	**B**ottom **U**p **M**ax **C**ommon
BCV	**B**yte **C**ode **V**isualizer
SCV	**S**ource **C**ode **V**isualizer
CFGF	**C**ontrol **F**low **G**raph **F**actory
TC	**T**ext **C**ompare

Chapter. 1

Introduction

Modern methods to compare of programming pieces of code are used to analyze code's changing, to explore development process and so on. Basically in current tools or plug-ins only text compare methods are used, that is not full sufficient to define code compare. Sometimes another techniques can be very helpful for such purposes. One of them is a structural code compare, based on building a trees, and methods to compare any similar or same structures.

TODO START: You can't write a good introduction until you know what the body of the paper says. Consider writing the introductory section(s) after you have completed the rest of the paper, rather than before. Be sure to include a hook at the beginning of the introduction. This is a statement of something sufficiently interesting to motivate your reader to read the rest of the paper, it is an important/interesting scientific problem that your paper either solves or addresses. You should draw the reader in and make them want to read the rest of the paper.

REDO: Code duplication or copying a code fragment and then reuse by pasting with or without any modifications is a well known code smell in software maintenance. Several studies show that about 5 to 20 percent of a software systems can contain duplicated code, which is basically the results of copying existing code fragments and using then by pasting with or without minor modifications. One of the major shortcomings of such duplicated fragments is that if a bug is detected in a code fragment, all the other fragments similar to it should be investigated to check the possible existence of the same bug in the similar fragments. Refactoring of the duplicated code is another prime issue in software maintenance although several studies claim that refactoring of certain clones are not desirable and there is a risk of removing them. However, it is also widely agreed that clones should at

least be detected. REDO

Tips: A statement of the goal of the paper: why the study was undertaken, or why the paper was written. Do not repeat the abstract.

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.

Structural compare stands for comparing two graphs. There are many possibilities how to create a graph from java source or byte 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. 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?

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.

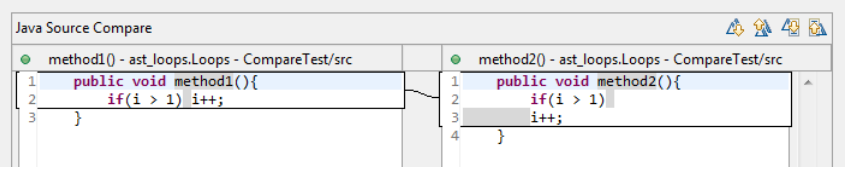

Figure 2.1: The simplest example of text-to-text comparison indicating that text compare sometimes is not enough to investigate code difference, however there are no changes regards the application logic

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 subtree and Bottom Up maximum common subtree 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

Graphs comparisons 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. 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 An algorithm for Top-down maximum common sub-tree isomorphism

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 $size[X_1] < size[X'_1]$.*

Figure 3.1: Example of standard text-to-text comparison of Java code

3.2 An algorithm for Bottom-Up maximum common sub-tree isomorphism

TODO: explain briefly what this technique, examples from Valiente

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 [8] 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 "text compare" is already implemented in Eclipse IDE [8], 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.

TODO: Add Text compare picture

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.

TODO: explain briefly how TEXT compare works(search in internet), and sometimes it's not enough to observe the difference

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).

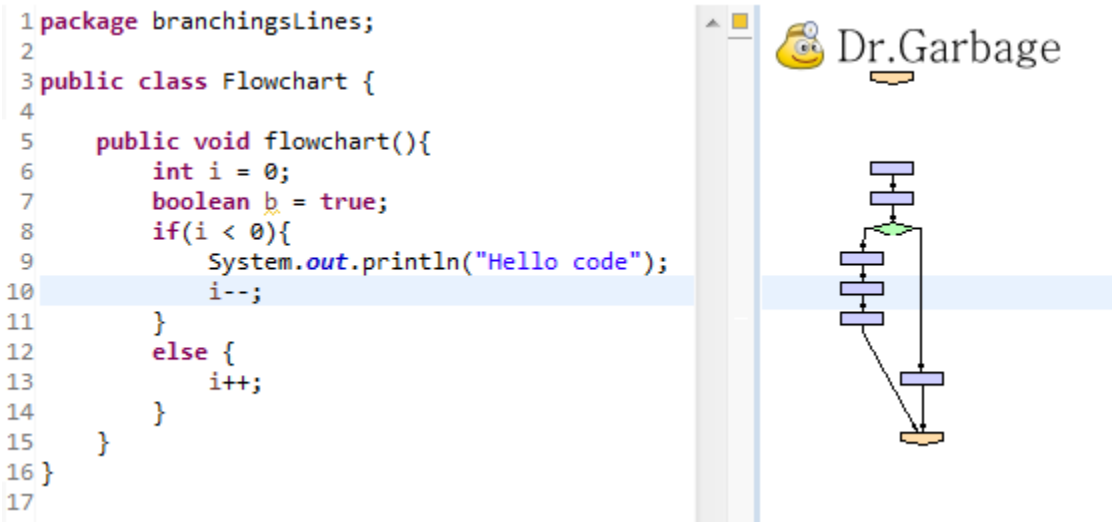

Figure 4.1: Example of source code visualizer

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 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.

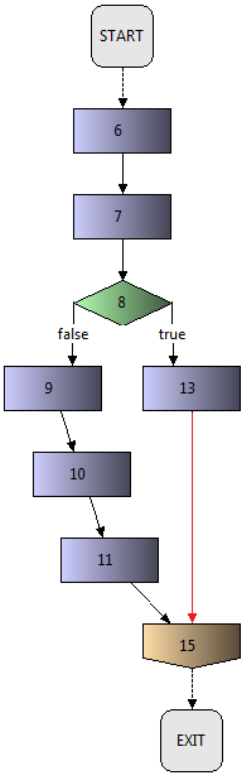

Figure 4.2: Extracted control flow graph from Java source code

For the estimation of structural difference there are criteria listed:

- 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

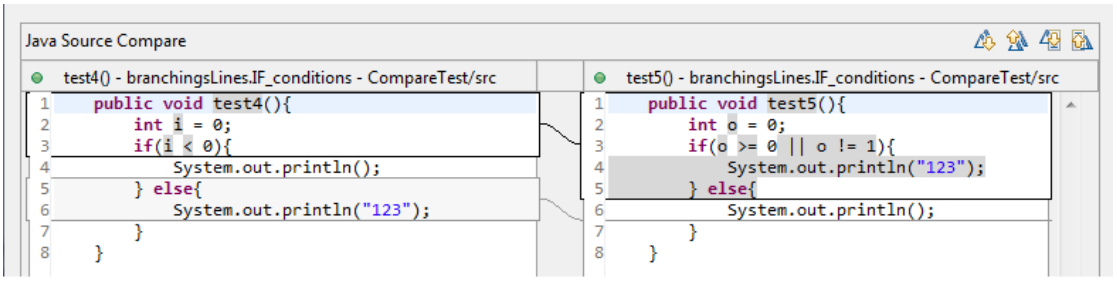

Figure 4.3: Two pieces of code are being compared with Eclipse Text Comparison

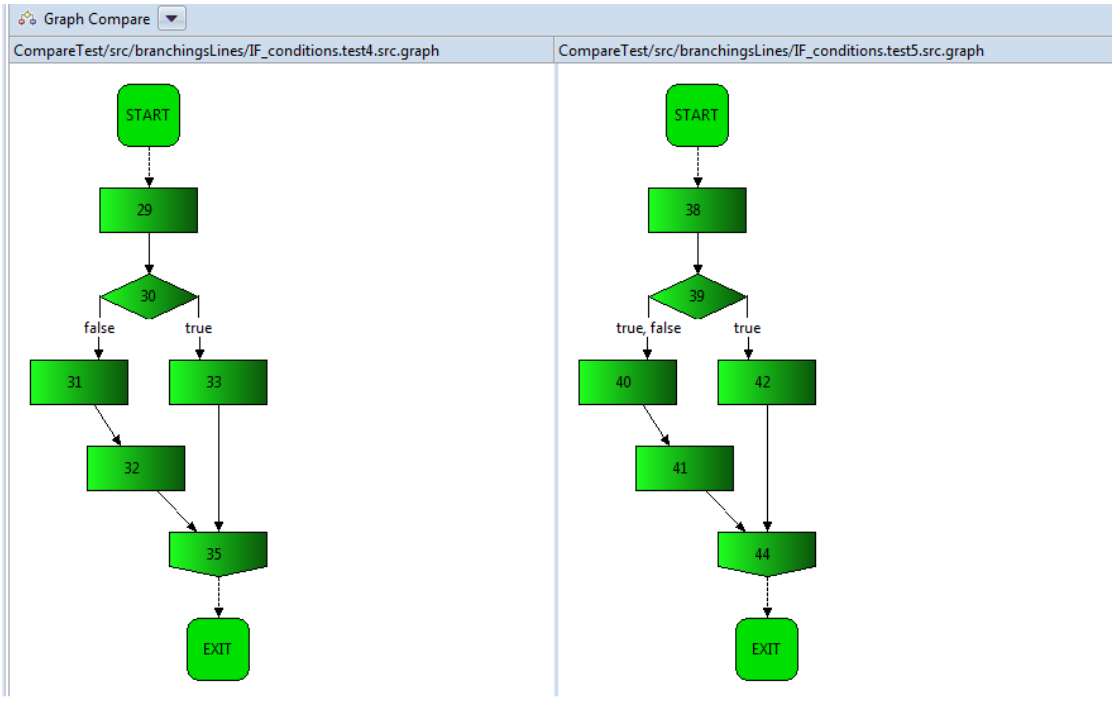

Figure 4.4: Compared source code graphs using TDMC algorithm

of code comparison. As it said above, the difference code can be figured out 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;

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

IDEA: compare AST AST optimization, for each node the label must be equal to the corresponding value in the code. go through the nodes, and compare the values in the nodes. Optimization: substitute some AST construction like `i++;` to the `i = i + 1;`, helps to find similarity AST native trees will more abstract (almost the same as JAVA compiler converts any code to). ATTACH tree AST and AST native

ARTICLE: Parsing: In case of parse tree-based approaches, the entire source code base is parsed to build parse tree or (annotated) abstract syntax tree (AST). In such representation, the source unit and comparison units are represented as subtrees of the parse tree or AST. Comparison algorithm then uses these subtrees to find clones [31, 213, 222]. Metrics-based approaches may also use such representation of code to calculate of the subtrees and find clones based on the metrics values [146, 178].

```
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;
			}
		}
	}
```

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 frameTeamLine = 10;
		for(int i = 1; i < frameTeamLine; i =+ 2)
```

```
            {
                    if(i*4 != 2){
                            frameTeamLine++;
                    }
            }
    }
```

The code fragments test1() and test2() have the same application logic but different variables. However in the second 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.

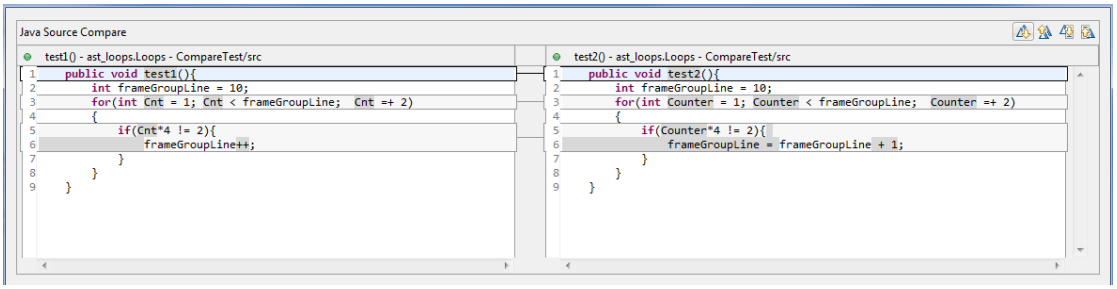

Figure 4.5: Example of standard text-to-text comparison of Java code

On figure 4.5 the example demonstrated 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.

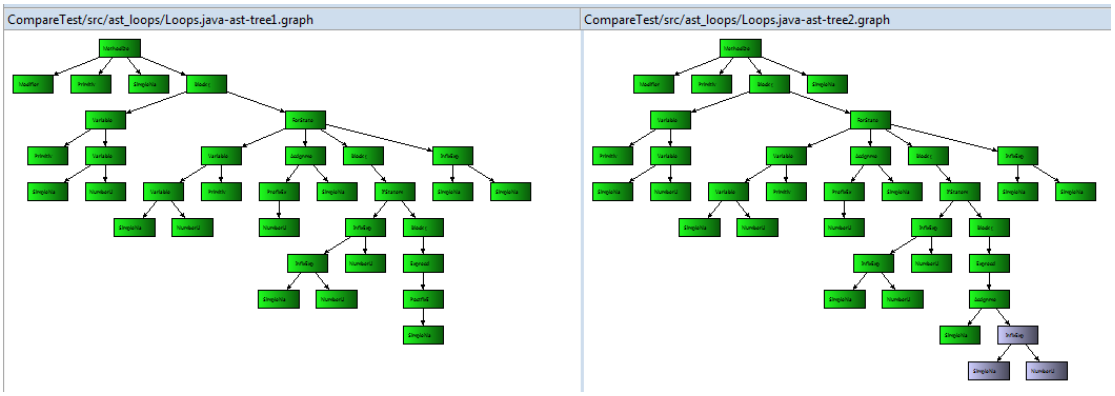

Figure 4.6: Graph comparison of function test1() and test2() using TDMC algorithm 3.1

From the figure 4.6 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.

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.

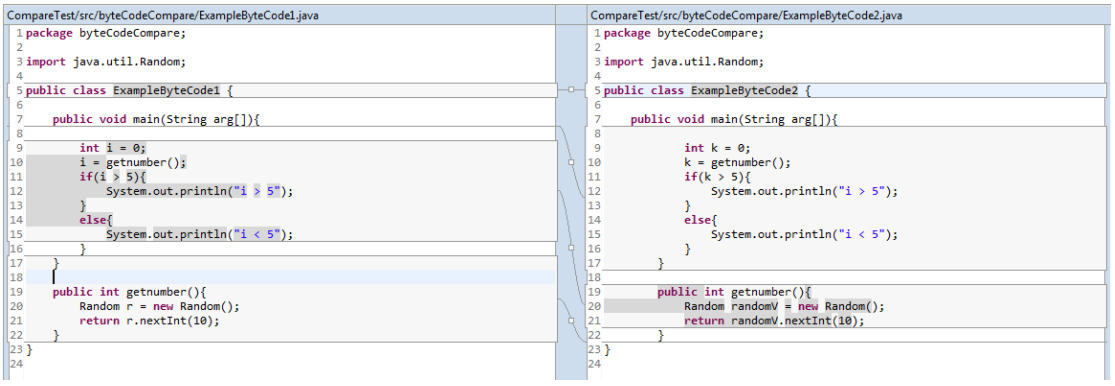

Figure 4.7: Java source code compared using text-to-text

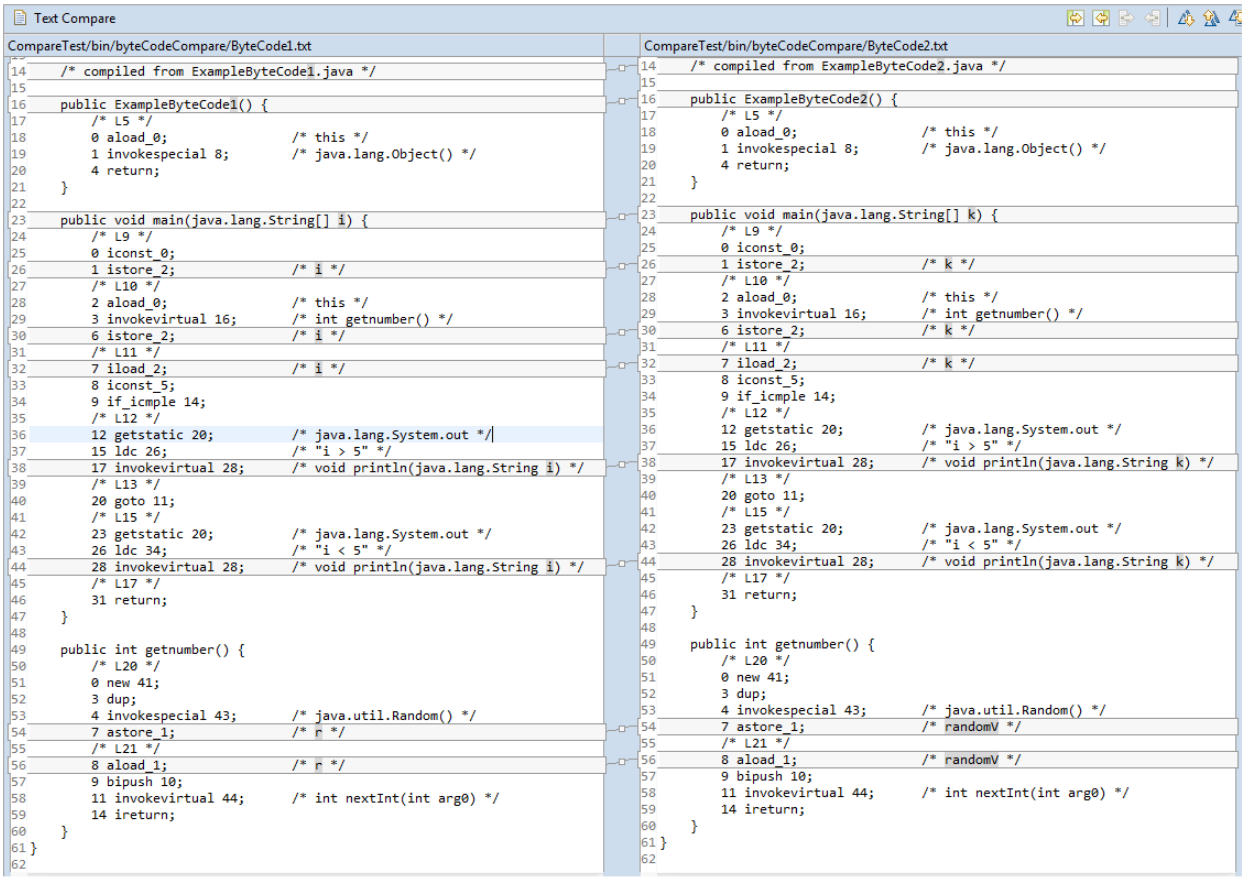

Figure 4.8: Java byte code compared using text-to-text

Compare experiments			Text -to-text compared	Graphcompared
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

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

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 build a tree from code

TODO: what is AST, examples;
TODO: what is BasicBlock, examples;

5.3 Convert graph to tree

TODO: decribe here how to remove edges in order to get spanning tree

5.4 Possible ideas

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 :

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

Chapter. 7

Conclusion

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

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

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