ABSTRACT
This paper presents graph theoretical algorithms to compare control flow graphs of program code based on graph transformation and decomposition into connected components. The essential idea is to transform graphs into reduced graphs and to calculate an isomorphic correspondence.

We also show some experimental results to illustrate the effectiveness of our algorithms in terms of detected structural differences in graphs that are compared. The presented algorithms in this paper have been implemented in the Dr. Garbage tool suite and are available for download under the Apache Open Source license.

KEY WORDS
Control flow graph comparison, graph isomorphism.

1 Introduction
The Control Flow Graph Comparison is important in several fields of application such as efficient testing of programs, merging between two versions of software, testing compiler optimization and code instrumentation tools.

To compare the structure of two control flow graphs it is essential to know if one graph contains the structure of the second graph. A computational task in which two graphs $G_1$ and $G_2$ are given as input, and one must determine whether $G_1$ contains a subgraph that has the same structure as the graph $G_2$, is called the subgraph isomorphism problem. The subgraph isomorphism problem is a fundamental problem in graph theory and it is known to be NP-complete [1]. Fortunately polynomial-time algorithms for the subgraph isomorphism problem are known for trees [2], two-connected outerplanar graphs [3], and two-connected series-parallel graphs [4].

In this paper we present graph theoretical algorithms to compare control flow graphs of program code and some extensions based on graph transformation and decomposition. The essential idea is to transform graphs into trees and to calculate an isomorphic correspondence. To consider the whole graph structure, the graph is decomposed into connected components and transformed to a reduced tree. We didn’t solve the graph isomorphism problem, but all algorithms presented in this paper have linear time complexity and show very good results in terms of detected structural differences in graphs that are compared.

All algorithms presented in this paper have been implemented in the context of the Dr. Garbage tool suite [5] and we present some experimental results and statistics of our implementation in section 5.

2 Related Works
The problem of comparing two graphs is known for a long time and it is one of the fundamental problems of graph theory. A. V. Aho, J. E. Hopcroft, and J. D. Ullman present in their book *The Design and Analysis of Computer Algorithms* [6] algorithms for solving general problems of subtree isomorphism, largest common subtree and smallest common supertree. S. M. Selkow defined the Top-Down subtree isomorphism [7]. A definition of the Bottom-Up subtree isomorphism was introduced by G. Valiente in [8], [9]. A systematic review of efficient subtree isomorphism algorithms can be found in his book *Algorithms on Trees and Graphs* [10]. The Top-Down and Bottom-Up subtree isomorphism algorithms are used in our proposal to compare the reduced graphs as trees. After the trees have been compared, the differences are represented in the graphs from which the trees have been derived.

There are a number of publications that deal with the comparison of program code. W. Yang presents an approach to compare the structure of two programs by using the subtree isomorphism [11]. T. Sager introduced an approach and a tool that measures similarity between Java classes [12]. The tool calculates similarity by using different subtree isomorphism algorithms on syntax tree representations of source code. Compared to our proposal, in both approaches syntax trees derived from the source code are used instead of control flow graphs. The problem is reduced to subtree isomorphism which can be solved in linear time. Unfortunately, this approach can not be applied directly for the comparison of two control flow graphs.

J. Laski and W. Szermer present an algorithm that finds differences of two programs by comparing control flow graphs [13]. In their paper they address the problem of the revalidation of a modified code in software mainte-
nance. The modifications are localized by using the control flow graphs of the original and modified programs. Both flow graphs are transformed into reduced flow graphs, between which an isomorphic correspondence is calculated. A similar work and a further improvement of the algorithm can be found in [14]. The authors introduced a hammock matching algorithm based on Laski and Szermer’s algorithm. This algorithm uses hammocks and reduced hammock graphs. Hammocks are subgraphs [15] which provide a way to impose a hierarchical structure on the control flow graph for matching. This approach is very close to our proposal. In our approach a different type of control flow graph reduction is performed and the comparison is based on subtree isomorphism.

Furthermore the problem of comparing the structure of two graphs can be turned into determining the similarity of two graphs, which is generally referred to as graph matching problem. There are a number of publications of graph matching algorithms. The graph matching has been applied to semantic networks [16], recognition of graphical symbols [17], three-dimensional object recognition [18], and others. However the applications and algorithms have a very low relevance to our approach because the graphs have always specific properties according to the field of application.

3 Subtree Isomorphism Algorithms

Since our approach is based on the subtree isomorphism, the most commonly used algorithms are explained here briefly. There are three basic types of subtree isomorphism: largest common [6], top-down [7] and bottom-up [8], [9] subtree isomorphism.

The largest common subtree of two given trees \( T_1 \) and \( T_2 \) (figure 1 ) is the largest subtree of \( T_1 \) that is structurally identical to a subtree in \( T_2 \).

![Figure 1. Largest common subtree.](image)

The top-down subtree of two given trees (figure 2) is the largest common subtree under the prerequisite that the root of the common subtree is identical with the root nodes of the compared trees \( T_1 \) and \( T_{2a} \). The bottom-up subtree is the largest isomorphic subtree of two given trees if the bottom-up subtree of \( T_1 \) rooted at node \( v_0 \) is isomorphic to the bottom-up subtree of \( T_{2b} \) rooted at node \( v_2 \).

![Figure 2. Top-down, bottom-up subtree isomorphism.](image)

There are many publications about the subtree isomorphism and corresponding algorithms. A very good overview of subtree isomorphism algorithms is the book by G. Valiente [10]. In his book Valiente presents a number of subtree isomorphism algorithms and provides efficient code implementation examples. We implemented a prototype based on the top-down maximum common subtree isomorphism algorithm by G. Valiente. Our approach can be easily adapted to the bottom-up maximum common subtree isomorphism or tree edit distance algorithms [10] and differs only in the last step by the use of another subtree similarity algorithm.

4 Graph Comparison Algorithms

The input for the comparison algorithms are control flow graphs (CFG) of a method. The CFG is defined as a tuple \( G = (V, A) \), where \( V \) is a nonempty set of vertices representing instructions of a method, \( A \) is a (possibly empty) set of arcs (or edges) representing transitions between the instructions. Formally, \( A \) is the finite set of ordered pairs of vertices \( (a, b) \), where \( a, b \in V \).

The algorithm COMPARE_SPANNING_TREES (section 4.1) computes spanning trees of the CFGs to be compared and calculates the subtree isomorphism. The differences detected in these trees are transferred into the original graphs. The algorithm COMPARE_BB_GRAPHS (section 4.3) extends the algorithm COMPARE_SPANNING_TREES by the graph transformation to basic block graphs. The subtree isomorphism procedure is applied to a substantial graph structure. The last algorithm COMPARE_BIBLOCK_GRAPHS (section 4.3) transforms the basic block graph in a so called biblock (biconnected blocks) graph. The biblock graph is a tree constructed by graph decomposition into connected components [19], [20]. The subtree isomorphism procedure is applied to the biblock tree and for all isomorphic biblocks the algorithm COMPARE_BB_GRAPHS is executed.

4.1 Spanning Tree Comparison Algorithm

As a CFG may contain loops, the CFG has to be transformed into a directed acyclic graph (DAG) by removing loop backedges as identified by a depth-first search of the CFG (figure 3: line 1). For the given CFGs the algorithm computes their respective ordered spanning trees. Each node in a program CFG has usually a unique instruc-
The procedure `createOrderedTree` (figure 3: lines 2-3) is a modified breadth-first search (BFS). It uses instruction code addresses to define the traversing order of outgoing edges and assigns a logical address to each node in the tree corresponding to the steps of the BFS algorithm. This ensures that a deterministically ordered spanning tree is achieved by the subtree isomorphism algorithm (figure 3: line 4). We implemented the top-down subtree isomorphism algorithm from G. Valiente [10]. After executing the `subtreeTopDownIsomorphism` procedure the isomorphic vertices are marked green and non isomorphic vertices red. Finally the edges missing in the spanning tree and previously removed back edges are compared and marked red, if the source’s or target’s logical addresses of the edge \( e \in T_1 \) is not equal to the addresses of the corresponding edge \( e' \in T_2 \).

The figure 4 illustrates an ordered spanning tree after the execution of the procedure `createOrderedTree`. The resulting spanning tree misses the edge \( v_2 \rightarrow v_3 \). The logical addresses are assigned to each node.

The algorithm `COMPARE_SPANNING_TREES` is very simple, but has a big disadvantage. If the difference in the top-down approach is detected right at the beginning, the remaining tree structure is not further compared by the algorithm.

### 4.2 Basic Block Graph Comparison Algorithm

To work around the limitation of the algorithm `COMPARE_SPANNING_TREES`, the compared CFGs are reduced to basic block graphs. A basic block graph is a control flow graph with basic blocks as vertices. A basic block (BB) is a maximal sequence of one or more consecutive instructions (previously separate vertices) with a single entry instruction, a single exit instruction, and no internal branches.

The algorithm `COMPARE_BB_GRAPHS` (figure 5) transforms the graphs \( G_1 \) and \( G_2 \) to BB-graphs \( B_1 \) and \( B_2 \) (line 1-2). The `COMPARE_SPANNING_TREES` algorithm is applied to the generated BB-graphs. Figure 5: line 3 generates spanning trees and calculates the subtree isomorphism. For isomorphic BBs detected by the algorithm the content is compared. In our prototype we compare the length of the instruction sequence in basic blocks. If the lengths are different the excessive vertices (instructions) are marked red.

The algorithm is able to detect if any instruction sequence have been inserted or deleted in the code and the comparison does not terminate after detecting the first difference. Figure 6 illustrates this step. The insertion or deletion of a sequence of vertices within the basic blocks can be easily detected. The algorithm `COMPARE_BB_GRAPHS` is sufficient for use cases which define the sequential modification of code. For example if an instrumentation tool inserts instructions for monitoring, it would be sufficient to check if the original and instrumented basic block graphs are isomorphic. The comparison will terminate as soon as complex structural differences are detected. Further reduction of graphs is necessary to avoid this problem.
4.3 Graph Comparison Algorithm by Decomposition

The idea to decompose graphs for comparison is not new. Valiente and Martinez for example present in [21] an algorithm for pattern-matching on arbitrary graphs that is based on decomposing graphs into connected components. For every connected component, the algorithm performs a combinatorial search.

Our algorithm in figure 7 transforms a control flow graph to basic block graphs (lines: 1-2) and creates so called biblock trees (lines: 3-4). The subtree isomorphism algorithm is applied to biblock trees (line 5) and for each detected isomorphic vertex the COMPARE_BB_GRAPHS algorithm is applied (lines: 6-8).

Graphs are usually decomposed into connected components, where a connected component is a maximal bi-connected subgraph or a subgraph generated from a bridge or an isolated vertex. G. Stiege introduced in [19] a similar graph decomposition called biblock decomposition, which is better suited for graph algorithmic problems. The decomposition uses undirected paths. The graph is decomposed into maximal 2-edge-connected subgraphs (subcomponents) and trees, which are connected with the subcomponents. The subcomponents are subdivided into maximal 2-vertex-connected subgraphs, the biblocks. A graph produced by the reduction of the biblocks into single nodes, is called a biblock-tree. An efficient algorithm to find the biblock structure can be found in [20].

Figure 8 shows control flow graph $G_1$ and the modified graph $G_2$. The vertex $v_0 \in G_1$ has been replaced by the branch instruction block $\{v_0, v_0', v_0''\} \subset G_2$. The instruction block $\{v_8, v_5, v_7, v_8\} \subset G_1$ has been replaced by the single instruction $v_5 \in G_2$. The algorithm COMPARE_SPANNING_TREES stops the comparison process after processing the first vertices, because the children of the first vertex $v_0$ in the generated spanning trees have different numbers of outgoing edges. It marks vertices $v_0, v_1, v_5 \in G_1$ and vertices $v_0, v_0', v_0'' \in G_2$ as isomorphic.

The COMPARE_BB_GRAPHS algorithm transforms the graphs $G_1$ and $G_2$ to biblock-trees and compares their structure. The biblock-trees of the graph $G_1$ and $G_2$ are shown in figure 9. The biblock-trees are isomorphic.
missing in $G_1$. The vertices $v_6, v_7, v_8 \in G_1$ are marked red, because these vertices are missing in $G_2$.

5 Experimental Evaluations

To evaluate our algorithms, we selected control flow graphs of program code, that contain if, switch, loop and return instructions. The program methods that have been used for the generation of graphs are listed in table 1. The method $m<\chi>a$ is produced from the method $m<\chi>$ by inserting a single instruction. The method $m<\chi>b$ represents the modification of the method $m<\chi>$ by inserting a block of instructions.

Table 2 illustrates the comparison results of two graphs $M1$ and $M2$. An acronym represents the used algorithm: $TD$ - COM-PARE_SPANNING_TREES, $BBTD$ - COM-PARE_BB_GRAPHS and $BiBTD$ - COM-PARE_BIBLOCK_GRAPHS.

Figure 11 represents the results from the column G+R1% of the table 2. The vertical axis is the ratio of all marked vertices (green + red) to all vertices in $M1$. The Light bar represents the result of the algorithm COM-PARE_SPANNING_TREES, the light-dark bar represents the results of the algorithm COM-PARE_BB_GRAPHS and the dark bar represents the results of the algorithm COM-PARE_BIBLOCK_GRAPHS.

The test case $m1-m1b$ is particularly interesting because the result of algorithm $BBTD$ is worse than the result of algorithm $TD$. This result is caused by the insertion of a block of instructions in the middle of the program code. The labels of the subsequent nodes in the control flow graph can not be assigned in the same order as in the original graph. So the isomorphism of the compared graphs cannot be uniquely determined. This effect is abrogated by using further graph decomposition into biblocks. In a more extensive example $m3-m3b$ the algorithm $BiBTD$ returns by

<table>
<thead>
<tr>
<th>$M$</th>
<th>$I$</th>
<th>$IF$</th>
<th>$S(c)$</th>
<th>$L$</th>
<th>$R$</th>
<th>$BB$</th>
<th>$BiB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m1$</td>
<td>32</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>$m1a$</td>
<td>33</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>$m1b$</td>
<td>37</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>$m2$</td>
<td>20</td>
<td>1</td>
<td>1(4)</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>$m2a$</td>
<td>21</td>
<td>1</td>
<td>1(4)</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>$m2b$</td>
<td>25</td>
<td>2</td>
<td>1(4)</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>$m3$</td>
<td>111</td>
<td>13</td>
<td>2(8)</td>
<td>3</td>
<td>7</td>
<td>49</td>
<td>14</td>
</tr>
<tr>
<td>$m3a$</td>
<td>113</td>
<td>13</td>
<td>2(8)</td>
<td>3</td>
<td>7</td>
<td>49</td>
<td>14</td>
</tr>
<tr>
<td>$m3b$</td>
<td>117</td>
<td>14</td>
<td>2(8)</td>
<td>3</td>
<td>7</td>
<td>52</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1. $M$: name of the method, $I$: Number of instructions, $IF$: Number of if-branches, $S(c)$: Number of switches and their cases, $L$: number of loops, $R$: Number of returns $BB$: Number of Basic Blocks, $BiB$: Number of BiBlocks
far the best result.

The most important conclusion is that the algorithm BiBTD always produces the same or better result than the algorithms TD and BBTD.

6 Conclusion

In this paper we presented algorithms for comparing two control flow graphs which are based on graph transformation into a tree and reduction by decomposition into connected components.

Experimental results show that the proposed algorithms can be successfully applied for comparing the code differences. The structural reduction of the control flow graphs makes it possible to capture the overall structure of a program code. On the other hand the results also show the limitations of the proposed algorithms. Although the decomposition does not provide a worse outcome, it helps only in certain cases to improve the comparison result.

In future work we will investigate additional heuristics to improve the matching results by the combination of graph theoretical and code specific approaches. We will also continue to improve our tools and collect more meaningful examples and statics.

Acknowledgement

The authors would like to thank the Dr. Garbage Project Community for supporting the implementation of the proposed algorithms and tools.

References


