Received December 16, 2019, accepted March 4, 2020, date of publication March 16, 2020, date of current version March 25, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2981260

Tree Inheritance Distance

DANIJEL MLINARIĆ, (Member, IEEE), BORIS MILAŠINOVIĆ, (Member, IEEE), AND VEDRAN MORNAR, (Member, IEEE)
Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
Corresponding author: Danijel Mlinarić (danijel.mlinaric@fer.hr)

ABSTRACT Tree comparison is an important method in various areas. In order to compare class hierarchy in object-oriented languages, there is a need to compare trees in the context of the hierarchy changes. This paper addresses tree dissimilarity based on two measures. First, changes in edges between nodes and parents are measured by introducing Edge Edit Distance (EED). Second, changes in inheritance relationships between nodes are measured by introducing Tree Inheritance Distance (TID). It is shown that EED and TID increase with the dissimilarity in the class hierarchies of the compared program versions.

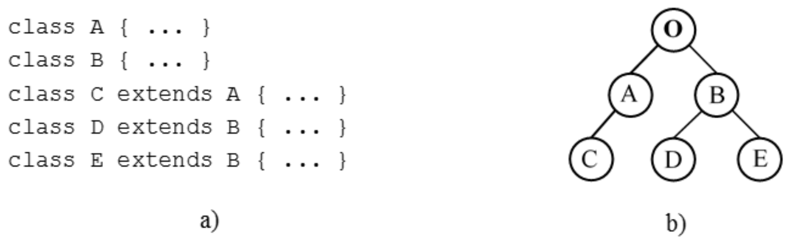
INDEX TERMS Edit distance, inheritance tree, program analysis, tree dissimilarity measures.

I. INTRODUCTION
As trees represent hierarchy organized data, they are widely used in various areas, such as computer vision [1], structured documents [2], natural language processing [3], phylogenetic studies [4], and molecular biology [5], [6]. The main goal in representing data patterns as trees or generally as graphs is to identify the changes in the data. Our motivation is to compare class hierarchy between program versions in object-oriented programming languages. In related literature [4], [7], [8] trees such as Abstract Syntax Trees (AST) are used to determine the difference between program versions on the syntax level, which can be used for program analysis on the intraprocedural level. However, AST can be used with other tree representations to compare classes as in [9]. In this paper, we are considering the relationship between classes in the hierarchy of object-oriented languages. The dissimilarity between hierarchies can be used for program analysis in various software engineering tasks such as detection of code clones [10], regression testing [11], and dynamic software updating [8]. To compare the class hierarchy between two program versions, we found an unordered labeled tree as the most suitable hierarchy structure. The unordered tree corresponds to the class hierarchy in object-oriented languages, where the root node corresponds to the elementary class, e.g., Object class in the Java programming language, as illustrated in Figure 1.

To compare trees, or generally graphs, it is necessary to use a dissimilarity measure. In related literature [12], [13], dissimilarity measures usually fall into two elementary categories: isomorphism [14] and edit distance [15]–[17]. Isomorphism represents an exact matching between two trees or subtrees. Edit distance, on the contrary, provides inexact, i.e., error-tolerant matching [13]. Isomorphism attempts to find a bijective function or mapping from one tree to another tree, or subtree, answering whether the tree is equal to another tree or subtree. On the other hand, Tree Edit Distance (TED) [18], [20] can compare entirely different trees, measuring the dissimilarity between them by the amount of modification of nodes and edges required to transform one tree into another.

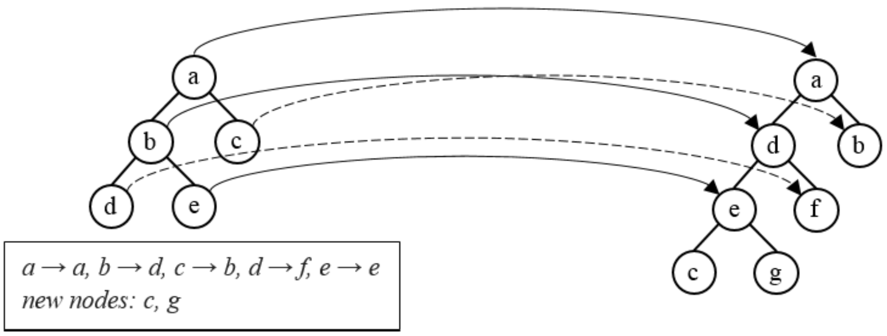
Tree edit distance is more suitable to detect differences in the trees representing the class hierarchy because various program versions can have an entirely different class hierarchy. However, to the best of our knowledge, existing tree edit distance measures are not appropriate to detect hierarchy changes in trees such as inheritance changes, because their goal is to find similar parts of trees. On the other hand, in this paper, we consider the problem of tree dissimilarity with an emphasis on the relationship between nodes. Unordered tree transformation is considered from the edge aspect, to observe changes in the relationship between nodes. As a contribution, Edge Edit Distance (EED) is introduced, a dissimilarity measure between unordered trees based on changes in the edge between node and parent. Furthermore, the major contribution is Tree Inheritance Distance (TID), a dissimilarity measure between unordered trees based on changes in the inheritance relationship between nodes. For both dissimilarity measures, efficient algorithms are presented and evaluated by experiments.

The associate editor coordinating the review of this manuscript and approving it for publication was Shuiguang Deng.



```
class A { ... }
class B { ... }
class C extends A { ... }
class D extends B { ... }
class E extends B { ... }
```

FIGURE 1. a) Class declaration b) Class hierarchy represented by a tree.



a
b
c
d
e
$a \rightarrow a$, $b \rightarrow d$, $c \rightarrow b$, $d \rightarrow f$, $e \rightarrow e$
new nodes: c, g
a
d
b
e
f
c
g

FIGURE 2. Tree edit distance operations (listed in the box) and node mapping (by arrows).

The rest of the paper is organized as follows. In section II, as preliminaries, the problem in detecting changes in the relationship between nodes is presented by considering tree operations defined in related literature from the aspect of edges. Furthermore, to find a solution, in section III, we define edge operations by using the fundamental tree property that nodes in a tree have only one parent node. Moreover, to enable edge operations on the entire tree, including the root node, we introduce Edge Extended Tree (EET). By edge operations applied on EET, we define a dissimilarity measure between unordered trees; EED, which reflects a direct relationship modification between nodes in the trees. In section IV, change in predecessor nodes is observed as a consequence of edge edit operations. Such changes are described by introduced inheritance edit operations. Furthermore, TID is defined as a dissimilarity measure between unordered trees that represent the inheritance difference between trees. In section V, dissimilarity algorithms defined in sections III and IV, are evaluated by detecting changes in the class hierarchy for subsequent and minor versions of two publicly available object-oriented programs. Moreover, to prove efficiency, results of EED are compared to the results of the TED algorithm based on the edit graph given in [12]. Furthermore, section VI, provides a brief overview of the existing literature. Finally, the last section is the conclusion.

II. PRELIMINARIES

Tree edit distance evolved from the string comparison [18]–[20]. In general, it is based on finding the lowest transformation cost from one tree to another. Tree transformation is a sequence of elementary edit operations on nodes and edges such as *add*, *delete* and *substitute*. The cost is assigned to each edit operation. Consequently, the total cost of transformation from one tree to another is the sum of the costs of all edit operations in the sequence. As there may exist several ways to transform a tree into another, resulting in different total costs, edit distance is defined as the lowest transformation cost [1], [12], [13], [16]–[18], [18], [20], [21]. Tree edit distance algorithms correspond to the process of mapping, i.e., fitting similar parts of two trees, according to the assigned cost. An example of the mapping of the unordered trees is shown in Figure 2.

Edges define the relationship between nodes, therefore to detect changes in a tree, both edge edit operations and node edit operations could be used. Although edge operations in the Graph Edit Distance (GED) [13] are represented as the result of node operations, the presented idea to detect changes can be introduced more intuitively by using edge operations and will be used later to explain the indirect effect of edge edit operations on trees.

A. EDGE EDIT OPERATIONS

We consider three elementary edge operations: *add*, *delete*, and *substitute* edge. Following the notation in related literature for edit distance [12], [13], let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be trees, where V_1 and V_2 are sets of nodes, and E_1 and E_2 are sets of edges. Let e_1 be an edge in E_1, e_2 in E_2, and λ represent empty edge not contained in E_1 and E_2. *Add* edge operation is denoted by $\lambda \rightarrow e_2$, *delete* by $e_1 \rightarrow \lambda$, and *substitution* by $e_1 \rightarrow e_2$.

Let $u, v \in V$, edge $e \in E$ is then defined by a pair of nodes u and v such that $e = (u, v)$. In the next three cases, edge operations and their relation to node operations can be recognized:

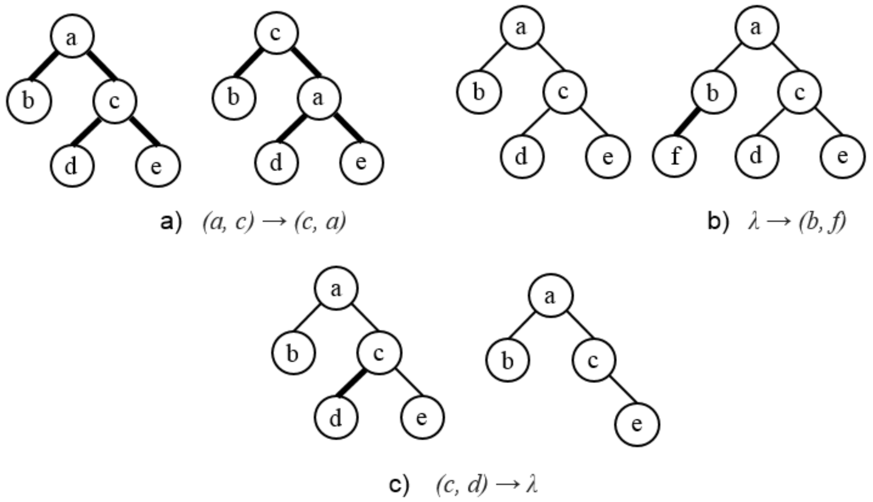


FIGURE 3. Edge edit operations: a) ***substitute*** b) ***add*** c) ***delete***.

1) Edge is substituted: $e_1 \to e_2$
   $e_1 = (u_1, v_1) \in E_1, e_2 = (u_2, v_2) \in E_2$ implies $u_1 \to u_2$ and $v_1 \to v_2$
   meaning node u_1 is substituted by u_2 and node v_1 is substituted by v_2
2) Edge is added: $\lambda \to e_2$
   $e_2 = (u_2, v_2) \in E_2$ implies $\nexists (u_1, v_1) \in E_1$ where $u_1 \to u_2$ and $v_1 \to v_2$
   meaning nodes u_2 and/or v_2 are added
3) Edge is deleted: $e_1 \to \lambda$
   $e_1 = (u_1, v_1) \in E_1$ implies $\nexists (u_2, v_2) \in E_2$ where $u_1 \to u_2$ and $v_1 \to v_2$
   meaning nodes u_1 and/or v_1 are deleted

These three cases are illustrated by Figure 3 in which the original tree could be transformed into one of three other trees, depending on if one edge is substituted (a), added (b), or deleted (c). Edge involved in the operation is marked with the bolded connecting line. It should be noted that, as the operation in Figure 3 a) is performed on non-leaf nodes, and both nodes are substituted, it implies additional substitute operations $(a, b) \to (c, b)$, $(c, e) \to (a, e)$, and $(c, d) \to (a, d)$.

B. RELATIONSHIP BETWEEN NODES

Our main motivation is to detect modification in the relationship between nodes from the aspect of inheritance change. As aforementioned, edge operations are the result of nodes operations. In Figure 2, node b is substituted by node d, and node c by node b, consequently, edge (a, b) is substituted by (a, d), and edge (a, c) by (a, b). However, the edge between nodes a and b exists in both trees. The relationship between nodes a and b is preserved, making node b and edge (a, b) substitution unnecessary. As TED performs operations on preserved edges between the trees, it is not suitable for observing direct relationship modification and, therefore, inheritance changes. In order to detect edge modifications and, based on this, changes in inheritance, the edge operations, and the new dissimilarity measure are presented in Section III.

III. EDGE EDIT DISTANCE

Instead of finding similar parts of a tree, in the case of the inheritance tree [3], [22], the relevant information is the change in the relationship between nodes. As every child node in a tree has only one parent, we can observe a child node and the corresponding edge to the parent as a single operation unit. Therefore, we define tree with additional empty node and operations, which reflects modification on the relationship between nodes.

A. EDGE EXTENDED TREE

By considering the node and its edge to its parent as a single unit, edge edit operations could be divided into three cases. *Add* operation consist of adding a new node that by a new edge connects to an existing or in a previous operation added parent node. *Delete* operation removes the node and its edge that connects it to its parent. Instead of *substitute* operation, the term *move* would be used, and the operation is the change of only one incident node, the parent node.

In this way, if the parent function is defined, then every edge in a tree could be denoted as pair $(parent(u), u)$ where $parent(u)$ and u are nodes in the tree. However, because the root node does not contain any edge to the parent, in order to define the parent function for the complete node-set, dummy or empty node $\varepsilon \notin V$ is introduced in such way that edge $e = (\varepsilon, r)$ exists, where r is the root node.

Definition 1: Edge Extended Tree (EET) is an unordered tree $X = (V \cup \{\varepsilon\}, E)$ where $\varepsilon \notin V$ is an empty node connected only to the root node r with an edge $e = (\varepsilon, r) \in E$.

Remark 1: Let n be a number of nodes in V, then the tree X contains n edges. This claim is a direct consequence of the fact that unordered labeled tree $T = (V, E)$ with n

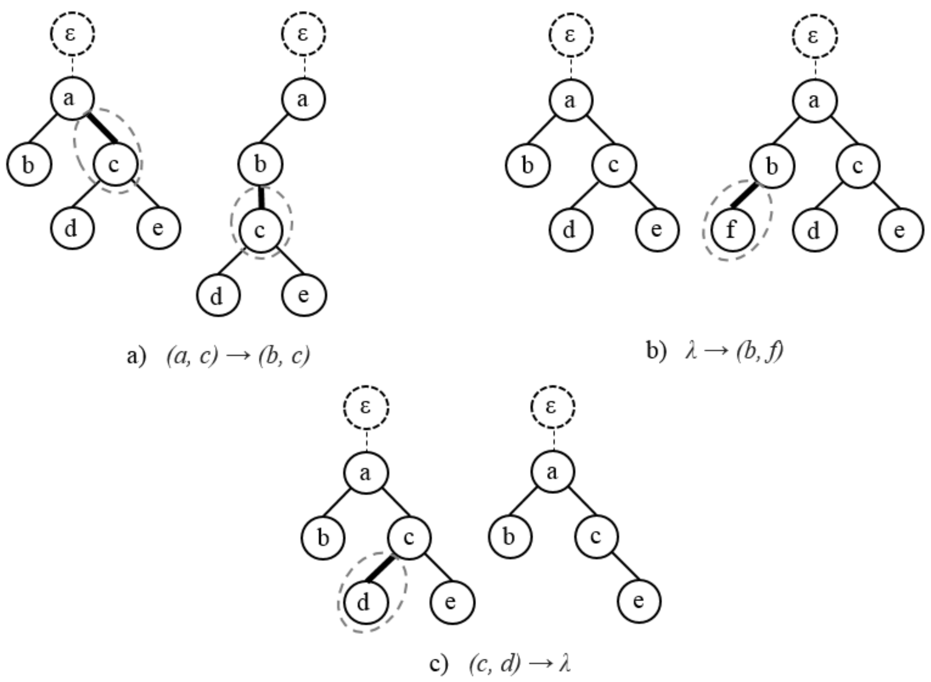

ε
a
b
c
d
e
f
a) $(a, c) \rightarrow (b, c)$
b) $\lambda \rightarrow (b, f)$
c) $(c, d) \rightarrow \lambda$

FIGURE 4. Edge edit operations on EET.

nodes contains $n - 1$ edges, and adding an empty node ε and corresponding edge (ε, r), leads to the tree $X = (V \cup \{\varepsilon\}, E)$ containing n edges.

Formally, the parent function for the edge extended tree $X = (V \cup \{\varepsilon\}, E)$ could be described as:

$$p : V \rightarrow V \cup \{\varepsilon\} \text{ such that } p(v) = u \text{ iff} \\ \exists u \in V \cup \{\varepsilon\} \mid (u, v) \in E$$

By using the parent function, we can define edge edit operations on the edge extended tree:

Definition 2: Let $X_1 = (V_1 \cup \{\varepsilon\}, E_1)$ and $X_2 = (V_2 \cup \{\varepsilon\}, E_2)$ are EET's with parent functions p_1 in X_1 and p_2 in X_2, edit edge operations are:

1) Edge is moved: $e_1 \rightarrow e_2$
Only one of the nodes incident to edges $e_1 = (p_1(v), v)$ and $e_2 = (p_2(v), v)$ is changed, i.e. the parent node.
This operation can be reduced to the following situation: $v \in V_1 \cap V_2$ and $p_1(v) \neq p_2(v)$
2) Edge is added: $\lambda \rightarrow e_2$
As e_2 could be defined as $(p_2(v), v) \in E_2$, this implies $\nexists (p_1(v), v) \in E_1$
Therefore, this operation is reduced to $v \notin V_1 \wedge v \in V_2$, i.e., $v \in V_2 \setminus V_1$
3) Edge is deleted: $e_1 \rightarrow \lambda$
Similar to the previous operation, it is reduced to $v \notin V_2 \wedge v \in V_1$, i.e., $v \in V_1 \setminus V_2$

Edge operations are shown in Figure 4, where edges involved in operations are bolded. Furthermore, child nodes included in the edge operation are marked with the dashed surrounding ellipse, together with the edge to the parent node - $(parent(u), u)$. Edge *move* operation from the edge (a, c) to edge (b, c) is shown in Figure 4 a). *Move* operation performs the move of the entire subtree rooted at the node c, from the position where the previous parent node is a to the position where node b is the new parent. *Add* edge (b, f) operation is shown in Figure 4 b). Adding edge is the result of adding node f to the tree, such that the parent of the new node is node a. *Delete* edge (c, d) operation is shown in Figure 4 c). *Delete* edge is the result of the node *delete* operation from the tree. *Add* and *delete* edge operations are shown only on leaf nodes, as non-leaf *add* and *delete* edge operations are followed by at least one more edge edit operation.

When a non-leaf node is added, then at least one child of a node, which becomes a parent to a new node, becomes a child of a newly added node. In Figure 5 a), node f is added as a non-leaf node to parent node a, with edge operation $\lambda \rightarrow (a, f)$. Node b as the previous child of the node a is consequently moved to be the child of the added node f, with edge operation $(a, b) \rightarrow (f, b)$. However, edge (a, c) to another child c of node a is preserved, therefore only one child node is moved. On the other hand, the deletion of the non-leaf node requires that all children of the deleted node change their parent to the parent of the deleted node. Figure 5 b) shows the deletion of non-leaf node c, with edge operation $(a, c) \rightarrow \lambda$, followed by changing the parent of node d to node a, with edge operation $(c, d) \rightarrow (a, d)$. Identical operation is performed on the node e, with edge operation $(c, e) \rightarrow (a, e)$. There is a specific case when the root node is deleted. In this case, by an arbitrary procedure, one of the child nodes of the previous root node is promoted to be the

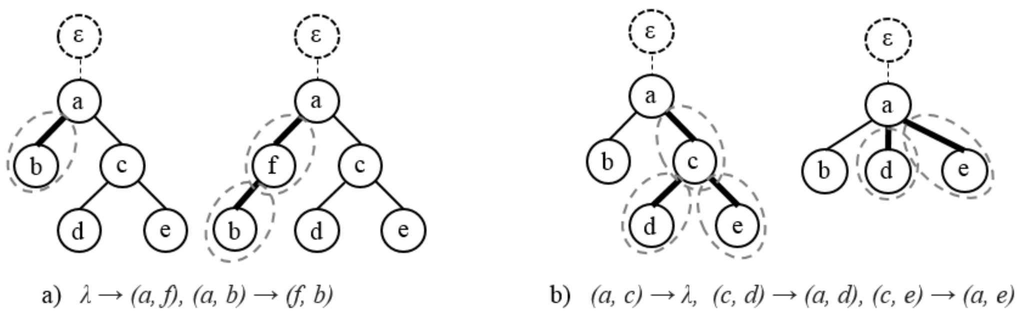

a) λ → (a, f), (a, b) → (f, b)
b) (a, c) → λ, (c, d) → (a, d), (c, e) → (a, e)

FIGURE 5. Non-leaf node *add* (a) and *delete* (b) edge operations.

new root node, and other siblings of this node are moved to be the children of the new root. It is mandatory because only one edge can exist from the empty node ε to the root node. Otherwise, there are a smaller number of edges than nodes, resulting in the fact that a tree is not EET. Furthermore, the root resolving procedure is arbitrary depending on the application, e.g., it could be based on specific node properties.

B. SET OF EDIT OPERATIONS

Generally, the transformation of tree T_1 to another tree T_2 can be observed by a sequence $s_1, s_2, \dots, s_n$ of edit operations on nodes or edges [1], [12], [16]–[18], [20]. For example, sequence of edge edit operations to transform a source tree in Figure 2 on the left side to the target tree on the right side is: $(a, b) \rightarrow (a, d)$, $(a, c) \rightarrow (a, b)$, $(b, d) \rightarrow (d, f)$, $(b, e) \rightarrow (d, e)$, $\lambda \rightarrow (e, c)$, $\lambda \rightarrow (e, g)$. Similar is for edge extended trees X_1 and X_2, where edge edit operations defined by Definition 2 are *add*, *delete* and *move*. Sequence of EET edge edit operations to transform tree in Figure 2 is: $(b, d) \rightarrow (a, d)$, $(b, e) \rightarrow (d, e)$, $\lambda \rightarrow (d, f)$, $(a, c) \rightarrow (e, c)$, $\lambda \rightarrow (e, g)$. There is a smaller number of EED than TED operations, since EED consider edge modifications based on unchanged node labels, whereas TED maps similar tree parts by nodes label transformation, i.e. nodes substitution.

The sequence of edit operations influences the intermediate results. If we apply edge edit operations where *add* edge operation is followed by a *move* and then by the *delete* operation, while consecutive *add* and *delete* operations are executed from leaf nodes upwards, each step in the sequence produces a tree. Otherwise, the intermediate result depending on the involved edge operations can be tree forest. TED fulfills these conditions in [7], [12], [20] since edit operations are performed only on leaf nodes. On the other hand, such conditions do not produce valid EET in each possible step, e.g., adding the new root node before moving or deleting the old root node results in two edges from the empty node. Similar to the [12] the sequence of edge edit operations that transforms a tree can be written as an ordered relation $R \subseteq (E_1 \cup \{\lambda\}) \times (E_2 \cup \{\lambda\})$, where an edge edit operation is represented as a pair of edges (e_1, e_2), such that $e_1 \in E_1 \cup \{\lambda\}$, $e_2 \in E_2 \cup \{\lambda\}$. However, in the rest of the paper, we are considering only the final result, i.e., we are observing operations between two trees, source, and target tree, without intermediate results. Therefore, instead of the edit sequence, where operations order is essential, we are using a set of edge edit operations S_e.

Considering that there may be several possible edit sets that perform the same transformation, let $W(X_1, X_2)$ denotes the set $\{S_{e1}, \dots, S_{en}\}$ of all sets of edit operations to transform X_1 to X_2, possibly containing superfluous operations, e.g., additional subsequent *delete* and *add* edge operations or *move* edge operation. Furthermore, *move* edge operation $e_1 \rightarrow e_2$ can be described as a *delete* edge operation $e_1 \rightarrow \lambda$, followed by an *add* edge operation $\lambda \rightarrow e_2$. These additional operations for TED [12] and GED [13] generally increase the additional cost to the tree transformation. However, we are asking the question of how relationships, i.e., edges between nodes in the source and target tree are changed. Accordingly, we are calculating the cost for the minimum number of required edge edit operations to transform a tree, which implicitly excludes such operations.

C. EDIT SET COST

To determine the cost for the set of edit operations S_e, for each edge edit operation, a cost by function $\gamma : E_1 \cup E_2 \cup \{\lambda\} \times E_1 \cup E_2 \cup \{\lambda\} \rightarrow \mathbb{R}$ is assigned. The total cost to transform from tree X_1 to X_2 by applying edge edit operations from the set of edit operations S_e is equal to:

$$c(S_e) = \sum_{s_i \in S_e} \gamma(s_i) \tag{1}$$

Edge dissimilarity measure between two trees is edge edit distance, formally defined as:

Definition 3: Edge Edit Distance between trees X_1 and X_2 is the total cost of the set of edit operations S_e that contains the minimum number of edge edit operations to transform X_1 to X_2:

$$d_e(X_1, X_2) = c(S_e) \,|\, S_e \in W(X_1, X_2) \quad \text{and} \\ |S_e| \leq |S'_e| \; \forall S'_e \in W(X_1, X_2)$$

If the cost for all edge edit operations is a constant number equal to 1, the above formulation is equal to finding the edit set with the minimum cost, although generally, the cost of the minimum number of operations could be greater than the minimum cost. Note that even with the same cost of operations, this formulation is not equal to the definition of TED [12] or GED [13] because of the different formulation of edit operations, primarily because the *move* edge operation is different from substitution. Furthermore, e.g., if the cost of a *move* operation is greater than the sum of the cost of *add* and *delete* operations, the set of the minimum number of edit operations is not changed compared to the equal cost of operations.

Since, for *move* operation, a node is changing the parent, the cost is equal to the cost of the parent change. For *add* and *delete* operations, the cost is equal to the cost of node adding to a tree or deleting from a tree, respectively. Furthermore, due to Definition 2 of edit operations, as we consider only the source and the target tree, finding minimal transformation edit set could be determined by nodes, and cost function could be defined as $\gamma : V_1 \cup V_2 \to \mathbb{R}$.

$$\gamma(v) = \begin{cases} 0, & v \in V_1 \cap V_2 \wedge p_1(v) = p_2(v) \\ m, & v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v) \\ a, & v \in V_2 \setminus V_1 \\ d & v \in V_1 \setminus V_2 \end{cases} \tag{2}$$

Here m, a and d are the cost of the *move*, *add* and *delete* edge operations. The costs of these operations could be a constant number, e.g., equal to 1, but it can be generalized to situations in which m, a, and d could be cost functions ($m(v)$, $a(v)$, $d(v)$) of a node v, e.g., based on the number of node properties or the distance from the root node.

By using cost function defined over a node, the edge edit distance between trees X_1 and X_2 is equal to:

$$d_e(X_1, X_2) = \sum_{v \in V_1 \cup V_2} \gamma(v) \tag{3}$$

Proof: Let the set of edge operations S_e consists of the following sets of operations: S_a, S_d, and S_m, where $(\lambda, e_2) \in S_a$, $(e_1, \lambda) \in S_d$, and $(e_1, e_2) \in S_m$. Without loss of generality, let the costs for *add*, *delete*, and *move* edge operations are equal to the constants a, d, and m. According to Definition 2, edge operations are reduced to nodes, and according to Definition 3, the following applies:

$$\begin{aligned} d_e(X_1, X_2) &= c(S_e) = \sum_{s_i \in S_e} \gamma(s_i) \\ &= \sum_{s \in S_a} a + \sum_{s \in S_d} d + \sum_{s \in S_m} m \\ &= \sum_{v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)} m + \sum_{v \in V_2 \setminus V_1} a + \sum_{v \in V_1 \setminus V_2} d \\ &= \sum_{v \in V_1 \cup V_2} \gamma(v) \end{aligned}$$

The number of operations is minimal since the number of edge operations is limited by the number of nodes, and the following applies $|V_1 \cup V_2| = |V_1 \setminus V_2| + |V_2 \setminus V_1| + |V_1 \cap V_2|$.

Furthermore, edge edit distance by using constants for m, a, and d can be formulated as:

$$d_e(X_1, X_2) = \sum_{v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)} m + a\,|V_2 \setminus V_1| + d\,|V_1 \setminus V_2|$$

Considering Definition 2 and Definition 3, we give Algorithm 1 to calculate edge edit distance. Note that constants for the cost of the edge edit operations could be replaced with functions $a(v)$, $d(v)$, and $m(v)$ or $\gamma(v)$.

Algorithm 1 Edge Edit Distance Algorithm

```
input  : trees $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$,
         parent functions $p_1 : V_1 \to V_1 \cup \{\varepsilon\}$, and
         $p_2 : V_2 \to V_2 \cup \{\varepsilon\}$, and cost of the edit
         operations $a$, $d$, and $m$
output: an edit set $S_e$ with the minimum number of
         operations and an edge edit distance $d_i$
1  $S_e \leftarrow \varnothing$, $d_i \leftarrow \varnothing$;
2  foreach node $v$ in $V_1$ do
3      if contains($V_2$, $v$) then
4          if $p_1(v) \neq p_2(v)$ then
5              $S_e \leftarrow S_e \cup \{(p_1(v), v) \to (p_2(v), v)\}$;
6              $d_i \leftarrow d_i + m$;
7          end
8      else
9          $S_e \leftarrow S_e \cup \{(p_1(v), v) \to \lambda\}$;
10         $d_i \leftarrow d_i + d$;
11     end
12 end
13 foreach node $v$ in $V_2$ do
14     if not contains($V_1$, $v$) then
15         $S_e \leftarrow S_e \cup \{\lambda \to (p_2(v), v)\}$;
16         $d_i \leftarrow d_i + a$;
17     end
18 end
```

In order to determine the time complexity of the algorithm, let the n_1 be the number of edges in X_1, and n_2 be the number of edges in X_2. Algorithm time complexity is $O(n_1+n_2) * O(\texttt{contains})$, since algorithm iterates through a set of nodes V_1 and V_2, and the size of node sets corresponds to the size of E_1 and E_2, equal to n_1 and n_2. The cost of the `contains` function could be $O(1)$ if an appropriate hashing is used.

Regarding the number of operations, the upper bound for edge edit operations is in the case when all edges from E_1 are deleted, and all edges from E_2 are added. On the other hand, when all edges from E_1 are moved in E_2, as a consequence that nodes between X_1 and X_2 are matched ($V_1 = V_2$), and all nodes from X_1 change parents in X_2, the number of edge operations is equal to $|V_1 \cap V_2|$. Since there are an equal number of edges and nodes, the maximum number of edge

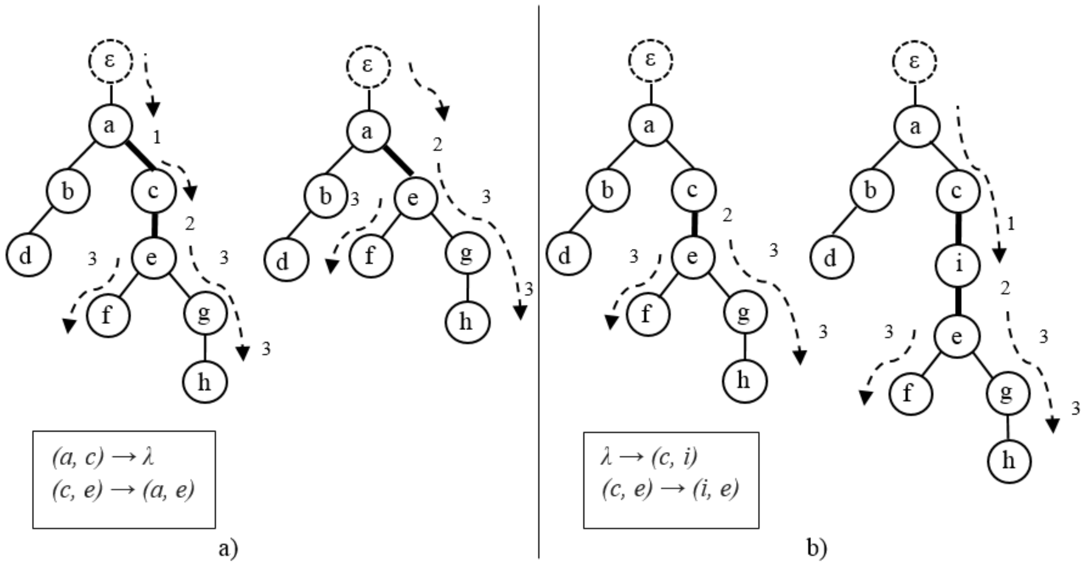


FIGURE 6. The impact of the *move* edge edit operation on the inheritance relationship induced by: a) deleted edge and b) added edge.

edit operations, i.e., the upper bound for edge edit operations n_{max} can be stated as:

$$n_{max} = |E_1| + |E_2| = |V_1| + |V_2| > |V_1 \cap V_2|$$

IV. TREE INHERITANCE DISTANCE

Edge edit operations are suitable to describe changes in the direct relationship between nodes. However, if we observe a tree as an inheritance tree [22], where the node is in the inheritance relationship to predecessor nodes, the node is affected by any changes in its predecessor nodes. Edge edit distance defined by appropriate cost function can describe changes in predecessor nodes only for nodes directly involved in edge edit operations, such as added, deleted, and moved nodes. On the other hand, indirect effects of edge edit operations on descendant nodes cannot be described by edge edit distance. Those changes are the consequence of edge edit operations. In the following subsections, we will describe the direct and indirect effects of editing operations on the inheritance relationship between nodes, and then define the indirect edit operation. Afterward, we introduce the inheritance operations, the cost function for inheritance operations, and tree inheritance distance.

A. TREE EDITING IMPACT ON THE INHERITANCE

A suitable example to explain the effects of edge edit operations on the inheritance relationship between nodes should include *add*, *delete*, and *move* edge edit operations. In Figure 6, for example, *delete* (Figure 6 a) and *add* (Figure 6 b) edge edit operations, performed on non-leaf nodes, are tagged with number 1 and induce *move* edge operation, tagged with number 2. From the inheritance aspect for the added node, all nodes on the path to the empty node are added to the inheritance relationship, i.e., the newly added node can inherit ancestors' nodes properties. On the other side, when a node is deleted, inheritance relationship to nodes on the path to the empty node is deleted, i.e., the deleted node does not any longer inherit prospective ancestors' properties. Furthermore, the inheritance relationship to ancestor nodes is also changed for the moved node, involving at least the parent node changed by an edge edit operation. In Figure 6 a), node c is deleted, and as a consequence, the edge (a, c) is also deleted, inducing *move* edge operation $(c, e) \rightarrow (a, e)$. Path (ε, a, c) is removed from the tree, meaning that node c loses inheritance relationship to the node a. Moreover, path (ε, a, c, e) is deleted, and path (ε, a, e) is added to the tree, meaning that node e loses inheritance relationship to the node c. Figure 6 b) shows a similar case where the edge (c, i) and the node i are added, inducing *move* edge operation $(c, e) \rightarrow (i, e)$. Node i obtain inheritance relationship to the nodes a and c, and node e obtains inheritance relationship to the added node i.

However, if the inheritance relationship is changed for nodes directly involved in edit operations, it is indirectly changed for their descendants, as paths from descendants to the empty node are changed. In Figure 6, descendant nodes of a moved node are affected by *move* operations. These indirect effects of edge edit operations are tagged with number 3. As we already observed in Figure 6 a), node c is removed from the path to the empty node, i.e., removed from the inheritance relationship for node e. Moreover, node c is also removed from the inheritance relationship for node e descendants f, g and h. Similar can be observed in Figure 6 b), where node i is added to the inheritance relationship for moved node e but also for its descendants f, g, and h.

Similar would happen if the *move* edge operation is not induced by other edge edit operations. If the tree in Figure 6 a) would be modified only by the *move* edge operation $(c, e) \rightarrow (d, e)$, node e would lose inheritance to node c and would obtain inheritance to nodes b and d. Furthermore, the same change in the inheritance relationship would be applied to the descendants of node e.

B. DETECTING INHERITANCE CHANGES

The previous section describes the change of the inheritance relationship as the consequence of edge edit operations. In order to determine changes in the inheritance relationship, it is necessary to compare nodes on the paths to the empty node from node involved in edge edit operations and its descendants.

Definition 4: Let $path(u, v)$ be the path from the node u to node v in the tree $X(V \cup \{\varepsilon\}, E)$, where $u \in V \cup \{\varepsilon\}$ and $v \in V$. Let $P(u, v)$, shortly $P_u(v)$ represents the set of nodes on the $path(u, v)$ without node v. $P_\varepsilon(v)$ then contains at least empty node ε for all nodes in V. Furthermore, if $v \notin V$ then $P_\varepsilon(v)$ is an empty set ($\emptyset$). Formally:

$$P_\varepsilon(v) = \begin{cases} \{u \in V \cup \{\varepsilon\} \mid u \in path(\varepsilon, v) \wedge u \neq v\}, & v \in V \\ \emptyset, & v \notin V \end{cases}$$

Node set $P_\varepsilon(v)$ contains a set of predecessor nodes for node v, therefore condition $u \neq v$ is required, as node v cannot precede itself. Note that in the rest of the paper for the set of nodes to the empty node $P_\varepsilon(v)$, the term predecessor node set will be used. If node sets $P_{\varepsilon1}(v)$ and $P_{\varepsilon2}(v)$ in edge extended trees X_1 and X_2 are different for node v from $V_1 \cup V_2$, then there is a change in inheritance relationship for node v. However, if the node sets $P_{\varepsilon1}(v)$ and $P_{\varepsilon2}(v)$ are equal, it does not indicate that the inheritance relationship is not changed for node v. In Figure 6, *move* edge operation implies a change of the parent node, and consequently different sets $P_{\varepsilon1}$ and $P_{\varepsilon2}$. On the other hand, in Figure 7, as the result of *move* edge edit operations, nodes a and b switched positions, which for node c and consequently node d caused different paths in two trees, although the node sets $P_{\varepsilon1}(v)$ and $P_{\varepsilon2}(v)$ are equal. There is a change in the inheritance relationship for node c and d in a form of change in the position between nodes on the path to the empty node. Change in position between nodes on the path to the empty node can result in a change of inherited properties, e.g., a node can lose inherited properties.

The change of position or order of predecessor nodes can be observed as the change in the distance between the nodes, i.e., the number of edges on the path between nodes. In Figure 7, the distance between nodes a and d in the source tree is equal to 3, and in the target tree to 2. Furthermore, the distance between nodes b and d is equal to 2 and 3, in the source and the target tree, respectively. Similar can be observed for node c, where predecessor node sets remain equal but the distance to nodes a and b changes.

The distance between nodes on the path to the empty node is equal to the difference of node depths, but to simplify distance comparison between nodes, we define the node distance

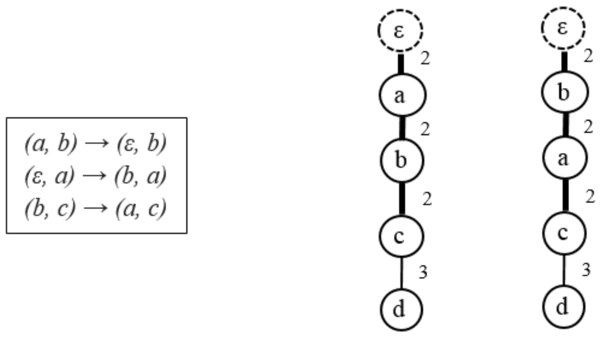


FIGURE 7. Move edge operations with equal sets of nodes $P_\varepsilon(c)$ and $P_\varepsilon(d)$.

as the cardinality of the set of nodes on the path from the node u to the node v. Formally, we define a distance function as:

$$d : V \times V \rightarrow \mathbb{R}, \quad \text{such that } d(u, v) = |P_u(v)|$$

In the example illustrated by Figure 7, we can write $d_1(a, c) \neq d_2(a, c)$ because $d_1(a, c) = 2$ and $d_2(a, c) = 1$, and $d_1(b, c) \neq d_2(b, c)$. Note that the distance function defined in this way reflects a structural change in the tree, but the distance function can also reflect modified properties of nodes or edges. We can conclude that the inheritance relationship for the node has changed if the predecessor node sets are different or if there is a node on the path that has changed the distance relative to the node.

Definition 5: Let $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$ be edge extended trees. Let $P_{\varepsilon1}(v)$ in X_1 and $P_{\varepsilon2}(v)$ in X_2 for $v \in V_1 \cup V_2$ be predecessor node sets. Let $d_1(u, v)$ be nodes distance function in X_1 and $d_2(u, v)$ in X_2, where $u \in P_{\varepsilon1}(v) \cup P_{\varepsilon2}(v)$. The inheritance relationship is changed, i.e., node v is edited, if the following applies:

$$P_{\varepsilon1}(v) \neq P_{\varepsilon2}(v) \text{ or } \exists u \in P_{\varepsilon1}(v) \cap P_{\varepsilon2}(v) \text{ such that } d_1(u, v) \neq d_2(u, v)$$

C. DIRECT AND INDIRECT EDIT OPERATIONS

Edge edit operations described by Definition 2 have direct and indirect effects on inheritance editing. To define indirect effects of edge edit operations on inheritance editing, first, we will determine the direct effects of edge edit operations on the changes in node sets $P_\varepsilon(v)$, and distance between nodes on the path to the empty node.

Let X_1 and X_2 be edge extended trees, $P_{\varepsilon1}(v)$ in X_1, and $P_{\varepsilon2}(v)$ in X_2 be set of predecessor nodes, for v from $V_1 \cup V_2$. Edge *add* operation implies empty set $P_{\varepsilon1}(v)$ and non-empty set $P_{\varepsilon2}(v)$ because the path for the added node v in X_1 is not defined, as the added node is not part of X_1. Similarly, for *delete* edge operation, set $P_{\varepsilon1}(v)$ is not empty, and $P_{\varepsilon2}(v)$ is empty, as the path to the deleted node v is not defined in X_2. Furthermore, *move* edge edit operation involves a change of the node v parent, resulting in different sets $P_{\varepsilon1}(v)$ and $P_{\varepsilon2}(v)$, or change in the distance between nodes in these sets.

By Definition 2 and Definition 5, for edge edit operations we can determine the following:

1) *add*: $\lambda \rightarrow e_2$ corresponds to $v \in V_2 \setminus V_1$, implies $P_{\varepsilon 1}(v) = \varnothing \wedge P_{\varepsilon 2}(v) \neq \varnothing$
2) *delete*: $e_1 \rightarrow \lambda$ corresponds to $v \in V_1 \setminus V_2$, implies $P_{\varepsilon 1}(v) \neq \varnothing \wedge P_{\varepsilon 2}(v) = \varnothing$
3) *move*: $e_1 \rightarrow e_2$ corresponds to $v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)$, implies $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $\exists u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$ such that $d_1(u, v) \neq d_2(u, v)$

On the other hand, as an indirect result of edge edit operations, by Definition 5, the path to the empty node is changed for descendant nodes. Consequently, descendant nodes are involved in indirect edit operations.

Definition 6: Node $v \in V_1 \cap V_2$ is indirectly edited if $p_1(v) = p_2(v)$ and $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $\exists u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$ such that $d_1(u, v) \neq d_2(u, v)$. Edge $e = (p(v), v)$ is indirectly edited if node v is indirectly edited, as a consequence of a change in inheritance relationship for the node v.

Add, *delete*, and *move* edge edit operations induce changed paths to the empty node. Given the structural changes in the tree, indirect operations are an indirect result of the *move* edge edit operations, and possibly an indirect result of the *add* and *delete* edge edit operations, in order 1-2-3 or 2-3, where the numbers represent operation type, as shown in Figure 6 and Figure 7 (1 – *add* or *delete*, 2 – *move*, 3 – *indirect*).

D. INHERITANCE EDIT OPERATIONS

In the previous subsections, inheritance change is detected by changes in paths to the empty node. These changes were observed only by detecting whether the inheritance relationship for a node is changed or not, i.e., whether the node is edited directly or indirectly. However, to compare inheritance changes between two trees, it is necessary to determine how inheritance has changed for each node. For a single node, one or more nodes can be added, deleted, or can change their position or other properties on the path to the empty node. Each such modification on the path to the empty node is a single inheritance operation. Similar to the edge edit operations, we define, *add*, *delete*, and *move* inheritance operations:

Definition 7: Let $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$ be edge extended trees, $P_{\varepsilon 1}(v)$ in X_1 and $P_{\varepsilon 2}(v)$ in X_2, for $v \in V_1 \cup V_2$ be sets of predecessor nodes, $d_1(u, v)$ be node distance function in X_1 and $d_2(u, v)$ be node distance function in X_2, where $u \in P_{\varepsilon 1}(v) \cup P_{\varepsilon 2}(v)$.

Inheritance edit operations on node $v \in V_1 \cup V_2$ are:

1) *add*, $u \in P_{\varepsilon 2}(v) \setminus P_{\varepsilon 1}(v)$,
node u is added to the inheritance relationship for node v
2) *delete*, $u \in P_{\varepsilon 1}(v) \setminus P_{\varepsilon 2}(v)$,
node u is deleted from the inheritance relationship for node v
3) *move*, $u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v) \wedge d_1(u, v) \neq d_2(u, v)$,
nodes u and v are moved relative to one another

Remark 2: We have already shown how edge edit operations implicitly result in inheritance changes for the involved node. Consequently, inheritance operations occurred by such changes are direct inheritance operations. On the other hand, inheritance operations occurred on nodes indirectly are indirect inheritance operations.

Single edge edit operation could result in multiple inheritance operations. In Figure 6, *move* edge operation $(c, e) \rightarrow (a, e)$ results in four *delete* inheritance operations because node c is removed from the path for node e and its descendants f, g, and h. On the other hand, *move* edge edit operations in Figure 7 resulted in overall six inheritance operations. Node b is added to the inheritance relationship for node a. Analogously, node a is removed from the inheritance relationship for node b. Consequently, nodes a and b are relatively moved to nodes c and d.

In previous figures, we intuitively observed inheritance operations as sets of added, deleted, and moved node sets for the edited node, which corresponds to *add*, *delete*, and *move* inheritance operations. By using Definition 7, we determine these sets as:

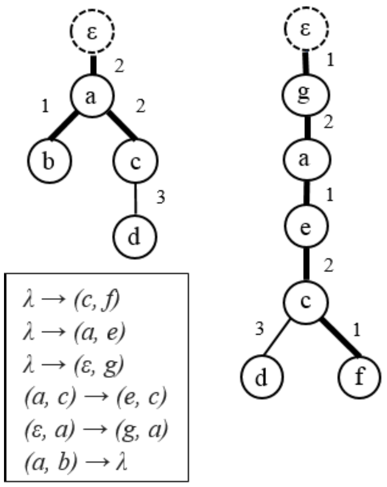
Definition 8: Let $P_{\varepsilon 1}(v)$ be set of predecessor nodes for v in X_1, $P_{\varepsilon 2}(v)$ in X_2, and $d_1(u, v)$ and $d_2(u, v)$ be distance functions in X_1 and X_2, where $u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$. Then, set of added nodes $V_a(v)$, deleted nodes $V_d(v)$, and moved nodes $V_m(v)$ on the path from empty node ε to the node v, for $v \in V_1 \cup V_2$ are defined as:

$$V_a(v) = P_{\varepsilon 2}(v) \setminus P_{\varepsilon 1}(v)$$
$$V_d(v) = P_{\varepsilon 1}(v) \setminus P_{\varepsilon 2}(v)$$
$$V_m(v) = P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v) \mid d_1(u, v) \neq d_2(u, v)$$

Inheritance operations can be conveniently shown in tabular form, where columns represent node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$ for every node v in the table rows, whether the node is edited or not. In Figure 8, for example, inheritance edit operations occur on all nodes. Consequently, for each node v in X_1 and X_2, at least one of the node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$ is not empty. From Table 1, we can detect that there are indirect inheritance operations. Node c and d have equal node sets because direct inheritance changes on node c are propagated to the node d.

From node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$, we can observe the connection between edge edit operations and inheritance operations. For some nodes v, sets $V_a(v)$ and $V_d(v)$ contain empty node ε, which means that these nodes are added or deleted. In Table 1, set $V_a(g)$ contains only node ε because node g is added as the new root node. On the contrary, nodes v without empty node ε in node sets $V_a(v)$ and $V_d(v)$, are edited by *move* edge operations. By Definition 2 and Definition 7, we can observe three cases, how inheritance edit operations affect $V_a(v)$, $V_d(v)$, and $V_m(v)$ node sets:

1) move, $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $d_1(u, v) \neq d_2(u, v)$ implies $V_a(v) \neq \varnothing$ or $V_d(v) \neq \varnothing$ or $V_m(v) \neq \varnothing$
2) add, $P_{\varepsilon 1}(v) = \varnothing \wedge P_{\varepsilon 2}(v) \neq \varnothing$ implies $V_a(v) \neq \varnothing \wedge V_d(v) = \varnothing \wedge V_m(v) = \varnothing$
3) delete, $P_{\varepsilon 1}(v) \neq \varnothing \wedge P_{\varepsilon 2}(v) = \varnothing$ implies $V_a(v) = \varnothing \wedge V_d(v) \neq \varnothing \wedge V_m(v) = \varnothing$



$\lambda \rightarrow (c, f)$
$\lambda \rightarrow (a, e)$
$\lambda \rightarrow (\varepsilon, g)$
$(a, c) \rightarrow (e, c)$
$(\varepsilon, a) \rightarrow (g, a)$
$(a, b) \rightarrow \lambda$

FIGURE 8. Inheritance edit operations example.

TABLE 1. Inheritance edit operations in tabular form.

$v \in V_1 \cup V_2$	$V_a(v)$	$V_d(v)$	$V_m(v)$
a	$\{g\}$	$\varnothing$	$\varnothing$
b	$\varnothing$	$\{\varepsilon, a\}$	$\varnothing$
c	$\{g, e\}$	$\varnothing$	$\{a\}$
d	$\{g, e\}$	$\varnothing$	$\{a\}$
e	$\{\varepsilon, g, a\}$	$\varnothing$	$\varnothing$
f	$\{\varepsilon, g, a, e, c\}$	$\varnothing$	$\varnothing$
g	$\{\varepsilon\}$	$\varnothing$	$\varnothing$

These cases are similar to the observation of edge edit operations impact on the predecessor node set P_ε, in the previous section. The main difference is in the first case, where both direct and indirect edit operations are included, i.e., cases where $p_1(v) \neq p_2(v)$ or $p_1(v) = p_2(v)$.

E. INHERITANCE COST

Inheritance operations describe inheritance changes between trees. In order to define inheritance dissimilarity measure, it is necessary to determine the cost of inheritance changes between trees, which is equal to the total cost of inheritance operations. In the previous subsection, we have determined how inheritance edit operation corresponds to nodes contained in sets $V_a(v)$, $V_d(v)$, and $V_m(v)$. Let $V_i(v)$ contain all nodes involved in the inheritance operation for node v such that $V_i(v) = V_a(v) \cup V_d(v) \cup V_m(v)$. Accordingly, to determine the inheritance cost of nodes contained in $V_i(v)$ on node v, we define cost function φ for the inheritance edit operation.

Definition 9: Let $\varphi : (V_1 \cup V_2) \times (V_1 \cup V_2) \rightarrow \mathbb{R}$ be the cost function of the inheritance operation of node $u \in V_i(v)$ on node $v \in V_1 \cup V_2$.

In the case of *add* operation, both node u from $V_a(v)$ and node v are contained in V_2. Similar is for the *delete* operation, where both node u from $V_d(v)$ and node v are in V_1. For *move* operation, node u from $V_m(v)$, and v are in $V_1 \cap V_2$. Furthermore, such arguments are associated with the symmetry property of the function φ, which ensures that adding and deleting the same node on the path to the empty node is complementary. Similar applies to complementary *move* inheritance operations, involving the same nodes u and v, where nodes repeatedly exchange their relative positions, resulting in the preserved distance.

Function φ can be as simple as $\varphi(v_a, v_b) = c$ for all v_a, $v_b \in V_1 \cup V_2$, where c is a constant number. Furthermore, function φ can be dependent on the distance between involved nodes: $\varphi(v_a, v_b) = c * \frac{d(v_a,v_b)}{k}$, or alternatively, $\varphi(v_a, v_b) = c * e^{-\frac{d(v_a,v_b)}{k}}$, where $d(v_a, v_b)$ is the distance function between nodes v_a and v_b, and k is the constant to control inheritance range and strength of the edit operation. Namely, in the context of edge operations, nodes closer to the edge operation are more influenced by the inheritance change than leaf nodes.

Inheritance editing of a single node has previously been shown by the set of nodes $V_i(v)$, which corresponds to the set of inheritance edit operations performed on the node v. Consequently, the inheritance cost for a single node v is defined by the sum of inheritance operations costs on the node v, i.e., by the sum calculated by using a function φ over nodes from set $V_i(v)$.

Definition 10: Let $V_i(v) = V_a(v) \cup V_d(v) \cup V_m(v)$ are node sets containing nodes involved in inheritance operations on node v. Inheritance cost function for node v is $\delta : V_1 \cup V_2 \rightarrow \mathbb{R}$, defined as:

$$\delta(v) = \sum_{u_a \in V_a(v)} \varphi(u_a, v) + \sum_{u_d \in V_d(v)} \varphi(u_d, v) + \sum_{u_m \in V_m(v)} \varphi(u_m, v)$$
$$= \sum_{u \in V_i(v)} \varphi(u, v)$$

where φ is an inheritance operation cost function $(V_1 \cup V_2) \times (V_1 \cup V_2) \rightarrow \mathbb{R}$.

If the function φ is constant for all nodes, then the inheritance cost for node f in Figure 8 is equal to 5, because set $V_a(f)$ contains five nodes ε, g, a, e, and c. If the source and target trees exchange places, then set $V_d(f)$ contains nodes ε, g, a, e, and c, while set $V_a(f)$ is empty, resulting in inheritance cost that is again equal to 5. We can observe how function δ inherits symmetry property from function φ because adding and deleting node are complementary operations regarding inheritance operations. The same applies to moved and, indirectly edited nodes.

The inheritance cost between trees is determined by the total cost of inheritance editing of all nodes. Let S_i be the set of edited nodes between trees X_1 and X_2, where $S_i = \{v_1, \ldots, v_i, \ldots, v_n\}$, such that according to Definition 5, $\forall v_i \in V_1 \cup V_2$ implies that $P_{\varepsilon 1}(v_i) \neq P_{\varepsilon 2}(v_i)$ or $d_1(u, v_i) \neq d_2(u, v_i)$. Now we can define inheritance tree distance:

Definition 11: Tree Inheritance Distance (TID) between edge extended trees X_1 and X_2 is the total cost of the edited nodes $S_i = \{v_1, \ldots, v_i, \ldots, v_n\}$ between X_1 and X_2:

$$d_i(X_1, X_2) = \sum_{v_i \in S_i} \delta(v_i)$$

Algorithm 2 Tree Inheritance Distance Algorithm

```
input  : trees $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$,
         distance functions $d_1 : V_1 \times V_1 \to \mathbb{R}$, and
         $d_2 : V_2 \times V_2 \to \mathbb{R}$, and cost function
         $\delta' : V_1 \cup V_2 \times V_1 \cup V_2 \to \mathbb{R}$
output: an inheritance edit distance $d_i$ and set of edited
         nodes $S_i$
$S_i \leftarrow \varnothing, d_i \leftarrow \varnothing$;
foreach node $v$ in $V_1$ do
    $V_i \leftarrow \varnothing$;
    $P_{\varepsilon 1} \leftarrow$ getPreds($X_1$, $v$);
    if contains($V_2$, $v$) then
        $P_{\varepsilon 2} \leftarrow$ getPreds($X_2$, $v$);
        $V_i \leftarrow$ inheritanceChanges($P_{\varepsilon 2}$, $P_{\varepsilon 1}$)
    else
        $V_i \leftarrow P_{\varepsilon 1}$;
    end
    if $V_i \neq \varnothing$ then
        $S_i \leftarrow S_i \cup \{v\}$;
        $d_i \leftarrow d_i + \delta(v, V_i)$;
    end
end
foreach node $v$ in $V_2$ do
    $V_i \leftarrow \varnothing$;
    if not contains($V_1$, $v$) then
        $V_i \leftarrow$ getPreds($X_2$, $v$);
        $S_i \leftarrow S_i \cup \{v\}$;
        $d_i \leftarrow d_i + \delta(v, V_i)$;
    end
end
```

We can observe that set of inheritance edited nodes S_i is an extension of the set of edge edit operations S_e with added indirect operations. Since S_i is an extension of S_e, the main difference of inheritance distance algorithm shown in Algorithm 2, from edge edit distance Algorithm 1 is in detecting indirect operations. Therefore, the condition at line 4 in Algorithm 1, where parent nodes are compared is replaced by detection of inheritance operations in Algorithm 2 at line 7 by using Algorithm 3 (procedure `detectInheritanceChanges`). Algorithm 3 detects inheritance operations between source and target trees by detecting changes in predecessor nodes. In Algorithm 2, procedure `detectInheritanceChanges` is used to detect inheritance operations that occurred on nodes that are equal in both trees. However, Algorithm 3 could be used to detect direct inheritance operations caused by *add* and *delete* edge edit operations, but such operations are straightforwardly detected over predecessor nodes. In order to detect predecessor nodes for a given node, by traversing the path to the empty node, function `getPreds` is used. Note that the result of function `getPreds` is used as input for Algorithm 3, where function `getPreds` is used to obtain predecessor

Algorithm 3 Procedure detectInheritanceChanges (v, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$, d_1, d_2)

```
procedure detectInheritanceChanges($v$, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$, $d_1$, $d_2$)
    $V_i \leftarrow \varnothing$;
    foreach node $v$ in $P_{\varepsilon 1}$ do
        if contains($P_{\varepsilon 2}$, $v$) then
            if $d_1(u, v) \neq d_2(u, v)$ then
                $V_i \leftarrow V_i \cup \{v\}$;
            end
        else
            $V_i \leftarrow V_i \cup \{v\}$;
        end
    end
    foreach node $v$ in $P_{\varepsilon 2}$ do
        if not contains($P_{\varepsilon 1}$, $v$) then
            $V_i \leftarrow V_i \cup \{v\}$;
        end
    end
    return $V_i$;
```

node sets for nodes equal in both trees (lines 4 and 6 in Algorithm 2). In Algorithm 2, cost function δ' is used to calculate inheritance operations cost. Input parameters to cost function δ' are node and its inheritance operations, as a set of changed predecessor nodes V_i.

Algorithm 3 is similar to Algorithm 1, since *add* and *delete* inheritance operations are determined by the difference between node sets, in this case by predecessor node sets. However, instead of parent functions, *move* inheritance operation is determined by given distance functions d_1 and d_2.

In order to determine the time complexity of the algorithm to calculate tree inheritance distance, let the n_1 be the number of nodes in X_1, and n_2 be the number of nodes in X_2. Let k_1 and k_2 denote the average distance from a node to the empty node, i.e., average node depth in X_1 and X_2, respectively. The time complexity of the tree inheritance distance algorithm (Algorithm 2) is $O(n_1 * k_1 + n_2 * k_2)$. It is assumed that an appropriate `contains` function is used with the complexity $O(1)$, e.g., hash function. Similar to Algorithm 1, this algorithm iterates through a set of nodes V_1 and V_2, searching for matching nodes by using the lookup function `contains`. This function determines predecessor nodes, with path traversal complexity $O(k)$.

Furthermore, let us consider the case when all nodes are moved. In this case $n = n_1 = n_2$. The traversing of paths in both trees is performed only for matched nodes, thus n times, with the complexity $O(k_1)$ in the first tree and $O(k_2)$ in the second tree. Therefore, the complexity is equal to $O(n * (k_1 + k_2))$.

Procedure `detectInheritanceChanges` (Algorithm 3) additionally compares paths to the empty node for each matched node in both trees with the complexity

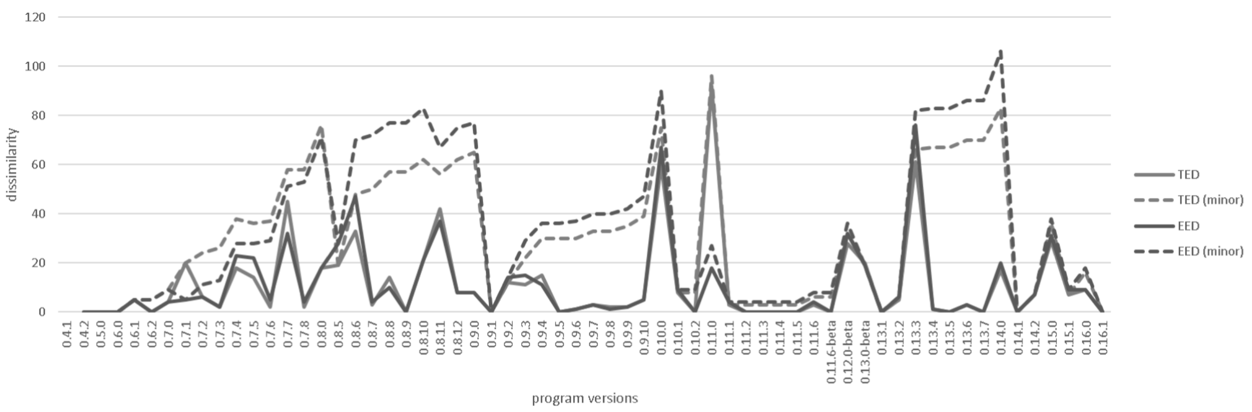


FIGURE 9. TED and EED result graph between Pinpoint program versions.

$O(k_1 + k_2)$, if an appropriate $O(1)$ function `contains` is used. Furthermore, the complexity of Algorithm 3, $O(k_1+k_2)$, is valid if the complexity of distance functions d_1 and d_2 are equal to $O(1)$. If the complexity of distance functions is greater than $O(1)$, it would increase the algorithm complexity. To calculate the tree inheritance distance, for each node, the traversing of predecessor nodes is performed twice, once in Algorithm 2, and once in Algorithm 3. However, the overall time complexity is not increased. E.g. in the case when all nodes are moved ($n = n_1 = n_2$), the complexity is equal to $O(n * (k_1 + k_2)) + n * O(k_1 + k_2) = O(2 * n * (k_1 + k_2)) = O(n * (k_1 + k_2))$.

V. EXPERIMENT RESULTS

The dissimilarity between class hierarchy in object-oriented languages can be used in program analysis for various tasks, such as regression testing, code clone detection, or dynamic software updating.

The class hierarchy can suitably be represented as a tree if multiple inheritances are omitted. Multiple inheritances enable inheritance from multiple classes and is supported by some programming languages, e.g., C++. However, it introduces an inheritance diamond problem [23], [24]. Most of the current object-oriented languages support only single inheritance, where a class can only inherit a single class. Consequently, in this paper, we consider the case of single inheritance only. On the other hand, Java interfaces define a set of methods that class supporting the interface implements. Interfaces are organized hierarchically and allow multiple inheritances, while classes can implement more than one interface. Interface hierarchy is not considered, since, the focus is on the class hierarchy. In this paper, if the class in the modified version implements the methods specified in the interface, the change reflects through the class hierarchy.

As an experiment, we measure the dissimilarity of class hierarchy between program versions in Java object-oriented language. In the first experiment, to evaluate the effectiveness of the EED algorithm introduced in section III, the results of the EED, are compared with the results of the TED algorithm [12]. To determine when the EED can be used instead of the TID algorithm introduced in section IV, in the second experiment, the results of the EED are compared to TID, both with the unit cost. In the third experiment, to evaluate the usefulness of TID, the results of a TID algorithm are analyzed when the cost is calculated based on the method changes in class. The results of the first experiment are shown in the subsection V-B, whereas the results for the second and third experiments are shown in subsection V-C.

A. EXPERIMENTS SETUP

The experiments consists in executing the dissimilarity measures algorithms on program versions of two publicly available Java programs, *Pinpoint*[1] and *NewPipe*.[2] *Pinpoint* is *Application Performance Management* (APM) tool with about 4000 classes in the latest version [25]. *NewPipe* is a streaming Android application with about 350 classes in the latest version [26]. Trees used as input for algorithms represent the class hierarchy of the single program version. In the first and second experiments, classes are considered without properties. Edge operations correspond to added, deleted, and moved class in the class hierarchy between two program versions. *Move* inheritance operations are detected as a change in the distance between nodes, i.e., as a structural change, whereas in the third experiment as a change in class methods. In the first and second experiments, operations cost is set to 1 for both edge and inheritance edit operations, whereas in the third experiment, cost function φ_m is based on the class method changes. Let $M_1(u)$ and $M_2(u)$ be sets of methods for class u in the source and target tree, respectively, then $\varphi_m(u, v)$ for $u \in V_i(v)$ is defined as follows:

$$\varphi_m(u, v) = \begin{cases} |M_1(u)|, & u \in V_d(v) \\ |M_2(u)|, & u \in V_a(v) \\ |M_1(u)\backslash M_2(u) \cup M_2(u)\backslash M_1(u)|, & u \in V_m(v) \end{cases}$$

[1] https://github.com/naver/pinpoint
[2] https://github.com/TeamNewPipe/NewPipe

Distance functions used to determine set of nodes $V_m(v)$, are defined as $d_1(u, v) = M_1(u)$ and $d_2(u, v) = M_2(u)$. Consequently, *move* inheritance operations are detected as changes in method sets, $M_1(u) \neq M_2(u)$ for class u. For the experiment, methods are compared by method signature.

Furthermore, for each experiment, there are two experimental setups. In the first setup, TED [12] and algorithms presented in this paper are executed on class hierarchy trees created from subsequent program versions, e.g., the first pair is the first and second version, the second pair is the second and third version, and so on until the last version pair. In the second experimental setup, each program version, i.e., revision, is compared to its corresponding minor version. Since some minor versions are not available, corresponding prerelease versions are used as minor versions instead, release candidate (RC) in *Pinpoint*, and beta in *NewPipe*. Comparison to the major version, could not be made, as major versions are not available for both programs.

B. EFFICIENCY

In the first experiment, we compare the EED algorithm with the ordered tree TED algorithm [12] based on the edit graph with the time and space complexity of $O(n_1 * n_2)$. Although this variant of TED is constrained by *add* and *delete* operations performed only on the leaf nodes, it is more efficient than the latest algorithm [27] with complexity in worst case $O(n^3)$, based on work [18] that is considered as the most general TED definition [21]. An ordered tree algorithm is used because algorithms for unordered trees are with greater complexity [21]. The results of both algorithms are compared to determine the extent to which results from algorithms correspond. As shown in section 2, it is expected that the TED can not be used to determine the exact number of changed edges as EED. However, the results can indicate whether EED could be used instead of TED in some usage scenarios.

Both EED and TED distances are calculated between the class hierarchy of all currently available 62 *NewPipe* program versions. It is expected that comparison to the minor versions will result in the greater distance for EED and TID results compared to subsequent versions, as program versions are more dissimilar as a program evolves over time. The results are shown in Figure 9. For the experimental setup with the comparison of subsequent versions, results of EDD and TED correspond, if not in the same or similar values, then in distance change tendency. The significant difference in distances is for versions *0.7.1.* where the value of the TED is higher than the EED, and a difference in the distance change tendency for program versions *0.9.3* and *v0.9.4*. On the other hand, e.g., in version *0.11.0*, the TED distance is much higher than the EED, whereas for the *0.8.5* and *0.8.13*, the EED distance is slightly higher than the TED, although the tendency of change in distances is the same. In the case with the comparison of the revision and minor versions, results conform to the results of the subsequent versions. If distances are not equal or similar, EED is with higher values than TED with an equal tendency in distance change. A greater difference in distance values is for the version *0.11.0*. The results of both distances conform to the prediction that distance values are higher in comparison with the minor version. On the other hand, smaller distance values can be used to determine if the difference between the program versions is small. From the results, it can be concluded that EED can be used instead of TED in scenarios where exact absolute value is not required. E.g., in computer vision, where the relative ratio between distance value for different patterns is relevant. Considering the time complexity of the EED algorithm is $O(n_1+n_2)$, EED provides a more efficient comparison than TED. On the other hand, the results of the experiment show that a significant difference between distance values can occur. Therefore, it is necessary to evaluate both distances for a specific use.

C. USEFULNESS

Before analyzing the results of the experiments, let us discuss introduced inheritance operations in the case of object-oriented classes. An object-oriented class can inherit members of the class, such as fields, methods, or properties, depending on the programming language. *Add* and *delete* inheritance operations are straightforward. If the class is added or deleted, to or from inheritance relationship, the descendant class inherits or does not any longer inherit members from the ancestor class. E.g., in dynamic software updating, it is important to detect which class members are valid in the updated program version. If a class implementing the method is moved in relation to the inheriting class, inheriting classes may still be able to use, e.g., the methods defined in ancestor classes. On the other hand, if the class members are changed, e.g., method is added or deleted, there is a change for subclasses through inheritance relationship. In the second experiment, edge edit operations are used from the structural aspect, similar to the first experiment, to show the connection between an edge and inheritance edit operations.

The results of the second experiment are shown in Table 2 for *Pinpoint* and Table 3 for *NewPipe*. The first column denotes the program version. The experiment is performed on all currently available versions, but the results shown in tables are from the last 14 versions for *Pinpoint* and the last 15 versions of *NewPipe*. Each class implicitly contains the root, i.e., object class as the ancestor class. As a root class does not change, it corresponds to the empty node in EET. Note that the properties, possibly inherited from the root class, are neglected regarding the specific program domain. On the other hand, there are two types of hierarchy in the tree created from the class hierarchy. First, a class hierarchy where the parent class is the root class, and the class does not have subclasses, do not introduce inheritance. Such class introduces a new behavior in the program, but the behavior is not further propagated through the hierarchy. It is a single class in the hierarchy with the depth equal to 1. However, to detect inheritance operation for such class is useful in program analysis to detect added and deleted functionality. The second type is the class hierarchy, where a class has

TABLE 2. Pinpoint experiment results.

Version	NOC	NOCH	EED p/m	TID p/m
1.7.0-RC1	3848	928	*/*	*/*
1.7.0-RC2	3854	931	3/*	5/*
1.7.0	3854	931	0/*	0/*
1.7.1	3854	931	0/0	0/0
1.7.2	3899	911	84/84	156/156
1.7.3-RC1	3901	909	2/84	4/156
1.7.3	3901	909	0/84	0/156
1.8.0-RC1	4339	959	100/183	189/343
1.8.0	4347	961	2/185	3/346
1.8.1-RC1	4563	970	104/104	218/218
1.8.1	4563	970	0/104	0/218
1.8.2-RC1	4567	968	2/102	4/214
1.8.2	4568	969	1/103	3/217
1.8.3	4569	969	0/103	0/217

* results from all versions are excluded

TABLE 3. NewPipe experiment results.

Version	NOC	NOCH	EED p/m	TID p/m
0.13.0-b	280	170	*/*	*/*
0.13.1	280	170	0/0	0/0
0.13.2	287	174	6/6	13/13
0.13.3	274	162	76/82	224/237
0.13.4	272	163	1/83	2/239
0.13.5	271	163	0/83	0/239
0.13.6	276	166	3/86	5/244
0.13.7	276	166	0/86	0/244
0.14.0	296	180	20/106	39/283
0.14.1	296	18	0/0	0/0
0.14.2	303	187	7/7	30/30
0.15.0	350	207	31/38	57/87
0.15.1	349	206	9/9	18/18
0.16.0	359	215	9/18	27/45
0.16.1	359	215	0/0	0/0

subclass, i.e., with class and, at least, one more class beside the root class, thus with a depth greater than 1. Program behavior introduced by such class is propagated through the class hierarchy. Consequently, the second column is the total number of classes (NOC) regardless of the hierarchy type, and the third column denotes the number of classes in hierarchies (NOCH) with the depth greater than 1. The fourth and fifth column refers to edge edit distance (EED) and tree edit distance (TID) experiment results. Distances are calculated both to the previous version, i.e., revision and to the previous minor version (p/m). Minor versions used for comparison are bolded in the first column.

From the aspect of usefulness for program analysis, results show that a program with a smaller number of classes can have a similar number of inheritance changes as a program with a larger number of classes. For example, in the case of *NewPipe*, inheritance distance between program versions *0.13.2* and *0.13.3* is equal to 224, whereas, in the case of *Pinpoint*, inheritance distance between versions *1.7.1* and *1.7.2* is equal to 156. Note that *NewPipe* in version *0.13.2* and *0.13.3* have 287 and 274 classes, respectively, whereas *Pinpoint* in versions *1.7.1.* and *1.72.* have 931 and 911 classes. Furthermore, as expected, the dissimilarity between subsequent versions is smaller than between revisions and major versions. Moreover, the results show the connection between edge edit and inheritance distance. The increasing dissimilarity of edges is reflected with increasing inheritance dissimilarity. Therefore, as an indication of structural changes in a class hierarchy, EED compared to TID is good enough, and with better efficiency.

The results for the third experiment are shown in Figure 10, similar to the first experiment, for all currently available 62 *NewPipe* program versions. The figure shows TID_m results with cost function φ_m, and TID results with the unit cost. Results for setup with subsequent versions show that TID results correspond to TID_m results. However, in some cases such as *v0.9.0*, *v0.9.3*, *v0.13.6*, and *v0.16.0*, there are also considerable differences in distance values. The similarity in the distance values is because *add* and *delete* inheritance operations prevail between the class hierarchy in subsequent program versions. Differences between distances occur when there is a significant number of classes "moved" based on the number of methods changes in classes that remain in the inheritance relationship with subclasses. On the other hand, the results of the comparison to minor version show that besides the comparison between *v0.8.9* to *v0.9.0*, there is no difference in distance changes. For the same reason as for the subsequent version comparison, *add* and *delete* inheritance operations prevail in comparison to the class hierarchy of the minor version. Based on the results, φ_m function is more suitable for use in scenarios for comparison of subsequent program versions. E.g., it can be decided whether to perform a dynamic update from version *v0.9.2* to *v0.9.3* or from *v0.11.6-beta* to *v0.12.0-beta*, as there is a significant number of method changes, which may lead to unexpected behavior after the update.

D. DISCUSSION

Based on the evaluation results, e.g., if inheritance distance is above some threshold, a decision can be made which classes of program version should be tested for regression testing, or which updates could be performed dynamically. Therefore, presented approaches to detect dissimilarity between class hierarchy could be used for program analysis, such as regression testing or dynamic software updating. The first experiment shows that EED compared to the TED [12] could be an efficient solution in some scenarios where accuracy and exact distances are not required. However, the experiment is performed by using unit costs and only for class hierarchy comparison. Particular usage requires evaluation and finding of appropriate cost functions. The results of

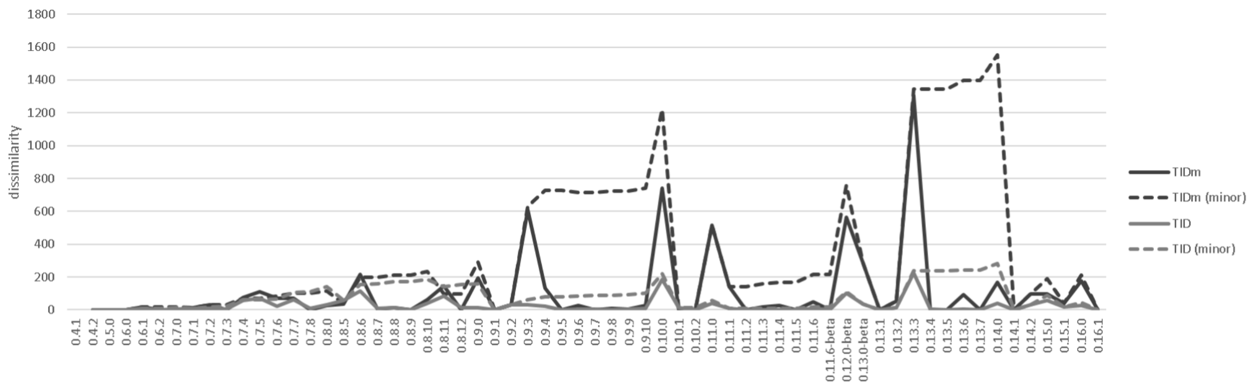


FIGURE 10. TID and TID$_m$ result graph between Pinpoint program versions.

the second and third experiments show that for class hierarchy comparison between revision and minor version, generally, EED could be used. The third experiment shows how more complex cost functions can provide more accurate results for a decision, based on specific usage, in this case of the methods changes. However, inheritance operations may affect inheriting classes if added or deleted class, or any class on the path to the root node overrides a method used in an inheriting class. This case could be detected with an appropriate distance function, e.g., the function can determine the distance to the first ancestor class that is implementing, i.e., overriding a class method. Consequently, the future work should include evaluating different approaches for cost functions, instead of the constant number and number of changed methods.

VI. RELATED WORK

There are various approaches in the related literature to determine dissimilarity between trees [12], [21]. Tree Edit Distance variants differ in the definition of used operations and the conditions of the distance calculation [21]. Most of the work is concentrated on the distance between ordered trees [16]–[18], [20], [27], [28]. Compared to unordered trees, there is an additional condition of order among siblings. The TED definition in [18] and derivative works on ordered [17], [27], and unordered [1] trees limit the possible operations by the condition of preserving the ancestor-descendant relationship in the mapping between the trees. In section II, the example shows that the resulting operations are not suitable for detecting relationship changes between nodes. Furthermore, edit distance is in general defined as the lowest transformation cost. However, the work most similar to ours is the unit cost algorithm [29] between ordered trees. Similar to our approach, distance is defined as the minimum number of operations, while the cost of node operations is a unit cost. The approach presented in this paper is based on unordered trees and cost for operations as cost functions. The approach in [28], similar to ours, defines operations on the edges instead of nodes, with *relabeling*, i.e., *substitute* instead of *move* edge operation. It is based on the work of [17] by using strings to calculate distance. On the other hand, the GED algorithm [13] works on graphs and does not set the condition of a preserved ancestor-descendant relationship. With the defined cost for *add*, *delete*, and *substitute* edge operation, GED can provide a similar result as EED. However, node operations are implicitly included, and the *move* edge operation defined in this paper is reflected as *delete* and *add* edge operation. Furthermore, the best-first algorithm is not efficient because the time complexity is exponential.

VII. CONCLUSION

In this paper, edit operations on trees are presented by edge edit operations, with the principal motivation to detect changes in relationships between nodes. Edge edit operations are introduced on the Edge Extended Tree (EET), to support the same operations on all nodes in the tree, including the root node. Based on EET and edge edit operations, Edge Edit Distance (EED) is introduced, an edge dissimilarity measure between unordered trees. It is shown that edge modifications cause indirect and inheritance changes in the relationship between nodes. In order to detect inheritance changes between unordered trees, inheritance operations are introduced, and a dissimilarity measure, Tree Inheritance Distance (TID) is defined. Efficient algorithms are presented for the introduced measures, which are evaluated in experiment analysis of two publicly available object-oriented programs. The results of comparing the EED with the TED variant indicate that the EED could be used in some scenarios instead of the TED. Furthermore, EED and TID experiment results on the class hierarchy between program versions, are promising in the program analysis domain, with emphasis on the software evolution. Future work will include the evaluation of various functions to observe changes in node and edge properties. Furthermore, edge edit and tree inheritance distance could be used in other application domains with a focus on the inheritance relationship, e.g., molecular biology or phylogeny.

REFERENCES

[1] D. Shasha, J. T.-L. Wang, K. Zhang, and F. Y. Shih, "Exact and approximate algorithms for unordered tree matching," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, no. 4, pp. 668–678, Apr. 1994.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *ACM SIGMOD Rec.*, vol. 25, no. 2, pp. 493–504, Jun. 1996, doi: 10.1145/235968.233366.

[3] W. Daelemans, K. De Smedt, and G. Gazdar, "Inheritance in natural language processing," *Comput. Linguistics*, vol. 18, no. 2, pp. 205–218, Jun. 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=142235.142243

[4] M. Hashimoto and A. Mori, "Diff/TS: A tool for fine-grained structural change analysis," in *Proc. 15th Work. Conf. Reverse Eng.*, Oct. 2008, pp. 279–288.

[5] B. A. Shapiro and K. Zhang, "Comparing multiple RNA secondary structures using tree comparisons," *Bioinformatics*, vol. 6, no. 4, pp. 309–318, 1990, doi: 10.1093/bioinformatics/6.4.309.

[6] S. Dulucq and L. Tichit, "RNA secondary structure comparison: Exact analysis of the Zhang–Shasha tree edit algorithm," *Theor. Comput. Sci.*, vol. 306, nos. 1–3, pp. 471–484, Sep. 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397503003232

[7] W. Yang, "Identifying syntactic differences between two programs," *Softw., Pract. Exper.*, vol. 21, no. 7, pp. 739–755, Jul. 1991. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210706

[8] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005, doi: 10.1145/1082983.1083143.

[9] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," in *Proc. Int. Workshop Mining Softw. Repositories (MSR)*. New York, NY, USA: ACM, 2006, pp. 65–71, doi: 10.1145/1137983.1138000.

[10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.

[11] M. Böhme, A. Roychoudhury, and B. C. D. S. Oliveira, "Regression testing of evolving programs," in *Advances in Computers*, vol. 89, A. Memon, Ed. Amsterdam, The Netherlands: Elsevier, 2013, ch. 2, pp. 53–88. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780124080942000023

[12] G. Valiente, *Algorithms on Trees and Graphs*. Berlin, Germany: Springer-Verlag, 2002.

[13] K. Riesen, *Structural Pattern Recognition With Graph Edit Distance: Approximation Algorithms and Applications*, 1st ed. Cham, Switzerland: Springer, 2016.

[14] J. Kobler, U. Schöning, and J. Toran, *The Graph Isomorphism Problem: Its Structural Complexity*. Boston, MA, USA: Birkhäuser, 2012.

[15] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Anal. Appl.*, vol. 13, no. 1, pp. 113–129, Feb. 2010, doi: 10.1007/s10044-008-0141-y.

[16] J. T. L. Wang and K. Zhang, "Finding similar consensus between trees: An algorithm and a distance hierarchy," *Pattern Recognit.*, vol. 34, no. 1, pp. 127–137, Jan. 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320399001995

[17] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989, doi: 10.1137/0218082.

[18] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, Jul. 1979, doi: 10.1145/322139.322143.

[19] R. A. Wagner and M. J. Fischer, "The String-to-String correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974, doi: 10.1145/321796.321811.

[20] S. M. Selkow, "The tree-to-tree editing problem," *Inf. Process. Lett.*, vol. 6, no. 6, pp. 184–186, Dec. 1977. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0020019077900643

[21] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, nos. 1–3, pp. 217–239, Jun. 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397505000174

[22] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[23] G. B. Singh, "Single versus multiple inheritance in object oriented programming," *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, Jan. 1995, doi: 10.1145/209866.209871.

[24] M. Sakkinen, "Disciplined inheritance," in *Proc. ECOOP*, vol. 89, Jul. 1989, pp. 39–56.

[25] (Jul. 2019). *APM, (Application Performance Management) Tool for Largescale Distributed Systems Written in Java.: Naver/Pinpoint*. [Online]. Available: https://github.com/naver/pinpoint

[26] (Jul. 2019). *A Libre Lightweight Streaming Front-End for Android.: TeamNewPipe/NewPipe*. [Online]. Available: https://github.com/TeamNewPipe/NewPipe

[27] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, Eds. Berlin, Germany: Springer, 2007, pp. 146–157.

[28] P. N. Klein, "Computing the edit-distance between unrooted ordered trees," in *Algorithms—ESA*, G. Bilardi, G. F. Italiano, A. Pietracaprina, and G. Pucci, Eds. Berlin, Germany: Springer, 1998, pp. 91–102.

[29] D. Shasha and K. Zhang, "Fast algorithms for the unit cost editing distance between trees," *J. Algorithms*, vol. 11, no. 4, pp. 581–621, Dec. 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0196677490900113

DANIJEL MLINARIĆ (Member, IEEE) received the M.Sc. degree in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, in 2009, where he is currently pursuing the Ph.D. degree in computer science. From 2009 to 2014, he was a Research Associate with the University of Zagreb, working on national information systems of application and enrollment in secondary schools and higher education institutions. He is also a Research and Teaching Assistant with the Faculty of Electrical Engineering and Computing, University of Zagreb. His research interests include software engineering focused on software evolution, program analysis, and dynamic software updating.

BORIS MILAŠINOVIĆ (Member, IEEE) received the degree from the Department of Mathematics, Faculty of Science, University of Zagreb, in 2001, and the M.Sc. and Ph.D. degrees in computing from the Faculty of Electrical Engineering and Computing, in 2006 and 2010, respectively. He is currently an Associate Professor with the Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb. His main research interests include software development methodologies and workflow management. He has been a member of editorial board of *Computer Science and Information Systems* journal, since 2018, and a program committee member of several international conferences.

VEDRAN MORNAR (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, in 1981, 1985, and 1990, respectively. Since 1982, he has been working with the Faculty of Electrical Engineering and Computing, University of Zagreb. From 2002 to 2006, he was the Vice Dean of the Faculty. From 2006 to 2010, he was the Dean. From 2009 to 2013, he was the President of the National Council for Higher Education. He is serving as the President of the Croatian Association for Information and Communication Technology (MIPRO). From 2014 to 2016, he held the Office of the Minister of Science, Education and Sports of the Republic of Croatia. He is currently a Full Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. He was the Project Leader of several projects on the national level. Most notably, the National Information System for application to HEI, which also provides complete organizational support for the State Matura exams. His professional interests are in e-learning, application of operational research in real-world information systems, database design, development, and implementation.