Location Aware Source Code Differencing for Mining Changes in Java

Jindae Kim and Sunghun Kim
The Hong Kong University of Science and Technology
{jdkim, hunkim}@cse.ust.hk

***Abstract*—Source code differencing techniques have played important roles in software development and research. One common strategy of differencing is converting source code into ASTs and matching equivalent nodes between ASTs to identify changed nodes. However, sometimes two similar, but not equivalent nodes are matched incorrectly and it leads to edit scripts which are distant from human's perception about changes. To mitigate this issue, we introduce LAS, a novel location aware source code differencing algorithm for mining changes. LAS employs multiple node matching steps considering syntactic and relative locations of nodes to find more appropriate node matches. In qualitative evaluation, we compared generated edit scripts for 10 changes with human's scripts independently written the same changes. The result indicates that LAS generated edit scripts more or equivalently similar to human-written scripts compared to the state-of-the-art technique for 7 out of 10 changes. Runtime performance evaluation on about a half million changes also shows that LAS generates edit scripts at least 10 times and twice faster for 27.93% and 95.78% of the changes, and saved 15 hours of wall clock time on a high performance server.**

I. Introduction

Source code differencing techniques have played important roles in software development and research. These techniques have been used in version control systems to help developers for identifying changes. They are also often employed in empirical studies involving change mining tasks to automatically collect a large number of changes from software repositories [9], [30], [18].

One common differencing strategy is converting source code related to a change into abstract form such as Abstract Syntax Tree (AST), and identifying changed nodes from converted ASTs. Differencing techniques using ASTs first try to match equivalent nodes existing in both ASTs. Once links between two ASTs are established by node matches, differencing techniques generate an edit script with edit operations such as insert, delete or update for unmatched nodes or matched nodes having differences. Therefore matching nodes which are not actually equivalent in two ASTs may lead to an inaccurate edit script with confusing description about a change.

To find node matches, differencing techniques often employ heuristics which compare node types or compute the similarity of two nodes based on the similarity of two subtrees rooted at the two nodes. The issue is that these techniques usually try to match as many nodes as possible to generate smaller edit scripts by identifying more unchanged nodes. During this process, some of nodes representing similar code fragments are matched although they do not remain unchanged and accidentally similar like code clones. Because of these wrong matches, generated edit scripts are sometime not in line with human developers' perception about actual source code changes. If a differencing algorithm is used for one single change, such glitches might be acceptable since users can quickly compare them with source code change. However, this issue may affect change mining tasks more significantly when generated edit scripts for numerous changes are accumulated and their statistics are analyzed.

We introduce a novel **L**ocation **A**ware **S**ource code differencing algorithm (LAS) to mitigate this issue for mining changes. LAS employs multiple node matching steps considering syntactic and relative locations of nodes to find more appropriate node matches. In the first two matching steps, LAS simultaneously traverses two ASTs in level order and tries to match two nodes visited at the same time. If these nodes are not matched, LAS expands search region to nodes close to them to find appropriate matches for the visited nodes. By matching more similar and more close nodes first, LAS tries to avoid matching similar but not equivalent nodes which might be appeared frequently due to repetitiveness of small size code fragments or changes [11], [21], [23]. At the next matching step, LAS identifies follow-up node matches based on already matched nodes. In human's eyes, two code fragments surrounded by the same code fragments often look corresponding although they are not quite similar. For example, two statements having the same previous and next statements is likely to correspond to each other although they are not very similar. This matching step reflects such intuition to find node matches which might be missed by similarity constraint.

To evaluate the quality of edit scripts generated by LAS, we randomly select 10 source code changes from five different Java open source projects. Then we generate edit scripts by apply LAS to these changes, and compare them with scripts independently written by human. We design the score of generated edit script representing the similarity of generated edit scripts to human-written scripts, and compare scores of edit scripts generated by LAS and the state-of-the-art source code differencing technique, GumTree [8]. In this way, we can systematically evaluate the quality of generated edit scripts with respect to human's perception about changes. Evaluation result shows that LAS generated edit scripts more or equivalently similar to human-written scripts for 7 out of 10 changes. This result indicates that LAS's edit scripts are more close to human's perception about changes, hence it provides more reliable results for change mining tasks.

We also evaluate runtime performance and the size of generated edit scripts. Since change mining tasks often process a huge amount of changes, runtime performance is an important factor to evaluate source code differencing technique. We collect 435,282 changes from the same five projects used in quality evaluation, and apply LAS and GumTree to generate edit scripts for the changes. In the evaluation results, LAS generated edit scripts at least 10 times faster for about 27.93% of the changes, and LAS performs at least two times faster for 95.78% of the changes. The actual saved wall clock time is almost 15 hours for less than a half million changes with a high performance server, hence this time reduction is quite meaningful particularly when LAS is used for change mining tasks which often involved with more than a million changes [30], [18], [21].

This paper makes the following contributions:

- We introduce a novel source code differencing algorithm LAS, which is designed for mining changes.
- We present a new systematic method of evaluating the quality of generated edit scripts with respect to human's perception about changes and evaluation results.
- We provide a large scale runtime performance evaluation results on 435,282 real changes from five different open source projects.

The remainder of this paper is organized as follows. We introduce LAS, our source code differencing algorithm in Section II with an example change. Section III describes how we evaluated the performance of LAS. We present the results of evaluation in Section IV, and we discuss about our findings in Section V. We argue the validity of our experiment design and results in Section VI. After that, we discuss about related work in Section VII and we conclude with future work in Section VIII.

II. Differencing Algorithm

LAS identifies differences and generates edit scripts in three steps: tree building, node matching and edit script generation. First LAS parses old and new versions of source code and builds old and new ASTs. Then it finds node matches between the two ASTs to use that for identifying inserted or deleted AST nodes as well as moved or updated nodes. After node matching is done, LAS generates node level edit operations for each changed AST node and connects them based on the connectivity of changed AST nodes to create an edit script with AST subtree level edit operations.

A. Tree Building

In tree building step, LAS parses old and new versions of source code related to a change, and builds old and new ASTs respectively. Each AST consists of labeled AST nodes, and each AST node has its hash string and node type vector which represents an AST subtree rooted at the node. The label of an AST node represents the type of the AST node, and if this node contains a specific value such as variable names, literals, modifiers or operators, this value is also included in the label. These hash strings and node type vectors will be used to find exact matches and similar matches in node matching step respectively.

Fig. 1 shows textual differences of an example change *derby2* used in our edit script quality evaluation. Fig. 2a and Fig. 2b are two AST subtrees related to source code shown in Fig. 1. Each AST node has a label which indicates the type of the node. If a node has a specific value, this value is shown in the node's label within brackets. For example, the operator "==" of an infix expression in Fig. 2a is shown in the label, hence LAS can recognize that the operator is changed to "!=" in Fig. 2b. Note that we omit some code fragments such a block - shown as the root node in the ASTs - in the source code for brevity.

B. Node Matching

Once old and new ASTs have been built, LAS finds node matches between the two ASTs with four matching steps. We first describe how LAS traverses ASTs and finds candidates for matches, and then we explain detailed methods how LAS matches two AST nodes in each matching step.

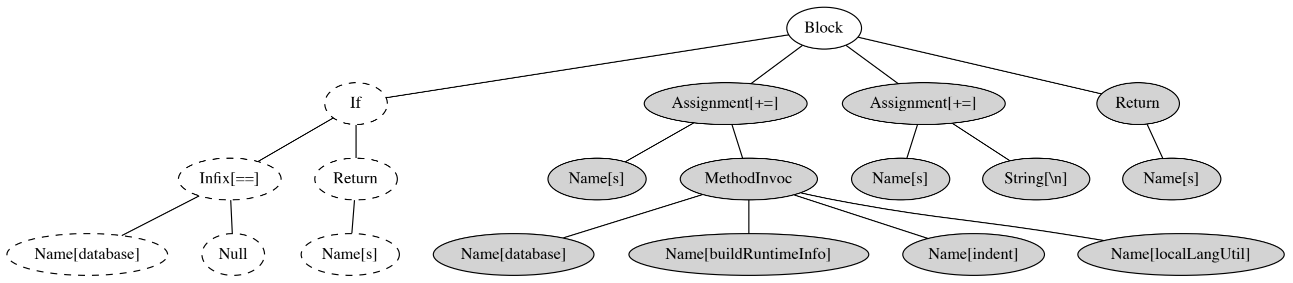
1) Identifying Match Candidates: To consider location of AST nodes during node matching, LAS visits AST nodes at the same depth of two ASTs one by one, and tries to match two nodes visited at the same time. For example, LAS first compares two Block nodes at the root of the trees in Fig. 2. Then LAS tries to match their child nodes by comparing them one at a time (the `If` and `VarDecl` nodes, the `Assignment[+=]` and `If` nodes, etc.). If one AST node has more child nodes than the other AST node, remaining nodes are compared with the last visited node. For instance, there are four nodes and three nodes at depth 2 in Fig. 2a and Fig. 2b respectively. If the nodes are visited one by one, Fig. 2b does not have a corresponding node for the last `Return` node in Fig. 2a. In this case, LAS simply tries to match this `Return` node with the last visited node in Fig. 2b, which is another `Return` node.

If a node x is not matched to another node y visited at the same time, LAS tries to match the node y to other candidate nodes close to y. Algorithm 1 represents how LAS identifies match candidates for given node x and y. To identify candidates, LAS lists nodes within distance and depth threshold in a specific order. First, LAS adds y's siblings within the distance threshold as candidates from close nodes to distant nodes (line 5-6). Next, nodes at the same depth as y are added to candidates (line 7-9). To identify the same depth nodes with the distance threshold, LAS first identifies siblings of y's parent within the distance threshold, then adds all their child nodes as candidates. After that, LAS includes descendants of y and its siblings within the depth thresholds to candidates (line 10-14). Finally, ancestors of y within the depth threshold are added to the candidates (line 15-21).

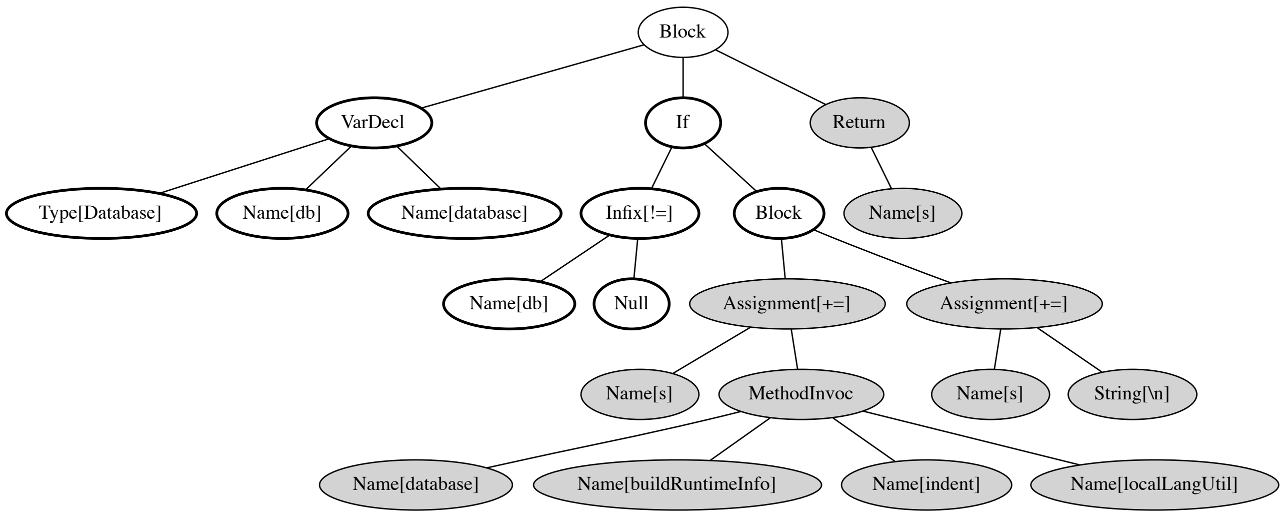
This specific order of identifying candidate is to apply heuristic which reflects the structure of AST according to source code. LAS first checks siblings since an added or deleted code fragment may simply shift an AST node to another position in the same parent node. Then LAS tries other nodes in the same depth. These same depth nodes often

```
-         if (database == null)
-             return s;
-         s += database.buildRuntimeInfo(indent,localLangUtil);
-         s += "\n";
+         Database db = database;
+         if (db != null) {
+             s += db.buildRuntimeInfo(indent, localLangUtil);
+             s += "\n";
+         }
          return s;
      }
```

Fig. 1: An Example change $derby2$ used in our qualitative evaluation. Parts of the source code including comments and whitespace are hidden for brevity.



(a) An old AST parsed from the old version of source code.



(b) A new AST parsed from the new version of source code.

Fig. 2: The AST pair parsed from the example change $derby2$. Node labels with brackets indicate that they have specific values shown in the brackets. Nodes filled with gray are matched in exact matching step. Nodes with dashed lines or bold lines are not matched nodes. Dashed line indicates a node is deleted, and bold line represents a node is inserted.

represent expressions in other statements in the same scope of source code. LAS also considers descendants before ancestors. Since surrounding code fragments with a block makes a node moved to a new node's descendant, it is more likely to find a right match from descendants than ancestors.

2) *Exact Matches:* Exact matching step is designed to match unchanged parts of AST by quickly matching entire AST subtrees instead of only one AST node pair. For each AST node, LAS assigns a hash string which represents the whole subtree rooted at the AST node. By comparing two hash strings, LAS can find exact matches of two AST nodes as well as matches of all their descendant nodes.

We use Dyck word hashing [6] of an AST for the exact matching step. The Dyck word D_n of a node n with k child nodes is defined as follows.

$$D_n = \{n.label\{D_{c_1}, D_{c_2}, \cdots, D_{c_k}\}\},$$
$$\text{where } c_1, c_2, ...c_k \text{ are child nodes of } n$$

For example, a Dyck word of a grey filled `Assignment` node in Fig. 2a is "{Assignment[+=]{Name[s],String[
n]}}." This Dyck word represents the structure of an AST subtree rooted at the node, as well as specific values included in the AST subtree. Therefore LAS can use Dyck words as hash strings to find the same AST subtree representing the same assignment in Fig. 2b. LAS can also match two return

Algorithm 1 Identify Match Candidates

```
1: DIST: distance threshold
2: DEPTH: depth threshold
3: procedure FINDCANDIDATES($x$, $y$)
4:     candidates: empty
5:     siblings ← getSiblings($y$)
6:     $candidates.addSameTypeNodes(siblings, x.type)$
   //Add nodes with type x.type only.
7:     parentSiblings ← GetSiblings($y.parent$)
8:     for all $n$ in parentSiblings do
9:         $candidates.addSameTypeNodes(n.children, x.type)$
10:    $depth \leftarrow 1$
11:    while $depth < DEPTH$ do
12:        $siblings \leftarrow siblings.children$
13:        $candidates.addSameTypeNodes(siblings, x.type)$
14:        depth++
15:    depth ← 1
16:    parent: y.parent
17:    while $depth < DEPTH$ do
18:        $ancestors \leftarrow parent+$GetSiblings($parent$)
19:        $candidates.addSameTypeNodes(ancestors, x.type)$
20:        $parent \leftarrow parent.parent$
21:        depth++
       return candidates
22: procedure GETSIBLINGS(y)
23:    $siblings$: empty
24:    $range$ = # of $y$'s siblings $\times DIST$
25:    $dist \leftarrow 1$
26:    while $dist \leq range$ do
27:        $siblings$ += node $n$ where $n.index = y.index - dist$
28:        $siblings$ += node $n$ where $n.index = y.index + dist$
29:        $dist++$
       return $siblings$
```

statements represented by grey nodes. Remind that Fig. 1 only shows one changed method of a Java class. LAS can quickly find exact matches of unchanged parts of AST such as methods or fields based on their hash strings, and it can even find some smaller exact matches of statements in a changed method too. Therefore LAS can save a lot of effort to compare each AST node one by one and improve the runtime performance.

Algorithm 2 describes exact matching step of LAS. As we explained in Section II-B1, LAS visits two ASTs in level order and tries to match two nodes visited at the same time (line 4-7). If the two ASTs do not have the same number of AST nodes at a certain level, LAS tries to match remaining nodes in one AST with the last visited node ($lastY$) in the other AST (line 8-11). Similarly, one AST might be deeper than the other tree and some nodes have no corresponding node at the same level. LAS also uses the nodes at the last visited level for exact matching in this case (line 12-19). To find an exact match for an individual AST node x, LAS compares the hash string of x with hash strings of y and other candidates found by Algorithm 1 (line 21-24). Once an exact match c of x is found, LAS updates the match of x with c (line 25). Since the exact match of two nodes also means that all their descendants are also matched, LAS updates the match of the descendant nodes accordingly (line 26-29).

Algorithm 2 Find Exact Matches

```
1: procedure EXACTMATCH(nodes1, nodes2)
2:     if $nodes1.size < nodes2.size$ then
3:         swap($nodes1$, $nodes2$)
4:     for all $y$ in $nodes2$ do
5:         $x$: $nodes1$.get($y.index$)
6:         if $x.match = null$ && $x \neq$ leaf node then
7:             EXACTMATCH($x$, $y$)
8:     $lastY$: the last node in $nodes2$
9:     for all $x$ in the remainder of $nodes1$ do
10:        if $x.match = null$ && $x \neq$ leaf node then
11:            EXACTMATCH($x$, $lastY$)
12:    $children1 \leftarrow$ children of unmatched nodes in $nodes1$
13:    $children2 \leftarrow$ children of unmatched nodes in $nodes2$
14:    if $children1.size > 0$ && $children2.size > 0$ then
15:        EXACTMATCH($children1$, $children2$)
16:    else if Only $children1.size = 0$ then
17:        EXACTMATCH($nodes1$, $children2$)
18:    else if Only $children2.size = 0$ then
19:        EXACTMATCH($children1$, $nodes2$)
20: procedure EXACTMATCH($x$, $y$)
21:    $candidates \leftarrow y$ + FINDCANDIDATES($x$, $y$)
22:    while $x.match = null$ && $candidates.size > 0$ do
23:        $c \leftarrow candidates.pop()$
24:        if $c.match = null$ && $c.hash = x.hash$ then
25:            $x.match \leftarrow c$
26:            for all $d_x, d_c$ in descendants of $x$ and $c$ do
27:                if $d_x.match = null$ && $d_c.match = null$ then
28:                    $d_x.match \leftarrow d_c$
29:                    $d_c.match \leftarrow d_x$
```

3) Similar Matches: In similar matching step, LAS finds similar matches from remaining unmatched nodes after the exact matching step. Algorithm 3 presents the similar matching step of LAS. Similar matching step is analogous to exact matching step, except for two major differences: the similarity threshold and mutual matching. To find a similar match for a given node x, LAS computes the similarities between node x and all identified match candidates, and finds the best match $x.best$ among candidates with similarity higher than the similarity threshold SIM (line-22-29). The similarity of two nodes is defined as the similarity of two node type vectors assigned to the nodes, and this similarity represents the ratio of common nodes to all nodes in two subtrees rooted at the given two nodes. After the best matches for all given nodes are identified, LAS confirms each match of two nodes only if they are mutually the best matches for each other (line 5-8). Once two nodes are confirmed as matched, LAS identifies similar matches for children of the two matched nodes (line 9), similar to exact matching step updates all descendant matches. Note that similar matching step does not consider matches for leaf nodes (line 20). Leaf nodes have no child nodes, hence their node type vectors do not provide enough information to find matches between them. Therefore LAS matches leaf nodes separately in leaf node matching step to find more correct matches among them.

Algorithm 3 Find Similar Matches

```
1: procedure SIMILARMATCH(nodes1,nodes2)
2:     SIM: similarity threshold
3:     UPDATECANDIDATES(nodes1, nodes2)
4:     UPDATECANDIDATES(nodes2, nodes1)
5:     for all $n$ in $nodes1$, $nodes2$ do
6:         if $n = n.best.best$ then
7:             $n.match \leftarrow n.best$
8:             $n.best.match \leftarrow n$
9:             SIMILARMATCH($n.children$, $n.match.children$)
10:    $children1 \leftarrow$ children of unmatched nodes in $nodes1$
11:    $children2 \leftarrow$ children of unmatched nodes in $nodes2$
12:    if $children1.size > 0$ && $children2.size > 0$ then
13:        SIMILARMATCH($children1$, $children2$)
14:    else if Only $children1.size = 0$ then
15:        SIMILARMATCH($children2$, $nodes1$)
16:    else if Only $children2.size = 0$ then
17:        SIMILARMATCH($children1$, $nodes2$)
18: procedure UPDATECANDIDATES($nodes1$,$nodes2$)
19:    for all $x$ in $nodes1$ do
20:        if $x.match = null$ && $x \neq$ leaf node then
21:            $y \leftarrow nodes2$.get($x.index$) or $lastX$
22:            $candidates \leftarrow y$ + FINDCANDIDATES($x$, $y$)
23:            $highest \leftarrow null$
24:            for all $c$ in $candidates$ do
25:                $sim \leftarrow$ similarity($x$,$c$)
26:                if $sim = 1$ then stop.
27:                else if $sim \geq SIM$ then
28:                    update $highest$ based on $sim$.
29:            $x.best \leftarrow highest$
```

Unlike general node type vectors using only AST node types [12], [6], LAS uses both AST node type and specific values shown in a node's label for node type vectors. Generally a node type vector of an AST subtree is a vector whose component represents the number of occurrences for each node type. For example, suppose there are three AST node types `Infix`, `Name`, and `Null`, and a node type vector is represented by (`Infix`, `Name`, `Null`). Then general node type vectors of the infix expression subtrees in Fig. 2a and Fig. 2b are both (1, 1, 1). However, LAS uses nodes' labels for node type vectors, and node type vectors can be represented with more dimensions like (Infix[==], Infix[!=], Name[database], Name[db], Null). Hence new node type vectors for the same infix expression subtrees are (1, 0, 1, 0, 1) and (0, 1, 0, 1, 1), and LAS can distinguish these two different code fragment, which cannot be distinguished with the previous node type vectors. In this way, a subtree with more common variable names and literals is considered more similar than another subtree of the same structure, but less common variable names and literals.

We can compute the similarity of two node type vectors by converting the vectors into two multisets and calculating the ratio of the common element number to the total element number. Equation 1 shows how to compute the similarity of two multisets v_1 and v_2 representing two node type vectors.

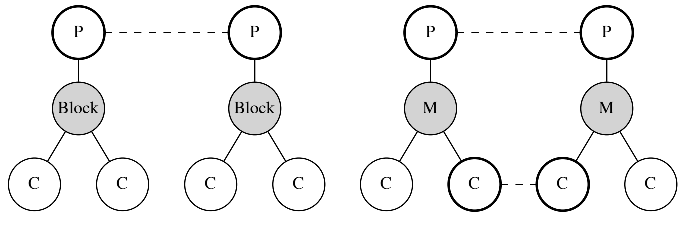
$$similarity(v_1, v_2) = \frac{2 \times |v_1 \cap v_2|}{|v_1| + |v_2|} \quad (1)$$

First we consider a node type vector as a multiset of node labels, since each component of the vector represents the multiplicity of a certain node label. The intersection of two multisets indicates the common AST nodes between two AST subtrees represented by the original node type vectors. Then, we divide the total number of elements by the number of common elements in two multisets to obtain the similarity. Note that we multiply the cardinality of the intersection by two since the common elements are included in the both multisets.

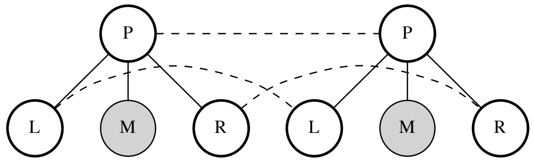
With the above definition of the similarity (Equation 1), we can control the generated edit scripts more intuitively by using different similarity thresholds. If only one node is changed in a subtree with three nodes, the similarity of two versions of the subtree is $\frac{4}{6} \simeq 0.67$. Since a subtree with three nodes is quite frequent (e.g. infix expressions, assignments) in AST, we use 0.65 as the default similarity threshold to allow only one node change for such cases. For example, in Fig. 2, the operator and one of the operands are changed in the infix expression. With the default similarity threshold, LAS considers these two operators are not matched, and generates an edit script indicating the infix expression in Fig. 2a is deleted and the infix expression in Fig. 2b is inserted. However, if we want to allow more generous matching and generate a smaller edit script, we can use the similarity threshold 0.3 to match two infix expressions and consider both operator and operand are updated.

4) Follow-up Matches: After exact matches and similar matches are found, LAS identifies follow-up matches from remaining unmatched nodes based on already matched nodes. For example, in Fig. 2, the root nodes (`Block`) of two ASTs indicate a method body before and after a change. The similarity of the two nodes is $\frac{24}{39} \simeq 0.62$, which is lower than the default similarity threshold. This low similarity is due to the inserted `VarDecl` subtree and the non-matched rooted at the `If` node. However, it is obvious that these two root nodes should be matched, since they represent the same method body in the source code shown in Fig. 1. Therefore LAS identifies such follow-up matches using the matching status of parents, siblings, and children.

Fig. 3 shows three types of follow-up matches identified by LAS. First, LAS matches two `Block` nodes if their parents are already matched (Fig. 3a). Since a block statement works as a container of other statements, we usually consider that a block belongs to a statement where it is attached. For example, a block attached to a method declaration is called a method body, and a block shown with an `if` or `else` statement is often referred to as a `then`-block or `else`-block. Therefore it is reasonable to match `Block` nodes based on their parent matches. Second, if two nodes have matched parents and at least one child node match, these two nodes are also matched (Fig. 3b). This match type is necessary since the similarity of two nodes might be low if subtrees rooted at the two nodes have small number of nodes. Although the similarity of two nodes is low, it is highly unlikely that the two nodes are not the same node if their parent nodes and one of their child nodes



(a) Type 1 - Block (b) Type 2 - Intermediate



(c) Type 3 - Surrounded

Fig. 3: Follow-up match types used in LAS. Label P, C, L, and R of nodes indicate parent, child, left and right nodes respectively. Nodes with bold lines are already matched nodes. Nodes filled with grey represent follow-up matches.

Algorithm 4 Find Follow-up Matches

```
1:  procedure FOLLOWUPMATCH($T_1$, $T_2$)
2:      $bfs \leftarrow$ breadth-first traversal of $T_1$
3:      for all $n$ in $bfs$ do
4:          if $n.match \neq null$ && $n \neq$leaf node then
5:              $pmatch \leftarrow n.parent.match$
6:              if $pmatch \neq null$ then
7:                  $candidates \leftarrow pmatch.children$
8:                  if $n.type$ =Block then
9:                      for all $c$ in $candiates$ do
10:                         if $c.type$ =Block then
11:                             match $n, c$ and stop.
12:                 else
13:                     FOLLOWUPMATCH($n$, $pmatch$)
14:     Repeat the above loop with $T_2$.
15: procedure FOLLOWUPMATCH($n$, $pmatch$)
16:     $SIM$: similarity threshold.
17:     for all $child$ in $n.children$ do
18:         $c \leftarrow child.match.parent$
19:         if $child.type$ = Block then
20:             FOLLOWUPMATCH($child$, $pmatch$)
21:         else if $c.match = null \&\& c.parent = pmatch$ then
22:             match $n, c$ and stop.
23:     if $n.match = null$ then
24:         $lmatch \leftarrow n.left.match$
25:         $rmatch \leftarrow n.right.match$
26:         if $lmatch.parent = pmatch$ then
27:             $c \leftarrow lmatch.right$
28:         else if $rmatch.parent = pmatch$ then
29:             $c \leftarrow rmatch.left$
30:         if leafsimilarity($n$, $c$) $\geq SIM$ then
31:             match $n, c$.
```

are already matched. Third, if two nodes are surrounded by matched nodes, LAS computes the ratio of leaf nodes having same labels between the two nodes, and match these nodes if the ratio is greater than the similarity threshold. As shown in Fig. 3c, it is highly likely that two nodes are matched if their parents are matched and both of their left and right nodes are matched too. However, still there exists a possibility that the two nodes are not matched since none of their child nodes are matched. If there exist any child node matches, they should be already found as the second type follow-up match. We cannot use the similarity defined in Equation 1 since it is already low due to some differences in the two nodes' descendants. Instead, LAS tests the leaf similarity of the two nodes (Equation 2), because leaf nodes often contain specific values such as names, types, modifiers and literals. For example, LAS can match two method declaration nodes based on their leaf nodes representing method name, modifiers and return type, even if the method body of this method is significantly changed.

$$\begin{aligned} &leafsimilarity(v,w) = \frac{2 \times L_c}{L_v + L_w}, \text{where} \\ &L_c : \text{the number of same label leaf nodes in } v, w, \\ &L_n : \text{the number of leaf nodes in } n \end{aligned} \tag{2}$$

Algorithm 5 Find Leaf Node Matches

```
1:  procedure LEAFMATCH($T_1$, $T_2$)
2:      $bfs \leftarrow$ breadth-first traversal of $T_1$
3:      for all $n$ in $bfs$ do
4:          if $n.match \neq null$ && $n \neq$leaf node then
5:              $L_n \leftarrow$ unmatched leaf nodes of $n$
6:              $L_m \leftarrow$ unmatched leaf nodes of $n.match$
7:              for all $l_n$ in $L_n$ do
8:                  for all $l_m$ in $L_m$ do
9:                      if $l_n.label = l_m.label$ then match $l_n, l_m$.
10:             Remove all matched leaf nodes from $L_n, L_m$.
11:             for all $l_n$ in $L_n$ do
12:                 for all $l_m$ in $L_m$ do
13:                     if $l_n.type = l_m.type$ then match $l_n, l_m$.
```

5) Leaf Node Matches: The last step of matching is identifying leaf node matches. Since previous matching steps do not match leaf nodes except for exact matching which updates all descendants of two matched nodes, remaining leaf nodes should be matched. Algorithm 5 shows the leaf node matching step. LAS first matches leaf nodes of each matched node pair based on the labels of leaf nodes (line 7-9). Some of the leaf nodes may not be matched due to updated values such as modifier changes or variable name changes, which are reflected in the label of leaf nodes. In this case, LAS examines unmatched leaf nodes from left to right, and matches two leaf nodes if they have the same AST node type (line 11-13).

C. Edit Script Generation

After finding node matches between two ASTs parsed from a change, LAS generates an edit script for the change based on matching results. Each edit script consists of AST subtree level edit operations indicating AST subtree changes in the two ASTs. To generate a subtree level edit operation, LAS generates and combines AST node level edit operations representing changed AST nodes in each changed AST subtree.

There are four different types of subtree level and node level edit operations. The four node level edit operation types generated by LAS are described below.

- **insert** n, p, i: insert node n as i-th child of node p.
- **delete** n, p, i: delete i-th child n from node p.
- **move** n, p, i: move node n from its original location ($n.parent$) to node p as p's i-th child.
- **update** n_1, n_2: update node n_1 to n_2, only if n_1, n_2 have different values.

Each type of edit operation is represented by the operation type, related nodes and the location. For instance, in an edit operation "*insert* n, p, i", *insert* is the operation type, n, p are the related nodes, and i is the location of a change. One exception is *update* operation, which indicates that a value is updated in the same node.

A subtree level edit operation is represented by a tree of node level edit operations. For a subtree level *insert, delete, move* operation, all node level edit operations op_i have the same operation type, and all changed nodes n_i of op_i belong to the same AST subtree. Since the subtree level edit operation is constructed based on the connectivity of nodes n_i, two trees formed by n_i and operations op_i are isomorphic. Similar to node level edit operations, *update* operation type is the only exception, which always has one root *update* edit operation.

Algorithm 6 Edit Script Generation

```
1:  procedure GenerateEditScript($T_{old}$, $T_{new}$)
2:      $stack$: an empty stack for tree construction.
3:      $script \leftarrow$ GenerateDelete($T_{old}.root$, $stack$)
4:      $stack \leftarrow empty$
5:      $script$ += GenerateIMU($T_{new}.root$, $stack$)
6:      $script$ += GenerateOrderChange($T_{old}.root$)
7:  procedure GenerateOrderChange($n$)
8:      $op \leftarrow$ empty
9:      if $n.match \neq null$ then
10:         $oldNodes \leftarrow n.children$
11:         $newNodes \leftarrow n.match.children$
12:         $LCS \leftarrow$ LCS of $oldNodes, newNodes$
13:         for all $c$ not in $LCS$ do
14:             if $c.match \neq null \&\& c$ is not moved then
15:                 $m \leftarrow c.match$
16:                 $mov \leftarrow$ move $c, m.parent, m.index$
17:                 $op$ += $mov$
18:     for all $child$ in $n.children$ do
19:         $op$ += GenerateOrderChange($child$)
        return $op$
```

Algorithm 6 provides the overall edit script generation algorithm of LAS. First, LAS traverses the old AST T_{old} and generates *delete* operations with Algorithm 7 (line 3). Then LAS generates *insert, move, update* operations with Algorithm 8 by visiting the new AST T_{new} (line 5). Finally, LAS identifies node order changes based on the difference between indices of matched nodes (line 6-19).

We first explain about order change operations represented

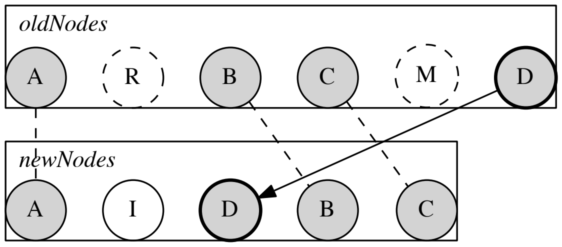


Fig. 4: Order change generation example. Edges between *oldNodes* and *newNodes* represent matched nodes. Node D is identified as moved from the last to next to node A. Node R indicates a removed node, node M means a moved node, and node I represents an inserted node.

by *move* edit operations. Fig. 4 shows an example of order change operation generation. $oldNodes$ and $newNodes$ show child nodes of a given node n and its match $n.match$ (line 9-11). Node A, B, C, D are matched nodes, and node R, M, I are unmatched nodes which are deleted, moved, and inserted respectively. Since indices of matched nodes B, C, D are changed, it is possible that the order of these nodes are changed. We can consider this example as nodes B and C are moved next to node D, or node D is moved before node B. To find nodes whose order is changed, LAS first identifies the longest common subsequence (LCS) of matched nodes between $oldNodes$ and $newNodes$ (line 12). In the example, LCS is nodes A, B, C. LAS considers these nodes unchanged, although their indices might be changed. Nodes R, M, I are ignored since they are changed nodes, and edit operations for these nodes are already generated by previous steps (line 1-5). Then the remaining node D, is the only node whose order is changed. LAS generates a move operation for each node identified as its order is changed (line 13-17). LAS repeats this step for all the children of the given node n too (line 18-19).

Algorithm 7 Generate Delete Operations

```
1:  procedure GenerateDelete($n$, $stack$)
2:      $op \leftarrow$ empty
3:      if $n.match = null$ then
4:          $del \leftarrow$ delete $n, n.parent, n.index$
5:          if $stack$ is empty then add $del$ to $op$
6:          else attach $del$ to $stack.top$
7:          $stack$.push($del$)
8:      for all $c$ in $n.children$ do
9:          $op$ += GenerateDelete($c$, $stack$)
10:     if $n$ was pushed then $stack$.pop()
        return $op$.
```

Algorithm 7 describes how LAS generates *delete* operations. In the old AST, all unmatched nodes are considered as deleted nodes. LAS visits each node in depth-first manner, and creates a *delete* operation if a node is unmatched (line 3-4). If there is no edit operation in the $stack$, the generated *delete* operation is the root of a new subtree level delete edit operation (line 5). Otherwise, LAS attaches the generated *delete* operation to $stack.top$, which is the edit operation generated for the parent node of the current unmatched node (line 6).

Algorithm 8 Generate Insert,Move,Update Operations

```
procedure GENERATEIMU($n$, $stack$)
    $op \leftarrow$ empty
    if $n$ is matched then
        if parents are different then
            $m \leftarrow n.match$
            $mov \leftarrow$ move $n, m.parent, m.index$
            if $stack.top =$ move then
                attach $mov$ to $stack.top$
            else $op$ += $mov$
            $stack$.push($mov$)
        if $n$ is matched, but labels are different then
            $upd \leftarrow$ update $n, n.match$
            add $upd$ to $op$
    else
        $ins \leftarrow$ insert $n, n.parent, n.index$
        if $stack.top =$ insert then attach $ins$ to $stack.top$
        else $op$ += $ins$
        $stack$.push($ins$)
    for all $c$ in $n.children$ do
        $op$ += GENERATEIMU($c$, $stack$)
    if $n$ was pushed then $stack$.pop()
    return $op$
```

Algorithm 8 shows the detailed process of generating *insert, move, update* operations. While visiting the new AST, LAS tests conditions for matched nodes and generates *move* and *update* operations (line 3-13). If node n has a match, but parent nodes p_1 and p_2 are not matched each other, then it indicates that the node n is moved from p_1 to p_2 (line 4-10). Also, if node n is matched, but labels of node n in two ASTs are different, then this means that the value of the node n is updated (line 11-13). Note that LAS matches nodes only if they have the same node type, and each label consists of the node type and value. Therefore a matched node with changed labels must have an update in its value. In addition, if LAS meets an unmatched node in the new AST, this node is considered inserted and an *insert* operation is generated for the node (line 14-18).

III. EVALUATION

In this section, we first present our research questions, and we explain our evaluation methods to address the questions.

A. Research Questions

- **How well does LAS represent human's perception about changes?** The result of change mining and analysis with a source code differencing algorithm is highly dependent to the quality of edit scripts generated by the algorithm. If a generated edit script does not match to human's perception about a change, it may lead to a wrong conclusion of an analysis. Therefore we evaluate how well edit scripts of LAS represent human's perception about changes.
- **How efficiently does LAS generate edit scripts?** For change mining task, execution time is an important factor to evaluate the performance of a source code differencing algorithm and its implementation. Even a small difference in execution time to analyze one change could be critical if we want to analyze millions of changes. Also, if an algorithm has high time complexity, it might show unreliable runtime performance when it is applied to large source code. Hence we evaluate the runtime performance of LAS with its execution time.

B. Experimental Setting

TABLE I: Subject information collected for evaluation.

Project	First Rev.	Last Rev.	Revisions	Changes
derby	22ff5fc	45a4a1d	149	90,124
jdt-core	23e866c	0ed94a3	87	2,862
lucene	8a3eb50	afd960c	496	341,134
math	b3c5dae	5f9cfa6	266	737
rhino	a4a1f13	0e0cf58	82	425
Total			1,080	435,282

Table I shows SHA reference of the first and the last revisions we used, the number of revisions and the number of changed files for each project. For evaluation of LAS, we first collected 435,282 changes from the five different Java open source projects. From a source code repository of each project, we identified all changed Java files from the latest release to the head revision. We consider one changed file as one source code change, since the implementations of source code differencing algorithms usually take two files from old and new versions as input. Collected changes include various aspects of source code changes such as implementing new features, adding new tests and system-wide changes due to refactoring.

Since LAS and GumTree implementations both have parameters to control the execution, we used our default settings (distance=0.5, depth=3, similarity=0.65) for LAS and the default settings (min. height=2, similarity=0.3, max. subtree size=1000) recommended in the deploying website for GumTree [1].

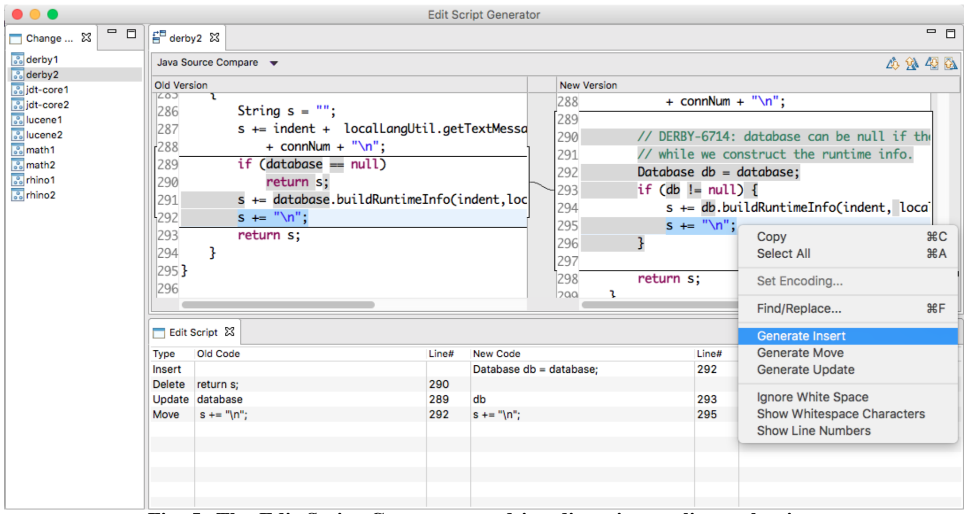
We executed both tools on a server having 2.67GHz Intel Xeon CPU and 64GB RAM and Linux installed.

C. Edit Script Quality

To evaluate the quality of generated edit scripts with respect to human's perception about changes, we propose a new systematic method instead of simply asking people's opinion about two different edit scripts. We first provide source code changes to participants, and ask them to describe changes with the same set of edit operations. In this way we can have two types of edit scripts for the same change, one written by human and one generated by a source code differencing algorithm. Then we can compare these two edit scripts to evaluate how well the generated edit script of a change represents human's perception about the change.

Participants may not be familiar to understand changes in AST node level. To reflect human's perception about changes more correctly, we provide an edit script generator as shown

[1]https://github.com/GumTreeDiff/gumtree/wiki/Core-options



```
286         String s = "";
287         s += indent +  localLangUtil.getTextMessa
288             + connNum + "\n";
289         if (database == null)
290             return s;
291         s += database.buildRuntimeInfo(indent,loc
292         s += "\n";
293         return s;
294     }
295 }
296
```

```
288             + connNum + "\n";
289
290         // DERBY-6714: database can be null if th
291         // while we construct the runtime info.
292         Database db = database;
293         if (db != null) {
294             s += db.buildRuntimeInfo(indent, loca
295             s += "\n";
296         }
297
298         return s;
```

Type	Old Code	Line#	New Code	Line#
Insert			Database db = database;	292
Delete	return s;	290		
Update	database	289	db	293
Move	s += "\n";	292	s += "\n";	295

Fig. 5: The Edit Script Generator used in edit script quality evaluation.

in Fig. 5. With this tool, participants can select changed code fragments in textual form, and generate four different types of edit operations same as LAS and GumTree do. Although we explained about the four types of edit operations, we did not constrain their usage much to catch participants' natural understanding about changes. Instead, we processed participants' edit operations later to match usage of edit operations. For example, in LAS and GumTree, move operations only used for the same code fragments in two source code versions since the same code fragment is moved. Also update operations are only used for leaf nodes, since this is originally used for value updates such as variable name change or literal value changes. If participants used such move and update edit operations in a different way, we simply converted them to one deletion from the old version and one insertion in the new version, which LAS and GumTree are able to generate.

Also, participants may write different edit scripts for the same change, and it is difficult to decide which edit script is better. To resolve this issue, we assign higher score for edit scripts written by more participants. For example, suppose there are two scripts, one is written by 6 out 10 participants and the other is written by four participants. We assign 0.6 for the first edit script, and 0.4 for the other edit script. Equation 3 shows how we compute the score of a human written edit script.

$$
\begin{aligned}
&S.score = \frac{|P_S|}{|P|}\\
&\text{where } S = \text{ a human written edit script},\\
&P = \{p \text{ is a participant}\},\\
&P_S = \{p : p \in P, p \text{ wrote an edit script same as } S\}
\end{aligned}
\tag{3}
$$

It is possible that source code differencing algorithm may not generate exactly the same edit script as human does. We compute the similarity of two edit scripts with the number of common edit operations in the edit scripts divided by the number of unique edit operations in the edit scripts. To compute the score of a generated edit script, we find the most similar human written edit script and multiply the score of the found edit script by the similarity of two edit scripts. Then divide the computed score by the maximum score of human written edit script to scale the score within range from 0.0 to 1.0.

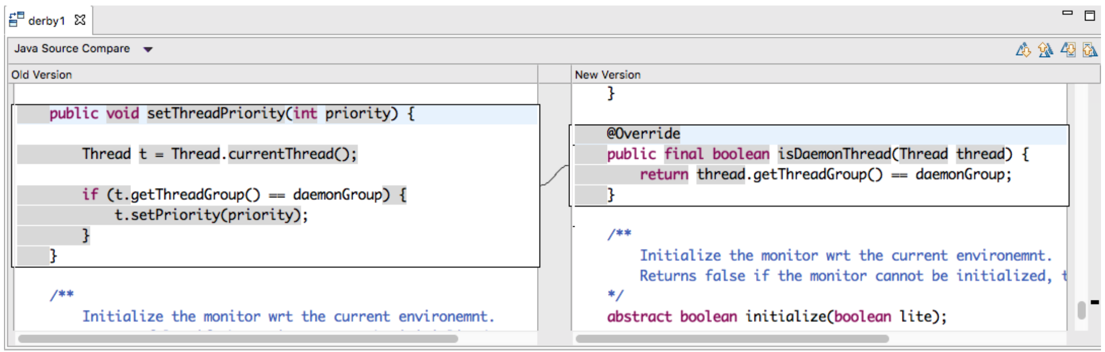
$$
\begin{aligned}
&score(S_1) = \frac{S_2.score * similarity(S_1, S_2)}{S.max}\\
&\text{where } HS = \{\text{human written edit scripts}\},\\
&GS = \{\text{generated edit scripts}\},\\
&S_1 = \{\text{a set of edit operations}\}, S_1 \in GS,\\
&S_2 = \underset{S \in HS}{\arg\max}\, similarity(S_1, S),\\
&S.max = \max_{S \in HS}(S.score),\\
&similarity(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}
\end{aligned}
\tag{4}
$$

Equation 4 shows how to compute the score of a generated edit script. In this way, we can assign a higher score to a generated edit script which is more similar to an edit script written by more participants.

For the quality evaluation, we recruited five graduate students and two professional software engineers, and asked them to describe 10 source code changes with the edit script generator we provided. These 10 changes were randomly chosen from changes of five open source Java projects (two changes for each project) collected for runtime performance evaluation shown in TABLE I.

Fig. 5 shows our edit script generator implemented as an Eclipse RCP application. The changes are provided within the edit script generator. When a participant opens a change, it shows Java source compare editor with highlighted change parts computed by Eclipse's compare algorithm, to help participants recognize changed parts. Participants can select changed code fragments, and use context menus to generate edit operations based on their opinion.

Once we collected edit scripts written by participants with our edit script generator, we converted human written edit scripts to the same formats which are used by LAS and GumTree respectively. After that, we generated edit scripts for the same changes with LAS and GumTree, and computed scores by Equation 4.



```
    public void setThreadPriority(int priority) {

        Thread t = Thread.currentThread();

        if (t.getThreadGroup() == daemonGroup) {
            t.setPriority(priority);
        }
    }

    /**
        Initialize the monitor wrt the current environemnt.
```

```
    }

    @Override
    public final boolean isDaemonThread(Thread thread) {
        return thread.getThreadGroup() == daemonGroup;
    }

    /**
        Initialize the monitor wrt the current environemnt.
        Returns false if the monitor cannot be initialized, t
    */
    abstract boolean initialize(boolean lite);
```

Fig. 6: Change *derby1* opened in Edit Script Generator. Differences are provided and highlighted by Eclipse's Java comparison.

IV. Results

In this section, we present our evaluation results obtained by experiments described in Section III and discuss about their implications.

A. Individual Edit Script Quality

TABLE II: LAS and GumTree's edit script scores and similarities computed by Equation 4. Similarities shown in brackets. Score Diff. column indicates the difference of LAS and GumTree's scores and shown in bold when LAS has a higher score.

Change	LAS (Sim.)	GumTree (Sim.)	Score Diff.
derby1	0.8462 (0.8462)	0.1418 (0.7089)	**0.7044**
derby2	0.8889 (0.8889)	0.6500 (0.6500)	**0.2389**
jdt-core1	0.8649 (0.8649)	0.7317 (0.7317)	**0.1332**
jdt-core2	0.9333 (0.9333)	0.8718 (0.8718)	**0.0615**
lucene1	0.7407 (0.7407)	0.8462 (0.8462)	-0.1054
lucene2	0.2222 (0.6667)	0.8571 (0.8571)	-0.6349
math1	0.5000 (1.0000)	0.5000 (1.0000)	0.0000
math2	0.9714 (0.9714)	0.9592 (0.9592)	**0.0122**
rhino1	0.1620 (0.8101)	0.2391 (0.9565)	-0.0771
rhino2	1.0000 (1.0000)	1.0000 (1.0000)	0.0000

Table II shows the scores of edit script generated by LAS and GumTree for the 10 changes. LAS and GumTree columns present scores of LAS and GumTree edit scripts respectively, and provides similarities to the best matched human-written scripts in brackets. Score Diff. column shows the difference of LAS and GumTree scores. For 9 out of 10 changes, LAS and GumTree generated different edit scripts. For the remaining change (*rhino2*), two techniques generated the same edit script, and it was also same as the human written edit scripts.

Among the 10 changes, LAS generated edit scripts more similar to human-written edit scripts for five changes which the differences are shown as bold. For these changes, all LAS edit scripts are at least 74% similar (*lucene1*) to the human-written script supported by the most participants. Note that we set the score of the most supported human-written script to 1.0 and penalize it based on similarity of generated edit scripts. Therefore if a generated edit script has the score identical to the similarity shown in brackets, it means that this script is most similar to the human-written script which the most participants agreed with. In case of GumTree, it also generates similar to the most supported human-written scripts for 4 out of 5 changes, but the similarities are lower than LAS edit script's similarities. This indicates that LAS can generate edit scripts which are more close to changes in participants' perception.

We manually investigated the difference between LAS and GumTree scripts, and it is mainly due to GumTree's excessive matching of AST nodes. In human-written scripts, participants tend to consider a whole code fragment is changed although there are some small parts which may be matched by the algorithms. For example, Fig. 6 is a screen capture of our edit script generator for change *derby1*. Grey highlighted area is changed parts provided by Eclipse's basic Java comparison. Although there exists some unchanged part in two methods `setThreadPriority()` and `isDaemonThread()`, 5 out of 7 participants described this change as the whole method on the left is deleted and the whole method on the right is inserted. However, GumTree tries to match as much AST nodes as it can. As a result, some of code fragments such as `Thread`, `getThreadGroup()`, and `deamonGroup` are considered moved from one method to the other method in GumTree's edit script. On the other hand, LAS generates an edit script with the left method deleted and the right method inserted. The only difference between LAS edit script and a matched human-written script is that LAS generates extra edit operations representing that the condition of the if statement `t.getThreadGroup() == daemonGroup` is moved to the return statement on the right method, and the variable name `t` is updated to `thread`. Although LAS generates a slightly different edit script from the human-written script, this small difference does not blur the human's understanding about this change much.

There are 3 out of 10 changes which GumTree performs better than LAS. For 2 out of 3 cases (*lucene2, rhino1*), LAS edit scripts are more matched to human-written scripts other than the most supported one. These changes contain small changed parts such as inserting type parameters in a parameterized type, but LAS considers the whole parameterized type is changed. This result reminds us that identifying changes in more large fragment may not surgically represent such small, fine-grained changes. However, it does not mean that LAS edit scripts for these changes are completely wrong, and they are still similar to one of the human-written scripts at least 67%.

For the remaining two changes, LAS and GumTree edit scripts are identical to one of the human-written scripts. In case of *rhino2*, all generated edit scripts and human-written scripts are identical, since the change is very simple which inserts a new field. For *math1*, LAS and GumTree generated different edit scripts which are identical to two different human-written scripts. For this change, participants wrote slightly different scripts since the change is quite complex.

Only two participants wrote the same script, and both LAS and GumTree failed to generated an edit script identical to this script. Hence the scores of edit scripts are both 0.5, although each edit script is identical to one of the human-written script. However, in this case, we need to interpret this result as the two generated edit scripts have the same quality, rather than both edit scripts have low quality due to the low score.

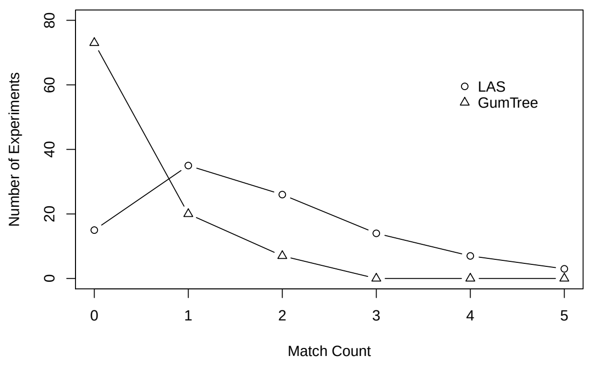
B. Change Statistics

TABLE III shows top 10 most frequent edit operations identified by Human, LAS and GumTree in each column. We aggregate individual edit operations in edit scripts of all 500 changes of the benchmark. Edit operations which insert, delete or move import declarations are excluded, since these edit operations are usually dependent to other changes and often not meaningful for change mining tasks.

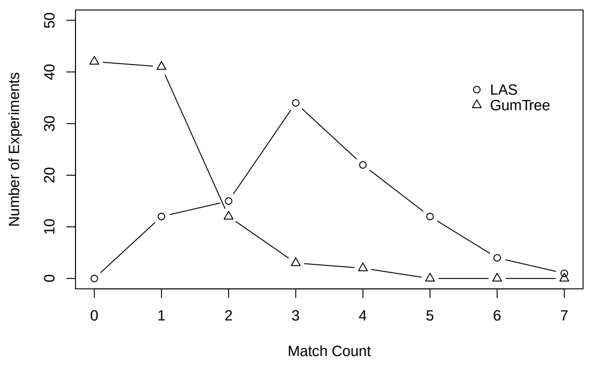
Overall, edit operation ranks of LAS is more similar to Human edit operations compared to GumTree edit operations. Among top 10 most frequent Human edit operations, eight of them present in top 10 LAS edit operations, while only two of them are shown in top 10 GumTree edit operations. One of the eight LAS edit operations has exactly same rank as Human edit operations (bold font), and four of them have similar rank which are no more than one rank different to original ranks of Human edit operations (marked with *). Remaining three edit operations have different ranks from Human edit operations (marked with **), but at least LAS does not miss these three operations and contains them in the top 10 list.

These results indicate that using LAS provides more precise change statistics and conclusion based on such statistics will be more accurate and convincing. If we use LAS, we can identify a half of top 10 most frequent changes without much distortion of its meaning. We can also identify 80% of the top 10 frequent changes, while using GumTree misses 80% of the top 10 frequent changes. Although LAS is not perfect, it is obvious that using LAS is a better choice to obtain accurate results for change mining studies.

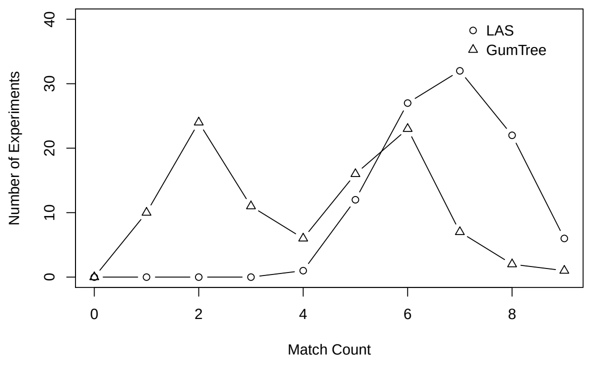
Fig. 7 shows box plots obtained from 100 rank comparison evaluation. From the benchmark of 500 changes, we randomly select 200 changes and identify top 10 most frequent edit operations from human, LAS and GumTree edit scripts. Then we compare ranks of the top 10 edit operations in the different edit scripts, and categorize them to three groups shown in X-axis: Exact, Similar, Appear. Exact group indicates that an edit operation has the same rank in both human edit scripts and generated edit scripts. Similar group contains edit operations with rank difference no more than 2. For instance, for one edit operation of rank 3 in human edit scripts, if the same edit operation has rank 1 to rank 5, it is included in similar group. Appear group counts all edit operations in human edit scripts which are also appeared in edit scripts generated by LAS or GumTree. This means that if an edit operation is appeared in top 10 edit operations of both human and LAS edit scripts, this edit operation is counted regardless of its rank. Y-axis of each plot indicates the number of edit operations in each group. Overall, LAS significantly outperforms GumTree. For all six cases of three different groups and with or without import and method declarations, LAS tends to have more matched top 10 edit operations.



(a) Exact Cases



(b) Similar Cases



(c) Present Cases

Fig. 7: Numbers of Exact, Similar, and Present cases in 100 repeated experiments.

V. DICUSSION

In this section, we further discuss about LAS and its evaluation. First, we analyze time complexity and runtime performance of LAS. In addition, we discuss about the effectiveness of each matching step which significantly affects the performance. Lastly, we present the characteristics of human-written scripts observed during the qualitative evaluation.

TABLE III: Top 10 most frequent edit operations from Human, LAS and GumTree edit scripts for 500 changes in the benchmark.

Rank	Human	LAS	GumTree
1	**update NumberLiteral**	**update NumberLiteral**	move SimpleName
2	update SimpleType	move MethodInvocation	update SimpleName**
3	delete MethodDeclaration	update SimpleType*	insert SwitchCase
4	insert MethodDeclaration	insert MethodInvocation*	move SimpleName
5	insert MethodInvocation	delete MethodDeclaration**	insert BreakStatement
6	insert IfStatement	delete MethodInvocation**	delete SwitchCase
7	delete Modifier	update SimpleName*	delete BreakStatement
8	update SimpleName	delete Modifier*	update QualifiedName
9	insert ParameterizedType	insert MethodDeclaration**	move MethodInvocation
10	delete MethodInvocation	delete VariableDeclarationStatement	update SimpleType**

A. Time Complexity and Runtime Performance

1) Time Complexity: The asymptotic worst-case time complexity of LAS for two ASTs $T_1, T_2, n = max(|T_1|, |T_2|)$ is $O(n^2)$. LAS builds two ASTs and assigns hash strings and node type vectors in $O(n)$, since it only requires visiting AST nodes in each tree. In node matching step, the worst case is that LAS compares each node in one AST to all nodes in another AST for each different matching step due to large distance and depth thresholds. Each comparison takes a constant time since it uses hash strings or fixed size node type vectors. Therefore the whole matching step requires $O(n^2)$. In edit script generation step, LAS traverses each AST once to generate edit operations and visits two ASTs again to find ordering changes, which takes $O(n)$ in total. Therefore the overall time complexity is $O(n^2)$.

2) Runtime Performance: Although we analyzed time complexity, it only represents the worst case scenario, and it is possible that only a few changes belong to such worst cases. Therefore it is necessary to verify that LAS shows reliable performance when it is applied to various changes.

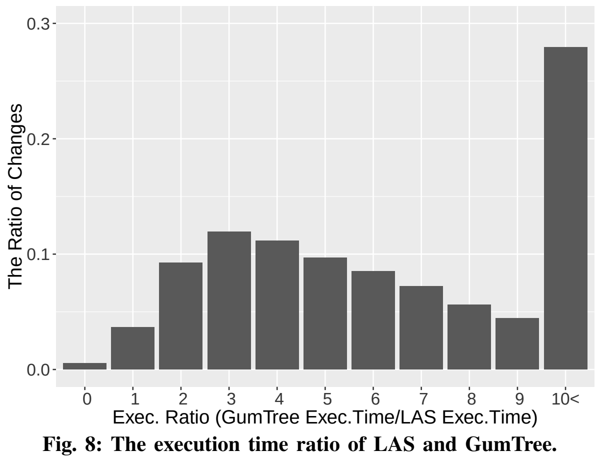


Fig. 8: The execution time ratio of LAS and GumTree.

Fig. 8 presents the differences in average execution time of LAS and GumTree for individual changes.

We first divide the execution time of GumTree by the execution time of LAS for each change to compute execution time ratio. Then we group changes based on their execution time ratio, to represent how many times LAS is faster than GumTree for how much of the changes. For example, the third bar in Fig. 8 indicates that LAS is at least two times faster, but less than three times faster for about 10% of the changes.

For individual changes, LAS shows more improved runtime performance than GumTree. For 27.93% of 435,282 changes, LAS is at least 10 times faster than GumTree in generating edit scripts. For most of the changes (95.78%), LAS generates an edit script at least twice faster than GumTree. GumTree shows better performance only for 0.56% of the changes. These results imply that LAS is contantly more efficient regardless of processed changes, although its performance varies for each change.

B. The Effectiveness of Matching Steps

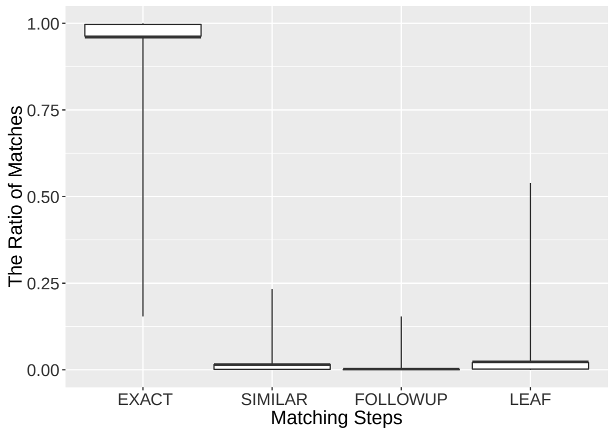


Fig. 9: The ratio of matches found in each matching step.

Fig. 9 shows how much of node matches are found in each matching step. To find changes between two ASTs, LAS first identifies node matches between the ASTs through four different matching steps. For each collected change, we counted the number of matches found in each matching step. Then we computed the ratio of the matches found in each step to the total number of matches found. Such ratios for all changes used in our evaluation is shown as box plots in Fig. 9.

The result indicates that the exact matching step of LAS effectively filters out unchanged parts in given ASTs. More than 99% of the node matches were found in exact matching step. This proves the assumption for our design that changed parts are much smaller than unchanged parts and thus we need to quickly rule out unchanged part. Note that LAS only compares hash strings for exact matching and updates matches for AST subtrees within a certain range. In this way, LAS can filter out a large unchanged portion and apply more sophisticated method for actually changed parts.

C. Characteristics of Human-written Scripts

As we explained in Section IV-A, we did not strictly constrain the usage of four edit operations when we collected human-written scripts. Therefore we can observe differences of human-written edit scripts compared to automatically generated edit scripts for the same changes. We can use these observations to improve source code differencing algorithm and design of edit scripts.

In collected human-written scripts, participants tend to describe changes in a whole code fragment. They ignore small similar parts of source code if changed parts of source code are relatively large. Change *derby1* in Fig. 6 is a good example of this case. Although there exist some similar code fragments, participants consider them as separated code fragments which are accidentally identical. If a source code differencing algorithm only considers the similarity of AST subtrees from a change, it may mark similar code fragments unchanged, but these code fragments may actually be changed in human's perspective. Therefore this finding indicates that it is not meaningful to blindly generate smaller edit scripts.

We also observed that many of the participants used update operations as replace operations. Update operations are originally used for leaf node changes to represent updates of values in LAS and GumTree. However, participants used an update operation to describe one code fragment is replaced by another code fragment. Such update operations can be broken into one insertion and one deletion, but the connection between a deleted code fragment and an inserted code fragment will be lost in two separated operations. Therefore, to describe changes better with respect to human's perspective, we need to consider employing replace operations in edit scripts.

VI. Threats To Validity

One major threat to validity of this study is our experimental design for the evaluation.

First of all, we used the default values for LAS and GumTree's runtime parameters. Since these parameters control differencing processes of the two tools, evaluation results would be different if we used different parameters. However, the default parameters we used for GumTree were recommended by its authors. In a situation we use these source code differencing tools, we cannot know which parameters will show the best performance for a given set of changes. It is highly likely that running GumTree with the default parameters shows better performance than using other values as parameters for the majority of changes. Therefore using default parameters for the evaluation was a reasonable choice.

It is also possible that our measurements for runtime performance suffer from noises. However, we applied LAS and GumTree 10 times to each change and computed average time consumption to avoid any bias or accidental noise. Also, the runtime performance difference is quite huge, and the implications of the results would not be changed even if there were some noises. For the confirmation, we repeated the whole runtime performance evaluation process at least twice, and obtained almost same results for every attempt.

Another threat to validity of this study is our qualitative evaluation of generated edit scripts. Since participants are not familiar with source code differencing studies and edit scripts, human-written scripts used in this study may not represent the human's perspective about the given changes properly. However, we provided a prototype tool for edit script generation, and the participants were asked to describe changes by using the tool in textual format with changed parts of source code highlighted as shown in Fig. 5. We also presented simple descriptions and instructions about how to use this tool to generate edit scripts. For 6 out of 10 changes, at least three participants described them with the exactly same edit scripts. Therefore the human-written scripts used in the evaluation actually describe the given changes looked in human's eyes.

Our choice of the subjects is also one of the threats to validity. We carefully selected Java software projects of various sizes, functionalities, and also managed by different groups. However, they are all open source software projects, and these projects may not be representative. The choice of the changes used in the qualitative evaluation also has the same issue. In spite of this limitation, LAS continuously shows promising performance for the majority of the changes, and the implications of our results will not be significantly changed even if we consider such limitation in subject selections.

VII. Related Work

One line of work which can be directly compared to LAS is differencing ASTs. ChangeDistiller [10] is an AST differencing algorithm which generates an edit scripts with various edit operations in statement-level. ChangeDistiller detects statement-level changes to support understanding changes, but it uses a slightly abstracted version of ASTs and sometimes miss changes smaller than statement-level. GumTree [8] is a fine-grained AST differencing algorithm shows excellent runtime performance. GumTree solves the issue of missing changes in ChangeDistiller by detecting AST node level changes, and it accurately generates edit scripts describing source code changes. However, GumTree focuses on generating smaller edit scripts for a change, which can pin-point changed parts of ASTs, and it may lead to edit scripts which are different from human's conception about the change. We designed a new evaluation method of edit scripts' quality, and found that LAS can generate edit scripts more similar to human's conceptual changes than GumTree.

Another method to detect changes is text differencing. Text differencing techniques can identify differences in lines of source code text [20], [19]. Also, there exist more advanced differencing techniques designed for tracking source code evolution [2], [24], [3]. These techniques are very efficient, and more importantly language independent since change detection is based on text. However, these techniques may not be suitable for change mining since text formatting differences can cause incorrect change detection and it is difficult to handle such incorrectness individually when we try to collect a large amount of changes. Also, text line level change detection often provides coarse-grained changes since each group of changed lines may contains many unchanged code fragments.

We may also adapt tree differencing algorithms to AST differencing. Some tree differencing algorithms based on tree distance [26], [29], [25], which is similar to source code fingerprinting techniques we used in LAS. Other tree differencing algorithms [5], [27], [7], [4], [1] are designed for structured document differencing such as LaTeX or XML, which are closely related to source code differencing. Since these tree differencing algorithms highly focus on the performance, we may benefit from such high performance when we handle a large amount of changes. However, there still remains the same qualitative issue discussed in Section IV-A, which means that these algorithms are not designed for source code, hence detected changes may not described mined changes correctly.

There exist other studies which can benefit from a new AST differencing algorithm particularly designed for change mining. There have been studies which tried to discover change patterns by analyzing edits identified by an AST differencing algorithm [9], [17]. Zhong et al. also used a source code differencing technique to identify repair actions in their large scale study on real bug-fixes collected from Apache Software Foundation projects [30]. These studies require analyzing a large amount of changes, and LAS can be useful for these studies since it is particularly designed for mining changes. Martinez and Monperrus built a repair model based on the change and entity types supported by an AST differencing algorithm, and evaluated the models with bug fixes mined from software repositories. [18]. Some of the automatic program repair techniques use a small number of pre-defined change templates [13], [15], [16] or several mutation operators to modify source code [14], [22], [28]. The proposed repair model can augment these templates by mining changes from human-written bug-fixes, and LAS can support such change mining task very efficiently.

VIII. Conclusion

In this paper, we have introduced LAS, a location aware source code differencing algorithm for mining changes. LAS uses multiple node matching steps to consider nodes' location to find more appropriate node matches and generates edit scripts more similar to human's perception about changes. For qualitative evaluation of edit scripts with respect to human's perception about changes, we propose a new systematic method comparing generated edit scripts with human's scripts independently written for the same changes. Our evaluation results show that LAS generates edit scripts more similar to human-written edit scripts, and it also generates edit scripts much faster than the other compared differencing technique. We also analyze human-written scripts collected for qualitative evaluation and found the characteristics of human-written scripts. We believe that these characteristics can help us to improve current source code differencing algorithms, and improving LAS with this information and developing new applications of LAS will be our future work.

References

[1] R. Al-Ekram, A. Adma, and O. Baysal. diffX: An Algorithm to Detect Changes in Multi-version XML Documents. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 1–11. IBM Press, 2005.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 230–239, Washington, DC, USA, 2013. IEEE Computer Society.

[3] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE Softw.*, 26(1):50–57, Jan. 2009.

[4] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Rec.*, 26(2):26–37, June 1997.

[5] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[6] M. Chilowicz, E. Duris, G. Roussel, and U. Paris-est. Syntax tree fingerprinting: a foundation for source code similarity detection, 2009.

[7] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 41–52. IEEE, 2002.

[8] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[9] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 463–466, Washington, DC, USA, 2008. IEEE Computer Society.

[10] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, Nov 2007.

[11] M. Gabel and Z. Su. A Study of the Uniqueness of Source Code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering.*, pages 96–105, 2007.

[13] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, 2013.

[14] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[15] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[16] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[17] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 388–391, Washington, DC, USA, 2013. IEEE Computer Society.

[18] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[19] W. Miller and E. W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[20] E. W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[21] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 180–190, Nov 2013.

[22] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[23] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The Uniqueness of Changes: Characteristics and Applications. Technical report, Microsoft Research Technical Report, 2014.

[24] S. P. Reiss. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 11–20, New York, NY, USA, 2008. ACM.

[25] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581 – 621, 1990.

[26] G. Valiente. An efficient bottom-up distance between trees. In *Proceedings Eighth Symposium on String Processing and Information Retrieval*, pages 212–219, Nov 2001.

[27] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530. IEEE, 2003.

[28] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[29] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

[30] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.