Move-Optimized Source Code Tree Differencing

Georg Dotzler, Michael Philippsen
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
Programming Systems Group
{georg.dotzler, michael.philippsen}@fau.de

ABSTRACT

When it is necessary to express changes between two source code files as a list of edit actions (an edit script), modern tree differencing algorithms are superior to most text-based approaches because they take code movements into account and express source code changes more accurately.

We present 5 general optimizations that can be added to state-of-the-art tree differencing algorithms to shorten the resulting edit scripts. Applied to Gumtree, RTED, JSync, and ChangeDistiller, they lead to shorter scripts for 18–98% of the changes in the histories of 9 open-source software repositories. These optimizations also are parts of our novel Move-optimized Tree DIFFerencing algorithm (MTDIFF) that has a higher accuracy in detecting moved code parts. MTDIFF (which is based on the ideas of ChangeDistiller) further shortens the edit script for another 20% of the changes in the repositories.

MTDIFF and all the benchmarks are available under an open-source license.

CCS Concepts

•Software and its engineering → Software maintenance tools; •Mathematics of computing → Trees;

Keywords

Tree Differencing; Optimizations; Source Code

1. INTRODUCTION

In many disciplines, it is common to express the changes between two character sequences as a list of edit actions, a so-called edit script. It often matters to find the shortest possible edit script, for instance between RNA structures in biology [4, 27]. When software developers examine changes in source code, it is also likely that the shortest edit script is the most beneficial to grasp the nature of a change. According to Falleri et al. [18], the length of the edit script is a proxy to the cognitive load for a developer to understand the essence of a commit (and thus file changes). It is also likely that code change recommendation tools [28, 29] can produce better results if they have access to the shortest edit scripts. Unfortunately, finding them is NP-hard [38] if the movement of subsequences (e.g. substrings or subtrees) is a permitted edit action.

To determine source code differences quickly, *diff* and its variants solve a simplified problem. First, they do not consider moved code. The only permitted edit actions are insertions and deletions of code lines. Second, they use Myers algorithm [31] to detect changes on the granularity of full text lines. Thus, even if a change only affects a part of a line (e.g., renames a variable), the full line is deleted and a new line is inserted. Both simplifications make the output of *diff* unnecessarily verbose due to a sub-optimal edit script.

Tree differencing approaches avoid the above granularity problem as a code line usually consists of a number of nodes in the abstract syntax tree (AST). The resulting edit scripts no longer mention unmodified parts of code lines. But since the general problem of finding the shortest edit script on ASTs with *moves* remains NP-hard [9, 10], state-of-the-art tree differencing algorithms also need to simplify the problem. RTED [33], for example, does not consider moved subtrees. Other state-of-the-art tree differencing algorithms that consider move-actions cannot be fast *and* find an optimal fine-grained edit script. They use different heuristics to match the requirements of their specific use cases.

ChangeDistiller (CD) [19] uses coarse-grained AST nodes that represent larger structures of the code, e.g., full loop conditions or full statements. The differences on the resulting (much smaller) trees are sufficient for CD's purpose to categorize and identify types of changes. CD's edit scripts are short, but the arguments of the individual edit actions are long, sometimes as long as in *diff*. JSync [32] is used in a clone detector framework. Thus, the algorithm is aimed at changes in methods and not at changes in complete files. Hence JSync leaves out irrelevant aspects from the ASTs, such as imports, fields, etc., and does not work on full files. Gumtree (GT) [18] works on fine-grained ASTs and full code files. For an upper bound on the runtime complexity ($O(n^2)$ for n AST nodes) GT finds a short edit script.

The goal of this paper is to achieve shorter fine-grained edit scripts that use move actions more frequently, even if the computation takes slightly longer than for GT or *diff*.

We analyzed the four tree-differencing algorithms from above to better understand why their edit scripts are longer than necessary. From this analysis, we developed 5 general optimizations. When they are added as pre- or post-

ASE'16, September 3–7, 2016, Singapore, Singapore
ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970315

Original code:

```
01: foo( xy + 2);
02: int a = 0;
03: int b = 1;
04: log();
05: if ( cond)
06:   m1(d, c);
07: log();
08: print( "42!" + "hello");
09: m2( 1 + 2, 3 * 4);
```

Modified code:

```
foo( xyz + 2, b);
int b = 1;
int a = 0;
log();
if ( cond != null)
  m1( 5, c, d);
log();
print( "hello" + "42!");
m2( 3.1 * 4, 1.1 + 2);
```

(a) GT takes 18 edit actions ≙ 18 boxes.

```
01: foo( xy + 2);
02: int a = 0;
03: int b = 1;
04: log();
05: if ( cond)
06:   m1( d, c);
07: log();
08: print( "42!" + "hello");
09: m2( 1 + 2, 3 * 4);
```

```
foo( xyz + 2, b);
int b = 1;
int a = 0;
log();
if ( cond != null)
  m1( 5, c, d);
log();
print("hello" + "42!");
m2( 3.1 * 4, 1.1 + 2);
```

(b) GT with our optimizations takes 15 edit actions.

```
01: foo( xy + 2);
02: int a = 0;
03: int b = 1;
04: log();
05: if ( cond)
06:   m1( d, c);
07: log();
08: print( "42!" + "hello");
09: m2( 1 + 2, 3 * 4);
```

```
foo( xyz + 2, b);
int b = 1;
int a = 0;
log();
if ( cond != null)
  m1( 5, c, d);
log();
print("hello" + "42!");
m2( 3.1 * 4, 1.1 + 2);
```

(c) MTDIFF only needs 12 edit actions.

Figure 1: Code change (left → right). The boxes show the edit actions that the tree differencing algorithms detect: Yellow for updates, green for insertions, red for deletions, blue for node movements.

processing phases to GT, RTED, CD, and JSync, the optimizations shorten their edit scripts. Note that this may not be necessary for the applications for which these tree differencing algorithms were initially developed.

Fig. 1 holds an artificial code change to illustrate the optimizations (original code on the left, modified code on the right). The code is identical in all three sub-figures. The boxes indicate the edit actions. Fig. 1(a) shows that GT produces an edit script of size 18. When the 5 general optimizations are added to GT, they shorten the edit script to the 15 actions shown in Fig. 1(b). The optimizations have similar effects on the other tree differencing algorithms, but due to their different ASTs, the boxes would be different.

Additionally, this work presents a novel Move-optimized Tree DIFFerencing algorithm (MTDIFF) that achieves even shorter edit scripts due to its higher accuracy in detecting moved code parts. It is based on CD and uses the 5 general optimizations. Applied to the same example, MTDIFF requires only 12 edit actions, see Fig. 1(c).

Sec. 2 briefly introduces tree differencing. Sec. 3 describes the 5 general optimizations and Sec. 4 presents MTDIFF. These two sections also address the details of the optimizations that lead to the results in Fig. 1 while Sec. 5 discusses time complexities. Sec. 6 holds both a quantitative evaluation on 126,162 changed files and a study to demonstrate that shorter edit scripts are more helpful to developers than longer ones. Sec. 7 discusses related work.

2. TREE DIFFERENCING

Compilers use ASTs to internally represent source files as rooted, labeled, ordered trees. Fig. 2 shows two examples. Each node has a label *lbl*, such as $Call_o$ and *SimpleName* (for xy_o). Some nodes also have a value attribute v. Fig. 2 uses their values (2_o or xy_o) instead of their labels.

An *edit script* is a list of edit actions that transforms the original AST_o into the modified AST_m. Each edit action has a cost s_i and the cost of an edit script is the sum of all s_i. An edit script is optimal if its cost is minimal. For the remainder of this work, all edit actions have the same cost (e.g. 1). Thus, there is no preference of an edit action. The optimal edit script then is the shortest one.

Most tree differencing algorithms approach the search for the shortest edit script in two phases. Phase 1 uses heuristics to generate a *mapping* between the nodes of the ASTs. The algorithms vary with respect to both the granularity of the AST they use and the heuristics they employ to derive the mapping. A mapping is a list of *matching* pairs of nodes (n_o, n_m), where node n_o from AST_o is mapped to node n_m in AST_m. Each mapped node is part of only one pair. It is a common restriction that only nodes with the same label can match, although this misses a few corner cases (e.g., *while* ↔ *for* with similar conditions). Our optimizations and also MTDIFF share this restriction. Based on the mapping from phase 1, phase 2 computes the edit script in $O(n^2)$ [12, 20]. As the algorithms for phase 2 are optimal [12], a shorter edit script can only result from a different mapping from phase 1. Thus, our work focuses on the heuristics in phase 1.

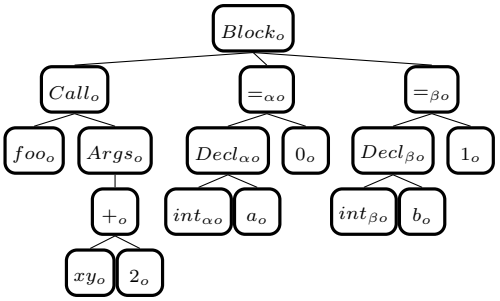
For our evaluation, we use the implementation of GT for phase 2 as it uses an optimal algorithm. This implementation uses 4 edit actions, namely *updateValue*, *add*, *delete*, and *move*. For a pair (n_o,n_m) with different values, GT adds an $updateValue(n_o, v_m)$ action to the script. If a node n_o is not in the mapping, GT adds $delete(n_o)$. If a node n_m is not in the mapping, GT adds $add(n_m, parent(n_m), i)$ where i specifies the i^{th} child of the parent of n_m. If there is a pair (n_o, n_m) in the mapping where $(parent(n_o), parent(n_m))$ is not mapped, GT adds a $move(n_o, parent(n_m), i)$ where n_m is the node mapped to n_o and i specifies the i^{th} child of the parent of n_m. If both (n_o, n_m) and $(parent(n_o), parent(n_m))$ are mapped, GT also adds a *move* action if n_o and n_m have different child-indices in their parents. Note that rather than moving a single node n, *move* affects the complete subtree rooted in n.

3. GENERAL OPTIMIZATIONS

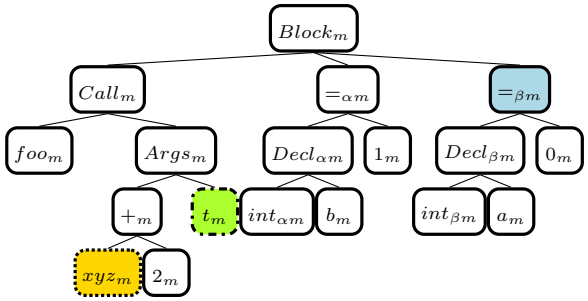
The following optimizations Θ_{A-E} are applicable to all examined tree differencing algorithms. For the discussion of Θ_{A-E} we use the code and ASTs from Figs. 1 and 2. For brevity, Fig. 1 only shows the situation for GT and we only demonstrate Θ_{B-E} on GT. Our evaluation shows that Θ_{A-E} shorten the edit scripts for the other tree differencing algorithms as well.

3.1 Identical Subtree Optimization Θ_A

Identifying unchanged code is crucial to avoid large edit scripts because any pairing of nodes from unchanged parts with nodes from changed parts causes avoidable edit actions. As example we apply CD to the AST in Fig. 2 (although it is more detailed than the CD-AST [19]). Since in this change the two assignments (=) switch places, a single *move* action for $=_{\alpha o}$ is sufficient. CD does not recognize that the subtrees of $=_{\alpha o}$ and $=_{\beta o}$ are left unchanged, albeit in a different order. Thus, CD creates four edit actions that modify each of the constants *(0,1)* and identifiers *(a,b)*. The main cause



(a) AST_o of the original code.



(b) AST_m of the modified code.

Figure 2: ASTs of the first 3 code lines in Fig. 1. Same legend as in Fig. 1.

is that CD pairs nodes from within the subtree $=_{\alpha o}$ to nodes from the two different subtrees of $=_{\alpha m}$ and $=_{\beta m}$.

Hence, the *Identical Subtree Optimization* Θ_A adds pairs of matching nodes from identical subtrees to the mapping before phase 1 of the tree differencing algorithm even starts. This reduces the number of pairs in the mapping that hold nodes from both unchanged and changed code. Θ_A is a built-in part of GT. Diff/TS [20, 21] uses a variant of it.

For each subtree, Θ_A generates a fingerprint. Identical subtrees (i.e. identical code parts) have the same fingerprint (i.e. the same string representation). Then Θ_A traverses AST_o top down. If a node's fingerprint also appears in the modified AST_m, Θ_A adds all pairs of nodes from the two subtrees to the mapping, provided that the fingerprints are unique in their respective ASTs. Since in most cases leaves are **not** unique (e.g., int_o), Θ_A ignores them for speed. As in Fig. 2, the fingerprints of the subtrees of $=_{\alpha o}$ and $=_{\beta m}$ match and are unique in AST_o and AST_m, Θ_A adds the pairs *($=_{\alpha o}$,$=_{\beta m}$)*, *($Decl_{\alpha o}$, $Decl_{\beta m}$)*, *($int_{\alpha o}$, $int_{\beta m}$)*, etc. to the mapping. Note that Θ_A not only improves the tree differencing results, it also reduces the runtime of all subsequent steps, including the tree differencing algorithm, as their complexities depend on the number of nodes that are not in the mapping.

3.2 LCS Optimization Θ_B

GT uses RTED to match smaller subtrees and inherits the inability to detect the move of *cond* in Fig. 1 (line 5). It creates two instead of one edit action (*delete* and *insert*).

The idea of the *Longest Common Subsequence* (LCS) post-processing Θ_B is to flatten two subtrees that have their roots in the mapping. If there are unmapped nodes that have the same labels within their two flattened sequences, Θ_B adds them as a pair to the mapping.

Θ_B in Fig. 3 initially traverses both ASTs top-down and collects nodes that are not yet mapped (lines 2-3). As input it uses the root nodes of both ASTs and the current mapping M. In the example, $cond_o$, $null_m$, $!=_m$, and $cond_m$ are not in the mapping. Afterwards Θ_B selects the first such node ($cond_o$) and tests whether its parent (if_o) has a partner in the mapping. As if_o is mapped to if_m, Θ_B flattens both subtrees and computes the LCS [8] in lines 12-14. A pair *(m_o, m_m)* is part of the LCS (*result*) if the two labels are identical and either the pair was previously mapped or both were unmapped (line 16). Θ_B adds the pair *($cond_o$, $cond_m$)* to the mapping. This leads to the *move* action and the shorter edit script. Θ_B does not add other pairs since the remaining nodes only appear in one sequence. To avoid duplicate work, Θ_B checks each parent only once (lines 6-9).

```
01: function LCSOpt(root_o, root_m, M)
02:     U_o ← unmappedNodesTopDown(root_o, M);
03:     U_m ← unmappedNodesTopDown(root_m, M)
04:     done ← ∅
05:     for u_o ∈ U_o do
06:         p_o ← parent(u_o)
07:         if p_o ∈ done then
08:             continue
09:         done ← done ∪ {p_o}
10:         p_m ← mappedNode(u_o, M)
11:         if p_m ≠ null then
12:             l_o ← nodesInPostOrder(p_o)
13:             l_m ← nodesInPostOrder(p_m)
14:             result ← LCS(l_o, l_m, U_o, U_m, M)
15:             for (m_o, m_m) ∈ result do
16:                 if m_o ∈ U_o ∧ m_m ∈ U_m then
17:                     M ← M ∪ {(m_o, m_m)}
18:                     U_o ← U_o \ {m_o}; U_o ← U_m \ {m_m}
19:     return M
```

Figure 3: LCS Optimization, Θ_B.

3.3 Unmapped Leaves Optimization Θ_C

Θ_C addresses the problem of *unmapped leaves* that are missing in the mapping. For instance, GT does not recognize the movement of c in Fig. 1 (line 6). This again is caused by the use of RTED to map small subtrees. Θ_B does not heal this as either c or d can be part of the LCS.

To fix this, the pseudo-code of Θ_C in Fig. 4 works on all unmapped leaves that have a parent that is mapped to a partner. In the example, c_o is not part of the mapping, but the argument list of method $m1_o$ is. Then *partnerChild* (line 7) finds a node x that is a suitable partner of c_o. Our implementation uses a cascade of 8 conditions to select the most suitable node. Condition 1: Select a node x that is among the children of $mappedP_m$, has both the same label and the same value as u_o, and is not in the mapping. Condition 2: Select a node x that is among the children of $mappedP_m$, has both the same label and the same position as u_o, and is not in the mapping. For brevity, we skip the less common remaining 6 conditions that can be found in the source code.

```
01: function UnmappedLeavesOpt(root_o, root_m, M)
02:     U_o ← unmappedLeaves(root_o, M);
03:     for u_o ∈ U_o do
04:         p_o ← parent(u_o)
05:         mappedP_m ← mappedNode(p_o, M)
06:         if mappedP_m ≠ null then
07:             x_m ← partnerChild(u_o, mappedP_m)
08:             if x_m ≠ null then
09:                 M ← M ∪ {(u_o, x_m)}
10:     return M
```

Figure 4: Unmapped Leaves Optimization, Θ_C.

As c_m is the most suitable node (same label, same value, not in the mapping) in the example, Θ_C adds *(c_o,c_m)* to the mapping (line 9). This shortens the edit script because it replaces *delete*(c_o) and *insert*(c_m) with one *move*.

Θ_C is invoked twice, once for the unmapped leaves in AST_o and once for those in AST_m, because an unmapped leaf in AST_m may have a parent in the mapping (lines 4-6), whereas its counterpart in AST_o has not.

3.4 Inner Node Optimization Θ_D

The post-processing *Inner Node Optimization* Θ_D is only effective when Θ_B or Θ_C have changed the mapping. Then it can further improve the results with the new information.

The conditions that trigger Θ_D only occur in larger code fragments. Fig. 5 shows an abstract example and its sub-optimal mapping in which only one child of O is mapped to a child of M. For two children (o_2 and o_3) of O there is a mapping to nodes in AST_m that are children of a different parent $N \neq M$. Thus, the edit script has two *move* actions.

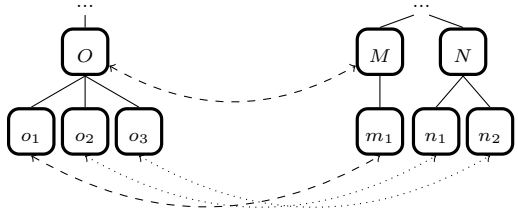
The key idea of Θ_D is to change the partner of the parent node O from M to N, so that afterwards fewer children have to *move*. In the example, after Θ_D, the edit script has only one *move* for *(o_1, m_1)*.

Fig. 6 holds the pseudo-code of Θ_D. For an inner node i_o in AST_o (O in Fig. 5), Θ_D searches for an inner node in AST_m that shares more mapped children with i_o than the current partner in the mapping (M). If found, Θ_D counts the number of its shared children (line 7). If at least half of them have a mapped partner in the subtree of the matching candidate (N), Θ_D removes the old pair *(O,M)* and adds the new pair *(O,N)* instead (line 10). Again, only pairs with the same label are considered (line 9).

3.5 Leaf Move Optimization Θ_E

Whereas Θ_C targets unmapped leaves only, Θ_E also examines pairs that are already in the mapping. GT generates two *update* actions to modify the string constants of *print* in Fig. 1 (line 8). Θ_E detects that a single *move* is sufficient.

The pseudo-code in Fig. 7 first identifies pairs of leaves in the mapping where both leaves have different values (line 2). These are candidates for the optimization. Then, for each such pair, e.g., *(42!$_o$, hello$_m$)*, Θ_E looks for a leaf that is a better partner because it has the same value (lines 5-24). To avoid unnecessary *moves*, only leaves from the partner of the parent node (e.g., the string concatenation in Fig. 1) are considered (lines 7-9). Using another parent always requires a *move* action and cannot shorten the edit script. If Θ_E finds such a node (p_m), it modifies the current mapping, but only if there is just a single other possible match (*unique*). Otherwise, Θ_E does not touch the mapping as there is no



(a) AST_o of original code. **(b) Modified AST_m.**

Figure 5: ASTs with sub-optimal mapping. O, M, ..., n_i are simplifications, e.g., of argument lists.

```
01: function OptimizeInnerNodes(root_o, root_m, M)
02:   I_o ← mappedInnerNodesInPostOrder(root_o, M);
03:   for i_o ∈ I_o do
04:     i_m ← mappedNode(i_o, M)
05:     max_m ← nodeWithMostMappedChildren(i_o, M)
06:     if max_m ≠ i_m then
07:       commonChildren ← commonChildren(i_o, max_m, M)
08:       if |commonChildren| > floor(|children(i_o)|/2) then
09:         if lbl(i_o) = lbl(max_m) then
10:           M ← M ∪ {(i_o, max_m)}; M ← M \ {(i_o, i_m)}
11:   return M
```

Figure 6: Inner Node Optimization, Θ_D.

fast way to decide which match prevents the edit script from growing. There is one exception. Θ_E updates the mapping if there are several leaves with identical values and the position of the possible match inside its parent's subtree is the same as the position of the leaf in the original subtree (1 for 42!$_o$).

As there is a better matching leaf in the example, Θ_E removes the old pair *(42!$_o$, hello$_m$)* in line 13 and adds the new pair *(42!$_o$, 42!$_m$)* in line 14. Afterwards, *hello$_m$* lacks a partner in the mapping. As the previous partner of 42$_m$ (*hello$_o$*) has the same label, Θ_E adds (*hello$_o$*, *hello$_m$*) to the mapping (lines 16-19). For Fig. 1, Θ_E is done at this point. It has replaced the two *updates* with a single *move*. If in other cases a pair with different values (instead of (*hello$_o$*, *hello$_m$*)) is added to the mapping, the pair is added to the work list for further optimization (lines 20-21, 25). Θ_E achieves termination with a second work list (line 21).

There may be no fitting node for the leaf of the original tree. If for the pair *(42!$_o$, hello$_m$)* there is no identifier 42!$_m$, then there still can be a better partner for *hello$_m$*. The second loop (lines 23-24) finds the partner in a way that is similar to the loop in lines 9-22. After a first run, Θ_E runs again with *parent(l_m)* instead of *mappedNode(p_o, M)* in line 7. This optimizes leaves for parents that are not in the mapping.

3.6 Order of Θ_{A-E}

Due to their design, the order (Θ_A; phase 1; Θ_B; Θ_C; Θ_D; Θ_E) results in the shortest edit scripts in our evaluation. Θ_A is a pre-processing step that does not require an initial mapping while the post-processing steps Θ_{B-E} do. Other

```
01: function LeafMoveOpt(root_o, root_m, M)
02:   work ← mappedLeavesWithValueDifferences(M)
03:   work' ← new List()
04:   while work ≠ ∅ do outer:
05:     for (l_o, l_m) ∈ work do
06:       p_o ← parent(l_o)
07:       mp_m ← mappedNode(p_o, M) // parent(l_m) in run 2
08:       pos ← childPosition(i_o, p_o)
09:       for c ∈ children(mp_m) do
10:         if isIdenticalLeaf(c, l_o) then
11:           pos_c ← childPosition(c, mp_m)
12:           if unique(l_o,children(mp_m)) ∨ pos=pos_c then
13:             M ← M \ {(l_o, l_m)}
14:             M ← M ∪ {(l_o, c)};
15:             c' ← mappedNode(c, M)
16:             if c' ≠ null then
17:               M ← M \ {(c', c)}
18:               M ← M ∪ {(c', l_m)};
19:               work ← remove(work, (c', c))
20:               if value(c') ≠ value(l_m) then
21:                 work' ← add(work',(c', l_m))
22:             break
23:       for c ∈ children(p_o) do
24:         ...
25:     work ← work'; work' ← ∅
26:   return M
```

Figure 7: Leaf Move Optimization, Θ_E.

orders or subsets of Θ_{B-E} are valid but in general result in longer edit scripts. Θ_D only makes sense after Θ_B or Θ_C. Θ_E is useful on its own.

4. MOVE-OPTIMIZED TREE DIFF

In contrast to the other considered tree differencing algorithms (only Θ_A is already part of GT), MTDIFF includes (and also requires) all the general optimizations Θ_{A-E} . The other major difference is in phase 1, i.e., in the way MTDIFF computes the mapping of nodes between ASTs.

Due to their design, GT and JSync struggle with movements of subtrees that are also altered while they are moved, e.g., the (moved and altered) +-subtree in line 9 of Fig. 1. As CD can detect such movements in its coarse-grained CD-AST, we use it as foundation of MTDIFF.

Both CD and MTDIFF take a two-step approach to create the mapping in phase 1. In their first step, both algorithms create pairs of leaf nodes for the mapping. Based on the leaves in the mapping, their second step creates pairs of inner nodes. The workings of these steps are different in CD and MTDIFF. The latter has to address two challenges that are not an issue for CD (due to its original purpose).

Challenge A: Since the values of nodes in the CD-AST contain larger pieces of the code, there are fewer nodes with identical labels and values than in a fine-grained AST. For example, there are two identical *int* nodes in Fig. 2, whereas the CD-AST represents the two non-identical statements in lines 2 and 3 as only two nodes with non-identical values. Thus, MTDIFF has to find a mapping that takes care of identical leaves (not relevant for CD) and that also yields a short edit script. Sec. 4.1 presents how MTDIFF finds pairs of (identical) leaves for the mapping.

Challenge B: Similarly, the coarse-grained CD-AST encodes loop conditions (and other pieces of the code) as values. Fine-grained ASTs express them as separate subtrees. Thus, MTDIFF requires new heuristics to measure the similarity of inner nodes when it computes a mapping for them. Additionally, MTDIFF has to examine more inner nodes to find suitable partners. Sec. 4.2 addresses the details.

Although MTDIFF can reuse the similarity function for leaves from CD, its heuristics require their own set of similarity thresholds. Sec. 4.3 explains our approach to identify suitable values.

4.1 Leaf Mapping

The first step of phase 1 creates pairs of leaf nodes and adds them to the mapping. The pseudo-code for this step is in Fig. 8. *LeafMapping* is called on the original AST_o and the modified AST_m. Initially it extracts lists L_o, L_m of all leaves that are not already part of the Mapping M due to Θ_A (lines 2-3). Then the loops (lines 5-9) compute the pairwise similarities of the unmapped leaves. MTDIFF shares with CD the similarity function sim_L for the leaves:

$$sim_L(l_o, l_m) = \begin{cases} 0 & lbl(l_o) \neq lbl(l_m) \\ sim_{2g}(v(l_o), v(l_m)) & otherwise \end{cases} .$$

AST-nodes with different labels *lbl* never match. Otherwise, MTDIFF uses bigrams [24] (sim_{2g}) to compute the string similarity of the values (v) [19] in the interval $[0, 1]$.

Then *LeafMapping* sorts the list of node Pairs by decreasing similarities. Nodes with the same similarities are sorted according to the post-order visiting sequence, first of AST_o and then of AST_m (line 10). To keep this example simple,

```
01: function LeafMapping(root_o, root_m, M, resolveAmbiguity)
02:     L_o ← unmappedLeaves(root_o, M)
03:     L_m ← unmappedLeaves(root_m, M)
04:     U ← L_o ∪ L_m; P ← ∅
05:     for l_o ∈L_o do
06:         for l_m ∈L_m do
07:             similarity ← sim_L(l_o, l_m)
08:             if similarity ≥ threshold_L then
09:                 P ← P ∪ {(l_o, l_m, similarity)}
10:     P ← orderBySimilarityAndVisitOrder(P);
11:     C ← P; A ← ∅
12:     if resolveAmbiguity then
13:         C ← unambiguousPairs(P)
14:         A ← ambiguousPairs(P)
15:     for (l_o, l_m, s) ∈ C do
16:         if l_o ∈ U ∧ l_m ∈ U then
17:             M ← M ∪ {(l_o, l_m)}
18:             U ← U \ {l_o}; U ← U \ {l_m}
19:     A ← handleAmbiguities(A, M)
20:     for (l_o, l_m, s) ∈ A do
21:         if l_o ∈ U ∧ l_m ∈ U then
22:             M ← M ∪ {(l_o, l_m)}
23:             U ← U \ {l_o}; U ← U \ {l_m}
24:     return M
```

Figure 8: Creates a mapping from two trees ($root_o$, $root_m$).

we ignore the results of the Θ_A pre-processing. P then contains the following ordered list of leaf pairs, with a similarity above zero: (foo_o, foo_m, 1), (2_o, 2_m, 1), ($int_{\alpha o}$, $int_{\alpha m}$, 1), ($int_{\alpha o}$, $int_{\beta m}$, 1), (a_o, a_m, 1), (0_o, 0_m, 1), ($int_{\beta o}$, $int_{\alpha m}$, 1), ($int_{\beta o}$, $int_{\beta m}$, 1), (b_o, b_m, 1), (1_o, 1_m, 1), (xy_o, xyz_m, 0.66). Only for the 7 pairs without *int* nodes there is no ambiguity.

Whereas CD does not need to handle Ambiguous nodes (resolveAmbiguity = *false*), MTDIFF separates them (in list A) from unambiguous ones with a Clear match (list C) in lines 12-14. Ambiguous nodes are nodes for which there is no single best match in P. All *int* nodes in Fig. 2 fall into this category. Then, MTDIFF examines all entries in C and adds them to the mapping if both nodes are part of U (set of Unmapped nodes). Thus, the for-loop in lines 15-18 adds all the 7 unambiguous node pairs in the above order.

Without an ambiguity treatment of the *int* nodes in Fig. 2 (and without Θ_A), the second step (mapping of inner nodes) would add the barely similar pairs ($Decl_{\alpha o}$, $Decl_{\alpha m}$, 0.5) and ($Decl_{\beta o}$, $Decl_{\beta m}$, 0.5) to the mapping. This would lead to an overly verbose edit script that moves the variable names (a_o and b_o) and declarations ($Decl_{\alpha o}$ and $Decl_{\beta o}$).

To prevent this, *handleAmbiguities* resolves such ambiguities before MTDIFF adds inner nodes to the mapping. It reconsiders the pairs based on the similarity values of all pairs in A and takes the previously determined pairs in M into account. Based on A, *handleAmbiguities* builds a matrix for each group of ambiguous leaves, see Fig. 9(a) for 2×2 ways to pair the *int* nodes. The key idea is that the parents of the leaves often can help to resolve the ambiguity because their similarities are connected to the choices of leaf pairs. Therefore, *handleAmbiguities* recursively uses *LeafMapping* on the (small) subtrees that have their parents as root. If, among all combinations, there are parent-rooted ASTs that have more mapped nodes than the others, MTDIFF has an indication for a selection of l_m among the ambiguous leaves.

	$int_{\alpha o}$	$int_{\beta o}$
$int_{\alpha m}$	1	1
$int_{\beta m}$	1	1

(a) Ambiguous leaf similarity matrix S.

	$int_{\alpha o}$	$int_{\beta o}$
$int_{\alpha m}$	$\frac{2}{3}$	1
$int_{\beta m}$	1	$\frac{2}{3}$

(b) Similarity matrix S after *handleAmbiguities*.

Figure 9: Ambiguous leaf similarity matrix for Fig 2.

```
01: function InnerNodeMapping(root_o, root_m, M)
02:   I_o ← unmappedInnerNodes(root_o, M)
03:   I_m ← unmappedInnerNodes(root_m, M)
04:   U ← I_o ∪ I_m; P ← ∅
05:   for i_o ∈I_o do
06:     for i_m ∈I_m do
07:       vsim ← sim_IV(i_o, i_m)
08:       csim ← sim_IC(i_o, i_m)
09:       similarity ← (w_I0 · vsim + w_I1 · csim) / 2
10:       if similarity ≥ threshold_I then
11:         P ← P ∪ {(i_o, i_m, similarity)}
12:   P ← orderBySimilarityAndVisitOrder(P);
13:   for (i_o, i_m, s) ∈ P do
14:     if i_o ∈ U ∧ i_m ∈ U then
15:       M ← M ∪ {(i_o, i_m)}
16:       U ← U \ {i_o}; U ← U \ {i_m}
17:   return M
```

Figure 10: Creates pairs of inner nodes.

In the example, the parent of the leaf $l_o = int_{\alpha o}$ is $Decl_{\alpha o}$. For the ambiguous choices for $l_m \in \{int_{\alpha m}, int_{\beta m}\}$, their parents are $Decl_{\alpha m}$ and $Decl_{\beta m}$, resp. Thus, MTDIFF calls *LeafMapping* twice: with *(*$Decl_{\alpha o}$*,* $Decl_{\alpha m}$*, M, false)* and also with *(*$Decl_{\alpha o}$*,* $Decl_{\beta m}$*, M, false)*. The adjusted similarity sim_A between ambiguous leaves is then based on the results of *LeafMapping*, i.e., the number of pairs in the mapping. The more pairs, the higher is the similarity between two leaves. With normalization, sim_A is:

$$sim_A(l_o, l_m) = \frac{|LeafMapping(parent(l_o), parent(l_m), M, false)|}{|max(nodes(parent(l_o)), nodes(parent(l_m)))|}.$$

For $Decl_{\alpha o}$ and $Decl_{\alpha m}$, this yields a similarity of $\frac{2}{3}$ as 2 of the 3 nodes in this subtree match, see cell ($int_{\alpha o}$,$int_{\alpha m}$) of the adjusted matrix in Fig. 9(b). Based on the matrix, the best mapping of leaves is *(*$int_{\alpha o}$*,* $int_{\beta m}$*)* and *(*$int_{\beta o}$*,* $int_{\alpha m}$*)* because this maximizes the total combined similarity (= 2).

If *LeafMapping* on the parents does not find a unique best pair for each row and column, MTDIFF recursively applies *LeafMapping* to the grand-parents of the best-rated parents.

It is still possible that even at the root nodes there are identical similarity values for several leaf pairs. In Fig. 1 this is the case for the *log* identifiers. They have the same parent hierarchy, identical values and labels. The only difference is their position within the AST. Therefore, *handleAmbiguities* also uses the position of the leaves. It generates a second matrix D (Distance) that holds the differences in the post-order position of the node pairs. For example, the first log_o and first log_m nodes both have the same post-order position and thus a difference d_{ij} of 0. Note that solely using the position matrix would lead to longer edit scripts.

For the best results, *handleAmbiguities* combines matrix D (with cells d_{ij}) with matrix S (cells s_{ij}) after the recursive application of sim_A. The combination uses weights (w_{L0} and $w_{L1} \in [0, 1]$) and inverts and also normalizes the values to $s'_{ij} = w_{L0} \cdot (1 - s_{ij}) \cdot 1000 + w_{L1} \cdot d_{ij}$. Sec. 4.3 shows how we determine the weights. The weighted combination is done in preparation for the next and final part of the leaf mapping that ensures a suitable choice of leaf pairs.

It is easy to see the best mapping of leaves in the matrix in Fig. 9(b). In the general case, finding the pairs with the best combined similarity is identical to solving the assignment problem [25], for which *handleAmbiguities* applies the Hungarian method [25] to the matrix as final step.

4.2 Inner Node Mapping

Next, MTDIFF uses the *InnerNodeMapping* algorithm in Fig. 10. Similar to the leaf mapping step, MTDIFF computes the similarities between all inner nodes and adds all pairs whose similarity is above a threshold$_I$ to the list P (of similar Pairs) in lines 5-11. Whereas CD always selects the first pair found above a threshold to prune the search space and to reduce the runtime, MTDIFF exploits the fact that Θ_A already added most inner nodes to the mapping which reduces the size of the problem. Thus, MTDIFF can afford to compute all similarities above the threshold$_I$ and to select the pair with the highest similarity.

Computing the similarity of two inner nodes is more difficult for MTDIFF than for CD. Due to challenge (B), the similarity functions of CD are not applicable. They expect that larger parts of the code (e.g., a complete loop condition) are included in the values of inner nodes. Instead MTDIFF uses two similarity heuristics (sim_{IV} and sim_{IC}) and combines them with two weights (w_{I0} and $w_{I1} \in [0, 1]$) in line 9. The first similarity heuristics handles the values of inner nodes: $sim_{IV}(i_o, i_m) =$

$$\begin{cases} 0.0 & lbl(i_o) \neq lbl(i_m), \\ 0.2 & v(i_o) \neq v(i_m) \wedge\ discrete(i_o) \\ sim_{2g}(v(i_o), v(i_m)) & otherwise \end{cases}.$$

This structure is similar to sim_L. The difference lies in the handling of inner node types for which only discrete values are allowed. An example for such inner nodes are arithmetic expressions. For them, only the values of the operation +, ∗, etc. are valid. In these cases, the similarity of different operations is set to 0.2 (based on preliminary experiments). Hence, MTDIFF can express an operator change in an arithmetic expression with a single *update* action. For all other types, sim_{IV} uses sim_{2g} to measure the textual similarity of the node values. The use of sim_{2g} has another positive side effect: it allows MTDIFF to use the CD-AST in the evaluation. Without the extra treatment of arithmetic expressions, sim_{2g} would return a similarity of 0, leading to two edit actions (*delete* and *insert*).

The second similarity heuristics sim_{IC} of MTDIFF takes the similarity of the children into account. Without it, *InnerNodeMapping* would add *(*$=_{\alpha o}$*,* $=_{m1\alpha}$*)* and *(*$Decl_{\alpha o}$*,* $Decl_{\alpha m}$*)* to the mapping for our example in Fig. 2. This requires 4 additional moves (for the leaves a_o, 0_o, b_o, 1_o).

To take the children into account, according to the evaluation of Fluri et al [19], it is sufficient to only examine the leaves of the children's subtrees. In contrast to CD, sim_{IC} has to take the importance of smaller subtrees (e.g., loop conditions) into account (challenge (B)). CD handles this together with the values of inner nodes as such code parts appear as values and not as subtrees in the CD-AST. Instead, sim_{IC} has to normalize the importance of the subtrees of the inner nodes i_o and i_m to make them independent of their sizes. To enforce that small subtrees (expressing a control flow condition) are as important as the subtrees of other siblings, regardless of their number of leaves, MTDIFF uses:

$$sim_{IC}(i_o, i_m) = \begin{cases} 0 & lbl(i_o) \neq lbl(i_m), \\ \frac{sumC(i_o, i_m)}{|ch(i_o)|+|ch(i_m)|} & otherwise \end{cases}.$$

Nodes with different AST labels never match. Otherwise, the similarity of two inner nodes depends on their subtrees (children ch). For illustration of sim_{IC}, consider the two $Args$ nodes in Fig. 2. Since $Args_o$ has $+_o$ as its only child and $Args_m$ has the two children $+_m$ and t_m, there is a division by $|\mathrm{ch}(i_o)| + |\mathrm{ch}(i_m)| = 1+2 = 3$ for the normalization of

$$sumC(i_o, i_m) = \sum_{q_o \in ch(i_o)} \frac{|comp(q_o, i_m)|}{|leaves(q_o)|} + \sum_{q_m \in ch(i_m)} \frac{|comp(q_m, i_o)|}{|leaves(q_m)|}.$$

For each of the children q of both trees, *sumC* computes similarity values by counting leaf pairs in the mapping (as determined by the previous leaf mapping step, i.e., xy_o and xyz_m is counted as one pair since the similarity is above the threshold). With *comp(q, i)*, MTDIFF compares all the leaves of the subtrees q and i. If one of the leaves has a mapped leaf somewhere in i it is counted. The resulting count is normalized by the number of leaves to stress the importance of small subtrees, e.g., a control flow condition. Note that a leaf is counted even if it has been moved to a different subtree in i. The rationale is that program structures remain similar even if somewhere within them code is moved around. In the example, the child $+_o$ of $Args_o$ has two leaves that both have a mapped node among the leaves of $Args_m$. Weighted by the two leaves in q, this contributes $\frac{2}{2}$ to *sumC*. Likewise, $+_m$ also contributes $\frac{2}{2}$. As there is no mapped leaf for t_m in i_o, this child adds $\frac{0}{1} = 0$ to *sumC*. Overall, the two *Args* nodes have a normalized similarity of $\frac{1}{3}(1+1+0) = 0.67$. With this approach, MTDIFF makes the condition and the body of a loop equally important and avoids longer edit scripts due to the problem of descendant subtree matching [19].

4.3 Thresholds and Weights

As mentioned above, MTDIFF uses four weights and two thresholds. To identify suitable values for those constants, we performed an extensive Particle Swarm Optimization (PSO) [22] that minimizes the edit scripts for each of the changed files in the histories of 4 different open source repositories (Checkstyle, Fitlibrary, JGraphT, JUnit). The target function for the PSO was the number of files for which the weights and thresholds result in shorter edit scripts compared to an initial set of scripts. This function achieves a balance between minimizing the edit scripts for files with a few changes and files with several thousand changes (the alternative to minimize the total number of all edit actions of all files favored large files). The best values determined by the PSO are $w_{L0} = 0.37$, $w_{L1} = 0.0024$, $w_{I0} = 0.40$, $w_{I1} = 0.25$, $threshold_L = 0.88$, and $threshold_I = 0.10$.

5. COMPLEXITY ANALYSIS

The evaluation shows that in practice Θ_{A-E} barely increase the runtime of the tree differencing algorithms. Θ_A can even make them faster. We now analyze the complexity for n nodes (i inner nodes, l leaves). With hash maps Θ_A has an average complexity of $O_{\Theta_A}(n)$ [18]. Θ_B has a worst case complexity of $O_{\Theta_B}(n^3)$. But real-world codes rarely (or never) have the worst case in which $\frac{n}{2}$ nodes from both trees are part of a mapping and where each LCS execution does not change this condition. Θ_C has a complexity of $O_{\Theta_C}(u)$ with u unmapped nodes. Θ_D has a complexity of $O_{\Theta_D}(i^2)$. Θ_E has a complexity of $O_{\Theta_E}(l^2 \cdot \log l)$ as it adds at most $\frac{l}{2}$ mappings to the work list.

In our evaluation, MTDIFF is faster than other examined algorithms, even though its asymptotic complexity is above $O_{CD}(l^2 \cdot \log l^2 + i \cdot l)$. For practical use, O_{Θ_A} must reduce the number of unmapped leaves and unmapped inner nodes for MTDIFF's phase 1.

In more detail: *LeafMapping* without ambiguity resolution is in $O(l^2 \cdot \log(l^2 + c_L))$. This is based on l^2 leaf comparisons with cost c_L and the sorting of the leaf pairs according to their similarities in $O(l^2 \cdot \log l^2)$. The worst case is $l \cdot (l-1)$ pairs of leaves with equal similarities and labels. Then *LeafMapping* with ambiguity resolution has to compare all i^2 combinations of subtrees of inner nodes to determine the best leaf matching. Each comparison requires l^2 executions of *LeafMapping* without ambiguity handling. This causes a complexity of $O(l^4 \cdot i^2 \cdot (\log l^2 + c_L))$. Dynamic programming reduces this to $O_L(l^2 \cdot i^2 \cdot \log(l^2 + c_L))$ as subtree comparisons can be reused. The Hungarian method (see Sec.4.1) is in $O_H(l^3)$. The position computation is in $O_P(l^2)$. The *InnerNodeMapping* has a complexity of $O_I(i^2 \cdot (\log i^2 + c_I))$ as there are i^2 comparisons of cost c_I. The cost of sorting the inner node pairs according to their similarities is in $O(i^2 \cdot \log i^2)$. Thus, if the comparison costs are in $O(1)$, the complexity of phase 1 of MTDIFF is in $O_L + O_H + O_P + O_I = O(l^2 \cdot i^2 \cdot \log(l^2 + c_L) + i^2 \cdot (\log i^2 + c_I) + l^3 + l^2) \subseteq O(l^2 \cdot i^2 \log l^2 + i^2 \cdot \log i^2 + l^3)$.

6. EVALUATION

The evaluation shows (a) that the general optimizations shorten the edit script for all tree differencing algorithms and (b) that MTDIFF achieves even shorter scripts. Moreover, it shows (c) that the optimizations and MTDIFF detect more *moves* and (d) that for both the optimizations and MTDIFF the performance is fast enough for practical use. Finally, we show (e) that shorter edit scripts are more beneficial to developers.

As a common code base, we use the GT framework [1]. This includes the unoptimized baselines of the tree differencing algorithms GT, RTED, and CD. In the framework, GT and RTED share a fine-grained AST (called GT-AST below). CD uses a coarse-grained CD-AST. For JSync we adopted the published source code [3] to the GT framework (with the JSync-AST) so that the four tree differencing algorithms use the same edit script generator (phase 2) from the GT framework. Because of the different ASTs, we never compare the results of GT, RTED, CD and JSync with each other. Instead, we show the effects of Θ_{A-E} for each of the algorithms on their ASTs separately. Since MTDIFF is configurable to use any of the three ASTs, we can compare it with the other algorithms in Sec. 6.2.

Our dataset uses all commits in the master branches (no merges) before Aug. 2015 taken from 9 active open-source Java projects (Ant, Checkstyle, Cobertura, DrJava, Eclipse JDT Core, Findbugs, Fitlibrary, JGraphT, JUnit). All the projects are part of the Qualitas Corpus [40]. We excluded 61,435 file revisions because they only contained changed comments and whitespace. We also excluded 5,496 file revisions with more than 20,000 AST nodes because RTED could not process them with the 128 GB of RAM our workstation is equipped with (CPU: 3.6 GHz Intel Xeon). We ignored 9,890 revisions because CD or RTED could not deal with them in our (artificially set) time limit of 3 minutes. For the remaining 126,162 file revisions in those commits (simply called *files* below) we ran the tree differencing algorithms on the original file before the commit and the modified file after the commit. As we ran the tree differencing algorithms in 5 combinations (without optimizations, with Θ_A, with Θ_B, with Θ_{C-E}, with Θ_{A-E}) and also MTDIFF with 3 different AST types, the time limits were necessary. Except for the RTED versions (due to the memory limitations) we also executed 10 changes in parallel. Even with the time limits and the parallel execution, the measurements took about 6 weeks (24/7) on the workstation.

6.1 Effects of the General Optimizations

Table 1 shows the effects of Θ_{A-E} (in this order) on the four state-of-the-art tree differencing algorithms. We show the effect of Θ_{A-E} on the number of pairs in the mapping (*map.*) and on the *size* of the resulting edit scripts. The third column shows for how many of the 126,162 files the baseline tree differencing algorithm (i.e., GT, RTED, CD, JSync) produces better results (without optimizations Θ_{A-E}). A mapping is better if it has more pairs of nodes; an edit script is better if it is shorter. Column four shows for how many files the optimized algorithm with Θ_{A-E} leads to better results compared to its original versions. The last column holds the rest, i.e., the number of files for which the scripts have the same size or the same number of mappings.

Across all the tree differencing algorithms, the five optimizations only have a harmful effect (both on the number of pairs in the mappings and the script sizes) for a negligible 1.3% of the files. We discuss some reasons for this in Sec. 6.2. For 18–98% of the files, Θ_{A-E} shorten the edit scripts for all tree differencing algorithms. The mean edit script size reduction by Θ_{A-E} is 279 edit actions (median: 61). For the same files, the mean edit script size produced by the baseline algorithms is 351 edit actions (median: 142).

RTED is a special case in our evaluation as it already finds optimal (shortest) edit scripts, although without *moves*. No optimization can improve them. But as the RTED in the GT framework produces a mapping of nodes, this mapping can be fed into an edit script generator from the GT framework that considers *moves*. The generated scripts can be shorter due to the *moves* and can also benefit from our optimizations. In fact, for 22% of all files, RTED with optimizations has shorter edit scripts. This also means that despite its original use, RTED with the optimizations can act as a tree differencing algorithm that takes *moves* into account.

On its own, each of the five optimizations also improves the results. As Θ_A is already part of the baseline GT, it has no effect there. But Θ_A has a huge impact on the CD results with an improvement for 97% of all files. The reason is that for its original purpose CD did not need to handle unchanged code explicitly. When it is added to JSync, Θ_A shortens the scripts for 10% of the files, although JSync already uses a textual line *diff* to detect unchanged code. From all optimizations, Θ_B has the largest impact on GT and shortens the scripts for 16% of the files without harmful impact. It also has a strong effect on CD (shorter scripts for 94% of all files) and on JSync (26%). Since individually, Θ_C-Θ_E have little effect, we measured them together. Θ_{C-E} shorten the scripts for GT for 6% of the files (CD: 71%, JSync: 6%). Overall, all optimizations reduce the script sizes for several thousand files.

6.2 Performance of MTDIFF

We now compare MTDIFF with the state-of-the-art tree differencing algorithms on their respective ASTs. For instance, MTDIFF uses the JSync-AST when compared to JSync. For the comparison we always switch on the optimizations Θ_{A-E} as they shorten the edit scripts or produce the same size for almost all files. The third column of Table 2 lists all files for which the state-of-the-art algorithms (with Θ_{A-E} activated) produce better results than MTDIFF. The fourth column shows the larger number of files for which MTDIFF is better. Across all algorithms, MTDIFF shortens the script sizes for 19%–42% of all files. Compared to the baseline algorithms without Θ_{A-E}, the improvement is even larger (25%–98%). The mean edit script size reduction by MTDIFF (compared to the algorithms with Θ_{A-E}) is 21 edit actions (median: 6). For the same files, the mean edit script size produced by the algorithms with Θ_{A-E} is 110 edit actions (median: 42). Note that the comparison also shows that MTDIFF is capable of producing short edit scripts for different types of ASTs.

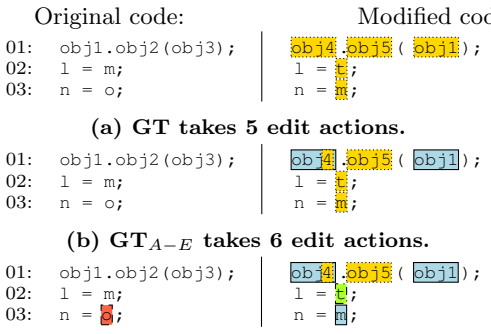


Figure 11: Tough code changes (legend from Fig. 1).

There are several cases in which the state-of-the-art algorithms produce shorter edit scripts, i.e., in which the optimizations Θ_{A-E} are harmful or for which MTDIFF produces longer edit scripts. A manual inspection of the cases revealed two main patterns of tough code changes. Fig. 11 illustrates both. First, if nodes with the same label are *moved* and at least one of them is *updated*, it is sometimes shorter to use only *update* actions in the edit script. Fig. 11(a) shows that GT can express the changes in line 1 with three *updates* as the underlying RTED finds the optimal *move*-free edit script. In some cases, JSync's heuristics also handle this situation well. In contrast, the optimizations Θ_{A-E} add as many pairs of identical nodes to the mapping as possible. Under some circumstances this can lead to longer edit scripts as shown in line 1 of Fig. 11(b+c). The second tough pattern are nodes that are *moved* and replaced in their former position with a node of the same label. Consider lines 2 and 3 of the example in Fig. 11. The identifier m is *moved* from line 2 to line 3 and a new identifier t takes its former place. While it is possible to express the change with the two *updates* (m_o, t_m) and (o_o, m_m), phase 1 of MTDIFF

Table 1: Effects of Θ_{A-E} on changed files.

		Baseline better	With Θ_{A-E} better	Same Size
GT (GT-AST)	# map.↑	3 (00.0%)	26,727 (21.2%)	99,432 (78.8%)
	# size↓	90 (00.1%)	22,427 (17.8%)	103,645 (82.2%)
RTED (GT-AST)	# map.↑	1,539 (01.2%)	26,729 (21.2%)	97,894 (77.6%)
	# size↓	1,325 (01.1%)	28,230 (22.4%)	96,607 (76.6%)
CD (CD-AST)	# map.↑	14 (00.0%)	120,149 (95.2%)	5,999 (04.8%)
	# size↓	2 (00.0%)	123,724 (98.1%)	2,436 (01.9%)
JSync (JSync-AST)	# map.↑	86 (00.1%)	38,269 (30.3%)	87,807 (69.6%)
	# size↓	131 (00.1%)	37,963 (30.1%)	88,068 (69.8%)

Table 2: Performance of MTDIFF on changed files.

		Alg. with Θ_{A-E} better	MTDIFF better	Same Size
GT$_{A-E}$ (GT-AST)	# map.↑	1,646 (01.3%)	20,155 (16.0%)	104,361 (82.7%)
	# size↓	8,966 (07.1%)	23,797 (18.9%)	93,399 (74.0%)
RTED$_{A-E}$ (GT-AST)	# map.↑	1,136 (00.9%)	24,425 (19.4%)	100,601 (79.7%)
	# size↓	9,124 (07.2%)	24,944 (19.8%)	92,094 (73.0%)
CD$_{A-E}$ (CD-AST)	# map.↑	1,150 (00.9%)	18,556 (14.7%)	106,456 (84.4%)
	# size↓	3,628 (02.9%)	52,369 (41.5%)	70,165 (55.6%)
JSync$_{A-E}$ (JSync-AST)	# map.↑	1,385 (01.1%)	21,845 (17.3%)	102,932 (81.6%)
	# size↓	8,287 (06.6%)	25,104 (19.9%)	92,771 (73.5%)

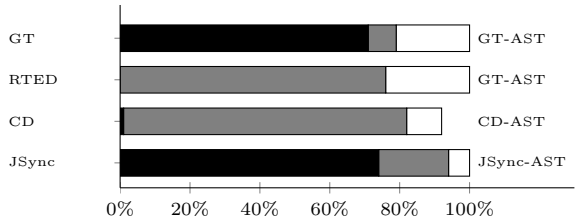


Figure 12: Fraction of *move*-only that are found on the respective ASTs (black for the baseline alg., gray with Θ_{A-E} added, and white for MTDIFF).

nevertheless adds the pair (m_o, m_m) to the mapping which not only causes the *move* in phase 2, but also a *delete* (o_o) and an *insert* (t_m) action. This yields a longer edit script, see Fig. 11(c). We plan to address these issues with an additional post-processing step in the future.

6.3 Move Accuracy

Not all possible *moves* reduce the size of the edit scripts (see Fig. 11). In general, it is hard to determine whether the *moves* in a script make it shorter. Falleri et al. [18] argue that a tree differencing algorithm is more precise in detecting *moves* and thus more likely to find the ideal number of *moves* if it identifies more scripts that solely consist of *moves*.

Based on this way to gauge the script quality, we count the files for which the tree differencing algorithms have found *move*-only scripts. Separately for each type of AST, we identify all such files that are found by any of the algorithms (with or without Θ_{A-E}) and by MTDIFF. Fig. 12 sets this total count of *move*-only scripts to 100% for each of the ASTs. The black areas in the figure show the fractions that the baseline versions identify, the gray areas show the fractions found when the optimizations Θ_{A-E} are activated. With the optimizations, all algorithms find more *move*-only scripts. Thus, the optimizations improve the accuracy in detecting *moves*. No single *move*-only script was missed because of the optimizations. There is no black area for the baseline RTED as it does not generate *moves*.

The white areas show that MTDIFF is always more accurate in detecting *moves*. For both the GT-AST and the JSync-AST, MTDIFF did not miss a single *move*-only script. On its AST, CD_{A-E} identifies some *move*-only scripts that MTDIFF does not detect (8% of all scripts). The reason is that the PSO (see Sec. 4.3) optimized MTDIFF's weights and thresholds for a fine-grained AST.

6.4 Runtime

To evaluate the runtime of the optimizations we compare the baseline algorithms with their optimized variants. To ease comparison with related work, we select the Jenkins [2] repository that is also used by Falleri et al. [18]. As dataset we use all 1,293 file changes in the commits between the versions 1.509.2 and 1.532.2. In a total of 27 hours, our workstation executed each tree differencing algorithm 5 times on these changes.

We only measure the time taken by phase 1 and by the optimizations as all the tree differencing algorithms use the same script generator (phase 2). For this dataset there is no need to exclude any files due to memory or runtime limitations. Also, in contrast to the previous measurements, we executed no changes in parallel.

The box plots in Fig. 13 capture the results (25%/75% quartiles, whiskers for ± 1.5 times the interquartile range). The optimizations only slightly slow down GT, RTED, and JSync. Across all files, the mean runtime increase for GT, RTED and JSync is 0.062s (median: 0.0032s). For most applications this is an acceptable overhead. With the optimizations switched on, CD runs even faster because Θ_A reduces the problem size, see the discussion in Sec. 3.1.

The lower three box plots in Fig. 13 show the runtime of MTDIFF on the different ASTs. Due to its higher complexity, MTDIFF is slower than GT. This is the price to pay for shorter edit scripts. Even though phase 1 of MTDIFF has a higher asymptotic complexity than CD, it is faster due to the effect of Θ_A. Compared to CD_{A-E}, MTDIFF is on par. Overall, MTDIFF requires less than 0.3s for the majority of files which we consider an acceptable runtime. The lower three box plots also illustrate the effect of the AST granularity on the runtime.

6.5 Script Size Questionnaire

The goal of Θ_{A-E} and MTDIFF is to produce shorter edit scripts. This section reports on a survey with eight developers (not the authors) to determine whether this is a reasonable goal and whether shorter edit scripts are more helpful to them. We base this study on the dataset that was used for Tables 1 and 2. From the 126,162 file changes, we excluded large ones (pairs in the mapping + edit actions ≥ 500) as they are hard to analyze for study participants.

From each of the $2 \times 4 \times 3 = 24$ shaded cells in the two tables, we randomly selected 10 file changes and the corresponding two edit scripts created by the two compared tree differencing algorithms (or variations). The alternative, a completely random selection of the changes would have favored the tree differencing algorithms with Θ_{A-E} over the baseline versions, MTDIFF over the algorithms with Θ_{A-E}, or same-size cases over cases with different script sizes. If there are less than 10 file changes in a cell (for instance there are only 2 changes for which the baseline CD is better than CD_{A-E}), we randomly picked the missing file changes from the same column. We also picked only changes with different edit scripts from the cells in the same-size columns. In the other cells, the size difference in the edit scripts was between 1 and 234 (mean: 1.5, median: 2.5).

We presented a questionnaire with all of the $24 \times 10 = 240$

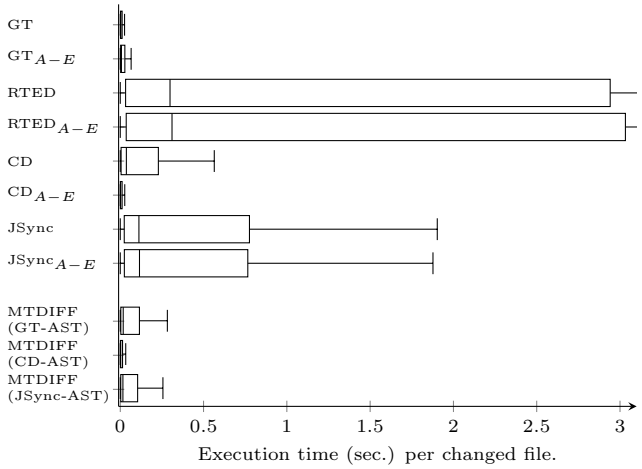


Figure 13: Runtimes for Jenkins.

Table 3: Questionnaire results (1,920 answers).

	First more helpful	Second more helpful	Unclear	Sum
First shorter	270	180	190	640
Second shorter	106	354	180	640
Same Size	197	238	205	640
Sum	573	772	575	1,920

pairs of edit scripts in a graphical representation (similar to Fig. 1) to the eight study participants that all hold a master or PhD degree in Computer Science or Mathematics. To avoid order effects, we randomized the order of the edit scripts in each pair and also showed the 240 pairs to each of the participants in a random order. For each of the pairs, we asked the participants "Is the first or the second representation more helpful in understanding the intention of the change? [First, Second, or Unclear]".

Table 3 holds the results of the 240×8 answers. When the first edit script was shorter, the majority of the participants also found the first script to be more helpful (270). Similarly, when the second script was shorter, more participants preferred the second script (354). When the scripts had the same size there was no clear difference and the results were more similar to a random distribution.

More formally, Pearson's chi-squared test [34] rejects our null hypothesis that the two nominal variables (script size difference in the pairs versus helpfulness) are independent ($p < 0.001$). This allows us to examine the strength of the effect that the size difference has on the helpfulness. Since the third row (same size) has no size difference and thus no effect to be measured, it is left out of the remaining evaluation. For the first and second row, Cramér's association measurement is $V = 0.32$ (where 0 indicates no association and 1 is a full association) [15]. Cohen's effect size measure is $w = V \cdot \sqrt{rows - 1} = 0.32$ [14], this indicates a medium effect of the script size difference on the helpfulness. However, w does not address the issue whether a shorter or a longer script increases the helpfulness. For that, we can use the observations from Table 3. As there are more "helpful"-votes for shorter scripts, the shorter scripts have a medium positive effect on the helpfulness.

6.6 Limitations and Threats to Validity

None of the five optimizations Θ_{A-E} uses language dependent features. Thus, they should also work for other programming languages even though the evaluation only uses Java code. This is also supported by the fact that there is no large variation in the results regardless of the type of AST we have used.

A threat to validity are possible bugs in the implementations. As far as possible we used publicly available baseline code. The parsers, the edit script generator, GT, RTED, and CD are part of the GT tool [1]. We also open-sourced our code changes (and the adaption of CD to use the results of Θ_A) to make our results reproducible. After adopting a publicly available JSync implementation to the GT framework, all four tree differencing algorithms produce comparable mappings and edit scripts whose variations are reasonable, both with and without the optimizations.

A threat to validity is the manual analysis of the script quality because the participants are members of the same university as the authors. Therefore we make the questionnaires and the raw answers available to allow re-evaluation.

7. RELATED WORK

We divide the related work into three categories, namely line differencing, tree differencing, and other approaches.

Miller and Myers [30, 31] describe the basic algorithm that is still used in the *diff* tool. Newer line differencing approaches [7, 11, 36] also detect *moved* lines but are still too coarse-grained for source code differencing.

There are several optimal tree differencing algorithms for ordered trees that do not consider *move* operations [16, 23, 39, 42]. Bille [9] provides a survey. CSLICER [26] uses a tree differencing algorithm with heuristics that does not support *moves*. RTED [33], the currently fastest optimal algorithm, has a complexity of $O(n^3)$. We discuss it throughout this paper and in our comparison.

This paper also works with and improves three publicly available tree differencing algorithms that support *moves*: ChangeDistiller that was created by Fluri et al. [19] based on an algorithm by Chawathe et al. [12], Gumtree [18] that already includes Θ_A and relies on RTED, and JSync [32] that uses a line-based *diff* to identify unchanged parts of the code but nevertheless still benefits from Θ_A.

Diff/TS [20, 21] is another tree differencing tool that supports *moves*. Its top-down step is similar to Θ_A. We are convinced that the general optimizations Θ_B-Θ_E presented in this paper are also useful for Diff/TS. But because the source code is not publicly available and as the papers are not detailed enough for a fair and competitive re-implementation, we cannot prove this.

There are also tree differencing algorithms that are tailored to specific languages, but still can benefit from our findings. For example there are several for XML [5, 13, 37], some of which also use fingerprints. Vdiff [17] is a tree differencing algorithm for Verilog HDL. There are also approaches that handle graphs instead of ASTs (e.g., UMLDiff [41], JDIFF [6], Dex [35]). With some adaptations, like breaking circles at appropriate places, some of our optimizations might also be useful for them.

8. CONCLUSION

This work introduces five general optimizations for tree differencing algorithms. We evaluate the performance of the optimizations with four state-of-the-art tree differencing algorithms (GT, RTED, CD, and JSync) on code from open-source archives. The optimizations shorten the sizes of the edit scripts for 18-98% of the 126,162 analyzed files (depending on the baseline tree differencing algorithm used).

We also introduce our novel Move-optimized Tree DIFFerencing algorithm MTDIFF. It uses a new approach to identify and map corresponding leaves between the original and modified AST. Additionally, MTDIFF uses a new metric to measure the similarity of inner nodes of the two ASTs. Our evaluation shows that MTDIFF produces even shorter scripts for around 20% of all files, independently of the type of AST used.

For reproducibility, we open-source all the measurements, including the questionnaire (https://github.com/FAU-Inf2/tree-measurements), and the source codes (https://github.com/FAU-Inf2/treedifferencing).

9. ACKNOWLEDGMENTS

This project has been supported by the Embedded Systems Institute (ESI) http://esi-anwendungszentrum.de/.

10. REFERENCES

[1] GumTreeDiff, 2016. https://github.com/GumTreeDiff/.

[2] Jenkins, 2016. https://github.com/jenkinsci/jenkins.

[3] JSync, 2016. https://sites.google.com/site/nguyenanhhoan/JSync4Public.zip.

[4] T. Akutsu. Tree edit distance problems: Algorithms and applications to bioinformatics. *IEICE Trans. Inf. Syst.*, 93D(2):208–218, Feb. 2010.

[5] R. Al-Ekram, A. Adma, and O. Baysal. diffX: An Algorithm to Detect Changes in Multi-version XML Documents. In *CASCON'05: Conf. Centre for Advanced Studies on Collaborative Research*, pages 1–11, Toranto, Canada, Oct. 2005.

[6] T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *ASE'04: Intl. Conf. Automated Softw. Eng.*, pages 2–13, Linz, Austria, Sep. 2004.

[7] M. Asaduzzaman, C. Roy, K. Schneider, and M. Di Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *ICSM'13: Intl. Conf. on Softw. Maintenance*, pages 230–239, Eindhoven, The Netherlands, Sep. 2013.

[8] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *SPIRE'00: String Processing and Inf. Retrieval Symp.*, pages 39–48, A Coruna, Spain, Sep. 2000.

[9] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1):217–239, June 2005.

[10] U. Boobna and M. de Rougemont. Correctors for XML Data. In *Database and XML Technologies*, volume 3186 of *Lecture Notes in Computer Science*, pages 97–111.

[11] G. Canfora, L. Cerulo, and M. Di Penta. Tracking Your Changes: A Language-Independent Approach. *IEEE Software*, 26(1):50–57, Jan. 2009.

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *SIGMOD'96: Intl. Conf. Management of Data*, pages 493–504, Montreal, Canada, June 1996.

[13] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE'02: Intl. Conf. Data Eng.*, pages 41–52, San Jose, CA, Feb. 2002.

[14] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 2nd edition, 1988.

[15] H. Cramér. *Mathematical Methods of Statistics*. Princeton University Press, Princeton, NJ, 1946.

[16] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An Optimal Decomposition Algorithm for Tree Edit Distance. *ACM Trans. Alg.*, 6(1):2:1–2:19, Dec. 2009.

[17] A. Duley, C. Spandikow, and M. Kim. A Program Differencing Algorithm for Verilog HDL. In *ASE'10: Intl. Conf. Automated Softw. Eng.*, pages 477–486, Antwerp, Belgium, Sep. 2010.

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ASE'14: Intl. Conf. Automated Softw. Eng.*, pages 313–324, Västerås, Sweden, Sep. 2014.

[19] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[20] M. Hashimoto and A. Mori. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE'08: Working Conf. Reverse Eng.*, pages 279–288, Antwerp, Belgium, Oct. 2008.

[21] M. Hashimoto, A. Mori, and T. Izumida. A comprehensive and scalable method for analyzing fine-grained source code change patterns. In *SANER'15: Intl. Conf. Softw. Anal., Evolution and Reengineering*, pages 351–360, Montréal, Canada, Mar. 2015.

[22] J. Kennedy and R. Eberhart. Particle swarm optimization. In *ICNN'95: Intl. Conf. Neural Networks*, pages 1942–1948, Perth, Australia, Nov. 1995.

[23] P. N. Klein. Computing the Edit-Distance between Unrooted Ordered Trees. In *ESA'98: Europ. Symp. Alg.*, pages 91–102, Venice, Italy, Aug. 1998.

[24] G. Kondrak. N-Gram Similarity and Distance. In *SPIRE'2005: String Processing and Inf. Retrieval Symp.*, pages 115–126, Buenos Aires, Argentina, Nov. 2005.

[25] H. W. Kuhn and B. Yaw. The Hungarian method for the assignment problem. *Naval Res. Logist.*, 2(1-2):83–97, Mar. 1955.

[26] Y. Li, J. Rubin, and M. Chechik. Semantic Slicing of Software Version Histories. In *ASE'15: Intl. Conf. Automated Softw. Eng.*, Lincoln, NE, Nov. 2015.

[27] B. Ma, L. Wang, and K. Zhang. Computing similarity between RNA structures. *Theoretical Computer Science*, 276(1-2):111–132, Apr. 2002.

[28] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does Automated Refactoring Obviate Systematic Editing? In *ICSE'15: Intl. Conf. Softw. Eng. - Volume 1*, pages 392–402, Florence, Italy, May 2015.

[29] N. Meng, M. Kim, and K. S. McKinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *ICSE'13: Intl. Conf. Softw. Eng.*, pages 502–511, San Francisco, CA, May 2013.

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

[31] E. W. Myers. AnO(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, Nov. 1986.

[32] H. A. Nguyen, T. T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone Management for Evolving Software. *IEEE Trans. Softw. Eng.*, 38(5):1008–1026, Sep. 2012.

[33] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *VLDB Endow.*, 5(4):334–345, Dec. 2011.

[34] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5*, 50(302):157–175, 1900.

[35] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool

for studying changes in large code bases. In *ICSM'04: Proc. Intl. Conf. on Softw. Maintenance*, pages 188–197, Chicago, IL, Sep. 2004.

[36] S. P. Reiss. Tracking Source Locations. In *ICSE'15: Intl. Conf. Softw. Eng.*, pages 11–20, Leipzig, Germany, May 2008.

[37] S. Rönnau and U. Borghoff. XCC: change control of XML documents. *Computer Science - Research and Development*, 27(2):95–111, 2012.

[38] D. Shapira and J. A. Storer. Edit Distance with Move Operations. In *CPM'02: Symp. on Combinatorial Pattern Matching*, pages 85–98, Fukuoka, Japan, July 2002.

[39] K.-C. Tai. The Tree-to-Tree Correction Problem. *J. of the ACM (JACM)*, 26(3):422–433, July 1979.

[40] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *APSEC'10: Proc. Asia Pacific Softw. Eng. Conf.*, pages 336–345, Sydney, Australia, Dec. 2010.

[41] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *ASE'05: Intl. Conf. Automated Softw. Eng.*, pages 54–65, Long Beach, CA, Oct. 2005.

[42] K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *J. on Computing (SIAM)*, 18(6):1245–1262, 1989.