Diff/TS: A Tool for Fine-Grained Structural Change Analysis

Masatomo Hashimoto and Akira Mori
National Institute of Advanced Industrial Science and Technology
1-18-13 Sotokanda, Chiyoda-ku, Tokyo 101-0021, Japan
{m.hashimoto, a-mori}@aist.go.jp

Abstract

This paper reports on a tool for fine-grained analysis of structural changes made between revisions of programs. The tool, called Diff/TS, calculates, visualizes and classifies edit operations including "moves" that will change one revision into another by means of detailed tree structural analysis on source code. Such analysis tends to be time consuming and inflexible. We have extended a general tree comparison algorithm with heuristics driven control configurable for multiple programming languages and have achieved both processing speed and analysis precision needed for investigating large-scale software projects. The tool is capable of processing Python, Java, C and C++ projects. We present several applications including software "archaeology" on a widely known open source software project and automated "phylogenetic" malware classification based on control flows. These examples suggest that tree differencing is useful for measuring distance or dissimilarity between tree structured artifacts, and offer good precision tests of the method.

1. Introduction

Identifying detailed difference between revisions of programs has many potential applications. For instance, it will help us:

- understand changes by way of neat visualization [19],
- analyze and classify code changes such as common and frequent bug fix patterns [14, 8],
- predict potential future modifications by mining change histories [20, 23],
- merge branches in ways that are suitable for target programming languages [10], and
- analyze safety of dynamically updating running programs [17, 9].

Yet, we rarely see such tools being used in practical software maintenance tasks. An obstacle lies in the difficulty in designing an efficient algorithm for computing changes that are precise in terms of evolving software. Such tools should support multiple programming languages coping with different syntactic features to cover broader range of software projects. Most tools proposed so far are limited to a single programming language and have little room for customization.

In these regards, it is natural to consider algorithms for tree-to-tree correction problems [18, 22, 4] since the syntactic structures of the program are represented by *abstract syntactic trees* (ASTs). Such algorithms extend string differencing algorithms [13] to compute an *edit sequence* composed of three basic edit operations `delete`, `insert` and `relabel` that transforms one tree into another with minimum cost, where the cost is given by the sum of the cost of each involved edit operation. The cost of the edit sequence with minimum cost is called the *edit distance*.

There are several issues to consider. Firstly, these algorithms are computationally complex (quasi-quadratic at best) and are hardly practical for real-world software projects that may well have thousands of lines of code and tens of thousands of AST nodes per compilation unit. Secondly, it often makes more sense to include `move` in representing changes between programs in addition to `delete`, `insert` and `relabel`. Thirdly, syntactic categories are ignored in these algorithms. For instance, it is possible that a `relabel` changing an integer constant to a conditional statement is generated.

In order to overcome these somewhat incompatible problems, we have designed a new algorithm combining tree differencing with configurable heuristics, and implemented a tool, called Diff/TS, for fine-grained analysis of structural changes made between revisions of programs. Features of Diff/TS include:

Fine-grained comparison: The comparison can be controlled in various syntactic granularities such as classes, functions, declarations, statements and expressions.

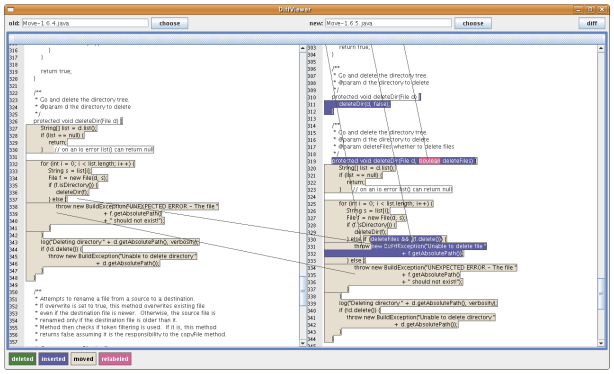

Figure 1. DiffViewer

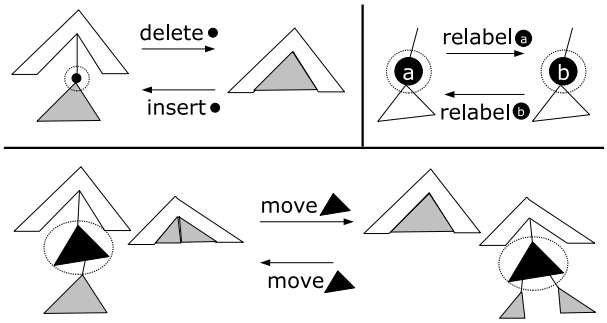

Figure 2. Edit Operations

Neat visualization: A two pane graphical user interface called DiffViewer is offered for browsing changes between revisions as shown in Figure 1. It overlays edit operations on source code texts, where change related texts are colored and sticky lines are drawn for `moves` and `relabels`.

Classification of edit operations: Since our differencing algorithm emits fine-grained edit sequence, we can classify a segment of edit operations into pre-defined types for coarse-grained change analysis as explored by Fluri and others [8].

Multilingual analysis: Programs written in Python, Java, C and C++ are accepted. The language specific heuristics can be specified in a language independent manner.

Distance measurement: Edit sequences can be used for measuring distance or dissimilarity between revisions of software.

Phylogenetic Analysis: Once the distance between each pair of revisions is calculated, a phylogenetic algorithm can be used to reconstruct evolution trees. This is particularly useful for archaeological study of historic software products and serves as a good precision test of the method.

For the last testing purpose, we have conducted a couple of experiments: reproduction of evolution process among early Emacs releases, and analysis of variation process of malware (i.e., computer viruses/worms) binaries spread in the wild. We will present compelling results of these as well as a few benchmark results to demonstrate the capability of the tool.

The rest of the paper is organized as follows. The differencing algorithm is illustrated in Section 2. Section 3 reports on the results of experiments. After related work is reviewed in Section 4, we conclude the work in Section 5.

2. Tree Differencing Algorithm

The tree edit distance problem is known to be NP-hard with `move` operations [11], which renders optimal algorithms impractical. It is polynomial time without `move` operations [18]. However, the algorithms proposed so far are quasi-quadratic at best in time complexity and quadratic in space complexity, and hence inefficient for trees with practical size. The idea of finding an optimal edit sequence with minimum cost must be abandoned for the sake of faster computation. Approximation is inevitable for concrete applications.

Our tree differencing algorithm approximates the edit distance between two rooted ordered labeled trees T_1 and T_2 by computing an edit sequence consisting of `delete`, `insert`, `relabel` and `move` that transforms T_1 into T_2 in a way that is as economical as possible.

In the following, we explain basic notations, as well as the edit operations we consider, and then describe the algorithm. For simplicity, we assign a cost value 1 to each of the four edit operations for the rest of the paper. This means that the cost of an edit sequence is given by the number of edit operations. Let T be a rooted ordered labeled tree. We denote by $nd(T)$ the set of nodes contained in T. We write $a \in T$ for $a \in nd(T)$ and $|T|$ for $|nd(T)|$. The label of node a is denoted by $lab(a)$. The root node of T is denoted by $root(T)$. If $a \in T$, we write $T\langle a \rangle$ for the subtree rooted at a. The number of children of a is denoted by $deg(a)$.

For transforming trees, we consider four edit operations, `delete`, `insert`, `relabel`, and `move` which are illustrated in Figure 2. A `delete` is denoted by $\mathrm{del}(a)$ where $a \in T$ and $a \neq root(T)$. Let p be the parent of a. The operation disconnects a from p, then inserts children of a into the position of a under p. The positions of children of p in the rightside of a shift to right accordingly. An `insert` is denoted by $\mathrm{ins}(a, b, i, s)$ where $b \in T$, $1 \leq i \leq deg(b) + 1$, and $0 \leq s \leq deg(b) - i + 1$. The operation detaches s nodes of the i-th to $(i+s-1)$-th children of b and makes them children of a, and then makes a the i-th child of b. The position of the $(i+s)$-th child of b becomes $i+1$. Note that `insert` is the opposite of `delete`. A `relabel` is denoted by $\mathrm{rel}(a, l)$ which changes the label of a to l. A `move` is denoted by $\mathrm{mov}(w, a, [b_1, \ldots, b_n], c, i, [(d_1, k_1), \ldots, (d_s, k_s)])$ where $w(> 0)$ is a move id, $a, c \in T$, $a \neq root(T)$, $b_j \in T\langle a \rangle$, $b_j \neq a$, b_j is not an ancestor of another b_h $(1 \leq j, h \leq n)$,

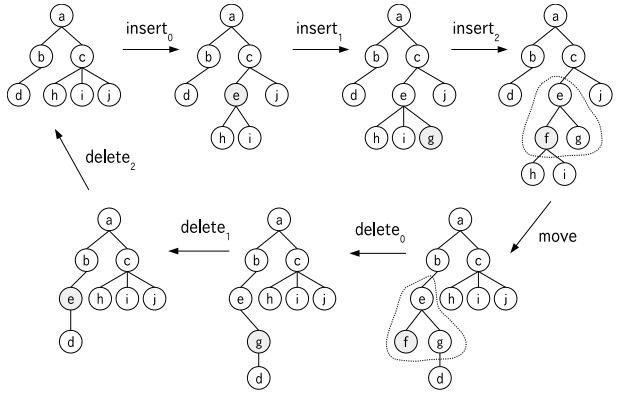

Figure 3. Examples of Edit Operations

the sequence $b_1, \ldots, b_n$ is ordered in left-to-right postorder in $T\langle a\rangle$, and $1 \leq i \leq deg(c) + 1$. Let p be the parent of a. The operation first disconnects a from p, then makes $b_1, \ldots, b_n$ children of p at the position of a. The positions of children of p in the rightside of a shift to right accordingly. Let T' be the tree rooted at a obtained by detaching $T\langle b_1\rangle, \ldots, T\langle b_n\rangle$ from $T\langle a\rangle$. The node d_j satisfies that $d_j \in T'$ and $1 \leq k_j \leq deg(d_j) + 1$ $(1 \leq j \leq s)$. The $(i + j - 1)$-th child of c becomes the k_j-th child of d_j for $1 \leq j \leq s$. Examples of edit operations are depicted in Figure 3 where $\texttt{insert}_0 = \mathrm{ins}(e, c, 1, 2)$, $\texttt{insert}_1 = \mathrm{ins}(g, e, 3, 0)$, $\texttt{insert}_2 = \mathrm{ins}(f, e, 1, 2)$, $\texttt{move} = \mathrm{mov}(1, e, [h, i], b, 1, [(g, 1)])$, $\texttt{delete}_0 = \mathrm{del}(f)$, $\texttt{delete}_1 = \mathrm{del}(g)$, and $\texttt{delete}_2 = \mathrm{del}(e)$. Note that `move` operates not only on subtrees, but also on any connected components of a tree.

Our differencing algorithm utilizes Zhang and Shashas' algorithm (ZS) [22] and maintains the size of the problem manageable by folding target tree pairs aggressively. It consists of the following steps: preprocessing, subtree comparisons, postprocessing, and edit sequence generation. In the preprocessing step, subtrees of certain categories are collapsed and shared subtrees are pruned to reduce the size of the target trees pairs. Then, the subtree comparison step calls the ZS algorithm to find matched node pairs, which are further examined for better outputs in the postprocessing step before edit sequences are finally generated.

It is noted here that the algorithm keeps track of a set M of matched pairs of nodes between target trees and we regarded M as a partial function between sets of nodes. For $(a, b) \in M$, we write $M(a) = b$ or $M^{-1}(b) = a$ as the match is always one to one.

2.1. Preprocessing

The preprocessing step consists of *subtree collapsing*, *prepruning*, and *prematching*. First, in the subtree collapsing step, collapsed form of the input trees are derived. Subtrees of certain syntactic categories are chosen as the collapse targets. In the case of Java programs, for example, nodes for classes, interfaces, declarations, methods, field declarations and blocks are collapsed. The root node of each collapsed subtree retains a digest value computed as the hash of serialized representation of the subtree for convenient identification. We denote the digest value of T by $Dgt(T)$. Expanding a node is the opposite operation of collapsing subtree rooted at the node as illustrated in Figure 4.

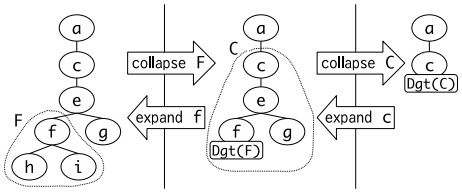

Figure 4. Collapse and Expand

Then, the prepruning step prunes and records common subtrees shared by given input trees by checking the digest values obtained in subtree collapsing. It is expected that early removal of common parts of the trees dramatically improves efficiency. It is noted that a subtree is prepruned only when its corresponding subtree in the other input tree is uniquely determined.

Finally, for T_1 and T_2, the prematching step finds and records unique one-to-one correspondences between nodes in T_1 and nodes in T_2. It is achieved in a manner similar to the prepruning step except that pruning is not performed. The recorded correspondences will be used later in the postprocessing step.

2.2. Subtree Comparisons

The whole tree comparison task is gradually divided into smaller subtree comparisons. For a pair of collapsed (sub)trees, in order to determine possible subtree pairs for further comparisons, we apply ZS to the (sub)tree pair. For T_1 and T_2, ZS computes an edit sequence with minimal cost that transforms T_1 into T_2 using edit operations other than `move`. ZS also returns a set of pairs of matched nodes. The algorithm of subtree comparison is shown in Algorithm 1.

In the algorithm, several external functions are used. $clp(T)$ denotes the set of collapsed nodes in T. $dgt(a)$ denotes the digest value assigned to node a, or $\perp$ when it is not defined yet. ZS returns an edit sequence and a set M' of pairs of matched nodes. Note that ZS uses not only labels but also digest values to equate nodes and M' may contain relabel candidate pairs of nodes. A function $cluster(T_1, T_2, M')$ partitions M' into disjoint *clusters* each of which contains maximum connected elements. If $a_1, b_1 \in T_1$ and $a_2, b_2 \in T_2$, we say (a_1, a_2) is con-

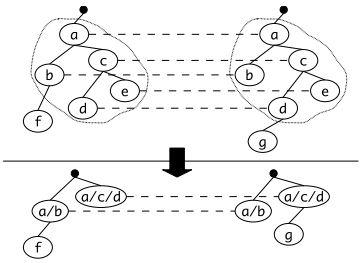

Figure 5. An Example of Flattening

nected to (b_1, b_2) when a_1 and a_2 are connected to b_1 and b_2 in T_1 and T_2, respectively. FLATTEN is another algorithm to be mentioned later for reducing the size of input trees by pruning clusters. If ZS matches a_1 with a_2 where $dgt(a_1) \neq dgt(a_2)$, and T_1 and T_2 are flat, i.e., nodes other than root are all leaves, the subtrees rooted at a_1 and a_2 are selected for further subtree comparison. They are expanded and compared again by ZS. In subtree comparison, identical pairs of subtrees are recorded to M and pruned. Unmatched collapsed nodes are expanded within user-defined size limit, although it is not explicitly shown in the algorithm. A subtree comparison continues until all collapsed subtrees are expanded or the size exceeds the limit. When the size exceeds the limit, the subtree comparison is aborted.

Algorithm 1 Subtree Comparison

```
 1: procedure COMPARE(T1, T2, M)
 2:     while clp(T1) ≠ ∅ or clp(T2) ≠ ∅ do
 3:         E, M' ← ZS(T1, T2)
 4:         for (a1, a2) ∈ M' do
 5:             if a1 and a2 are collapsed then
 6:                 A1 ← nd(T1) \ {root(T1)}
 7:                 A2 ← nd(T2) \ {root(T2)}
 8:                 if dgt(a1) = dgt(a2) then
 9:                     prune T1⟨a1⟩ and T2⟨a2⟩
10:                     M ← M ∪ {(a1, a2)}
11:                 else if ∀b ∈ A1 ∪ A2, deg(b) = 0 then
12:                     expand a1 and a2
13:                     COMPARE(T1⟨a1⟩, T2⟨a2⟩, M)
14:                 end if
15:             end if
16:         end for
17:         C ← cluster(T1, T2, M')
18:         FLATTEN(T1, T2, M, C)
19:         for a ∈ clp(T1) ∪ clp(T2) do
20:             expand a
21:         end for
22:     end while
23: end procedure
```

In the *subtree flattening* procedure FLATTEN, clusters are eliminated and recorded to further reduce the size of input trees. For example, in Figure 5, a cluster which consists of ten nodes labeled a, b, c, d, and e is eliminated and replaced by four nodes labeled a/b and $a/c/d$ where dashed lines denote matches. a/b represents a part of the path of f, and $a/c/d$ path of g. Note that node a/b in the right tree is not eliminated since the node a/b has a child node in the left tree. The node $a/c/d$ in the left tree is not eliminated for the same reason. An algorithm of flattening is shown in Algorithm 2. We assume that $[(a_1, b_1), \ldots, (a_n, b_n)]$ is sorted in postorder. $\lambda x.F$ denotes an anonymous function where F denotes a formula. A function $(x\ not\ in\ L)$ returns $true$ if x is not contained in L, and $false$ otherwise. $filt$ is a filter function for lists. $filt(f, L)$ denotes the list constructed from L by eliminating $x \in L$ such that $f(x) = false$. $chn_T(a)$ denotes the list of children of a in T. $append(L, x)$ appends x to L. $path_T(a, b)$ denotes a newly created node labeled with a path from a to b in T.

Algorithm 2 Flatten Trees

```
 1: procedure FLATTEN(T1, T2, M, C)
 2:     for [(a1, b1), ..., (an, bn)] ∈ C do
 3:         I, J ← [a1, ..., an], [b1, ..., bn]
 4:         F1, F2 = [], []
 5:         for k ← 1, n do
 6:             M ← M ∪ {(ak, bk)}
 7:             G1 ← filt((λx.(x not in I)), chn_T1(ak)))
 8:             G2 ← filt((λx.(x not in J)), chn_T2(bk)))
 9:             if G1 ≠ [] or G2 ≠ [] then
10:                 a' ← path_T1(an, ak)    ▷ an is a root of the cluster
11:                 append(F1, a')
12:                 let G1 be children of a'
13:                 b' ← path_T1(bn, bk)    ▷ bn is a root of the cluster
14:                 append(F2, b')
15:                 let G2 be children of b'
16:             end if
17:         end for
18:         prune an in T1
19:         insert F1 to the former position of an
20:         prune bn in T2
21:         insert F2 to the former position of bn
22:     end for
23: end procedure
```

2.3. Postprocessing

After all subtree comparisons are finished, we postprocess the resulting set M of matched node pairs. The postprocessing step is responsible for the following: eliminating *enclaves*, eliminating odd `relabels`, and detecting `relabels`.

An enclave in T_1 and T_2 is a pair of holes, i.e., edit operations contained in other edit operations, in a cluster of T_1 and T_2, as depicted in Figure 6. Enclaves are produced mainly because matches are fragmented by aggressive pruning in the previous steps. It is expected that edit cost is reduced by eliminating such holes by swapping edit operations. In order to detect enclaves, we employ ZS algorithm again for subtrees which have matched root nodes. ZS maximizes matches preserving parent-children and sibling relations. If more nodes in a subtree match than before, the increased part of the nodes is marked as an enclave. As we

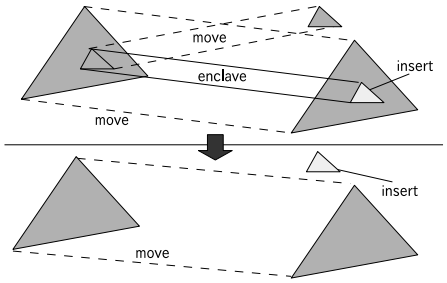

Figure 6. Enclave Elimination

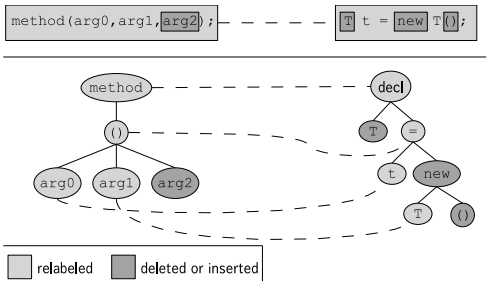

Figure 7. Odd Relabels

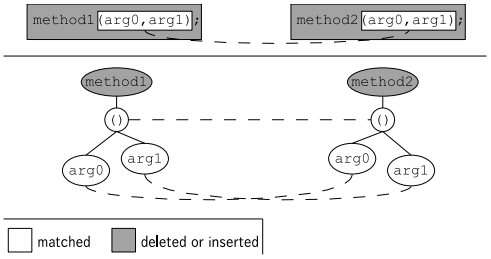

Figure 8. An Overlooked Relabel

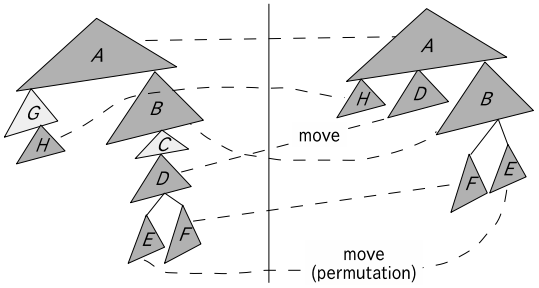

Figure 9. Move Generation

use ZS to find enclaves, enclave elimination step may get stuck at ZS when it is given excessively large inputs. In order to cope with the situation, we only feed subtrees of size less than specified threshold to ZS. The threshold may be altered according to the syntax of the language.

Tree differencing algorithms for minimum edit costs tend to produce `relabels` rather than `deletes` and `inserts` since a single relabel operation is usually assigned lower cost than the sum of the cost of `delete` and `insert` operations. However, in some cases, `deletes` followed by `inserts` are more understandable than the corresponding `relabels`. For example, consider `relabels` in pieces of Java source code shown in Figure 7. The corresponding ASTs are shown below in the figure. Actually, we observed a number of `relabels` of this kind in a set of test cases taken from several open source products. They hinder understanding of the result significantly. A reason for this is that those algorithms relate nodes in different syntactic categories, such as a method invocation and a variable declaration. We regard such `relabels` as odd and eliminate them. Suppose that a node label $lab(a)$ is relabeled to $lab(a')$. If they belong to different syntactic categories, the `relabel` is decomposed into `delete` and `insert`. In addition, we can explicitly specify the allowed pairs of syntactic categories for `relabels`.

Whereas odd `relabels` are eliminated by relabel elimination, some meaningful `relabels` might be overlooked as demonstrated in Figure 8. Each of lost `relabels` is decomposed into `delete` and `insert` and buried in the edit sequence. Such `deletes` and `inserts` should be dug up and glued. We employ heuristics for it: a node adjacent to exactly matched nodes tends to also match either exactly or by `relabel`, and it is dangerous to glue `deletes` and `inserts` on nodes for anonymous structures such as blocks and arguments. Of course, generated `relabels` between different syntactic categories except allowed pairs are filtered out from the candidate set.

2.4. Edit Sequence Generation

Finally, the edit sequence for T_1 and T_2 is generated from the computed matching M. Generation of edit operations other than `moves` is rather trivial. The nodes in T_1 and T_2 which have no matching companions are deleted and inserted, respectively. For $(a_1, a_2) \in M$, a_1 is relabeled $lab(a_2)$ if $lab(a_1) \neq lab(a_2)$. Thus, we concentrate on the generation of `moves`. First, we detect root nodes of moved trees. For $(a_1, a_2) \in M$, if the nearest ancestor of a_1 in the domain of M and that of a_2 in the range of M differ, a_1 is regarded as a moved root. Hence, $root(D)$ in Figure 9 is identified as being moved while $root(H)$ is not. Also *permutations*, namely `moves` which change the order of siblings (ignoring deleted or inserted trees) are detected such as E in the figure. After permutations are detected, boundaries of moved trees are determined.

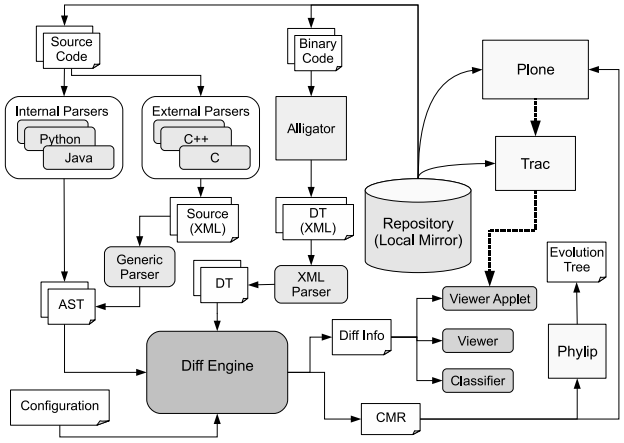

Figure 10. Diff/TS System

2.5. Complexity

It is not difficult to confirm that overall worst-case time complexity cannot be more than $O(n^2)$. Most of the algorithms are linear. Among others, subtree comparison is even linear since the size of input for ZS is bounded to a certain threshold. Generation of moves has relatively expensive complexity of $O(n^2)$ since sorting and sequence differencing are involved.

3. Experiments

Several experiments are conducted using the Diff/TS system, whose architecture is illustrated in Figure 10. The target source code is parsed to obtain ASTs in XML format, which are then passed to the differencing engine.

3.1. Small Test Cases

We first report on six small test cases taken from two well-known open source products. They contain considerable changes including structural moves. In these tests, we compare the edit costs computed by Diff/TS with the ones computed by three other available differencing tools. The samples are five pairs of source code of A) WebappClassLoader, B) ResourceFactory, C) Request, D) Http11Protocol, and E) DeltaSession, between Apache Tomcat[1] versions 5.5.15 and 5.5.16, and a pair of source code of F) Move, between Apache Ant[2] versions 1.6.4 and 1.6.5. The size of the samples varies from 143 to 2,523 in LOC and from 233 to 5,073 in the number of AST nodes.

[1] `http://tomcat.apache.org/`
[2] `http://ant.apache.org/`

All samples are written in Java. The tools that are compared with Diff/TS are XyDiff [5], DeltaXML [7], and XMLDiff[3]. XyDiff is a very fast differencing engine written in C++. It is designed for massive volume of XML documents and can detect `moves`. We used version 2.6.1. DeltaXML is a commercial differencing engine used for managing changes in XML data. We used version 5.0 of DeltaXML Core. XMLDiff is an implementation of Chawathe's algorithm [3] which can detect `moves`. It is written in Python and C. We used version 0.6.8. Note that these algorithms compute edit costs in restricted use of edit operations. For example, XMLDiff only considers `move` operations on *subtrees* and other operations on *leaves*. A larger number of operations are needed in general for computing an edit sequence compared to Diff/TS. Since all these engines are designed for XML documents, we prepared XML documents equivalent to the ASTs generated by Diff/TS for comparison. The edit costs and the total time in seconds are shown in the following table.

	XyDiff	DeltaXML	XMLDiff	Diff/TS
A	146	3052	241	101
B	180	86	121	80
C	17	17	31	13
D	57	50	76	43
E	264	261	235	94
F	595	691	736	312
total edit cost	1259	4157	1440	643
total time	0.819	20.213	95.313	12.262

We used a PC with Intel Xeon CPU (1.6GHz) with 2GB RAM running under Linux kernel 2.6.20. Note that the edit costs computed by each algorithm are adjusted for guaranteeing comparisons under the same condition. The total cost computed by Diff/TS is much smaller than those by other engines. In terms of obtaining precise information about changes, Diff/TS is better than the others while XyDiff is by far the fastest in processing speed. It is found by manual inspection that only Diff/TS was able to detect moves correctly. It is also found that the precision of XyDiff is not good enough for the phylogenetic experiments in 3.3.

We tried testing the pure ZS algorithm on the same samples, but it runs out of memory on sources larger than 400 LOC (1,000 in the number of AST nodes). This suggests that memory efficiency is a major obstacle for optimal algorithms in practice.

3.2. Analyzing Larger Code Bases

Here we apply Diff/TS to larger code bases. We built local mirrors of source code repositories of two open source projects: MythTV[4] and Boost C++ Libraries[5] as

[3] `http://www.logilab/projects/xmldiff/`
[4] `http://www.mythtv.org/`
[5] `http://www.boost.org/`

of March, 2008 to feed Diff/TS. For each project, released revisions are extracted from `tags` or `tags/release` directories in the mirror. We analyzed the changes between contiguous released revisions. Most part of the source code is written in C++ or C. It is too costly to build all revisions of programs for preparing complete parse trees since it in general requires processing include files. We used a parser based on CDT [6] for the experiment, which is able to parse source code under limited information in such a case where not all include files are present. Several statistics are listed below.

	MythTV	Boost
revisions	25	67
files	21,718(8,861)	248,774(31,480)
file comparisons	6,209	21,677
LOC	9,672,416(4,622,723)	38,456,232(5,066,585)
missing LOC	22,014(11,311)	1,121,133(191,305)
LOC coverage	99.77(99.76)	97.08(96.22)

The set of pairs of files to be compared is computed by the directory differencing mode of Diff/TS. In parentheses, the files used for the comparisons are counted. The missing LOC denotes the number of lines which the parser failed to parse because of a variety of parse errors or some internal errors in CDT. It took about 6 hours and about 33 hours to parse the selected revisions of MythTV and Boost, respectively.

For each target project, the total edit cost is also computed by the three engines other than XMLDiff that is too slow for this experiment. The edit cost and time in minutes (in parentheses) for the comparison are shown in each entry of the following table.

	XyDiff	DeltaXML	Diff/TS
MythTV	4,900,810(34)	11,891,155(230)	3,771,516(600)
Boost	4,146,016(37)	3,180,617(470)[7]	2,139,585(203)

The same machine as for the small test cases is used. We observed that Diff/TS slows down on huge (more than 50,000 nodes) inputs in the case of MythTV.

3.3. Phylogeny

Phylogeny is a new method of bio-informatics stemmed from molecular biology, which aims at mathematical modelling of specific evolution based on the difference of gene structures [15]. Even though there are obvious differences between software and species, if we regard software products as species and source code as genes, evolution of software can be investigated in the same way as evolution of species.

Phylogenetic analysis starts by computing distance between each pair of revisions to generate a *distance matrix*. Diff/TS employs the distance value which we call *cost-match ratio* (CMR) obtained by dividing total edit cost by the number of matched pairs of nodes. Diff/TS performs this round-robin computation for all revisions in the requested project. An open source tool called PHYLIP [8] is used for generating evolution trees from a distance matrix. There are two major algorithms available in PHYLIP: Neighbor Joining (NJ) Algorithm and Fitch-Margoliash (FITCH) Algorithm. PHYLIP. In general, the NJ algorithm is much faster than the FITCH algorithm since FITCH tries to minimize sums of distance in evolution trees. In our experience, both algorithms give similar results for most cases. However, for a heterogeneous mix of samples, they tend to disagree in slight favor of FITCH. We always try both algorithms and compare their results. In fact, a result by FITCH is presented for the example of old Emacsen in Section 3.3.1 and a result by NJ for the malware example in Section 3.3.2.

We used a PC with Quad-Core Intel Xeon CPU (3.0GHz) with 16GB RAM running under Linux kernel 2.6.22 for experiments below.

3.3.1 Archaeology

For testing Diff/TS, we have chosen Emacs editor as a target software. We have collected twenty six versions of early Emacs editors including ancient Emacs-13.8.5 released in 1985. Pairwise distance of the releases are computed to produce a distance matrix by Diff/TS. To produce phylogenetic tree, the distance matrix is fed to the FITCH algorithm of the PHYLIP package for inferring phylogenetic trees. In a phylogenetic tree, a node with descendants represents the most recent common ancestor of the descendants, and the horizontal edge lengths correspond to distance (dissimilarity) estimates. It took about 32 hours to compute the distance matrix with a single process, and about 1.5 seconds to compute phylogenetic tree. Throughout the experiment, 12,174 file comparisons are performed. There are more than 200 trees that contain more than 10,000 nodes. The resulting phylogenetic tree is shown in Figure 11. Each leaf is labeled with a version number where `mu`, `ne`, `ep`, `le`, and `xe` denotes branches of Emacs, namely Mule, NEmacs, Epoch, Lucid Emacs, and XEmacs, respectively. Remarkably enough, the phylogenetic tree is consistent with the "Emacs Timeline" [21] recorded by one of the main contributers of Lucid Emacs and XEmacs while the distance matrices computed by GNU diff and XyDiff failed substantially to reproduce a "correct" timeline. The example provides a good evidence that accurate tree differencing is useful for automated analysis in software archaeology.

[6] `http://www.eclipse.org/cdt/`

[7] More than 1,000 pairs of ASTs escaped from edit cost calculation due to the failure caused by DeltaXML.

[8] `http://evolution.gs.washington.edu/phylip.html`

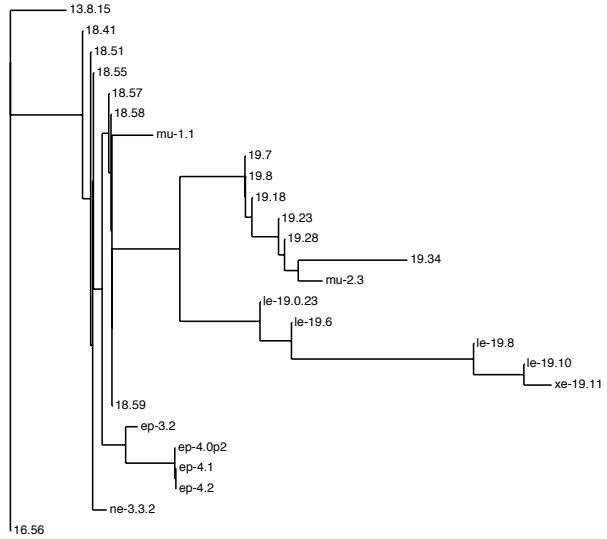

Figure 11. Phylogenetic Tree of Old Emacsen

3.3.2 Malware Phylogeny

In this part, we present the results of phylogenetic analysis of malware behaviors. Although malware is in fact software that may have a number of revisions (called variants for malware), source code is hardly available. To make matters worse, binary code is usually encrypted and/or obfuscated in order to escape from analysis tools such as disassemblers, debuggers and virtual machines. This means that static binary code analysis does not help to understand malware behaviors at all.

We use a tool called Alligator developed by the second author for automated malware scrutinization [12]. Alligator defeats encryption and obfuscation and is able to generate control flow graphs (CFGs) observed in a software controlled virtual execution environment.

Since CFGs are graphs and are not amenable to methods based on tree differencing, we have extended Alligator to export *dominator trees* [6] in an XML format computed from CFGs hoping that the essence of malware behaviors is captured in them. In a dominator tree, a node represents a basic block [9] and may be labeled with names of API functions called in the corresponding basic blocks. Nodes having no labels are treated alike as "anonymous".

From malware samples captured by a Nepenthes honeypot system [10] over a year period, approximately ninety samples identified as Sdbot worms by BitDefender [11] are fed to Alligator, which then generates CFGs up to the point where network related behaviors are visible, computes dominator trees, and passes them to the differencing engine of Diff/TS.

At the same time, three clusters of Sdbot worms are identified in the samples by manual inspection:

- Cluster 1: samples that display socket based botnet interactions, encrypted by normal packers,
- Cluster 2: samples that open internet connections using WININET library and start a service process with duplicated selves, encrypted by normal packers,
- Cluster 3: samples that have the same behaviors as Cluster 1, but protected by a powerful commercial executable protector called Themida [12].

Figure 12 shows an output evolution tree together with the above clustering overlaid on it. Note that distance is computed using the NJ algorithm. In the tree, each leaf is labeled by 1) first ten characters of an MD5 hash string of the malware file, and 2) a timestamp embedded in the header of the executable if it is valid.

Now it is easy to confirm that the analysis result is consistent with the clustering identified by manual inspection as samples belonging in the same cluster are located in the same branch of "evolution". Let us take a closer look into Cluster 3. Firstly, the worm seems to have evolved from Cluster 1 to become more complex ones as they get protected by Themida. Secondly, the worm seems to get more complex as Themida upgrades to newer versions while the core malicious behaviors stay the same. Thirdly, the timestamp does not match the way the worm changes. This is not surprising since it it a well known fact that most hackers only alter the way the malware code is wrapped by packers/protectors since that is enough for cheating anti-virus scanners for a short while. We can not expect that phylogenetic study reveals real malware evolution in order of time. Rather, it reveals how malware code is shared and reused in hacker's community. Let us turn to the shadowed part in Figure 12. It contains samples that are not Sdbot variants as their behaviors are considerably different from those of other Sdbot samples. That is why those samples are placed in a remote spot in the phylogenetic tree. The situation is clarified by clustering samples using the same distance matrix. Figure 13 shows a clustering result calculated by R statistical computing package [13].

In summary, our tools Diff/TS and Alligator are able to analyze and classify malware to suggest variation process in full automation. It turned out that our tools are able to demonstrate a marginal area in malware identification. Edit costs calculated by Diff/TS for dominator trees computed

[9] A basic block is a sequence of code that enjoys a "single entry and single exit" property.

[10] http://nepenthes.mwcollect.org

[11] http://www.bitdefender.com

[12] To the best our knowledge, there is no tool that can automatically unpack Themida protected executables yet.

[13] http://www.r-project.org/

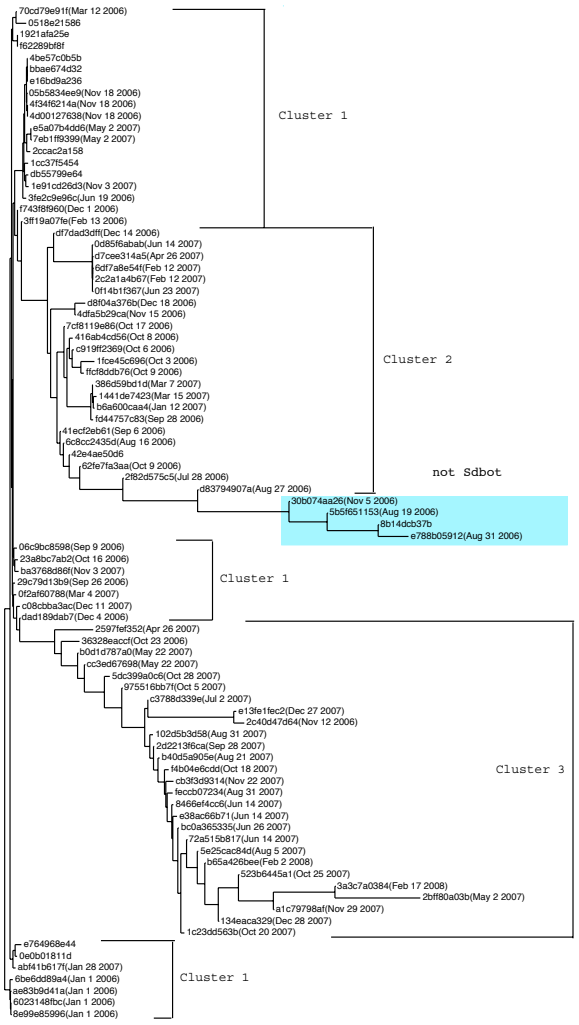

Figure 12. Phylogenetic Tree of Sdbot Worms

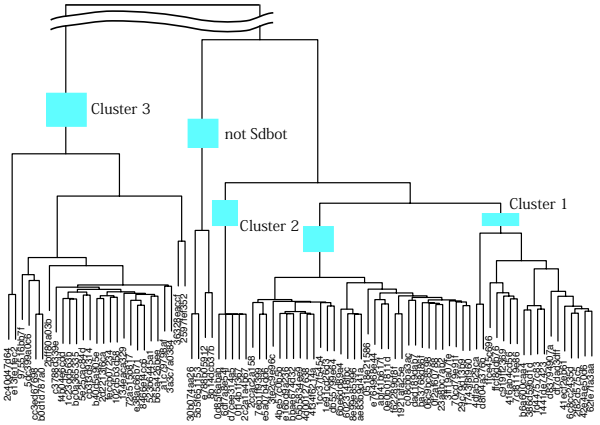

Figure 13. Clustering of Sdbot Worms

by Alligator well approximate similarity among malware behaviors.

The benchmark data for this experiment are: 279 minutes for calculation of a distance matrix and 350 milliseconds for generating a phylogenetic tree using the NJ algorithm. The node numbers for Themida protected samples are as large as 30,000.

4. Related Work

Several tree differencing algorithms are known to compute the minimum edit cost. Variants of Kuo-Chun Tai's algorithm [18] compute the minimum edit cost between rooted ordered labeled trees with three basic edit operations [22, 4]. Application to tree-structured data including RNA secondary structures, XML documents, and software source code has been explored. However, these algorithms are quasi-quadratic at best in time complexity [4] and have suffered from computational inefficiency as the input size increases.

Three basic edit operations, namely `delete`, `insert`, and `relabel`, are not sufficient for reconstructing actual changes made between revisions of software. Barnard and others [1] extended Zhang and Shashas' algorithm with *swapping* of subtrees which can be regarded as a special case of our `move`. Such extension complicates the problem for practical use even if it guarantees minimality of the edit cost. More operations are necessary for representing a `move` operation in our method and the computation speed is even slower than ZS.

Efficient optimal differencing algorithms can be formulated by restricting use of edit operations [16, 3, 2]. For instance, mmdiff [2] restricts basic edit operations on *leaves* only. Since optimality is relative to imposed restrictions, the effectiveness of such algorithms depends on target application domains and it is difficult to compare them with other approaches such as the one proposed in the current paper.

There are several tree differencing algorithms specialized for source code of the programs that utilize domain specific heuristics [14, 8]. They exhibit good processing speed and quality of analysis for a wide range of practical problems. Dex [14] proposes an automated method of collecting detailed information about syntactic and semantic changes in C programs to identify specific bug fix patterns. ChangeDistiller [8] classifies source code changes according to pre-defined taxonomy such as addition of parameters to function definitions, and type changes in variable declarations, for better understanding of evolving natures of software. Both Dex and ChangeDistiller are bound to single language (C and Java respectively) although application to other languages is suggested in the papers.

5. Conclusion and Future Work

We developed a novel tool, called Diff/TS, for fine-grained structural change analysis between versions of programs. The tool calculates, visualizes and classifies edit operations including "moves" on source code that will transform one revision into another by means of detailed tree structural analysis on parsed source code. Such analysis tends to be time consuming and inflexible, however, we have extended a general tree comparison algorithm with heuristics driven control and achieved both processing speed and analysis precision needed for investigating large-scale software projects. Diff/TS covers wide range of software written in Python, Java, C and C++, as well as dominator trees derived from control flow graphs gathered in virtual execution. The capability of Diff/TS is benchmarked against other similar tools on several test cases taken from open source projects. We find the result positive, especially in terms of performance and quality of the analysis.

Another type of evaluation is provided by the fact that edit cost can be used for measuring distance or dissimilarity between revisions of software. Applying a tree differencing algorithm to phylogenetic analysis reveals the accuracy of the algorithm. We chose old/ancient version of Emacs editor as a target for phylogeny and successfully reconstructed an evolution tree for it by Diff/TS. We have also presented an experiment of malware phylogeny. Malware's behaviors are extracted from its control flows in the form of dominator trees. Phylogenetic analysis revealed a compelling variation process of a botnet family called Sdbot.

Future work includes addition of more complex edit operations such as `copy` and performance improvement on huge trees. Annotated differencing in the style of Dex is also promising. For instance, semantic information such as scope of variables may be embedded in ASTs for semantic change analysis.

References

[1] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical Report 1995-372, School of Computing, Queen's University, 1995.

[2] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., 1999.

[3] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[4] W. Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40(2):135–158, 2001.

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

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] DeltaXML. DeltaXML – Managing Change in an XML Environment. `http://www.deltaxml.com/`.

[8] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[9] M. Hashimoto. A method of safety analysis for runtime code update. In *Proceedings of the 11th Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 60–74. Springer, 2006.

[10] J. J. Hunt and W. F. Tichy. Extensible language-aware merging. In *Proceedings of the 18th IEEE International Conference on Software Maintenance*, pages 511–520, 2002.

[11] F. Magniez and M. de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007.

[12] A. Mori, T. Izumida, T. Sawada, and T. Inoue. A tool for analyzing and detecting malicious mobile code. In *Proceedings of the 28th International Conference on Software Engineering*, pages 831–834, 2006.

[13] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[14] 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 *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, 2004.

[15] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[16] S. M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[17] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 183–194, 2005.

[18] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[19] W. Yang. Identifying Syntactic Differences Between Two Programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[20] A. T. T. Ying, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004. Member-Gail C. Murphy.

[21] J. Zawinski. Emacs timeline, 1999. `http://www.jwz.org/doc/emacs-timeline.html`.

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

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.