Fine-grained Change Detection in Structured Text Documents

Hannes Dohrn
Friedrich-Alexander-University
Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
+49 9131 85 27621
hannes.dohrn@fau.de

Dirk Riehle
Friedrich-Alexander-University
Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
+49 9131 85 27621
dirk@riehle.org

ABSTRACT

Detecting and understanding changes between document revisions is an important task. The acquired knowledge can be used to classify the nature of a new document revision or to support a human editor in the review process. While purely textual change detection algorithms offer fine-grained results, they do not understand the syntactic meaning of a change. By representing structured text documents as XML documents we can apply tree-to-tree correction algorithms to identify the syntactic nature of a change.

Many algorithms for change detection in XML documents have been propsed but most of them focus on the intricacies of generic XML data and emphasize speed over the quality of the result. Structured text requires a change detection algorithm to pay close attention to the content in text nodes, however, recent algorithms treat text nodes as black boxes.

We present an algorithm that combines the advantages of the purely textual approach with the advantages of tree-to-tree change detection by redistributing text from non-overlapping common substrings to the nodes of the trees. This allows us to not only spot changes in the structure but also in the text itself, thus achieving higher quality and a fine-grained result in linear time on average. The algorithm is evaluated by applying it to the corpus of structured text documents that can be found in the English Wikipedia.

Categories and Subject Descriptors

I.7.1 [**Computing Methodologies**]: Document and Text Editing; F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems; E.1 [**Data**]: Data Structures

General Terms

Algorithms, Design, Performance

Keywords

XML; WOM; structured text; change detection; tree matching; tree differencing; tree similarity; tree-to-tree correction; diff

DocEng'14, September 16–19, 2014, Fort Collins, Colorado, USA.

ACM 978-1-4503-2949-1/14/09 ...$15.00
http://dx.doi.org/10.1145/2644866.2644880.

1. INTRODUCTION

Change detection is important in many applications. It can be used for temporal queries (when did a certain change occur in an article) or to maintain an index (just update what we know has changed, don't re-index). It can be used for merging documents that have diverged from a common ancestor or to visualize the changes between two revisions of a document (in version control systems). By only storing differences between revisions, change detection algorithms can be used to compress data, and in classification it can help to understand the intent of an author or identify unwanted contributions like spam.

We focus on change detection in structured text documents as generated by word processors or markup languages like HTML, in order to help authors understand the nature of changes by visualization and to automatically classify changes. Structured text documents are composed of mainly text interspersed with formatting elements (e.g. bold font, hyperlink, section heading) that we refer to as syntactic markup. To detect changes between two revisions of a document textual differencing algorithms are commonly used. However, these tools treat structured text as a sequence of characters, without paying special attention to its syntactic markup. This leads to misalignment of content between the revisions under comparison and makes it difficult to discover the syntactic nature of changes.

These problems can be avoided by using tree-to-tree correction algorithms that are applied to the syntax tree representation of structured text documents. However, unlike textual differencing tools, which generate fine-grained information on the character level, available tree differencing algorithms treat continuous blocks of text as atomic entities. Assume a sentence in which bold formatting is applied to a word. If the sentence was stored in a single text node previously, the text node will be split in the new revision and in between the two halves of the text the bold formatting node is inserted, with the formatted word as its only child node. This and similar changes are common in structured text, however, current tree differencing algorithms are unable to properly address this situation since they can only perform a one-to-one mapping between the nodes of the old and new tree.

Another problem we face when applying tree differencing tools to structured text is the focus on speed and greedy matching behavior and in some cases the reliance on XML intricacies like IDs or keys to find matching nodes. Since we aim to support humans and automatic text classification the quality of the generated change set is important and overly greedy behavior for the sake of speed is not constructive.

Our main contribution is the novel treatment of text leaves. As detailed in the following chapters we analyze unmatched text and subdivide text nodes to achieve a fine-grained matching where other algorithms report the removal and insertion of whole text nodes. We further modify existing differencing algorithms to perform less greedy and emphasize ancestor relationships between nodes when searching for a matching.

Our algorithm has the following features:

- It operates on an XML representation of structured text called WOM [6]. That is, it operates on rooted, ordered labeled trees in which only attributes and leaves can have values.
- It does not assume the presence of IDs or other unique identifiers that would otherwise simplify the matching process.
- It produces an edit script that features the operations insert, delete, move and update.
- It computes an edit script in near linear time and space on average.

The remainder of the paper is structured as follows. In section 2 we discuss related work. We then introduce terms and definitions and define the input data that our algorithm operates on in section 3. Afterwards we present and analyze our algorithm in section 4. Finally, we evaluate the algorithm in section 5 and conclude in section 6.

2. RELATED WORK

Often software for describing and visualizing differences between two versions of the same document relies on a purely textual, line-based representation of the document. A prominent example is the GNU diffutils[1] package which uses an algorithm described by Myers in [11]. Myers presents an algorithm that solves the *longest common subsequence* (LCS) problem in $O(nd)$ time, where n is the combined lengths of two strings A and B and d is the size of the minimum edit script that transforms A into B.

The textual approach is appealing for its simplicity and its broad applicability. Any document that has a textual, line-based representation (which also includes almost any kind of XML document) can be efficiently compared using this algorithm. On the other hand, such a generic algorithm does not consider syntactic subtleties of the document. Many document formats in use today, however, exhibit a rich syntactic structure either implicitly or explicitly.

Many algorithms have been devised to calculate the differences between two trees, also called the tree-to-tree correction problem [17]. One way to classify existing tree differencing algorithms is by the type of tree they operate on (ordered/unordered). Another way is to ask whether the algorithm strictly minimizes a cost function to produce a minimal edit script or if it uses a heuristic approach that orientates itself on a cost metric but is not guaranteed to produce a minimal edit script for that metric. Although we want to operate on ordered trees exclusively, it is instructive to see what algorithms exist for unordered trees as well. In the following discussion n_i denotes the number of nodes, $d_{\max,i}$ the maximum depth and l_i the number leaves of tree T_i. If no index is given, the quantities are summed up from both trees.

[1] `http://www.gnu.org/software/diffutils/`

In [17] Tai defines the distance $d(T_1, T_2)$ between two trees as the cost of the minimum cost edit sequence s, according to a restricted cost function $c(s)$ and presents an algorithm that solves this minimization problem in time $O(n_1 \cdot n_2 \cdot d^2_{\max,1} \cdot d^2_{\max,2})$. The algorithm generates the edit operations update, delete and change label.

In [20] Zhang and Shasha improve on Tai's algorithm with sequential time in $O(n_1 \cdot n_2 \cdot \min(d_{\max,1}, l_1) \cdot \min(d_{\max,2}, l_2))$, while supporting the same edit operations.

Chawathe et al. are the first to introduce a heuristic algorithm called *LaDiff* in [3]. Instead of only considering labeled trees they also take node values into account to deal with LaTeX documents. The finest level of subdivision that Chawathe et al. use are sentences; the leaf nodes are therefore large text nodes. They are also the first to propose approximate text node matching using an edit cost function to calculate the similarity of two text nodes. After first matching the leaf nodes of a document using the LCS algorithm by Myers they then propagate matches to the inner nodes, again using the LCS algorithm.

Their heuristic assumes that input documents contain only few identical nodes. The produced edit script is always correct, however, if the assumption does not hold the result may be sub-optimal. They try to compensate for identical nodes with a post-processing step and achieve an overall run time complexity in $O(ne + e^2)$, where e is the weighted edit distance and typically $e \ll n$. The algorithm generates the edit operations insert, delete, update and move.

In [2] Chawathe et al. assert that the change detection problem for unordered labeled tress that considers move and copy operations is $\mathcal{NP}$-hard. They propose a heuristic algorithm, called *MH-DIFF*, that transforms the tree-to-tree correction problem to the problem of finding the minimum cost edge cover of a bipartite graph. Its worst case performance is in $O(n^3)$ but most often requires time in $O(n^2)$.

Cobéna and Marian [4] focus on performance in terms of speed and space. Their heuristic algorithm *XyDiff* makes use of node IDs and first matches nodes that have the same ID. Then identical subtrees are matched by computing hash values for subtrees to allow fast look-ups, always matching the next biggest subtree first. The remaining nodes are matched by propagating matches bottom-up similar to [3], followed by an additional lazy-down pass. The matching rules during the propagation pass are kept simple but greedy which can lead to bad mappings as observed by [18], especially if there are many small identical subtrees. The algorithm generates the edit operations insert, delete, update and move and runs in $O(n \log(n))$ time. It therefore does not slow down when faced with lots of changes.

Xu et al. [19] transform the problem of finding a match between trees to finding a match between so-called key trees. In a key tree each node is a label and all paths from the root to a leaf generate a unique sequence of labels. If a label is not unique among its siblings, it is replaced by a value from the original node's subtree that is expected to be unique. Their algorithm *KF-Diff+* supports the edit operations insert, delete and update and runs in $O(n)$. It can be extended to support node moves among siblings (called alignment).

Wang et al. [18] implement an XML change detection algorithm called *X-Diff* that assumes that left-to-right order among siblings is not important and instead focuses on ancestor relationship. They drastically reduce the search space by only matching nodes whose parents match as well and

who have the same signature, where signature(x) = label(p_1)/label(p_2)/.../label(p_{i-1})/type(x) and ($p_1, \ldots, p_{i-1}, x$) is a path from the root node p_1 to node x. They achieve a run time complexity in $O(n)$ and support the edit operations insert, delete and update.

Lindholm et al. [10] transform both trees into sequences of nodes. To match the trees they slide windows of decreasing size over the sequences and search for matches using a rolling hash function. They achieve worst case performance in $O(n^2)$ if both documents have nothing in common and $O(n)$ if both documents are identical. The supported edit operations are insert, delete, update and move. We refer to their algorithm under the name *FcDiff*.

Fluri et al. [7] apply various improvements to the *LaDiff* algorithm [3] to adapt it for detecting and classifying changes in source code. Their algorithm *change distilling* produces the same edit operations as the original.

Rönnau et al. [13] present the algorithm *DocTreeDiff* which, similar to the *LaDiff* algorithm [3], uses a leaf-based LCS to compute an initial matching. Unlike *LaDiff* they operate on hash values of leaf nodes which also incorporate the node's depth. Using the initial matching, strucutral changes among the parent nodes are encoded as updates while all remaining un-matched nodes are recorded as deletes and inserts. Their algorithm performs in $O(lD+n)$, where D denotes the number of edit operations required.

Rönnau et al. analyzed requirements for version control of XML documents produced by word processors or spreadsheets in [14]. For a broader selection of algorithms and more in-depth information on the individual algorithms one can consult the survey from Peters [12] on change detection in XML trees or the survey by Bille [1] on solutions for the general tree edit distance problem.

3. PRELIMINARIES

In the following chapters variable names a_i, s, t, u and x refer to nodes in T_1, b_i, s', t', u' and y refer to nodes in T_2, where T_1 and T_2 refer to the tree representation of the old and new document respectively.

3.1 Edit Script and Tree Format

An edit script generated from two documents A and B is a list of operations that when applied to document A transforms it into document B. Which edit operations an algorithm uses in an edit script depends on its design. We are not aware of a standardized format for presenting edit scripts and rely on the operations introduced by Chawathe in [3].

The design of tree nodes differs between implementations. Our algorithm is designed to expect trees similar to XML documents that consist of two types of nodes: elements and text nodes. An element node n has a label l(n) (called tag in XML), a set of attributes, where an attribute is a (name, value) pair and an ordered list of children. Elements can be leaf nodes if their list of children is empty. Text nodes, on the other hand, cannot have children and are therefore always leaf nodes. They have a value v(n) but no label or attributes. It is possible to map other tree designs onto XML trees.

3.2 Input Data Format

We use wiki articles as test corpus for our algorithm. Articles in wikis are usually written in a markup language called wiki markup. For the wiki markup dialect used in MediaWiki we have implemented a parser in [5] that produces an Abstract Syntax Tree (AST). This representation is further converted into a wiki independent exchange format called Wiki Object Model (WOM) [6], which we will use as input data in our evaluation.

A distinctive feature of the WOM is the optional support of so-called Round Trip Data (RTD) tags. These tags preserve the syntactic markup from the original source and guarantee that the formatting of the original source can be restored from WOM trees after a transformation. For illustration consider the following piece of wiki markup:

```
'''Tree''' differencing
```

which translates to the following WOM document:

```
<article><body><p>
  <b><rtd>'''</rtd><text>Tree</text><rtd>'''</rtd>
  </b><text> differencing</text>
</p></body></article>
```

Three ticks denote the use of bold font and the ticks themselves are stored as RTD information. As this short example shows, the WOM format has some idiosyncrasies that require special attention. Our differencing algorithm does not operate on the WOM directly but uses an adapter mechanism that allows processing of various data structure designs. In the case of the WOM the adapter hides `<rtd>` elements when the algorithm asks for text nodes explicitly, however, they are reported in a traversal of the tree. When traversing the tree, the adapter reports `<rtd>` and `<text>` nodes as leaf text nodes instead of elements that contain XML text nodes. Finally, it also represents certain element attributes (e.g. the target attribute of a link) as child nodes.

The reason for presenting attributes as elements is that our algorithm does not consider attributes when evaluating the similarity of nodes. This adjustment therefore guarantees that important information that is stored in attributes is considered by the algorithm while keeping the algorithm itself simple. The reasons for the other adjustments are explained when the algorithm is described. All of the mentioned variations are reversible and other document formats may not require any adaptions at all.

3.3 Challenges of Structured Text Documents

In the English Wikipedia we find the following text in the article "Danish pastry" [2]:

```
Danish pastry is formed of flour, milk,
eggs, and butter -- especially butter.
```

In the next revision an editor has turned the words "flour", "milk", "eggs" and "butter" into links so that they point to the respective articles in Wikipedia:

```
Danish pastry is formed of [[flour]], [[milk]],
[[egg]]s, and [[butter]] -- especially butter.
```

Removing or adding styles happens frequently in structured documents as [13] note as well, and line-based diff algorithms can cope with this change well and report only the insertion of square brackets, however, they are unaware of the syntactic implications.

Another common practice is the rearrangement of text within a sentence or paragraph. Line-based diff algorithms report the movement of text as insertion and deletion since they often don't support move operations. However, they usually only report the text span that has actually changed. They don't report the removal or insertion of a whole sentence if only part of it has changed.

[2]Revision 657740 and 1019401

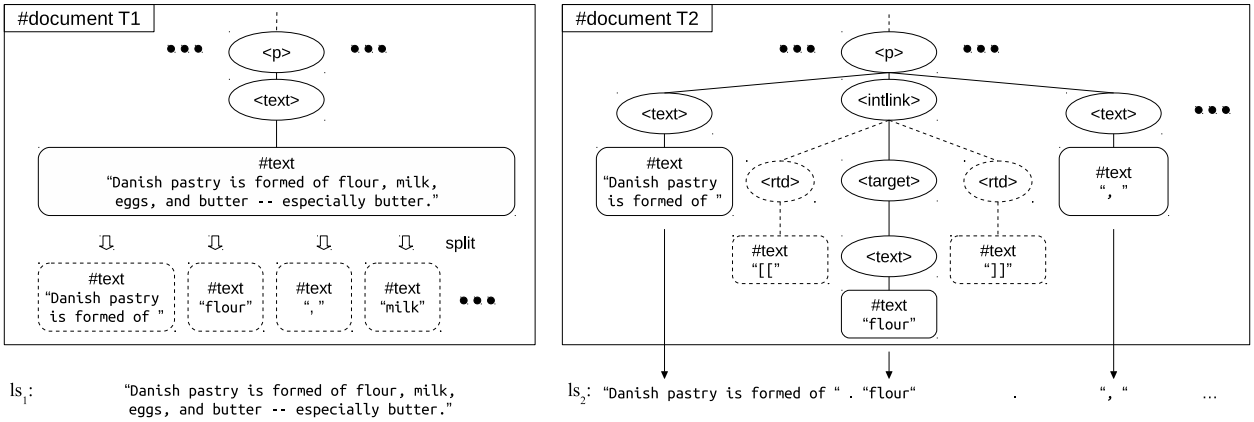

Figure 1: Distribution of text over nodes in the old and new revision of an article and the corresponding leaf strings. Only a part of the new tree (T_2) is shown for clarity. Both trees have more paragraphs than those depicted. In T_1 the dashed nodes indicate the possible splits of the text leaf.

In both of the above cases tree differencing algorithms behave differently. An excerpt of the corresponding WOM trees for both revisions is shown in figure 1. The article "Danish pastry" has more than just one paragraph (<p>) but we do not show them for reasons of clarity. In the old revision the whole text was contained within one text node. In the new revision text nodes are interspersed with internal links (<intlink>) which contain a target node (that is also the title) and two <rtd> tags that store the syntactic markup.

If we only extract and concatenate the <text> nodes and not the <rtd> nodes from these trees, we obtain the same string for both documents. In other words: No changes with regard to textual content have occurred between the two revisions. Tree differencing algorithms, however, can only match nodes one-to-one. Algorithms that perform approximate text node matching might associate the initial "Danish pastry is formed of" text node in the new revision with the entire <text> node in the old revision, because it yields the smallest edit distance between all candidates. Still all other nodes in the new tree remain unmatched and the algorithms usually report the complete removal of the whole paragraph and new node insertions. While [13] identify this challenge as well, they do not present a solution to this particular problem.

The same problem occurs when text is moved within a text node, as shown in figure 2. In that case the number of text nodes stays the same, however, the old and new text nodes don't match any more and an insertion and a removal is reported. In the case of approximate matching the result depends on the nature of the change within the text node. If the edit distance between old and new text stays below a preconfigured threshold, an update is reported. Otherwise, insertion and removal is reported. In the following section we describe how we address these challenges.

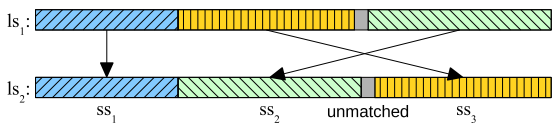

Figure 2: Two leaf strings ls_1 and ls_2, that are possibly spread over multiple text nodes, were concatenated. The NOCS algorithm found three common substrings s_1, s_2 and s_3 that have been rearranged by the edit. A small part of the leaf strings is not part of a common substring and the corresponding text node splits will not be matched.

4. CHANGE DETECTION WITH HDDiff

Most tree differencing algorithms run through a sequence of processing phases. First a matching between the two trees is computed. It pairs a node from the old tree with exactly one and only one partner node from the new tree. Nodes in the old tree that do not have a partner are reported as deleted nodes. Nodes in the new tree that do not have a partner are reported as inserted nodes. Nodes that are partners but have different parent nodes (according to the matching) are reported as moved nodes. If nodes have the same parents but have changed position among their siblings, they are reported as moved nodes as well, however, the process of finding those cases is called *alignment*. Except for one exception ([2]) we are not aware of heuristic algorithms that support copy operations (one-to-many mappings). Once a mapping is computed, a list of operations called an edit script is generated that transforms the old tree into the new tree.

4.1 Matching Substrings of Text Nodes

In section 3.3 we have illustrated what happens if editors mark up words or move text around among paragraphs. While the textual content stays the same (the order may change) the atomic treatment of text nodes does not allow existing differencing algorithms to properly address the situation. As solution we introduce the **split** operation. This operation is an auxiliary edit script operation. It is not reported to the user since it doesn't change the content of a document, however, it allows us to internally operate on substrings that originally were part of a bigger text node.

When looking at the example in figure 1 again it becomes clear that by splitting the original and only text node into a sequence of smaller text nodes we can match the complete original document. We only have to report the newly inserted links and their RTD information in the new document, but no text will be reported as deleted or inserted. By concatenating the splits we can restore the text from the original text node, which is why we regard this operation as effectless.

If we have found a good splitting of text nodes that allows us to match a greater portion of old and new content, we apply the operation to the trees and add the necessary operations to the edit script. The relevance of the split operations in the edit script depends on the purpose of the edit script. If the edit script is used to report information about changes to the user, splits can be ignored. If the edit script is used to transform a

document to its new revision, splits have to be applied since other edit operations will depend on the already split nodes.

The split operation is also the reason why the WOM tree adapter presents <rtd> and <text> elements as text leaves and not as XML elements that contain XML text. If our algorithm would split the XML text nodes and not the combination of XML element and XML text, the XML text splits would become the children of only one <text> element. However, in a WOM tree every XML text needs its own <text> parent element.

4.1.1 Finding a Good Split

Our goal is therefore to find those text leaves in both documents that, when split, will allow us to precisely match unchanged textual content in both revisions. A naive approach is to split all text nodes into individual words. This, however, is difficult when dealing with languages that do not support simple word segmentation (e.g. they don't require spaces between words) and it would also force matches by aligning words from arbitrary locations in the document.

Instead we search for continuous substrings that are shared by both documents and satisfy certain requirements (e.g. are of sufficient length or contain enough words). To this end we concatenate the text from all text leaves into the leaf strings ls_1 for T_1 and ls_2 for T_2 as illustrated in figure 1. Between these strings we search for *non-overlapping common substrings* (NOCSs) [16]. Once we have a set of NOCSs, we examine the text leaves from which a NOCS originates and split the nodes in such a way that the tree-to-tree correction algorithm can build a one-to-one mapping between the text leaves.

4.1.2 Finding all Common Substrings

To find all common substrings from which we compute the NOCSs we use *Suffix Arrays* (SA) and *Longest Common Prefix* (LCP) information. A suffix array *sa* for a string s is an array of indices, where each index sa_i points to the first character of a suffix $s_{sa_i \ldots n}$ in s with $n = \text{len}(s)$. The array is ordered in such a way that it refers to suffixes in lexicographical order. Kärkkäinen et al. show in [8] how to compute a suffix array in $O(n)$.

Suffix arrays can be augmented with longest common prefix (LCP) information. LCP information assigns a pair of consecutive suffix array indices (sa_{i-1}, sa_i) an additional number lcp_i that indicates the length of the longest common prefix $s_{sa_{i-1} \ldots sa_{i-1}+lcp_i}$ or $s_{sa_i \ldots sa_i+lcp_i}$ that both consecutive suffixes share. When a suffix array has been computed, LCP information can be added in $O(n)$ as shown by Kasai et al. in [9]. By iterating through the suffix array by decreasing LCP length, we obtain a list of substrings ordered from longest to shortest.

Using this tool set we can compute the longest common substrings within one string [15]. To compute the longest common substrings between two strings s_1 and s_2, we concatenate both strings:

$$s_{1,2} = s_1 \,.\, "\$_1" \,.\, s_2 \,.\, "\$_0"$$

where "." is the concatenation operator and "$\$_0$" and "$\$_1$" are unique terminator characters. The terminator character $\$_0$ is required by the SA algorithm. Terminator $\$_1$ is used to separate the two strings. Since $\$_1$ cannot be part of one of the strings it is unique within the concatenated string and assures no substrings cross the terminator. To guarantee that a substring is shared by both strings s_1 and s_2 we have to make sure that the two associated suffixes are located to the left and right of the separator character. This part of the algorithm is implemented in *saLcpBucketSort* in listing 3. For

```
findNOCSs(ls1, ls2):
  n1 = len(ls1); n2 = len(ls2)
  input = ls1 + '$1' + ls2 + '$0'
  sa = computeSuffixArray(input)
  lcp = computeLcp(sa, input)
  buckets = saLcpBucketSort(sa, lcp, n1)
  return greedyCover(buckets, n1, n2)

saLcpBucketSort(sa[], lcp[], n1):
   L1: for i = 1 to len(lcp):
      len = lcp[i]
      # Only accept long enough substrings
      if len < minLen: continue L1
      start1 = sa[i-1]
      start2 = sa[i]
      # Skip duplicates
      j = i + 1
      while (j < lcp.length) and
          (lcp[j] == len): ++j
      if j > i + 1:
        continue L1 with i = j
      # Only accept substrings from both strings
      if (start1 < n1) == (start2 < n1):
         continue L1
      # Correct start1 and start2
      if (start2 < start1):
         (start1, start2) = (start2, start1)
      start2 -= n1 + 1
      # Bucket sort by len
      buckets[len].add(new CS(start1, start2, len))
   return buckets

greedyCover(buckets, n1, n2):
   # Initialize cover arrays
   for i in (0, n1]: covered1[i] = false
   for i in (0, n2]: covered2[i] = false
   # From longest to shortest substring
   for bucket in reverse(buckets):
      # For every common substring
      L2: for cs in bucket:
         # Substring alreay covered?
         if covered1[cs.start1] or
            covered2[cs.start2]: continue L2
         if not isValid(cs.start1, cs.len):
           continue L2
         L3: for j in (0, cs.len]:
            k1 = cs.start1 + j
            k2 = cs.start2 + j
            # Substrings already partially covered?
            if covered[k1] or covered[k2]:
               cs.len = j
               break L3
            covered1[k1] = covered2[k2] = true
         # Add NOCS
         result.add(cs)
   return result
```

Figure 3: Finding all non-overlapping common substrings.

computing the suffix array and LCP information please refer to [8] and [9].

Now we found all common substrings shared by ls_1 and ls_2, including substrings of substrings. Finding an optimal set of non-overlapping common substrings for ls_1 and ls_2 is $\mathcal{NP}$-hard [15]. Instead we use a greedy approach. In `greedyCover` in listing 3 we first accept the longest common substring. Then we accept the next longest substring unless it overlaps. If the next substring should overlap with an already accepted substring, we shrink the substring until it no longer overlaps. This process continues until no more appropriate substrings remain. Substrings are only considered appropriate above a certain length and structure which is checked by `isValid`.

4.1.3 Splitting Nodes

Once we have obtained a non-overlapping set of substrings we can split the nodes in both trees as required. This step is facilitated by another data structure that was built when the two leaf strings were concatenated from the leaf nodes. In order to locate a substring within one of the trees, we con-

struct two arrays that for every character in the leaf strings ls_1 and ls_2 contain the information from which node and from which position in the node's text the character came. With this information at hand we split nodes as detailed in *splitNodesWithNocs* in listing 4.

```
splitNodesWithNocs(nocs):
  # Split nodes if nocs doesn't start at node boundary
  curNode1 = node in which nocs starts in T1
  if nocs does not start at node boundary in T1:
    split(curNode1, at position where substring starts)
    curNode1 = right node of split
  curNode2 = node in which nocs starts in T2
  if nocs does not start at node boundary in T2:
    split(curNode2, at position where substring starts)
    curNode2 = right node of split

  # Find node discontinuities in nocs
  endNocs = false
  L1: while not endNocs:
    i = search for next position where the nocs crosses
      a node boundary in T1 or T2 or where the nocs ends

    # Find out which nodes we have to split
    endNocs = end of nocs reached?
    if endNocs:
      break1 = node continues in T1 at end of nocs
      break2 = node continues in T2 at end of nocs
    else:
      break1 = nocs crosses node boundary in T2 (!)
      break2 = nocs crosses node boundary in T1 (!)

    # Split nodes
    leftNode1 = curNode1
    if (break2 and not break1 and not endNocs) or
        (break1 and endNocs):
      split(curNode1, at position where string crosses
        node boundary in T2 or where nocs ends)
      curNode1 = right node of split
    leftNode2 = curNode2
    if (break1 and not break2 and not endNocs) or
        (break2 and endNocs):
      split(curNode2, at position where string crosses
        node boundary in T1 or where nocs ends)
      curNode2 = right node of split
    if break1:curNode1=node in which nocs continues in T1
    if break2:curNode2=node in which nocs continues in T2

    # Match left parts of split
    if break1 or break2 or endNocs:
      match(leftNode1, leftNode2)
```

Figure 4: Splitting nodes after a non-overlapping common substring (NOCS) is found. The parameter *nocs* contains information about the starting positions in the leaf strings and the length of the NOCS.

Every time a node is split into a left and a right part within the loop `L1`, the left part is matched to its partner node in the other tree. An implementation has to make sure that the boundary splits to the left and to the right of the substring retain their partners, if any. This procedure guarantees that all nodes and split nodes that are part of the common substring are matched after this phase is over. To speed up the search for the next discontinuity position i one can use a binary search.

Complexity analysis: Let n be the length of the input to the respective algorithm. The SA and LCP algorithms are shown to require time in $O(n)$ in [8] and [9]. *saLcpBucketSort* has one loop that runs over the length of the LCP array and requires time in $O(n)$. In *greedyCover* the innermost loop runs at most twice for every character in the shorter leaf string and therefore runs in time $O(n)$. Finally, in *splitNodesWithNocs* the search for the next discontinuity examines at most every character in the leaf string and runs in time $O(n)$. Hence the whole splitting process runs in time $O(n)$, where n is the length of the two leaf strings combined.

4.2 Embedding the Split Algorithm into a Tree Differencing Algorithm

While matching substrings of nodes can be done in linear time, it still is expensive since it is a character based process with high constant costs and documents usually have considerably more characters than leaf nodes. On the other hand, big portions of the old and new document usually remain unchanged between revisions. Therefore we can drastically reduce the search space before we start looking for substrings by identifying the unchanged portions of the documents.

By computing a mapping between old and new tree every differencing algorithm identifies the unchanged portions of the documents. A first outline of the complete algorithm therefore starts by computing a mapping with an existing differencing algorithm in phase one. In phase two we concatenate yet unmatched text leaves to leaf strings for which we compute NOCSs. Using the NOCSs, nodes are split and matched. If nodes were not matched in phase one, their parents and siblings often cannot be matched in phase one either. This requires us to complete the mapping in phase three and four by propagating the matches found in phase one and two to parents and siblings.

Most algorithms we are aware of are not suitable for this task because they are computationally too expensive [2, 3, 7] or not applicable since they make assumptions that do not hold for our data [3, 18, 19]. The *XyDiff* algorithm [4] is a good candidate since its phases two to four map well to phases one, three and four of our outline. In the next sections we will describe how we adapted the *XyDiff* algorithm to finalize the mapping between the old and the new tree and how we integrated the edit script building from *LaDiff* into our own algorithm.

4.2.1 Phase 1 - Reducing the Search Space

In a precomputation step the *XyDiff* algorithm computes weights and hash values for subtrees in both T_1 and T_2. We adopt this step and compute a hash value h_n for every subtree that is rooted at node n. Text leaves compute their hash value from the text they store. All other nodes compute a hash value from their label and then combine this value with the hash values of their children. The weight of a text leaf is a function of the length l of its text. Inner nodes compute their weight by summing up the weights of their children and adding a constant w_{inner} for themselves. While this is subject to tuning, we use the text length l directly as weight function and set $w_{\text{inner}} = 3$.

Next *XyDiff* matches subtrees by comparing the hash values from T_1 with the hash values from T_2, starting with the heaviest subtree. If there is more than one candidate to match a subtree, simple and greedy heuristics help to decide which two subtrees are matched or the subtrees remain unmatched at this point. After matching a subtree *XyDiff* immediately tries to propagate the match to the ancestor nodes of the subtree. How many ancestor nodes are matched depends on the subtree's weight.

Greedy matching and propagation heuristics can easily lead to mismatched node, as [18] note and as our own observations confirm. Especially when dealing with many small subtrees or with many small identical subtrees. On the other hand we already reach our primary goal of reducing the search space by only considering big subtrees without duplicates, that can be matched unambiguously. Hence we modify Cobena's algorithm and only consider subtrees without duplicates that are above a certain weight (we use a minimum

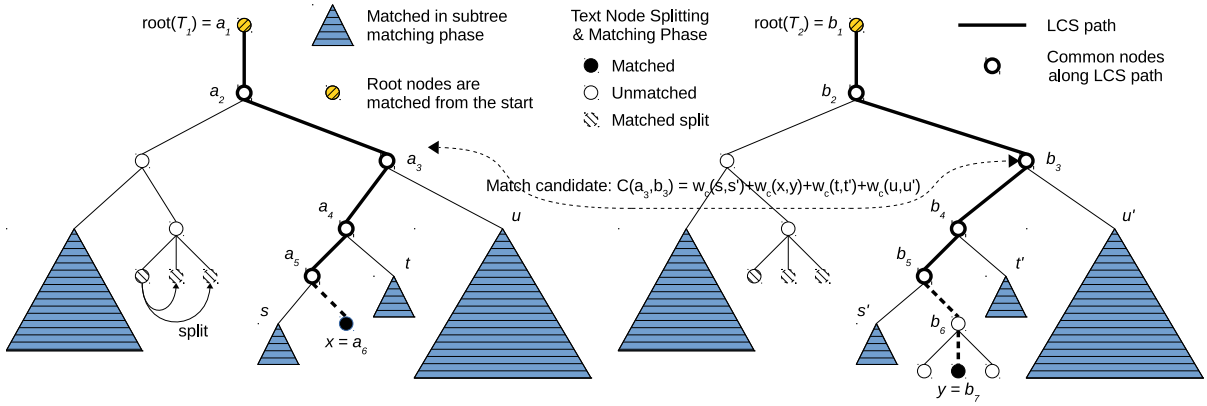

Figure 5: Two document trees T_1 and T_2 during step 3 (propagating matches to ancestors). Large subtrees have already been matched in step 1, and in step 2 a text node has been split into 3 smaller text nodes that were immediately matched. The algorithm is currently examining node y, distributing the common weight $w_c(x, y)$ to all ancestors that are part of the LCS of the paths $[\text{root}(T_1) \dots x]$ and $[\text{root}(T_2) \dots y]$. After all candidates have been gathered, match candidates $C(a_2, b_2)$, $C(a_3, b_3)$, $C(a_4, b_4)$ and $C(a_5, b_5)$ are assigned the combined weight of all of their respective descendant shared subtress.

weight of 12) and we do not match ancestors based on subtree matches.

Since element attributes do not feature in the subtree hash computation we have to check node attributes for changes which we report as node updates in the edit script. After checking subtrees for real equality, to deal with the possibility of hash collisions, and after recording node updates, our algorithm will not revisit matched subtrees.

Complexity analysis: The precomputation of weights and subtree hashes requires time in $O(|T_1| + |T_2|)$. Actually matching subtrees using hash maps for fast look ups requires time in $O(|T_2|)$. Sorting by weight requires time in $O(n_s \cdot \log(n_s))$, where n_s is the number of shared heavy subtrees (excluding subtrees of subtrees). Since usually $n_s \ll |T_2|$, phase one requires $O(|T_1| + |T_2|)$.

4.2.2 Phase 3 - Propagating Matches to Ancestors

In figure 5 we illustrate the situation in an exemplary pair of documents after phase one and phase two have run. The largest subtrees have been matched in phase one and in phase two the text in unmatched leaves was concatenated and the search for NOCSs has led to splits and matches between leaves. At this point, two kinds of nodes remain unmatched: ancestor nodes of matched subtrees and text leaves, and subtrees that could not be matched in phase one because they have duplicates or their weight is too small.

This leads us back to *XyDiff*'s propagation rules to match ancestors from step one and their BULD (Bottom-Up Lazy-Down) matching phase. To avoid mismatching nodes and generating needless move operations we propose the following procedure. It is motivated by *XyDiff*'s propagation rules combined with *LaDiff*'s LCS breadth-first matching of inner nodes with the same label.

Step 3.1) Assume we found two matching nodes x and y from both trees. All pairs of ancestor nodes from T_1 and T_2 along the paths from the root nodes to x and y are potential candidates for a match. In order for a pair of ancestor nodes to be considered as partners their labels have to be equal. Furthermore, if children have been moved between revisions, a parent node in one tree can have multiple potential partners in the other tree. In order to decide which pair of candidates is matched we accumulate the weight of the already matched descendant subtrees that the candidates share.

Let $C(x, y) \rightarrow w_c$ be a map of candidate tuples $(x, y) \in (T_1, T_2)$ onto a weight $w_c \in \mathbb{N}_{\geq 0}$. Such a mapping implies that the two mapped candidate nodes x and y share common descendant subtrees that have at least a combined weight of w_c in each tree. We can see in figure 5 that after step two, if an inner node is matched, all its descendants are matched as well down to the leaves. Consequently the new tree T_2 is traversed pre-order until we reach a matched node $y \in T_2$ and the path $p_2 = (b_2, b_3, \dots, b_{j-1})$ from node $\text{root}(T_2) = b_1$ to $y = b_j$ is stored as a sequence of nodes. For the old tree T_1 we also build a path $p_1 = (a_2, a_3, \dots, a_{i-1})$ from $\text{root}(T_1) = a_1$ to $x = a_i$, where $x \in T_1$ is the matched partner of y. Myers LCS algorithm is applied to both sequences and two nodes are considered equal by the LCS algorithm if their labels match. We get $s = \text{lcs}(p_1, p_2), s_i \in (T_1, T_2)$ for $1 \leq i \leq |s|$, the longest sequence of nodes from both paths that share the same label. Let w_c be the weight shared by the subtrees rooted at x and y. For each $s_i \in s$, if $s_i \in C$ then update the mapping $C(s_i) \rightarrow C(s_i) + w_c$. Otherwise add $C(s_i) \rightarrow w_c$ to the mappping.

After we have processed node y in the described way we do not descend into its subtree but continue with the traversal until we reach the next matched node.

Step 3.2) Once the traversal is complete we have a mapping of node pairs with an associated common weight. Each pair is a candidate to be matched. We sort the candidates by descending common weight w_c and match the heaviest pair first. We then continue with the next heaviest pair and match its two nodes unless one or both of the nodes have already been matched by a previous candidate.

By now we have propagated the matches from step one and two to ancestor nodes. We use a greedy strategy as well, however, we make sure that it considers the weights of all matched children of a pair of nodes instead of making a decision by looking only at the heaviest child.

Complexity analysis: Let $n = \max(|T_1|, |T_2|)$. If we assume that the trees are balanced, their maximum height can be approximated by $h = \log(n)$. The cost for $\text{lcs}(s_1, s_2)$ is $O(ld)$, where $l = |s_1| + |s_2|$ and $d \in (1 \dots l)$ is the edit distance between both sequences. In the worst case the LCS computation requires $O(l^2)$, in the best case the algorithm finishes in $O(l)$. The bigger the subtrees are that were matched in step one, the shorter the paths from the root to a subtree become and the LCS effort approaches $O(1)$. Sorting partner candidates

by weight requires $n_s \cdot \log(n_s)$, where n_s is the number of subtree candidates. Thus the ancestor pass requires time in $O(n)$ to $O(n_s \cdot h^2)$.

4.2.3 Phase 4 - Building an Edit Script

When looking at figure 6 we can see that some nodes and subtrees are still not matched. Furthermore, we also have not discussed the generation of an edit script yet. In this section we explain how we match the remaining nodes and at the same time build an edit script. We adapt the algorithm by Chawathe et al. [3] and use absolute child node indices computed from the new document instead of a context dependent index that Chawathe et al. call k. This allows us to match remaining nodes and subtrees and generate edit script operations in the same traversal of the tree since it doesn't require a finished mapping of both trees up front.

The types of nodes and subtrees that remain unmatched and the reasons therefor are explained in figure 6. The details of the algorithm are explained in listing 7. We proceed by traversing T_2 pre-order, where n_2 is the current node in T_2 and n_1 is its partner, if it has been matched. We then build sequences of unmatched children for n_1 and n_2 and compute the LCS according to subtree hash equality (`subtreeLCS`). Pairs of subtrees in the LCS are then matched in the same way as in phase one. The subtree LCS step is optional, however, it improves edit script quality since it prevents the following step from matching a subtree root node with a single unmatched node by label although a better match between two complete subtrees would have been possible.

In the next step we test the children of n_1 and n_2 for proper alignment. This can be expensive if both nodes have many misaligned children. However, since we have to perform this step anyway, it presents a good opportunity to also match nodes that are aligned by label. To this end we compute the LCS of all children of n_1 and n_2, where two nodes are considered equal if they are already partners or have the same label (`matchedOrLabelLCS`). In the following loop we generate move operations for matched but misaligned nodes and match yet unmatched nodes that occur in order and have the same label.

Now we visit all children of n_2. If a child has still no partner in T_1, an insert operation is generated. Otherwise we check each pair of matched children for changed attributes or text content and generate update operations accordingly. Then we descend to the child of n_2 that was just processed.

Finally, after the traversal of T_2 is complete, we traverse T_1 and for every node that does not have a partner in T_2 we generate a delete operation (`traverseAndDelete`). This step concludes the *HDDiff* algorithm.

Complexity analysis: The most expensive step is the LCS computation over all children of both nodes. In the worst case the majority of all nodes are children of a single parent and have been completely replaced between old and new revision. In this case phase four requires time in $O(n^2)$, where $n = |T_1| + |T_2|$. In the best case the trees are well balanced and we can assume a constant number of children per inner node and no alignments. Then phase four requires time in $O(n)$.

5. EVALUATION

Our focus is better support of change detection in structured text documents and a fine-grained analysis of changes between two document revisions in reasonable time. To validate that our algorithm meets these goals, we use a sample of articles from the English Wikipedia. Initially we have randomly selected 400,000 revisions from the English Wikipedia that have a predecessor revision. Since the parser software sometimes fails to parse a revision 371,451 pairs of revisions remain after conversion to the WOM XML format.

We have applied our *HDDiff* algorithm as well as the *XyDiff* [4] and the *FcDiff* [10] algorithm to the data, and called the algorithms "HDDiff", "XyDiff" and "FcDiff" in the charts respectively. In order to compare the algorithms' results we parse in the edit scripts and analyze them for the number of character insertions and deletions they require. Sometimes *XyDiff* timed out and sometimes our parser was not able to understand *FcDiff*'s edit script format. If we failed to obtain an edit script analysis from any of the algorithms we removed the sample for all algorithms. After this step 360,246 pairs of revisions remain. Furthermore our sample contains various kinds of content from Wikipedia. We tell apart articles from other content by only selecting revisions from the *main namespace*, after which 266,233 pairs of revisions remain.

First we examine the quality of *HDDiff*'s edit script in figure 8. If mostly textual changes are performed, character-based diff algorithms outperform conventional tree diff algorithms in the number of character insertions and deletions that the edit script requires to transform the old into the new revision. This is due to the fact that conventional tree diff algorithms cannot deal with situations in which text from one node is spread over multiple text nodes in the new revision, even though the textual content does not change. However, if content is moved inside a document, tree diff algorithms can outperform a character-based algorithm that does not support move operations and which instead has to report the movement of content as deletion in the old and insertion in the new revision. Since *HDDiff* combines features from both worlds, we expect it to not only outperform other tree diff algorithms in the number of character insertions and deletions, but also to outperform Myers textual diff algorithm for some documents, which is confirmed by figure 8.

By applying the three algorithms to pairs of revisions stored in the WOM XML format we can compare the change detection performance of the algorithms as a whole. As shown in figure 8 our algorithm (green, finely dashed) requires significantly less character insertions and deletions than *XyDiff* (brown circles) and *FcDiff* (red squares) on average. This is especially remarkable since our algorithm is specifically designed to avoid move and align operations if it is not clear whether such an operation conforms with the modification that was actually applied to the old revision. *HDDiff* rather issues insertions and deletions in such a situation, a restriction that does not apply to *XyDiff* and *FcDiff*.

We further want to evaluate how well our phases three and four, which are the part of our algorithm that solves the tree-to-tree-correction problem, perform, compared to other tree diff algorithms. To this end we use *HDDiff* to split text nodes in each pair of revisions as required for a one-to-one mapping of text nodes. We also attach XML IDs to pairs of split text nodes since this mapping information is available to our algorithm in step three as well, however, only *XyDiff* can make use of this information. The algorithms *XyDiff* and *FcDiff* are then applied to the modified WOM XML and are called "XyDiff+" and "FcDiff+" in figure 8. While both algorithms can improve with the split text nodes and *FcDiff* almost draws even with the textual LCS algorithm, *HDDiff* still outperforms the other algorithms. This supports our

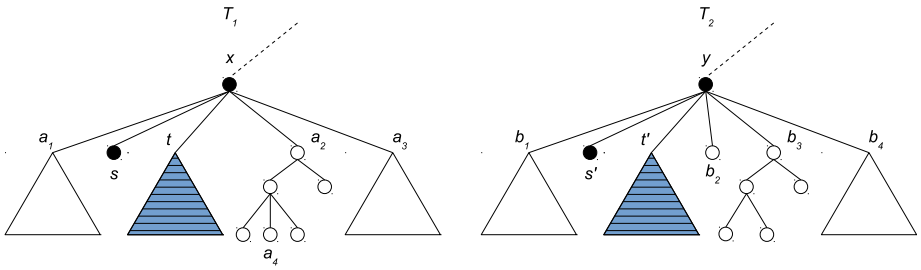

Figure 6: Two document trees T_1 and T_2 after step 3.2 (propagating matches to ancestors). (s, s') and (t, t') caused the match of ancestors (x, y). a_1, a_3, b_1 and b_4 are all identical but were not matched earlier due to the exclusion of duplicates. b_2 is inserted in T_2 and the subtrees at a_2 and b_3 are identical except for the deleted node a_4. Since the individual nodes of the subtrees have not enough weight none of them were matched.

claim that phase three and four of our algorithm are well chosen.

Next we examine the performance of *HDDiff* in terms of speed[3]. The upper chart in figure 9 gives an impression of *HDDiff*'s speed depending on the size of the input documents. However, since the speed depends on multiple factors, a clear trend is not discernible. To confirm that *HDDiff* requires nearly linear time on average, we have computed a least squares fit of a linear model that depends on (a) the combined number of nodes from both documents, (b) the minimum number of nodes from both documents, (c) the number of nodes that were initially matched in phase one n_{st} and which are a coarse measurement of the similarity of both documents and (d) the combined lengths of the leaf strings from unmatched nodes n_{ls} that are used to compute the NOCSs. The result is plotted in the lower chart of figure 9.

The R^2 measure of the fit is 0.78 and our model therefore confirms, that our algorithm works in linear time on average. When analyzing the phases individually, precomputation, greedy subtree matching and NOCSs computation behave strictly linear and are well predictable. Phase three and four are difficult to predict with the variables from our model and are responsible for almost all the remaining variance. When confronted with degenerated trees (e.g. long lists of items that are all children of a single parent and many alignment operations have taken place) phase four can lead to super-linear behavior.

6. CONCLUSION AND FUTURE WORK

We have presented a tree-to-tree correction algorithm that is specifically tailored to structured text documents. The algorithm pays special attention to the fact that text documents tend to feature large text leaves in which many of the modifications occur. Existing algorithms that treat text nodes as atomic elements therefore can only report removal or insertion of whole text nodes where purely textual differencing tools can report changes on the character level. We introduce an algorithm that offers the advantages of both approaches, by adding a novel node splitting step which allows the subsequent tree-to-tree correction algorithm to perform a fine-grained analysis and indication of the differences.

Since we focus our efforts on the support of users in understanding changes and in the automatic classification of changes, we take special care to avoid needless move operations by emphasizing ancestor relationships in the matching algorithm. We prefer that insertions and deletions of minor subtrees are reported instead of spurious moves. Unlike other works in this domain speed is not our primary concern. Still our algorithm delivers solid performance in near linear time on average.

In future work we want to investigate other tree-to-tree correction algorithms that follow the text splitting phase. Another direction of research is the simplification of the algorithm and the reduction of processing passes. To improve classification performance we would like to investigate support for copy operations and duplicates.

The implementation of *HDDiff* will be made available upon publication at `http://sweble.org/projects/hddiff`.

[3]All tests were run on an Intel Xeon Processor E5-2630 (15M Cache, 2.30 GHz) with the Oracle Java HotSpot VM 1.7.0_51

7. ACKNOWLEDGEMENT

We would like to thank Georg Dotzler for getting us started with his implementation of the *Change Distilling* algorithm.

8. REFERENCES

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

[2] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *ACM SIGMOD Record*, volume 26, pages 26–37. ACM, 1997.

[3] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *ACM SIGMOD Record*, volume 25, pages 493–504. ACM, 1996.

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

[5] H. Dohrn and D. Riehle. Design and implementation of the sweble wikitext parser: unlocking the structured data of wikipedia. In *Proceedings of the 7th International Symposium on Wikis and Open Collaboration*, pages 72–81. ACM, 2011.

[6] H. Dohrn and D. Riehle. Wom: An object model for wikitext. Technical report, Technical Report CS-2011-05, University of Erlangen, Dept. of Computer Science, 2011.

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

[8] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

```
topDown(root1, root2):
  checkUpdate(root1, root2)
  topDownRec(root1, root2)
  traverseAndDelete(root1) # not elaborated

topDownRec(n1, n2):
  if n2 is leaf: return
  # We don't have to process matched subtrees
  if n2 is matched subtree: return

  if n1 != None:
    # Match duplicate subtrees below n1 and n2
    s1, s2 = unmatched children of n1, n2
    s = subtreeLCS(s1, s2)
    for c in s: matchSubtree(c)

    # Match by label and generate align ops
    s1, s2 = all children of n1, n2
    s = matchedOrLabelLCS(s1, s2)
    i = 1
    for b in s2:
      if (b has partner) and (b != s[i][2]):
        a = partner(b)
        if (parent(a) == partner(parent(b))):
          # Nodes a and b are misaligned
          generate move op for (a, b)
        else:
          i += 1
      else if (b has no partner) and (b == s[i][2]):
        # Match children with same label
        matchNodes(s[i][1], s[i][2])
        i += 1

  # Generate update, insert and move ops
  for c2 in children(n2):
    if c2 has partner:
      c1 = partner(c2):
      checkUpdate(c1, c2)
      if parent(c1) != n1:
        # Node was moved and is now child of n2
        generate move op for (c1, c2)
    else:
      # Node in n2 still has no partner
      c1 = None
      generate insert op

    # Descend
    topDownRec(c1, c2)
```

Figure 7: Phase 4 - Building an Edit Script.

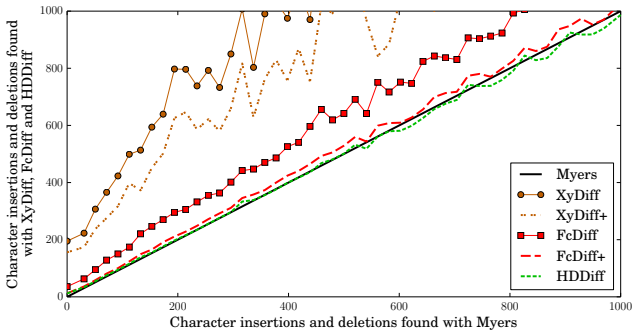

Figure 8: The number of character insertions and deletions generated by each algorithm compared to the number of insertions and deletions required by Myers LCS algorithm. The data is spread over 50 bins. The lines indicate the mean number of operations in each bin per algorithm.

[9] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[10] T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple xml tree differencing by sequence alignment. In *Proceedings of the 2006 ACM symposium on Document engineering*, pages 75–84. ACM, 2006.

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

[12] L. Peters. Change detection in xml trees: a survey. In *3rd Twente Student Conference on IT*, 2005.

[13] S. Rönnau, G. Philipp, and U. M. Borghoff. Efficient change control of xml documents. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 3–12. ACM, 2009.

[14] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards xml version control of office documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 10–19. ACM, 2005.

[15] D. Shapira and J. A. Storer. Edit distance with move operations. In *Combinatorial Pattern Matching*, pages 85–98. Springer, 2002.

[16] D. Shapira and J. A. Storer. In place differential file compression. *The Computer Journal*, 48(6):677–691, 2005.

[17] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

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

[19] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. Kf-diff+: Highly efficient change detection algorithm for xml documents. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 1273–1286. Springer, 2002.

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

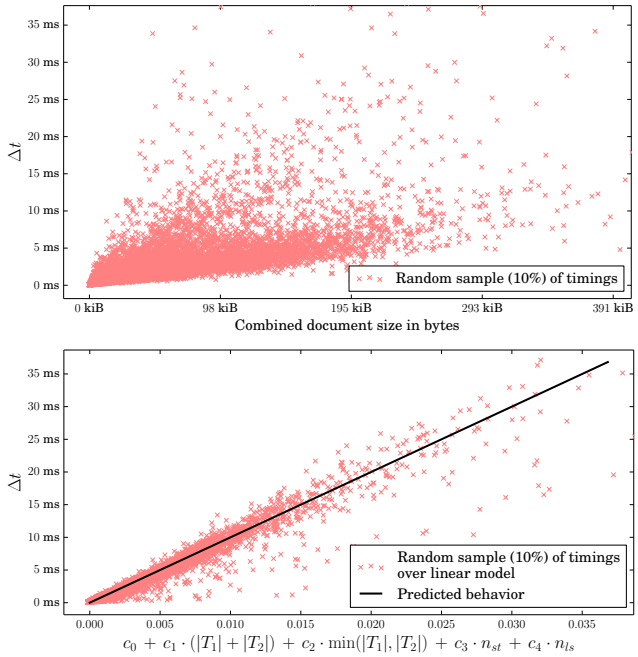

Figure 9: Scatter plot of timings of *HDDiff* over the combined length of two documents (upper chart) and a least squares fit of a linear model of the timings (lower chart).