Efficient Change Control of XML Documents

Sebastian Rönnau Geraint Philipp Uwe M. Borghoff

Institute for Software Technology
Universität der Bundeswehr München
Werner-Heisenberg-Weg 39
85577 Neubiberg, Germany
Sebastian.Roennau@unibw.de

ABSTRACT

XML-based documents play a major role in modern information architectures and their corresponding workflows. In this context, the ability to identify and represent differences between two versions of a document is essential. Several approaches to finding the differences between XML documents have already been proposed. Typically, they are based on tree-to-tree correction, or sequence alignment. Most of these algorithms, however, are too slow and do not support the subsequent merging of changes.

In this paper, we present a differencing algorithm tailored to ordered XML documents, called DocTreeDiff. It relies on our context-oriented XML versioning model which allows for document merging [21, 20]. An empiric evaluation demonstrates the efficiency of our approach as well as the high quality of the generated deltas.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing—*Document management, Version control*

General Terms

Algorithms, Management, Performance, Reliability.

Keywords

XML diff, XML merge, version control, office documents, tree-to-tree correction

1. INTRODUCTION

Documents are an important pillar of today's office work. Teams are built across conventional hierarchies, with everyday inter-team cooperation and collaboration. Since most people are part of different teams, there exist a high amount of collaborative relationships between them. Conventional business processes cannot be applied to this highly dynamic environment, where a major part of the common knowledge is represented by documents. This collaboration is ad-hoc and it has become accepted practice to send documents via e-mail, and to compare different document versions manually, which is both time-consuming and error-prone.

In recent years, efforts have been made to standardize the technical representation of documents in order to ensure interoperability. The most prominent office formats are OpenDocument (ODF) [4] and Office OpenXML [17]. Both have been been approved as ISO standards, and rely on XML to ensure interoperability. Both use a narrative XML model, where XML trees are ordered. In the domain of web documents, XHTML has become a standard for the XML representation of web pages [18]. Throughout this paper, a XML document is considered having an ordered tree, with most of the content stored in the leaf nodes.

Comparing two XML documents can be regarded a special case of the tree-to-tree editing problem defined in [24]. The idea is to compute a so-called *delta* that contains a list of *edit operations* that, when applied to one of the documents, will yield the other one. The computation of the delta is called *diff*. In previous work, we have considered the question of change control of office documents using an XML diff and have performed an evaluation of different XML diff approaches [22].

In the domain of source code control, merging two versions of a document is a common task. This is often done by applying a delta to a version of the document the delta has not been computed for. In the XML domain, however, this appears to be a more difficult task, impossible to solve by most existing approaches. The main reason for this are non-persistent paths in the XML tree that become invalid during merge operations. Therefore, we have introduced a new delta model using so-called context fingerprints. The context fingerprints allow for the identification of edit operationss in a highly reliable way by taking the surrounding nodes into account [21]. Using these fingerprints, merging of XML documents including conflict detection becomes possible. In [20], we have shown the mapping of tracked changes from office documents onto our delta model. The tracked changes are recorded during an editing session and stored within the document itself.

Apart from the extraction of tracked changes, we have shown how to map the deltas generated by the jXyDiff implementation onto our delta model. JXyDiff relies on an algorithm presented in [8] that appeared to be the most time efficient approach available in our last evaluation [22]. Nevertheless, the transformation process is inefficient, as the compared documents have to be parsed twice – first by the

DocEng'09, September 16–18, 2009, Munich, Germany.

diff algorithm, and second by the transformator that constructs the context fingerprints. Furthermore, jXyDiff produces sub-optimal deltas due to a greedy subtree matching algorithm [8]. These drawbacks have led us to the idea of designing a new XML diff algorithm tailored to the needs of XML documents.

In this paper, we present a novel diff algorithm, called *DocTreeDiff*, to efficiently compute the differences between two XML documents, basing on a dynamic programming approach. The key idea is to compute the longest common subsequence of the leaf nodes. Changes on higher levels of the XML tree are detected in a second step. Our algorithm is highly efficient in terms of time, and space. Its time complexity is $O(\text{leaves}(T)D + n)$, with D denoting the number of edit operations on the leaf level, and n the number of non-leaf nodes. DocTreeDiff computes the delta with linear space complexity in $O(T + D)$.

The remainder of this paper is organized as follows: In Section 2, we introduce the tree-to-tree editing problem as well as our delta model in more detail. Section 3 gives an overview of already existing XML diff approaches. Our algorithm will be described in Section 4, including a theoretical complexity analysis. The efficiency of our implementation, and the quality of our deltas are then evaluated using two empirical test scenarios in Section 5.

2. PRELIMINARIES

In order to prevent notational confusion, we first give a brief overview of the tree-to-tree editing problem for ordered trees in Section 2.1. Then, we present our XML model in Section 2.2. In Section 2.3, we define our XML delta model, which differs from the one used for the tree-to-tree editing problem in some important details. Finally, we make assumptions about the documents to compare in Section 2.4 that have led us to the design of our XML diff algorithm.

2.1 Tree-to-Tree Editing

The problem of finding the differences between two XML documents is a use case of the tree-to-tree editing problem defined in [24]. The challenge is to find a sequence of so-called *edit operations* to transform one tree into another one. This sequence is also called an *edit script*. The available edit operations are *insert*, *update*, and *delete*. An *update* operation changes the label of a node within the tree, *insert* adds a new node to the tree, whereas *delete* removes an existing node. Note that by this definition, any edit operation addresses just a single node.

Another interesting property of this definitions made is the fact that nodes can be inserted and deleted arbitrarily within the tree. If a node being the root of a subtree is deleted, its subtree is placed at the position of the deleted node, which means that all nodes in the subtree are pulled one tree level towards the root[1]. Vice versa, an inserted node pushes an already existent node forest down one tree level, becoming its parent node. This behavior is highly inspired by the graph theory, but is not very suitable for the domain of XML in our opinion. The main reason for this is the fact that parent-children relationships play an important role in most XML dialects. We will introduce a different tree edit model tailored to XML in Section 2.2.

[1]By this definition, the root node cannot be deleted if it has more than one child, as this would result in a forest.

Basically, there exists an arbitrary number of possible solutions for the tree-to-tree editing problem. However, many solutions may not be very appropriate. To measure the quality of a result, the tree edit distance has been introduced in [25], depending on the amount of edit operations needed to transform one tree into another one. The goal is to find the minimum edit distance. To be able to compute the minimum edit distance, a corresponding cost model for edit operations has to be defined. Usually, a fixed-cost model is used, assigning each edit operation type a cost value. Recently, probabalistic cost models have been introduced [2].

Some research has been spent on the question how to find a minimum edit distance between two trees efficiently [3]. Most approaches make strict assumptions on the characteristics of the trees to improve efficiency. For example, [7] is tailored to thin and deep trees, which makes it inappropriate for our application (see Section 2.4). The most prominent general solution has been presented by [29]. It has a space comlexity of $O(nm)$, and a time complexity of $O(nm \min(\text{depth}(T_1), \text{leaves}(T_1)) \min(\text{depth}(T_2), \text{leaves}(T_2)))$, with n, m denoting the number of nodes of the trees T_1, T_2 to compare.

As the amount of nodes in a document can easily become very large, the goal to compute a minimum edit script is usually dropped to achieve a better time efficiency. An approach with a time complexity of $O(le + e^2)$, with $l = \text{leaves}(T_1) + \text{leaves}(T_2)$, and e denoting the "weighted" edit distance has been presented in [6]. The term weighted edit distance indicates the edit distance computed by this algorithm. Usually, it should be close or equal to the minimum edit distance. However, the minimality of the edit script cannot be guaranteed, especially in case that many identical node labels exist, which meets the assumptions concerning our use-case (see Section 2.4).

Instead of the tree editing presented in [24], tree alignment can be used to represent the differences between two trees [13, 12]. However, the number of required operations is usually much higher compared to the tree edit distance [13], without a significant gain of efficiency. Therefore, we do not follow this approach.

The tree edit distance presented before is also called top-down distance [26]. In contrast to this, [26] presents the bottom-up distance of two trees. In context of this bottom-up distance, a node is only considered to be unchanged if none of its descendants has been changed. Using this definition, if one leaf node has been changed, all its ancestors up to the root node are considered to be changed, too, thus leading to a high increase of edit operations. Although the bottom-up distance is efficient to compute, we consider this edit model to be inappropriate for handling documents for the reason of meaningfulness of the computed result.

Up to now, we only considered ordered trees. The tree-to-tree correction problem for unordered trees appears to be NP-hard [30, 5]. As our XML model relies on ordered trees (see Section 2.2), we omit further discussion on the comparison of unordered trees.

2.2 XML Model

In context of tree-to-tree editing, only node labels have been mentioned. XML, however, knows different types of nodes, including attributes, and namespaces. To map the XML data model onto the tree-to-tree editing problem, we introduce a corresponding XML model. We regard a node

label as being the hash value of the content of the node. The content of the node depends on the type of the node. For text nodes, the content is the text they contain. Element nodes are hashed on a normalized representation of the node names, and their attributes. The content of a processing instruction node is the normalized node name. Note that in this paper, we use the term *node*, and *node label* equally. Node contents are normalized to ensure the equality of the hash nodes of semantically identical nodes with different syntax. For example, namespaces have to be considered, different encodings or attribute orderings. Details of the normalization steps performed can be found in our previous work [21][2]. One important fact is that we assume two nodes with identical hash value to be identical, thus ignoring possible hash collisions. We believe that this assumption does not restrict the generality and reliability of our model, as we assume node contents to be rather small compared to the hash size, thus lowering the probability of a hash collision to a negligible magnitude.

Our XML model claims that the node ordering is a significant property of XML documents. In this context, the term *document order* denotes the order in which nodes are encountered, one after another, when the document that contains them is parsed. This behaviour is also called the *ordered tree model*, and is used in all further considerations about XML if not explicitly stated contrarily.

The terms *ancestor*, *child*, *descendant*, *parent* and *sibling* are used as defined in [9]. A *subtree* is defined as exactly one node, called *root(subtree)*, that can have any descendant nodes without restriction. A *tree sequence* is a list of adjacent subtrees rooted on the same hierarchy level. In terms of graph theory, the tree sequence can be regarded as ordered forest.

2.3 XML Delta Model

We follow the tree-to-tree editing described before by assuming that the basic operations to be performed onto a tree are *insert*, *update* and *delete*. However, our view towards XML documents has led us to a different definition of these operations, including the introduction of a *move* operation, as well as a different definition of the edit script. We will discuss our delta model only briefly, a thorough, and more formal desription can be found in [21, 20].

An *insert* opertion addresses a node within the tree, where a tree sequence is to be inserted. An already existent node, and all of its subsequent nodes are shifted in direction of the document order. Vice versa, a *delete* operation deletes a complete tree sequence. An *update* operation addresses a single node arbitrarily within the tree. This operation allows for a efficient description of structure-preserving changes within the document, which would otherwise result in large insert, and delete operations. Additionally, we introduce the *move* operation, which basically consists of a pair of insert and delete operations, linked by a common ID attribute.

Figure 2 shows the corresponding document trees of the example in Figure 1. First, a node sequence is inserted, containing the nodes "bold", and "italic" on the same tree level. Second, an attribute change is represented by an update operation. Third, a subtree is replaced by a tree sequence. This will be represented as a delete, and an insert operation.

[2]To be precise, we have switched the hash algorithm from MD5 to FNV [20]. The normalization procedure is not affected by this implementation detail.

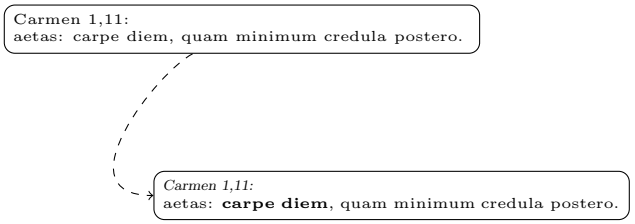

Figure 1: Two versions of a document, showing simple markup changes. The corresponding document tree is shown in Figure 2.

There exist different ways of storing the edit script. The most common approach is to place all edit opertions within a delta. In context of our approach, we relax the sequence property, and demand a delta to be a set. This definition has an important impact on the adressing of the edit operations. Each operation has to be addressed relating to the original version of the document. In contrast to this, most approaches address the operations relating to the original version, assuming the edit operations being before in the edit script to be already performed. This way, any edit operation not being performed would result in wrong addresses of all subsequent edit operations within that edit script. Examples for this behaviour are shown in [21].

In our delta model, edit operations have to be invertible. This mostly affects update operations that have to inclose the old, and the new value of the node to update. An inversion of an edit operation allows the reconstruction of the prior version out ouf the new one. Additionally, this allows for conflict detection during the merge process, as it can be verified, whether the targeted node (or subtree) has changed in the meantime [21].

2.4 Expected Modification Pattern

In prior work we expressed several properties of the XML tree representation of office documents [22]. Among others, we stressed that the trees are not deep, but wide. Basically, the larger a document becomes, the wider the corresponding XML tree will be. The depth is usually constrained by the document format, whereas the width is not restricted[3].

A key question during the design of a diff algorithm is the expected modification pattern performed on the document. Following the specifications of the most common document formats (see [4, 17] for office documents, or [18] for web documents), most of the content of the document is stored within the leaves of the XML tree, e.g. containing the paragraphs, or cell values. Inversely, the non-leaf nodes mostly contain structural information and markup information. This leads to three important assumptions concerning the expected modification pattern:

1. Content is stored within the leaves.
2. Changes to structure and markup will be performed on a higher level within the tree.
3. Many non-leaf nodes are equal due to identical markup.

[3]In fact, the maximum document size can be restricted by the corresponding application. E.g., OpenOffice Calc is not able to handle more than 67,108,864 cells.

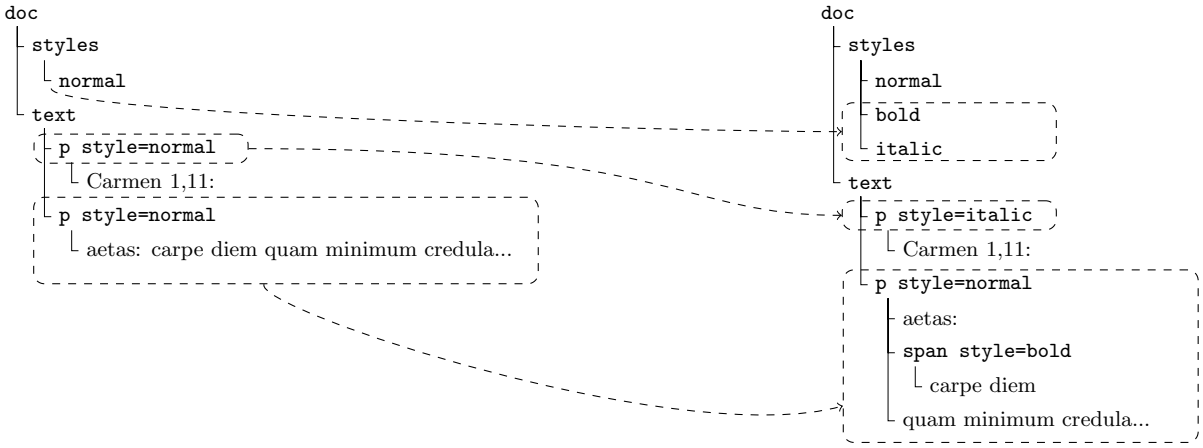

Figure 2: The XML trees of the example documents shown in Figure 1. The markup changes lead to more comprehensive changes than one might expect.

We assume format changes within paragraphs to be a frequent operation performed on documents, resulting in interesting implications on the XML representation of the documents. As stated before, the corresponding paragraph would be stored as a leaf of the tree, most likely as a text node. Emphasizing a word or sentence with a bold font figure would lead to a split of the paragraph into two text nodes containing the parts of the paragraph before and after the format change, and a subtree containing the emphasized text as leaf and an element containing the format information. This behaviour has been illustrated in [20]. Referring to the expected modification pattern, it is likely that one leaf node will be replaced by a node sequence, presumably containing subtrees, as shown in the example tree in Section 4.2. Additionally, this is an important argument for a subtree-oriented insert/delete behavior, as node-based delta models cannot display correlations between insert operations. Especially in the case of a subsequent merge, it is likely that some edit operations cannot be applied. Edit operations performed on related nodes can easily lead to a incosistent merged document if they are treated independently during the merge. On a larger scale, this argumentation has lead us to the introduction of the move operation to be able to display a move as an atomic operation.

Our last assumption affects the quantity of the expected edit operations. As office documents can become very large, we assume that the total number of changes performed to the document is significantly smaller than the document itself. We assume that in typical working scenarios, comprehensive changes on the entire documents are infrequent.

3. RELATED WORK

We recall that the best known general solution for the tree-to-tree editing problem for ordered trees ensuring a minimum edit script was presented in [29]. For two trees T_1, T_2 to compare, the algorithm has a time complexity of $O(nm \min(\text{depth}(T_1), \text{leaves}(T_1)) \min(\text{depth}(T_2), \text{leaves}(T_2)))$, with n, m denoting the number of nodes of T_1, and T_2. Its space complexity is $O(nm)$. It uses insert, delete, and update operations as descibed in Section 2.1. Microsoft has proposed a C# -implementation of this algorithm for Windows systems[4]. The delta model used by this implementation differs from the original definition. It additionally allows for subtree insertion/deletion, as well as for subtree moves. Therefore, this delta model is very close to ours.

We already mentioned the tree editing alogorithm presented in [6], called FastMatch EditScript (FMES). It has a time complexity of $O(le + e^2)$, with $l = \text{leaves}(T_1) + \text{leaves}(T_2)$, and e denoting the weighted edit distance (see Section 2.1), and $O(nm)$ space complexity. This approach relies on the usual tree-to-tree edit operations insert, delete, and update that all address single nodes. Additionally, it makes use of the move operation. Basing on this algorithm, an XML-aware application has been implemented by Adrian Mouat in his `diffxml`-tool[5]. The implementation uses the Java language, and has been updated recently. The delta model is based on [6]. It is stored in a self-defined DOM-style XML representation called DUL (Delta Update Language) that uses XPath expressions for node addressing. However, the edit operations are not invertible. Beside this approach, the `xmldiff` project[6] claims to implement FMES, however the implementation uses an edit model similar to ours, thus adressing subtrees by insert, and delete operations. Therefore, we are not convinced that this tool implements the FMES algorithm. Beside of this, this Python-based implementation already appeared to be inefficient for larger documents in a previous evaluation [22]. In recent tests, we confirmed that the processing speed is still the crucial handicap of this implementation. Therefore, we omit this implementation in our evaluation performed in Section 5.

In previous work, the XyDiff implementation based on [8] appeared to be best available XML diff [22]. The algorithm has a time complexity of $O(n \log n)$, and a linear space complexity [8]. It generates deltas that resemble our delta model. Insert, and delete operations address subtrees, whereas updates address single nodes. XyDiff additionally supports the use of move operations. All operations are invertible. The intuition of XyDiff is to identify nodes using hash values, and to perform a greedy search for common

[4] `http://msdn.microsoft.com/en-us/xml/bb190622.aspx`
[5] `http://diffxml.sourceforge.net`
[6] `http://www.logilab.org/859`

subtrees. The core is the so-called bottom-up lazy-down (BULD) algorithm, which strength is the "laziness" towards changes on lower levels of the tree. This allows the algorithm to efficiently compute an edit script, thus generating non-optimal deltas. The implementation used in our first evaluation [22] was based on C++. It does not seem to be still maintained and needs to be compiled on outdated compiler and XML parser versions. Therefore, we use a newer Java implementation of XyDiff, called jXyDiff[7].

Since our last evaluation, a novel approach to XML differencing has been proposed. Faxma transforms an XML document to a sequence of tokens [15]. On this token sequence, a greedy sequence alignment is performed using Rabin fingerprints [19]. It has a worst-case time complexity of $O(n^2)$, yielding almost linear time complexity for the average case. The space complexity seems to be linear, too. Due to the linearization of the XML document in sequences, faxma neglects the XML hierarchy, thus using a similar edit model as [24], allowing to arbitrarily insert and delete nodes within the hierarchy. Interestingly, faxma does not create a delta in terms of an edit script. The output produced by faxma is a script called "XML with references" (XMLR) that uses pointers with a XPath-resembling syntax to refer to the corresponding (unchanged) parts of the original document. Between these pointers, inserted parts can be placed. This delta model makes it impossible to invert a delta, as no deletion can be identified using the delta. A Java-based implementation of faxma is publically available[8].

Note that none of the presented approaches is able to use the generated deltas to merge document afterwards. Each patch of a document with a delta computed using another version would most probably result in erroneously adressed edit operations, or even program aborts due to invalid adresses [21]. In case of XyDiff, such a patch run would not be allowed, as every node is marked using a persistent identifier, called XID. If any identifier does not match, the patch run is aborted completely. The Microsoft implementation acts likewise – it uses a hash value computed over the entire document to ensure that no delta is applied to a version of the document it was not computed for.

All the mentioned approaches rely on a kind of delta model, as described before. There exist, in fact, other approaches that do not rely on a delta model. Their intention is to annotate an existing document with version information [10], or temporal information [23]. In our opinion, this annotation approach does not comply with our intended collaboration model, as the document is containing all version information within itself. First, this implies that the whole document has to be transmitted every time, which is inapropriate in low-bandwidth environments. Second, it might be unwanted to transmit all editing information including the metadata for privacy or security reasons. Therefore, we do not consider these approaches in this paper.

4. A DOCUMENT-CENTRIC XML DIFF

We introduce a novel algorithm to XML tree differencing, called DocTreeDiff, relying on our XML delta model presented in Section 2.3. Its main goal is to efficiently compute a correct and compact, yet non-optimal matching between two XML trees. It is based on the longest common subsequence problem, described in Section 4.1. The key concept of our algorithm is the application of the longest common subsequence problem to the leaves of the XML tree, as described in Section 4.2. Section 4.3 discusses several properties of our approach.

4.1 The Longest Common Subsequence

In the domain of string editing, the longest common subsequence (LCS) of two strings is defined as the maximum length of a seqence of characters being included in both strings [27]. Basically, the LCS computation can be solved in quadratic time. Several algorithms try to optimize the computation of the LCS, [1] gives a concise overview on the most common approaches. Usually, a dynamic programming approach is used to reduce time complexity, thus yielding a sub-quadratic run-time.

Strings are known to have a limited alphabet, which makes the comparison of characters quite simple. In terms of XML, the input alphabet is limited, too, but can be arbitrarily large. Our XML model relies on a node-centric granularity (see Section 2.2). As complete paragraphs can be stored within one node (see Section 2.4), the comparison of nodes can be quite costly. Therefore, we compare two nodes using their hash values, as described before. Using these hash values, the size of the input alphabet can be reduced significantly to the number of possible hash values. As a matter of course, the input alphabet is still larger compared to strings by order of magnitude. As the complexity of some LCS algorithms depends on the size of the input alphabet [1], they are not suitable for our use case.

Beside the time complexity, the space complexity is an important issue when using LCS algorithms. Most approaches compare two strings with sizes n and m using a matrix, thus leading to an $O(nm)$ space complexity. Nevertheless, some algorithms are able to compute the LCS with linear space, which is an important achievement when applying the LCS problem to XML documents. As documents can have an arbitrary length, the size of the corresponding matrix could easily reach a critical point. Therefore, a solution of the LCS problem that could be computed with linear space complexity is preferable.

In our approach, we rely on an implementation of [16] that runs in $O(ND)$, where $N = n + m$ denotes the summarized size of the compared documents, and D denotes the size of the minimum edit script. It appears to be well suitable for documents with $N \gg D$, which meets our expected modification pattern in Section 2.4. Its space complexity is linear with $O(N)$. This algorithm is also used in the well-known GNU diff tool, as it finds a good balance between time and space complexity.

Basically, the longest common subsequence generated is a minimum edit script. Under certain circumstances, this assertion does not hold. In cases that there are (semantically) different nodes with the same label, they are considered to be equal, thus possibly confusing the algorithm, leading to non-optimal and confusing results [6].

4.2 Diff Algorithm

The key idea of our algorithm is straightforward: we compute the LCS of all leaf nodes, and use a dynamic programming approach to compare the corresponding parent nodes of the leaves. The algorithm works in three steps:

[7] `http://potiron.loria.fr/projects/jxydiff`
[8] `http://code.google.com/p/fc-xmldiff/`

4.2.1 Longest Common Subsequence of Leaf Nodes

First, we collect the hash values of all leaf nodes of the trees to compare. These hash values are enriched with the depth of the corresponding leaf, i.e. the length of the path to the root element. We compute the longest common subsequence of all leaf nodes using the algorithm presented in [16]. As a result, we obtain a longest common subsequence of all similar leaf nodes on the same XML tree level.

Accordings to [16], the LCS is computed in $O(ND)$, with N denoting the number of elements to compare, and D the number of edit operations. In our application, N must be considered as the leaf nodes to compare, thus leading to $O(\text{leaves}(T)D)$, with T representing both trees.

4.2.2 Identification of Structure-preserving Updates

In a second step, we check the parents of the already matched nodes within the LCS. Therefore, for a pair of matched leaf nodes, we traverse both trees up to the root in parallel, comparing the hash values of the corresponding nodes. A mismatch of the hash values is interpreted as an update operation, because they are structure-preserving. A corresponding update operation will be created, and added to the delta. Note that all changes performed on parents of matched leaves will always be represented using update operations. This is no limitation, as no structural change can be performed on the parents without affecting the leaves. A structural change, e.g. by insertion or deletion of a parent subtree, would result in a different depth of the leaf node, thus avoiding the leaves to match when computing the LCS.

During the bottom-up traversal of the tree, each node is marked when it is investigated for the first time. If an already visited node is reached, the bottom-up traversal is aborted, as all further nodes on the path to the root have already been visited. This dynamic programming approach ensures that each node is visited only once, therefore resulting in a linear complexity of the bottom-up tree traversal.

4.2.3 Constructing Insert and Delete Operations

The task of the third step is to construct the corresponding insert, and delete operations for the non-matched leaf nodes. Any leaf node of the first tree that is not part of the second tree is basically considered as deletion, the reverse holds for inserts. For any leaf node to delete, the nearest matching leaf node being child of the same parent is searched. If the parent node does not have a child in the LCS, its parent is checked for a corresponding child. This search for a child being part of the LCS is propagated until a child is found, or the root node is reached. This bottom-up pass allows for identification of large subtrees to delete. The same procedure is performed for subtrees to insert.

A special case holds if two leaf nodes at the same position are detected as an insert and a delete, respectively, and their parent nodes are identical. In this case, these two operations are replaced by a single update operation.

Adjacent insert or delete edit operations of the same operation type, having the root of the subtree to insert or delete on the same depth are glued together into one edit operation containing a tree sequence. This way, adjacent edit operations can be united to one large edit operation that will be performed atomically. The edit operations are added to the delta after the glueing process. During this procedure, complementary insert and delete operations are identified as move operation, and linked using an ID attribute.

This step of our algorithm relies on a dynamic programming approach again. As each visited node is marked during the bottom-up passes, it will be visited only once, thus performing in linear complexity.

4.2.4 Complexity Analysis

The heart of our algorithm is the LCS run performed on the leaves. As already stated, it runs in $O(\text{leaves}(T)D)$ time. The bottom-up traversal for the detection of update operations runs in linear time, namely $O(\text{ancestors}(LCS))$, depending on the number of ancestor nodes of the matched longest common subsequence. Constructing the insert and delete operations creates two bottom-up traversals in linear time depending on the number of the ancestors of the nodes not part of the LCS. These three runs in linear time can be represented as one run in $O(n)$, with n denoting the number of non-leaf nodes. In conclusion, we claim that our algorithm has an overall time complexity of $O(\text{leaves}(T)D + n)$.

The space complexity of our algorithm is linear, we claim the upper bound to be $O(T + D)$. We justify this by following considerations: the chosen LCS algorithms runs in linear space [16], yielding $O(\text{leaves}(T))$. During the bottom-up pass, additional space is required to compare all nodes, thus leading to $O(T)$. As each edit operation is stored in a separate list, we have to extend the upper bound to $O(T + D)$.

4.3 Discussion

In recent years, performance of personal computer systems and laptops was mainly increased by introducing multi-core CPUs. The competition about the number of cores seems to supersede previous races for higher clock speeds. With multi-core CPUs becoming widely deployed, parallel programming becomes more and more attractive. In the domain of tree differencing, first approaches have been made [28]. Beside the additional complexity caused by the parallelization, documents do not offer a "natural" segmentation that could be used to separate the different parts being analyzed in parallel. Our evaluation shows that a linear diff is efficient enough for common tasks (see Section 5). In the context of large document repositories, a parallel approach might be more appropriate.

For some scenarios, a three-way diff is helpful. It allows for a direct comparison of two independent version of one base document (that should be a common ancestor in terms of the document evolution). Our approach could basically be extended to a three-way comparison model. However, a three-way comparison has a significant higher complexity. For example, a three-way LCS algorithm with complexity of $O(n^2 + nm^2)$ has been presented by [11]. In addition to this increase of complexity, the comparison of the parent nodes has to be extended for three-way comparison. This results in the loss of the linear complexity of that part of our algorithm. Therefore, we do not consider three-way differencing of documents in this paper. An XML-aware three-way diff has been presented in [14].

5. EVALUATION

As already stated before, the performance of the XML diff implementation is a crucial parameter when evaluating the practicability of an approach. Theoretical considerations like the complexity class of an algorithm are a good indication of the expected performance. However, as most algorithms use heuristics to improve their performance for the

Implementation	Time Complexity	Delta Model	Target of Insert and Delete	Move Supported	Minimum Edit Script	Invertible
diffxml [6]	$O((\text{leaves}(T_1) + \text{leaves}(T_2))e + e^2)$	sequence of edit operations	nodes	no	no	no
DocTreeDiff (presented here)	$O(\text{leaves}(T)D + n)$	set of edit operations	subtrees	yes	no	yes
faxma [15]	$O(n)$ (average case) $O(n^2)$ (worst case)	insert operations and references to original version	both	yes	no	no
jXyDiff [8]	$O(n \log n)$	sequence of edit operations	subtrees	yes	no	yes
Microsoft XML Diff [29]	$O(nm \min(\text{depth}(T_1), \text{leaves}(T_1))$ $\min(\text{depth}(T_2), \text{leaves}(T_2)))$	sequence of edit operations	both	yes	yes	no

Table 1: These five XML diff implementations have to solve two different test scenarios. They differ in terms of their complexity class and the delta model used.

expected average case, one cannot rely only on the complexity class as distinguishing factor. Therefore, we present two test scenarios to evaluate the applicability of our algorithm. We compare our implementation with the implementations presented in Section 3 in terms of speed, and delta quality. Table 1 gives an overview of the compared approaches.

First, we describe two test scenarios in Section 5.1. Afterwards, we present our test results concerning the runtime and the memory footprint in Section 5.2, and evaluate the resulting deltas in Section 5.3.

5.1 Test Scenarios

As our algorithm is intended to be suitable for XML documents in general, we have created two test scenarios with different properties, based on different document types. The first scenario is based on a large ODF spreadsheet document, on which we have performed changes manually. In the second scenario, we have tracked the evolution of a news site on the web, without any knowledge about the changes that have been performed on it. We describe the scenarios in more detail at the end of this Section.

During our tests, we have focussed on the question how the amount of changes performed on the document affects the runtime and the memory usage of the different implementations. In our evaluation, we require the amount of nodes to remain almost constant. The motivation for this decision is to be able to track the performance of the compared implementations w.r.t. to the amount of edit operations. As the complexity of all algorithms depends on the amount of nodes, we try to omit the influence of an alternating node quantitiy.

All test runs have been performed on a machine equipped with an Intel Core 2 Quad Q9650 3.00 GHz processor, 4 GB RAM, running Windows Vista Business SP1 (64 bit) and Sun Java JDK 1.6.0_10 (32 bit).

For the first test scenario, we have created a large ODF spreadsheet document of 357 KByte, consisting of around 5700 nodes. We have created 19 versions of this document, containing 5 to 100 changes. In this context, we define a change by means of the office application (inserting, deleting, and updating a cell). As each cell is stored within the document even if it is empty, the influence of changes on the total number of nodes is not significant. We have compared all versions with the original one. In this test scenario, we have a precise knowledge about the changes performed on the document, and can create an order of the difficulty of the tasks to perform.

In the second test scenario, we have tracked the evolution of the Wired news site[9]. We have taken 17 snapshots within one day from 7am to 10pm. The snapshots have been converted to XHTML, resulting in XML documents of almost constantly 108 KByte and 2400 nodes. The main intention for this scenario is to track the evolution of a web document. This includes rearrangements of the headlines, and alternating advertisements. Therefore, we cannot create an order of the difficulty of the test runs. The test runs are ordered by the distance of the snapshots to the original version on the timeline.

5.2 Runtime and Memory Usage

Figure 3 shows the runtime of the different implementations to compute the deltas. The test results for diffxml for the ODF scenario do not appear on the Figure, as this implementation performs slower by an order of magnitude. Diffxml took constantly about 35s to solve the given task. Interestingly, diffxml shows a reasonable performance for the XHTML scenario. Microsoft XML Diff acts inversely. While showing a good performance in the ODF scenario, it performs some of the test runs very slow in the XHTML scenario. We do not have an explanation for this behavior and can only guess that the search for an minimum edit script is very complex in these cases. Faxma and DocTreeDiff solved the test runs in nearly constant time, which conforms with the complexity class claimed. For jXyDiff, the comparison of the identical document versions is very fast, indicating an efficient heuristic for this case.

The memory footprint of all implementations is reasonable, as shown in Figure 4. Only Microsoft XML Diff showed a significant increase of the allocated memory depending on the amount of edit operations to find. On the other hand, this implementation has a modest memory usage in nearly similar tests. We assume this implementation to have an efficient heuristic for similar documents. The same holds

[9] `http://www.wired.com`

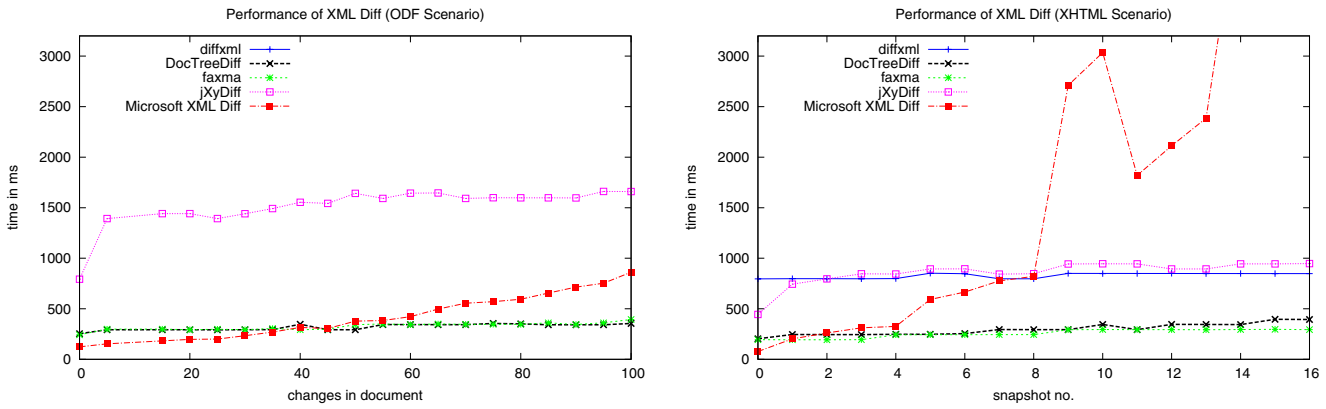

Figure 3: DocTreeDiff and faxma are the fastest implementations by far.

for diffxml, which needs almost half the memory for identical documents, but needs almost constant memory when comparing differing documents.

5.3 Quality of the Delta

All approaches make use of different delta models, which makes it difficult to measure the quality of the resulting deltas. First, we compare the size of the deltas, shown in Figure 5. In the XHTML scenario, the results are comparable. In the ODF scenario, however, the deltas generated by jXyDiff and diffxml are very large and exceed the size of the whole document by far. As diffxml ist performing an LCS run on every hierarchy level of the XML tree, we assume the high amount of equivalent nodes describing the markup to confuse the algorithm, which is a known hazard [6].

To adress the issue of the different delta models, we investigate the computed deltas in more detail. Apparently, we do not consider the deltas generated by faxma, as they are not exactly a delta in terms of an edit script, and are therefore not comparable to the other deltas. Figure 6 shows the total amount of edit operations per test run for each implementation. Interestingly, DocTreeDiff constantly generates the lowest amount of edit operations in its deltas, whereas the Microsoft XML Diff needs more edit operations. At first sight, this contradicts the fact that the Microsoft implementation creates minimum edit scripts. However, a detailed examination of the results shows that our deltas are non-optimal in terms of the edit cost model [25, 6]. A tree sequence to insert or delete is counted as one operation in our evaluation, but has to be considered as the sum of all of its nodes w.r.t. the edit cost. Anyhow, we assume the goal of a small quantity of edit operations to be important, especially to prevent ambiguous edit operations during a following merge process [21].

A second parameter to compare the approaches is the mixture of the different operation types. Table 2 shows the average percentage of each operation type in the deltas. The results are astonishing, the dispersal of the operations types differs both between the algorithms, as well as between the scenarios. Diffxml creates a large amount of move operations in both scenarios, although this operations has not been used in the ODF scenario. Here, the algorithm is confused by the high amount of similar non-leaf nodes. JXyDiff has similar problems in the ODF scenario. DocTreeDiff

(a) ODF Scenario

	insert	delete	update	move
diffxml	2.68%	3.46%	6.49%	87.37%
DocTreeDiff	34.51%	34.43%	31.06%	0.00%
jXyDiff	5.85%	5.92%	18.77%	69.46%
Microsoft XML Diff	7.88%	20.68%	66.85%	4.59%

(b) XHTML Scenario

	insert	delete	update	move
diffxml	19.52%	20.04%	5.54%	54.90%
DocTreeDiff	20.45%	31.25%	41.11%	7.19%
jXyDiff	26.41%	27.72%	20.88%	24.99%
Microsoft XML Diff	38.33%	38.38%	2.36%	20.93%

Table 2: The mixture of the operation types in the deltas differs both between the algorithms, as well as between the scenarios.

makes extensive use of the update operation. Interestingly, Microsoft XML Diff acts likewise in the ODF scenario, but not in the XHTML scenario. Here, a more thorough investigation would be interesting, which has to be omitted due to the limited space.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel XML diff algorithm tailored to XML documents, called DocTreeDiff. It is highly efficient in terms of time and space. An empirical evaluation has substantiated our theoretical time and space bounds. Our dynamic programming algorithm is able to find all changes between documents by concentrating on the leaves of the document that presumably contain the content of the document. Changes on non-leaf nodes are identified during a second step.

The generated deltas are small compared to most of the competing approaches. This intuitively makes the delta more readable and more coherent. In order to allow for merging the delta with other versions of the document, the edit operations stored in our deltas are enriched with context fingerprints [21].

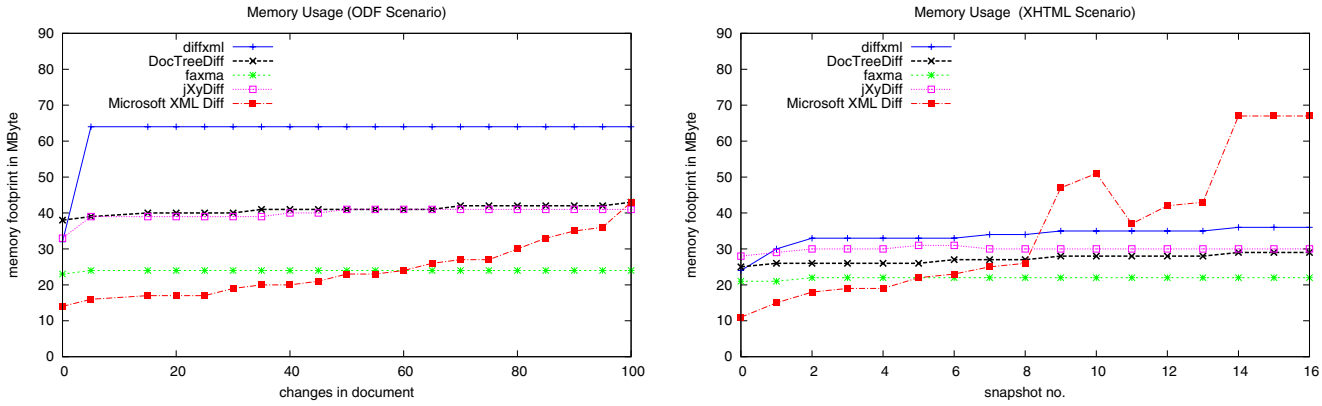

Figure 4: The memory footprint is nearly constant for most implementations. However, Microsoft XML Diff shows a significant increase in memory usage.

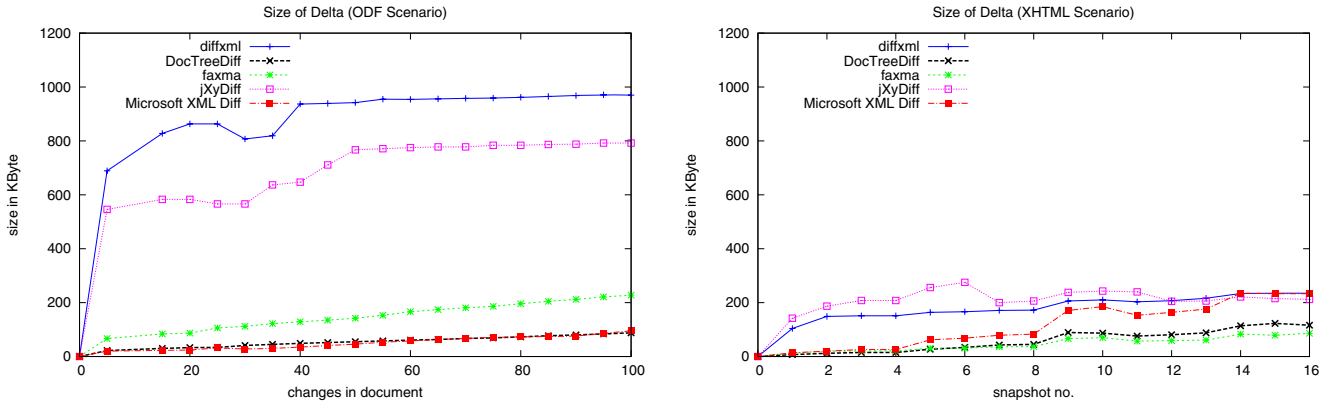

Figure 5: DocTreeDiff, faxma, and Microsoft XML Diff create significant smaller deltas, especially in the ODF scenario, where jXyDiff and diffxml create sub-optimal deltas.

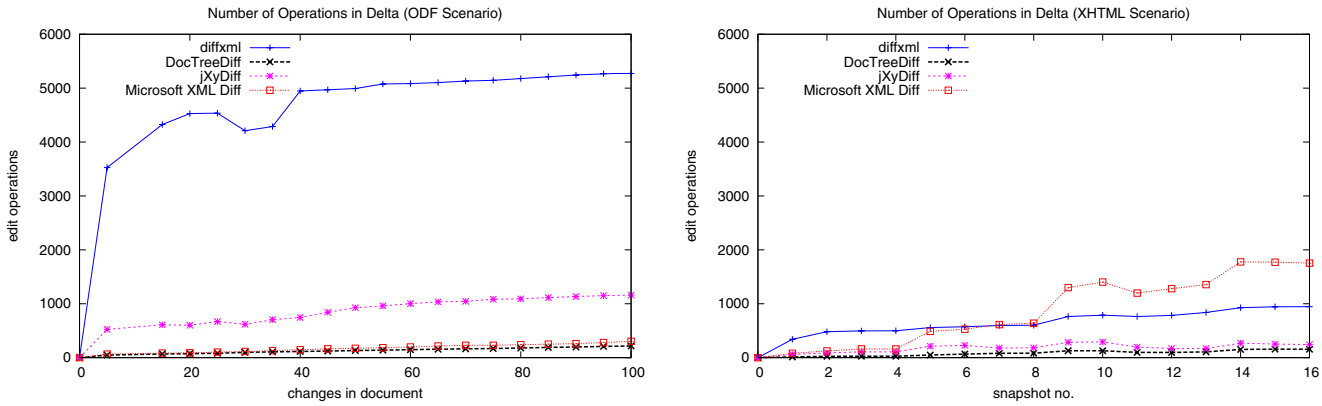

Figure 6: Diffxml needs basically more edit operations due to its node-oriented delta model. The small amount of edit operations shows the benefits of addressing tree sequences by DocTreeDiff.

In future work, we will concentrate on the question of how to present the generated deltas appropriately to the user. This includes graphical user interfaces allowing the user to interactively merge document versions. We also want to investigate the question of how the changes on the XML layer can be displayed in a rendered representation on the application layer, which is a less precise yet more user-friendly way of change control.

7. REFERENCES

[1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *SPIRE '00: Proceedings of the 7th International Symposium on String Processing Information Retrieval*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[2] M. Bernard, L. Boyer, A. Habrard, and M. Sebban. Learning probabilistic models of tree edit distance. *Pattern Recogn.*, 41(8):2611–2629, 2008.

[3] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[4] M. Brauer, R. Weir, and M. McRae. *OpenDocument v1.1 specification*, 2007.

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

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

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

[8] G. Cobéna, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*, pages 41–52. IEEE Computer Society, 2002.

[9] S. DeRose and J. Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[10] R. L. Fontaine. Merging xml files: a new approach providing intelligent merge of xml data sets. In *Proceedings of XML Europe 2002*, 2002.

[11] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng. Efficient algorithms for finding interleaving relationship between sequences. *Information Processing Letters*, 105(5):188 – 193, 2008.

[12] J. Jansson and A. Lingas. A fast algorithm for optimal alignment between similar ordered trees. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 232–240, London, UK, 2001. Springer-Verlag.

[13] T. Jiang, L. Wang, and K. Zhang. Alignment of trees – an alternative to tree edit. *Theoretical Computer Science*, 143(1):137 – 148, 1995.

[14] T. Lindholm. A three-way merge for xml documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10, New York, NY, USA, 2004. ACM.

[15] T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple xml tree differencing by sequence alignment. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 75–84, New York, NY, USA, 2006. ACM.

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

[17] J. Paoli, I. Valet-Harper, A. Farquhar, and I. Sebestyen. *ECMA-376 Office Open XML File Formats*, 2006.

[18] S. Pemberton. XHTML^{TM} 1.0 the extensible hypertext markup language (second edition). W3C recommendation, W3C, Aug. 2002. http://www.w3.org/TR/2002/REC-xhtml1-20020801.

[19] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981.

[20] S. Rönnau and U. M. Borghoff. Versioning xml-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, 2009.

[21] S. Rönnau, C. Pauli, and U. M. Borghoff. Merging changes in xml documents using reliable context fingerprints. In *DocEng '08: Proceeding of the eighth ACM symposium on Document engineering*, pages 52–61, New York, NY, USA, 2008. ACM.

[22] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards xml version control of office documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 10–19, New York, NY, USA, 2005. ACM.

[23] L. A. Rosado, A. P. Márquez, and J. M. Gil. Managing branch versioning in versioned/temporal xml documents. In *XSym 2007: Proceedings of 5th International XML Database Symposium*, pages 107–121, 2007.

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

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

[26] G. Valiente. An efficient bottom-up distance between trees. In *SPIRE*, pages 212–219, 2001.

[27] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[28] K. Zhang. Efficient parallel algorithms for tree editing problems. In *CPM '96: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 361–372, London, UK, 1996. Springer-Verlag.

[29] 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.

[30] K. Zhang, J. T.-L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *CPM '95: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.