

Fast and Simple XML Tree Differencing by Sequence Alignment

Tancred Lindholm
Helsinki Institute for
Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
tancred.lindholm@hiit.fi

Jaakko Kangasharju
Helsinki Institute for
Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
jkangash@hiit.fi

Sasu Tarkoma
Helsinki Institute for
Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
sasutarkoma@hiit.fi

ABSTRACT

With the advent of XML we have seen a renewed interest in methods for computing the difference between trees. Methods that include heuristic elements play an important role in practical applications due to the inherent complexity of the problem. We present a method for differencing XML as ordered trees based on mapping the problem to the domain of sequence alignment, applying simple and efficient heuristics in this domain, and transforming back to the tree domain. Our approach provides a method to quickly compute changes that are meaningful transformations on the XML tree level, and includes subtree move as a primitive operation. We evaluate the feasibility of our approach and benchmark it against a selection of existing differencing tools. The results show our approach to be feasible and to have the potential to perform on par with tools of a more complex design in terms of both output size and execution time.

Categories and Subject Descriptors

I.7.1 [Document and Text Processing]: Version Control; F.2.2 [Nonnumerical Algorithms and Problems]: Computations on discrete structures

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

XML, differencing, sequence alignment, move, ordered tree

1. INTRODUCTION

Extensible Markup Language (XML) [27] has come of age. It has become the *lingua franca* for information interchange, and will perhaps even surpass unstructured text some day. A rich infrastructure of tools and services has been built to support the use of XML: editors, style sheets and formatting

languages for display, transformation tools, databases, and query languages.

An important tool in the tool chest of the XML document engineer is a *differencing*, also known as a *diff*, tool. A diff tool takes two input documents, and outputs the difference (*diff*) between the documents in some format. Diffs are usually minimum cost with respect to some change model, i.e., they express a minimal set of changes. Document differencing (*diffing*) is useful in a vast array of tasks, including managing different versions, maintaining document histories, aiding when merging changes between documents, visualizing changed parts, etc.

Like any other text-based format, XML may be differenced by *linear* tools intended for character data, such as diff [13] and xdelta [12]. In contrast to these, there are diffs that are aware of XML structure. These *XML diffs*, which we focus on in this paper, express changes as operations on the document tree that the XML document encodes.

The choice of linear or XML diff depends on the application area. XML diffs are needed when the intended usage of the diff, either by human or machine, is such that the changes need to be expressed in XML terms. Examples include cases when the diff is utilized by other XML tools, when presenting semantic changes to end users, or when expressing changes to XML-specific constructs. An XML diff is also more robust to changes that are irrelevant to the modeled data, such as modifications to the order of attributes, whereas such changes may show up in a linear diff. On the other hand, if we use diffs to reconstruct past versions of a document, including formatting (e.g., in a document repository application), a linear diff may be a better choice, due to advantages in terms of computational speed and output size (as will be seen further on in this paper).

XML diffs may be classified into those that use an *ordered* tree model, where child element order matters, or an *unordered* tree model where it does not. Disregarding application semantics, XML itself is ordered. An ordered model is also used in many important applications, such as office documents [18], SVG [25] drawings, and XHTML [24] documents. An unordered model may be more appropriate for, e.g., databases [22].

A large concern with XML diffs is performance, as general minimum-cost tree diffing algorithms exhibit big-O complexities from quadratic upwards [5]. To address this concern, approaches based on heuristics have been devised [3, 5]. Heuristic approaches have traditionally been seen as “second-class” approaches, but this point can be argued [5]:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DocEng'06, October 10–13, 2006, Amsterdam, The Netherlands.
Copyright 2006 ACM 1-59593-515-0/06/0010 ...\$5.00.

An optimal diff is only optimal with respect to some cost model. Several models may be constructed, and it is not obvious which model corresponds to the “correct” diff.

Typically, XML diff tools detect node insert, delete and update operations. A useful addition to this set of *basic operations* is the subtree *move* operation, which allows the diff to more accurately model XML restructuring, such as item reordering, rich text restructuring, or directory tree manipulations. Including the move as a primitive in tree differencing has been further motivated in [3–5, 15], and [10].

In this paper we present an ordered XML diff algorithm based on heuristics that supports subtree moves in addition to the basic operations. We approach the problem by introducing a mapping between a tree and a token sequence model of XML, and then doing the performance-critical diff computation in the sequence domain. To compute the diff, we apply a greedy heuristic that is both simple and fast.

Comparing to other work, the closest ones appear to be the `xydiff` [5] and `diffmk`¹ tools. With respect to `xydiff`, our approach has the advantage of being simpler, especially regarding the heuristics used. While `diffmk` operates in the sequence domain, it does not support the move operation, nor does it produce a space-saving diff, as it encodes changes by annotating the input document.

Looking at our approach from a software engineering point of view, we demonstrate how to separate the concerns of computing the changes from the output diff format by using an intermediate metadiff model. This helps supporting several XML diff formats, which is important as standardization on these is still in its infancy. The implementation of our approach will be released as Open Source in 2007.

The remainder of this paper begins by presenting background on XML differencing and sequence representations in Section 2. We then look at aligning XML documents expressed as sequences in Section 3, and proceed to our specific approach in Section 4. Our approach is then evaluated for feasibility, and its performance compared to other tools, in Section 5. This performance comparison of diff tools complements existing results. In Section 6, we then conclude and look at further work that remains to be done.

2. XML DIFFERENCING

XML differencing is commonly stated as a *tree-to-tree correction problem*, where a given set of tree operations and associated costs is used to find the minimum-cost sequence of operations transforming one tree to another. The cost is usually known as the *edit distance*. The first non-naive approach is due to [19]. It was improved upon in [11] and [16].

The quadratic upper bound of the general approach has been improved by making various assumptions regarding the operations used and the nature of the data [2, 4, 31]. Most relevant to our work is [4], which supports the move operation, and has been implemented in the tools `xmldiff`² and `diffxml`³. By making assumptions about the uniqueness of nodes and allowable parent-child relationships the algorithm achieves $O(ne + e^2)$ complexity, where e is the edit distance between the inputs and n the input size. The algorithm consists of two phases: building a matching between the input documents, and finding the minimum-cost edit script corre-

sponding to the matching. To the best of our knowledge, however, the move operation is not supported by any fully general minimum-cost algorithm for ordered trees.

An alternate way to further improvements in speed is to drop the requirement of an optimal edit script altogether, rather than to change its definition. The `xydiff` [5] tool computes the diff using a heuristic algorithm that supports the basic operations and the move operation. The algorithm has a computational complexity of $O(n \log n)$, which makes it very attractive for use on larger documents. The designers of the XML versioning API for Office Documents [15] found `xydiff` to be the best tool on which to base their design.

Rönnau et al. [15] examine version control of XML office documents. The study enumerates a set of requirements for XML differencing tools in this domain, and proceeds by nominating a set of suitable candidate implementations. The candidates are evaluated on a set of OpenOffice Write documents. The evaluation shows that execution time is a major issue with current tools. As the examined domain is similar to ours, we use a superset of the tools and test data used in the study in our performance evaluation.

We briefly mention that the unordered XML differencing problem, which is even more computationally taxing, has been studied in [3] and [22]. These results are, however, outside the scope of this paper.

2.1 Basic Definitions and Example

Several tree models for XML documents exist [26, 28, 29], all similar in essence. Relating to the XML Infoset specification [28], we use the term *element (node)* to refer to Element Information Items (II) with associated attributes (Attribute II) and *text (node)* for Content II. These are organized into an ordered tree according to the structure of the XML document. For brevity we omit other Infoset parts, such as Processing Instruction and Comment IIs, in this paper.

In the context of differencing we define the *base* and *modified* XML documents. The diff encodes a set of changes that transforms the base document into the modified document. Throughout the paper the base document is denoted d_0 and the modified document d_1 .

To illustrate our differencing approach we will use the case shown in Figure 1 as a running example. The base document d_0 is the document on the left and the modified document d_1 the one on the right. We use the letters a, b, ... to refer to the elements $\langle a \rangle$, $\langle b \rangle$, etc.

Compared to the base document, d_1 contains an insert (the i element), a deletion (the e element is removed), an update (the root node is changed from r to R), and a move (the subtrees rooted at a and b have been swapped). The example is deliberately unrealistically busy in order to illustrate the full set of editing operations.

2.2 XML as Sequences

We use the term *sequence* to denote an ordered list of *tokens* and use the notation $\langle t_0, \dots, t_{n-1} \rangle$ where the t_i are tokens. Assuming $s = \langle t_0, \dots, t_{n-1} \rangle$, the partial sequence $\langle t_i, \dots, t_{j-1} \rangle$, $i < j \leq n$ is denoted $s^{i \dots j}$ and is called a *subsequence* of s . To denote the length of a sequence we use $|\cdot|$, i.e., the length of s is $|s|$.

XML data may be split into tokens at various semantic levels. One obvious tokenization is to use the input characters as tokens. In the PYX⁴ format, XML is converted into a

¹<http://www.nwalsh.com/java/diffmk>

²<http://www.logilab.org/projects/xmldiff/>

³<http://diffxml.sourceforge.net/>

⁴<http://pyxie.sourceforge.net>

d_0 : <r> <a> <c/><d/><e/> <f/> </r>		d_1 : <R> <f/> <a> <c/><d/> <i/> </R>
---	--	---

Figure 1: Differencing example. Original document (d_0) to the left, changed (d_1) to the right.

Table 1: Central XAS event types

Node	XAS Event
Element node e , start	SE(e)
Element node e , end	EE(e)
Text node t	C(t)
Attribute n with value v	AT(n, v)

line-based format, where each line corresponds to an item of XML data, such as start element, attribute, text, etc. In the XAS [7] Java API⁵, the tokenization roughly corresponds to the events generated by an XmlPull⁶ parser. The tokens are called *events* in XAS terminology. The central XAS events are show in Table 1.

3. MATCHING XML SEQUENCES

To compute which parts of the document tree have changed between d_0 and d_1 we may construct a matching between the nodes in d_0 and d_1 or construct an edit script transforming d_0 into d_1 . In our case we use the matching approach, in which pairs ($u \in d_0, v \in d_1$) of nodes are matched. We constrain the matching so that each node in d_0 or d_1 is matched to no more than one node, and so that any pair of matched nodes have equal content.

Any node in d_0 without a match is said to be *deleted* and any node in d_1 without a match is *inserted*. A node in d_1 that is matched is considered *moved* (regardless of any change in position compared to d_0). Our constraints on the matching leave out matches encoding the similarly defined *update*, *copy*, and *glue* [3] operations. Updates are not needed, since we required matched nodes to always have equal content. Copies, glues, and other operations of higher complexity were left out since we considered the potential gain in output quality to be very limited compared to the accompanying increase in complexity.

In this work, we match sequences of tokens encoding the input documents, rather than the document tree structures. Transforming to the sequence domain allows us to apply string algorithm research, such as longest common subsequence algorithms [1], algorithms supporting block moves [17, 20], and heuristic algorithms for finding common subsequences [12, 21]. Using sequences rather than trees should also save memory, as we need no parent, child, etc. pointers.

We choose to use tokens based on XML parser data rather than characters, and use the XAS tokenization model as it is familiar to us. XAS tokenization will automatically enforce some useful matching constraints, such as the alignment of

element boundaries and matching only the same types of nodes. A character-based approach, in contrast, could for instance align the string tag in both documents, regardless of it occurring in the context SE(tag) in one document and in C(tag) in the other. On the other hand, a character-based approach might do better in aligning large text nodes.

We expect sequence-based matching to perform best when there is a straightforward correspondence between the operations on a tree T and the operations on the corresponding sequence s_T . We use sequences constructed by *depth-first traversal*, since we find that in this case tree operations map well to the corresponding operations on subsequences of s_T . In particular, we note that the move operation usually corresponds to moving a *continuous* subsequence of s_T , which should help sequence matching by providing longer identical runs than in the case where a tree-level move maps to several shorter sequence-level moves.

Moving on, we introduce a few matching notations and definitions to build our approach on. The XAS sequences corresponding to d_0 and d_1 are denoted s_0 and s_1 . To aid the reading of sequences, we optionally include the position of an event in a sequence as a superscript to the event. Turning to our example, for instance, we have $s_0 = \langle {}^0\text{SE}(r), {}^1\text{SE}(a), \dots, {}^{13}\text{EE}(r) \rangle$ for d_0 .

The alignment, i.e., matching, of tokens in the sequences s_0 and s_1 is expressed using the matching operations cpy() and ins(). A matching of a subsequence $s_0^{i\dots j}$ to $s_1^{k\dots k+(j-i)}$ is denoted cpy(k, i, j). An unmatched subsequence $u = s_1^{k\dots l}$ is denoted ins(k, u). The domain and range of an operation are the sets of tokens the operation addresses in s_0 and s_1 , i.e., the domain and range of cpy(k, i, j) are $s_0^{i\dots j}$ and $s_1^{k\dots k+(j-i)}$, and the domain and range for ins(k, u) are \emptyset and $u = s_1^{k\dots k+|u|}$.

For our purposes, we need a matching that unambiguously constructs s_1 from non-overlapping parts of s_0 . Such a matching is said to be *well-formed*. More precisely, a *matching* is a set of matching operations, and a matching is *well-formed* if the combined range of the operations in the list is exactly s_1 and the domains and ranges do not overlap, i.e., each token in s_1 is encoded by exactly one cpy() or ins() operation, and each token from s_0 is addressed at most once.

A well-formed matching may be unambiguously ordered by the start positions of the ranges into a *match list*. We may leave out the initial argument of an operation in a match list, as it is implicitly given by the length of the combined range of the preceding operations.

The *optimal matching* with respect to some cost function f is the matching \bar{m} for which $f(m) \geq f(\bar{m})$, for all $m \in M$, where M is the set of well-formed matchings. If f measures the cost of an edit script corresponding to the matching, the minimum-cost edit script is obtained.

In contrast to the optimal matching we also define the *correct matching*, which matches tokens according to the actual edits made between d_0 and d_1 . We may then consider the *quality* of a matching to be a measure of how close the matching is to the correct matching, i.e., how precisely the matching encodes the actual changes made between d_0 and d_1 . It is important to recognize that there is a difference between the correct and (any) optimal matching: the former always models the actual edits (by definition), whereas the latter only models the actual edits if they coincide with the optimum of f .

⁵<http://hoslab.cs.helsinki.fi/savane/projects/xebu>

⁶<http://www.xmlpull.org>

4. THE DIFFERENCING ALGORITHM

Our differencing algorithm consists of two main steps. First, we match XAS token subsequences of the input documents to each other, yielding a *match list*. In the second step, this match list is encoded as an XML document which encodes the differences between the input documents, i.e., the XML diff.

4.1 Greedy Matching for Flexibility and Speed

The matching phase is the heart of the differencing algorithm, as the computational efficiency and quality of result are determined by this phase. When choosing a matching algorithm, we find that in general there is a tradeoff between performance, quality of the match, and compactness. We focus on practical performance, as this seems to be the bottleneck of current XML differencing algorithms. Still, we want reasonable diff quality and size as well.

Given these requirements we decided to base our approach on a heuristically guided greedy matcher. Greedy matching of sequences, as demonstrated by the `xdelta` [12] and `rsync` [21] tools, offer very good performance, while being simple enough to allow easy tuning with different heuristics. That is, a greedy matcher gives us the ability to choose an appropriate tradeoff between performance, quality, and size.

In particular, we did not choose a matching algorithm based on worst-case computational complexity in this work. Instead, we decided to focus on verifying the feasibility of the sequential approach, and on constructing an implementation that uses sequence alignment algorithms that work well in practice. A way to further evolve our work is to enforce a bound on computational complexity by limiting worst-case search performance (in a similar manner to what was done in [5]), or to use some existing sequence alignment algorithm with well-known computational requirements.

Figure 2 shows pseudocode for the matcher. The core of our algorithm is `rollmatch(u, v)`, which finds the smallest start and largest end offsets i and j for which $u^{0\dots(j-i)}$ is equal to a subsequence of v , i.e., $u^{0\dots(j-i)} = v^{i\dots j}$. To find the match we slide a window of length s over v , computing a hash over the window for each step. A match candidate is found when the window hash values are equal. If the candidate can be verified, a match has been found. To improve on the naive $O(|u| \cdot |v|)$ complexity, we use a Rabin fingerprint [14] (“rolling hash”), which allows us to update the hash in $O(1)$ at each step, yielding performance close to $O(|u| + |v|)$ (assuming no hash collisions). This technique is also used in the `xdelta` and `rsync` tools.

During matching, we keep track of matched subsequences in match lists, so that matching sequences are expressed as `cpy()` operations and unmatched ones as `ins()` operations. The task of the `match` and `findmatch` procedures is to match sequences from the input match lists to each other, and to keep track of matched and unmatched sequences.

Procedure `match(u, v, s)` matches unmatched sequences from \mathbf{u} to sequences of at least s tokens in any of the sequences in \mathbf{v} . This is done by repeatedly removing the next `ins()` operation from \mathbf{u} , and replacing it with `cpy()` and `ins()` operations encoding matched and unmatched parts of the operation, respectively. The match list \mathbf{u} automatically combines pairs of operations into one when applicable (e.g., $\langle \dots, \text{cpy}(0, 1), \text{cpy}(1, 2), \dots \rangle \rightarrow \langle \dots, \text{cpy}(0, 2), \dots \rangle$) to avoid artificial boundaries and to keep the output simple.

The lower level `findmatch(u, v, s)` procedure scans the

```

procedure match( $\mathbf{u}, \mathbf{v}$ : Matchlist,  $s$ : Integer )
1: Cursor  $c_u := \text{beforefirst}(\mathbf{u})$  { Cursor into u }
2: while Operation  $t := \text{removenext}(c_u, \mathbf{u})$  do
3:   if  $t$  is cpy() then
4:     addbefore( $c_u, \mathbf{u}, t$ )
5:   else
6:     Operation  $c := \text{findmatch}(u, \mathbf{v}, s)$  where  $t = \text{ins}(u)$ 
7:     if  $c \neq \text{nil}$  then
8:       addbefore( $c_u, \mathbf{u}, c$ )
9:       addafter( $c_u, \mathbf{u}, \text{ins}(u^{|\mathbf{c}|\dots|\mathbf{u}|})$ )
10:    else
11:      addbefore( $c_u, \mathbf{u}, \text{ins}(u^{0\dots s})$ )
12:      addafter( $c_u, \mathbf{u}, \text{ins}(u^{s\dots|\mathbf{u}|})$ )
13:    end if
14:  end if
15: end while
endproc

procedure findmatch( $u$ : Sequence,  $\mathbf{v}$ : Matchlist,
 $s$ : Integer): Operation
1: for all  $\text{ins}(v) \in \mathbf{v}$  using a zigzag pattern do
2:    $(i, j) := \text{rollmatch}(u, v, s)$ 
3:   if  $(i, j) \neq \text{nil}$  then
4:     in  $\mathbf{v}$ , replace  $\text{ins}(v)$  with  $\langle \text{ins}(v^{0\dots i}), \text{ins}(v^{j\dots|\mathbf{v}|}) \rangle$ 
5:     return cpy( $v^{i\dots j}$ )
6:   else
7:     return nil
8:   end if
9: end for
endproc

procedure rollmatch( $u, v$  : Sequence,  $s$ : Integer): Range
1: Integer  $h_u := \text{winhash}(u, 0, s)$ 
2: Integer  $h_v := \text{winhash}(v, 0, s)$ 
3: for  $0 \leq i < (|v| - s)$  do
4:   if  $h_v = h_u$  then
5:     if  $u^{0\dots s} = v^{i\dots(i+s)}$  then
6:       Integer  $j := i + s$ 
7:       while  $v^j = u^{j-(i+s)}$  do
8:          $j = j + 1$  { Greedy extension of match }
9:       end while
10:      return  $(i, j)$ 
11:     end if
12:   end if
13:    $h_v := \text{roll}(v, h_v, i, i + s + 1)$ 
14: end for
15: return nil
endproc

```

Figure 2: Heuristic matching of document sequences

subsequences in \mathbf{v} for a subsequence matching u . The \mathbf{v} match list is traversed using a zigzag pattern, which means that we visit $\mathbf{v}(p+1)$, $\mathbf{v}(p-1)$, $\mathbf{v}(p+2)$, \dots where $\mathbf{v}(p)$ denotes the p^{th} operation in \mathbf{v} , and p is the index of the previous operation from \mathbf{v} where a match was found. Whenever a match is found we extend it as much as possible. The auxiliary procedure `winhash(u, i, j)` computes the initial window hash over $u^{i\dots j}$ and `roll(u, h, i, j)` returns the hash for $u^{i\dots j}$ based on the hash h over $u^{i-1\dots j-1}$.

Note that any matching subsequence is removed from \mathbf{v} to ensure that any subsequence of \mathbf{u} has maximally 1 match

in \mathbf{v} . This also means that \mathbf{v} holds any unmatched, i.e., deleted, sequences when `match()` returns.

To match the input documents s_0 and s_1 we initially set $\mathbf{u} = \langle \text{ins}(s_1) \rangle$ and $\mathbf{v} = \langle \text{ins}(s_0) \rangle$. We then repeatedly invoke `match(u, v, s)` with decreasing values for s . By first matching longer sequences, and then shorter ones among those left unmatched, we increase the probability that matches in the base document are unique, and hence likely to be correct.

We found that the values $S = \langle 48, 32, 16, 8, 4, 2, 1 \rangle$ for s in combination with greedily extending matches and the zigzag search pattern worked well as a heuristic for finding the correct match. The greedy part takes maximum advantage of a match once it is found. Looking for the next match close to the previous one is based on the assumption that operations more often have local rather than global scope, which seemed to work well in our case.

The maximum value of S (48) seemed sufficient to ensure unique matches, although a larger value could be introduced for very self-similar documents, and conversely, a smaller for those less self-similar. Generally speaking, adding values larger than the longest match decreases performance by introducing iterations with no progress. We did, however, not investigate this, or other effects of varying S , in this work.

Looking briefly at the computational complexity of the matcher, we claim that worst case performance is $O(n^2)$ (assuming no hash collisions). This happens for inputs where there are no matching subsequences between s_0 and s_1 , in which case `rollmatch` is called $O(n)$ times, with each call taking $O(n)$ (since $v = s_0$ in each call). On the other hand, if $s_0 = s_1$, matching is completed by a single call to `rollmatch`, in which case the overall complexity of the matcher becomes $O(n)$. Thus, we expect the performance of the matcher to degrade from $O(n)$ to $O(n^2)$, as the amount of change in d_1 increases from none to completely changed. Note that the $O(n^2)$ worst-case bound may be lowered by limiting the scan for a match to a constant-size neighborhood, as was done in `xydiff`.

Turning to our example, we take the XAS event sequences s_0 and s_1 obtained by parsing d_0 and d_1 :

$$s_0 = \langle {}^0\text{SE}(r), {}^1\text{SE}(a), {}^2\text{SE}(c), {}^3\text{EE}(c), {}^4\text{SE}(d), \\ {}^5\text{EE}(d), {}^6\text{SE}(e), {}^7\text{EE}(e), {}^8\text{EE}(a), {}^9\text{SE}(b), \\ {}^{10}\text{SE}(f), {}^{11}\text{EE}(f), {}^{12}\text{EE}(b), {}^{13}\text{EE}(r) \rangle$$

and

$$s_1 = \langle {}^0\text{SE}(R), {}^1\text{SE}(b), {}^2\text{SE}(f), {}^3\text{EE}(f), {}^4\text{EE}(b), \\ {}^5\text{SE}(a), {}^6\text{SE}(c), {}^7\text{EE}(c), {}^8\text{SE}(d), {}^9\text{EE}(d), \\ {}^{10}\text{EE}(a), {}^{11}\text{SE}(i), {}^{12}\text{EE}(i), {}^{13}\text{EE}(R) \rangle$$

The longest common subsequence $s_0^{1\cdots 6}$ has a length of 5, so no progress can be made until we call `match` with $s = 4$. During that lap we try to match the sequences $s_1^{0\cdots 4}$, $s_1^{4\cdots 8}$, and $s_1^{8\cdots 12}$, for which there are no sufficiently long matches in s_0 . However, for size 2 we find matches (both being extended past 2 tokens by greedy matching): $s_1^{2\cdots 5}$ is matched to $s_0^{10\cdots 13}$, and $s_1^{5\cdots 10}$ to $s_0^{1\cdots 6}$ yielding

$$\mathbf{u} = \langle \text{ins}(\langle \text{SE}(R), \text{SE}(b) \rangle), \text{cpy}(10, 13), \text{cpy}(1, 6), \\ \text{ins}(\langle \text{EE}(a), \text{SE}(i), \text{EE}(i), \text{EE}(R) \rangle) \rangle$$

For size 1 we find the remaining matches, yielding

$$\mathbf{u} = \langle \text{ins}(\langle \text{SE}(R) \rangle), \text{cpy}(9, 13), \text{cpy}(1, 6), \text{cpy}(8, 9), \\ \text{ins}(\langle \text{SE}(i), \text{EE}(i), \text{EE}(R) \rangle) \rangle$$

The match list \mathbf{v} encodes the deleted parts of the base document after the final iteration. In this case:

$$\mathbf{v} = \langle \text{ins}(\langle {}^0\text{SE}(r) \rangle), \text{ins}(\langle {}^6\text{SE}(e), {}^7\text{EE}(e) \rangle), \\ \text{ins}(\langle {}^{13}\text{EE}(r) \rangle) \rangle$$

The matching will be well-formed, since we always replace insert operations in \mathbf{u} with copy operations with the same range, our initial condition is $\mathbf{u} = \langle \text{ins}(s_1) \rangle$, and tokens from s_1 are assigned maximally one match in s_0 .

4.2 From Matched Sequence to XMLR Diff

Having computed the match list, which effectively expresses a diff in the sequence domain, we need to go back to the tree domain in order to construct a proper XML diff. In other words, we need to map the concepts of sequence insert and copy to the XML domain. We also need to choose a format for the XML diff.

Choosing an XML diff format is encumbered by the lack of standardization (formal and *de facto*). Furthermore, candidate formats such as XUpdate [8], and the recently initiated Remote Events for XML [30], do not include a move operation. In light of this, we decided to split diff encoding into two phases: match-to-tree and tree-to-diff.

The purpose of the match-to-tree phase is to solve the problem of mapping the matched sequences onto an XML tree model. The actual serialized diff is then generated from the tree model in the tree-to-diff phase. The tree model needs to support the notion of (partially) matched trees, i.e., it should be able to capture the result of the sequential diff onto a tree level.

Due to its intermediary position between matching sequences and XML diff, we refer to the tree model as a *metadiff*. As the metadiff we chose to use the XML-with-references (XMLR) model developed by us in previous work [10], although other choices are possible (e.g., the W3C DOM [26]). XMLR fits our purpose, as it is lightweight and includes a notion of matching between trees. Note that we need not specify any serialization or externalizable node addressing scheme for the metadiff.

In XMLR, we conceptually add two additional tags to an existing XML language: the node reference and the tree reference tags. The *node reference* tag (with accompanying end tag) acts as a placeholder for a node taken from another document. The *tree reference* tag correspondingly acts as a placeholder for a subtree from another document.

The overall idea in translating from the match list to XMLR is to let each `ins()` operation map to the XML encoded by that operation, and let each `cpy()` operation map to node or tree placeholder XMLR tags, which reference the XML copied from s_0 by the operation. This straightforward idea is, however, somewhat complicated by sequence operations not necessarily being aligned on subtree boundaries.

To define the mapping between match list operations and reference tags we introduce the notion of an encoding unit (EU). An *encoding unit* is a list of XAS events, which contains either a content event, an end tag event, or a start tag event with accompanying attribute events. An EU is

```

procedure encodediff(  $s_1, o$  : XASSequence;  $q$  : Queue;  $p$  :
  Boolean ): (NumericalXPath,State)
1: { The encoded XAS events are sent to  $o$ 
    $q$  is the queue of tentative reference tags
    $p$  is set if changes were found by the caller }
2: { Section 1: Encode opening tags }
3: EncodingUnit  $e := \text{removefirst}(s_1)$ 
4: if  $e = \text{EE}()$  then
5:   return DONE;
6: end if
7: {  $c$  indicates if  $e$  is changed compared to  $s_0$  }
8: Boolean  $c := \neg \text{matched}(e) \vee \text{hasdeletedattributes}(e)$ 
9: NumericalXPath  $p := \text{numericalxpathof}(e)$ 
10: if  $c \vee p$  then
11:   emitall( $o, q$ ) { Found changes, emit all; also clears  $q$  }
12:   append( $o, e$ )
13: else
14:   append( $q, \text{makenoderef}(p)$ ) { Put tentative reference }
15: end if
16: { Section 2: Encode child nodes }
17: Mark  $m := |q|$  { Remember current size of  $q$  }
18: repeat
19:   {  $p_c$  is path of encoded child node,  $s$  the status }
20:   ( $p_c, s$ ) := encodeDiff( $s_1, o, q, c$ )
21:    $c := c \vee s = \text{CHANGED}$ 
22:   if  $\neg c$  then
23:     append( $q, \text{maketreeref}(p_c)$ )
24:   end if
25: until  $s = \text{DONE}$ 
26: { Section 3: Shrink queue, emit any endtags }
27: if  $\neg c \wedge \text{childorderchanged}()$  then
28:   emitall( $o, q$ ) { Emit reordered childlist treerefs }
29: else if  $\neg c \wedge \neg \text{childorderchanged}()$  then
30:   cleartomark( $q, m$ ) { Discard all subtree references }
31: end if
32: if  $\neg c \wedge p$  then
33:   append( $o, \text{maketreeref}(p)$ ) { Emit treeref for caller }
34: else if  $c \wedge e = \langle \dots, \text{EE}() \rangle$  then
35:   append( $o, \text{EE}()$ ) { Close tag }
36: end if
37: return ( $p, c ? \text{CHANGED} : \text{NOT\_CHANGED}$ )
endproc

```

Figure 3: Diff encoding algorithm

matched if and only if each event in the unit originates from the same cpy() operation in the match list.

Figure 3 shows pseudocode for a recursive algorithm that generates an XMLR metadiff using the sequence matching from the previous stage. The main concern is that we do not necessarily know which events to output for a subtree until we reach its closing tag. We solve this by maintaining a queue q of tentative tree and node reference tags, which is either sent to the output or partially discarded, depending on the match status of the processed EUs.

The algorithm has three sections, as indicated in the pseudocode. In the first section, an EU is consumed from s_1 , and either shipped to the output along with q if the EU is unmatched, or appended to q as a reference, if the EU is matched. In the second section we recursively encode the child elements of the current EU, and append tree reference tags to q as long as matching subtrees are encountered. In

the final section, we use the encoding status of the subtree (any combination of changes to the opening tag, changes to the descendants, and changes to the order of the child elements) to discard or emit references from q . For readability, we have omitted text node handling, and the folding of start and end node references into a tree reference.

The algorithm has some noteworthy details. First, we do not look at the match status of an EE() event, since the output is already determined by the corresponding opening tag. For SE() EUs, we need to check that the matching attribute list in the base document is identical, so as not to encode too many attributes with the reference tag. Finally, the check for reordered children can simply be made by checking that all children belong to the same cpy() operation, as we merged adjacent cpy() operations in the matching stage.

Some diff formats, such as XUpdate, express the diff as an edit script of change operations. To obtain an edit script from the matched tree model of the metadiff we may use an algorithm similar to the cover-to-script algorithm presented in [3]. In the case of XUpdate, we would also have to substitute any move operations with corresponding insert and deletes, as XUpdate currently does not support moves.

Figure 4 shows the in-memory XMLR metadiff tree for the example scenario. Node references to d_0 are shown using a dark background, and ordinary XML nodes on a light. By substituting the reference nodes with their targets, we see that the metadiff is indeed equivalent to d_1 . The inserted i and updated R nodes appear as ordinary XML. The swapped order of the subtrees a and b is evident from the two first children of the root. The deletion of e is not as obvious. However, it forces a node reference (a) and a list of tree references, rather than a single tree reference.

4.3 The Diff Format

We base our diff format on a simple serialization of the XMLR metadiff. The format consists of a serialization for the reference nodes, and an addressing scheme for persistently identifying base document nodes. We do not optimize the format for compactness *per se* (e.g., by using short tag names). Instead we rely on storing the output inside a compressed zip [6] archive, as this has become an increasingly popular method for space-efficient storage of XML. In support of this, we try to make the format regular so that it compresses well. We also eliminate some obvious repetitiveness for increased readability and additional compressibility.

To address nodes we use a subset of XPath [23], where paths are of the form $[\cdot] \{ /*[n_i] \}^*$, $0 \leq i < k$ where $n_0 \dots n_{k-1}$ are non-negative integers and * denotes repetition (as an example, consider $/*[1]/*[42]$). Such XPaths identify their target node by either starting from the root or the current node, and then selecting the n_i^{th} child for each successive tree level $0 \dots k - 1$ from the originating node. We use XPath expressions because of interoperability, and our particular subset because of convenience. We abbreviate the paths as $/n_0/n_1/\dots/n_{k-1}$ for easier reading.

Frequently, the XMLR tree placeholders reference a consecutive sequence of subtrees. Looking at such references as XPath expressions, we find that they exhibit the pattern $p/n, p/n + 1, \dots, p/n + (i - 1)$, where p is some recurring prefix. We encode such patterns as $\langle \text{copy src}='p/n'$ run=' i' />, including the trivial case of a single tree reference ($i = 1$). As an additional optimization, we express p/n as a relative path from its parent node in cases when the path of

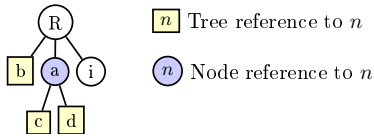


Figure 4: Metadiff between d_0 and d_1

```

1 <R>
2 <copy src="/0/1" run="1" />
3 <node id="/0/0">
4 <copy src="./0" run="2" />
5 </node>
6 <i />
7 </R>

```

Figure 5: Diff between d_0 and d_1

the parent node is a prefix of p/n . To encode a single node reference p , we use `<node id="p">`.

The diff encoding of our example is shown in Figure 5. The structural similarity to Figure 4 is evident. At line 4, note the compaction of the consecutive tree references `/0/0/0` and `/0/0/1` using the `run` attribute, and the relative reference in the `id` attribute.

We think that the format is suitable for human inspection. Inserted and updated nodes are clearly shown as non-reference tags. The overall structure of the document is also preserved.

Deleted nodes are, however, not as evident, as these are not present in the diff. To remedy this, we could add a special section to the diff that would hold the deleted fragments along with their original locations. Computing the set of deleted subtrees can be done by keeping track of paths in d_0 that are not referenced by the metadiff (and hence not included in d_1). Alternatively, `v` could be used, but that would require it to be translated to the tree domain as well.

5. EVALUATION

To evaluate the feasibility of our approach, and to obtain indications of its performance, we executed an implementation of our algorithm on a set of XML differencing tasks. These tasks were also executed on a selection of other differencing tools in order to provide preliminary comparative results on performance, diff compactness, and scalability.

Our implementation was written in the Java programming language on top of the XAS XML processing API and the `kxml`⁷ XML parser. The tests were run on a workstation running the Ubuntu⁸ “Breezy Badger” Linux distribution and Sun Microsystems’ Java software development kit (JDK) build 1.5.0_06-b05. The workstation had a 3.0 GHz Intel Pentium 4 processor with 2.0 GB of RAM.

5.1 Test Sets

Our test data consists of two sets of XML documents. The *directory* set consists of pairs of synthetically generated XML-encoded file system directory trees. The set is parameterized, with output tree size, number of edit operations, and relative frequency of the edit operations being the

⁷<http://kxml.sourceforge.net/>

⁸<http://www.ubuntu.com>

main parameters. The supported edit operations are insert, delete, update, and move of files and directories. The move and delete operations act both on subtrees and leaf nodes.

The *usecases* set consists of pairs of different revisions of documents in common XML formats, with document sizes varying from 1 to 150 kB. The set is divided into 6 *projects*, each of which has an associated set of differencing tasks.

Howto consists of variants of the Linux HOWTO document in OpenOffice Write⁹ format. There are 12 differencing tasks in the project.

Rönnau consists of the OpenOffice Write documents from [15]. There are three base documents, whose XML content is 6, 13, and 148 kB respectively, and revisions of these. The document revisions contain a small, medium, and large amount of changes compared to the base document, for a total of 9 differencing tasks.¹⁰

Homepage The homepage project consists of XHTML and custom XML files. The XHTML pages are personal home pages with some contact information and links. The custom XML files contain contact information, such as telephone number and address. The project has 4 differencing tasks.

Chess Club consists of two similar XHTML pages for a chess club, where one page is optimized for desktops, and the other for PDAs. The project has 2 tasks.

SVG The project consists of a Scalable Vector Graphics [25] drawing and variants, and XHTML documents with the drawing inlined, for a total of 6 differencing tasks.

Shoppinglist The project consists of 12 differencing tasks on an XML-encoded shopping list intended for machine-to-machine data exchange.

This data is available on <http://www.icsi.berkeley.edu/~jan/projects/00VersionControl/> for the Rönnau project and on <http://tdm.berlios.de> for the others. All projects except Rönnau were constructed by us in previous work [9].

5.2 Test Measurements and Results

To run the tests we used `diffxml`, `xmldiff` running the [4] algorithm (the [2] algorithm is also supported), and `xydiff`. We also included the `xdelta` tool, which is a generic binary file diff and patch tool that produces very compact diffs, as well as `diffmk`¹¹, which uses the standard Unix diff algorithm on a list representation of the input XML. Our approach is labeled `faxma` and `faxma+`, with the former denoting execution in a fresh Java Virtual Machine (VM) with no warm-up, and the latter execution in a pre-warmed VM with no startup overhead.

We measured differencing performance in terms of execution time and output size. In the output size tests, we compressed the output diffs with `gzip` before measuring their size. This was done to more accurately reflect the actual use of XML diffs in an environment where compactness is

⁹Openoffice files are zip archives. XML differencing was applied to the `content.xml` file in the archive.

¹⁰Note: the public data set, which we used, contains identical files for small and large amount of changes in the large file case. This appears to be a mistake.

¹¹<http://nwalsh.com/java/diffmk/>. Version 1.0

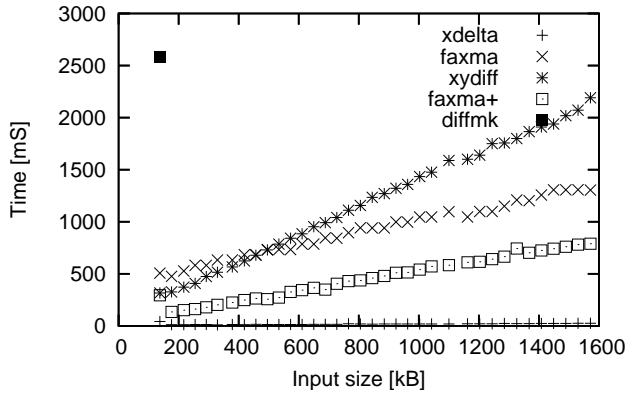
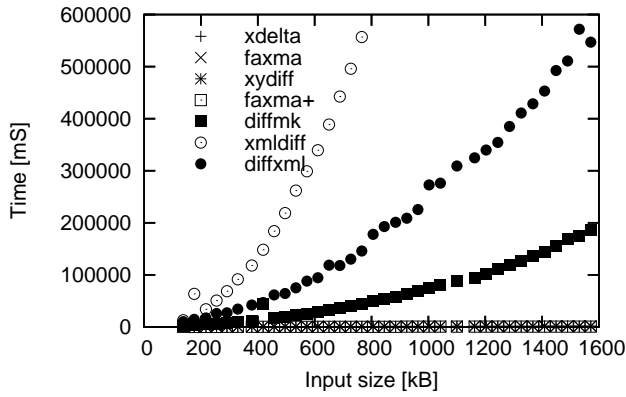


Figure 6: Execution time for increasing input size and 1 edit operation.

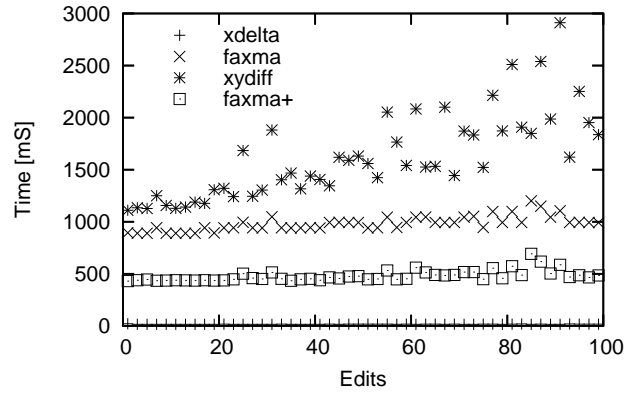
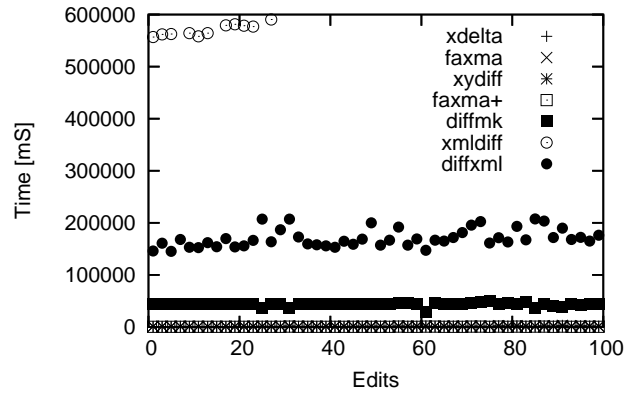


Figure 7: Execution time for constant input size and an increasing number of edits.

an issue. Also, by compressing the output we expect to get results that are less dependent on the encoding and more closely related to the amount of actual information. The difference in output size due to some tools generating XML and others binary diffs should be mitigated by compression.

The execution time results for the directory test set are shown in Figures 6 and 7. In both cases, edits were randomly generated from a uniform distribution of the basic and move operations. Note that there are two graphs in each figure, as the performance numbers vary by more than 2 orders of magnitude between the tools. A time limit of 600 s was enforced for each run, and if a tool did not complete in that time, the result was considered missing.

We find that the initial execution time of our approach is somewhat lagging in the fresh VM case. On the other hand, our approach appears to scale better than the other XML diff tools on this particular workload, both when looking at increasing input size and an increasing number of edits. The matching heuristics used in xydiff, which utilizes uniqueness in element names, appears to work less than optimally in this case, as our test data uses only two distinct element names. The xdelta linear diff tool shows a clear performance advantage.

Looking at output size in Figures 8 and 9, we see that the minimum edit distance approach of xmldiff has a slight advantage. This advantage does, however, come at a very significant degradation in performance, as was seen in Figures 6 and 7. We note that the xydiff and diffxml tools exhibit a very large variance in Figure 8 (this was also present

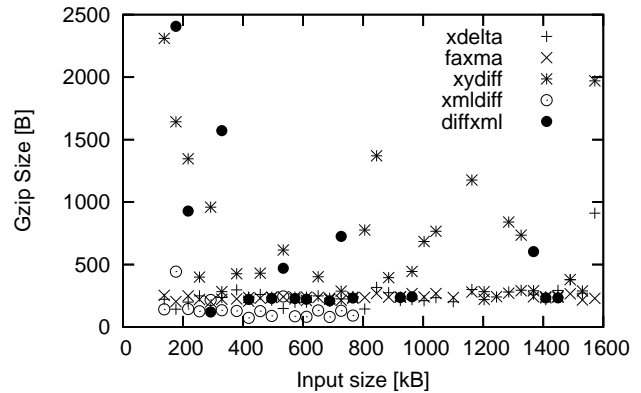


Figure 8: Compressed diff size for increasing input size and 1 edit operation.

in the uncompressed output), suggesting that they in some cases were unable to successfully capture the edit. There are few results for diffxml and xmldiff as these tools frequently exceeded the time limit.

Our approach compares favorably to xydiff, and is almost on par with the binary xdelta algorithm, which comes out as the overall top performer. It appears that the high variance in xydiff measurements is due to the bad fit between our test set and the matching heuristics used. The diffmk tool expresses diffs by inlining them into the d_1 document, so no size comparison is available for that tool.

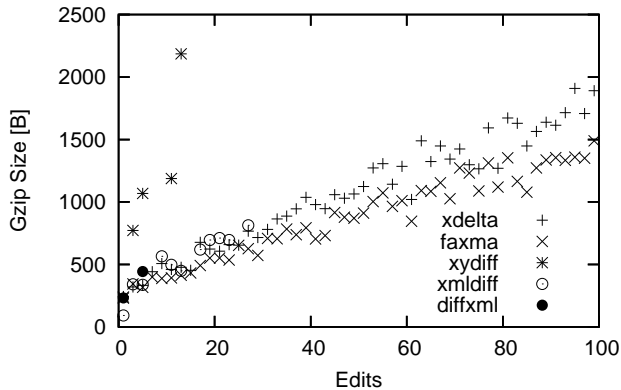


Figure 9: Compressed diff size for constant input size and an increasing number of edits.

Table 2: Execution time in milliseconds by project

	Rönnau	Home	Chess	Howto	SVG	Shop
diffxml	430254	-	2861	75170	-	8986
faxma	12060	1220	738	5683	2630	3802
faxma+	10110	20	22	578	286	40
xydiff	4197	141	93	2910	4840	395
xdelta	144	31	14	112	92	90
xmldiff	-	807	639	63867	84625	2260
diffmk	-	921	654	9848	4023	2575

Table 3: Compressed output size in bytes by project

	Rönnau	Home	Chess	Howto	SVG	Shop
diffxml	91276	-	1504	84710	-	5489
faxma	47895	1601	1001	22579	46951	4019
xydiff	51793	1899	1744	89957	121932	5203
xdelta	30240	1374	774	20402	52874	5110
xmldiff	-	1216	781	39705	52901	4377

It is interesting that `xdelta` shows very good results in terms of size, although we kept the input files properly indented, i.e., subtree moves caused significant changes in whitespace. This should degrade `xdelta` performance, as these changes will break up any large matching blocks. On the other hand, the XML formats encode the change locations using an XML syntax, which is more verbose than the binary file offsets emitted by `xdelta`. We believe that this is the main reason that `xdelta` outperforms the XML formats.

Tables 2 and 3 show the aggregated execution time and compressed output size for each of the projects in the use-cases data set. Missing data, due to failing to produce a diff or exceeding a 600 s timeout for each task, is shown as `-`. The ratios between the top performers for each project are illustrated graphically in Figures 10 and 11.

We find our approach to be comparable in performance to `xydiff`. When comparing execution times between these, we see that the advantage varies from project to project. Our approach is worse in the Rönnau project, whereas `xydiff` fares worse for the SVG project. We conjecture that these variations are due to the different matching heuristics and their suitability for the project in question.

Looking at the overall picture it seems that our approach is viable and has the potential to perform on par with tools of a significantly more complex design, such as `xydiff`. In particular, our approach seems to be scalable and produce

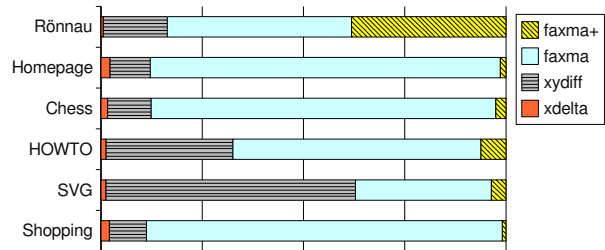


Figure 10: The ratios of execution times of the fastest tools by project.

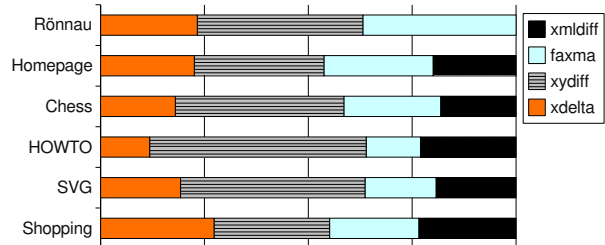


Figure 11: The ratios of compressed output sizes for the most compact diff tools by project.

compact output. Comparing with the linear `xdelta` tool, we see a clear advantage in performance and compactness compared to the XML diffs. It appears that there is a tradeoff between having XML-level changes on one hand and speed and compactness on the other.

6. CONCLUSIONS AND FUTURE WORK

Performance in terms of execution time, rather than output size or diff quality, is currently the most limiting factor of XML differencing tools. We addressed this issue by presenting an XML differencing approach that is based on transforming the XML differencing problem into the domain of sequences, and applying a fast and simple greedy heuristics to compute the difference in this domain. The approach supports the important subtree move operation. We produce XML-level diffs by the virtue of a method for mapping back from matched sequences to matched trees. In short, our approach demonstrates a simple and fast way to calculate XML diffs.

Our implementation aligns sequences of XML parser tokens adhering to the XAS model, and uses an efficient subsequence matching algorithm similar to that of `rsync` for maximum speed. Our design features a clean split between computing the XML difference and encoding it in an XML diff format. This split is advantageous, as it allows us to easily extend our tool with different diff formats. Initial tests on execution time and output compactness are promising, and suggest that our tool can perform on par with high-performance tools of a significantly more complex design. This is especially true with respect to output size.

In addition to benchmarking our approach, our tests illustrate the performance concerns of several current XML diff tools. The tests also demonstrate the advantages in terms of speed and compactness offered by the character-level `xdelta` tool, which may be used when an XML diff is not needed.

In further work we intend to improve differencing of structured text documents by utilizing similarity of text nodes. We will also consider computational complexity, and testing our approach on a larger set of documents and measurements to validate our initial results. Identifying cases where our greedy heuristics fails in terms of speed, compactness, or quality appears to be especially important.

7. REFERENCES

- [1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 39–48, Los Alamitos, CA, USA, Sept. 2000. IEEE Computer Society.
- [2] S. S. Chawathe. Comparing hierarchical data in external memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [3] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press, May 1997.
- [4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, June 1996.
- [5] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18th International Conference on Data Engineering*, pages 41–52, Feb. 2002.
- [6] J.-l. Gailly and M. Adler. *zlib 1.1.4 Manual*, Mar. 2002.
- [7] J. Kangasharju and T. Lindholm. A sequence-based type-aware interface for XML processing. In M. H. Hamza, editor, *Proceedings of the Ninth IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 83–88. ACTA Press, Feb. 2005.
- [8] A. Laux and L. Martin. *XUpdate — XML Update Language*. XML:DB Initiative for XML Databases, Sept. 2000. Working draft.
- [9] T. Lindholm. A 3-way merging algorithm for synchronizing ordered trees — the 3DM merging and differencing tool for XML. Master’s thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, Sept. 2001.
- [10] T. Lindholm, J. Kangasharju, and S. Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In *Fourth International ACM Workshop on Data Engineering for Wireless and Mobile Access*, June 2005.
- [11] S. Y. Lu. A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):219–224, 1979.
- [12] J. MacDonald. File system support for delta compression. Master’s thesis, UC Berkeley, May 2000.
- [13] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
- [14] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.
- [15] S. Rönna, J. Scheffczyk, and U. M. Borghoff. Towards XML version control of office documents. In *ACM Symposium on Document Engineering*, pages 10–19, New York, NY, USA, Nov. 2005. ACM Press.
- [16] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, Dec. 1977.
- [17] D. Shapira and J. A. Storer. Edit distance with move operations. In A. Apostolico and M. Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 85–98, Fukuoka, Japan, July 2002. Springer-Verlag.
- [18] Sun Microsystems. *OpenOffice.org XML File Format 1.0*, Dec. 2002.
- [19] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.
- [20] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.
- [21] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Canberra, Australia, Feb. 1999.
- [22] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In *19th International Conference on Data Engineering*, pages 519–530, Mar. 2003.
- [23] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Path Language (XPath) 1.0*, Nov. 1999. W3C Recommendation.
- [24] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, Aug. 2002. W3C Recommendation.
- [25] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Scalable Vector Graphics (SVG) 1.1 Specification*, Jan. 2003. W3C Recommendation.
- [26] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Document Object Model (DOM) Level 3 Core Specification*, Apr. 2004. W3C Recommendation.
- [27] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Extensible Markup Language (XML) 1.0*, 3rd edition, Feb. 2004. W3C Recommendation.
- [28] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XML Information Set*, 2nd edition, Feb. 2004. W3C Recommendation.
- [29] World Wide Web Consortium, Cambridge, Massachusetts, USA. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, Nov. 2005. W3C Candidate Recommendation.
- [30] World Wide Web Consortium, Cambridge, Massachusetts, USA. *Remote Events for XML (REX) 1.0*, Feb. 2006. W3C Working Draft.
- [31] K. Zhang and D. Shasha. Fast algorithm for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, Dec. 1990.