A comparative study for XML change detection

Grégory Cobéna[*] — **Talel Abdessalem**[**] — **Yassine Hinnach**[**]

[*] *INRIA, France*
Domaine de Voluceau, Rocquencourt BP105, 78153 Le Chesnay Cedex
Gregory.Cobena@inria.fr

[**] *ENST, France*
46, rue Barrault, 75013 Paris
Talel.Abdessalem@enst.fr
YassineHinnach@yahoo.com

ABSTRACT. *Change detection is an important part of version management for databases and document archives. The success of XML has recently renewed interest in change detection on trees and semi-structured data, and various algorithms have been proposed. We study here different algorithms and representations of changes based on their formal definition and on experiments conducted over XML data from the Web. Our goal is to provide an evaluation of the quality of the results, the performance of the tools and, based on this, guide the users in choosing the appropriate solution for their applications.*

RÉSUMÉ. *Dans le cadre des bases de données temporelles ou celui de l'archivage de documents, la détection de changements est un aspect essentiel de la gestion de versions. Le succès de XML a apporté un regain d'intérêt pour les algorithmes de diff s'appliquant à des structures arborescentes et notamment aux données semi-structurées. Récemment, plusieurs algorithmes et modèles ont été proposés, et nous avons souhaité mener une étude comparative de ces solutions. Nous étudions ici, à partir de leurs définitions formelles et des expériences conduites sur les données XML du Web, les différents algorithmes proposés ainsi que les représentations de changements. Notre objectif est d'évaluer la performance des outils et la qualité des résultats obtenus afin d'aider au choix d'une solution appropriée qui réponde aux besoins spécifiques de chaque application.*

KEYWORDS: *XML, Semi-structured Data, diff, Change Detection, Versions, Tree edit problem, Tree pattern matching*

MOTS-CLÉS : *XML, données semi-structurées, détection de changement, versions*

1. Introduction

The context for the present work is change control in XML data warehouses. In such a warehouse, documents are collected periodically, for instance by crawling the Web. When a new version of an existing document arrives, we want to understand changes that occured since the previous version. Considering that we have only the old and the new version for a document, and no other information on what happened between, a *diff* needs to be computed. A typical setting for the *diff* algorithm is as follows: the input consists in two files representing two versions of the same document, the output is a *delta* file representing the changes that occurred.

In this paper, we consider XML input documents and XML *delta* files to represent changes. The goal of this survey is to analyze the different existing solutions and, based on this, assist the users in choosing the appropriate tools for their applications. We study two dimensions of the problem: (i) the representation of changes (ii) the detection of changes.

Representing Changes. To understand the important aspects of changes representation, we point out some possible applications:

– In Version management [CHI 00, MAR 01], the representation should allow for effective *storage strategies* and efficient *reconstruction of versions* of the documents.

– In Temporal Applications [CHA 99b], the support for a *persistent identification* of XML tree nodes is mandatory since one would like to identify (i.e. trace) a node through time.

– In Monitoring Applications [CHE 00, NGU 01], changes are used to detect events and trigger actions. The trigger mechanism involves *queries on changes* that need to be executed in real-time. For instance, in a catalog, finding the product whose type is 'digital camera' and whose price has decreased.

As mentioned above, the *deltas* we consider here are XML documents summarizing the changes. The choice of XML is motivated by the need to exchange, store and query these changes. XML allows to support better quality services as in [CHE 00] and [NGU 01], in particular real query languages [W3C b, AGU 00], and facilitates data integration [W3C a]. Since XML is a flexible format, there are different possible ways of representing the changes on XML and semi-structured data [CHA 98, La 01, MAR 01, XML], and build version management architectures [CHI 00]. In Section 3, we compare change representation models and we focus on recent proposals that have a formal definition, a framework to query changes and an available implementation, namely *DeltaXML* [La 01], *XyDelta* [MAR 01], *XUpdate* [XML] and *Dommitt* [Dom]

Change Detection. In some applications (e.g. an XML document editor) the system knows exactly which changes have been made to a document, but in our context, the sequence of changes is unknown. Thus, the most critical component of change control is the *diff* module that detects changes between an old version of a document and the new version. The input of a *diff* program consists in these two documents, and possibly

their DTD or XMLSchema. Its output is a *delta* document representing the changes between the two input documents. Important aspects are as follow:

– *Correctness:* We suppose that all diffs are "correct", in that they find a set of operations that is sufficient to transform the old version into the new version of the XML document. In other words, they miss no changes.

– *Minimality:* In some applications, the focus will be on the minimality of the result (e.g. number of operations, edit cost, file size) generated by the *diff* . This notion is explained in Section 2. Minimality of the result is important to save storage space and network bandwidth. Also, the effectiveness of version management depends both on minimality and on the representation of changes.

– *Semantics:* Some algorithms consider more than the tree structure of XML documents. For instance, they may consider keys (e.g. ID attributes defined in the DTD) and match with priority two elements with the same tag if they have the same key. In the world of XML, the semantics of data is becoming extremely important [W3C a] and some applications may be looking for semantically correct results or impose semantic constraints, e.g. that a *product* in a catalog is identified by its *name* and that only its *price* might be modified.

– *Performance and Complexity:* With dynamic services and/or large amounts of data, good performance and low memory usage become mandatory. For example, some algorithms find a minimum edit script (given a cost model detailed in Section 2) in quadratic time and space.

– *"Move" Operations:* The capability to detect *move* operations (see Section 2) is only present in certain *diff* algorithms. The reason is that it has an impact on the complexity (and performance) of the *diff* and also on the minimality and the semantics of the result.

To explain how the different criteria affect the choice of a *diff* program, consider the application of cooperative work on large XML documents. Large XML documents are replicated over the network. We want to permit concurrent work on these documents and efficiently update the modified parts. Thus, a *diff* between XML documents is computed. The semantic support of ID attributes allows to divide the document into finer grain structures, and thus to efficiently handle concurrent transactions. Then, changes can be applied (propagated) to the files replicated over the network. When the level of replication is low, priority is given to performance when computing the *diff* instead of minimality of the result.

Experiment Settings. Our comparative study relies on experiments conducted over XML documents found on the web. Xyleme [xyl] crawled more than five hundred millions web pages (HTML and XML) in order to find five hundred thousand XML documents. Because only part of them changed during the time of the experiment (several months), our measures are based roughly on hundred thousand XML documents. Most experiments were run on sixty thousand of them (because of the time it would take to run them on all the available data). It would also be interesting to run

it on private data (e.g. financial data, press data). Such data is typically more regular. We intend to conduct such an experiment in the future.

Observe that our work is intended to XML documents. It can also be used for HTML documents by XML-izing them, a relatively easy task that mostly consists in properly closing tags. However, change management (detection+representation) for a "true" XML document is semantically much more informative than for HTML. It includes pieces of information such as the insertion of particular subtrees with a precise semantics, e.g. a new product in a catalog.

The paper is organized as follows. First, we first present the data, operations and cost model in Section 2. Then, we compare change representations in Section 3. The next section is an in-depth state of the art in which we present change detection algorithms and their implementation programs. In Section 5 we present a performance analysis (speed and memory). Finally, we study the quality of the results of diff programs in Section 6. The last section concludes the paper.

2. Preliminaries

In this section, we introduce the notions that will be used along the paper. The data model we use for XML documents is labeled ordered trees as in [MAR 01]. We will also briefly consider some algorithms that support unordered trees.

Operations. The change model is based on editing operations as in [MAR 01], namely *insert* , *delete* , *update* and *move* . There are various possible interpretations for these operations. For instance, in Kuo-Chung Tai's model [TAI 79], deleting a node means making its children become children of the node's parent. But this model may not be appropriate for XML documents, since deleting a node changes its depth in the tree and may also invalidate the document structure according to its DTD.

Thus, for XML data, we use Selkow's model [SEL 77] in which operations are only applied to leaves or subtrees. For instance, when a node is deleted, the entire subtree rooted at the node is deleted. This captures the XML semantic better, for instance removing a product from a catalog by deleting the corresponding subtree. Important aspects presented in [MAR 01] include (i) management of positions in XML documents (e.g. the position of sibling nodes changes when some are deleted), and (ii) consistency of the sequence of operations depending on their order (e.g. a node can not be updated after one of its ancestors has been deleted).

Edit Cost. The *edit cost* of a sequence of edit operations is defined by assigning a cost to each operation. Usually, this cost is 1 per node touched (inserted, deleted, updated or moved). If a subtree with n nodes is deleted (or inserted), for instance using a single delete operation applied to the subtree root, then the edit cost for this operation is n. Since most *diff* algorithms are based on this cost model, we use it in this study. The *edit distance* between document A and document B is defined by the

minimal edit cost over all edit sequences transforming A in B. A *delta* is *minimal* if its edit cost is no more than the edit distance between the two documents.

One may want to consider different cost models. For instance, assigning the cost 1 for each edit operation, e.g. deleting or inserting an entire subtree. But in this case, a minimal edit script would often consist in the two following operations: (i) delete the first document with a single operation applied to the document's root (ii) insert the second document with a single operation. We briefly mention in Section 6 some results based on a cost model where the cost for *insert* , *delete* and *update* is 1 per node but the cost for *moving* an entire subtree is only 1.

The *move* operation. The semantics of *move* is to identify nodes (or subtrees) even when their context (e.g. ancestor nodes) has changed. Some of the proposed algorithms are able to detect *move* operations between two documents, whereas others do not. We recall that most formulations of the change detection problem with *move* operations are NP-hard [ZHA 95]. So the drawback of detecting *moves* is that such algorithms will only approximate the minimum edit script. The improvement when using a *move* operation is that, in some applications, users will consider that a *move* operation is less costly than a *delete* and *insert* of the subtree. In temporal databases, *move* operations are important to detect from a semantic viewpoint because they allow to identify (i.e. trace) nodes through time better than *delete* and *insert* operations.

Mapping/Matching. In this paper, we will also use the notion of "mapping" between the two trees. Each node in A (or B) that is not deleted (or inserted) is "matched" to the corresponding node in B (or A). A *mapping* between two documents represents all matchings between nodes from the first and second documents. In some cases, a *delta* is said "minimal" if its edit cost is minimal for the restriction of editing sequences compatible with a given "mapping"[1].

The definition of the mapping and the creation of a corresponding edit sequence are part of the "change detection". The "change representation" consists in a data model for representing the edit sequence.

3. Comparison of the Change Representation models

XML has been widely adopted both in academia and in industry to store and exchange data. [CHA 99b] underlines the necessity for querying semistructured temporal data. Recent works [CHA 99b, La 01, CHI 00, MAR 01] study version management and temporal queries over XML documents. Although an important aspect of version management is the representation of changes, a standard is still missing.

In this section we recall the problematic of change representation for XML documents, and we present main recent proposals on the topic, namely *DeltaXML* [La 01] and *XyDelta* [MAR 01]. Then we present some experiments conducted over Web data.

---

1. a sequence based on another mapping between nodes may have a lower edit cost

As previously mentioned, the main motivations for representing changes are: version management, temporal databases and monitoring data. Here, we analyse these applications in terms of (i) versions storage strategies and (ii) querying changes.

Versions Storage Strategies. In [CHI], a comparative study of version management schemes for XML documents is conducted. For instance, two simple strategies are as follow : (i) storing only the latest version of the document and all the deltas for previous versions (ii) storing all versions of the documents, and computing deltas only when necessary. When only deltas are stored, their size (and edit cost) must be reduced. For instance, the delta is in some cases larger than the versioned document. We have analyzed the performance for reconstructing a document's version based on the delta. The time complexity is in all cases linear in the edit cost of the delta. The computation cost for such programs is close to the cost of manipulating the XML structure (reading, parsing and writing).

One may want to consider a flat text representation of changes that can be obtained for instance with the Unix diff tools. In most applications, it is efficient in terms of storage space and performance to reconstruct the documents. Its drawback are: (i) that it is not XML and can not be used for queries (ii) files must be serialized into flat text and this can not be used in native (or relational) XML repositories.

Querying Changes. We recall here that support for both indexing and persistent identification is useful. On one hand, labeling nodes with both their prefix and postfix position in the tree allows to quickly compute ancestor/descendant tests and thus significantly improves querying [AGU 00]. On the other hand, labeling nodes with a persistent identifier accelerates temporal queries and reduces the cost of updating an index. In principle, it would be nice to have one labeling scheme that contains both structure and persistence information. However, [COH 02] shows that this requires longer labels and uses more space.

Also note that using *move* operations is often important to maintain persistent identifiers since using *delete* and *insert* does not lead to a persistent identification. Thus, the support of *move* operations improves the effectiveness of temporal queries.

3.1. *Change Representation models*

We now present change representation models, and in particular *DeltaXML* [La 01] and *XyDelta* [MAR 01]. In terms of features, the main difference between them is that only *XyDelta* supports *move* operations. Except for *move* operations, it is important to note that both representations are formally equivalent, in that simple algorithms can transform a *XyDelta* delta into a *DeltaXML* delta, and conversely.

DeltaXML: In [La 01] (or similarly in [CHA 99b]), the delta information is stored in a "summary" of the original document by adding "change" attributes. It is easy to present and query changes on a single delta, but slightly more difficult to aggregate deltas or issue temporal queries on several deltas. The delta has the same look and feel

as the original document, but it is not strictly validated by the DTD. The reason is that while most operations are described using attributes (with a DeltaXML namespace), a new type of tag is introduced to describe text nodes updates. More precisely, for obvious parsing reasons, the old and new values of a text node cannot be put side by side, and the tags `<deltaxml:oldtext>` and `<deltaxml:newtext>` are used to distinguish them.

There is some storage overhead when the change rate is low because: (i) position management is achieved by storing the root of unchanged subtrees (ii) change status is propagated to ancestor nodes. A typical example would be:

```
<catalog deltaxml:delta='modified'>
  <product deltaxml:delta='unchanged' />
  <product deltaxml:delta='modified'>
    <status deltaxml:delta='deleted'>Unavailable</status>
    <name>Digital Camera</name>
    <description>...</description>
    <price deltaxml:delta='inserted'>$399</price>
  </product>
</catalog>
```

Note that it is also possible to store the whole document, including unchanged parts, along with changed data.

XyDelta: In [MAR 01], every node in the original XML document is given a unique identifier, namely XID, according to some identification technique called *XidMap*. The *XidMap* gives the list of all persistent identifiers in the XML document in the prefix order of nodes. Then, the delta represents the corresponding operations: identifiers that are not found in the new (old) version of the document correspond to nodes that have been deleted (inserted)[2]. The previous example would generate a delta as follows. In this delta, nodes 15-17 (i.e. from 15 to 17) that have been deleted are removed from the *XidMap* of the second version $v2$. In a similar way, the persistent identifiers 31-33 of inserted nodes are now found between node 23 and node 24.

```
<xydelta
  v1_XidMap="(1-30)"
  v2_XidMap="(1-14;18-23;31-33;24-30)">
  <delete xid=(15-17) parent=6 position=1>
    <status>Not Available</status>
  </delete>
  <insert xid=(31-33) parent=6 position=4>
    <price>$399</price>
  </insert>
</xydelta>
```

2. move and update operations are described in [MAR 01]

XyDeltas have nice mathematical properties, e.g. they can be aggregated, inverted and stored without knowledge about the original document. Also the persistent identifiers and *move* operations are useful in temporal applications. The drawback is that the delta does not contains contexts (e.g. ancestor nodes or siblings of nodes that changed) which are sometimes necessary to understand the meaning of changes or present query results to the users. Therefore, the context has to be obtained by processing the document.

XUpdate [XML] provides means to update XML data, but it misses a more precise framework for version management or to query changes.

Dommitt [Dom] representation of changes is in the spirit of *DeltaXML*. However, surprisingly, instead of using change attributes, new node types are created. For instance, when a *book* node is deleted, a *xmlDiffDeletebook* node is used. A drawback is that the delta DTD is significantly different from the document's DTD.

Remark. No existing change representation can be valitaded by (i) either a generic DTD (because of document's specific tags) (ii) or the versioned document's DTD (because of text nodes updates as mentioned previously). These issues will have to be considered in order to define a standard for representing changes of XML documents in XML.

3.2. *Change Representation Experiments*

Figure 1 (page 9) shows the size of a *delta* represented using *DeltaXML* or *XyDelta* as function of the edit cost of the delta. The delta cost is defined according to the "1 per node" cost model presented in Section 2. Each dot represents the average[3] delta file size for deltas with a given edit cost. It confirms clearly that DeltaXML is slightly larger for lower edit costs because it describes many unchanged elements. On the other hand, when the edit cost becomes larger, its size is comparable to XyDelta. The *deltas* in this figure are the results of more than twenty thousand XML diffs, roughly twenty percent of the changing XML that we found on the web.

4. State of the art in Change Detection

In this section, we present an overview of the abundant previous work in this domain. The algorithms we describe are summarized in Figure 2 (page 14).

A *diff* algorithm consists in two parts: first it matches nodes between the two (versions of the same) document(s). Second it generates a document, namely a *delta*, representing a sequence of changes compatible with the matching.

---

3. although fewer dots appear in the left part of the graph, they represent each the average over several hundred measures

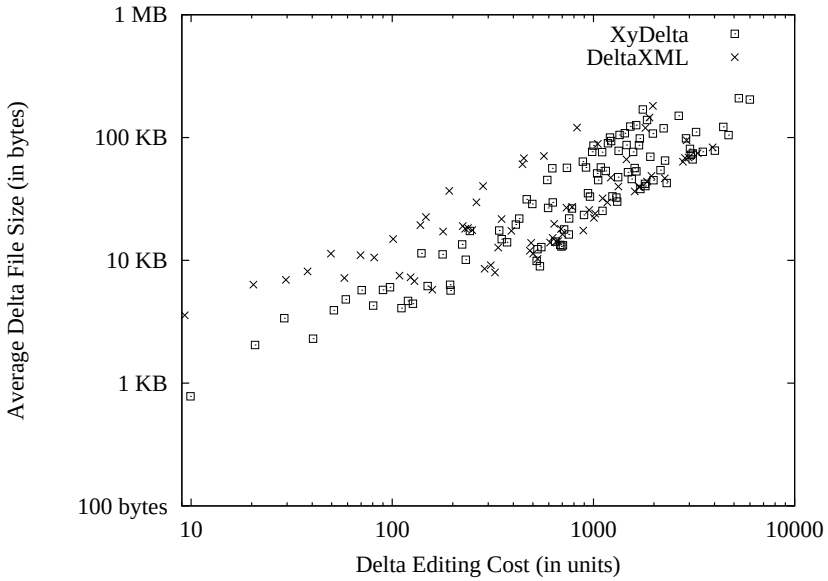

Figure 1. *Size of the delta files*

For most XML *diff* tools, no complete formal description of their algorithms is available. Thus, our performance analysis is not based on formal proofs. We compared the formal upper bounds of the algorithms and we conducted experiments to test the average computation time. Also we give a formal analysis of the minimality of the *delta* results.

Following subsections are organized as follows. First, we introduce the String Edit Problem. Then, we consider optimal tree pattern matching algorithms that rely on the string edit problem to find the best matching. Finally we consider other approaches that first find a meaningful mapping between the two documents, and then generate a compatible representation of changes.

4.1. *Introduction: The String Edit Problem*

Longest Common Subsequence (LCS). In a standard way, the *diff* tries to find a minimum *edit script* between two strings. It is based on edit distances and the string edit problem [APO 97, LEV 66, SAN 83, WAG 74]. Insertion and deletion correspond to inserting and deleting a (single) symbol in a string. A cost (e.g. 1) is assigned to each operation. The string edit problem corresponds to finding an edit script of minimum cost that transforms a string x into a string y. A solution is obtained by considering the cost for transforming prefix substrings of x (up to the i-th symbol) into prefix subtrings of y (up to the j-th symbol). On a matrix $[1..|x|] * [1..|y|]$, a direc-

ted acyclic graph (DAG) representing all operations and their edit cost is constructed. Each path ending on (i, j) represents an edit script to transform $x[1..i]$ into $y[1..j]$. The minimum edit cost $cost(x[1..i] \rightarrow y[1..j])$ is then given by the minimal cost of these three possibilities:

$$cost(deleteCharSymbol(x[i])) + cost(x[1..i-1] \rightarrow y[1..j])$$
$$cost(insertCharSymbol(y[j])) + cost(x[1..i] \rightarrow y[1..j-1])$$
$$cost(updateCharSymbol(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1])$$

The edit distance between x and y is given by $cost(x[1..|x|] \rightarrow y[1..|y|])$, and the minimum edit script by the corresponding path. Note that for example the cost for $updateCharSymbol(x[i], y[j])$ is zero when the two symbols are identical.

The sequence of nodes that are not modified by the edit script (nodes on diagonal edges of the path) is a common subsequence of x and y. Thus, it is equivalant to finding the "Longest Common Subsequence" (LCS) between x and y. Note that each node in the common subsequence defines a matching pair between the two corresponding symbols in x and y.

The space and time complexity are $O(|x| * |y|)$. This algorithm has been improved by Masek and Paterson using the "four-russians" technique [MAS 80] in $O(|x| * |y|/log|x|)$ and $O(|x| * |y| * log(log|x|)/log|x|)$ worst-case running time for finite and arbitrary alphabet sets respectively.

D-Band Algorithms. In [MYE 86], a $O(|x| * D)$ algorithm is exhibited, where D is the size of the minimum edit script. Such algorithms, namely *D-Band* algorithms, consist is computing cost values only close to the diagonal of the matrix. A diagonal k is defined by (i, j) couples with the same difference $i - j = k$, e.g. for $k = 0$ the diagonal contains $(0, 0), (1, 1), (2, 2), ...$ When using the usual "1 per node" cost model, diagonal areas of the matrix, e.g. all diagonals from $-K$ to K, contain all edit scripts of cost lower than a given value K. Obviously, if a valid edit script of cost lower than K is found to be minimum inside the diagonal area, then it must be the minimum edit script. When k is zero, the area consists solely in the diagonal starting at $(0, 0)$. By increasing k, it is then possible to find the minimum edit script in $O(max(|x| + |y|) * D)$ time. Using a more precise analysis of the number of deletions, [WU 90] improves significantly this algorithm performance when the two documents lengths differ substantially. This *D-Band* technique is used by the famous **GNU diff** [FSF] program for text files.

4.2. *Optimal Tree Pattern Matching*

Serialized XML documents can be considered as strings, and thus we could use a "string edit" algorithm to detect changes. This may be used as a raw storage and raw version management, and can indeed be implemented using *GNU diff* that only

supports flat text files. However, in order to support better services, it is preferable to consider specific algorithms for tree data that we describe next. The complexity we mention for each algorithm is relative to the total number of nodes in both documents. Note that the the number of nodes is linear in the document's file size.

Previous Tree Models. Kuo-Chung Tai [TAI 79] gave a definition of the edit distance between ordered labeled trees and the first non-exponential algorithm to compute it. The time and space complexity is quasi-quadratic.

In Selkow's variant [SEL 77], which is closer to XML, the LCS algorithm described previously is used on trees in a recursive algorithm. Considering two documents $D1$ and $D2$, the time complexity is $O(|D1| * |D2|)$. In the same spirit is Yang's [YAN 91] algorithm to find the syntactic differences between two programs.

MMDiff and XMDiff. In [CHA 99a], S. Chawathe presents an external memory algorithm *XMDiff* (based on main memory version *MMDiff*) for ordered trees in the spirit of Selkow's variant. Intuitively, the algorithm constructs a matrix in the spirit of the "string edit problem", but some edges are removed to enforce that deleting (or inserting) a node will delete (or insert) the subtree rooted at this node. More precisely, (i) diagonal edges exists if and only if corresponding nodes have the same depth in the tree (ii) horizontal (resp. vertical) edges from (x, y) to $(x + 1, y)$ exists unless the depth of node with prefix label $x + 1$ in $D1$ is lower than the depth of node $y + 1$ in $D2$. For **MMDiff**, the CPU and memory costs are quadratic $O(|D1| * |D2|)$. With **XMDiff**, memory usage is reduced but IO costs become quadratic.

Unordered Trees. In XML, we sometimes want to consider the tree as unordered. The general problem becomes NP-hard [ZHA 92], but by constraining the possible mappings between the two documents, K. Zhang [ZHA 96] proposed an algorithm in quasi quadratic time. In the same spirit is **X-Diff** [WAN] from NiagaraCQ [CHE 00]. In these algorithms, for each pair of nodes from $D1$ and $D2$ (e.g. the root nodes), the distance between their respective subtrees is obtained by finding the minimum-cost mapping for matching children (by reduction to the minimum cost maximum flow problem [ZHA 96, WAN]). More precisely, the complexity is $O(|D1| * |D2| * (deg(D1) + deg(D2)) * log(deg(D1) + deg(D2))$, where $deg(D)$ is the maximum outdegree (number of child nodes) of D. We do not consider these algorithms since we did not experiment on unordered XML trees. However, their characteristics are similar to *MMDiff* since both find a minimum edit script in quadratic time.

DeltaXML. One of the most featured product on the market is DeltaXML [DEL]. It uses a similar technique based on longest common subsequence computations, more precisely it uses Wu [WU 90, MYE 86] *D-Band* algorithm to run in quasi-linear time. The complexity is $O(|x| * D)$, where $|x|$ is the total size of both documents, and D the edit distance between them. Because the algorithm is applied at each level separately, the result is not strictly minimal. The recent versions of DeltaXML supports the addition of keys (either in the DTD or as attributes) that can be used to enforce correct matching (e.g. always match a *person* by its *name* attribute). DeltaXML also supports unordered XML trees.

Others. In a similar way, IBM developed *XML treediff* [IBM] based on [CUR 99] and [SHA 90]. A first phase is added which consists in pruning identical subtrees based on their hash signature, but it is not clear if the result obtained is still minimal. Sun also released an XML specific tool named *DiffMK* [Sun] that computes the difference between two XML documents. This tool is based on the Unix standard *diff* algorithm, and uses a *list* description of the XML document, thus losing the benefit of the tree structure in XML. The tests that we conducted, and other results found on the web seem to indicate that the current version is not "correct".

For both programs, we experienced difficulties in running the tools on a large set of files[4]. Thus, these two programs were not included in our experiments.

We were surprised by the relatively weak offer in the area of XML diff tools since we are not aware of more featured XML diff products from important companies. We think that this may be due to a missing widely accepted XML change protocol. It may also be the case that some products are not publicly available. Fortunately, the algorithms we tested represent well the spirit of today's tools: quadratic minimum-script finding algorithm (MMDiff), linear-time approximation (DeltaXML), and tree pattern matching with move operations (see next).

4.3. *Tree pattern matching with a* move *operation*

The main reason why few *diff* algorithms supporting *move* operations have been developed earlier is that most formulations of the tree diff problem are NP-hard [ZHA 95, CHA 97] (by reduction from the "exact cover by three-sets"). One may want to convert a pair of *delete* and *insert* operations applied on a similar subtree into a single *move* operation. But the result obtained is in general not minimal, unless the cost of *move* operations is strictly identical to the total cost of deleting and inserting the sutree.

LaDiff. Recent work from S. Chawathe includes *LaDiff* [CHA 96, CHA 97], designed for hierarchically structured information. It introduces a matching criteria to compare nodes, and the overall matching between both versions of the document is decided on this base. A minimal edit script -according to the matching- is then constructed. Its cost is in $O(n * e + e^2)$ where n is the total number of leaf nodes, and e a weighted edit distance between the two trees. Intuitively, its cost is linear in the size of the documents, but quadratic in the number of changes between them. Note that when the change rate is maximized, the cost becomes quadratic in the size of the data. Since we do not have an XML implementation of LaDiff, we could not include it in our experiments.

XyDiff. It has been proposed with one of the authors of the present paper in [COB 02]. *XyDiff* is a fast algorithm which supports *move* operations and XML features like the DTD ID attributes. Intuitively, it matches large identical subtrees found in both doc-

4. other users on the Web seemed to have similar problems

uments, and then propagates matchings. A first phase consists in matching nodes according to the key attributes. Then it tries to match the largest subtrees and considers smaller and smaller subtrees if matching fails. When matching succeeds, parents and descendants of identical nodes are also matched as long as the mappings are unambiguous (e.g. an unambiguous case is when two matched nodes have both a single child node with a given tag name). Its cost in time and space is quasi linear $O(n*log(n))$ in the size n of the documents. It does not, in general, find the minimum edit script.

4.4. *Summary of tested* diff *programs*

As previsouly mentioned, the algorithms are summarized in Figure 2 (page 14). The time cost given here (quadratic or linear) is a function of the data size, and corresponds to the case when there are few changes.

For GNU diff, we do not consider minimality since it does not support XML (or tree) editing operations. However, we mention in Section 6 some analysis of the result file size.

5. Experiments: Speed and Memory usage

As previously mentioned, our XML test data has been downloaded from the web. The files found on the web are on average small (a few kilobytes). To run tests on larger files, we composed large XML files from DBLP [LEY] data source. We used two versions of the DBPL source, downloaded at an interval of one year.

The measures were conducted on a Linux system. Some of the XML diff tools are implemented in C++, whereas others are implemented in Java. Let us stress that we ran tests that show that these algorithms compiled in Java (Just-In-Time compiler) or C++ run on average at the same speed, in particular for large files.

Let us analyze the behaviour of the time function plotted in Figure 3(page 15) . It represents, for each diff program, the average computing time depending on the input file size. On the one hand, *XyDiff* and *DeltaXML* are perfectly linear, as well as *GNU Diff*. On the other hand, *MMDiff* increase rate corresponds to a quadratic time complexity. When handling medium files (e.g. hundred kilobytes), there are orders of magnitude between the running time of linear vs. quadratic algorithms.

For *MMDiff*, memory usage is the limiting factor since we used a 1Gb RAM PC to run it on files up to hundred kilobytes. For larger files, the computation time of *XMDiff* (the external-memory version of *MMDiff*) increases significantly when disk accesses become more and more intensive.

In terms of implementation, *GNU Diff* is much faster than others because it doesn't parse or handle XML. On the contrary, we know -for instance- that *XyDiff* spends

Figure 2. *Quick Summary*

Program Name	**Author**	**Time**	**Memory**	**Moves**	**Minimal Edit Cost**	**Notes**
fully tested						
DeltaXML	DeltaXML.com	linear	linear	no	no	
MMDiff	Chawathe and al.	quadratic	quadratic	no	yes	(tests with our implementation)
XMDiff	Chawathe and al.	quadratic	linear	no	yes	quadratic I/O cost (tests with our implementation)
GNU Diff	GNU Tools	linear	linear	no	-	no XML support (flat files)
XyDiff	INRIA	linear	linear	yes	no	
not included in experiments						
LaDiff	Chawathe and al.	linear	linear	yes	no	criteria based mapping
XMLTreeDiff	IBM	quadratic	quadratic	no	no	
DiffMK	Sun	quadratic	quadratic	no	no	no tree structure
XML Diff	Dommitt.com					we were not allowed to discuss it
Constrained Diff	K. Zhang	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping
X-Diff	Y. Wang, D. DeWitt, Jin-Yi Cai (U. Wisconsin)	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping

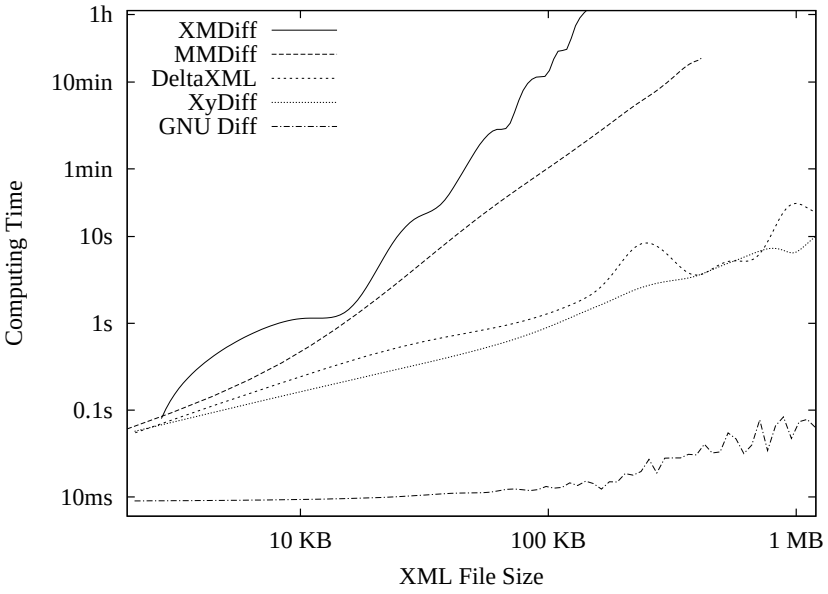

Figure 3. *Speed of different programs*

ninety percent of the time in parsing the XML files. This makes *GNU Diff* very performant for simple text-based version management schemes.

A more precise analysis of *DeltaXML* results is depicted in Figure 4 (page 16). Its shows that although the average computation time is linear, the results for some documents are significantly different. Indeed, the computation time is almost quadratic for some files. We found that it corresponds to the worst case for *D-Band* algorithms: the edit distance D (i.e. the number of changes) between the two documents is close to the number of nodes N. For instance, in some documents, 40 percent of the nodes changed, whereas in other documents less than 3 percent of the nodes changed. This may be slight disadvantage for applications with strict time requirements, e.g. computing the diff over a flow of crawled documents as in NiagaraCQ [CHE 00] or Xyleme [NGU 01]. On the contrary, for *MMDiff* and *XyDiff*, the variance of computation time for all the documents is small. This shows that their average complexity is equal to the upper bound.

6. Experiments: Quality of the result

The "quality" study in our benchmark consists in comparing the sequence of changes generated by the different algorithms. We used the result of *MMDiff* and *XMDiff* as a

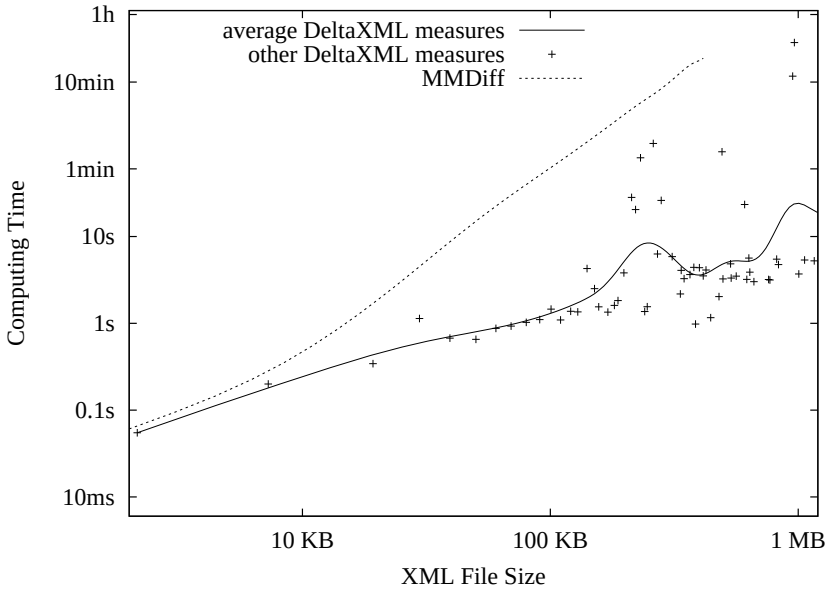

Figure 4. *Focus on DeltaXML speed measures*

reference because these algorithms find the minimum edit script. Thus, for each pair of documents, the quality for a diff tool (e.g. DeltaXML) is defined by the ratio

$$r = \frac{C}{C_{ref}}$$

where C is the delta edit cost and C_{ref} is *MMDiff* delta's edit cost for the same pair of documents. A quality equals to one means that the result is minimum and is considered "perfect". When the ratio increases, the quality decreases. For instance, a ratio of 2 means that the delta is twice more costly than the minimum delta. In our first experiments, we didn't consider *move* operations. This was done by replacing for *XyDiff* each *move* operation by the corresponding pair of *insert* and *delete* . In this case, the cost of moving a subtree is identical to the cost of deleting and inserting it.

In Figure 5 (page 17), we present an histogram of the results, i.e. the number of documents in some range of quality. *XMDiff* and *MMDiff* do not appear on the graph because they serve as reference, meaning that all documents have a quality strictly equal to one. *GNU Diff* do not appear on the graph because it doesn't construct XML (tree) edit sequences.

These results in Figure 5 show that:

– **(i) DeltaXML:** For most of the documents, the quality of *DeltaXML* result is perfect (strictly equal to 1). For the others, the delta is on average thirty percent more costly than the minimum.

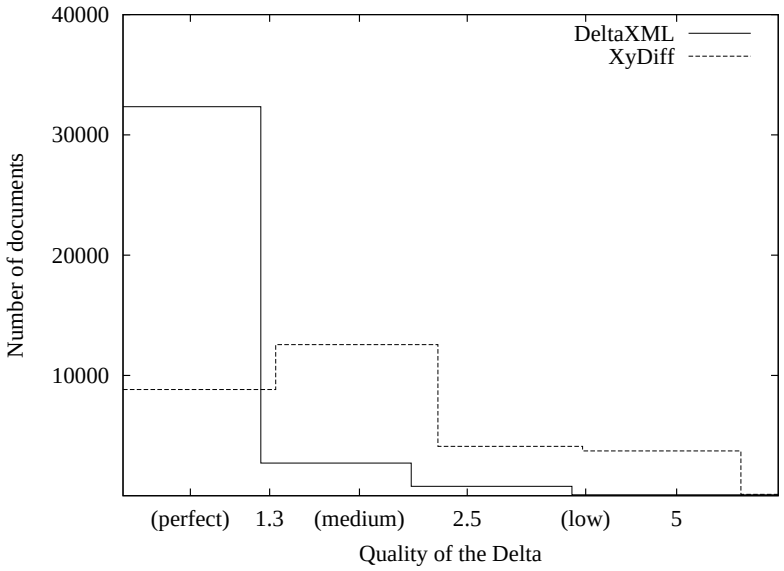

Figure 5. *Quality Histogram*

– **(ii) XyDiff:** Almost half of the deltas are less than twice more costly than the minimum. The other half costs on average three times the minimum.

Result file size. In terms of file sizes, we also compared the different delta documents, as well as the flat text result of *GNU Diff*. The result *diff* files for *DeltaXML*, *GNU Diff* and *XyDiff* have on average the same size. The result files for *MMDiff* are on average twice smaller (using a *XyDelta* representation of changes).

Using "move". We also conducted experiments by considering *move* operations and assigning them the cost 1. Intuitively this means that *move* is considered cheaper than deleting and inserting a subtree, e.g. moving files is cheaper than copying them and deleting the original copy. Only *XyDiff* detects *move* operations. On average, *XyDiff* performs a bit better, and it particular becomes better than *MMDiff* for five percent of the documents.

Finally, note that this quality measure focuses on the minimality of results. In some applications, the semantics of the results is more important. But the semantic value can not be easily measured. An interesting aspect is the support of (semantic) matching rules by some programs (DeltaXML, XyDiff). More work is clearly needed in the direction of evaluating the semantic quality of results. We also intend to conduct experiments on *LaDiff* [CHA 96] which is a good example of criteria-based mapping and change detection.

7. Conclusion

In this paper, we described existing works on the topic of change detection in XML documents.

We first presented the two recent proposals for change representation, and compared their features through analysis and experiments. Both support XML queries and version management, but the identification-based scheme (XyDelta) is slightly more compact for small deltas, whereas the delta-attributes based scheme (DeltaXML) is more easily integrated in simple applications. A key feature of *XyDelta* is the support of node identifiers and *move* operations that are used in temporal XML databases.

More work is clearly needed in that direction to define a common standard for representing changes.

The second part of our study concerns change detection algorithms. We compared two main approaches, the first one consists in computation of minimal edit scripts, while the second approach relies on meaningfull mappings between documents. We underlined the need for semantical integration in the change detection process. The experiments presented show (i) a significant quality advantage for minimal-based algorithms (DeltaXML, MMDiff) (ii) a dramatic performance improvement with linear complexity algorithms (GNU Diff, XyDiff and DeltaXML).

On average, DeltaXML [DEL] seems the best choice because it runs extremely fast and its results are close to the minimum. It is a good trade-off between XMDiff (pure minimality of the result but high computation cost) and XyDiff (high performance but lower quality of the result). We also noted that flat text based version management (GNU Diff) still makes sense with XML data for performance critical applications.

Although the problem of "diffing" XML (and its complexity) are better and better understood, there is still room for improvement. In particular, *diff* algorithms could take better advantage of semantic knowledge that we may have on the documents or may have infered from their histories.

Acknowledgments We would like to thank Serge Abiteboul, Vincent Aguiléra, Robin La Fontaine, Amélie Marian, Tova Milo, Benjamin Nguyen and Bernd Amann for discussions on the topic.

8. References

[AGU 00] AGUILÉRA V., CLUET S., VELTRI P., VODISLAV D., WATTEZ F., "Querying XML Documents in Xyleme", *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

[APO 97] APOSTOLICO A., GALIL Z., Eds., *Pattern Matching Algorithms*, Oxford University Press, 1997.

[CHA 96] CHAWATHE S., RAJARAMAN A., GARCIA-MOLINA H., WIDOM J., "Change detection in hierarchically structured information", *SIGMOD*, vol. 25, num. 2, 1996, p. 493-504.

[CHA 97] CHAWATHE S., GARCIA-MOLINA H., "Meaningful Change Detection in Structured Data", *SIGMOD*, Tuscon, Arizona, May 1997, p. 26-37.

[CHA 98] CHAWATHE S., ABITEBOUL S., WIDOM J., "Representing and querying changes in semistructured data", *ICDE*, 1998.

[CHA 99a] CHAWATHE S., "Comparing Hierarchical Data in External Memory", *VLDB*, 1999.

[CHA 99b] CHAWATHE S. S., ABITEBOUL S., WIDOM J., "Managing Historical Semistructured Data", *Theory and Practice of Object Systems*, vol. 5, num. 3, 1999, p. 143–162.

[CHE 00] CHEN J., DEWITT D. J., TIAN F., WANG Y., "NiagaraCQ: a scalable continuous query system for Internet databases", *SIGMOD*, 2000.

[CHI] CHIEN S., TSOTRAS V., ZANIOLO C., "A Comparative Study of Version Management Schemes for XML Documents", TimeCenter Technical Report TR51, Sept. 2000.

[CHI 00] CHIEN S.-Y., TSOTRAS V. J., ZANIOLO C., "Version Management of XML Documents", *WebDB (Informal Proceedings)*, 2000.

[COB 02] COBÉNA G., ABITEBOUL S., MARIAN A., "Detecting Changes in XML Documents", *ICDE*, 2002.

[COH 02] COHEN E., KAPLAN H., MILO T., "Labeling dynamic XML trees", *PODS*, 2002.

[CUR 99] CURBERA F., EPSTEIN D., "Fast difference and update of XML documents", *XTech*, 1999.

[DEL] DELTAXML, "Change Control for XML in XML", www.deltaxml.com.

[Dom] DOMMITT INC., "XML Diff and Merge tool", www.dommitt.com.

[FSF] FSF, "GNU Diff", www.gnu.org/software/diffutils/diffutils.html.

[IBM] IBM, "XML Treediff", www.alphaworks.ibm.com/.

[La 01] LA FONTAINE R., "A Delta Format for XML: Identifying changes in XML and representing the changes in XML", *XML Europe*, 2001.

[LEV 66] LEVENSHTEIN V. I., "Binary codes capable of correcting deletions, insertions, and reversals", *Cybernetics and Control Theory 10*, , 1966, p. 707-710.

[LEY] LEY M., "DBLP", dblp.uni-trier.de/.

[MAR 01] MARIAN A., ABITEBOUL S., COBÉNA G., MIGNET L., "Change-centric Management of Versions in an XML Warehouse", *VLDB*, , 2001.

[MAS 80] MASEK W., PATERSON M., "A faster algorithm for computing string edit distances", *J. Comput. System Sci.*, 1980.

[MYE 86] MYERS E., "An O(ND) difference algorithm and its variations", *Algorithmica*, 1986.

[NGU 01] NGUYEN B., ABITEBOUL S., COBÉNA G., PREDA M., "Monitoring XML Data on the Web", *SIGMOD*, 2001.

[SAN 83] SANKOFF D., KRUSKAL J., "Time warps, String Edits, and Macromolecules", *Addison-Wesley, Reading, Mass.*, , 1983.

[SEL 77] SELKOW S. M., "The tree-to-tree editing problem", *Information Processing Letters, 6*, , 1977, p. 184-186.

[SHA 90] SHASHA D., ZHANG K., "Fast algorithms for the unit cost editing distance between trees", *J. Algorithms, 11*, , 1990, p. 581-621.

[Sun] SUN MICROSYSTEMS, "Making All the Difference", http://www.sun.com/xml/developers/diffmk/.

[TAI 79] TAI K., "The tree-to-tree correction problem", *Journal of the ACM, 26(3)*, july 1979, p. 422-433.

[W3C a] W3C, "Resource Description Framework", www.w3.org/RDF.

[W3C b] W3C, "XQuery", www.w3.org/TR/xquery.

[WAG 74] WAGNER R., FISCHER M., "The string-to-string correction problem", *Jour. ACM 21*, , 1974, p. 168-173.

[WAN] WANG Y., DEWITT D. J., CAI J.-Y., "X-Diff: A Fast Change Detection Algorithm for XMLDocuments", http://www.cs.wisc.edu/ yuanwang/xdiff.html.

[WU 90] WU S., MANBER U., MYERS G., "An O(NP) sequence comparison algorithm", *Information Processing Letters*, 1990, p. 317-323.

[XML] XML DB, "XUpdate", http://www.xmldb.org/xupdate/.

[xyl] "Xyleme", `www.xyleme.com`.

[YAN 91] YANG W., "Identifying syntactic differences between two programs", *Software - Practice and Experience, 21, (7)*, , 1991, p. 739-755.

[ZHA 92] ZHANG K., STATMAN R., SHASHA D., "On the editing distance between unordered labeled trees", *Information Proceedings Letters 42*, , 1992, p. 133-139.

[ZHA 95] ZHANG K., WANG J. T. L., SHASHA D., "On the editing distance between undirected acyclic graphs and related problems", *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, 1995, p. 395-407.

[ZHA 96] ZHANG K., "A Constrained Edit Distance Between Unordered Labeled Trees", *Algorithmica*, 1996.