Detecting Changes in XML Documents*

Grégory Cobéna, Serge Abiteboul
INRIA, Rocquencourt France
<firstname>.<lastname>@inria.fr

Amélie Marian
Columbia University/ N.Y. USA
amelie@cs.columbia.edu

March 16, 2001

Abstract We present a *diff* algorithm for XML data. This work is motivated by the support for change control in the context of the Xyleme project that is investigating warehouses capable of storing massive volume of XML data found on the web. Because of the context, the algorithm has to be very efficient in terms of speed and memory space even at the cost of some loss of "quality". Also, it considers, besides insertions, deletions and updates (standard in diffs), a *move* operation on subtrees that is essential in the context of XML.

Intuitively, our *diff* algorithm uses signatures to match (large) subtrees that were left unchanged between the old and new versions. Such exact matchings are then possibly propagated to ancestors and descendants to obtain more matchings. We provide a performance analysis of the algorithm. We show that it runs in average in linear time vs. quadratic time for previous algorithms. We present experiments on synthetic data that confirm the analysis. This linear time is obtained by trading some quality. We present experiments (again on synthetic data) that show that the output of the algorithm is reasonably close to the optimal in terms of quality. Finally we present experiments on a small set of versions of XML pages found on the Web.

Keywords: XML, Diff, Delta, Tree Pattern Matching, Change Control, Version.

1 Introduction

Data publication on the web is constantly increasing. Users are often not only interested in the current values of documents and query answers but also in changes. They want to see changes as information that can be used to learn about the evolution of the web. The present work has been performed in the context of the Xyleme project [30] that is studying and building a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing massive volume of XML data found on the web. Xyleme supports change control mechanisms based on *deltas* (a representation of changes) and persistent identifiers for document nodes [19]. The core of the versioning system is a *diff* mechanism for XML data that is the topic of the present paper. The *diff* allows to identify nodes in consecutive versions of the same document. It produces the delta between the two versions. Clearly, such a *diff* is useful in a much wider setting.

The main contribution of the paper is thus a new *diff* algorithm. There has been a lot of work on *diff* algorithms for strings, e.g., [13, 11, 1], for relational data, e.g., [17], or even

*Work supported in part by R.N.R.T.

for tree data, e.g., [28, 9]. The originality of our work comes from the particular nature of the data we handle, namely XML, and from strict performance requirements imposed by the context of Xyleme. We discuss in the paper motivations for yet a new *diff* that come primarily from specific requirements imposed by the context. We present the algorithm and analyze it. Finally, we present some experimental results obtained in the framework of the Xyleme project.

XML is becoming the new standard for semistructured data exchange over the Internet [27, 2]. Thanks to XML, it is becoming possible to deliver higher level services that are difficult or impossible to support with the current HTML technology. In particular, with XML, it is possible to support a *real* query languages [12, 23] or provide some automatic semantic integration. It is also possible to offer a more sophisticate control of changes. For instance, one can achieve monitoring at a finer grain than just the document level and, for instance, detect specific changes to particular elements, e.g., an insertion of a new product in a catalog [22]. In a system such as Xyleme, the management of changes is essential for a number of reasons ranging from traditional support for versions and temporal queries, to more specific ones such as index maintenance or support for query subscriptions. These motivations are briefly considered in Section 2.

The system acquires XML data from the web and maintains it up-to-date [21]. Thus, for a particular document, a sequence of snapshots [3] is obtained. Also, each continuous query is evaluated regularly and produces a sequence of query answers, i.e., a sequence of snapshots of an XML document as well. Note that the system does not have a real time vision of the data. The time we fetch a document may differ from the time of the last update as posted in the header of the document that itself may differ from the actual time of this update. Furthermore, we may typically "miss" some updates. This has to be accepted in the current web context. In the paper, we ignore this issue; and when we mention the time of a version, we mean the time the system acquired this version.

The sequence of snapshots forms the basis of our management of changes. The modifications that occurred between time t_{i-1} and time t_i are computed using the *diff* algorithm. In a standard way, we try to find a minimum *edit script* between the versions at time t_{i-1} and t_i. Because of the context, our algorithm has to be very efficient in terms of speed and memory space even at the cost of some loss of quality. This is because the computation of *diff* should not slow down the loading of documents in the system. We also take advantage of the specificities of XML data. Intuitively, the algorithm works as follows. It tries to detect (large) subtrees that were left unchanged between the old and new versions. These are matched. Starting from there, the algorithm tries to match more nodes by considering ancestors and descendants of matched nodes and taking labels into consideration. A brief analysis shows that we should expect the algorithm to run in linear time.

As previously mentioned, the present work has been realized in the context of the Xyleme project. Like the rest if the system, the *diff* and the versioning system are implemented in C++, under Linux, with Corba for communications. We performed tests to validate our choices. We briefly present some experimentation. They show that the algorithm indeed runs in linear time. We also analyze experimentally the quality of the *diff*. For that, we ran it on synthetic data. We will see that the computed changes are very close in size to the synthetic (perfect) changes. We also ran it on a small set of real data (versions of XML documents obtained on the web). We will see that the size is comparable to that of the Unix Diff. This should be viewed as excellent since our description of changes typically contains much more information than a Unix Diff.

We present motivations in Section 2 and consider specific requirements for our *diff*. A brief overview of the change model of [19] is given in Section 3. In Section 4, we present our *diff* algorithm, and its analysis. We compare it to previous *diff* algorithms in Section 5. Measures are presented in Section 6. The last section is a conclusion.

2 Motivation and Requirements

In this section, we consider motivations for the present work. We then consider some specific requirements for our *diff* algorithm.

As mentioned in the introduction, the role of the *diff* algorithm is to provide support for the control of changes in a warehouse of massive volume of XML documents found on the web. Detecting changes in such an environment serves many purposes:

- Versions: [19] We may want to version a particular document, (part of) a Web site, or the results of a continuous query. This is the most standard use of versions, namely recording history.

- Querying the past: One might want to ask a query about the past, e.g., ask for the value of some particular element at some previous time. Also, one might want to query changes, e.g., ask for the list of all items recently introduced in a catalog.

- Learning about changes: A user may request some XML documents or some XML query result, say V_i at time i. Later, the user may request a new version, say V_{now}. The delta from time i to *now* is a possible description of the changes. It allows to update the old version V_i and also to explain the changes to the user. This is in the spirit, for instance, of the Information and Content Exchange, ICE [29, 14, 16].

- Monitoring changes: We implemented a subscription system [22]. We want to be able to detect changes of interest in XML documents, e.g., that a new product has been added to a particular catalogue. To do that, at the time we obtain a new version of some data, we compute the delta and verify if some of the changes that have been detected are relevant to some subscriptions. Related works on subscription systems that use filtering tools for information dissemination have been presented, e.g., in [31, 5].

- Indexing: In Xyleme, we maintain a full-text index over a large volume of XML documents. To support queries using the structure of data, we need to store structural information for every indexed word of the document [4]. We are considering the possibility to use the delta to maintain such indexes.

- Querying changes: Deltas are XML documents and thus may be queried like any other documents. For instance, one may ask for the list of all items recently introduced in a catalog, or for the new movies showing in London this week. Since deltas are XML documents, such queries are regular queries over documents.

- Sharing resources: In this scenario, different users work on the same XML document. They modify it off-line, and later want to synchronize their respective versions. The deltas and the diff algorithm are an appropriate way of detecting and describing the modifications that occurred in order to detect conflicts and resolve some of them [25].

Based on these requirements, we designed a logical and physical representation of changes that we describe in companion paper [19]. For each document obtained from the web or continuous query, the starting point is a sequence of snapshots of an XML document. We use a representation of the history based on *deltas*. A brief overview of the model of deltas we use is presented in Section 3. Our deltas use identifiers for nodes in the document. A main role of the *diff* is to assign identifiers to nodes by "matching" nodes from consecutive versions. The history is a sequence of deltas where each delta describes a set of updates to transform one version into the next one.

The *diff* therefore plays a central role in the Xyleme system. Consider a portion of the architecture of the Xyleme system in Figure 1. When a new version of a document $V(n)$ is loaded from the web, it is installed in the repository. It is sent to the *diff* module that also acquires the previous version $V(n-1)$ from the repository. The *diff* modules computes a delta, i.e., an XML document describing the changes. This delta is appended to the existing sequence of delta for this document. The old version is then removed from the repository. The alerter is in charge of detecting, in the document $V(n)$ or in the delta, patterns that may interest some subscriptions [22]. Efficiency is here a key factor. In the system, one web crawler loads millions of Web pages per day. Among those, we expect more and more will be XML. The *diff* has to run at the speed of the indexer (not to slow down the system). It also has to use little memory so that it can share a PC with other modules such as the Alerter (to save on communications).

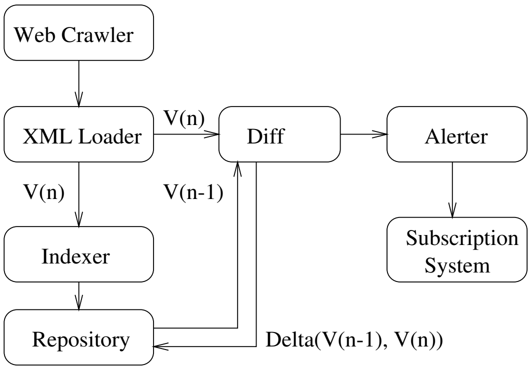

Figure 1: Xyleme-Change architecture

These performance requirements are essential. The context also imposes requirements for the deltas: they should allow (i) reconstructing an old version, and (ii) constructing the changes between some versions n and n'. These issues are addressed in [19]. The *diff* algorithm should also satisfy some quality requirements. Typically, quality is described by some minimality criteria. More precisely, the *diff* should construct a minimum set of changes to transform on version into the next one. Minimality is important because it captures to some extent the semantics that a human would give when presented with the two versions. It is important also in that we store the delta and more compact deltas provide savings in storage. However, in our context, it is acceptable to trade some (little) minimality for better performance.

We will see that using specificities of the context (in particular the fact that documents are in XML) allows the algorithm to obtain changes that are close to the minimum and to do that very efficiently. The specific aspects of our *diff* algorithm are as follows:

- Our *diff* is, like [7, 9], tailored to tree data and takes advantage of specificities of XML such as the existence of labels.
- The *diff* has insert/delete/update operations as in other tree *diff* such as [7, 9]. It also supports a *move* operation to avoid having to insert and delete large subtrees of data that did not change.

3 Brief overview of the Model

In this section, we present some aspects of a model for change [19] that we use in the present paper. The presentation will be very brief and omit many aspects of the complete model.

The simple model for XML data we consider roughly consists of ordered trees (each node may have a *list* of children) [2]. Nodes also have values (data for text nodes and label for element nodes). We will briefly mention later some specific treatment for attributes. The starting point for the change model is a sequence of snapshots of some XML data. A delta is an XML document that represents the changes between two snapshot versions of an XML document. It uses persistent node identifiers in a critical way. We consider next the persistent identification of XML nodes, and then the *deltas*, a novel representation of changes in XML documents.

Persistent identification of nodes The persistent identification of nodes is the basis of the change representation for XML documents we use. We start by assigning to every node of the first version of an XML document a unique identifier, for example its postfix position. When a new version of the document arrives, we use the *diff* algorithm to match nodes between the two versions. Matched nodes in the new document thereby obtain their (persistent) identifiers from their matching in the previous version. New persistent identifiers are assigned to unmatched nodes. Given a set of matchings between two versions of an XML document, there are only few *deltas* that can describe the corresponding changes. The differences between these *deltas* essentially come from move operations that reorder a subsequence of child nodes for a given parent [19]. More details on the definition and storage of our persistent identifiers, that we call XIDs, are given in [19].

Representing changes The delta is a *set* of the following elementary operations: (i) the deletion or insertion of subtrees; (ii) an update of the value of some node (value of a text node or an attribute); and (iii) a move of a node. Note that it is a *set* of operations. Positions in operations are always referring to positions in the source or target document. For instance, $move(m, n, o, p, q)$ specifies that node o is moved from being the n-th child of node m to being the q-th child of p. The management of positions greatly complicates the issue comparing to, say, changes in relational systems. Note also that the model of change we use relies heavily on the persistent identification of XML nodes. It is based on "completed" deltas that contain redundant information. For instance, in case of updates, we store the old and new value. Indeed, a delta specifies both the transformation from the old to the new version, but the inverse transformation as well. Nice mathematical and practical properties of completed deltas are shown in [19]. In particular, we can reconstruct any version of the document given another version and the corresponding delta, and we can aggregate and inverse *deltas*. Finally, observe that the fact that we consider *move* operations is a key difference with most previous works. It seems necessary in an XML context at least to deal with permutations of the children of a node (a frequently occurring situation) but to handle more general moves as well, that we believe are rather frequent.

For example consider the following XML document:

```
<Category>
  <Title>Digital Cameras</Title>
  <Discount>
    <Product>
      <Name>tx123</Name><Price>$499</Price>
      </Product></Discount>
  <NewProducts>
    <Product>
      <Name>zy456</Name><Price>$799</Price>
      </Product></NewProducts></Category>
```

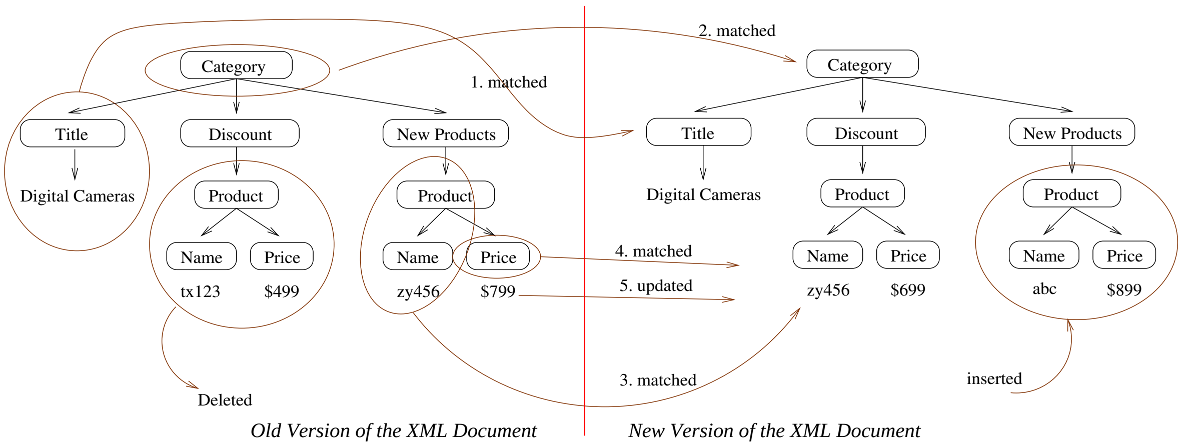

Figure 2: Matching subtrees

Its tree representation is given in the left part of Figure 2. When the document changes, Figure 2 shows how we identify the subtrees of the new version to subtrees in the previous version of the document. This identification is the main goal of the *diff* algorithm we present here. Once nodes from the two versions have been matched, it is possible to produce a delta. (The main difficulty is the management of ranks.) Assuming some identification of nodes in the old version (namely postfix order in the example), the delta representing changes from the old version to the new one may be:

```
<delete XID=7 XID-map="3-7" parentXID=8 pos=1>
  <Product>
    <Name>tx123</Name><Price>$499</Price>
    </Product>
</delete>
<insert XID=20 XID-map="16-20" parentXID=14 pos=1>
  <Product>
    <Name>abc</Name><Price>$899</Price>
    </Product>
</insert>
```

```
<move XID=13 fromParent=14 fromPos=1
               toParent= 8 toPos  =1/>
<update XID=11>
  <oldval>$799</oldval><newval>$699</newval>
  </update>
```

The XID-map [19] is a mechanism for assigning identifiers to nodes (here assigning them to new nodes) that we will not explain here because of space limitations.

It is not easy to evaluate the quality of a *diff*. Indeed, in our context, different usages of the *diff* may use different criteria. Typical criteria could be the size of the delta or the number of operations in it. Choices in the design of the algorithm or in its tuning may result in different deltas, and so different interpretations of the changes that happened between two versions.

4 The BULD Diff Algorithm

In this section, we introduce a novel algorithm that computes the difference between two XML documents. Its use is mainly to match nodes from the two documents and construct a delta that represents the changes between them. We provide a cost analysis for this algorithm. A comparison with previous works is given in Section 5. Intuitively, our algorithm finds matchings between common large subtrees of the two documents and propagate these matchings. BULD stands for Bottom-Up, Lazy-Down propagation. This is because matchings are propagated bottom-up and (most of the time) only lazily down. This approach was prefered to other approaches we considered because it allows to compute the *diff* in linear time. We next give some intuition, then a more detailed description of the algorithm.

4.1 Intuition

To illustrate the algorithm, suppose we are computing the changes between XML document $D1$ and XML document $D2$, $D2$ being the most recent version.

The starting point of the algorithm is to match the largest identical parts of both documents. So we start by registering in a map a unique signature for every subtree of the old document $D1$. Then we consider every subtree in $D2$, starting from the largest, and try to find whether it is identical to some of the registered subtrees of $D1$. If so, we match both subtrees. (This results in matching every node of the subtree in $D1$ with the respective node of the subtree in $D2$.) For example, in Figure 2, we do not find an identical subtree for the tree starting at *Category*, but the subtree starting at *Title* is matched.

We can then attempt to match the parents of two matched subtrees. We do that only if they have the same labels. Clearly, there is a risk of forcing wrong matches by doing so. Thus, we control the propagation of a matching bottom-up based on the length of the path to the ancestor path and the weight of the matching subtrees. For example, a large subtree may force the matching of its ancestors up to the root, whereas matching a small subtree may not even force the matching of its parent.

The fact that the parents have been matched may then help detect matchings between descendants because pairs of such subtrees are considered as good *candidates* for a match. The matching of large identical subtrees may thus help matching siblings subtrees which are slightly different. To see an example, consider Figure 2. The subtree *Name/zy456* is matched. Then its parent *Product* is matched too. The parents being matched, the *Price* nodes may eventually be matched, although the subtrees are different. (This will allow detecting that the price was updated.) When both parents have a single child with a given label, we propagate the match immediately. (It is possible to use data structures that allow detecting such situations at little cost.) Otherwise, we do not propagate the matching immediately. Future matchings (of smaller subtrees) may eventually result in matching them at little cost.

The lazy propagation downward of our algorithm is an important distinction from previous works on the topic. Note that if the two matched nodes have m and m' children with the same label ℓ, we have $m \times m'$ pairs to consider. Attempting this comparison on the spot would result in a quadratic computation.

We start by considering the largest subtrees in $D2$. The first matchings are clear, because it is very unlikely that there is more than one large subtree in $D1$ with the same signature. However it is often the case that when the algorithm goes on and considers smaller and smaller subtrees, more than one subtrees of $D1$ match. We say then that these subtrees are candidates to matching the considered subtree of $D2$. At this point, we use the precedent matches to determines the best candidate among them, by determining which is closest to the existing matches. Typically, if one of the candidate has its parent already matched to the parent of the considered node, it is certainly the best candidate. And thanks to the order in which nodes are considered, the position plays an important role too.

When this part of the algorithm is over, we have considered and perhaps matched every node of $D2$. There are two reasons why a node would have no matching: either because it represents new data that has been inserted in the document, or because we missed matching it. The reason why the algorithm failed may be that at the time the node was considered, there was no sufficient knowledge or reasons to allow a match with one of its candidates. But based on the more complete knowledge that we have now, we can do a peephole optimization pass to retry some of the rejected nodes.

In Figure 2, the nodes *Discount* has not been matched yet because the content of its subtrees has completely changed. But in the optimization phase, we see that it is the only subtree of node *Category* with this label, so we match it.

Once no more matchings can be obtained, unmatched nodes in $D2$ correspond to inserted nodes, whereas unmatched nodes in $D1$ correspond to deleted nodes. For instance, in Figure 2, the subtrees for products *tx123* and *abc* could not be matched and so are respectively considered as deleted and inserted data. Finally, a computationally non negligible task is to consider each matching node and decide if the node is at its right place, or whether it has been moved.

4.2 Detailed description

The various phases of the algorithm are detailed next.

Phase 1 (Compute signatures and order subtrees by weight): In one traversal of each tree, we compute the signature of each node of the old and new documents. The signature is computed using the node's content, and its children signatures. Thus it depends on the entire subtree rooted at that node. A weight is computed simultaneously for each node. It is the size of its content for text nodes and the sum of the weights of its children for element nodes.

We construct a priority queue designed to contain subtrees from the new document. The subtrees are represented by their roots, and the priority is given by the weights. The queue is

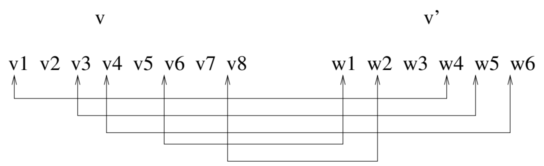

Figure 3: Local moves

used to provide us with the next heaviest subtree for which we want to find a match. (When several nodes have the same weight, the first subtree inserted in the queue is chosen.) To start, the queue only contains the root of the entire new document.

Phase 2 (Try to find matchings starting from heaviest nodes): We remove the heaviest subtree of the queue, e.g. a node in the new document, and construct a list of *candidates*, e.g. nodes in the old document that have the same signature. From these, we get the best candidate (see later), and match both nodes. If there is no matching and the node is an element, its children are added to the queue. If there are many candidates, the best candidate is one whose parent matches the reference node's parent, if any. If no candidate is accepted, we look one level higher. The number of levels we accept to consider depends on the node weight.

When a candidate is accepted, we match the pair of subtrees and their ancestors as long as they have the same label. The number of ancestors that we match depends on the node weight.

Phase 3 (Use structure to propagate matchings): We traverse the tree top-down and try to match nodes from the old and new documents such that their parents are matching and they have the same label. This propagation pass significantly improves the quality of the delta and more precisely avoids detecting unnecessary insertions and deletions.

Phase 4 (Compute the delta): This last phase can itself be split in 3 steps:

1. Inserts/Deletes: Find all unmatched nodes in the old/new document, mark them as *deleted/inserted*; record the effect of their deletion/insertion into position-tables for their parents.

2. Moves: Find all nodes that are matched but with non matching parents. These correspond to *moves*. We record the effect of the delete part of the move and that of the insert part in the position-tables for their parents. Nodes that have the same parent in the new document as in the old document may have been moved within these parents. This is discussed further.

3. These operations are reorganized and the delta is produced. (Details omitted.)

Let us now consider the issue of *moves* within the same parents. For this, consider two nodes v (in the old) and w (in the new) that have been matched. There may have been deletions and moves from v, insertions and moves to w. The remaining children of v and w are the same. More precisely, they match each other. However, they need not be in the same order. When they are not, we need to introduce more moves to capture the changes. See Figure 3 where the lines represent matchings. To compute a minimum number of moves that are needed, it suffices to find some *largest order preserving subsequence*. (It does not have to be unique.) Here such a sequence is $v1, v3, v4$ that matches $w4, w5, w6$ while preserving the order. Then we need only to add move operations for the other nodes, here $v6, v8$. This gives us an optimal number of moves.

We use in the implementation, for performance reasons, a different way of computing the local moves. It does not guarantee minimality but is faster. Our approach considers each child of the node in the new document (in the example each child of w) in the order of the document. It assumes it is not moved if it does not violate the order decided so far. So, in the example, $w1$ is assumed not to have been moved, and similarly for $w2$. Then a move is needed for $v1$ (it was before $v8$) and similarly for $v2$ and $v3$. In this case, the solution we implemented would yield one more move than the optimal.

Tuning Parts of the algorithm require tuning. First, we have to select an appropriate definition for weight. A better weight would improve the accuracy of matches, and so both the quality and speed of the algorithm. For the moment we use the following definition of weight: The weight of a text node is $1 + log(length(text))$ and the weight of an element node is $1+sum(weight(children))$. Also, it is not easy to choose how far we can go up in matching ancestor nodes in the hierarchy. A too small distance would result in missing matches whereas a too large one may generate erroneous matches.

Other XML features We briefly mention here two other specific aspects of XML that have impacts on the *diff*, namely attributes and DTDs.

First, consider attributes. Attributes in XML are different from element nodes in some aspects. First, a node may have at most one attribute of label ℓ for a given ℓ. Also, the ordering for attributes is irrelevant. For these reasons, we do not provide identifiers to attributes, i.e., a particular attribute node is identified by the identifier of its parent and its label. Thus we have specific update operations for attributes. When two elements are matched between two consecutive versions, the attributes with the same label are automatically matched. Attributes may also serve in identifying their parents. In particular, the matching of two ID attributes (XML identifiers) is an excellent indication that the nodes owning these attributes should be matched as well. (Indeed, from a user viewpoint, such identifiers carry more meaning that system-generated IDs.) We do not yet use this information, but we are planning to introduce it in the near future. It is rather easy to modify our algorithm to take such IDs into account.

Now consider DTDs, a most important property of XML documents that allows to type them. We have considered using this information to improve our algorithm. For instance, it may seem useful to use the information that an element of label ℓ has at most one child of label ℓ' to perform matching propagation. This turns out not to help much because we can obtain this information at little cost on the document itself, even when the DTD does not specify it. The DTD or XMLSchema information of types can also be obtained using node's label in the document. On the other hand, the DTD (or a data guide in absence of DTD) is an excellent structure to record statistical information. It would therefore be a useful tool to introduce learning features in the algorithm, e.g. learn that a price node is more likely to change than a description node.

4.3 Complexity analysis

We discuss next the data structures used in the algorithm. We then study briefly an upper bound and an average cost for the algorithm. Then we determine an upper bound for the cost of the algorithm. We use the C++ Standard Template Library *map* and *vector* data structures. The tree representation of the XML documents is obtained using *Xerces-C++* library [10].

In the first phase, both documents are read, and a signature is computed for every subtree. Computing these signatures is in linear time, and uses no more stack than the depth of the document. Storing the results, does not cost more in terms of space or time than the size of both documents.

In the second phase, which role is to find matching, the worst-case scenario is to have to consider every single node, e.g. this happens when none of them is matched. In this case, every node is placed into the priority queue that is implemented with an ordered heap with an inserting cost of $log(n)$. This results in a total upper bound of $n * log(n)$, where n is the number of nodes in $D2$. The memory usage is linear in the size of the new document. The function that gets the best candidate for a node needs no more time than $O(c * d)$ where c is the number of candidates, and d the depth of the document. It does not use any extra space. So the overall cost in time is $O(n * (log(n) + c * d))$. The space usage is only linear is the size of the data, as we only use the stack for local variables.

The third phase works by considering some specific optimization possibilities on each node, here considered as a parent. According to our lazy down propagation strategy, we avoid costly tests. For instance, finding unique labels in a list of children has a time and space cost linear in the number of children. Checking the presence of similar labels in the parent's match list of children is also linear in time and space. A node can only appear once in a list of children, so the total time and space cost is linear in the number of nodes of the document.

The fourth phase is a bit more costly. Finding nodes that have been deleted or inserted only requires to test if nodes of both documents have been matched. So it is linear in time and space. But in order to detect nodes that have been moved within the same parent node, we need to keep track of every node's position. For every element node, we keep a summary of all children that have been deleted (or moved to another element node), inserted (or moved from another element node), and we construct a list of the remaining nodes and their position.

As mentioned above, we could obtain the minimal set of move operations that transform a sequence of children into another by computing a longest common subsequence of both lists [19]. These algorithms have typically a time cost of $O(s^2/log(s))$, where s is the number of children, and a space cost of $O(s^2)$. However, in practical applications, using the simple heuristic of going from first child to the last proved to be sufficient, and the time and space cost is no more than $O(s)$. The cost is linear in the number of children, so the total is linear in the number of nodes in the document.

So the overall worst-case cost is $O(n * (log(n) + c * d))$ where n is the size of the document, c the maximum number of candidates that is considered and d the depth of the document. The memory usage is linear in the total size of both documents. Note that we use the size of the document (in bytes) as an upper bound to the number of nodes.

It turns out that, in practice, the number of candidate c is small for most nodes, especially given that we start with the largest subtrees. Considering that for most well structured documents, the average depth is no more than $log(n)$, we may expect to have an average cost of $O(n * log(n))$ where n is the total size of both documents.

5 State of the art

The basis of edit distances and minimum edit script is the string edit problem [6, 13, 11, 1]. Insertion and deletion correspond to inserting and deleting a symbol in the string, each operation being associated with a cost. Now 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 prefixes substrings of x and y up to the i-th symbol, and constructing a directed acyclic graph (DAG) in which path cost is evaluated by the minimal cost of these three possibilities:

$$\begin{array}{l} cost(x[1..i] \rightarrow y[1..j]) = min: \\ cost(delete(x[i])) + cost(x[1..i-1] \rightarrow y[1..j]) \\ cost(insert(y[j])) + cost(x[1..i] \rightarrow y[1..j-1]) \\ cost(subst(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1]) \end{array}$$

Note that for example $subst(x[i], y[j])$ is zero when the symbols are equals. The space and time complexity are $O(|x| * |y|)$.

Detecting changes in an XML document, i.e., a string, can be compared to string pattern matching. Using several steps of string pattern matching algorithm on an XML document allows to detect all changes in the document, and it is possible to take into account the structure of the document by adding unique identifiers to the XML structure delimiters of both documents. But then the cost of understanding the structure of the document is added to the cost of detecting the changes, whereas the structure of the XML document is already known and we should use it to improve the algorithm efficiency. So it is preferable to consider specific algorithms for tree pattern matching.

Kuo-Chung Tai [26] gave a definition of the edit distance between ordered labeled tree and the first non-exponential algorithm to compute it. The insert and delete operations are similar to the operations on strings: deleting a node means making its children become children of the node's parent. Inserting is the complement of deleting. The difficulty is when nodes are involved in substitution, and the resulting algorithm has a complexity of $O(|D1| * |D2| * depth(D1)^2 * depth(D2)^2)$ in time and space.

Lu's algorithm [18] uses another edit based distance. The idea is, when a node in subtree $D1$ matches with a node in subtree $D2$, to use the string edit algorithm to match their respective children.

In Selkow's variant [24], insertion and deletion are restricted to the leaves of the tree. Thus, applying Lu's algorithm in the case of Selkow's variant results in a time complexity of $O(|D1| * |D2|)$. Depending on the considered tree data, this definition may be more accurate. It is used for example, in Yang's [32] algorithm to find the syntactic differences between two programs. Due to XML structure, it is clear that the definition is also accurate for XML documents. An XML Document structure may be defined by a DTD, so inserting and deleting a node and changing its children level would change the document's structure and may not be possible. However, inserting and deleting leaves or subtrees happens quite often, because it corresponds to adding or removing objects descriptions, e.g. like adding or removing people in an address book.

Recently, Sun released an XML specific tool named *DiffMK* [20] 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 tree structure of XML.

We do not consider here the unordered tree problem [33], nor the tree alignment [15].

Perhaps the closest in spirit to our algorithm is *LaDiff* [9, 8]. It is also designed for XML documents. It introduces a matching criteria to compare nodes, and the overall matching between both versions of the document is decided on this base. The faster version of the matching algorithm uses longest common subsequence computations for every element node starting from the leaves of the document. 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. e is the sum of the number of deleted and inserted subtrees, and the total size of subtrees that moved for the shortest edit script.

Then a minimum cost edit script conforming to the given matching is constructed in a cost of $O(n * d)$ where n is the total number of nodes, and d the total number of children moving within the same parent.

Like other algorithms that compute a minimal edit script (for some given criteria), the worst case cost, obtained here considering that large subtrees have moved, is quadratic in the size of the data.

Our algorithm is in the spirit of Selkow's variant, and ressembles Lu's algorithm. The differences come from the use of the structure of XML documents. In Lu's algorithm, once a node is matched, we try to match its children using the string algorithm. For this, children are identified using their label. But this would not apply in practice on XML documents, as many nodes may have the same label. So we use a signature computed over the children's subtree. But then, children may not be matched only because of a slight difference in their subtree, so we had to extend the algorithm by taking into consideration those children and their subtree and matching part of it if possible.

Using this edit definition, we could add the support of move operations. Note that a move operation can be seen as the succession of a deletion and an insertion. However it is different in that we consider the cost of *move* to be much less than the sum of deleting and inserting the subtree. Thus it is clear that previous algorithm wouldn't compute the minimal edit script as we defined it.

Last but not least, our algorithm goal is slightly different from previous algorithms in that for performance reasons, we do not necessarily want to compute the very minimal edit script.

The reasons why our algorithm does not obtain the minimal result are as follows. First, as we saw, it does not guarantee a minimum reordering of children for the same parent. Also, nodes matching, and so ancestor matching, are ordered depending on node's weight. It may occur that a single child has the highest weight and is (wrongly) matched first, whereas the sum of the weights of its siblings nodes would be more than its own weight.

6 Experiments

In this section we present an experimental study of the algorithm. We show that it achieves its goals, in that it runs in linear time, and computes good quality deltas. (The linear space bound is obvious and will not be discussed.) We first present result on some synthesized data (changes simulated on XML documents). We then briefly consider changes observed on the web.

6.1 Measures on simulated changes

Here we present the results that are related to the algorithm itself. The goal is to confirm the complexity of the algorithm by studying its behavior in different situations, and, as it does not guarantee an optimal result, to analyze the quality of its result.

In order to be able to make a large number of changes for which we control the interesting parameters, we use a change simulator that we describe next. Then we verify the time cost of the algorithm, which turns out to be linear in the size of data. In the last part we study the quality achieved by the algorithm in various contexts.

Change simulator We present next our change simulator that allows to generate changes on some input XML document. Its role and design is very important as any artifact or deviation in the change simulator may eventually have consequences in the test set. We tried to keep the architecture of the change simulator very simple. The change simulator reads an XML document, and stores its nodes in arrays. Then, based on some parameters (probabilities for each change operations) the four types of simulated operations are created in three phases:

delete Given a delete probability, we delete some nodes. As nodes may be roots of rather large subtrees, the probability of a deletion is divided by the number of nodes in the subtree.

update The remaining text nodes can be updated, based on their update probability. Note that the new text we use is always original.

insert/move We choose random nodes in the remaining element nodes and insert a child to them, depending on the insert and move probability. The type of the child node (element or text) has to be chosen according to the type of its siblings, e.g. we can't insert a text node next to another text node, or else both data will be merged in the parsing of the resulting document. So according to the type of node inserted, and the move probability we do either insert data that had been deleted, e.g. that corresponds to a move, or we insert "original" data. For original data, we try to match to the XML style of the document. If the required type is text, we can just insert any original text using counters. But if the required node has to be a tag, we try to copy the tag from one of its siblings, or cousin, or ascendant; this is important for XML document in order to preserve the distribution of labels which is, as we have seen, one of the specificities of XML trees.

Note that because we focused on the structure of data, all probabilities are given *per node*. A slightly different model would be obtained if it was given *per byte of data*. Note also that because the number of nodes after the first phase is less than the original number of nodes of the document, we recompute update and insert probabilities to compensate.

The result of the change simulator is both a delta representing the exact changes that occurred, which will be useful to compare later with the algorithmically computed delta, and a new version of the document. It is not easy to determine whether the change simulator is good or not. But based on statistical knowledge of changes that occurs in the real web (see next) we will be able to improve its quality. We tried to verify both by human evaluation of resulting documents and by the control of measurable parameters (e.g. size, number of element nodes, size of text nodes, ...) that the change simulator behaves properly. The change simulator we used here was the result of a few iterations. It seems now to conform reasonably to human expectations.

Performance We verify next that the complexity is no more than the expected $O(|D| * depth(D))$ time. To do that, we use our change simulator to create arbitrary sized data and measure the time needed to compute the *diff* algorithm. In the experiment we discuss next, the change simulator was set to generate a fair amount of changes in the document, the probabilities for each node to be modified, deleted or have a child subtree inserted, or be moved were set to 10 percent. Measures have been conducted many times, and using different original XML documents.

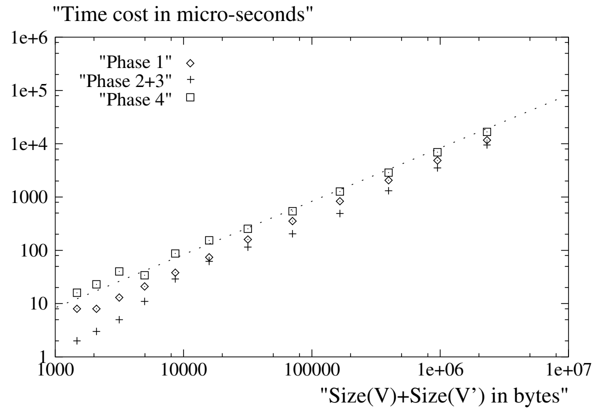

Figure 4: Time cost for the different phases

The results (see Figure 4) show clearly that the algorithm's cost is almost linear in time.

More precisely, the first phase, consisting of reading both documents, parsing and constructing the DOM tree (see [10]), and registering the largest subtrees is clearly linear, which was expected due to its very simple execution path. The last phase, consisting of creating the delta document using the computed matching is also linear, which comes from the fact that we use a number of modifications proportional to the size of the document.

Phases 2 + 3, the core of the *diff* algorithm, are clearly the fastest part of the whole process. The progression is also linear. It may seem a bit different but that comes from the fact that the text nodes we insert are on average smaller than text nodes in the original document.

Quality We analyze next the quality of the *diff* in various situations, e.g. if the document has almost not changed, or if the document changed a lot. We paid particular attention to move operations, because detecting move operations is a main contribution of the algorithm.

Using our change simulator, we generated different amounts of changes in the document, including a high proportion of move operations. In Figure 5, we compare the size of the delta obtained by using our algorithm to the size of the original delta created by the change simulator. The delta that has been applied to the original document is in that sense, perfect. Delta's sizes are expressed in bytes. The original document size is 37744 bytes, which is about twice the average size of an XML document on the web.

The experiment shows that the delta produced by *diff* is about the size of the delta produced by the simulator. This is the case even when there are many updates including

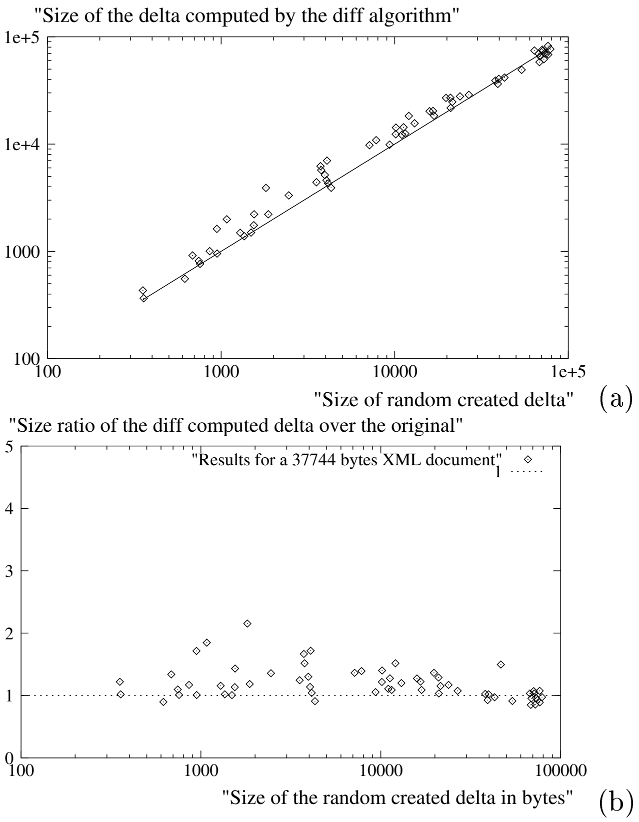

Figure 5: Quality of Diff

many move operations.

For an average number of changes, when about ten to twenty percent of nodes are modified, the delta computed by the *diff* algorithm is about fifty percent larger. This is precisely due to the large number of move operations that modify the structure of the document. But when the change rate increases further, the delta gains in efficiency again, and is even sometimes more accurate than the original delta, in that it finds ways to compress the set of changes generated by the simulator.

Note that the efficiency lost in the middle of the range is very acceptable, because (i) the corresponding change rate is already more than what is generally found on real web documents; and (ii) the presence of many moves operations modifying the structure of the document is rare on real web documents.

6.2 Measures on real web data

We present here results obtained by running the algorithm over more than two thousands XML documents crawled on the web [21]. Unfortunately, few XML documents we found changed during the time-frame of the experiment. We believe that it comes from the fact that XML is still in its infancy and XML documents on the web are less likely to change than HTML documents. This is also due to the fact that the time-frame of the experiment was certainly too short. We are currently running a new longer term experiments.

We present here results obtained on about twenty XML documents that changed on a per-week basis. This sample is certainly too small for statistics, but its small size allowed a human analysis of the *diff* outputs. Since we do not have here a "perfect" delta as in the case of synthesized changes, we compare to Unix Diff.

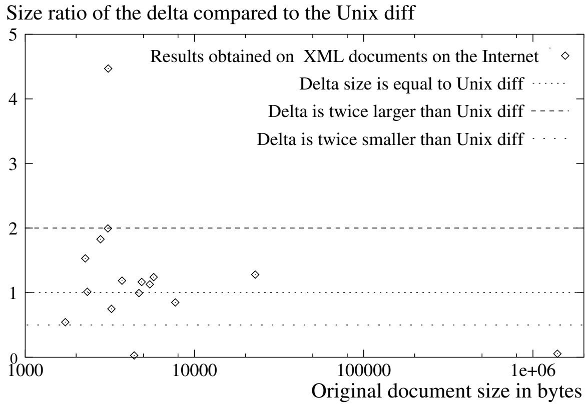

Figure 6: Delta over Unix Diff size ratio

The most remarkable property of the *deltas* is that they are on average roughly the size of the Unix Diff result (see Figure 6). The outputs of Unix Diff and of our algorithm are both sufficient to reconstruct one version from another, but *deltas* contain a lot of additional information about the structure of changes. It is interesting to note that the cost paid for that is very small in average.

It is also important to compare the delta size to the document's size, although this is very dependent on how much the document changed. Other experiments we conducted [19] showed that the delta size is usually less than the size of one version. In some cases, in particular for larger documents (e.g. more than 100 kilobytes), the delta size is less than 10 percent of the size of the document.

One reason for the delta to be significantly better in size compared to the Unix Diff is when it detects moves of big subtrees. In practice, this did not occur often. The Unix Diff uses *newline* as separator. In a few cases, the Unix Diff was particularly inefficient because the XML files used long text lines. The Unix Diff itself is never more than twice the size of the document. The worst case occurs when there is not a single line in common between the two versions of the document. Our worse case is only marginally worse since we also record structural information.

To conclude this section, we want to stress the fact that the test set was very small and that more experiments are clearly needed.

7 Conclusion

All the ideas described here have been implemented and tested. As we have seen by comparing the algorithm with existing tree pattern matching algorithms or standard *diff* algorithms, the use of XML specificities leads to significant improvements.

We already mentioned the need to gather more statistics about the size of deltas and in particular for real web data. To understand changes, we need to also gather statistics on change frequency, patterns of changes in a document, in a web site, etc. Many issues may be further investigated. We mentioned improvements to the *diff* using key attributes and DTD. Other aspects of the actual implementation could be improved for a different trade-off in quality over performance, e.g. using longest common subsequence algorithms to find the minimal set of move operations for siblings nodes.

Acknowledgments We would like to thank M. Preda, F. Llirbat and L.Mignet for discussions on the topic.

References

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

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, October 1999.

[3] M.E. Adiba and B.G Lindsay. Database snapshots. In *VLDB*, October 1980.

[4] V. Aguiléra, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML Documents in Xyleme. In *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

[5] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64, 2000.

[6] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxfort University Press, 1997.

[7] S. Chawathe. Diff on trees.
http://www.cs.umd.edu/ chaw/stdiff/.

[8] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[9] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):493–, 1996.

[10] Initial code base derived from IBM's XML4C Version 2.0. Xerces c++. http://xml.apache.org/xerces-c/index.html.

[11] Sankoff D. and J. B. Kruskal. Time warps, string edits, and macromolecules, 1983. Addison-Wesley, Reading, Mass.

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql/.

[13] Levenshtein V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory 10*, pages 707–710, 1966.

[14] Oasis, ICE resources, http://www.oasis-open.org/cover/ice.html.

[15] L. Wang Jiang, T. and K.Zhang. Alignement of tree - an alternative to tree edit, 1994.

[16] Kinecta, http://www.kinecta.com/products.html.

[17] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the International Conference on Very Large Databases*, Bombay, India, September 1996.

[18] S. Lu. A tree-to-tree distance and its application to cluster analysis, 1979.

[19] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an xml warehouse, 2001. Technical Report.

[20] Sun Microsystems. Making all the difference. http://www.sun.com/xml/developers/diffmk/.

[21] L. Mignet, M. Preda, S. Abiteboul, S. Ailleret, B. Amann, and A. Marian. Acquiring XML pages for a WebHouse. In *proceedings of Base de Données Avancées conference*, 2000.

[22] B. Nguyen, S. Abiteboul, G. Cobena, and L. Mignet. Query subscription in an xml webhouse. In *SIGMOD*, 2001.

[23] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). http://www.w3.org/TandS/QL/QL98.

[24] S. M. Selkow. The tree-to-tree editing problem, 1977.

[25] Concurrent Versions System. Cvs. http://www.cvshome.org.

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

[27] W3C. EXtensible Markup Language (xml) 1.0. http://www.w3.org/TR/REC-xml.

[28] J.T.L. Wang, K. Zhang, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.

[29] N. Webber, C. O'Connell, B. Hunt, R. Levine, L.Popkin, and G. Larose. The Information and Content Exchange (ICE) Protocol. http://www.w3.org/TR/NOTE-ice.

[30] Xyleme. `www.xyleme.com`.

[31] Tak W. Yan and Hector Garcia-Molina. The SIFT Information Dissemination System. In *TODS 24(4): 529-565*, 1999.

[32] W. Yang. Identifying syntactic differences between two programs, 1991.

[33] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees, 1992.