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

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 dynamic warehouses capable of storing massive volume of XML data. 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". 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. It also uses XML specific information such as ID attributes. 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. Since this problem is NP-hard, the linear time is obtained by trading some quality. We present experiments (again on synthetic data) that show that the output of our algorithm is reasonably close to the "optimal" in terms of quality. Finally we present experiments on a small sample of XML pages found on the Web.*

1 Introduction

Users are often not only interested in the current value of data but also in changes. Therefore, there has been a lot of work around *diff* algorithm for all kinds of data. With the Web and standards such as HTML and XML, tree data is becoming extremely popular which explains a renewed interest for computing changes in tree-structured data. A particularity of the Web is the huge volume of data that has to be processed. For instance, in the Xyleme project [31], we were lead to compute the *diff* between the millions of documents loaded each day and previous versions of these documents (when available). This motivates the study of an extremely efficient, in terms of speed and memory space, *diff* algorithm for tree data.

As mentioned above, the precise context for the present work is the Xyleme project [32] that is studying and building a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing massive volume of XML data. XML, the new standard for semistructured data exchange over the Internet [28, 2], allows to support better quality services and in particular allows for *real* query languages [11, 23] and facilitates semantic data integration. In such a system, managing 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 most critical component of change control in Xyleme is the *diff* module that needs to be extremely efficient. This is because the system permanently receives XML data from the Web (or internal) crawlers. New versions of the documents have to be compared to old ones without slowing down the whole system.

Observe that the *diff* we describe here is for 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, the result of *diff* for a true XML document is semantically much more informative than for HTML. It includes pieces of information such as the insertion of a new product in a catalog.

Intuitively, our 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. Our algorithm also takes advantage of the specificities of XML data. For instance, it knows of attributes and attribute updates and treat them differently from element or text nodes. It also takes into account ID attributes to match elements. The matching of nodes between the old and new version is the first role of our algorithm. Compared to existing *diff* solutions [13, 20], our algorithm is faster and has significantly better matchings.

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

The other role of our algorithm is a representation of the changes using a *delta*. We use the *delta* representation of [19] that is based on inserts, deletes, updates and moves. For completeness, we present it in Section 4. Given a matching of nodes between the two documents, a *delta* describes a representation of changes from the first to the second. A difficulty occurs when children of a node are permuted. It is computationally costly to find the *minimum* set of move operations to order them.

We show first that our algorithm is "correct" in that it finds a set of changes that is sufficient to transform the old version into the new version of the XML document. In other words, it misses no changes. Our algorithm runs in $O(n * log(n))$ time vs. quadratic time for previous algorithms. Indeed, it is also noticeable that the running time of our algorithm significantly decreases when documents have few changes or when specific XML features like ID attributes are used. In Section 3, we recall that the problem is NP-hard. Therefore, to obtain these performance we have to trade-in something, an ounce of "quality". The delta's we obtain are not "minimal". In particular, we may miss the best match and some sets of move operations may not be optimal. It should be observed that any notion of minimality is somewhat artificial since it has to rely on some arbitrary choice of a distance measure. We present experiments that show that the *delta*'s we obtain are of very good quality.

There has been a lot of work on *diff* algorithms for strings, e.g., [12, 10, 1], for relational data, e.g., [17], or even for tree data, e.g., [29, 7]. 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. Like the rest of the system, the *diff* and the versioning system are implemented in C++, under Linux, with Corba for communications. Some of the programs described in this paper are available for download at [8].

We performed tests to validate our choices. We briefly present some experimentation. The results show that the complexity of our algorithm is indeed that determined by the analysis, i.e., quasi linear time. We also evaluate experimentally the quality of the *diff*. For that, we ran it on synthetic data. As we shall see, 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). 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 also used the *diff* to analyze changes in portions of the web of interest, e.g., web sites described as XML documents (Section 6).

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 4. In Section 5, we present our *diff* algorithm, and its analysis. We compare it to previous *diff* algorithms in Section 3. 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. Most of these motivations for changes detection and management are similar to those described in [19].

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. Detecting changes in such an environment serves many purposes:

Versions and Querying the past: [19] One 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. Later, one might want to ask a query about the past, e.g., ask for the value of some element at some previous time, and to query changes, e.g., ask for the list of items recently introduced in a catalog. Since the *diff* output is stored as an XML document, namely a *delta*, such queries are regular queries over documents.

Learning about changes: The *diff* constructs 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 [30, 14, 16]. Also, different users may modify the same XML document off-line, and later want to synchronize their respective versions. The diff algorithm could be used to detect and describe the modifications in order to detect conflicts and solve some of them [26].

Monitoring changes: We implemented a subscription system [22] that allows to detect changes of interest in XML documents, e.g., that a new product has been added to a catalog. To do that, at the time we obtain a new version of some data, we *diff* it and verify if some of the changes that have been detected are relevant to subscriptions. Related work on subscription systems that use filtering tools for information dissemination have been presented in [33, 4].

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 store structural information for every indexed word of the document [3]. We are considering the possibility to use the *diff* to maintain such indexes.

To offer these services, the *diff* 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 received (or crawled from the web), it is installed in the repository. It is then 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 possibly 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 of the web crawlers loads millions of Web or internal pages per day. Among those, we expect many to 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).

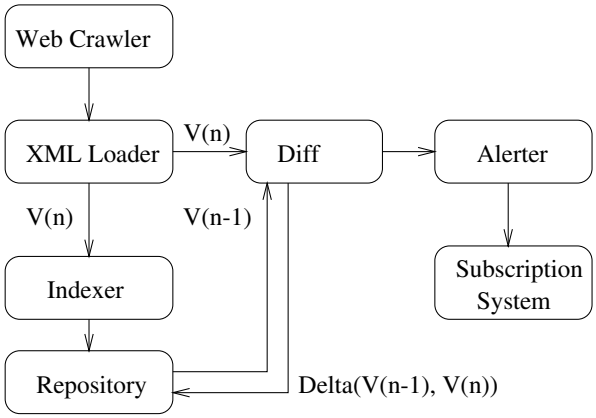

Figure 1. Xyleme-Change architecture

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* must be correct, in that it constructs a *delta* corresponding to these requirements, and it 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 one 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 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 our 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 [6, 7], tailored to tree data. It also takes advantage of specificities of XML such as ID attributes defined in the DTD, or the existence of labels.
- The *diff* has insert/delete/update operations as in other tree *diff* such as [7], and it supports a *move* operation as in [6]. Our *move* operation is tailored to large subtrees of data.

3 State of the art

In a standard way, the *diff* tries to find a minimum *edit script* between the versions at time t_{i-1} and t_i. The basis of edit distances and minimum edit script is the string edit problem [5, 12, 10, 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 $cost(x[1..i] \rightarrow y[1..j])$ is evaluated by the minimal cost of these three possibilities:

$$cost(delete(x[i])) + cost(x[1..i-1] \rightarrow y[1..j])$$
$$cost(insert(y[j])) + cost(x[1..i] \rightarrow y[1..j-1])$$
$$cost(subst(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1])$$

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 our algorithm's efficiency. So it is preferable to consider specific algorithms for tree pattern matching.

Kuo-Chung Tai [27] 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 considering two documents $D1$ and $D2$, 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 algo-

rithm 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 [34] 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 [35, 25] nor the tree alignment [15] problems.

Perhaps the closest in spirit to our algorithm is *LaDiff* or *MH-Diff* [7, 6]. 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 an 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 most other algorithms, the worst case cost, obtained here considering that large subtrees have moved, is quadratic in the size of the data.

The main reason why few *diff* algorithm supporting *move* operations have been developed earlier is that most formulations of the tree diff problem are NP-hard [36, 6] (by reduction from the 'exact cover by three-sets'). *MH-Diff*, presented in [6] provides an efficient heuristic solution based on transforming the problem to the edge cover problem, with a worst case cost in in $O(n^2 * log(n))$.

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 our 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 "perfect" result are as follows. First, we may miss the best match. Also some sets of move operations may not be optimal.

4 Brief overview of the change representation model

In this section, we present some aspects of the change model [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, namely XIDs, 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. Persistant identifiers can be used to easily track parts of an XML document through time. 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. As previously reported, 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 doc-

ument, 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]. We also define the XID-map, a string attached to a subtree that describes the XIDs of its nodes.

Representing changes The delta is a *set* of the following elementary operations: (i) the deletion of subtrees; (ii) the insertion of subtrees; (iii) an update of the value of a text node or an attribute; and (iv) a move of a node or a part of a subtree. 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 work. Not only is it necessary in an XML context to deal with permutations of the children of a node (a frequently occurring situation) but also to handle more general moves as well. Moves are important to detect from a semantic viewpoint. 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>
```

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, shown in Section 5, is to manage positions. 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>
```

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 our algorithm or in its tuning may result in different deltas, and so different interpretations of the changes that happened between two versions.

5 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 work is given in Section 3. 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 our algorithm.

5.1 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 (e.g. a hash value) for every subtree of the old document $D1$. If ID attributes are defined in the DTD, we will match corresponding nodes according to their value, and propagate these matching in a simple bottom-up and top-down pass.

Then we consider every subtree in $D2$, starting from the largest, and try to find whether it is identical to some of the

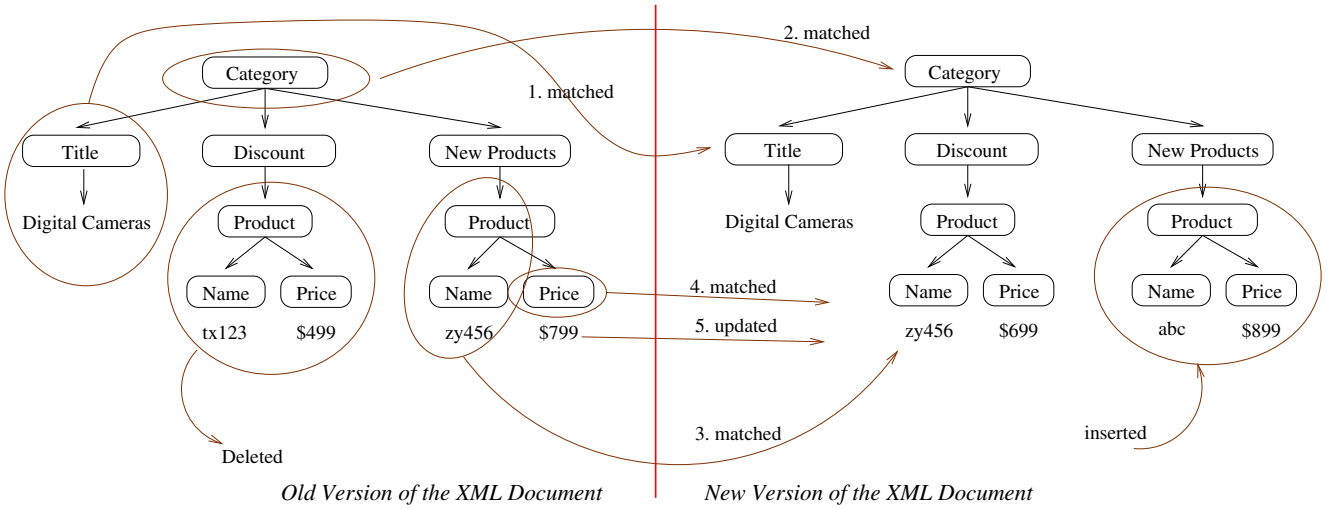

Figure 2. Matching subtrees

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 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 (lazy down). 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 work 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 subtrees, more than one subtrees of $D1$ are identical to it. 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 among siblings 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. Aspects on this bottom-up and top-down simple pass are considered in Section 5.3.

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 sub-

tree of node *Category* with this label, so we match it.

Once no more matchings can be obtained, unmatched nodes in $D2$ (resp. $D1$) correspond to inserted (resp. 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.

5.2 Detailed description

The various phases of our algorithm are detailed next.

Phase 1 (Use ID attributes information): In one traversal of each tree, we register nodes that are uniquely identified by an ID attribute defined in the DTD of the documents. The existence of ID attribute for a given node provides a unique condition to match the node: its matching must have the same ID value. If such a pair of nodes is found in the other document, they are matched. Other nodes with ID attributes can not be matched, even during the next phases. Then, a simple bottom-up and top-down propagation pass is applied. Note that if ID attributes are frequently used in the documents, most of the matching decision have been done during this phase.

Phase 2 (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 a hash value computed using the node's content, and its children signatures. Thus it uniquely represents the content of the entire subtree rooted at that node. A weight is computed simultaneously for each node. It is the size of the content for text nodes and the sum of the weights of 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 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 3 (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.

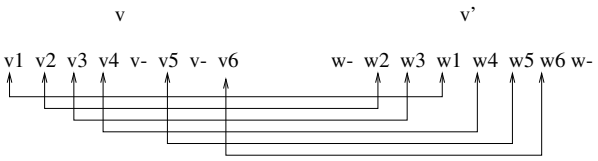

Figure 3. Local moves

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 4 (Optimization: Use structure to propagate matchings): We traverse the tree bottom-up and then 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. The main issue of this part is to avoid expensive computations, so specific choices are explained in Section 5.3.

Phase 5 (Compute the delta): This last phase can itself be split in 3 steps:

1. Inserts/Deletes/Updates: Find all unmatched nodes in the old/new document, mark them as *deleted/inserted*; record the effect of their deletion/insertion to the position of their siblings. If a text node is matched but its content has changed, we will mark it as *updated*.

2. Moves: Find all nodes that are matched but with non matching parents. These correspond to *moves*. 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 a (not necessarily unique) *largest order preserving subsequence*. Here such a sequence is $v2, v3, v4, v5, v6$ that matches $w2, w3, w4, w5, w6$ while preserving the order. Then we need only to add move operations for the other pair of nodes, here $(v1, w1)$. We also use a more general

definition and algorithm where the cost of a move corresponds to the weight of the node. This gives us an optimal set of moves.

Finding the largest order preserving subsequence is expensive for large sequences. Thus, for performance reasons, we use a heuristic which does not guarantee optimality, but is faster and proves to be sufficient in practice. It is used when the number of children is too large, and it works by cutting it into smaller subsequences with a maximum length (e.g. 50). We apply on them the longest common subsequence algorithms(see Section 5.3), and merge the resulting subsequences. The subsequence obtained is clearly a common subsequence of the two original lists of children.

In our example, by cutting both lists in two parts, we would find subsequences $(v2, w2), (v3, w3)$ and $(v5, w5), (v6, w6)$, and thus we miss $(v4, w4)$ compared to the optimal solution.

Tuning The details of our algorithm require some choices that we describe next. We also consider the tuning of some parameters of the system.

First, we have to select an appropriate definition for weight. The choice of a weight has impact on accuracy of matches, and therefore both on the quality and speed of the algorithm. We will see in Section 5.3, that the weight of an element node must be no less than the sum of its children. It should also grow in $O(n)$ where n is the size of the document. We use $1 + sum(weight(children))$. For text nodes (i.e. leaves), we consider that when the text is large (e.g. a long description), it should have more weight than a simple word. We use $1 + log(length(text))$ as a measure.

Also, when matching two subtrees, it is not easy to choose how far to 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 (e.g. matching many ascendants because two insignificant text are identical). For complexity reasons, we show in Section 5.3 that the corresponding depth value must stay in $O(log(n) * \frac{W}{W_0})$ where W is the weight of the corresponding subtree, and W_0 the weight for the whole document. Note that for a given subtree with weight W, the upper bound decreases when W_0 grows to infinity. In 5.3 we explain how to use indexes to handle the case when the distance d would go to zero. More precisely, we use $d = 1 + \frac{W}{W_0}$.

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 persistent identifiers to attributes, i.e., a particular attribute node is identified by the persistent identifier of its parent and its label (so in our representation of *delta*, we use specific update operations for attributes). When two elements are matched between two consecutive versions, the attributes with the same label are automatically matched. We also use ID attributes (XML identifiers) to know if the nodes owning these attributes should (or can't) be matched as well.

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. Such reasoning is costly because it involves the DTD and 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. On the other hand, the DTD or XMLSchema (or a data guide in absence of DTD) is an excellent structure to record statistical information. It is therefore 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.

5.3 Complexity analysis

In this part we determine an upper bound for the cost of our algorithm. For space reasons, we do not present the algorithmic of the different functions here.

Note that the number of nodes is always smaller than n where n is the size of both document files. Reading both documents, computing the hash value for signatures, and registering ID attributes in a hash table is linear in time and space. The simple bottom-up and top-down pass -used in the first and fourth phase- works by considering some specific optimization possibilities on each node. These passes are designed to avoid costly tests. They focus on a fixed set of features that have a constant time and space cost for each (child) node, so that their overall cost is linear in time and space:

1. propagate to parent: Consider that node i is not matched. If it has a children c matched to some node c' we will match i to the parent i' of c'. If i has many matched children $c1, c2, ...$, then there are many possibilities for i'. So we will prefer the parent i' of the larger (weight) set of children $c'1, c'2, ...$ The computation is done in postfix order with a tree traversal.

2. propagate to children: If a node is matched, and both it and its matching have a unique children with a given label, then these two children will be matched. Again the cost is no more than of a tree traversal.

During the BULD algorithm, the worst-case occurs when no node is matched. In this case, every node is placed into the priority queue, with an inserting cost of $log(n)$ (ordered heap). This results in a total upper bound of $n * log(n)$. The memory usage is linear in the size of the documents.

For every node, a call is made to the function that finds the best candidate (the set of candidates is obtained using a hash table created in the first phase). The general definition of this function would be to enumerate all candidates and choose the best one as the one with the closest ascendant. It works by enumerating candidates and testing the ascendant up to a given depth. Thus the time cost is in $O(c * d)$ where c is the number of candidates, and d the maximum path length allowed for ancestor's look-up. As previously described, we make d depend on the weight W of the subtree. Thanks to the first rules defined in previous section, and because identical subtrees can not overlap, c is smaller than W_0/W where W_0 is the weight for the subtree representing the whole document. The second rule states that $d = O(log(n) * W/W_0)$. So the cost of a function call is in $O(log(n))$. The overall cost is then in $O(n * log(n))$. But the upper limit for d means that when the document's size increases and W_0 goes to infinity, d goes to zero. To achieve this result, it is not possible to evaluate all candidates. The issue occurs when there are multiple occurances of a short text node in a large document, e.g. the product manufacturer for every product in a catalog. Our solution to support $d = 1 + \frac{W}{W_0}$ is to use a secondary index (a hash table) which is created during initialization, and gives access by their parent's identifier to all candidate nodes for a given signature. Thus we can find (if it exists) the first candidate with a matching parent in constant time. This generic solution works for any lower bound of d by using as many indexes to access nodes by their grand-parent or ascendant identifier.

The second part consists of constructing the *delta* using the matchings obtained previously. Finding nodes that have been deleted or inserted only requires to test if nodes of both documents have been matched. It is also clear that a node has moved if its parent and the parent of its matching do not match. So this first step is linear in time and space. The difficulty comes with nodes that stay within the same parent. If their order has changed, it means that some of them have 'moved'. As mentioned above, to obtain the optimal *delta*, we should apply a 'longest common subsequence' algorithm on this sequence of children [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, applying this algorithm on a fixed-length set of children (e.g. 50), and merging the obtained subsequences, provides excellent results and has a time and space cost in $O(s)$. We choosed this heuristic, so the total cost for the document is then in $O(n)$.

So the overall worst-case cost is $O(n * (log(n))$ where n is the size of the document files (including the DTD, if any, that we also have to read). The memory usage is linear in the total size of both documents.

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. Due to space limitations only a small portion will be presented here. However, they illustrate reasonably well what we learned from the experiments.

6.1 Measures on simulated changes

Our measures show that our algorithm is very fast, almost linear in the size of data. Also, since it does not guarantee an optimal result, we analyze the quality of its result and show experimentally that it is excellent. For these experiments, we needed large test sets. More precisely, we needed to be able to control the changes on a document based on parameters of interest such as deletion rate. To do that, we built a change simulator that we describe next.

Change simulator The change simulator allows to generate changes on some input XML document. Its 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 and its entire subtree.

[update] The remaining text nodes are then updated (with original text data) based on their update probability.

[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 our expectations.

Performance We verify next that the complexity is no more than the expected $O(n * log(n))$ 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 each. Measures have been conducted many times, and using different original XML documents.

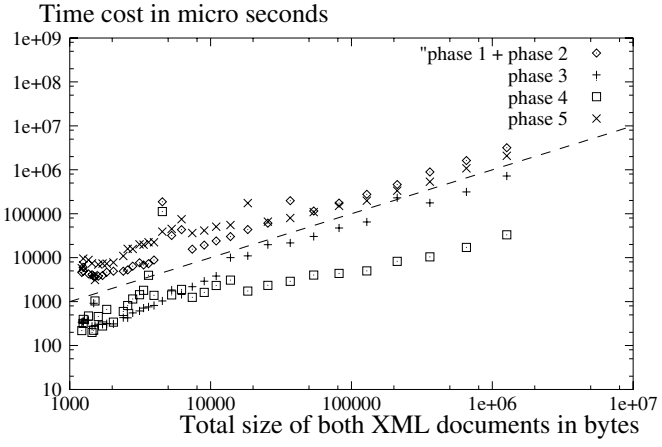

Figure 4. Time cost for the different phases

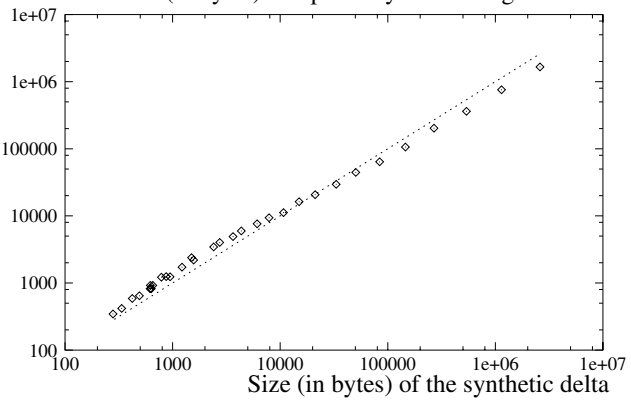

Figure 5. Quality of Diff

The results (see Figure 4) show clearly that our algorithm's cost is almost linear in time. [1] We have analyzed precisely the time spent in every function, but due to lack of space we do not provide full details here. Phases $3 + 4$, the core of the *diff* algorithm, are clearly the fastest part of the whole process. Indeed, most of the time is spent in parts that manipulate the XML data structure: (i) in phase 1 and 2, we parse the file [9] and hash its content; (ii) in phase 5, we manipulate the DOM tree [9]. The progression is also linear. The graph 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.

A fair and extensive comparison with other *diff* programs would require a lot more work and more space to be presented. An in-depth comparison, would have to take into account speed, but also, quality of the result ("optimality"), nature of the result (e.g., moves or not). Also, the comparison of execution time may be biased by many factors such as the implementation language, the XML parser that is used, etc. Different algorithms may perform differently depending on the amount and nature of changes that occured in the document. For example, our *diff* is typically excellent for few changes.

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 our algorithm.

Using our change simulator, we generated different amounts of changes for a sequence of documents, including a high proportion of move operations. In Figure 5, we compare the size of the delta obtained by using our algorithm to

[1] A few values are dispersed because of the limitations of our profiling tool.

the size of the original delta created by the change simulator. The delta obtained by the simulator captures the edit script of operations that has been applied to the original document to change it, and, in that sense, it can be viewed as *perfect*. Delta's sizes are expressed in bytes. The original document size varies from a few hundred bytes, to a megabyte. The average size of an XML document on the web is about twenty kilobytes. The points in Figure 5 are obtained by varying the parameters of the change. Experiments with different documents presented the same patterns.

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 many move operations. For an average number of changes, when about thirty 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 much 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 mention next results obtained by running our algorithm over more than ten 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 longer term set of experiments.

We present here results obtained on about two hundred 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. Our test sample also contains about two hundred large XML documents representing metadata about web sites. We also applied the *diff* on a few large XML files (about five megabytes each) representing metadata about the entire INRIA web site.

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

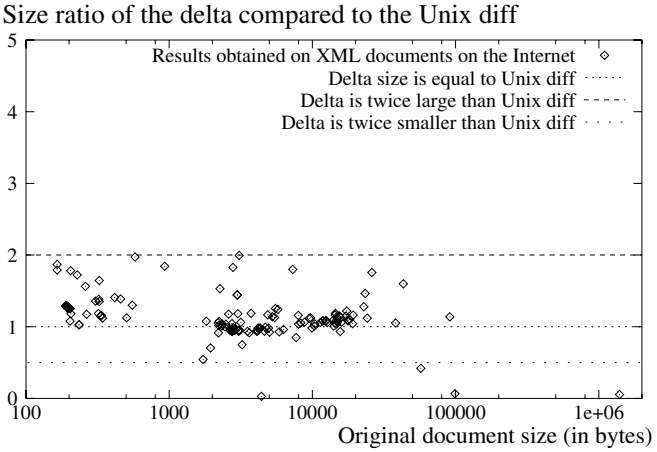

Figure 6. Delta over Unix Diff size ratio

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 extra information 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, but in practice, this does not occur often. A drawback of the Unix Diff is that it uses *newline* as separator, and some XML document may contain very long lines. The worst case size for the Unix Diff output is twice the size of the document. Our worse case is marginally worse due to the storage overhead for structural information.

We have also tested our *diff* on XML documents describing portions of the web, e.g., web sites. We implemented a tool that represents a snapshot of a portion of the web as a set of XML documents. Given two such snapshots, our *diff* computes what has changed in the time interval. For instance, using the site www.inria.fr that is about fourteen thousands pages, the XML document is about five million bytes. Given the two XML snapshots of the site, the *diff* computes the delta in about thirty seconds. Note that the core of our algorithm is running for less than two seconds whereas the rest of the time is used to read and write the XML data. The *delta*'s we obtain for this particular site are typically of size one millions bytes. 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. A recent version of our *diff* program can be downloaded at [8]. We showed by comparing our algorithm with existing tree pattern matching algorithms or standard *diff* algorithms, that 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. For example we can extend our use of DTDs to XMLSchema. Other aspects of the actual implementation could be improved for a different trade-off in quality over performance, e.g. we could investigate the benefits of intentionally missing *move* operations for children that stay with the same parent. We are also extending the *diff* to observe changes between websites compared to changes to pages here.

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

References

[1] W. 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] 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.

[4] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, pages 53–64, 2000.

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

[6] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, pages 26–37, Tuscon, Arizona, May 1997.

[7] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 25(2):493–504, 1996.

[8] G. Cobena, S. Abiteboul, and A. Marian. Xydiff, tools for detecting changes in XML documents. http://www-rocq.inria.fr/~cobena/XyDiffWeb/.

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

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

[11] 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/.

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

[13] IBM. XML treediff. www.alphaworks.ibm.com/.

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

[15] L. W. Jiang T. and K. Zhang. Alignement of trees - an alternative to tree edit. *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching, pp.75-86*, 1994.

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

[17] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB*, Bombay, India, Sept. 1996.

[18] S. Lu. A tree-to-tree distance and its application to cluster analysis. *in IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 2*, 1979.

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

[20] S. 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 M. Preda. Monitoring XML data on the web. 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. *Information Processing Letters, 6*, pages 184–186, 1977.

[25] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms, 11*, pages 581–621, 1990.

[26] C. V. System. Cvs. http://www.cvshome.org.

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

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

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

[30] 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.

[31] Xyleme Project. `www-rocq.inria.fr/verso/`.

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

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

[34] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience, 21, (7)*, pages 739–755, 1991.

[35] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Proceedings Letters 42*, pages 133–139, 1992.

[36] K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.