diffX: An Algorithm to Detect Changes in Multi-Version XML Documents

Raihan Al-Ekram, Archana Adma and Olga Baysal
School of Computer Science, University of Waterloo
{rekram@swag, aadma@cs, obaysal@cs}.uwaterloo.ca

Abstract

This paper presents the **diffX** algorithm for detecting changes between two versions of an XML document. The identified changes are reported as a script of edit operations. The script, when applied to the first version of the XML document, will produce the second version. The goal is to optimize the runtime of mapping the nodes between the two versions and to minimize the size of the edit script. To achieve this goal an isolated tree fragment mapping technique is used, in order to iteratively identify the largest matching tree fragments between the tree representations, of the two versions of the document. The mapping technique is robust enough to handle differences in both the structure and the content of the two trees. The generated edit script from the mapping acknowledges the different order sensitiveness of element and attributes of XML data model. The primitives for the edit script comprise both the atomic (node) and non-atomic (subtree) edit operations natural to XML document modification. The runtime of the algorithm is O(n^2).

1 Introduction

The Internet is a huge repository of documents that are being modified constantly. With the invention and the increasing use of XML as a standard document format for web publishing, detection and management of changes in multi-version XML documents has become an active area of research.

XML has been widely adapted to store and exchange data in academia and in industry. Detection of changes between the two versions of a document is an important function in many applications. A user may visit certain XML documents (web sites) repeatedly and is interested to know how each document has changed since the last visit. For example, a stock trader will be interested to know how the market is changing continuously every moment. The stock price can be changed; there can be new quotes available or previous quotes sold out. Being able to identify changes between multiple versions of XML documents, effectively and efficiently, will invoke research for version management systems, better storage and retrieval mechanisms to avoid duplicate information on the web.

The Unix `diff` utility [19] is probably the most popular change detection tool to-date. It is based on the Longest Common Subsequence matching algorithm [6] between two strings and works on flat files. There has been significant amount of work done on differencing algorithms for strings [1, 5, 6, 9] and some work on relational data in data warehouses [12]. But these algorithms cannot be trivially generalized to handle XML data, since they do not understand the structure and the context indicated by the tags in XML data [11]. The data model underlying an XML document is essentially a tree structure [19], the nodes of the tree being labeled and typed. Accordingly, the change detection techniques for XML [15, 17] and XML-like structured or semi-structured documents [11, 13] are based on the tree distance algorithms [2, 3, 4, 7, 8, 14].

This paper presents **diffX**, an algorithm for identifying changes between two versions of an XML document, which are reported as a script of edit operations. The script, when applied to the first version of the XML document, will produce the second version. We analyze the performance and quality of the generated edit script in comparison with the existing algorithms like X-Diff [17] and XyDiff [15].

The remainder of the paper is organized as follows: Section 2 provides a brief review of related and existing algorithms for XML differencing. We discuss three classes of algorithms: tree distance algorithms, differencing algorithms for XML-like semi-structured data and change detection algorithms for XML documents. We investigate their individual pros and cons that motivated the development of the **diffX** algorithm. In Section 3, the **diffX** approach is explained with the basic algorithm and an example. Section 4 provides some enhancements on the basic algorithm. Section 5 gives an analysis of the **diffX** algorithm in terms of runtime and space complexity. Section 6 presents possible future directions in the area of change management systems. Finally, Section 7 concludes the paper with references in Section 8.

2 Related Work

Many researchers have studied the general problem of detecting changes between documents. Most of it was focused on computing differences between flat files. The famous GNU `diff` utility uses Longest Common Subsequence algorithm to compare two plain text files. `CVS` [22] another GNU utility, uses `diff` to detect differences between different versions of programs and stores the deltas. Chawathe et al. [11] pointed out that the techniques employed by these two programs cannot be generalized to handle structured data because they do not understand the hierarchical structure information contained in such data sets. Typically, hierarchical structured data, e.g. SGML and XML, place tags on each data segment to indicate context. Standard plain-text change detection tools have problems matching data segments between two versions of data.

2.1 Tree Distance Algorithms

The tree distance algorithms laid the foundation for change detection in structured documents and XML [3, 4, 8, 10]. A good example of them is the Valiente's algorithm [14], for detecting distance between two tree structures based on the number of edit operations needed to transform one tree into another. This algorithm is based on bottom-up mapping of the rooted trees given by the largest common forest between them. In order to identify the largest common forest between trees T_1 and T_2 a compact acyclic directed graph G is computed from the forest consisting of the disjoint union of T_1 and T_2. The graph G partitions T_1 and T_2 into isomorphism equivalence classes. A mapping M of nodes from T_1 to T_2 is obtained from the resulting correspondence between the nodes of T_1 and T_2 to the nodes of G by collecting an unmapped isomorphic subtree in T_2 for each unmapped subtree of T_1 during a pre-order traversal of T_1. This algorithm is the fastest among all tree distance algorithms with a runtime of only O(n), where $n = n_1+n_2$, the number of nodes in T_1 is n_1 and T_2 is n_2. But like any other bottom-up mapping algorithm, it will give a distance measure far from optimum, if most of the differences between the two trees are in the leaf nodes. Hence, it is not a good candidate for differencing XML documents.

Tai presented a tree-to-tree correction algorithm [3] by defining the ordered mapping between trees. This algorithm gives the best possible matching between two trees but with a runtime of $O(n^2h^4)$, where $n = \max(n_1,n_2)$, $h = \max(h_1,h_2)$, h_1 = height of T_1 and h_2 = height of T_2.

Selkow's tree-to-tree editing algorithm [2] is a restrictive variation of Tai's algorithm. It uses a top-down mapping approach, and the matching is performed only on the nodes of the same level giving a runtime complexity of $O(n_1n_2) \approx O(n^2)$.

Lu's tree-to-tree distance algorithm [4] solves the problem for trees of no more than two levels with a runtime of $O(n_1n_2) \approx O(n^2)$.

In another similar study on trees [8, 10], Shasha and Zhang propose a dynamic programming based edit distance algorithm to improve upon Tai's runtime to $O(n^2h^2)$. The difference is that the tree traversal is performed in post-order instead of pre-order.

The tree distance algorithms are based on atomic edit operations on nodes not considering subtree level operations, e.g. subtree deletes or moves, which are natural in case of XML documents.

2.2 Structured Document Differencing Algorithms

Change detection was also studied for LaTex and nested-object documents in the LaDiff [11] and the MH-Diff [13] algorithms.

The LaDiff algorithm takes two versions of a LaTex document as input and produces another

LaTex document, with the changes marked as the output. The change detection problem is split into two sub problems. Good matching problem and minimum confirming edit script problem. The former finds a matching between the two objects of the two versions and the later computes a minimum cost edit script. The run time of LaDiff is O(ne+e^2), where e is the weighted edit distance.

The MH-Diff algorithm on the other hand is based on transforming the change detection problem to a problem of computing the minimum cost edge cover of a bi-partite graph. The MH-Diff process consists of constructing an induced graph from the input trees, pruning the induced graph, finding a minimum cost edge cover of the pruned induced graph and finally using this edge cover to obtain an edit script. It compares the structure and content of the nodes to see if they are related. This algorithm identifies changes between unordered structured documents in O(n^2) time. The approach of generating an edit script is based on the weighted matching problem and does not yield a good result in the presence of node duplications in the input trees.

2.3 XML Differencing Algorithms

XMLTreeDiff [18] computes the difference between two XML documents based on heuristic similarity measures. It computes the hash values for the nodes of both the documents using DOM-Hash [23] and then reduces the size of the two trees by removing identical subtrees. With Zhang and Shasha's algorithm it then generates the difference between the two simplified trees.

The XyDiff algorithm was proposed by Cobena et al. [15] for change detection in XML documents. This algorithm detects changes in ordered trees using a bottom-up mapping technique and then propagating the mapping throughout the trees. XyDiff starts with matching the nodes by matching their ID attributes. Next, the algorithm computes a signature and a weight for each node of both trees in a bottom-up traversal. Signature is a hash value computed using the node content and its children’s signatures. Weight is the sum of the sizes of all the text nodes below current node. So the root of the two trees will end up with the largest weights. Based on the priority of the weight of the nodes, the signatures of the nodes between the two trees are matched, in order to find the heaviest subtree match. When XyDiff finds the match, it propagates the match bottom-up to their ancestors in one pass and then top-down from matching ancestors to their unmatched descendants in another pass. If there is more than one candidate for matching, XyDiff uses simple heuristic rules to select one of them in order to avoid full evaluation of the candidates. The process repeats for the next heaviest subtree match. The main focus of XyDiff is the speed rather than the minimality of the generated edit operations. It utilizes XML specific information, e.g. unique identifiers and smart heuristics that give a runtime as good as O(n log n). But the main drawback of XyDiff is that the bottom-up mapping technique can completely fail if all the leaves are modified or some structural change isolates the top and bottom part of the old version in the new one.

The X-Diff algorithm [17] was proposed by Wang et al. to achieve minimum edit script for unordered XML documents. X-Diff uses a complete top-down mapping mechanism relying on node signature, the path from the root to the given node, when matching the nodes. The X-Diff algorithm includes three main phases: parsing and hashing, matching and computing edit script. The two XML documents are first parsed into trees and a hash value is computed for each node in both the trees. The hash values of root nodes of the trees are then compared. If these hash values are equal then the two trees are equivalent, otherwise a minimum-cost matching is performed between the two trees. Matching is done only on nodes with same signatures. Based on the matching the edit script is computed. X-Diff has a polynomial running time of O(n^2 d log d), where d is the maximum degree of any node in the trees. X-Diff is best suited for detecting changes between two versions of XMLized relational data sets. Where the structure of the XML document will remain unchanged and only the contents at the bottom of the tree will be modified, which is often not the case with semi-structured XML documents.

3 The Basic diffX Algorithm

Consider the tree representation of an XML document containing the item catalog of a hypothetical store as given in Figure 1. The store carries sci-fi related books and movies and the catalog currently contains two books titled `Foundation` and `A Space Odyssey` and a DVD titled `Star Wars Trilogy`.

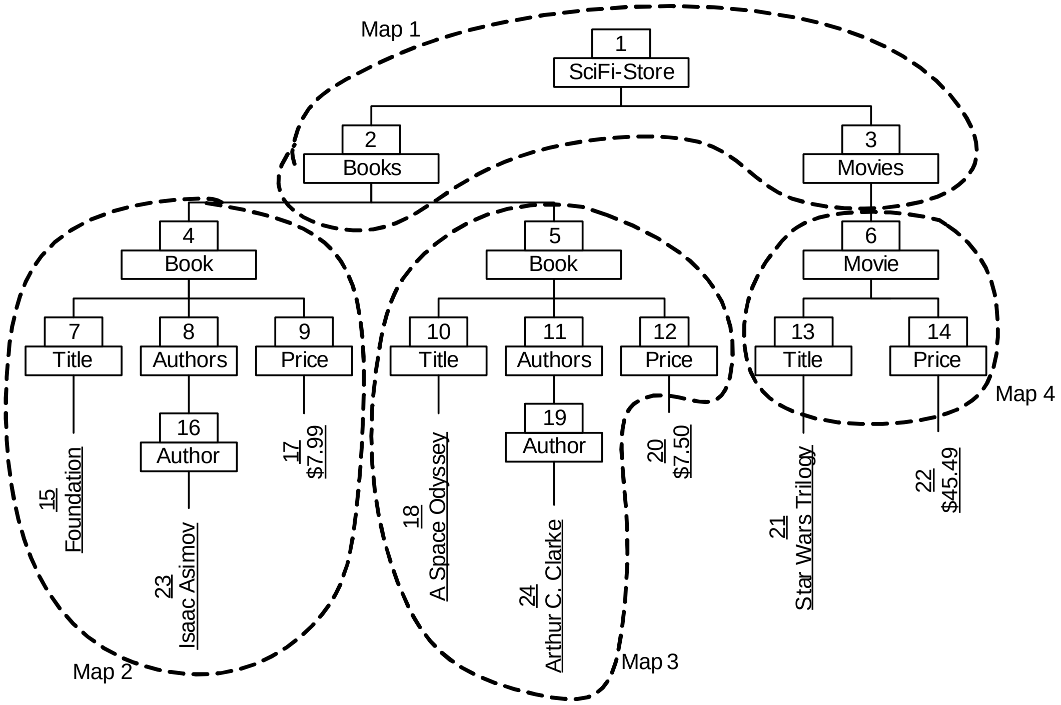

Figure 1: Sci-Fi Store Catalog Version 1

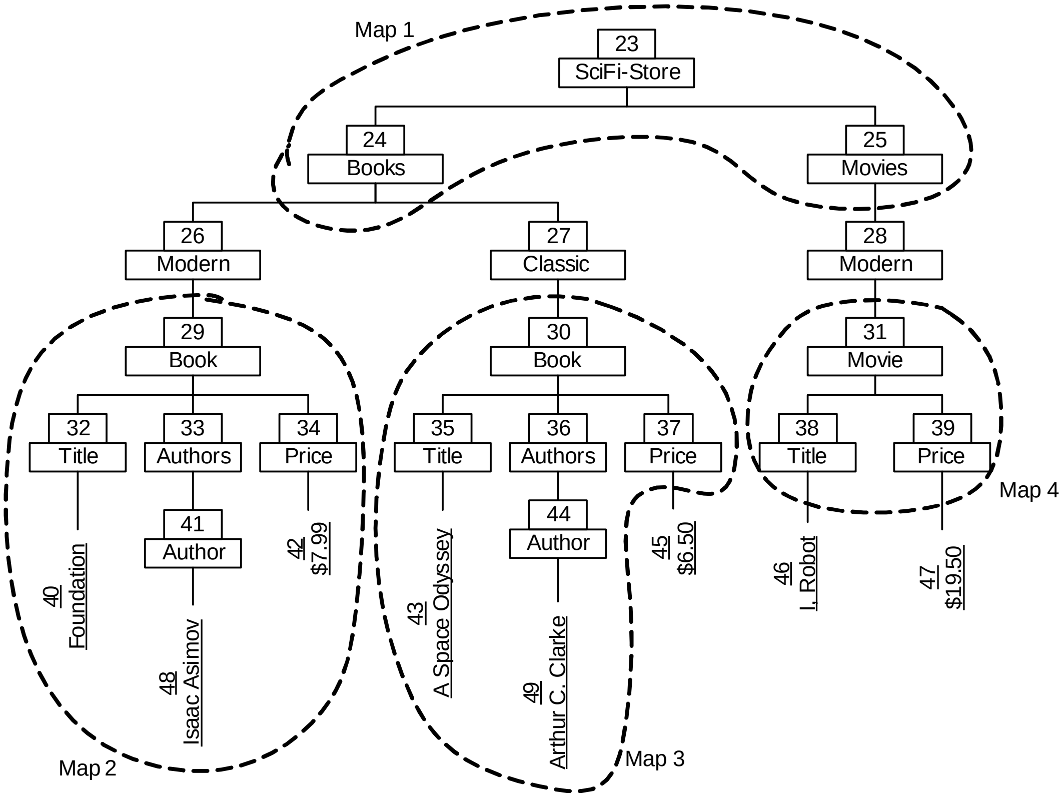

Figure 2: Sci-Fi Store Catalog Version 2

Figure 2 gives the second version of the catalog, where both the structure and the content of the catalog have changed. The books and the movies are now classified under either `Modern` or `Classic`, which pertains to a structural change. Also the price of the book `A Space Odyssey` is updated, the DVD `Star Wars Trilogy` is no more available and there is a new DVD titled `I, Robot`. These are the content changes in the document.

A pure top-down mapping like X-Diff will stop after matching the first two levels of the tree and report the rest of it as unmatched. On the other hand a bottom-up mapping like XyDiff will report the DVDs as unmatched although the structure of the DVD remained the same. It will also miss the top two levels of the tree that remained unchanged. The proposed **diffX** algorithm performs an isolated tree fragment mapping to identify the largest matching fragments between the two trees starting from the root of the older tree and repeats the process until all the nodes of the tree are checked for a match. The goal is to ensure a maximum matching in presence of changes in both the structure and the content of the document.

3.1 Data Model

The internal data model for **diffX** algorithm is the tree model of XML documents. The nodes of the tree are labeled and typed. The three main types of nodes are `element`, `attribute` and `text`. An element has a name and may consist of a number of ordered sub-elements, a number of unordered attributes and a single text node as its children. An attribute is a leaf node of the tree and has a name and a value for it. A text node is also a leaf and has only a value. The root of the tree, the document node, is a specialized element node with no attributes. The labels of an element, an attribute and a text node consist of their name, name and value pair and a value respectively. Two nodes from the same tree or from two different trees are considered equal if both their types and labels are equal. A node identifier uniquely identifies each node across both the trees. Each node knows its parent, its children by their position and its own position among the siblings.

In Figures 1 and 2, the nodes with horizontal texts are elements and vertical texts are text nodes. For simplicity attribute nodes, which can be treated similarly to the text nodes are omitted. The number on each node is the unique identifier for the node. It can be any arbitrary unique number or text but we used the position of the node in a level-order traversal for the first tree and continued with the numbering for the second tree without restarting. The text on each node gives the label for it.

3.2 Isolated Tree Fragment Mapping

The algorithm starts with identifying largest isolated tree fragment mapping, which is an iterative top-down mapping technique. Let T_1 and T_2 be the tree representation of the two XML documents. The number of nodes in T_1 is n_1 and T_2 is n_2. The mapping from T_1 to T_2 is given by a set M of ordered pairs (x, y), where x is a node of T_1 and y is a node of T_2. M is an isolated tree fragment mapping, if for all x_1, y_1, x_2, y_2 a fragment of T_1 rooted at x_1 matches with a fragment of T_2 rooted at x_2 and a fragment of T_1 rooted at y_1 matches with a fragment of T_2 rooted at y_2; then the there will be no overlapping between the matching tree fragments rooted at x_1 and y_1 and also at x_2 and y_2.

The algorithm works as follows: In the top-down level-order traversal of T_1, find all the nodes in T_2 equal to the current node of T_1. For each matching node in T_2, recursively check for matching children until there is no more match or encountered a node that was matched previously. The node of T_2 that gives the largest matching tree fragment is added to the mapping M along with all the matched children. In case of a tie, priority can be given to nodes with matching parents and then the node with lower identifier value. Repeat the process for all the unmatched nodes in T_1. Algorithm 1 presents the complete algorithm.

If the algorithm is run on the example given in Figure 1 and 2, there is only one node, node 23, in T_2 that matches node 1 of T_1. Now if we try to propagate the match to their children, we find two more matches, thus getting a tree fragment mapping of size 3: (1,23), (2,24) and (3,25), as shown by *Map* 1 in the figures. The next unmatched node of T_1 is node 4 that gives two matches in T_2, node 29 and 30. Propagating the match to the children gives a match of tree fragment size 8 for node 29 and a tree fragment size of 5 for node 30. The largest match is taken and a mapping of (4,29), (7,32), (8,33), (9,34), (15,40), (16,41), (17,42) and (23,48), as indicated by *Map* 2 is recorded. From the rest of the tree we get two more tree fragment mappings of size 7 and 3: (5,30), (10,35), (11,36), (12,37), (18,43), (19,44), (24,49) and (6,31), (13,38), (14,39) in *Map* 3 and *Map* 4 respectively.

```
Procedure Mapping ( $T_1$, $T_2$, $M$ )
    Input $T_1$, $T_2$ : tree
    Output $M$ : map
Begin
    index the nodes of $T_2$
    traverse $T_1$ in a level-order sequence
        let $x$ be the current node
        if $(x, \_) \in M$ then
            skip current node
        end-if
        let $y[]$ = all nodes from $T_2$ equal to $x$
        $M'' = \varnothing$
        for $i = 1$ to size of $y[]$
            if $(\_, y[i]) \notin M$ then
                $M' = \varnothing$
                Match-Fragment ( $x$, $y[i]$, $M$, $M'$ )
                if size of $M'$ > size of $M''$ then
                    let $M'' = M'$
                end-if
            end-if
        end-for
        let $M = M \cup M''$
    end-traverse
End
```

```
Procedure Match-Fragment ( $x$, $y$, $M$, $M'$ )
    Input $x$, $y$ : node ; $M$ : map
    Output: $M'$ : map
Begin
    if $(x, \_) \notin$ M and $(\_, y) \notin$ M and node($x$) = node($y$) then
        let $M' = M' \cup \{(x, y)\}$
        for $i = 1$ to minimum of number of children between $x$ and $y$
            Match-Fragment ( $i$-th child of $x$, $i$-th child of $y$, $M$, $M'$ )
        end-for
    end-if
End
```

Algorithm 1: Isolated Tree Fragment Mapping

[Note: The _ (underscore) is a wildcard in the pair (*x*, _)]

```
Procedure Generate-Script ( $T_1$, $T_2$, $M$ )
    Input $T_1$, $T_2$ : tree ; $M$ : map
    Output $S$ : list
Begin
    traverse $T_2$ in a level-order sequence
        let $y$ be the current node
        if ( _, $y$ ) $\notin M$ then
            add to $S$ "insert ( $y$, $parent(y)$, $position(y)$ )"
        else
            retrieve $x$ such that $(x, y) \in M$
            if ( $parent(x)$, $parent(y)$ ) $\notin M$ then
                add to $S$ "move ( $x$, $parent(y)$, $position(y)$ )"
            else-if $position(x) \neq position(y)$ then
                add to $S$ "move ( $x$, $parent(y)$, $position(y)$ )"
            end-if
        end-if
    end-traverse
    traverse $T_1$ in a level-order sequence
        let $x$ be the current node
        if ( $x$, _ ) $\notin M$ then
            add to $S$ "delete ( x )"
        end-if
    end-traverse
End
```

Algorithm 2: Edit Script Generation

```
insert(26,24,1) ⇒ insert(Element(Modern),/ScFi-Store/Books[1],1)
insert(27,24,2) ⇒ insert(Element(Classic),/ScFi-Store/Books[1],2)
insert(28,25,1) ⇒ insert(Element(Modern),/ScFi-Store/Movies[1],1)
move(4,26,1) ⇒ move(/ScFi-Store/Books[1]/Book[1],/ScFi-Store/Books[1]/Modern[1],1)
move(5,27,1) ⇒ move(/ScFi-Store/Books[1]/Book[2],/ScFi-Store/Books[1]/Classic[1],1)
move(6,28,1) ⇒ move(/ScFi-Store/Movies[1]/Movie[1],/ScFi-Store/Movies[1]/Modern[1],1)
insert(45,37,1) ⇒ insert(Text($6.50),/ScFi-Store/Books[1]/Classic[1]/Book[1]/Price[1],1)
insert(46,38,1) ( insert(Text(I,Robot),/ScFi-Store/Movies[1]/Modern[1]/Movie[1]/Title[1],1)
insert(47,39,1) ( insert(Text($19.50),/ScFi-Store/Movies[1]/Modern[1]/Movie[1]/Price[1],1)
delete(20) ( delete(/ScFi-Store/Books[1]/Classic[1]/Book[1]/Price[1]/Text[2])
delete(21) ( delete(/ScFi-Store/Movies[1]/Modern[1]/Movie[1]/Title[1]/Text[2])
delete(22) ( delete(/ScFi-Store/Movies[1]/Modern[1]/Movie[1]/Price[1]/Text[2])
```

Figure 3: The Generated Edit Script

Table 1: Comparison of the Diff Algorithms

Algorithm	**Data Model**	**Mapping**	**Edit Operation**				**Run Time**
			I	D	R	M	
Valiente	Ordered Unordered	Bottom-up	√	√	√	X	O(n)
Tai	Ordered	Ordered	√	√	√	X	O($n^2 h^4$)
SZ	Ordered	Ordered	√	√	√	X	O($n^2 h^2$)
LaDiff	Ordered	Bottom-up	√	√	√	√√	O(ne+e^2)
MH-Diff	Unordered	Induced graph	√	√	√	√√	O(n^2)
XyDiff	Ordered	Bottom-up Lazy-down	√√	√√	√	√	O(nh)
X-Diff	Unordered	Top-down	√√	√√	√	X	O(n^2d log d)
diffX	Ordered element Unordered attribute	Bottom-up Iterative-down	√	√√	√	√√	O(n^2)

I – Insert, **D** – Delete, **R** – Replace, **M** – Move
√ **-** Node, √√ - Node and Subtree, X – Not available
n – number of nodes, h – height, d – maximum degree, e – edit distance

3.3 Edit Script Generation

The next step is the generation of a sequence of edit operations from the mapping, which when applied to T_1 will produce T_2. The general idea is to delete all the unmatched nodes from T_1, insert all the unmatched nodes in T_2 and move all the matching nodes with unmatched parents or unmatched position to the position in T_2. So the primitive operations for the edit script are:

- `delete` (*x*): Delete the sub-tree rooted at node *x*
- `insert` (*x*, *y*, *p*): Insert node *x* as *p*-th child of node *y*
- `move` (*x*, *y*, *p*): Move the subtree rooted at node *x* as the *p*-th child of node *y*

The algorithm works as follows. In the top-down level-order traversal of T_2, if the current node is unmatched add an insert operation for it in the script. If the current node in T_2 matches a node in T_1 with unmatched parents or with unmatched position add a move operation for it in the script. The level-order traversal ensures that the parent of a node is in the right place before inserting or moving a node as its child. After processing all the matching nodes, the unmatched nodes in T_1 will be located at the bottom of the tree. Finally in a top-down level-order traversal of T_1 adding a delete operation for all the nodes rooted at an unmatched node will complete the edit script. Algorithm 2 presents the algorithm.

In the internal data model, node position is used in level-order traversal as a unique identifier for each node. But in XML technology the standard way of identifying nodes in a document is using XPath [21] expressions. Hence in the generated edit script XPath expressions are used to identify the existing nodes and node constructors to create new nodes. Applying the algorithm on our example gives the edit script in Figure 3.

4 Enhancements on the Basic Algorithm

Following are some enhancements on the basic algorithm to improve the mapping efficiency and quality of the generated edit script.

4.1 Valiente's Mapping

Valiente's [14] bottom-up mapping algorithm identifies the unchanged subtrees between two trees in O(n) time. Applying Valiente's algorithm on the XML documents before applying the basic **diffX** can considerably reduce mapping space for **diffX**. The overall worst case complexity, analyzed in Section 5, still remains O(n^2), but the size of n can get significantly smaller in many cases.

4.2 Replace Operation

An update on any node of the document, e.g. the price change of the book `A Space Odyssey` in our example, is modeled as an `insert` followed by a `delete` operation in the basic algorithm. A new primitive `replace` can be introduced to reduce the size of the generated script.

- `replace` (*x*,*y*): Replace the node *x* with the node *y*

The `replace` operations can be identified by two unmatched text nodes with matching parents, two unmatched attribute nodes with same name and with matching parents and finally two unmatched element nodes with matching parents and position.

4.3 Attribute Ordering

The order of a node among its siblings is important in the XML data model, except for the attribute nodes. The basic algorithm recognizes the ordered model, even for the attribute nodes. In order to make the basic algorithm fully compliant with XML data model it is required to eliminate the `move` operations of attribute nodes within the same parents. It can be accomplished by simply skipping the `else-if` part in line 13 of the `Generate-Script` procedure if the current node is an attribute.

5 Complexity Analysis

In this section we analyze the time and space complexity of the proposed **diffX** algorithm.

5.1 Runtime

In Algorithm 1, procedure `Mapping`, line 5, indexing the nodes of T_2 takes an $O(n_2 \log n_2)$ amount of time, where the total number of nodes in T_2 is n_2. The `traverse` block in line 6 iterates for a total of the number of nodes in T_1, which is n_1. In line 11, searching T_2 using the index takes $O(\log n_2)$ time. In the worst case the search can return all the nodes in T_2.Therefore the `for` loop in line 13 may iterate n_2 times, each time making a call to `Match-Fragment` procedure in the worst case. In `Match-Fragment` procedure, the `for` loop in line 31 and the recursive call to `Match-Fragment` procedure in line 32 attempts to propagate the match between the two nodes downward the trees as far as possible. In the worst case it will cover all the n_1 nodes in T_1. In Algorithm 2 procedure `Generate-Script`, each of the tree is traversed once in a sequence in line 5 and 18, giving a runtime of $O(n_1 + n_2)$. Therefore, the total runtime complexity of the algorithm is:

$$O(n_2 \log n_2) + n_1 (O(\log n_2) + n_2 n_1) + O(n_1 + n_2)$$
$$= O(n_1 n_2 n_1)$$

There is one important observation that a node participating in a match in the `Match-Fragment` procedure is not going to be further considered for matching in the `Mapping` procedure because of line 8. Hence each node in T_1 only contributes to either of the n_1 factor in the O notation, not both. Which gives an amortized runtime complexity of

$$O(n_1 n_2) \approx O(n^2).$$

5.2 Space

The internal tree representations T_1 and T_2 for the two XML documents require a space of $O(n_1+n_2)$. The mapping table M in the worst case maps all the nodes of T_1 to some nodes in T_2 requiring a space of $O(n_1)$. `y[]` in the worst case can return all the nodes of T_2 requiring a space of $O(n_2)$. The temporary maps M' and M'' may also require a space of $O(n_1)$ each. Giving the worst case memory requirement of

$$O(n_1 + n_2) + O(n_1) + O(n_2) + O(n_1) + O(n_2)$$
$$= O(4n_1 + 2n_1) \approx O(n).$$

6 Comparative Analysis

Table 1 illustrates a comparative analysis of **diffX** with all other algorithms discussed in the related work section.

Comparing the algorithms in Table 1 we can say that the fastest algorithm so far is Valiente's algorithm that runs in linear time. But the bottom-up distance approach does not suit well with XML documents as the modifications in the XML documents are expected to be on the leaf nodes mostly. The XyDiff algorithm is also almost linear in runtime complexity. The mapping, XML specific heuristics and sub-tree move operation make it attractive for XML-differencing. But certain types of changes in the document will make XyDiff to produce an edit script that is far from optimal. Our **diffX** is not as fast as XyDiff but its runtime is better than X-Diff. Like XyDiff, **diffX** supports subtree move and delete operations. Also **diffX** has the advantage of being able to deal with both the structure and the content differences between two documents.

7 Future Work

So far this paper has presented an algorithmic analysis and demonstration of the **diffX** algorithm with an example. The immediate next step is to implement the algorithm as a tool, and compare the performance and quality of the generated script with X-Diff and XyDiff on large data sets. The issue of edit operations needs to be further investigated. We believe that more edit operations, e.g. copy, can make the edit script more compact, thus to improve the change composition.

The long-term goal is to develop an integrated change detection and management system for XML data warehouses, where XML documents are collected periodically by crawling the web. When a new version of an existing document is found, only the reverse delta between the old and new versions, and the new document are stored in the warehouse, purging the old version. Reverting to the old versions can be achieved by applying the reverse delta on the new version, while optimizing the storage requirement.

8 Conclusion

This paper has presented the outcome of the authors study on change detection techniques for XML documents. The authors have proposed an algorithm for effective detection and recording of changes in multi-version XML documents. The proposed algorithm **diffX** addresses the following issues:

- The algorithm honors ordered element and unordered attribute model of XML document.
- The mapping technique is robust enough to handle changes in both the structure and the content of an XML document.
- The identified differences are represented in terms of an edit script with both atomic and non-atomic primitives like insert, delete, replace and move operations.
- The runtime of the algorithm is quadratic in order.

Acknowledgements

The authors would like to thank Alejandro López-Ortiz for his feedback and advice throughout this research and his valuable comments on the draft of the paper. We also acknowledge Charlie Clarke for his helpful comments on the paper.

Archana Adma is supported by IBM Canada through the National Institute for Software Research and by Communications and Information Technology Ontario. Olga Baysal is supported by the Software Telecommunications Group through the Ontario Research and Development Challenge Fund (ORDCF).

About the Authors

Raihan Al-Ekram is a PhD student at the school of Computer Science in the University of Waterloo. He obtained his MASc degree from the Department of Electrical and Computer Engineering from the same institution in 2004. His research interests include software architecture, software evolution, software re-engineering, and program comprehension. He is a member of the Software Architecture group in the University of Waterloo.

Archana Adma obtained her BTech degree in Computers and Information Technology from Jawaharlal Nehru Technological University, Hyderabad, India. In January 2005 she started to pursue her Masters as a member of the Programming Languages Group, in the School of Computer Science at University of Waterloo, Canada. Her proposed research is in the field of Information Retrieval from Spoken Medical Discussions.

Olga Baysal received her BA degree in Computer Science and English Language from Vyatka State University, Russia in 2001. She began graduate study in School of Computer Science of University of Waterloo in January 2005, and is a member of Waterloo's Software Architecture Group.

References

[1] R. A. Wagner and M. J. Fischer. “The string-to-string correction problem”. *Journal of the Association for Computing Machinery*, 21: 168--173, January 1974.

[2] S. M. Selkow. “The tree-to-tree editing problem”. *Information Processing Letters*, 6: 184-186, 1977.

[3] K. C. Tai. “The tree-to-tree correction problem”. *Journal of the Association for Computing Machinery*, 26: 485-495, 1979.

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

[5] W. F. Tichy. “The string-to-string correction problem with block moves”. *ACM Transactions on Computer Systems*, 2: 309 -- 321, November 1984

[6] E. W. Myers. “An O(ND) difference algorithm and its variations”. *Algorithmica*, 1: 251-266, 1986

[7] K. Zhang and D. Shasha. “Simple fast algorithms for the editing distance between trees and related problems”. *SIAM Journal of Computing*, 18(6): 1245-1262, 1989.

[8] D. Shasha and K. Zhang. “Fast algorithms for the unit cost editing distance between trees”. *Journal of Algorithms*, 11: 581- 621, 1990.

[9] S. Wu, U. Manber, G. Myers and W. Miller. “An O(NP) sequence comparison algorithm”. *Information Processing Letters*, 35: 317-323, September 1990.

[10] K. Zhang, R. Statman and D. Shasha. “On the editing distance between unordered labeled trees”. *Information Processing Letters*, 42: 133-139, 1992.

[11] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom. “Change detection in hierarchically structured information”. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June 4-6, 1996.

[12] W. Labio and H. Gracia-Molina. “Efficient snapshot differential algorithms for data warehousing”. *Proceedings of the 22nd International Conference on Very Large Data Bases*, Mumbai, India, September 3-6, 1996.

[13] S. Chawathe and H. Garcia-Molina, “Meaningful change detection in structured data”. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 13-15, 1997.

[14] G. Valiente. “An efficient bottom-up distance between trees”. *Proceedings of the 8th International Symposium on String Processing and Information Retrieval*, Santiago, Chile, November 13-15, 2001.

[15] G. Cobena, S. Abiteboul and A. Marian. “Detecting changes in XML documents”. *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, USA, February 26 - March 1, 2002.

[16] Raymond K. Wong and Nicole Lam. “Managing and querying multi-version XML data with update logging”. *Proceedings of the 2002 ACM Symposium on Document Engineering*, McLean, Virginia, USA, November 8-9, 2002.

[17] Yuan Wang, David J. DeWitt and Jin-Yi Cai. “X-Diff: An effective change detection algorithm for XML documents”. *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March 5-8, 2003.

[18] F. P. Cubera, D. A. Epstein. “Fast Difference and Update of XML Documents”, *Xtech*, San Jose, March 1999.

[19] “Introduction to Diffutils”. *GNU Diffutils Page*. Viewed as of March 13, 2005. http://www.gnu.org/software/diffutils/diffutils.html

[20] “The XML Data model”. *W3C XML Page*, Viewed as of March 13, 2005 http://www.w3.org/XML/Datamodel.htm

[21] “XML Path Language”. *W3C XPath Page*, Viewed as of March 13, 2005 http://www.w3.org/TR/xpath

[22] “Concurrent Versions Systems”. *GNU CVS Page*. Viewed as of March 13, 2005. http://www.gnu.org/software/cvs

[23] H. Mayurama, K. Tamura and R. Uramoto, “Digest values for DOM (DOMHash) proposal”, *IBM Tokyo Research Laboratory*, http://www.trl.ibm.co.jp/projects/xml/domhash.htm, 1998.