# X-Diff: An Effective Change Detection Algorithm for XML Documents

Yuan Wang David J. DeWitt Jin-Yi Cai

University of Wisconsin – Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53705, USA
{yuanwang, dewitt, cjy}@cs.wisc.edu

## Abstract

Over the next several years XML is likely to replace HTML as the standard web publishing language and data transportation format. Since online information changes frequently, being able to quickly detect changes in XML documents is important to Internet query systems, search engines, and continuous query systems. Previous work in change detection on XML or other hierarchically structured documents used an ordered tree model, in which left-to-right order among siblings is important and it affects the change result. In this paper, we argue that an unordered model (only ancestor relationships are significant) is more suitable for most database applications. Using an unordered model, change detection is substantially harder than using the ordered model, but the change result that it generates is more accurate. We propose X-Diff, an effective algorithm that integrates key XML structure characteristics with standard tree-to-tree correction techniques. We also analyze the algorithm and compare it with XyDiff [CAM02], a published XML diff algorithm. We provide an experimental evaluation on both algorithms.

## 1. Introduction

"The eXtensible Markup Language (XML) is the universal format for structured documents and data on the Web," and it is likely to replace HTML as the standard format for publishing and transporting documents on the web [W3C]. Since online information changes frequently a tool is needed to detect such changes. In order to handle large volumes of changing documents this tool needs to work very efficiently. The following example illustrates the problem. Suppose a parent is interested in buying books for her children at an online auction site through a search engine that is equipped with such a tool. On the first visit she obtains a list of currently offered books and related information. Two hours later, the search engine retrieves updated data and uses the tool to figure out what has been changed during the past two hours. Part of the information received on the two visits is shown in Figures 1.1 and 1.2.

As a first step, the change-detection tool will determine whether or not the two versions are identical. If not, it next tries to match each book segment in the old version with every one in the new version to determine which books are still available, which have been sold, and which ones are new. In the example below, although the order of the two books has changed, both of them are still available. Next, for each book that is still available, the change-detection tool will determine what information has been modified. Based on the data in Figure 1.1 and 1.2, it should notify the consumer that there are two fewer hours to submit a bid for both books. The current bid price of the Harry Porter book is $10 by Mark whose rating is 125, and the current bid price of the Tom Sawyer book is $4.50 and the bidder has not changed.

Such a change-detection tool can be very useful to a query system in at least two ways,

- ***Incremental Query Evaluation*** When a user has a standing query against a time-varying data source, a change-detection tool can provide the query engine the delta data on which the query will be re-evaluated. Thus, the user doesn't receive old results and the query engine

```
<Books>
  <Book>
    <Title>Harry Potter and the Sorcerer's Stone</Title>
    <Author>J.K. Rowling</Author>
    <Seller>
      <ID>Mike</ID>
      <Rating>30</Rating>
    </Seller>
    <First_Bid>$5.00</First_Bid>
    <Current_Bid Time_Left = “36 hrs.”>$8.50</Current_Bid>
    <Bidder>
      <ID>Steve</ID>
      <Rating>25</Rating>
    </Bidder>
  </Book>
  <Book>
    <Title>The Adventures of Tom Sawyer</Title>
    <Author>Mark Twain</Author>
    <Seller>
      <ID>Sean</ID>
      <Rating>100</Rating>
    </Seller>
    <First_Bid>$2.00</First_Bid>
    <Current_Bid Time_Left = “4 hrs.”>$3.50</Current_Bid>
    <Bidder>
      <ID>Tim</ID>
      <Rating>5</Rating>
    </Bidder>
  </Book>
```

**Figure 1.1 A piece of auction data of old version**

```
<Books>
  <Book>
    <Title>The Adventures of Tom Sawyer</Title>
    <Author>Mark Twain</Author>
    <Seller>
      <ID>Sean</ID>
      <Rating>100</Rating>
    </Seller>
    <First_Bid>$2.00</First_Bid>
    <Current_Bid Time_Left = “2 hrs.”>$4.50</Current_Bid>
    <Bidder>
      <ID>Tim</ID>
      <Rating>5</Rating>
    </Bidder>
  </Book>
  <Book>
    <Title>Harry Potter and the Sorcerer's Stone</Title>
    <Author>J.K. Rowling</Author>
    <Seller>
      <ID>Mike</ID>
      <Rating>30</Rating>
    </Seller>
    <First_Bid>$5.00</First_Bid>
    <Current_Bid Time_Left = “34 hrs.”>$10.00</Current_Bid>
    <Bidder>
      <ID>Mark</ID>
      <Rating>125</Rating>
    </Bidder>
  </Book>
```

**Figure 1.2 A piece of auction data of new version**

avoids repeated work. Since the *delta* data is usually much smaller than the original data, query evaluation will also be much faster.

- ***Trigger Condition Evaluation*** In a continuous query or trigger system [CDTW00], the condition of firing a trigger is often defined on a specific change to one or more data sources. The change detection tool can quickly report such changes, filtering out other changes.

This paper describes X-Diff, an algorithm for computing the differences between two versions of an XML document. The key features of this algorithm include:

- ***XML Structure Information*** An XML document is generally a hierarchically structured document, and can be represented in a tree structure. For instance, the data in Figure 1.1 is shown as a tree in Figure 1.3. However, an XML document has other features that distinguish it from a general labeled tree. We introduce the notion of *node signature* and a new matching between the trees corresponding to the two versions of a document. Together, these two features are used to find the minimum-cost matching and generate a minimum-cost edit script that is capable of transforming the original version of the document to the new version.
- ***Unordered Trees*** Since XML documents can be represented as trees, the change detection problem is related to the problem of change detection on trees. Algorithms to compute the difference between trees can be divided into two categories depending on whether they deal with *ordered* or *unordered* trees. An *ordered* tree is one in which both the ancestor (parent-child) relationship and the left-to-right ordering among siblings are significant. An *unordered* tree is one in which only ancestor relationships are significant, while the left-to-right order among siblings is not significant. For database applications of XML we believe that the *unordered* tree model is more important. Thus, X-Diff is designed to handle *unordered* tree representations of XML documents. This is one major difference between our work and earlier efforts in this area [CRGW96, CE99, CAM02].
- ***High Performance*** Change detection on *unordered* trees is substantially harder than that on *ordered* trees; Zhang et al. have shown it to be NP-Complete in general case [ZSS92]. By exploiting certain features of XML documents, we present a polynomial algorithm that computes the “optimal” difference between two XML documents. We also propose an improvement on the algorithm that achieves high efficiency while generating near-optimal result.

The remainder of the paper is organized as follows. Related work is contained in Section 2. In Section 3, we formulate the problem and give an overview of our approach. We also give definitions of basic operations, edit scripts, cost model, node signature and matching. Section 4 presents the details of our algorithm with

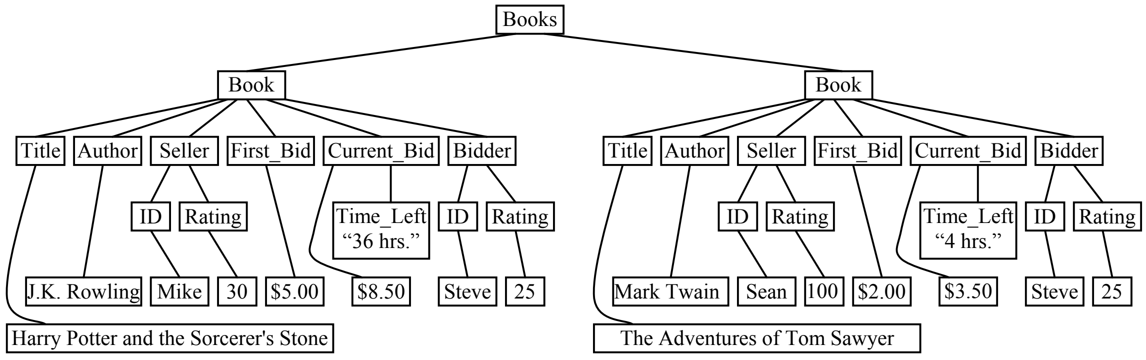

**Figure 1.3 Tree representation of document in Figure 1.1**

complexity analysis. Section 5 gives some preliminary performance results. Our conclusions and future research directions are contained in Section 6.

## 2. Related Work

Most previous work in change detection has focused on computing differences between flat files. The GNU *diff* utility is probably the most famous one. This algorithm uses the LCS (Longest Common Subsequence) algorithm [Myers86] to compare two plain text files. *CVS*, another GNU utility, uses *diff* to detect differences between two versions of programs [CVS]. Chawathe et al. [CRGW96] 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. Typical hierarchically 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.

The AT&T Internet Difference Engine [DB96, DBCK98] uses *HtmlDiff* [Berk] to determine the differences between two HTML pages. *HtmlDiff* treats two HTML pages as two sequences of tokens (a token is either a sentence-breaking markup or a sentence) and uses a weighted LCS algorithm [Hirs77] to find the best matching between the two sequences. This method cannot be applied to XML documents because markups in XML data provide context, and contents within different markups cannot be matched.

Since XML documents can be represented as trees, it is a natural idea to utilize tree-to-tree correction techniques [Selk77, Tai79, HD82], to detect changes in XML documents. Zhang and Shasha proposed a fast algorithm to find the minimum cost editing distance between two ordered labeled trees [ZS89]. Given two ordered trees $T_1$ and $T_2$, in which each node has an associated label, their algorithm finds an optimal edit script in time $O(|T_1| \times |T_2| \times \min\{\text{depth}(T_1), \text{leaves}(T_1)\} \times \min\{\text{depth}(T_2), \text{leaves}(T_2)\})$, which is the best known result for the general tree-to-tree correction problem.

Chawathe et. al. [CRGW96] formulated the change detection problem on hierarchically structured documents, and proposed an efficient algorithm based on the following key assumption:

Given two labeled trees, $T_1$ and $T_2$, there is a "good" matching function *compare*, so that given any leaf *s* in $T_1$, there is at most one leaf in $T_2$ that is "close" enough to match *s*.

This algorithm runs in time $O(ne + e^2)$, where $n$ is the number of tree leaves and $e$ is "weighted edit distance" (typically, $e << n$). This assumption holds well for many SGML documents that do not contain duplicate or similar objects, but it does not hold for many XML documents. For example, in the documents contained in Figure 1.1 and 1.2, there may be many users with the same rating, and many books that have the same bid price. In those cases, the algorithm is not guaranteed to generate the optimal result. Chawathe also proposed an external-memory algorithm for computing a minimum-cost edit script for two rooted trees [Chaw99].

XMLTreeDiff[1] [CE99] computes the difference between two XML documents. First, it computes hash values for the nodes of both documents using DOMHash [MTU98][2] and reduces the size of the two trees by removing identical subtrees (i.e., ones with identical hash values). Second, it uses Zhang and Shasha's algorithm [ZS89] to generate the difference between the two simplified trees. While using DOMHash to filter out identical subtrees can reduce the size of the two trees, its use conflicts with the cost model employed by Zhang and Shasha's algorithm. Thus, XMLTreeDiff may not generate an optimal result.

Recently, Cobéna et al. [CAM02] proposed XyDiff, an algorithm for detecting changes in XML documents. The algorithm first computes a signature (i.e., hash value) and a weight (i.e., subtree size) for every node in both documents in a bottom-up fashion (the root nodes of the two documents end up with the largest weights). Next, starting with the root nodes of the two documents XyDiff compares the signatures of the two nodes. If they are equal, the two nodes are matched; otherwise, their child

[1] Available at http://www.alphaworks.ibm.com.
[2] DOMHash was initially defined for two applications, computing digital signatures for XML documents, and synchronization of DOM structures.

nodes will be inserted into a priority queue in which the subtrees with the largest weights are **always** compared first. Whenever XyDiff finds an exact match between two subtrees, it attempts to propagate the match to the respective parents of the two nodes with the weight of each subtree determining how many levels the matching is propagated. Whenever there is more than one potential candidate for matching, XyDiff uses a few simple heuristic rules to pick one in order to avoid having to perform a full evaluation of the alternatives. This algorithm achieves O(*n*log*n*) complexity in execution time and generates fairly good results in many cases. However, XyDiff cannot guarantee any form of optimal or near-optimal result because of the greedy rules used in the algorithm. In fact, as we will demonstrate in Section 5, in many cases it will mismatch subtrees resulting in the generation of incorrect results.

Both XMLTreeDiff and XyDiff focus on change detection for ordered trees. On the other hand, there are very few algorithms capable of handling unordered trees. Zhang et al. [ZSS92] proved that general unordered tree-to-tree correction problem is NP-complete. Zhang also proposed a polynomial-time algorithm based on a restriction that matching is only performed between nodes at the same level [Zha93]. As mentioned in the previous section, XML documents have more features than general labeled trees that can be exploited to pursue an efficient algorithm. Our X-Diff algorithm uses the notions of matching and editing distance, which are introduced in [ZS89][3], adding special criteria that apply to XML documents.

Tararinov et. al. [TIHW01] proposed a set of primitive operations for modifying the structure and content of an XML document. The output of our algorithm can be easily adapted into such format.

## 3. Problem Overview and Preliminaries

In this section, we formulate the change detection problem. In Section 3.1, we describe the tree-structure representation of XML documents. We also explain why our approach focuses on change detection on unordered trees. Section 3.2 defines the basic edit operations, and we describe edit scripts in Section 3.3. These scripts consist of a list of basic editing operations, transforming one tree to another. Section 3.4 defines a cost model for edit scripts and defines the minimum-cost edit script. In Section 3.5, we introduce the notion of node signature that distinguishes nodes in an XML context. We use node signature to define a new tree-to-tree matching.

### 3.1 Tree Representation of XML Documents

In order to design an efficient algorithm to detect changes to XML documents, we need to understand the hierarchical structure in XML. Based on the Document Object Model (DOM) specification [W+00], an XML document can be represented as a tree.

Here we discuss three kinds of nodes in DOM tree[4].

- **Element** nodes – non-leaf nodes with one label, *name*.
- **Text** nodes – leaf nodes with one label, *value*.
- **Attribute** nodes – leaf nodes with two labels, *name* and *value*.

According to the DOM specification, element nodes and text nodes are *ordered*, while attribute nodes are *unordered*. In many applications XML documents can be treated as *unordered* trees – only ancestor relationships are significant, while the left-to-right order among siblings is not significant. In the document showed by Figure 1.1 and 1.2, the order of two books is reversed, but this is not significant.

In X-Diff, change detection is focused on unordered trees, which is the one major difference from [CRGW96, CE99, CAM02]. Most *ordered* tree-to-tree correction approaches cannot be applied to *unordered* trees because their correctness generally depends on preserving the left-to-right order when matching nodes.

We term two trees *isomorphic* if they are identical except for the orders among siblings. X-Diff considers two trees are equivalent if they are *isomorphic*.

### 3.2 Edit Operations

In this section, we define three basic edit operations on DOM Trees.

**Definition 3.1**

- **Insert**($x$(*name*, *value*), $y$) – insert a node $x$, with node name "*name*" and node value "*value*", as a leaf child node of node $y$.
- **Delete**($x$) – delete a leaf node $x$.
- **Update**($x$, *new_value*) – change the value of a leaf node $x$ to *new_value*. Note, $x$ has to be either a text node or an attribute node. Update can only modify a node's value, but not its name.

Notice that all basic operations are defined on leaf nodes. For convenience, we also have two composite operations:

- **Insert**($T_x$, $y$) – insert a subtree $T_x$, which is rooted at node $x$, to node $y$.
- **Delete**($T_x$) – delete a subtree $T_x$, which is rooted at node $x$. We also use **Delete**($x$) if there is no confusion.

Both operations represent a list of basic operations respectively.

The definition of these three operations is similar to that in [CRGW96, CAM02] except:

For insertion, we do not need to specify which position among $y$'s child nodes to insert node because we are dealing with unordered trees.

[3] The counterpart of "matching" in [ZS89, Zha93] is "mapping".

[4] X-Diff can also handle other kinds of nodes, such as Comment Nodes, and PI (Processing Instructions) Nodes.

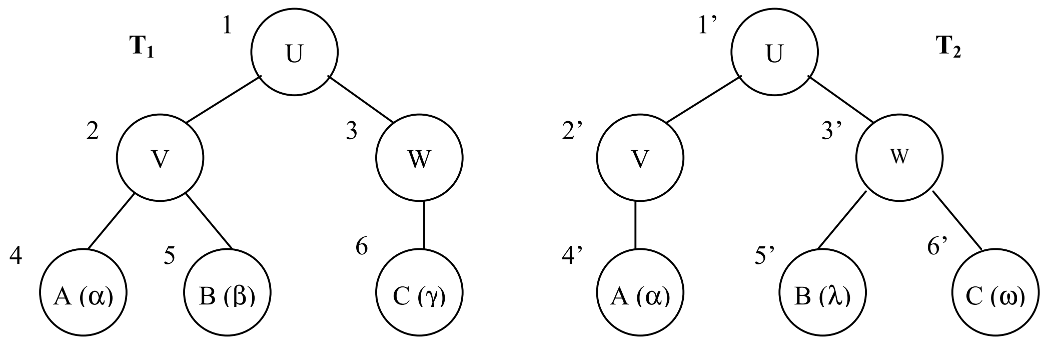

**Figure 3.1 An example for edit scripts**

We do not have a "move" operation, which transfers a node or a subtree from one position to another. Many "move" operations are not necessary in the unordered tree model because the order among siblings is not important. Other "move" operations can be replaced by a combination of "delete" and "insert" operations.

## 3.3 Edit Scripts

An *edit script* is a sequence of basic edit operations that convert one tree into another [CRGW96].

**Example 3.1** Consider the following trees $T_1$ and $T_2$ in Figure 3.1 (capital letters denote node name; Greek letters denote node value), the following edit script transforms $T_1$ into $T_2$:

$E(T_1 \rightarrow T_2)$ = Delete(5), Insert(5(B,λ),3), Update(6,ω).

## 3.4 A General Cost Model for Edit Scripts

Intuitively, given two trees, there can be many valid edit scripts capable of transforming one tree to the other. For instance, consider the following edit script E′ for the example in Figure 3.1:

$E'(T_1 \rightarrow T_2)$ = Update(5,λ), Delete(5), Insert(5(B,λ),3), Update(6, ω).

Apparently, E′ is not as good as E, but we need to define a standard cost model to evaluate alternative edit scripts to determine which one(s) is(are) best. The cost model will also affect the algorithm design. We use a simple cost model in X-Diff.

**Definition 3.2** Given an edit script E, **Cost**(E) = n, where $E = O_1 O_2 \ldots O_n$ and $O_i$ is a basic edit operation defined in Definition 3.1.

This definition can be easily extended by assigning different costs to different operations, which can also be applied to X-Diff. We believe the cost model above always accurately reflects real situations.

Based on the definition of the cost of edit scripts, we can define *minimum-cost edit script*, or *optimal edit script*.

**Definition 3.3** E is an edit script that transforms tree $T_1$ to $T_2$. E is called a *minimum-cost edit script*[5], or an *optimal edit script* for $(T_1 \rightarrow T_2)$ iff ∀ edit script E′ of $(T_1 \rightarrow T_2)$, Cost(E′) ≥ Cost(E).

We also define the *editing distance* between two trees. Let **Dist**($T_1$, $T_2$) denote the *editing distance* from $T_1$ to $T_2$.

**Definition 3.4 Dist**($T_1$,$T_2$) = Cost(E), where E is a minimum-cost edit script for $(T_1 \rightarrow T_2)$.

Both *minimum-cost edit script* and *editing distance* can be defined on subtree pairs.

**Definition 3.5** E is an edit script that transforms subtree $T_x$ to $T_y$. E is called a *minimum-cost edit script* for $(T_x \rightarrow T_y)$ iff ∀ edit script E′ for $(T_x \rightarrow T_y)$, Cost(E′) ≥ Cost(E); **Dist**($T_x$,$T_y$) = Cost(E).

## 3.5 Node Signature and Minimum-Cost Matching

In order to find the difference (generating an edit script) between two trees, we first need to find a matching of corresponding nodes in the two trees. Intuitively, we are not willing to try to match every node in the first tree to every node in the second tree because each node in XML has its own context. "Bad" matching will violate the context and cause unnecessary computation.

For example, a "Title" node won't match an "Author" node. Similarly, we don't match nodes with different node types – Text nodes should not be matched with Element nodes, or Attribute nodes. Notice it is not enough to compare node type and node name of two nodes to decide if they can be matched. For instance, we should not match the "Name" node within a "Seller" node to the "Name" node of a "Bidder" node.

Here we define node *signature* as the first criterion for matching two nodes. Given a DOM tree T, let Root(T) denote the root of T. Given a node $x$ in T, let Type($x$) denote the node type of $x$, Name($x$) denote the node name of $x$, and Value($x$) denote the node value of $x$[6].

**Definition 3.6** Suppose $x$ is a node in a DOM tree T, **Signature**($x$)=/Name($x_1$)/Name($x_2$)/…/Name($x_n$)/Name($x$) /Type($x$), where $x_1$ is the root of T, ($x_1$, $x_2$, $x_3$, …, $x_n$, x) is

[5] There can be more than one minimum-cost edit scripts in a transformation, but their costs are equal.

[6] A text node does not have a name. An element node does not have a value.

the path from root to $x$. If $x$ is a text node, **Signature**($x$) = /Name($x_1$)/Name($x_2$)/ … /Name($x_n$)/Type($x$).

We obtain the *signature* of a node by concatenating the names of all its ancestors with its own name and type[7]. In our algorithm, we only match nodes that have the same signature. Since all ancestor nodes are non-leaf nodes and non-leaf nodes must be element nodes, we do not include the types of ancestor nodes in the signature.

Next we define the notion of a *matching*.

**Definition 3.7** A set of node pairs ($x$, $y$), M, is called a *matching* from $T_1$ to $T_2$, iff

1) $\forall$ ($x$, $y$) $\in$ M, $x \in T_1$, $y \in T_2$, Signature($x$) = Signature($y$).
2) $\forall$ ($x_1$, $y_1$) $\in$ M, and ($x_2$, $y_2$) $\in$ M, $x_1 = x_2$ iff $y_1 = y_2$; (one-to-one)
3) M is prefix closed, i.e., given ($x$, $y$) $\in$ M, suppose $x'$ is the parent of $x$, $y'$ is the parent of $y$, then ($x'$, $y'$) $\in$ M.

Clearly, M preserves ancestor relationships.

**Lemma 3.1** Suppose ($x_1$, $y_1$) $\in$ M, ($x_2$, $y_2$) $\in$ M, $x_1$ is an ancestor of $x_2$ iff $y_1$ is an ancestor of $y_2$.

Criterion 3 prevents children being matched if their ancestors are not matched. This criterion reflects the integrity of XML segments.

**Lemma 3.2** M is a mapping from $T_1$ to $T_2$, M = {} iff (Root($T_1$), Root($T_2$)) $\notin$ M.

Criteria 1 and 3 represent the major differences between our definition of matching and that in [Zha93]. They reduce the matching space dramatically and make the algorithm efficient.

Based on a matching M from $T_1$ to $T_2$, we can generate an edit script for ($T_1 \rightarrow T_2$). Basically, we delete nodes in $T_1$ that do not exist in M, insert nodes in $T_2$ that do not exist in M, and update nodes that are in M but have different values. The complete algorithm is described in Section 4.5.

In the following theorem, we show that a *minimum-cost edit script* can be generated from the best matching.

**Theorem 3.1** There is a matching M from $T_1$ to $T_2$, from which we can generate a minimum-cost edit script for ($T_1 \rightarrow T_2$). M is called a *minimum-cost matching* from $T_1$ to $T_2$, denoted by $M_{\min}(T_1, T_2)$.

Before we prove Theorem 3.1, we give two lemmas about editing distance.

**Lemma 3.3** Suppose both $x$ and $y$ are leaf nodes, $x \in T_1$, $y \in T_2$; $\phi$ denotes *null*.

1) *Dist*($x$, $y$) = 0 iff Signature($x$) = Signature($y$), Value($x$) = Value($y$) (identical);
2) *Dist*($x$, $y$) = 1 iff Signature($x$) = Signature($y$), Value($x$) $\neq$ Value($y$) (update);
3) *Dist*($x$, $\phi$) = *Dist*($\phi$, $y$) = 1 (delete & insert).

**Lemma 3.4** **Dist**($T_x$, $\phi$) = Cost(Delete($T_x$)); **Dist**($\phi$, $T_y$) = Cost(Insert($T_y$)); **Dist**($T_x$, $T_y$) = **Dist**($T_x$, $\phi$) + **Dist**($\phi$, $T_y$) iff Signature($x$) $\neq$ Signature($y$).

[7] The "/" here is a little different from that used in Xpath [CD+99].

Here we give a brief proof for Theorem 3.1.

**Proof:** We perform mathematical induction on the height of both trees.

1) height($T_1$) = height($T_2$) = 1, i.e., both $T_1$ and $T_2$ are a single node.
   Suppose $T_1$ is node $x$, $T_2$ is node $y$. According to Lemma 3.2, the minimum-cost matching is,
   $M_{\min}(T_1, T_2)$ = {(x, y)} iff Signature($x$) = Signature($y$); otherwise $M_{\min}(T_1, T_2)$ = {}.
2) Suppose the theorem is true when height($T_1$) = $h_1$, height($T_2$) = $h_2$, $h_1 \geq 1$, $h_2 \geq 1$.
   Consider height($T_1$) = $h_1$+1, height($T_2$) = $h_2$+1.

a) Signature(Root($T_1$)) $\neq$ Signature(Root($T_2$)).
   According to Lemma 3.2, obviously
   $M_{\min}(T_1, T_2)$ = {}; Dist($T_1$, $T_2$) = Cost(Delete($T_1$), Insert($T_2$)).
b) Signature(Root($T_1$)) = Signature(Root($T_2$)).
   Suppose $x_1, x_2, \ldots, x_m$ are second-level nodes in $T_1$, $y_1, y_2, \ldots, y_n$ are second-level nodes in $T_2$. $\forall$ $x_i$ and $y_j$, there is a minimum-cost matching $M_{\min}(Tx_i, Ty_j)$ between $Tx_i$ and $Ty_j$, which editing distance is Dist($Tx_i$, $Ty_j$). Suppose $W$ is a 1-1 (partial) bipartite mapping between ($x_1, x_2, \ldots, x_m$) and ($y_1, y_2, \ldots, y_n$), then

$$Dist(T_1,T_2) = \min_{W}\{ \sum_{(x_i,y_j)\in W} Dist(T_{x_i},T_{y_j}) + \sum_{x_i\, is\, unmapped} Dist(T_{x_i},\phi) + \sum_{y_j\, is\, unmapped} Dist(\phi,T_{y_j}) \}$$

i.e., if $W_{\min}$ is a minimum-cost bipartite mapping between ($x_1, x_2, \ldots, x_m$) and ($y_1, y_2, \ldots, y_n$),

$$Dist(T_1,T_2) = \sum_{(x_i,y_j)\in W_{\min}} Dist(T_{x_i},T_{y_j}) + \sum_{x_i\, is\, unmapped} Dist(T_{x_i},\phi) + \sum_{y_j\, is\, unmapped} Dist(\phi,T_{y_j})$$

So we obtain the minimum-cost matching between $T_1$ and $T_2$,

$$M_{\min}(T_1,T_2) = \bigcup_{(x_i,y_j)\in W_{\min}} M_{\min}(T_{x_i},T_{y_j}) \bigcup \{(Root(T_1), Root(T_2))\} \qquad \square$$

We will describe the *Matching* algorithm in Section 4.4.

## 4. Change Detection with X-Diff

In this section we describe the X-Diff algorithm. Section 4.1 describes the overall algorithm that consists of several phases. We discuss each phase in detail in the Sections 4.2 through 4.4. In Section 4.5 we analyze the algorithm and estimate its time complexity. Section 4.6 presents a variant of the algorithm with improved performance but which does not guarantee an optimal result.

### 4.1 Outline of the X-Diff Algorithm

Given two XML documents, $DOC_1$ and $DOC_2$, $T_1$ and $T_2$ are their Tree representations. X-Diff determines if $DOC_2$

is different from $DOC_1$ based on the *unordered* model. If so, X-Diff finds a *minimum-cost matching* from $T_1$ to $T_2$, and generates a *minimum-cost edit script* for ($T_1 \rightarrow T_2$).

There are three steps in X-Diff as shown in Figure 4.1:

1. **Parsing and Hashing** X-Diff parses $DOC_1$ and $DOC_2$ into XTrees $T_1$ and $T_2$. During the parsing process, X-Diff will compute an XHash value for every node, which is used to represent the entire subtree rooted at the node.
2. **Matching** First, X-Diff compares XHash values of Root($T_1$) and Root($T_2$). $T_1$ and $T_2$ are considered equivalent if two XHash values are equal; otherwise, X-Diff finds $M_{min}(T_1, T_2)$, a minimum-cost matching between two trees.
3. **Generating Minimum-Cost Edit Script** X-Diff generates a minimum-cost edit script E for ($T_1 \rightarrow T_2$), based on the $M_{min}(T_1, T_2)$ found in Step 2.

```
Input: ($DOC_1$, $DOC_2$)
/* Parsing and Hashing. */
Parse $DOC_1$ to $T_1$ and hash $T_1$;
Parse $DOC_2$ to $T_2$ and hash $T_2$;
/* Checking and Filtering. */
If ( XHash (Root($T_1$)) = XHash (Root($T_2$)) )
  $DOC_1$ and $DOC_2$ are equivalent, stop.
else
  Do *Matchng* – Find a minimum-cost matching $M_{min}(T_1, T_2)$
  from $T_1$ to $T_2$.
/* Generating minimum-cost edit script */
Do *EditScript* – Generate the minimum-cost edit script E from
$M_{min}(T_1, T_2)$.
```

**Figure 4.1 Outline of X-Diff Algorithm**

## 4.2 Parsing and Hashing

This step is the preprocessing step in X-Diff. Two input XML documents, $DOC_1$ and $DOC_2$, are parsed into two Xtrees first. An Xtree provides a subset of interface of what a DOM tree does, and it is more efficient than a DOM tree.

During the parsing process, X-Diff uses a special hash function, XHash, to compute a hash value for every node on both trees. Similar to DOMHash [MTU98], the XHash value of a node represents the entire subtree rooted at this node. The difference is that DOMHash is used for the *ordered* tree model, while XHash is for the *unordered* tree model so that two *isomorphic* trees should have the same XHash value.

## 4.3 Matching

In this step we use the algorithm *Matching*, which is shown in Figure 4.2, to find a minimum-cost matching between $T_1$ and $T_2$. In Section 3.5, we proved that the minimum-cost matching can be found by computing the editing distance between $T_1$ and $T_2$, which is the core of *Matching*. Here we use an example to illustrate this algorithm.

To find a minimum-cost matching between $T_1$ and $T_2$, first we filter out equivalent subtrees between two root nodes by comparing the XHash values of second-level child nodes. Subtrees with identical XHash values can be considered to be equivalent because this is true with extremely high probability[8]. Since many XML documents are only slightly modified between versions, this step will reduce the tree size effectively.

```
Input: Tree $T_1$ and $T_2$.
Output: a minimum-cost matching $M_{min}(T_1, T_2)$.
Initialize: set initial working sets
  $N_1$ = {all leaf nodes in $T_1$}, $N_2$ = {all leaf nodes in $T_2$}.
  Set the Distance Table DT = {}.
/* Step 1: Reduce matching space */
Filter out next-level subtrees that have equal XHash values.
/* Step 2: compute editing distance for ($T_1 \rightarrow T_2$) */
DO {
  For every node $x$ in $N_1$
    For every node $y$ in $N_2$
        If Signature($x$) = Signature($y$)
          Compute Dist($x$, $y$);
          Save matching ($x$, $y$) with Dist($x$, $y$) in DT.
  Set $N_1$ = {parent nodes of previous nodes in $N_1$};
      $N_2$ = {parent nodes of previous nodes in $N_2$}.
} While (both $N_1$ and $N_2$ are not empty).
/* Step 3: mark matchings on $T_1$ and $T_2$. */
Set $M_{min}(T_1, T_2)$ = {}
If Signature(Root($T_1$)) $\neq$ Signature(Root($T_2$))
  Return;/* $M_{min}(T_1, T_2)$ = {}*/
Else
  Add (Root($T_1$), Root($T_2$)) to $M_{min}(T_1, T_2)$.
  For every non-leaf node mapping ($x$, $y$) $\in$ $M_{min}(T_1, T_2)$
      Retrieve matchings between their child nodes that are
stored in DT.
      Add child node matchings into $M_{min}(T_1, T_2)$.
```

**Figure 4.2 Matching Algorithm**

Second we compute the editing distance for each of rest subtree pairs and get minimum-cost matchings between subtrees. Finally, we compute the editing distance between $T_1$ and $T_2$, and obtain the minimum-cost matching $M_{min}(T_1, T_2)$. On each level we always use XHash values of child nodes two filter out equivalent subtrees to reduce the matching space.

In the *Matching* algorithm, we use dynamic programming to compute Dist($T_1$, $T_2$). We start computing the editing distance from leaf node pairs and move upward. The editing distance between two leaf nodes or two subtrees, associated with their minimum-cost matching, is stored in a Distance Table, which is available when we compute the editing distance between subtrees that are rooted at their parent nodes. When computing the editing distance between subtrees, we use the minimum-cost maximum flow algorithm [Tar83,

[8] A full tree-to-tree comparison can be performed here to double-check the equivalence between two subtrees. The cost of this test is linear to the number of nodes in the subtrees.

Zha93] to find the minimum-cost bipartite mapping (recall the proof for Theorem 3.1).

Notice that Theorem 3.1 shows that we only need to compute the editing distance between nodes that have the same signature. This is critical to this algorithm because it reduces the mapping space significantly and helps our algorithm achieve polynomial time in complexity. Otherwise, we have to compute editing distance between all possible node pairs, which has been proven to be NP-Complete [ZSS92].

## 4.4 Generating Minimum Cost Edit Script

In this phase, we generate a minimum-cost edit script based on the minimum-cost matching found in the previous phase. This generation procedure is performed recursively from roots to leaves, shown in Figure 4.3.

```
Input: Tree $T_1$ and $T_2$, a minimum-cost matching $M_{min}(T_1, T_2)$, the distance table DT.
Output: an edit script E.
Initialize: set E = Null;
$x$ = Root($T_1$), $y$ = Root($T_2$).
If (x, y) $\notin M_{min}(T_1, T_2)$  /* Subtree deletion and insertion */
    Return "Delete($T_1$), Insert($T_2$)".
Else if Dist($T_1$, $T_2$) = 0
    Return "";
Else {
    For every node pair ($x_i$ , $y_j$) $\in M_{min}(T_1, T_2)$, $x_i$ is a child
     node of x, $y_j$ is a child node of $y$.
        If $x_i$ and $y_j$ are leaf nodes
            If Dist($x_i$, $y_j$) = 0
                Return "";
            Else        /* Update leaf node */
                Add Update($x_i$ , Value($y_j$)) to E;
        Else            /* Call subtree matching */
            Add EditScript($Tx_i$, $Ty_j$) to E;
            Return E;
    For every node $x_i$ not in $M_{min}(T_1, T_2)$
        Add "Delete($Tx_i$)" to E;
    For every node $y_j$ not in $M_{min}(T_1, T_2)$
        Add "Insert($Ty_j$ )" to E;
    Return E. }
```

**Figure 4.3 EditScript Algorithm**

## 4.5 Algorithm Analysis

In this section we briefly analyze the complexity of our algorithm. First we estimate the complexity of each step in the algorithm. $|T_1|$ and $|T_2|$ denote the number of nodes in $T_1$ and $T_2$.

**1. Parsing and Hashing** The time to parse two documents and construct trees is $O(|T_1|+|T_2|)$. Hashing is performed during parsing. Since we need to sort child node XHash values before computing parent node XHash values, the upper bound for the complexity of this step is $O(|T_1|\times\log(|T_1|) + |T_2|\times\log(|T_2|))$.

**2. Mapping** As described in Section 4.3, in this step, we compute the editing distance between every node pair $(x, y)$ (where $x \in T_1$, $y \in T_2$, Signature($x$) = Signature($y$)), from leaf nodes to roots. Here we analyze the complexity of this step in the worst case in which we cannot filter out any equivalent subtrees by comparing their XHash values although this is very unlikely to happen. The minimum-cost matching from $T_1$ to $T_2$ is obtained when the editing distance between the two root nodes is found. We estimate the complexity of this step by dividing it into two parts, leaf nodes matching and non-leaf nodes matching.

First, we consider leaf nodes matching. According to Lemma 4.3, the cost of computing the editing distance between two leaf nodes is **O**(1). So the cost of computing the editing distance for all leaf node pairs is bounded by

$$O(|T_1|\times|T_2|). \quad (1)$$

Second, we consider non-leaf nodes matching. According to Lemma 4.2, the editing distance of each non-leaf node pair $(x, y)$ (where $x \in T_1$, $y \in T_2$, Signature($x$) = Signature($y$)) is obtained by finding a minimum-cost matching between their child nodes. Let deg($x$) denote the out-degree of node $x$, i.e., the number of its child nodes. The cost of computing editing distance between $x$ and $y$ is bounded by

$$O(\deg(x)\times\deg(y)\times\max\{\deg(x),\deg(y)\} \times \log_2(\max\{\deg(x),\deg(y)\})) \quad \text{[Zha93]} \ (2)$$

Suppose there are $M$ common non-leaf signatures between $T_1$ and $T_2$, denoted by $S_1$ to $S_M$. $N_{1k}$ and $N_{2k}$ are the number of nodes in $T_1$ and $T_2$ whose signature is $S_k$. $x_{ki}$ and $y_{kj}$ denote nodes whose signature is $S_k$. The cost of computing the editing distance for all non-leaf node pairs is bounded by

$$\sum_{k=1}^{M}\sum_{i=1}^{N_{1k}}\sum_{j=1}^{N_{2k}}\{O(\deg(x_{ki})\times\deg(y_{kj})\times\max\{\deg(x_{ki}),\deg(y_{kj})\} \times\log_2(\max\{\deg(x_{ki}),\deg(y_{kj})\}))\} \quad (3)$$

Let deg($T_1$) and deg($T_2$) denote the maximum out-degree in $T_1$ and $T_2$, then

$$(3) \le \sum_{k=1}^{M}\sum_{i=1}^{N_{1k}}\sum_{j=1}^{N_{2k}}\{O(\deg(x_{ki})\times\deg(y_{kj})\times\max\{\deg(T_1),\deg(T_2)\} \times\log_2(\max\{\deg(T_1),\deg(T_2)\}))\} \quad (4)$$

Since $\sum_{k=1}^{M}\sum_{i=1}^{N_{1k}}\deg(x_{ki}) < |T_1|$ and $\sum_{k=1}^{M}\sum_{j=1}^{N_{2k}}\deg(y_{kj}) < |T_2|$,

$$(4) \le O(|T_1|\times|T_2|\times\max\{\deg(T_1),\deg(T_2)\}\times \log_2(\max\{\deg(T_1),\deg(T_2)\})) \quad (5)$$

Combining (1) and (5), the complexity of this step is bounded by

$$O(|T_1|\times|T_2|\times\max\{\deg(T_1),\deg(T_2)\}\times \log_2(\max\{\deg(T_1),\deg(T_2)\})) \quad (6)$$

**3. Generating Minimum-Cost Edit Script** The minimum-cost edit script is generated recursively by traversing all nodes once in $T_1$ and $T_2$. The time complexity is $O(|T_1|+|T_2|)$.

We can see that the running time in step 2 is the most significant of all steps, so the complexity of our algorithm is the complexity of step 2, shown by **(6)**.

### 4.6 Performance Improvement

The primary focus of the X-Diff algorithm is to compute and generate the best possible difference between two XML documents. In the previous section we demonstrated that X-Diff has a polynomial running time. In some cases, however, this may not satisfy users' needs. Assume, for example, that the document shown in Figure 1.1 contains 10,000 books and that each hour 1,000 <Book> elements are changed. In order to compute the optimal difference between the two versions of the document, X-Diff must compute the minimum editing distance between every updated <Book> element in the old and new versions, which means that it needs to compute the editing distance for 1 million pairs of nodes. While X-Diff guarantees to generate the best difference result, some users may be willing to sacrifice some degree of accuracy in exchange for improved response time. In this section we discuss an approach for improving X-Diff's response time.

Our motivation is to speed up X-Diff without suffering too much loss in result quality. We believe that generally when people attempt to compute the difference between two versions of a document, the two documents will generally not be much different. In the example above, except for recently inserted or deleted books, most <Book> elements are likely changed only slightly, such as the bidder's name, or the bidding price. That suggests that for every updated <Book> element in one document, it is very likely that we will find one and only one "good" match for the element in the other document. Thus, the editing distance between this element (subtree) and its "good" match is very likely to be significantly less than the editing distance between it and all the other elements. As a result, when we compute the editing distance for an element in X-Diff, as soon as we find out one such match, we can immediately match both nodes and not consider any other matching for these two nodes.

A natural idea is to use a **threshold value** to decide whether or not two nodes are a "good" match. Obviously, a good threshold should not be a static value across documents or even different levels within a document. If the threshold is too high, it tends to mismatch elements. On the other hand, if the threshold is too low, it cannot locate good matches and avoid a full evaluation. Our solution is to use sampling to calculate this threshold whenever there are more than a couple of updated nodes in the two documents. At each level when computing the editing distance for node pairs, we first randomly select a small number[9] of nodes from the first document. Second, for each node in this sample we compute the editing distance between this node and every candidate in the other document and find the smallest value, which represents the best match for this node. Then the threshold value is calculated as the average of the editing distances obtained by sampling.

In Section 5, we will show that the improved X-Diff algorithm runs much faster than the optimal X-Diff algorithm while still generating the optimal results in most cases.

## 5. Performance Evaluation

In this section we examine the performance of both the optimal X-Diff algorithm and the improved X-Diff algorithm. We compare both algorithms with XyDiff by showing some preliminary results on their running time and result quality.

### 5.1 Experimental Settings and Testing Dataset

We implemented X-Diff in C++[10], using Xerces C++ XML parser[11] v1.4.0, the same parser used by the XyDiff[12] algorithms. Both programs read in two versions of an XML document and generate the difference. All following experiments were performed on a Pentium® III 550 MHz PC with 256 MB memory. The operating System is RedHat® Linux 6.2.

We use the Actors data set[13], whose DTD is shown in Figure 5.1. The size of the documents used in our experiments ranges from 10 KB to 1MB. A program generates all three types of changes, "insert", "delete", and "update", randomly at each level. The program takes a parameter, the percentage of nodes being changed. All changes are equally distributed in half of the <*Actor*> elements.

```
<!ELEMENT Actors (Actor)* >
<!ELEMENT Actor (Name, Filmography) >
<!ELEMENT Name (FirstName, LastName) >
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Filmography (Movie)*>
<!ELEMENT Movie (Title,Year) >
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
```

**Figure 5.1 DTD of Actors data set**

Notice that XyDiff uses the *ordered* tree model. In order to provide a fair comparison, our change generator

[9] According to the large number / central limit theory, $\sqrt{n}$ is a "safe" number if there are *n* nodes [Fel71].

[10] Both C++ and Java version of X-Diff are available at http://www.cs.wisc.edu/~yuanwang/xdiff.html.

[11] Available at http://xml.apache.org/xerces-c/

[12] Downloaded from http://www-rocq.inria.fr/~cobena/XyDiffWeb/

[13] Downloaded from http://www.cs.wisc.edu/niagara/data.html

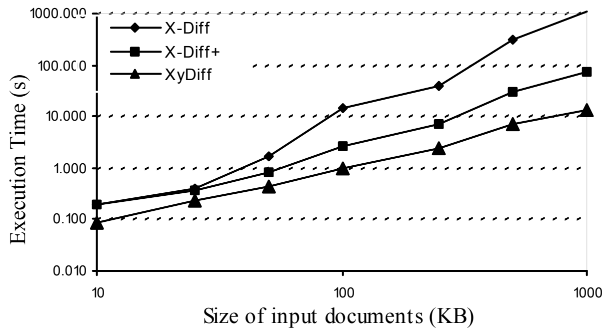

**Figure 5.2 Execution time on 1% change**

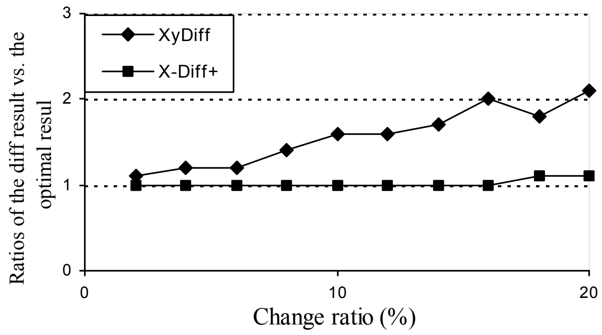

**Figure 5.4 Quality of diff result (1)**

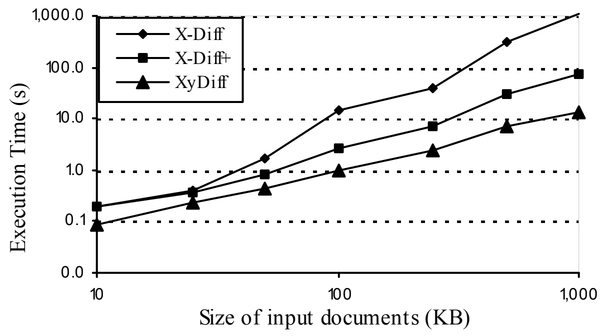

**Figure 5.3 Execution time on 5% change**

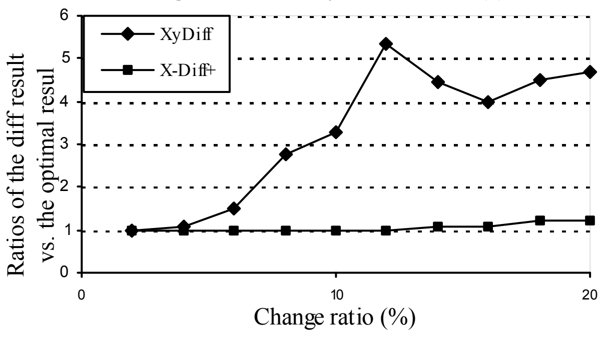

**Figure 5.5 Quality of diff result (2)**

```
<Actors>
  <Actor>
    <Name>
      <FirstName>Mike</FirstName>
      <LastName>Johnson</LastName>
    </Name>
    <Movies>
      <Title>movie1</Title>
      <Title>movie2</Title>
      <Title>movie3</Title>
    </Movie>
  </Actor>
  <Actor>
    <Name>
      <FirstName>Mike</FirstName>
      <LastName>Goodman</LastName>
    </Name>
    <Movies>
      <Title>movie1</Title>
      <Title>movie2</Title>
      <Title>movie3</Title>
    </Movie>
  </Actor>
</Actors>                    Document #1
```

```
<Actors>
  <Actor>
    <Name>
      <FirstName>Mike</FirstName>
      <LastName>Johnson</LastName>
    </Name>
    <Movies>
      <Title>movie4</Title>
      <Title>movie2</Title>
      <Title>movie3</Title>
    </Movie>
  </Actor>
  <Actor>
    <Name>
      <FirstName>Bill</FirstName>
      <LastName>Goodman</LastName>
    </Name>
    <Movies>
      <Title>movie1</Title>
      <Title>movie2</Title>
      <Title>movie3</Title>
    </Movie>
  </Actor>
</Actors>                    Document #2
```

**Figure 5.6 Two sample documents**

does not permute the order of any nodes; otherwise, it would bias the results in favor of the Xdiff algorithm.

## 5.2 Execution Time

First, we evaluate the execution time of the three algorithms, X-Diff, the improved X-Diff (represented by X-Diff+) and XyDiff on documents of different sizes. In Figure 5.2, 1% of the nodes are modified. In Figure 5.3, 5% of then nodes are changed.

Both figures show that X-Diff performs well on small- and medium-size documents. However, due to the complexity of the algorithm, its execution time is fairly long when the two input files are large. On the other hand, XyDiff is very efficient in that its running time is almost linear in the size of the document. Notice, however, while the improved X-Diff algorithm is still slower than XyDiff,

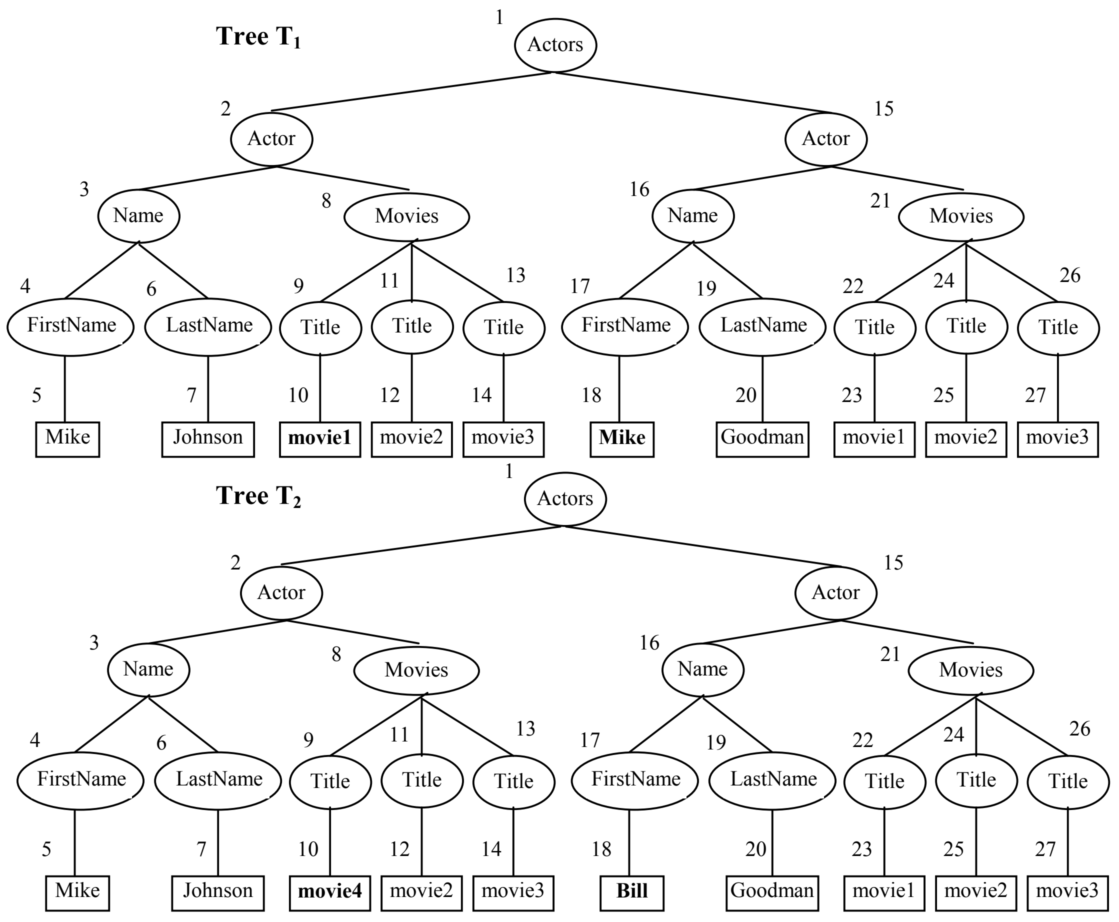

**Figure 5.7 Tree representation for both documents in Figure5.6**

it is much faster than the original X-Diff algorithm as using a threshold value obtained by sampling avoids unnecessary comparisons. Notice also that the absolute execution time of the two X-Diff algorithms does not increase significantly when the percentage of changed nodes increases from 1% to 5% (Figure 5.3 vs. Figure 5.2).

This is consistent with our complexity analysis which demonstrated that the execution time of the algorithm depends primarily on the total number of nodes and not the number of changed nodes.

## 5.3 Result Quality

In the next set of tests the result quality of each algorithm is compared. Since the original X-Diff algorithm was shown to always find the optimal difference in Section 3(and it does!), we only compare the improved X-Diff algorithm with XyDiff.

First we construct one hundred 50KB documents in which elements are randomly selected from the base data set used in the previous experiments. Then a series of new versions for each document was generated by varying the change ratio. X-Diff+ and XyDiff were run to compare the original version of the document with each of the new versions to obtain a series of differences for each algorithm. The results of each diff operation were then compared to the results obtained using the original X-Diff algorithm and the ratio plotted in Figure 5.4. The improved X-Diff algorithm almost always finds out the optimal difference until the change ratio reaches 18% where its result is very close to the optimal difference. On the other hand, the result generated by XyDiff is generally about 50% worse than the optimal result.

One of the reasons that XyDiff generates non-optimal results is that it has a tendency to mismatch nodes when guided by its greedy matching rules. For example, we illustrate two simple documents in Figure 5.6, and the tree representation of both documents is shown in Figure 5.7. The difference between the two documents is displayed in bold font. The editing list computed by both X-Diff and X-Diff+ is,

$E(T_1 \rightarrow T_2)$ = Update(10, movie4), Update(18, Bill).

However, the diff result generated by XyDiff is,

$E'(T_1 \rightarrow T_2)$ = Move(16, 2, 1)[14], Move(3, 15, 1), Update(18, Bill), Update(10, movie4), Move(2, 1, 2).

This is because XyDiff matches the *<Movies>* element of *Mike Johnson* to the *Bill Goodman*'s when it finds both subtrees are identical, although it is not a good

[14] This operation means "move the subtree rooted at node 16 to be a child of node 2 at position 1".

match from the higher-level's point of view. In this type of situation, no matter if the match is propagated to the upper level or not, it will generate a much longer difference than the optimal result.

In fact, the above example illustrates that when there are many small identical elements in both documents, XyDiff is likely to generate a significantly larger diff result than the optimal result. On the other hand, although X-Diff+ also uses a heuristic matching method, threshold matching, its top-down fashion avoids aggressive matching on small elements. Notice that the example is not that unusual. Considering the motivation example shown in Figures 1.1 and 1.2, different books may have the same author, or the same publisher, or even the same price, etc.

We use the next experiment to demonstrate this difference between X-Diff/X-Diff+ and XyDiff. Similar to the previous test, we randomly construct one hundred 50KB documents, but this time there are at average of five duplicate elements for every different <*Movie*> element across each document. We also randomly generate a series of new versions for each document and run both X-Diff+ and XyDiff over them. Figure 5.5 shows the ratios of the diff results of both algorithms compared to the optimal result. X-Diff+ generates significantly shorter diff results than XyDiff.

## 6 Summary and Future Work

X-Diff is motivated by the problem of efficiently detecting changes to XML documents on the web. Previous work in change detection on XML or other hierarchically structured data [CRGW96, CE99] used the ordered-tree model. In this paper, we argue that using the unordered-tree model is more suitable for most database and web applications, although it is substantially harder than using the ordered-tree model. We study the XML domain characteristics and introduce several key notions, such as node signature, and XHash. Using these techniques in combination with standard tree-to-tree correction techniques [Zha93], we propose X-Diff, an efficient algorithm for computing the optimal difference between two versions of an XML document. We present and analyze the algorithm, and also propose an improved X-Diff algorithm that runs much faster than the original algorithm while still generating at least near-optimal results. We present a preliminary performance evaluation of our algorithms, compared with XyDiff [CAM02]. The experiments show that the improved X-Diff algorithm generally generates more accurate results than XyDiff does, although it runs slower than XyDiff. It is suitable for the situations that users want to get more accurate results.

We are working on improving the performance of our algorithms. Other interesting future work includes change detection on XML data streams and incremental index update as XML documents are changed.

## 7 References

[Berk] E. Berk, "HtmlDiff: A Differencing Tool for HTML Documents", *Student Project*, Princeton University, http://www.htmldiff.com.

[CAM02] G. Cobéna, S. Abiteboul, A. Marian, "Detecting Changes in XML Documents", *ICDE*, Feb., 2002.

[CD+99] J. Clark, S. DeRose, et al., "XML Path Language (Xpath) Version 1.0", November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.

[CDTW00] J. Chen, D. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *SIGMOD Conference*, 2000.

[CE99] Curbera, D. A. Epstein, "Fast Difference and Update of XML Documents", *XTech'99*, San Jose, March 1999.

[Chaw99] S. Chawathe, "Comparing Hierarchical Data in External Memory", *VLDB Conference,* 1999.

[CRGW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, "Change Detection in Hierarchically Structured Information", *SIGMOD Conference,* 1996.

[CVS] "Concurrent Versions System (CVS)", *Free Software Foundation*, http://www.gnu.org/manual/cvs-1.9.

[DB96] F. Douglis, T. Ball, "Tracking and Viewing Changes on the Web", *1996 USENIX Annual Technical Conference*, 1996.

[DBCK98] F. Douglis, T. Ball, Y. F. Chen, E. Koutsofios, "The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web", *World Wide Web*, 1(1): 27-44, Jan. 1998.

[Fel71] W. Feller, "An Introduction to Probability Theory and Its Applications", 2nd ed., John Wiley & Sons, 1971.

[Hirs77] D. S. Hirschberg, "Algorithm for the Longest Common Subsequence Problem", *Journal of the ACM*, 24(4): 664-675, October 1977.

[HD82] C. M. Hoffmann, M. J. O'Donnell, "Pattern Matching in Trees", *Journal of the ACM*, 29: 68-95, 1982.

[MTU98] H. Maruyama, K. Tamura and R. Uramoto, "Digest values for DOM (DOMHash) proposal", *IBM Tokyo Research Lab.*, http://www.trl.ibm.co.jp/projects/xml/domhash.htm, 1998.

[Myers86] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations", *Algorithmica*, 1(2): 251-266, 1986.

[Selk77] S. M. Selkow, "The Tree-to-Tree Editing Problem", *Information Processing Letters*, 6(6): 184-186, 1977.

[Tai79] K. C. Tai, "The Tree-to-Tree Correction Problem", *Journal of the ACM*, 26: 485-495, 1979.

[Tar83] R. E. Tarjan, "Data Structures and Network Algorithms", *CBMS-NSF Regional Conference Series in Applied Mathematics*, 1983.

[TIHW01] I. Tatarinov, Z. Ives, A. Halevy, D. Weld, "Updating XML", *SIGMOD Conference*, 2001.

[W+00] L. Wood, et. al., "Document Object Model (DOM) Level 1 Specification (Second Edition)", *World Wide Web Consortium*, http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/, 2000.

[W3C] "Extensible Markup Language (XML)", *World Wide Web Consortium*, http://www.w3.org/XML/.

[Zha93] K. Zhang, "A New Editing based Distance between Unordered Labeled Trees", *Combinatorial Pattern Matching*, 1: 254 – 265, 1993.

[ZS89] 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.

[ZSS92] K. Zhang, R. Statman, D. Shasha, "On the Editing Distance between Unordered Labeled Trees", *Information Processing Letters*, 42: 133-139, 1992.