# X-Diff: A Fast Change Detection Algorithm for XML Documents


## 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 ordered model, but the change result that it generates is more accurate. We propose X-Diff, a fast algorithm that integrates key XML structure characteristics with standard tree-to-tree correction techniques. We also analyze the algorithm and study its performance.


## 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 [2]. A number of web sites have begun to provide documents in XML format and several query languages have been developed for querying repositories of XML documents [6, 15, 16]. XML can also be used for specialized domains. For example, biotechnology industries are now using XML to store genetic information [12].

Since information on the web 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 his children at an online auction site. On the first visit she obtains a list of currently offered books and related information. Two hours later, she receives 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 below.

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.

```
<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>
</Books>
```

**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>
</Books>
```

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

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 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, 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. The X-Diff algorithm is very efficient because it exploits these differences. 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.

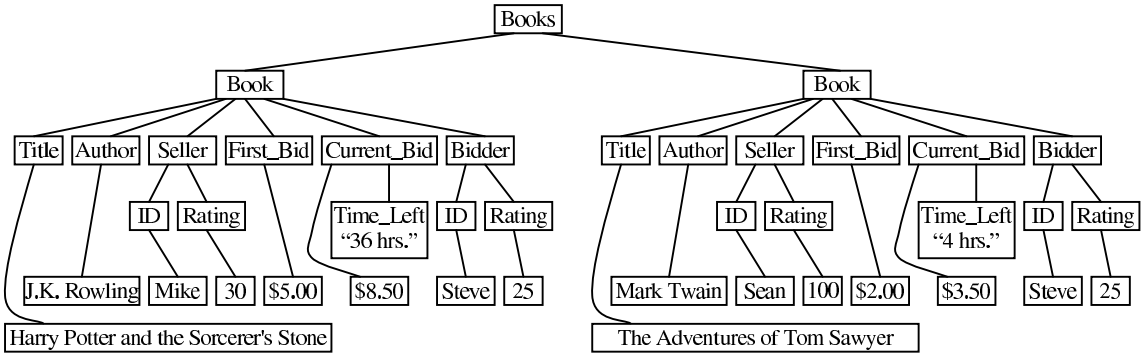


**Figure 1.3 Tree representation of document in Figure 1.1**

- **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 the major difference between our work and earlier efforts in this area [4, 5].
- **High Performance**. Change detection on *unordered* trees is substantially harder than that on *ordered* trees; [21] has shown it to be NP-Complete in general case. By exploiting certain features of XML documents, our algorithm achieves polynomial time in complexity.

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 an analysis of our algorithm. Section 5 discusses implementation issues and 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 [13] to compare two plain text files. *CVS*, another GNU utility, uses *diff* to detect differences between two versions of programs [1]. In [4], Chawathe et al. 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 [7, 8] uses *HtmlDiff* [3] 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 [9] 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 [10, 14, 17], 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 [20]. 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. formulated the change detection problem on hierarchically structured documents [4], 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*.

The algorithm runs in time $O(ne + e^2)$, where $n$ is the number of tree leaves and $e$ is "weighted edit distance" (typically, $e << n$). Although the assumption may hold for many SGML documents which do not contain duplicate or similar objects, it does not hold for most 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 proposed in [4] is not guaranteed to generate the optimal result.

XMLTreeDiff[1] [5] computes the difference between two XML documents. First, it computes hash values for the nodes of both documents using DOMHash [11] and then 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 [20] to generate the difference (in terms of an editing script) between the two simplified trees. While using DOMHash to filter out identical subtrees can dramatically 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.

Both [4] and [5] focus on change detection for ordered trees. On the other hand, there are very few algorithms capable of handling unordered trees. [21] proved that general unordered tree-to-tree correction problem is NP-complete. Zhang proposed a polynomial-time algorithm based on a restriction that matching is only performed between nodes at the same level [22]. 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 [20, 22][2], adding special criteria that apply to XML documents.

# 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 specification [19], an XML document can be represented as a tree.

**Example 3.1** Figure 3.1 shows a partial DOM tree of the document in Figure 1.1.

[1] Available at http://www.alphaworks.ibm.com.
[2] The counterpart of "matching" in [20, 22] is "mapping".

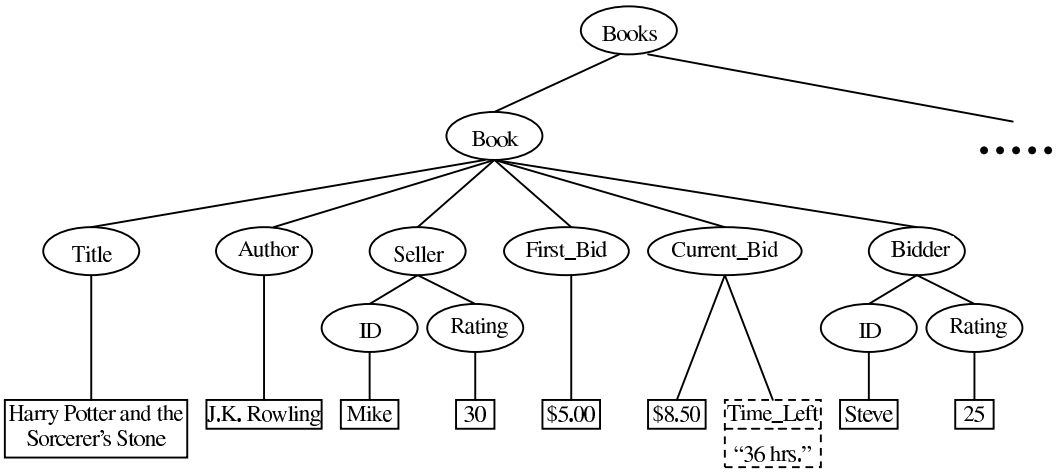


**Figure 3.1 DOM representation of partial document in Figure 1.1**

Here we discuss three kinds of nodes in DOM tree[3].

- **Element** nodes – represented by ovals, such as "Books" and "Book". Element nodes are non-leaf nodes with one label, *name*.
- **Text** nodes – represented by solid rectangles. Text nodes are leaf nodes with one label, *value*.
- **Attribute** nodes – represented by dashed rectangles. Attribute nodes are also leaf nodes, but they have 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, such as the example in Figure 3.1, 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 major difference from [4, 5]. *Ordered* tree-to-tree correction approaches [4, 5] 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*.

In this paper, we assume there are no keys or object identifiers in XML documents in general. As indicated in [4], we rely on structure and value information to detect change. The only exception is that some XML documents have "ID" attributes within element nodes. In this case those element nodes can be

[3] X-Diff can also handle other kinds of nodes, such as Comment Nodes, and PI (Processing Instructions) Nodes. Details can be found in [24].

easily matched by their "ID" attributes. We treat other attributes, such as "IDREF" and "IDREFS", as string attributes.

## 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 [4] except:

- For insertion, we do not need to specify which position among *y*'s child nodes to insert node *x* because we are dealing with unordered trees.
- 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 [4].

**Example 3.2** Consider the following trees $T_1$ and $T_2$ in Figure 3.2 (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).

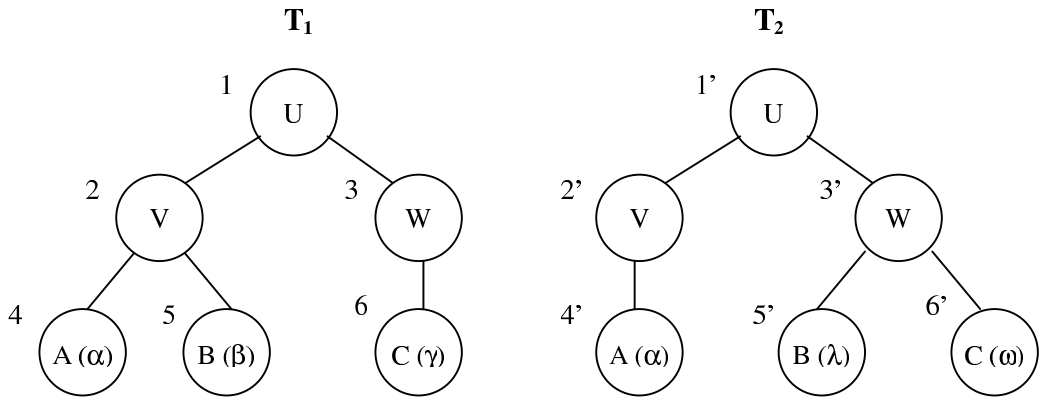


**Figure 3.2 Example for edit scripts**

## 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 3.2:

$E'(T_1 \rightarrow T_2)$ = 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. Similar to [4, 22], we use a simple cost model in X-Diff.

**Definition 3.2** Given an edit script E, **Cost**(E) = n, where E = $O_1$ $O_2$ … $O_n$ 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*[4], 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.

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

**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 $\forall$ 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 $x$[5].

**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 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[6]. 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. We use nodes in Figure 3.1 to illustrate the definition of node *signature*:

**Example 3.3** The signature of the "Author" node is "Books**.**Book**.**Author**.**$Element$"; the signature of the text node, "J.K. Rowling", is "Books**.**Book**.**Author**.**$Text$"; and the signature of the "Time_Left" node is "Books**.**Book**.**Current_Bid**.**Time_Left**.**$Attribute$".

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) $\forall$ ($x$, $y$) ∈ M, $x$ ∈ $T_1$, $y$ ∈ $T_2$, Signature($x$) = Signature($y$).
2) $\forall$ ($x_1$, $y_1$) ∈ M, and ($x_2$, $y_2$) ∈ M, $x_1 = x_2$ iff $y_1 = y_2$; (one-to-one)

[5] A text node does not have a name. An element node does not have a value.
[6] We use "**.**" as delimiter in the string of signature; "$" indicates node type.

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 $(Root(T_1), Root(T_2)) \notin M$.

**Example 3.4** Figure 3.3 shows a matching between two trees $T_1$ and $T_2$. $M = \{(1, 1'), (2, 2'), (3, 3'), (5, 4'), (7, 7')\}$. Notice that two leaf nodes in a node pair do not need to have same value. According to criterion 1, (6, 5') cannot be in M because of their different signatures. Although node 8 and node 6' have the same signature, (8, 6') cannot exist in M because their parents' node pair (4, 3') is not in M.

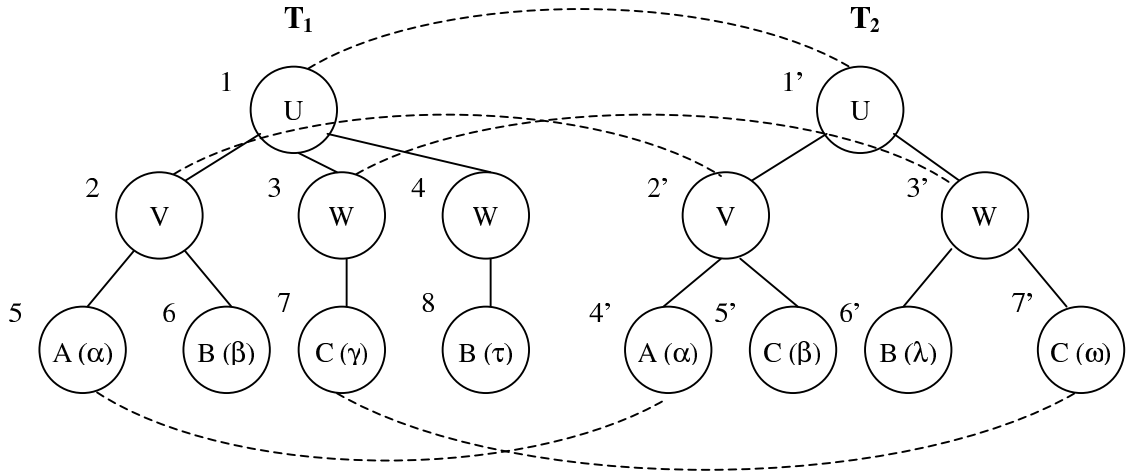


**Figure 3.3 An Example of Matching**

Criteria 1 and 3 represent the major differences between our definition of matching and that in [22]. 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.

**Example 3.5** According to the matching showed in Figure 3.3, E = Delete(6), Delete(8), Delete(4), Insert(5'(C, β), 2), Insert(6'(B, λ), 3) , Update(7, ω).

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($x$) ≠ Value($y$) (update);

3) $Dist(x, \phi) = Dist(\phi, y) = 1$ (delete & insert).

**Lemma 3.4** $\mathbf{Dist}(T_x, \phi) = \text{Cost}(\text{Delete}(T_x))$; $\mathbf{Dist}(\phi, T_y) = \text{Cost}(\text{Insert}(T_y))$;

$\mathbf{Dist}(T_x, T_y) = \mathbf{Dist}(T_x, \phi) + \mathbf{Dist}(\phi, T_y)$ iff Signature($x$) ≠ Signature($y$).

Here we give a brief proof for Theorem 3.1; the detailed proof can be found in [24].

**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($T_1$)) ≠ Signature(Root($T_2$)).
      According to Lemma 3.2, obviously
      $M_{min}(T_1, T_2) = \{\}$; $Dist(T_1, T_2) = \text{Cost}(\text{Delete}(T_1), \text{Insert}(T_2))$.
   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 ($y_1, y_2, \ldots, y_n$), which is illustrated by Figure 3.4, then

$$Dist(T_1, T_2) = \min_W \{ \sum_{(x_i, y_j) \in W} Dist(Tx_i, Ty_j) + \sum_{x_i \text{ is unmapped}} Dist(Tx_i, \phi) + \sum_{y_j \text{ is unmapped}} Dist(\phi, Ty_j) \}$$

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

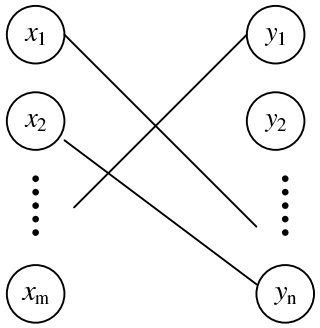


**Figure 3.4 Bipartite mapping between child nodes**

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

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

$$M_{\min}(T_1, T_2) = \bigcup_{(x_i, y_j) \in W_{\min}} M_{\min}(Tx_i, Ty_j) \quad \cup \quad \{(Root(T_1), Root(T_2))\} \qquad \square$$

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

# 4. Change Detection with X-Diff

In this section we describe the algorithm of X-Diff. Section 4.1 describes the overall algorithm which consists of several phases. We discuss each phase in detail in the Sections 4.2 through 4.5. In Section 4.6 we analyze the algorithm and estimate its time complexity.

## 4.1 Outline of the X-Diff Algorithm

Given two XML documents, $DOC_1$ and $DOC_2$, $T_1$ and $T_2$ are their DOM 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 several steps in X-Diff as shown in Figure 4.1:

1. **Parsing and Hashing** X-Diff parses $DOC_1$ and $DOC_2$ into DOM Trees $T_1$ 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. **Checking and Filtering** 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 will start to find the difference between $T_1$ and $T_2$. Then, X-Diff reduces the size of $T_1$ and $T_2$ by removing equivalent second-level subtrees.
3. **Matching** X-Diff finds $M_{min}(T_1, T_2)$, a minimum-cost matching between two trees.

4. **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 3.

Input: ($DOC_1$, $DOC_2$)

/* Parsing and Hashing. */

1. Parse $DOC_1$ to $T_1$ and hash $T_1$;
   Parse $DOC_2$ to $T_2$ and hash $T_2$;
   /* Checking and Filtering. */
2. a) If ( XHash (Root($T_1$)) = XHash (Root($T_2$)) )
   $DOC_1$ and $DOC_2$ are equivalent, stop.
   else go to b).
   b) For any second-level node pair ($x_1$, $x_2$),
   if (XHash($x_1$) = XHash($x_2$))
   Remove subtree rooted at $x_1$ from $T_1$;
   Remove subtree rooted at $x_2$ from $T_2$;
   /* Matching */
3. Algorithm *Matchng* – Find a minimum-cost matching $M_{min}(T_1, T_2)$ from $T_1$ to $T_2$.
   /* Generating minimum-cost edit script */
4. Algorithm *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 DOM trees first. Although a DOM tree contains much unnecessary information and consumes more memory, it provides a better programming interface to higher-level applications, such as a query engine.

During the parsing process, X-Diff uses a special hash function, XHash, to compute a hash value for every node on both trees. The XHash value of a node represents the entire subtree rooted at this node. The idea of XHash is inspired by DOMHash [11][7], while DOMHash is for the *ordered* tree model, XHash is for the *unordered* tree model. XHash uses the same basic function in DOMHash to compute hash value for leaf nodes, but a different function for non-leaf nodes.

Suppose **hash**(string) is a general hash function, such as MD5 and SHA, which takes a string as input and outputs a fixed length (128-bit) string as hash value. Suppose x is a leaf node, either a text node or an

[7] N. Maruyama et. al. proposed DOMHash [11] to compute hash value for nodes in DOM trees. DOMHash is defined for two applications, computing digital signatures for XML documents, and synchronization of DOM structures.

attribute node, we apply the hash function on the concatenation of its type, name, and value.

**XHash**($x$) = **hash**( Type($x$) • Name($x$) • Value($x$) ).

For a non-leaf node $x$, an element node, which has a list of child nodes, $x_1$, $x_2$, … $x_n$. XHash($x$) is obtained as follows:

1. Compute XHash($x_1$), XHash($x_2$), … XHash($x_n$).
2. Sort these XHash values, so that XHash($x_{1'}$) ≤ XHash($x_{2'}$) ≤ … ≤ XHash($x_{n'}$), $x_{1'}$,…, $x_{n'}$ are the nodes after sorting.
3. Concatenate Type($x$) and Name($x$) with the sorted XHash values, and compute the XHash value of $x$: **XHash**($x$) = **hash**( Type($x$) • Name($x$) • XHash($x_{1'}$) • XHash($x_{2'}$) • … • XHash($x_{n'}$) ).

By including the XHash values of $x$'s children nodes into $x$'s hash value, the XHash algorithm insures that the XHash value of a node x incorporates both its parent-child and child information. On the other hand, by sorting the child nodes the information about the order of child nodes is not included in $x$'s hash value. Thus, the XHash value of a non-leaf node can fully represent the subtree in the unordered model.

### 4.3 Checking and Filtering

Two trees (subtrees) with identical XHash values can be considered to be equivalent because this is true with extremely high probability. Thus we use the XHash values of the two root nodes to determine if two documents are equivalent. If both values are equal, X-Diff will claim that the two trees are equivalent, or isomorphic. If not, there must be some difference between the two trees.

Even if the two trees are actually different, X-Diff can use XHash values to reduce the size of both trees by removing equivalent second-level subtrees.

**Example 4.1** Figure 4.2 shows the top two level nodes of two versions of an XML document in DOM Trees. Dashed lines match nodes with equal XHash values. Equivalent subtrees will be removed, as shown in Figure 4.3.

Since many XML documents are only slightly modified between versions, this step will reduce the tree size effectively, avoiding unnecessary computations in subsequent phases of the algorithm.

Notice that we remove subtrees only at the second level instead of subtrees at any level as in [5]. It is because equivalent second-level subtrees must match each other in the minimum-cost matching, but other-level subtree pairs may not exist in the matching even if they have equal XHash values, as indicated by matching criterion 3 in Section 3.5.

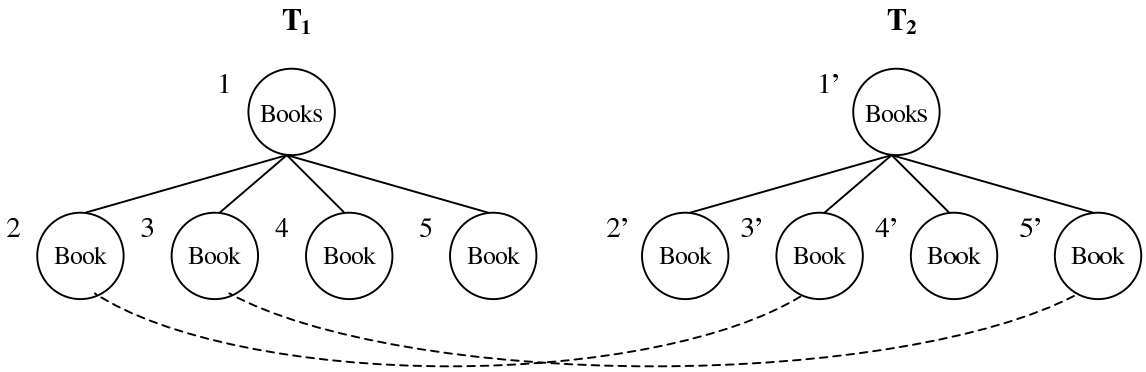


**Figure 4.2 Matching equivalent second-level subtrees**

**Figure 4.3 Trees after removing equivalent second-level subtrees**

## 4.4 Matching

The goal of this step is to find a minimum-cost matching between $T_1$ and $T_2$. The algorithm, *Matching*, is shown in Figure 4.5. 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.

**Example 4.2** To find a minimum-cost matching between $T_1$ and $T_2$ in Figure 4.4, first we compute the editing distance for each leaf node pairs (Lemma 3.3). Then we compute the editing distance for each subtree pairs, such as ($T_2$, $T_{2'}$) and ($T_2$, $T_{5'}$), 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 $M_{min}(T_1, T_2)$.

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 [18, 22] to find the minimum-cost bipartite mapping (recall the proof for Theorem 3.1).

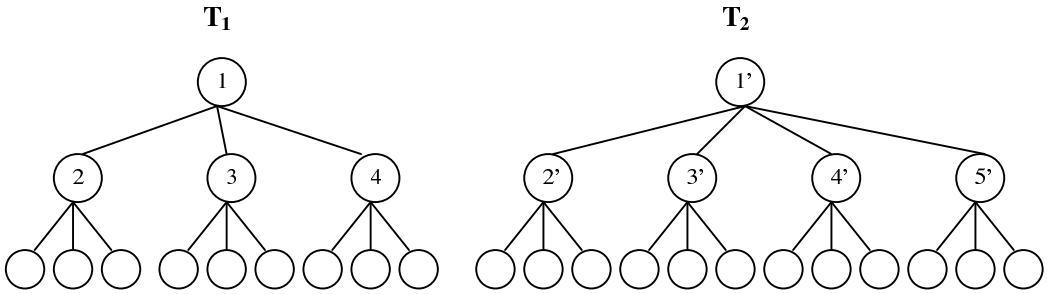


**Figure 4.4 Example for *Matching* Algorithm**

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: 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 2: mark matchings on $T_1$ and $T_2$. */

Set $M_{min}(T_1, T_2)$ = { }

If Signature(Root($T_1$)) ≠ 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*) ∈ $M_{min}(T_1, T_2)$

Retrieve matchings between their child nodes which are stored in DT.

Add child node matchings into $M_{min}(T_1, T_2)$.

**Figure 4.5 *Matching* Algorithm**

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 [21].

## 4.5 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 4.6.

```
Input: Tree T1 and T2, a minimum-cost matching Mmin(T1, T2), the distance table DT.
Output: an edit script E.
Initialize: set E = Null;
x = Root(T1), y = Root(T2).
If (x, y) ∉ Mmin(T1, T2)
   Return "Delete(T1), Insert(T2)".  /* Subtree deletion and insertion */
Else if Dist(T1, T2) = 0
   Return "";
Else
   For every node pair (xi , yj) ∈ Mmin(T1, T2), xi is a child node of x, yj is a child node of y.
      If xi and yj are leaf nodes
           If Dist(xi, yj) = 0
              Return "";
           Else
              Add Update(xi , Value(yj)) to E;  /* Update leaf node */
      Else
           Add EditScript(Txi, Tyj) to E; /* Call subtree matching */
           Return E;
   For every node xi  not in Mmin(T1, T2)
       Add "Delete(Txi)" to E;
   For every node yj not in Mmin(T1, T2)
       Add "Insert(Tyj )" to E;
    Return E.
```

**Figure 4.6 *EditScript* Algorithm**

## 4.6 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 $O(|T_2|+|T_2|)$. 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 $O(|T_2|\times\log(|T_1|)+|T_2|\times\log(|T_2|))$.

2. **Checking and Filtering** In this step, our algorithm compares the XHash values of both root nodes, and searches equivalent second-level nodes with time complexity $O(|T_2|)$.

3. **Mapping** As described in Section 4.4, 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. 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

   $$O(|T_2|\times|T_2|). \quad \textbf{(1)}$$

   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

   $$O(\deg(x)\times\deg(y)\times\max\{\deg(x),\deg(y)\}\times\log_2(\max\{\deg(x),\deg(y)\})) \text{ [22,24]} \quad \textbf{(2)}$$

   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

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

Let deg($T_1$) and deg($T_2$) denote the maximum out-degree in $T_1$ and $T_2$, then **(3)**

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

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

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

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

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

4. **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 $O(|T_2|+|T_2|)$.

We can see that the running time in step 3 is the most significant of all steps, so the complexity of our algorithm is the complexity of step3, shown by **(6)**.

# 5. Implementation and Performance

We have implemented X-Diff as a stand-alone tool. In Section 5.1, we describe the implementation of this program. In Section 5.2, we examine its performance and show some preliminary result of its running time. We also built X-Diff in a continuous query system [23].

## 5.1 Implementation

We have implemented X-Diff in C++ (we plan to provide a Java version in the near future). It takes two XML documents as input. First, it uses the IBM XML for C++ parser (XML4C) to parse two documents into in-memory DOM trees. Second, it computes signature and XHash value for every node on the two trees. We use the "UserData" pointer on each node, which is provided by DOM tree, to store both the signature and XHash value. X-Diff compares the XHash values of both roots to determine if two documents are different. If so, XML will compute the minimum-cost matching from one tree to the other. Based on the matching, X-Diff produces the *delta* tree, in which nodes are marked with "Insert", "Delete", and "Update".

Our program does not require two XML documents to conform the same DTD (Document Type Definition). In fact, DTD files are not necessary for change detection. However, if two XML documents conform to different DTDs, which are available, X-Diff can also detect change on DTD files. The details can be found in [24].

## 5.2 Performance

In this section, we study the performance of our algorithm. IBM's XMLTreeDiff [5] is the only other XML change detection tool that we are aware of. However, since XMLTreeDiff uses the ordered tree model, the performance of the two algorithms is not comparable. Thus, we only show some preliminary performance results obtained by running only our algorithm.

We test the X-Diff program on a Pentium® III 550 MHz PC with 256 MB memory. The operating System is RedHat Linux 6.2. We used the IBM XML4C 3.0.1 DOM parser.

### 5.2.1 Data Set

We obtain testing documents from an "Actors" database, whose DTD is shown in Figure 5.1. Notice that the unordered-tree model is appropriate for documents that conform to this DTD. For example, an actor's name can be placed either in front of his/her filmography or behind of it, which does not affect the content. The size of the documents used in our tests range from 10 KB to 1MB. Table 5.1 shows the number of nodes in each test file. The average node size is about 30 bytes, and the number of nodes is proportional to the size of document.

```
<?xml encoding="ISO-8859-1"?>
<!ELEMENT Actors (Actor)* >
<!ELEMENT Actor ( (Name, Filmography) | (Filmography, Name) )>
<!ELEMENT Name ( (FirstName, LastName) | (LastName, FirstName) )>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Filmography (Movie)*>
<!ELEMENT Movie ( (Title,Year) | (Year, Title) )>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
```

**Figure 5.1 DTD of testing XML documents**

| Size | No. of Nodes |
|---|---|
| 10KB | 315 |
| 50KB | 1646 |
| 100KB | 3731 |
| 500KB | 15990 |
| 1MB | 36006 |

**Table 5.1 Size of Documents**

We use a program to generate the second input document as the "new" version:

- *Identical.* The DOM tree of the new version is identical to that of the old version. Notice it does not require two documents to be identical.
- *Isomorphic.* The left-to-right order of some siblings is changed, but two DOM trees are isomorphic. We generate the new version by randomly permuting sibling nodes at every level.
- *Change.* We generate all three changes, "insert", "delete", and "update", randomly at each level. The program takes a parameter, the percentage of nodes being changed.

### 5.2.2 Impact of Data size and Change percentage

First, we examine the impact of data size and change percentage on the execution time. In Figure 5.2, which uses logarithmic scale on both axes, there are five curves. We can only see four curves because the first two, *Identical* and *Isomorphic*, overlap. In both cases, X-Diff can quickly determine whether two input documents are equivalent and most time is spent parsing two documents into DOM trees (recall Section 4.2).

For the three *Change* tests (1%, 5%, and 10%), X-Diff performs well on small- and medium-size documents. Due to the complexity of the algorithm, however, its execution time is fairly long when its input files are large. It suggests that users might want to use X-Diff to generate the optimal result only when the execution time is not a concern.

Notice when the change percentage increases, the execution time increases more and more slowly. This is consistent with our analysis on complexity in that the execution time mostly depends on the number of total nodes instead of the number of changed nodes.

All execution times in Figure 5.2 include the cost of parsing and hashing. Apparently, if two documents are equivalent, the parsing and hashing time is the dominant factor in X-Diff. In Figure 5.3, we show the decomposed execution time for the 1% change test. Hashing and parsing time is less than 20% of the total execution time except when documents are very small (see the 10KB case).

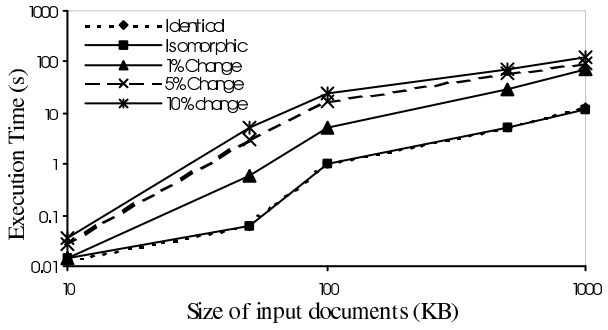


**Figure 5.2 Impact of data size and change percentage**

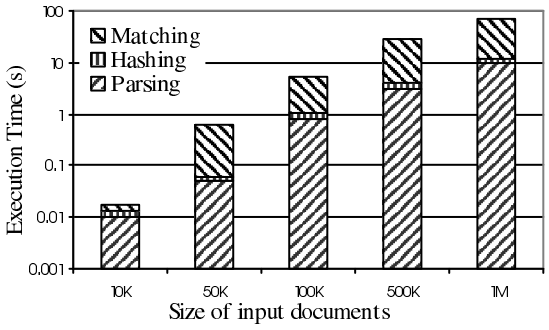


**Figure 5.3 Detailed execution time of 1% change case**

### 5.2.3 Impact of Change Locality

In the tests in the previous section, "changed" nodes were selected randomly, spanning different levels as well as all nodes at each level. In many situations, changes are concentrated in one region of the tree.

XHash helps by reducing the size of the input documents by removing equivalent second-level subtrees. In this section, we compare the performance of X-Diff with and without XHash. We generate the "new" version by restricting that changes only happen within 10% of second-level subtrees, i.e., only 10% of actors' information have been changed. The results in Figure 5.4 demonstrate that using XHash

significantly reduces the execution time of X-Diff.

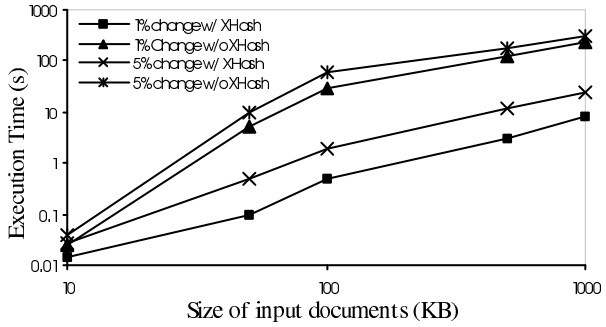


**Figure 5.4 Impact of change locality**

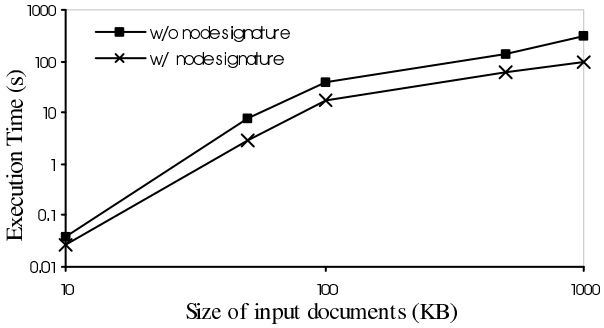


**Figure 5.5 Contribution of node *signature***

### 5.2.4 Contribution of Node Signature

In X-Diff, we introduce the notion of node *signature*, which is used to select nodes when searching for the minimum-cost matching from one tree to the other. It reduces the matching space largely because only nodes with same signature can be mapped. Figure 5.5 illustrates how its use improves the performance.

In the test documents, 5% of the nodes, randomly selected from the entire document, are changed. In Figure 5.5, one curve shows the execution time of regular X-Diff; the other curve shows the execution time of X-Diff without checking node signatures. There is a significant difference between the performance of the two methods, favoring the notion of node signature.

## 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 [4, 5] 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 technique [22], we propose a fast algorithm for computing the difference between two versions of an XML document. We present and analyze the algorithm, and also show some preliminary performance results. We have implemented X-Diff as a stand-alone tool and also used it as a component of an Internet query system. More details about X-Diff can be found in [24].

We are working on improving the performance of our algorithm. We also plan to develop a faster variant of the algorithm capable of producing "near optimal" results on very large documents. Other interesting future work includes change detection on XML data streams and incremental index update on change of XML documents.

# References


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

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

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

[4] 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, June 1996.

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

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML", *Proceeding of the Eighth International World Wide Web Conference*, Toronto, May 1999.

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

[8] 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, January 1998.

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

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

[11] H. Maruyama, 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.

[12] D. McArthur, "LSR Gene Expression RFI Response", *Rosetta Inpharmatics*, http://cgi.omg.org/cgi-bin/doc?lifesci/99-08-11, August 1999.

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

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

[15] J. Robie, J. Lapp, D. Schach, "XML Query Language (XQL)", *World Wide Web Consortium*, http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[16] J. Robie, D. Chamberlin, D. Florescu, "Quilt: an XML query language", *XML Europe 2000*, June 2000.

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

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

[19] 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.

[20] 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.

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

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

[23] Reference removed for double blind reviewing.

[24] Reference removed for double blind reviewing.