Change Detection in Hierarchically Structured Information*

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom

Department of Computer Science
Stanford University
Stanford, California 94305
{chaw,anand,hector,widom}@cs.stanford.edu

Abstract

Detecting and representing changes to data is important for active databases, data warehousing, view maintenance, and version and configuration management. Most previous work in change management has dealt with flat-file and relational data; we focus on hierarchically structured data. Since in many cases changes must be computed from old and new versions of the data, we define the hierarchical change detection problem as the problem of finding a "minimum-cost edit script" that transforms one data tree to another, and we present efficient algorithms for computing such an edit script. Our algorithms make use of some key domain characteristics to achieve substantially better performance than previous, general-purpose algorithms. We study the performance of our algorithms both analytically and empirically, and we describe the application of our techniques to hierarchically structured documents.

1 Introduction

We study the problem of detecting and representing changes to hierarchically structured information. Detecting changes to data (henceforth referred to as *deltas*) is a basic function of many important database facilities and applications, including active databases [WC96], data warehousing [HGMW+95, IK93, ZGMHW95], view maintenance [GM95], and version and configuration management [HKG+94].

For example, consider the World-Wide Web. A user may visit certain (HTML) documents repeatedly and is interested in knowing how each document has changed since the last visit. Assuming we have saved the old version of the document (which many web browsers do already for efficiency), we can detect the changes by comparing the old and new versions of the document. In addition, we are interested in presenting the changes in a meaningful way. For example, a paragraph that has moved could be marked with a "tombstone" in its old position and be highlighted in its new position. Similarly, insertions, deletions, and updates could be marked using changes in colors and fonts.

The work on change detection reported in this paper has four key characteristics:

- *Nested Information.* Our focus is on hierarchical information, not "flat" information (e.g., files containing records or relations containing tuples). With flat information deltas may be represented simply as sets of tuples or records inserted into, deleted from, and updated in relations [GHJ+93, LGM95]. In hierarchical information, we want to identify changes not just to the "nodes" in the data, but also to their relationships. For example, if a node (and its children) is moved from one location to another, we would like this to be represented as a "move" operation in the delta.

- *Object Identifiers Not Assumed.* For maximum generality we do not assume the existence of identifiers or keys that uniquely match information fragments across versions. For example, to compare structured documents, we must rely on values only since sentences or paragraphs do not come with identifying keys. Similarly, objects in two different design configurations may have to be compared by their contents, since object-ids may not be valid across versions. Of course, if the information we are comparing does have unique identifiers, then our algorithms can take advantage of them to quickly match fragments that have not changed.

- *Old, New Version Comparison.* Although some database systems, particularly active database systems, build change detection facilities into the system itself [WC96], we focus on the problem of detecting changes given old and new versions of the data.

*Research supported by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, by the Air Force Rome Laboratories under ARPA Contract F30602-95-C-0119, and by equipment grants from Digital and IBM Corporations.

We believe that a common scenario for change detection, especially for applications such as data warehousing, or querying and browsing over changes, involves "uncooperative" legacy databases (or other data management systems), where the best we can hope for is a sequence of data snapshots or "dumps" [HGMW+95, LGM95].

- *High Performance.* Our goal is to develop high performance algorithms that exploit features common to many applications and can be used on very large structures. In particular, [ZS89, SZ90] present algorithms that always find the most "compact" deltas, but are expensive to run, especially for large structures. (The running time is at least quadratic in the number of objects in each structure compared. The properties of these algorithms are described in more detail in Section 2.) Our algorithms are significantly more efficient (intuitively, our running time is proportional to the number of objects times the number of changes), but may sometimes find non-minimal, although still correct, deltas. We show that if the application domain has a certain property—very intuitively, that there are not "too many duplicate objects,"—then our algorithm also always generates minimal deltas. Our empirical studies suggest that this property does hold in practice.

To describe a delta between two versions of hierarchical data, we use the notion of a *minimum cost edit script*. The minimum cost edit script for two trees is defined using *node insert*, *node delete*, *node update*, and *subtree move* as the basic editing operations. An interesting feature of our approach is that there is a clean separation of the change detection problem into two subproblems: (1) finding a matching between objects in the two versions, and (2) computing an edit script. If objects have unique identifiers, the first problem is simplified, and we can use this property to achieve a speed-up.

Although a minimum cost edit script is a good formal notion of the delta between two trees, it is not always the most convenient method for displaying or querying deltas. We have developed a second, equivalent representation scheme called a *delta tree* for this purpose. Due to lack of space, we do not describe delta trees in this paper; they are described in [CRGMW95].

To demonstrate our approach and algorithms, we have implemented a system to detect, mark, and display changes in structured documents, based on their hierarchical structure. Our system, called *LaDiff*, takes two versions of a Latex document as input and produces as output a Latex document with the changes marked. We have used this system to experimentally evaluate the performance of our algorithms; results are presented in Section 6.2. We have also implemented our algorithms in change detection modules for HTML pages and for a simple nested-object model [PGMW95].

The remainder of the paper is organized as follows. We discuss related work in Section 2. Section 3 describes our general approach, divides our problem into two distinct subproblems, and provides preliminary definitions. Our algorithms for solving the two subproblems are discussed in Sections 4 and 5. Section 6 describes the application of our techniques to hierarchically structured documents and presents our empirical performance study. Conclusions and ongoing work are covered in Section 7. Due to space constraints, several details and proofs of theorems are not presented in this paper, and may be found in [CRGMW95].

2 Related Work

Most previous work in change management has dealt only with flat-file and relational data. For example, [LGM95] presents algorithms for efficiently comparing sets of records that have keys. The GNU *diff* utility compares flat text files by computing the LCS[1] of their lines using the algorithm described in [Mye86]. There are also a number of front-ends to this standard *diff* program that display the results of *diff* in a more comprehensible manner. (The *ediff* program [Kif95] is a good example.) However, since the standard *diff* program does not understand the hierarchical structure of data, such utilities suffer from certain inherent drawbacks. Given large data files with several changes, *diff* often mismatches regions of data. (For example, while comparing Latex files, an `item` is sometimes matched to a `section`, a sentence is sometimes matched to a Latex command, and so on.) Furthermore, these utilities do not detect moves of data—moves are always reported as deletions and insertions. Some commercial word processors have facilities for comparing documents and marking changes. For example, Microsoft Word has a "revisions" feature that can detect simple updates, inserts, and deletes of text. It cannot detect moves. WordPerfect has a "mark changes" facility that can detect some move operations. However, there are restrictions on how documents can be compared (on either a word, phrase, sentence, or paragraph basis). Furthermore, these approaches do not generalize to non-document data.

The general problem of finding the minimum cost edit distance between ordered trees has been studied in [ZS89]. Compared to the algorithm presented there, our algorithm is more restrictive in that we make some assumptions about the nature of the data being represented. Our algorithm always yields correct results, but if the assumptions do not hold it may produce sub-

[1] We define the Longest Common Subsequence (LCS) in Section 4.

optimal results. Because of our assumptions, we are able to design an algorithm with a lower running-time complexity. In particular, our algorithm runs in time $O(ne + e^2)$, where n is the number of tree leaves and e is the "weighted edit distance" (typically, $e \ll n$). The algorithm in [ZS89] runs in time $O(n^2 log^2 n)$ for balanced trees (even higher for unbalanced trees).[2] Our work also uses a different set of edit operations than those used in [ZS89]. The two sets of edit operations are equivalent in the sense that any state reachable using one set is also reachable using the other. A more detailed comparison of the two sets of edit operations is in [CRGMW95].

We believe our approach and that in [ZS89] are complementary; the choice of which algorithm to use depends on the domain characteristics. In an application where the amount of data is small (small tree structures), or where we are willing to spend more time (biochemical structures), the more thorough algorithm [ZS89] may be preferred. However, in applications with large amounts of data (object hierarchies, database dumps), or with strict running-time requirements, we would use our algorithm. The efficiency of our method is based on exploiting certain domain characteristics. Even in domains where these characteristics may not hold for all of the data, it may be preferable to get a quick, correct, but not guaranteed optimal, solution using our approach.

3 Overview and Preliminaries

In this section, we formulate the change detection problem and split it into two subproblems that are discussed in later sections. We first introduce these problems informally using an example, and then present the formal definitions and terms used in the rest of the paper.

Hierarchically structured information can be represented as *ordered trees*—trees in which the children of each node have a designated order. We address our problem of detecting and representing changes in the context of such trees. (Hereafter, when we use the term "tree" we mean an ordered tree.) We consider trees in which each node has a *label* and a *value*.[3] We also assume that each tree node has a unique identifier; identifiers may be generated by our algorithms when they are not provided in the data itself. Note that the nodes that represent the same real-world entity in different versions may not have the same identifier. We refer to the node with identifier x as "node x" for conciseness.

[2] Efficient parallel algorithms for unit-cost editing are presented in [SZ90], which also presents a uniprocessor variant that runs in time $O(e^2 n_1 min(n_1, n_2))$, where n_1 and n_2 are the tree sizes.

[3] We have found this label-value model to be useful for semistructured data in general [PGMW95]. We have defaults for the label and value of a node that does not specify them explicitly.

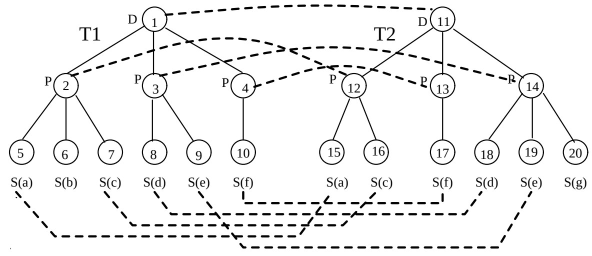

Figure 1: Running example

As a running example, consider trees T_1 and T_2 shown in Figure 1, and ignore the dashed lines for the moment. The number inside each node is the node's identifier and the letter beside each node is its label. All of the interior nodes have null values, not shown. Leaf nodes have the values indicated in parentheses. (These trees could represent two structured documents, where the labels `D`, `P`, and `S` denote `Document`, `Paragraph`, and `Sentence`, respectively. The values of the sentence nodes are the sentences themselves.) We are interested in finding the delta between these two trees. We will assume that T_1 represents the "old" data and T_2 the "new" data, so we want to determine an appropriate transformation from tree T_1 to tree T_2.

Our first task in finding such a transformation is to determine nodes in the two trees that correspond to one another. Intuitively, these are nodes that either remain unchanged or have their value updated in the transformation from T_1 to T_2 (rather than, say, deleting the old node and inserting a new one). For example, node 5 in T_1 has the same value as node 15 in T_2, so nodes 5 and 15 should probably correspond. Similarly, nodes 4 and 13 have one child node each, and the child nodes have the same value, so nodes 4 and 13 should probably correspond. The notion of a correspondence between nodes that have identical or similar values is formalized as a *matching* between node identifiers. Matchings are one-to-one. We say that a matching is *partial* if only some nodes in the two trees participate, while a matching is *total* if all nodes participate. Hereafter, we use the term "matching" to mean a partial matching unless stated otherwise.

Hence, one of our problems is to find an appropriate matching for the trees we are comparing. We call this problem the *Good Matching* problem. In some application domains the Good Matching problem is easy, such as when data objects contain object identifiers or unique keys. In other domains, such as structured documents, the matching is based on labels and values only, so the Good Matching problem is more difficult. Furthermore, not only do we want to match nodes that are identical (with respect to the labels and values of the nodes and their children), but we also want to match nodes that are "approximately equal." For instance, node 3 in Figure 1 probably should match node 14 even though node 3 is missing one of the children of 14. Details of the Good Matching problem—including what constitutes a "good" matching—are addressed in Section 5. A matching for our running example is illustrated by the dashed lines in Figure 1.

We say that two trees are *isomorphic* if they are identical except for node identifiers. For trees T_1 and T_2, once we have found a good (partial) matching M, our next step is to find a sequence of "change operations" that transforms tree T_1 into a tree T_1' that is isomorphic to T_2. Changes may include inserting (leaf) nodes, deleting nodes, updating the values of nodes, and moving nodes along with their subtrees. Intuitively, as T_1 is transformed into T_1', the partial matching M is extended into a total matching M' between the nodes of T_1' and T_2. The total matching M' then defines the isomorphism between trees T_1' and T_2. We call the sequence of change operations an *edit script*, and we say that the edit script *conforms* to the original matching M provided that $M' \supseteq M$. (As will be seen, an edit script conforms to partial matching M as long as the script does not insert or delete nodes participating in M.) Edit scripts are defined in more detail shortly.

We would like our edit script to transform tree T_1 as little as possible in order to obtain a tree isomorphic to T_2. To capture minimality of transformations, we introduce the notion of the *cost* of an edit script, and we look for a script of minimum cost. Thus, our second main problem is the problem of finding such a minimum cost edit script; we refer to this as the *Minimum Conforming Edit Script (MCES)* problem. The remainder of this section formally defines edit operations and edit scripts. Our algorithm for the MCES problem is presented in Section 4, and Section 5 presents our algorithm for the Good Matching problem. Note that we consider the MCES problem before the Good Matching problem, despite the fact that our method requires finding a matching before generating an edit script. As will be seen, the definition of a good matching relies on certain aspects of edit scripts, so for presentation purposes we consider the details of our edit script algorithms first.

3.1 Edit Operations

In an ordered tree, if nodes $v_1, \ldots, v_m$ are the children of node u, then we call v_i the ith child of u. For a node x, we let $l(x)$ denote the label of x, $v(x)$ denote the value of x, and $p(x)$ denote the parent of x if x is not the root. We assume that labels are chosen from a fixed but arbitrary set. In the definitions of the edit operations, T_1 refers to the tree on which the operation is applied, while T_2 refers to the resulting tree. The four edit operations on trees are the following:

Insert: The *insertion* of a new leaf node x into T_1, denoted by $\textsc{ins}((x, l, v), y, k)$. A node x with label l and value v is inserted as the kth child of node y of T_1. More precisely, if $u_1, \ldots, u_m$ are the children of y in T_1, then $1 \leq k \leq m+1$ and $u_1, \ldots, u_{k-1}, x, u_k, \ldots, u_m$ are the children of y in T_2. The value v is optional, and is assumed to be null if omitted.

Delete: The *deletion* of a leaf node x of T_1, denoted by $\textsc{del}(x)$. The result T_2 is the same as T_1, except that it does not contain node x. $\textsc{del}(x)$ does not change the relative ordering of the remaining children of $p(x)$. This operation deletes only a leaf node; to delete an interior node, we must first move its descendents to their new locations or delete them.

Update: The *update* of the value of a node x in T_1, denoted by $\textsc{upd}(x, val)$. T_2 is the same as T_1 except that in T_2, $v(x) = val$.

Move: The *move* of a subtree from one parent to another in T_1, denoted by $\textsc{mov}(x, y, k)$. T_2 is the same as T_1, except x becomes the kth child of y. The entire subtree rooted at x is moved along with x.

Figure 2 shows examples of edit operations on trees. In the figure, node 6 has label A and value *foo*. The labels and values of the other nodes are not shown.

3.2 Edit Scripts

Informally, an edit script gives a sequence of edit operations that transforms one tree into another. Formally, we say $T_1 \xrightarrow{e_1} T_2$ when T_2 is the result of applying the edit operation e_1 to T_1. Given a sequence $E = e_1, \ldots, e_m$ of edit operations, we say $T_1 \xrightarrow{E} T_{m+1}$ if there exist $T_2, \ldots, T_m$ such that $T_1 \xrightarrow{e_1} T_2 \xrightarrow{e_2} \ldots \xrightarrow{e_m} T_{m+1}$. A sequence E of edit operations *transforms* T_1 into T_2 if $T_1 \xrightarrow{E} T_1'$ and T_1' is isomorphic to T_2. (Recall that two trees are isomorphic if they differ only in the identifiers of their nodes.) We call such a sequence of edit operations an *edit script of* T_1 *with respect to* T_2. Notice that an edit script does not tell us how the original matching between T_1 and T_2 should be modified to obtain the total matching between T_1' and T_2. This will be done as the edit script is generated; see Section 4.

Example 3.1 Consider the trees T_1 and T_2 shown in Figure 3. The following edit script below transforms T_1 into T_2:

$$\textsc{ins}((11, \mathit{Sec}, \mathit{foo}), 1, 4), \textsc{mov}(5, 11, 1), \textsc{del}(2), \textsc{upd}(9, \mathit{baz})$$

Figure 3 also shows the intermediate trees in the transformation specified by the above edit script. (The last update is not shown in order to save space.)

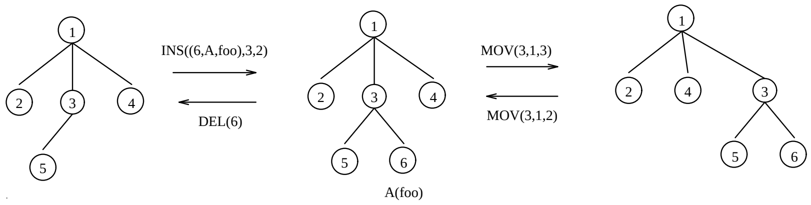

Figure 2: Edit operations on a tree

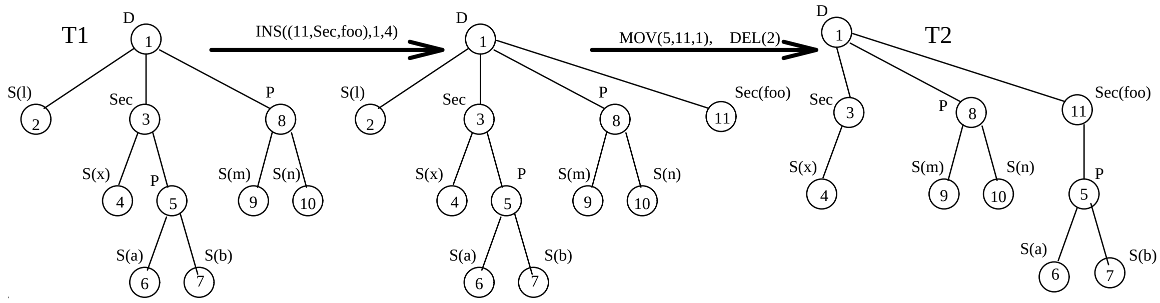

Figure 3: Applying the edit script of Example 3.1

3.3 A Cost Model for Edit Scripts

Given two trees, in general there are many edit scripts that transform one tree to the other. Even when an edit script must conform to a given matching, there may be many correct scripts. (Recall that we defined the concept of an edit script conforming to a matching in Section 3.) For example, the following edit script, when applied to the initial tree in Example 3.1, produces the same final tree as that produced by the edit script in the example:

$\text{INS}((11, \mathit{Sec}, \mathit{foo}), 1, 4), \text{DEL}(6), \text{DEL}(7), \text{DEL}(5),$
$\text{INS}((12, S, a), 11, 1), \text{INS}((13, S, b), 11, 2), \text{UPD}(9, \mathit{baz})$

Intuitively, this edit script does more work than necessary, and is thus an undesirable representation of the delta between the trees. To formalize this idea, we introduce the *cost* of an edit script.

We first define the costs of edit operations and then use these costs to define the cost of edit scripts. The cost of an edit operation depends on the type of operation and the nodes involved in the operation. Let $c_D(x)$, $c_I(x)$, and $c_U(x)$ denote respectively the cost of deleting, inserting, and updating node x, and let $c_M(x)$ denote the cost of moving the subtree rooted at node x. In general, these costs may depend on the label and the value of x, as well as its position in the tree. In this paper, we adopt a simple cost model where deleting and inserting a node, as well as moving a subtree, are considered to be unit cost operations. That is, $c_D(x) = c_I(x) = c_M(x) = 1$ for all x.

Now consider the cost $c_U(x)$ of updating the value of a node x. We assume that this cost is given by a function, *compare*, that evaluates how different x's old value v is from its new value v'. This *compare* function takes two nodes as arguments and returns a number in the range $[0, 2]$. Although the nature of the *compare* function is arbitrary, it should be consistent with the costs of the other edit operations in the following sense: Suppose x is moved, and its value v is updated so that v is very similar to v'. Then $\mathit{compare}(v, v')$ should be less than 1, so that the cost of moving and updating x is less than the cost of deleting x and replacing it with a new node with value v'. If v and v' are very different, we would rather have the edit script contain a delete/insert pair, so the update cost should be greater than 1. Finally, the cost of an edit script is the sum of the costs of its individual operations.

4 Generating the Edit Script

In this section we consider the *Minimum Conforming Edit Script* problem, motivated in the previous section. The problem is stated as follows. Given a tree T_1 (the *old tree*), a tree T_2 (the *new tree*), and a (partial) matching M between their nodes, generate a minimum cost edit script that conforms to M and transforms T_1 to T_2. Our algorithm starts with an empty edit script E and appends edit operations to E as it proceeds. To explain the working of the algorithm, we apply each edit operation to T_1 as it is added to E. When the algorithm terminates, we will have transformed T_1 into a tree that is isomorphic to T_2. In addition, the algorithm extends the given partial matching M by adding new pairs of nodes to M as it adds operations to E. When the algorithm terminates, M is a total matching between the nodes of T_1 and T_2.

4.1 Outline of Algorithm

The algorithm is most easily explained as consisting of the five phases that we describe below. We use our running example from Figure 1. We are required to find a minimum cost edit script that transforms T_1 into T_2, given the matching M shown by the dashed lines in the figure.

The Update Phase: In the update phase, we look for pairs of nodes $(x, y) \in M$ such that the values at nodes x and y differ. For each such pair (in any order) we add the edit operation $\text{UPD}(x, v(y))$ to E (recall that for a node x, $v(x)$ denotes the value of x), and we apply the update operation to T_1. At the end of the update phase, we have transformed T_1 such that $v(x) = v(y)$ for every pair of nodes $(x, y) \in M$.

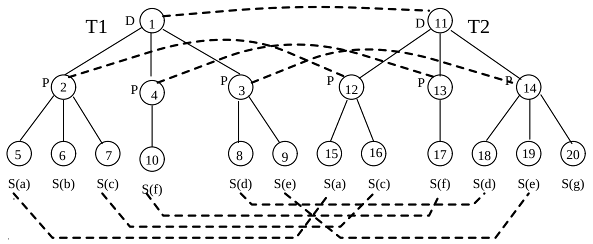

Figure 4: Running example: after align phase

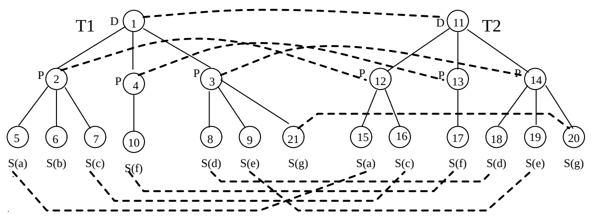

Figure 5: Running example: after insert phase

The Align Phase: Let the *partner* of a node denote the node to which it is matched (by a given matching). Suppose $(x, y) \in M$. We say that the children of x and y are *misaligned* if x has matched children u and v such that u is to the left of v in T_1 but the partner of u is to the right of the partner of v in T_2. In Figure 1, the children of the root nodes 1 and 11 are misaligned. In the align phase we check each pair of matched internal nodes $(x, y) \in M$ (in any order) to see if their children are misaligned. If we find that the children are misaligned, we append move operations to E to align the children. We explain how the move operations are determined in Section 4.2 below. In our running example, we append $\text{MOV}(4, 1, 2)$ to E, and we apply the move operation to T_1. The new T_1 is shown in Figure 4.

The Insert Phase: We assume, without loss of generality, that the roots of T_1 and T_2 are matched in M.[4] In the insert phase, we look for an unmatched node $z \in T_2$ such that its parent is matched. Suppose $y = p(z)$ (i.e., y is the parent of z) and y's partner in T_1 is x. We create a new identifier w and append $\text{INS}((w, l(z), v(z)), x, k)$ to E. The position k is determined with respect to the children of x and z that have already been aligned with respect to each other; details are in Section 4.3. We also apply the insert operation to T_1 and add (w, z) to M. In our running example we append $\text{INS}((21, S, g), 3, 3)$. The transformed T_1 and the augmented M are shown in Figure 5. At the end of the insert phase, every node in T_2 is matched but there may still be nodes in T_1 that are unmatched.

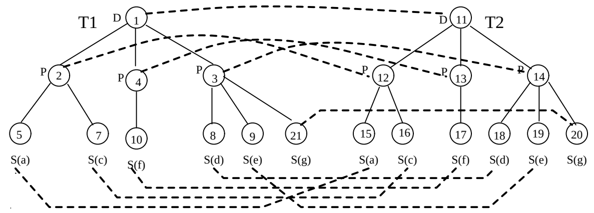

Figure 6: Running example: after delete phase

The Move Phase: In the move phase we look for pairs of nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$. (Recall from Section 3.1 that $p(x)$ denotes the parent of x.) Suppose $v = p(y)$. We know that at the end of the insert phase, v has some partner u in T_1. We append the operation $\text{MOV}(x, u, k)$ to E, and we apply the move operation to T_1. Here the position k is determined with respect to the children of u and v that have already been aligned, as in the insert phase. At the end of the move phase T_1 is isomorphic to T_2 except for unmatched nodes in T_1. In our running example, we do not need to perform any actions in this phase.

The Delete Phase: In the delete phase we look for unmatched leaf nodes $x \in T_1$. For each such node we append $\text{DEL}(x)$ to E and apply the delete operation to T_1. (Note that this process will result in a bottom-up delete—descendents will be deleted before their ancestors.) At the end of the delete phase T_1 is isomorphic to T_2, E is the final edit script, and M is the

[4] If the roots of T_1 and T_2 are not matched in M, then we add new dummy roots that are matched.

total matching to which E conforms. Figure 6 shows the trees and the matching after the delete phase.

4.2 Aligning Children

The align phase of the edit script algorithm presents an interesting problem. Suppose we detect that for $(x, y) \in M$, the children of x and y are misaligned. In general, there is more than one sequence of moves that will align the children. For instance, in Figure 7 there are at least two ways to align the children of nodes 1 and 11. The first consists of moving nodes 2 and 4 to the right of node 6, and the second consists of moving nodes 3, 5, and 6 to the left of node 2. Both yield the same final configuration, but the first one is better since it involves fewer moves.

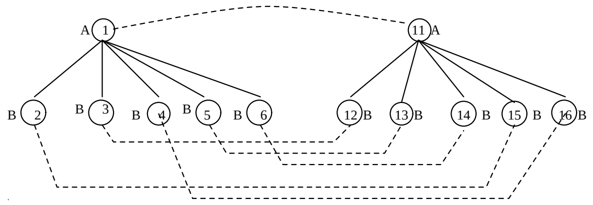

Figure 7: A matching with misaligned nodes

To ensure that the edit script generated by the algorithm is of minimum cost, we must find the shortest sequence of moves to align the children of x and y. Our algorithm for finding the shortest sequence of moves is based on the notion of a *longest common subsequence*, described next.

Given a sequence $S = a_1a_2 \ldots a_n$, a sequence S' is a *subsequence* of S if it can be obtained by deleting zero or more elements from S. That is, $S' = a_{i_1} \ldots a_{i_m}$ where $1 \leq i_1 < i_2 < \ldots < i_m \leq n$. Given two sequences S_1 and S_2, a *longest common subsequence (LCS)* of S_1 and S_2, denoted by $LCS(S_1, S_2)$, is a sequence $S = (x_1, y_1) \ldots (x_k, y_k)$ of pairs of elements such that (1) $x_1 \ldots x_k$ is a subsequence of S_1; (2) $y_1 \ldots y_k$ is a subsequence of S_2; (3) for $1 \leq i \leq k$, $equal(x_i, y_i)$ is true for some predefined equality function *equal*; and (4) there is no sequence S' that satisfies conditions 1, 2, and 3 and is longer than S. The length of an LCS of S_1 and S_2 is denoted by $|LCS(S_1, S2)|$.

We use an algorithm due to Myers [Mye86] that computes an LCS of two sequences in time $O(ND)$, where $N = |S_1| + |S_2|$ and $D = N - 2|LCS(S_1, S_2)|$. We treat Myers' LCS algorithm as having three inputs: the two sequences S_1 and S_2 to be compared, and an equality function $equal(x, y)$ used to compare $x \in S_1$ and $y \in S_2$ for equality. That is, we treat it as the procedure $LCS(S_1, S_2, equal)$.

The solution to the alignment problem is now straightforward. Compute an LCS S of the matched children of nodes x and y, using the equality function $equal(u, v)$ that is true if and only if $(u, v) \in M$. Leave the children of x that are in S fixed, and move the remaining matched children of x to the correct positions relative to the already aligned children. In Figure 7, the LCS is $3, 5, 6$ (matching the sequence $12, 13, 14$). The moves generated are $\text{MOV}(2, 1, 5)$ and $\text{MOV}(4, 1, 5)$. In [CRGMW95], we show that our LCS-based strategy always leads to the minimum number of moves.

4.3 The Complete Algorithm

We now present the complete algorithm to compute a minimum cost edit script E conforming to a given matching M between trees T_1 and T_2. In the algorithm, we combine the first four phases of Section 4.1 (the update, insert, align, and move phases) into one breadth-first scan on T_2. The delete phase requires a post-order traversal of T_1 (which visits each node after visiting all its children). The order in which the nodes are visited and the edit operations are generated is crucial to the correctness of the algorithm. (For example, an insert may need to precede a move, if the moved node becomes the child of the inserted node.) The algorithm applies the edit operations to T_1 as they are appended to the edit script E. When the algorithm terminates, T_1 is isomorphic to T_2. The algorithm also uses a matching M' that is initially M, and adds matches to it so that M' is a total matching when the algorithm terminates. As mentioned earlier, we assume without loss of generality that the roots of T_1 and T_2 are matched in M.

The algorithm is shown in Figure 8. It uses two procedures, *AlignChildren* and *FindPos*, shown in Figure 9. The claims made by the two statements in Algorithm *EditScript* that are marked with (*) are substantiated in [CRGMW95], where it is also proved that Algorithm *EditScript* generates a minimum cost edit script conforming to the given matching M.

Let us now consider the running time of this algorithm. We first define the notion of *misaligned nodes*. Suppose $x \in T_1$ and $y = p(x)$. A move of the form $M(x, y, k)$ for some k is called an *intra-parent move* of node x; such moves are generated in the align phase of the algorithm. The number of misaligned nodes of T_1 with respect to T_2 is the minimum number of intra-parent moves among all minimum cost edit scripts. We can show [CRGMW95] that the running time of Algorithm *EditScript* is $O(ND)$, where N is the total number of nodes in T_1 and T_2 and D is the total number of misaligned nodes. (Note that D is typically much smaller than N.)

5 Finding Good Matchings

In this section we consider the *Good Matching* problem, motivated in Section 3. We want to find an appropriate

1. $E \leftarrow \epsilon$, $M' \leftarrow M$
2. Visit the nodes of T_2 in breadth-first order
   /* *combines the update, insert, align, and move phases* */
   (a) Let x be the current node in the breadth-first search of T_2 and let $y = p(x)$. Let z be the partner of y in M'. (*)
   (b) If x has no partner in M'
      i. $k \leftarrow FindPos(x)$
      ii. Append $\text{INS}((w, a, v(x)), z, k)$ to E, for a new identifier w.
      iii. Add (w, x) to M' and apply $\text{INS}((w, a, v(x)), z, k)$ to T_1.
   (c) else if x is not the root /* *x has a partner in M'* */
      i. Let w be the partner of x in M', and let $v = p(w)$ in T_1.
      ii. If $v(w) \neq v(x)$
         A. Append $\text{UPD}(w, v(x))$ to E.
         B. Apply $\text{UPD}(w, v(x))$ to T_1.
      iii. If $(y, v) \notin M'$
         A. Let z be the partner of y in M'. (*)
         B. $k \leftarrow FindPos(x)$
         C. Append $\text{MOV}(w, z, k)$ to E.
         D. Apply $\text{MOV}(w, z, k)$ to T_1.
   (d) $AlignChildren(w, x)$
3. Do a post-order traversal of T_1. /* *the delete phase* */
   (a) Let w be the current node in the post-order traversal of T_1.
   (b) If w has no partner in M' then append $\text{DEL}(w)$ to E and apply $\text{DEL}(w)$ to T_1.
4. E is a minimum cost edit script, M' is a total matching, and T_1 is isomorphic to T_2.

Figure 8: Algorithm *EditScript*

Function $AlignChildren(w, x)$

1. Mark all children of w and all children of x "out of order."
2. Let S_1 be the sequence of children of w whose partners are children of x and let S_2 be the sequence of children of x whose partners are children of w.
3. Define the function $equal(a, b)$ to be true if and only if $(a, b) \in M'$.
4. Let $S \leftarrow LCS(S_1, S_2, equal)$.
5. For each $(a, b) \in S$, mark nodes a and b "in order."
6. For each $a \in S_1$, $b \in S_2$ such that $(a, b) \in M$ but $(a, b) \notin S$
   (a) $k \leftarrow FindPos(b)$.
   (b) Append $\text{MOV}(a, w, k)$ to E and apply $\text{MOV}(a, w, k)$ to T_1.
   (c) Mark a and b "in order."

Function $FindPos(x)$

1. Let $y = p(x)$ in T_2 and let w be the partner of x ($x \in T_1$).
2. If x is the leftmost child of y that is marked "in order," return 1.
3. Find $v \in T_2$ where v is the rightmost sibling of x that is to the left of x and is marked "in order."
4. Let u be the partner of v in T_1.
5. Suppose u is the ith child of its parent (counting from left to right) that is marked "in order." Return $i + 1$.

Figure 9: Functions *AlignChildren* and *FindPos*

matching between the nodes of trees T_1 and T_2 that can serve as input to Algorithm *EditScript*. In applications in which the data has object-ids or keys, we can match nodes using these object-ids or keys. However, as described in Section 1, our focus here is on applications where information may not have keys or object-ids that can be used to match "fragments" of objects in one version with those in another. We use the term *keyless data* for hierarchical data that may not have identifying keys or object-ids.

When comparing versions of keyless data, there may be more than one way to match objects. Thus we need to define *matching criteria* that a matching must satisfy to be considered "good" or appropriate. In general, the matching criteria will depend on the domain being considered. One way of evaluating matchings that is desirable in many situations is to consider the minimum cost edit scripts that conform to the matchings (and transform T_1 into T_2). Intuitively, a matching that allows us to transform one tree to the other at a lower cost is a better matching. Formally, for matchings M and M', we say that M *is better than* M' if a minimum cost edit script that conforms to M is cheaper than a minimum cost edit script that conforms to M'. Our goal is to find a *best matching*, that is, a matching M that satisfies the given matching criteria and such that there is no better matching M' that also satisfies the criteria.

Unfortunately, if our matching criterion only requires that matched nodes have the same label, then finding the best matching has two difficulties. The first difficulty is that many matchings that satisfy only this trivial matching criterion may be unnatural in certain domains. For example, when matching documents, we may only want to match textual units (paragraphs, sections, subsections, etc.) that have more than a certain percentage of sentences in common. The second difficulty is one of complexity: the only algorithm known to us to compute the best matching as defined above (based on post-processing the output of the algorithm in [ZS89]) runs in time $O(n^2)$ where n is the number of tree nodes [Zha95]. To solve the first difficulty, we restrict the set of matchings we consider by introducing stronger matching criteria, as described below. These criteria also permit us to design efficient algorithms for matching. In the rest of this section, we describe some matching criteria for keyless data, using structured documents as an example.

5.1 Matching Criteria for Keyless Data

Our goal in this section is to augment the trivial label-matching criterion with additional criteria that simultaneously yield matchings that are meaningful in the domains of the data being considered, and that make possible efficient algorithms to compute a best matching.

Our first matching criterion states that nodes that are "too dissimilar" may not be matched with each other. For leaf nodes, this condition is stated as follows.

Matching Criterion 1 For leaf nodes $x \in T_1$ and $y \in T_2$, (x, y) can be in a matching only if $l(x) = l(y)$ and $compare(v(x), v(y)) \leq f$ for some parameter f such that $0 \leq f \leq 1$. (Recall that $l(x)$ and $v(x)$ denote the label and value of node x, and that *compare* is defined in Section 3.3 as a function used for determining the cost of updating a leaf node.) □

We also want to disallow matching internal nodes that do not have much in common. Here a more natural notion than the value (which is often null in the label-value model) is the number of common descendants. Let us say that an internal node x *contains* a node y if y is a leaf node descendent of x, and let $|x|$ denote the number of leaf nodes x contains. The following constraint allows internal nodes x and y to match only if at least a certain percentage of their leaves match (where t is a parameter):

Matching Criterion 2 Consider a matching M containing (x, y), where x is an internal node in T_1 and y is an internal node in T_2. Define $common(x, y) = \{(w, z) \in M \mid x \text{ contains } w, \text{and } y \text{ contains } z\}$. Then in M we must have $l(x) = l(y)$ and $\frac{|common(x,y)|}{\max(|x|,|y|)} > t$ for some t satisfying $\frac{1}{2} \leq t \leq 1$. □

Recall from Section 1 that one of the features of our work is that we use domain characteristics to design efficient algorithms. We now introduce these domain characteristics and formalize them by stating two assumptions that they let us make.

The hierarchical keyless data we are comparing has labels, and these labels usually follow a *structuring schema*, such as the one defined in [ACM95]. Many structuring schemas satisfy an *acyclic labels* condition, formalized in the following assumption:

Assumption 1 There is an ordering $<_l$ on the labels in the schema such that a node with label l_1 can appear as the descendent of a node with label l_2 only if $l_1 <_l l_2$.

In schemas where this condition is not satisfied, we can use domain semantics to merge labels that form a cycle, so that the resulting schema satisfies this condition [CRGMW95].

Our next assumption states (informally) that the *compare* function is a good discriminator of leaves. It states that given any leaf s in the old document, there is at most one leaf in the new document that is "close" to s, and vice versa. For example, consider a world-wide web "movie database" source listing movies, actors, directors, etc. A tree representation of this data may contain movie titles as leaves. This assumption says that, when comparing two snapshots of this data, a movie title in one snapshot may "closely resemble" at most one movie title in the other.

Assumption 2 For any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $compare(v(x), v(y)) \leq 1$. Similarly, for any leaf $y \in T_2$, there is at most one leaf $x \in T_1$ such that $compare(v(x), v(y)) \leq 1$. □

This assumption may not hold for some domains. For example, legal documents may have many sentences that are almost identical. The algorithms we describe below are guaranteed to produce an optimal matching when Assumption 2 holds. When Assumption 2 does not hold, our algorithm may generate a suboptimal, but still correct, solution. However, we can often post-process such a suboptimal solution to obtain an optimal solution. We discuss this issue further in Section 6.3.

Matching Criteria 1 and 2 and the assumptions that we have introduced above allow us to simplify the best matching problem as follows. (Recall that a best matching is a matching that can be used to produce an edit script of the lowest cost among all matchings satisfying the Matching Criteria.) We say that a matching is *maximal* if it is not possible to augment it without violating the Matching Criteria. We can show that our Matching Criteria imply that there is a unique maximal matching. Furthermore, given our assumptions, we can show that this unique maximal matching is also the best matching. These statements are formalized in the following theorem, which is proven in [CRGMW95]:

Theorem 5.1 (Unique Maximal Matching) If T_1 and T_2 are trees satisfying Matching Criteria 1 and 2 and Assumptions 1 and 2, then there is a unique maximal matching M of the nodes of T_1 and T_2. Moreover, M is also the unique best matching that satisfies the matching criteria.

5.2 A Matching Algorithm

Theorem 5.1 allows us to design a simple algorithm for computing the best matching. This algorithm, called algorithm *Match*, is presented in [CRGMW95], and runs in quadratic time. Below, we present a faster algorithm, called *FastMatch*, for computing the unique maximal matching. Our algorithm uses a function *equal* to compare nodes. For leaf nodes, $equal(x, y)$ is true if and only if $l(x) = l(y)$ and $compare(v(x), v(y)) \leq f$, where f is a parameter valued between 0 and 1. For internal nodes, $equal(x, y)$ is true if and only if $l(x) = l(y)$ and $\frac{|common(x,y)|}{\max(|x|,|y|)} > t$, where $t > 0.5$ is a parameter.

Figure 10 shows Algorithm *FastMatch*, which uses the longest common subsequence (LCS) routine, introduced earlier in Section 4.2, to perform an initial matching of nodes that appear in the same order. Nodes still unmatched after the call to LCS are processed using linear search. We assume that all nodes with a given label l in tree T are chained together from left to right. Let $chain_T(l)$ denote the chain of nodes with label l in tree T. Node x occurs to the left of node y in $chain_T(l)$ if x appears before y in the in-order traversal of T when siblings are visited left-to-right.

1. $M \leftarrow \phi$
2. For each leaf label l do
   (a) $S_1 \leftarrow chain_{T_1}(l)$.
   (b) $S_2 \leftarrow chain_{T_2}(l)$.
   (c) $lcs \leftarrow LCS(S_1, S_2, equal)$.
   (d) For each pair of nodes $(x, y) \in lcs$, add (x, y) to M.
   (e) For each unmatched node $x \in S_1$, if there is an unmatched node $y \in S_2$ such that $equal(x, y)$ then
      A. Add (x, y) to M.
      B. Mark x and y "matched."
3. Repeat steps 2a–2e for each internal node label l.

Figure 10: Algorithm *FastMatch*

Now we analyze the running time of Algorithm *FastMatch*. Define the *weighted edit distance* e between trees T_1 and T_2 as follows. Let $E = e_1e_2 \ldots e_n$ be the shortest edit script that transforms T_1 to T_2. Then the weighted edit distance is given by $e = \sum_{1 \leq i \leq n} w_i$ where w_i, for $1 \leq i \leq n$, is 1 if e_i is an insert or a delete, $|x|$ if e_i is a move of the subtree rooted at x, and 0 otherwise. Recall that $|x|$ denotes the number of leaf nodes that are descendants of node x. Intuitively, the weighted edit distance indicates how different the two trees are "structurally," where the degree of difference associated with the move of a subtree depends on the number of leaves in the subtree. In [CRGMW95] we show that the running time of Algorithm *FastMatch* is proportional to $(ne + e^2)c + 2lne$ where n is the total number of leaf nodes, c is the average cost of comparing two leaves (using the *compare* function), l is the number of internal node labels, and e is the weighted edit distance between T_1 and T_2.

6 Implementation and Performance

To validate our method for computing and representing deltas, as well as to have a vehicle for studying the performance of our algorithms, we have implemented a program for computing and representing changes in structured documents. In Section 6.1, we describe the implementation of this program, called *LaDiff*. In Section 6.2, we study the running time of FastMatch, and in Section 6.3, we discuss the effect of the Assumption 2 of Section 5 on the quality of the solution produced by FastMatch.

6.1 Implementation

In the following description, we focus on Latex documents, but the implementation also handles other kinds of structured documents (e.g., HTML). *LaDiff* takes as input two files containing the old and new versions of a Latex document. These files are first parsed to produce their tree representations (the old tree and new tree, respectively). Currently, we parse a subset of Latex consisting of sentences, paragraphs, subsections, sections, lists, items, and document. It is easy to extend our parser to handle a larger subset of Latex, and we plan to do so in the future. Next, the edit script and delta tree are computed using the algorithms of Sections 4–5. Our program takes the match threshold t (Section 5) as a parameter. Our comparison function for leaf nodes—which are sentences—first computes the LCS (recall Section 4.2) of the words in the sentences, then counts the number of words not in the LCS. Interior nodes (paragraphs, items, sections, etc.) are compared as described in Section 5. Finally, a preorder traversal of the delta tree is performed to produce an output Latex document with annotations describing the changes. Our implementation uses a modified version of the LCS algorithm from [Mye86]. Note that we cannot use the LCS algorithm used by the standard UNIX *diff* program, because it requires inequality comparisons in addition to equality comparisons.

6.2 Empirical Evaluation of FastMatch

Recall from Section 5 that the running time of Algorithm *FastMatch* is given by an expression of the form $r_1c + r_2$. In this expression, r_1 represents the number of leaf node comparisons (invocations of function *compare*), c is the average cost of comparing leaf nodes, and r_2 represents the number of node partner checks. Partner checks are implemented in *LaDiff* as integer comparisons. We know that r_1 is bounded by $(ne+e^2)$, and that r_2 is bounded by $2lne$, where n is the number of tree nodes, e is the weighted edit distance between the two trees, and l is the number of internal node labels. The parameter e depends on the nature of the differences between the trees (recall the definition of weighted edit distance in Section 5.2).

There are two reasons for studying the performance of FastMatch empirically. The first reason is that the formula for the running time contains the weighted edit distance, e, which is difficult to estimate in terms of the input. A more natural measure of the input size is the number of edit operations in an optimal edit script, which we call the *unweighted* edit distance, d. We can show analytically that the ratio e/d is bounded by $\log n$ for a large class of inputs, but we believe that in real cases, its value is much lower than $\log n$. We therefore study the relationship between e and d empirically. The second reason is that we would like to test our conjecture that the analytical bound on the running time of FastMatch is "loose," and in most practical situations the algorithm runs much faster.

For our performance study, we used three sets of files.

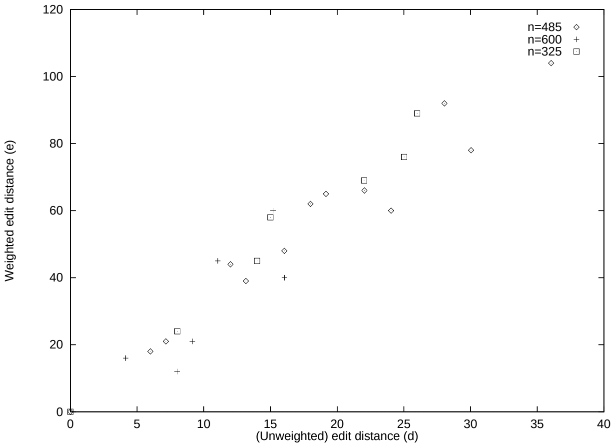

Figure 11: Weighted edit distance

The files in each set represent different versions of a document (a conference paper). We ran FastMatch on pairs of files within each of these three sets. (Comparing files across sets is not meaningful because we would be comparing two completely different documents.) In Figure 11 we indicate how the weighted edit distance (e) varies with the unweighted edit distance (d), for each of the three document sets. Recall that n is the number of tree leaves, that is, the number of sentences in the document. We see that the relationship between e and d is close to linear. Furthermore, the variance with respect to the three document sets is not high, suggesting that e/d is not very sensitive to the size of the documents (n). The average value of e/d is 3.4 for these documents.

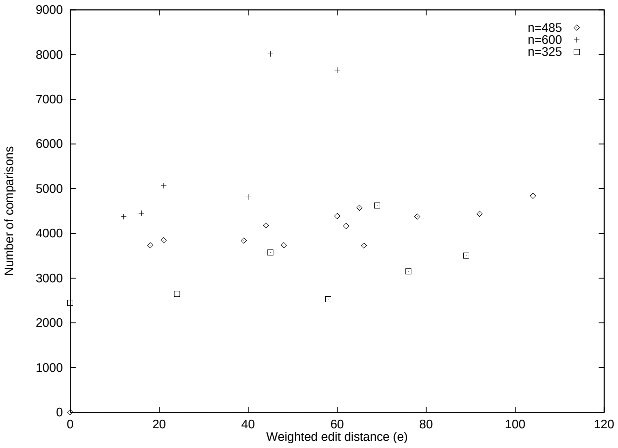

Figure 12: Running time

In Figure 12 we plot how the running time of FastMatch varies with the weighted edit distance e. The vertical axis is the running time as measured by the number of comparisons made by FastMatch and the horizontal axis is the weighted edit distance. Note that the analytical bound on the number of comparisons made by FastMatch is much higher than the numbers depicted in Figure 12; on the average, FastMatch makes approximately 20 times fewer comparisons than those predicted by the analytical bound, supporting our conjecture that the analytical bound on the running time is a loose one. We also observe that Figure 12 suggests an approximately linear relation between the running time and e, although there is a high variance. This variance may be explained by our first observation that the actual running time is far below the predicted bound.

6.3 Quality of FastMatch's matching

Another issue that needs to be addressed is the effect of Assumption 2 on the quality of the solution produced by FastMatch. Recall from Section 5 that FastMatch is guaranteed to produce an optimal matching only when Assumption 2 holds. When Assumption 2 does not hold, the algorithm may produce a suboptimal matching. We describe a post-processing step that, when added to FastMatch, enables us to convert the possibly suboptimal matching produced by FastMatch into an optimal one in many cases: Proceeding top-down, we consider each tree node x in turn. Let y be the partner of x according to the current matching. For each child c of x that is matched to a node c' such that $parent(c') \neq y$, we check if we can match c to a child c'' of y, such that $compare(c, c'') \leq f$, where f is the parameter used in Matching Criterion 1. If so, we change the current matching to make c match c''. This post-processing phase removes some of the suboptimalities that may be introduced if Assumption 2 does not hold for all nodes.

Even with post-processing, it is still possible to have a suboptimal solution, as follows: Recall that FastMatch begins by matching leaves, and then proceeds to match higher levels in the tree in a bottom-up manner. With this approach, a mismatch at a lower level may "propagate," causing a mismatch at one or more higher levels. Our post-processing step will correct all mismatches other than those that propagated from lower levels to higher levels. It is difficult to evaluate precisely those cases that in which this propagation occurs without performing exhaustive computations. However, we can derive a necessary (but not sufficient) condition for propagation, and then measure that condition in our experiments. Informally, this condition states that in order to be mismatched, a node must have more than a certain number of children that violate Assumption 2, where the exact number depends on the match threshold t. Actually, this condition is weak; a node must satisfy many other conditions for the possibility of a mismatch to exist, and even then a mismatch is not guaranteed.

For the same document data analyzed earlier, Table 1 shows some statistics on the percentage of paragraphs that *may* be mismatched for a given value of the match

Threshold t:	0.5	0.6	0.7	0.8	0.9	1.0
% mismatched $\leq$:	0.4	1	3	7	9	10

Table 1: Mismatched paragraphs in *FastMatch*.

threshold t. For example, we see that with $t = 0.6$, we may mismatch at most 1% of the paragraphs. A lower value of t results in a lower number of possible mismatches. We see that the number of mismatched paragraphs is low, supporting our claim. Since the condition used to determine when a mismatch may occur is a weak one, the percentage of mismatches is expected to be much lower than suggested by these numbers. Furthermore, note that a suboptimal matching compromises only the quality of an edit script produced as the final output, not its correctness. In many applications, this trade-off between optimality and efficiency is a reasonable one. For example, when computing the delta between two documents, it is often not critical if the edit script produced is slightly longer than the optimal one.

7 Summary and Future Work

We have motivated the problem of computing and representing changes in hierarchically structured data. Our formal definition of the change detection problem for hierarchically structured data uses the idea of a matching and a minimum cost edit script that transforms one tree to another. We have split the change detection problem into two subproblems: the *Good Matching* and the *Minimum Conforming Edit Script* problems. We have presented algorithms for these problems, and we have studied our algorithms both analytically and empirically. Finally, as an application of some of these ideas, we have implemented a program for computing and representing changes in structured documents. More details about the implementation, including a sample "run," can be found in [CRGMW95].

We are working on generalizing our algorithms to detect changes in data that can be represented as graphs but not necessarily trees. We are also investigating other matching criteria to improve the performance of our algorithms, especially for non-document domains. We plan to further investigate the tradeoff between optimality and efficiency to produce a parameterized algorithm $\mathcal{A}(k)$ where the parameter k specifies the desired level of optimality. We are also improving the implementation of our *LaDiff* program, and extending it to HTML and SGML documents. We also plan to incorporate the diff program in a web browser so that users can monitor web pages of interest and track their changes over time.

References

[ACM95] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[CRGMW95] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. Available by anonymous ftp from `db.stanford.edu` in directory `pub/chawathe/1995/`, 1995.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[HKG+94] H.C. Howard, A.M. Keller, A. Gupta, K. Krishnamurthy, K.H. Law, P.M. Teicholz, S. Tiwari, and J. Ullman. Versions, configurations, and constraints in CEDB. CIFE Working Paper 31, Center for Integrated Facilities Engineering, Stanford University, April 1994.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse*. QED Publishing Group, Boston, Massachussetts, 1993.

[Kif95] M. Kifer. EDIFF–a comprehensive interface to diff for Emacs 19. Available through anonymous ftp at `ftp.cs.sunysb.edu`, 1995.

[LGM95] W. Labio and H. Garcia-Molina. Efficient algorithms to compare snapshots. Manuscript, available by anonymous ftp from `db.stanford.edu` in `pub/labio/1995/`, 1995.

[Mye86] E. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[SZ90] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11:581–621, 1990.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.

[Zha95] K. Zhang. Personal communication, May 1995.

[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.