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 [WC95], data warehousing [HGMW$^+$95, IK93, ZGMHW95], view maintenance [GM95], and version and configuration management [HKG$^+$94].

As one 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.

As an example in configuration management, consider the correlation of data stored in an architect's database with data stored in an electrician's database, where both databases are for the same building project. For autonomy reasons, the databases are updated independently. However, periodic consistent configurations of the entire design must be produced. This can be done by computing the deltas with respect to the last configuration and highlighting any conflicts that have arisen [HKG$^+$94].

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

*Research sponsored by the Wright Laboratory Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. This work was also supported by the Anderson Faculty Scholar fund and by equipment grants from Digital Equipment Corporation and IBM Corporation.

- *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 [WC95], we focus on the problem of detecting changes given old and new versions of the data. 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]. While our change detection algorithms are designed for this scenario, our basic approach and representation schemes should be applicable to built-in change management facilities as well.

- *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. We present experimental evidence suggesting 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.

While an edit script succinctly describes a delta, it is often useful to represent the delta by annotating the new version of the data with the changes. We call this annotated tree a *delta tree*, and we show how it can be used to visually display the changes to a user, i.e., by superimposing the new version of the data with markings that represent the changed, inserted, deleted or moved objects.

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. Figure 16 in Appendix A shows a sample run of the system. We have used this system to experimentally evaluate the performance of our algorithms; results are presented in Section 8.

In summary, the main contributions of this paper are the following:

- A formal definition of the problem of detecting deltas in hierarchically structured data given the old and new versions of the data.
- Efficient algorithms for computing a minimum cost edit script from tree representations of hierarchical data.
- Analytical and empirical performance studies of our algorithms.
- A general scheme, called a delta tree, to represent deltas in hierarchically structured information.
- A powerful LaDiff system for detecting and representing changes in hierarchically structured Latex documents that demonstrates the utility of our approach.

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 delta trees. In Section 7 we describe the application of our techniques to hierarchically structured documents. Our empirical performance study is described in Section 8. Conclusions and future work are covered in Section 9. Due to space constraints, several appendices—not needed for the reader to understand our approach, algorithms, or results—are attached. Appendix A shows a sample run of our *LaDiff* program. Appendix B provides details of the performance analysis of our algorithms. Appendix C provides proofs for formal results stated in the body of the paper.

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 paper [Mye86] describes an algorithm for flat text. It is used by the GNU *diff* utility, and compares two files by computing the LCS[1] of their lines. There are also a number of front-ends to this standard *diff* program that display the results of *diff* in a nicer manner. (The *ediff* program [Kif95] is a good example.) However, since

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

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-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. However, our delete operation is less general than the one in [ZS89], which is more symmetrical with the insert operation. We have added flexibility due to the move operation, although moves have been added to the [ZS89] algorithm in a post-processing step [WZS95]. The application domain usually determines which edit operations are more natural. In a general tree structure, the delete operation of [ZS89], which makes the children of the deleted node the children of its parent, is natural. However, in an object hierarchy, this may be undesirable due to restrictions on types and composite-object memberships. (For example, an object representing a library may have a number of book objects as subobjects. If a book is deleted, it is unnatural to have the subobjects of book (such as author, title, etc.) become subobjects of the library object.)

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.

[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.

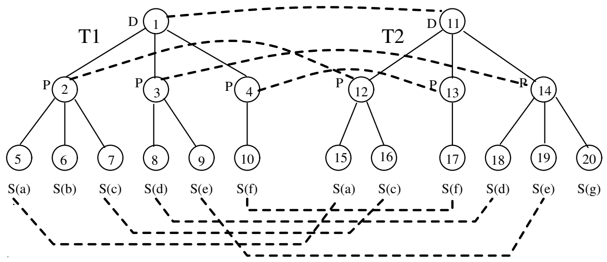


Figure 1: Running example (dashed edges represent matching)

3 Overview and Preliminaries

In this section, we formulate the change detection problem and split it into the following two subproblems which are discussed in later sections:

- Finding a "good" matching between the nodes of the two trees;
- Finding a minimum "conforming" edit script for the two trees given a computed matching.

In Section 3.1, we introduce these problems informally using an example. The formal definitions follow in Section 3.2, which also introduces some notation and terms used in the rest of the paper.

3.1 The Change Detection Problem

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.

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

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

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 (leaf) 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 formalizes and gives examples of edit scripts, while details of our algorithms for the MCES problem are addressed in Section 4. 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.

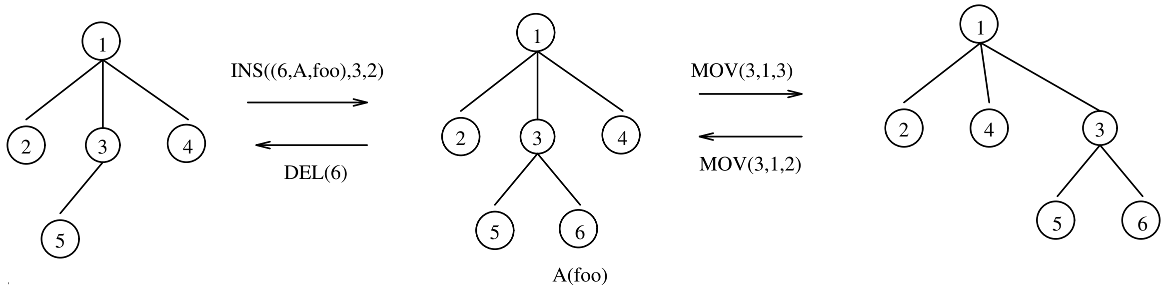


Figure 2: Edit operations on a tree

3.2 Edit Operations, Edit Scripts, and Costs

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:

Insert: The *insertion* of a new leaf node x into T_1, denoted by $\text{INS}((x, l, v), y, k)$. 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 $\text{DEL}(x)$. The result T_2 is the same as T_1, except that it does not contain node x. $\text{DEL}(x)$ 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 $\text{UPD}(x, val)$. 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 $\text{MOV}(x, y, k)$. 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.

Edit Scripts

Informally, an edit script gives a sequence of edit operations that transforms one tree into another. Formally, suppose e_1 is an edit operation that transforms T_1 to T_2, denoted by $T_1 \stackrel{e_1}{\to} T_2$. We say that a sequence

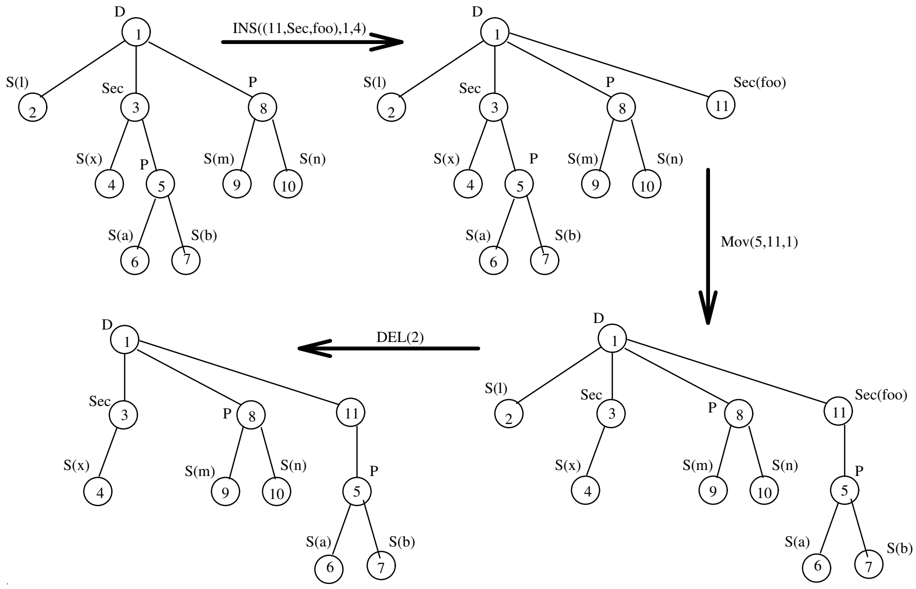


Figure 3: Applying the edit script of Example 3.1

$E = e_1, \ldots, e_m$ of edit operations takes T_1 to T_{m+1}, denoted by $T_1 \stackrel{E}{\to} T_{m+1}$, if there exist $T_2, \ldots, T_m$ such that $T_1 \stackrel{e_1}{\to} T_2 \stackrel{e_2}{\to} \ldots \stackrel{e_m}{\to} T_{m+1}$. A sequence E of edit operations *transforms* T_1 into T_2, denoted by $T_1 \stackrel{E}{\mapsto} T_2$, if $T_1 \stackrel{E}{\to} T_1'$ 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 than 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 tree T_1 shown in Figure 3. The edit script below transforms T_1 into T_2. The figure shows the intermediate trees in the transformation. (The last update is not shown in order to save space.)

$$\text{INS}((11, \textit{Sec}, \textit{foo}), 1, 4), \text{MOV}(5, 11, 1), \text{DEL}(2), \text{UPD}(9, \textit{baz})$$

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 (recall above), still there may be many correct scripts. 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:

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

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 usually 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 $compare(v, v')$ 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. Given two trees T_1 and T_2, and a matching M between their nodes, a *Minimum Conforming Edit Script (MCES)* for T_1, T_2 and M is an edit script that conforms[4] to M, and that transforms T_1 into T_2 such that there is no other edit script conforming to M that transforms T_1 into T_2 and has a lower cost.

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 described as consisting of five phases: the *update phase*, the *align phase*, the *insert phase*, the *move phase*, and the *delete phase*. We describe each phase in turn. Let us call a node that is not matched in M an *unmatched node*. The *partner* of a matched node is the node to which it is matched in M. 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 shown by the dashed lines in the figure.

[4]We defined the notion of an edit script conforming to a matching in Section 3.1.

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 $(x, y) \in M$.

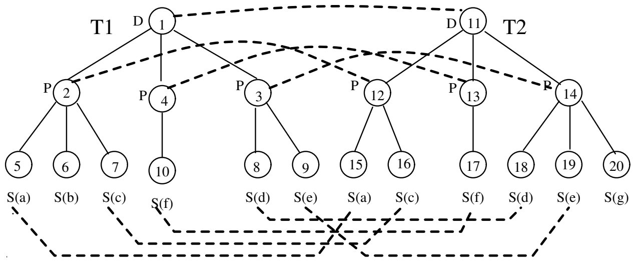


Figure 4: Running example: after align phase

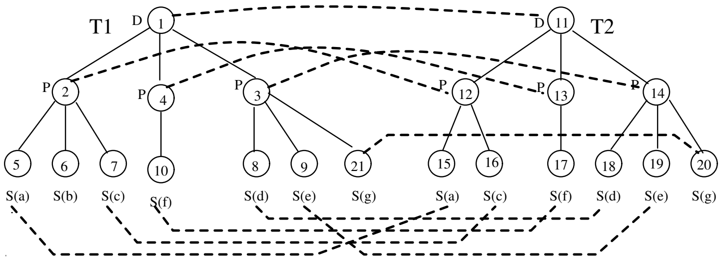


Figure 5: Running example: after insert phase

The Align Phase: 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: If the roots of T_1 and T_2 are not matched in M, then we add new (dummy) root nodes x to T_1 and y to T_2, and add (x, y) to M. The old root of T_1 is made the lone child of x and the old root of T_2 is made the lone child of y. Hereafter we assume without loss of generality that the roots of T_1 and T_2 are matched in M.

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 INS$((21, S, g), 3, 3)$. The transformed T_1 and the augmented M are shown in Figure 5.

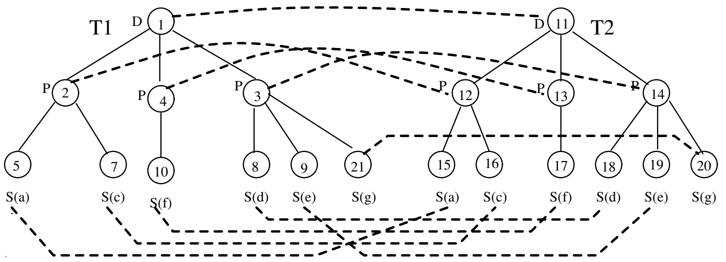


Figure 6: Running example: after delete phase

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.

The Move Phase: In the move phase we look for pairs of nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$. 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 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 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 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.

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 $S = a_1 a_2 \ldots a_n$, 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

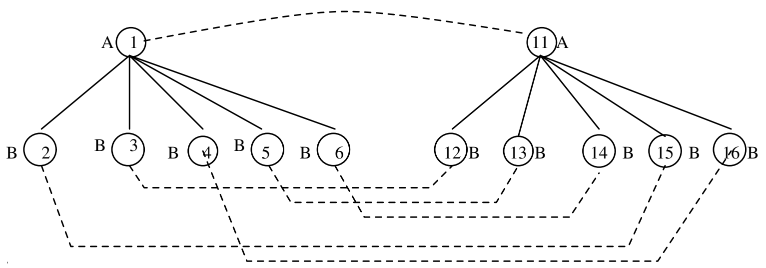


Figure 7: A matching with misaligned nodes

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 $1 \le i \le k$, $equal(x_i, y_i)$ 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 $\text{MOV}(4, 1, 5)$. We show in Appendix C 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

1. $E \leftarrow \epsilon, M' \leftarrow M$
2. Visit the nodes of T_2 in breadth-first order
   / this traversal 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 INS$((w, a, v(x)), z, k)$ to E, for a new identifier w.
      iii. Add (w, x) to M' and apply INS$((w, a, v(x)), z, k)$ to T_1.
   (c) else if x is not a 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 UPD$(w, v(x))$ to E.
         B. Apply 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 MOV(w, z, k) to E.
         D. Apply MOV(w, z, k) to T_1.
   (d) *AlignChildren*(w, x)
3. Do a post-order traversal of T_1. */* this is 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 DEL(w) to E and apply 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*

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.

The algorithm is shown in Figure 8. It uses two procedures, *AlignChildren* and *FindPos*, shown in Figure 9. The two statements in Algorithm *EditScript* that are marked with (*) claim that certain nodes have partners. These claims are substantiated in Appendix C, 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. Appendix C shows 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 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 matching between the nodes of trees T_1 and T_2 that can serve as input to Algorithm *EditScript*.

As discussed in the introduction, if the data has object ids, then the matching problem is trivial. However,

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 MOV(a, w, k) to E and apply 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* used by Algorithm *EditScript*

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. For example, the objects we are comparing, say sentences or paragraphs, may simply be characters with no meaningful object-id. In other cases the objects may have database identifiers but the ids may not be consistent between the two versions. For instance, the record representing a pillar in the architect's database may have id `778899`, but the same pillar in a subsequent version may have id `12345`. Here again, we need to match the pillars based on the value of the record (e.g., location and height of the pillar), as well as by its relationship to other objects (e.g., are the two pillars in the same room?). We use the term *keyless data* for hierarchical data that may not have identifying keys or object-ids. (Note that we are not ruling out keys for some objects; if they exist they can be used to match those objects quickly.)

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 in real domains. 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.

The hierarchical keyless data we are comparing does have labels, and these labels usually follow a *structuring schema*, such as the one defined in [ACM95]. Many structuring schemas satisfy an *acyclic labels* condition: 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 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. For example, consider Latex documents, restricted to nodes with labels `Sentence`, `Paragraph`, `Subsection`, `Section`, and `Document`. Observe that there is a natural ordering of the labels (`Sentence` < `Paragraph` < `Subsection` < `Section` < `Document`) that satisfies the acyclicity condition. If we add to this subset of Latex lists of type `itemize`, `enumerate`, and `description`, we have cycles. (An itemized list may contain an enumerated list, for example.) However, there is a simple way of overcoming this problem. Since all three kinds of lists are semantically similar, we merge their labels into a single `list` label. This acyclic labels condition will prove useful in designing our matching algorithms.

Our first matching criterion may be stated informally as follows: Nodes that are "too dissimilar" may not be matched with each other. Considering our example of structured documents, this means that we want to exclude from consideration matchings that match two completely different sentences, paragraphs, sections, etc. In stating this matching criterion formally, we consider the cases of leaf nodes and internal nodes separately. Recall (from Section 3) that the cost of an update operation is defined based on a function *compare*(s_1, s_2) that, given nodes s_1 and s_2, returns a number in [0, 2] that represents the "distance" between nodes s_1 and s_2. We choose a parameter f such that $0 \leq f \leq 1$, and we do not allow leaf nodes to match if the distance between them other is more than f.

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.) $\square$

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 sentences match.

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

$$\textit{common}(x, y) = \{(w, z) \in M \mid x \text{ contains } w, \text{and } y \text{ contains } z\}$$

Then in M we must have $l(x) = l(y)$ and

$$\frac{|\textit{common}(x, y)|}{\max(|x|, |y|)} > t$$

for some t satisfying $\frac{1}{2} \leq t \leq 1$. □

In the above definition, t is a parameter that we may vary between $\frac{1}{2}$ and 1.

Recall that one of our goals in setting up our matching criteria is to simplify the task of finding a best matching. The following lemma shows that Matching Criteria 1 and 2 allow us to look at only those matchings that are maximal—matchings M such that adding any new pair of nodes (x, y) to M will violate Matching Criterion 1 or 2. The lemma is proved in Appendix C.

Lemma 5.1 For matchings M and M' that satisfy Matching Criterion 1 if $M \subseteq M'$ then M is not better than M'.

Our next matching criterion 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. One tree representation of this data would contain movie titles as leaves. Then this matching criterion 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.

Matching Criterion 3 For any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $\textit{compare}(v(x), v(y)) \leq 1$. Similarly, for any leaf $y \in T_2$, there is at most one leaf $x \in T_1$ such that $\textit{compare}(v(x), v(y)) \leq 1$. □

This matching criterion 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 Matching Criterion 3 holds. When Matching Criterion 3 does not hold, our algorithm may generate a sub-optimal solution. However, we can often post-process a sub-optimal solution to obtain an optimal solution; we discuss this issue further in Section 8. In the remainder of this section, we will assume that the matching criteria are valid.

Given Matching Criterion 3, we can show that there is a unique maximal matching of leaf nodes. Then, using matching criterion 2, combined with the acyclicity condition on labels discussed earlier, we can show that there is a unique maximal matching of all nodes. The following theorem is proved in Appendix C.

1. $M \leftarrow \phi$
2. Mark all nodes of T_1 and T_2 "unmatched."
3. Proceed bottom-up on tree T_1

   For each unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ such that *equal*(x, y) then

   i. Add (x, y) to M.

   ii. Mark x and y "matched."

Figure 10: Algorithm *Match*

Theorem 5.2 (Unique Maximal Matching) Suppose T_1 and T_2 satisfy the acyclicity condition for labels. Then, given Matching Criteria 1, 2, and 3, there is a unique maximal matching M of the nodes of the two trees T_1 and T_2.

5.2 A Simple Matching Algorithm

Theorem 5.2 allows us to construct a straightforward algorithm to obtain the best matching that satisfies our matching criteria. For each node $x \in T_1$, we simply compare x with each unmatched node $y \in T_2$ that has the same label as x. We use the following function *equal* for leaf nodes, where f is a parameter such that $0 \leq f \leq 1$:

$$equal(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and } compare(v(x), v(y)) \leq f \\ false & \text{otherwise} \end{cases}$$

We use the following function *equal* for internal nodes ($t > \frac{1}{2}$ is a parameter):

$$equal(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and } \frac{|common(x,y)|}{\max(|x|,|y|)} > t \\ false & \text{otherwise} \end{cases}$$

The algorithm must match leaves before matching internal nodes to ensure that the equality function for internal nodes can be evaluated. Figure 10 shows this simple matching algorithm, which we call Algorithm *Match*.

Example 5.1 We illustrate our simple matching algorithm on the trees from our running example in Figure 1. The algorithm first examines each leaf node of T_1 in turn, and attempts to pair it with a leaf node of T_2. This process produces the following matching of leaf nodes:

$$M = \{(5, 15), (7, 16), (8, 18), (9, 19), (10, 17)\}$$

The algorithm then tries to pair nodes with the label `P`, and adds the pairs $(2, 12)$, $(3, 14)$, and $(4, 13)$ to the matching. Finally, pairing nodes with label `D` yields the pair $(1, 11)$. The final matching that results is shown in Figure 1 using dashed lines. □

In Appendix B we show that the running time of Algorithm *Match* is proportional to

$$n^2 c + mn \tag{1}$$

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) Pair unmatched nodes with label l as in Algorithm *Match*, adding matches to M.
3. Repeat steps 2a through 2e for each internal node label l.

Figure 11: Algorithm *FastMatch*

where n is the total number of leaf nodes in T_1 and T_2, m is the total number of internal nodes in T_1 and T_2, and c is the average cost of executing $compare(x, y)$ for a pair of leaf nodes x and y. (Section 7 describes how we compare sentences in our implementation.)

5.3 A Faster Matching Algorithm

We can significantly reduce the number of comparisons in Algorithm *Match* when T_1 and T_2 are nearly alike, which is often the case in practice. We modify Algorithm *Match* to Algorithm *FastMatch*, shown in Figure 11. Algorithm *FastMatch* 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 as in Algorithm *Match*. The function *equal* in the LCS call is as defined in Section 5.2.

In Algorithm *FastMatch* 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.

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 $E = e_1 e_2 \ldots e_n$ 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 defined as follows:

$$w_i = \begin{cases} 1 & \text{if } e_i \text{ is an insert or a delete} \\ |x| & \text{if } e_i \text{ is a move of the subtree rooted at node } x \\ 0 & \text{if } e_i \text{ is an update} \end{cases}$$

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.

It is shown in Appendix B that the running time of Algorithm *FastMatch* is proportional to

$$(ne + e^2)c + 2lne \tag{2}$$

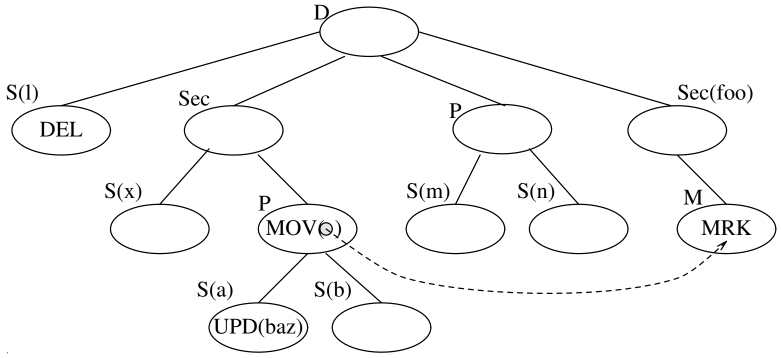


Figure 12: Delta tree for edit script in Example 3.1

where n and c are the same as in Formula (1) of Section 5.2, l is the number of internal node labels, and e is the weighted edit distance between T_1 and T_2. A comparison of Formula (2) with Formula (1) shows that Algorithm *FastMatch* is substantially faster than Algorithm *Match* when e is small compared to n, as is typically the case. Section 8 presents results from our empirical performance study of Algorithm *FastMatch*.

6 Delta Trees

In this section we describe a representation for deltas in hierarchically structured data that is more natural and useful than edit scripts for certain scenarios. As we have seen above, an edit script gives us the sequence of operations needed to transform one tree to another, and thus is a simple "operational" representation of deltas. One problem with edit scripts is that they refer to tree nodes using node identifiers. Node identifiers may be system-generated and thus not meaningful to the user. Furthermore, the flat, sequential structure of an edit script may make it difficult to use for querying and browsing hierarchical deltas.

In a relational database, deltas usually are represented using *delta relations*: For a relation R, delta relations *inserted(R)* and *deleted(R)* contain the tuples inserted to and deleted from R, while delta relations *old-updated(R)* and *new-updated(R)* contain the old and new values of updated tuples [GHJ$^+$93, WC95]. One can contrast this representation with the relational version of an edit script, which would (presumably) be a list of tuple-level inserts, deletes, and updates, possibly based on tuple identifiers. We are interested in a representation comparable to delta relations but for hierarchically structured data.

We define a structure called a *delta tree* for representing deltas. Intuitively, one can think of a delta tree as "overlaying" an edit script onto the data using node annotations.[5] As an example, the delta tree corresponding to the edit script from Example 3.1 is shown in Figure 12. Note that we do not need node identifiers since the annotated nodes are at the appropriate positions in the delta tree.

More formally, let T_1 and T_2 be two trees. A delta tree for T_1 with respect T_2 is a tree ΔT such that, in addition to a label and value, each node in ΔT has exactly one of the following *annotations*:

[5]In this sense, a delta tree differs from a delta relation in that delta relations are kept separate from the original data. In practice delta relations often are joined with their corresponding relation [WC95], and we are effectively representing this join explicitly.

- IDN, indicating that the node corresponds to a node in the original tree. (In Figure 12, IDN annotations appear as blanks.)
- UPD(v), indicating that the value of the node is updated to v.
- INS(l,v), indicating that the node is inserted with label l and value v.
- DEL, indicating deletion of the subtree rooted at the node.
- MOV(x), indicating that the node is moved to the position of the "marker node" x.
- MRK, indicating that the node is the destination of a move operation.

A *correct* delta tree for T_1 with respect to T_2 must have the property that there is at least one edit script E such that:

1. E transforms T_1 to T_2.
2. There is a total order over the nodes of ΔT such that outputting the edit operations corresponding to the node annotations in this order yields edit script E.

Note that there may be more than one such edit script. In general, we are interested in correct delta trees corresponding to minimum cost edit scripts.

In our implementation of the algorithms described in Sections 4 and 5, we construct the delta tree directly as a side-effect of producing an edit script. Essentially, this is achieved by modifying algorithm *EditScript* (Section 4) to emit a call to add a node to the delta tree every time an operation is added to the edit script being computed. Our implementation uses the delta tree representation rather than the edit script in order to produce meaningful output, as described in the next section.

7 Implementation

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. Below, we describe the implementation of this program, called *LaDiff*. We focus on Latex documents, but the implementation can easily handle other kinds of structured documents (e.g., HTML) by changing the parsing routines. Our performance study is presented in Section 8.

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.

A sample run of *LaDiff* is shown in Appendix A. To save space, the sample run uses a “toy” document that illustrates only some of the features of the program. 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.

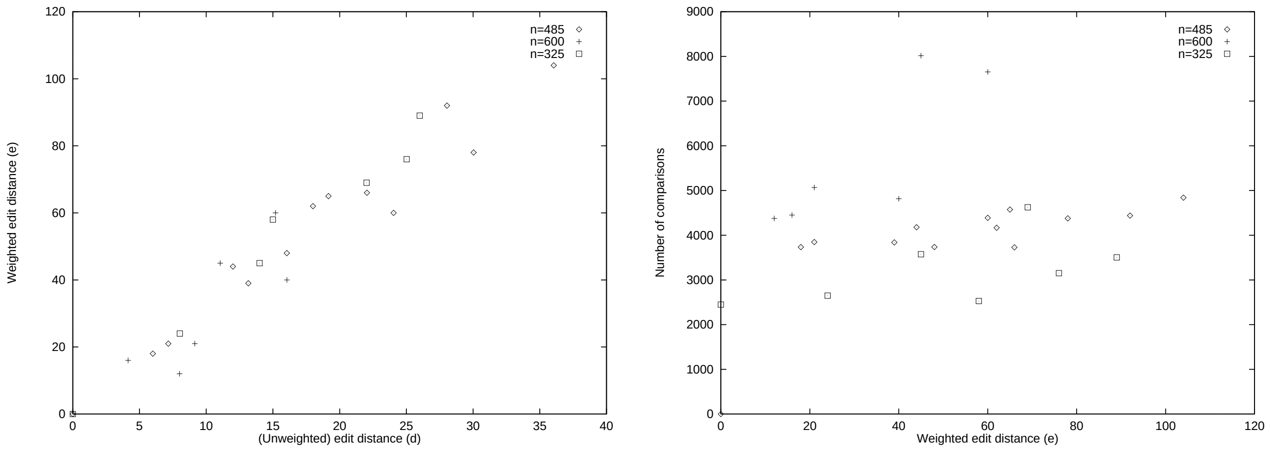
8 Empirical evaluation of FastMatch

In Section 5 we presented Algorithm *FastMatch* to find a matching between two trees, and we stated that its running time 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 5.3).

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. 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 13(a) 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, note that the variance with respect to the three document sets is not high. This suggests 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.

In Figure 13(b) 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 13(b); on the average, FastMatch makes approximately 20 times fewer comparisons than those predicted by the analytical bound.



(a) Weighted edit distance

(b) Running time

Figure 13: Performance of algorithm *FastMatch*

This supports our conjecture that the analytical bound on the running time is a loose one. We also observe that Figure 13(b) 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.

Another issue that needs to be addressed is the effect of the Matching Criterion 3 on the quality of the solution produced by FastMatch. Recall from Section 5 that FastMatch is guaranteed to produce an optimal matching only when Matching Criterion 3 holds. When Matching Criterion 3 does not hold, the algorithm may produce a sub-optimal matching. We describe a post-processing step that, when added to FastMatch, enables us to convert the possibly sub-optimal 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 sub-optimalities that may be introduced if Matching Criterion 3 does not hold for all nodes.

Even with post-processing, it is still possible to have a sub-optimal 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,

Match threshold (t):	0.5	0.6	0.7	0.8	0.9	1.0
Upper bound on mismatches (%):	0.4	1	3	7	9	10

Table 1: Mismatched paragraphs in *FastMatch*.

this condition states that in order to be mismatched, a node must have more than a certain number of children that violate Matching Criterion 3, 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. 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 non-optimal 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.

9 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. We also have defined a representation scheme called a delta tree for capturing deltas in hierarchical data. Finally, as an application of some of these ideas, we have implemented a program for computing and representing changes in structured documents.

As ongoing work, we are addressing on the following:

- Generalizing our algorithms to detect changes in data that can be represented as graphs but not necessarily trees.
- Investigating other matching criteria to improve the performance of our algorithms, especially for non-document domains.
- Further studying the tradeoff between optimality and efficiency to produce a parameterized algorithm $\mathcal{A}(k)$ where the parameter k specifies the desired level of optimality.

- Designing and implementing query, browsing, and active rule languages for hierarchical data based on our edit scripts and delta trees [WU95].
- 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.

References

[ACM95] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file update. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 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` in `/pub/TechReports/kifer/ediff.tar.Z`, 1995.

[Knu86] D. Knuth. *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1986.

[LGM95] W. Labio and H. Garcia-Molina. Efficient algorithms to compare snapshots. Available through 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.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1995.

[WU95] J. Widom and J. Ullman. C^3: Changes, consistency, and configurations in heterogeneous distributed information systems. Unpublished project description, available through the URL `http://www-db.stanford.edu/c3/synopsis.html`, 1995.

[WZS95] T-L. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases. In *SIGMOD Demo*, 1995.

[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.

Textual Unit	Edit Operation			
	Insert	Delete	Update	Move
Sentence	Bold font	Small font	Italic font	Footnote, label
Paragraph	Marginal note			Marginal note, label
Item	Marginal note			Marginal note, label
Subsection	Annotation(ins,del,upd,mov) in heading			
Section	Annotation(ins,del,upd,mov) in heading			

Table 2: Mark-up conventions used by *LaDiff*.

A Sample Run of *LaDiff*

In Section 7, we described an application of our ideas on change detection in hierarchical data to the domain of Latex documents. This application, called *LaDiff*, takes two versions of a Latex document as input, and produces as output a marked-up version of the document that highlights the changes. In this appendix, present a simple sample run of *LaDiff*. Due to limited space, we show a short example, based on an excerpt from the TEXbook [Knu86], that illustrates only some of the change detection features.

Figures 14 and 15 show the old and new versions of the example document. We tried the UNIX *diff* program on these documents, and the output was not very useful. Figure 16 shows the output of *LaDiff*. The conventions used by *LaDiff* for marking various changes in the output document are shown in Table 2. Sentence level changes are marked using changes in font: inserted sentences are in bold font, while deleted and updated sentences are in small and italic fonts respectively. Sentence moves are marked by putting the sentence in small font, labeling it, and referencing the label with a footnote at the new position of the sentence. (See the first and last sentences in the third section in Figure 16, for example.) Paragraph changes are marked using marginal notes indicating whether the paragraph is inserted, deleted, moved, or updated. In the case of paragraph moves, the old position of the paragraph is marked with a label which is referenced from the marginal note in its new position. (See the third paragraph in Figure 16, for example.) Changes in sections, subsections, and itemized lists are marked using similar schemes, as summarized by the table.

Note that sentences, as well as other textual units, may be moved and updated at the same time. The mark-up conventions used by *LaDiff* allow us to mark these changes simultaneously. For example, the first sentence in Figure 16 is in italic font, indicating that it was updated, and also has a footnote telling us that it was moved from position S1 (near the end of the document).

We can see that *LaDiff* properly detects insertions, deletions, updates, and moves of sentences and paragraphs. Representing the changes in an intuitive manner is a challenging problem, and we plan to work on it further.

1 First things first

Computer system manuals usually make dull reading, but take heart: This one contains JOKES every once in a while, so you might actually enjoy reading it. (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully.*)

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

2 Another way to look at it

In order to help you internalize what you're reading, exercises are sprinkled through this manual. It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure.

3 Conclusion

The TEX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details. Both languages have been called TEX; but henceforth the old language should be called TEX78, and its use should rapidly fade away. Let's keep the name TEX for the language described here, since it is so much better, and since it is not going to change any more.

Figure 14: Old version of document

1 Introduction

The TEX language described in this book has a predecessor, but the new system differs from the old one in literally thousands of details. Computer manuals usually make extremely dull reading, but don't worry: This one contains JOKES every once in a while, so you might actually enjoy reading it. (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully.*)

2 The details

English words like 'technology' stem from a Greek root beginning with letters $\tau\epsilon\chi$...; and this same Greek work means *art* as well as technology. Hence the name TEX, which is an uppercase of $\tau\epsilon\chi$.

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. This feature may seem strange, but it isn't. When certain concepts of TEX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

3 Moving on

It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure. In order to help you better internalize what you read, exercises are sprinkled through this manual.

4 Conclusion

Both languages have been called TEX; but henceforth the old language should be called TEX78, and its use should rapidly fade away. Let's keep the name TEX for the language described here, since it is so much better, and since it is not going to change any more.

Figure 15: New version of document

1 (upd) *Introduction*

[*The TeX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details.*][1] *Computer manuals usually make extremely dull reading, but don't worry: This one contains* JOKES *every once in a while, so you might actually enjoy reading it.* (However, most of the jokes can only be appreciated properly if you understand a technical point that is being made—so read *carefully.*) P1

2 (ins) *The details*

Inserted para

English words like 'technology' stem from a Greek root beginning with letters $\tau\epsilon\chi$...; and this same Greek work means *art* as well as technology. Hence the name TeX, which is an uppercase of $\tau\epsilon\chi$. Moved from P1

Another noteworthy characteristic of this manual is that it doesn't always tell the truth. **This feature may seem strange, but it isn't.** When certain concepts of TeX are introduced informally, general rules will be stated; afterwards you will find that the rules aren't strictly true. In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

3 *Moving on*

S2:[In order to help you internalize what you're reading, exercises are sprinkled through this manual.]
It is generally intended that every reader should try every exercise, except for questions that appear in the "dangerous bend" areas. If you can't solve a problem, you can always look up the answer. But please, try first to solve it by yourself; then you'll learn more and you'll learn faster. Furthermore, if you think you do know the solution, you should turn to Appendix A and check it out, just to make sure. [In order to help you better internalize what you read, exercises are sprinkled through this manual.][2]

4 **Conclusion**

S1:[The TeX language described in this book is similar to the author's first attempt at a document formatting language, but the new system differs from the old one in literally thousands of details.]
Both languages have been called TeX; but henceforth the old language should be called TeX78, and its use should rapidly fade away. Let's keep the name TeX for the language described here, since it is so much better, and since it is not going to change any more.

[1] Moved from S1
[2] Moved from S2

Figure 16: Output document (marked up)

B Analysis of Matching Algorithms

For a label a, let n_a be the total number of nodes with label a in T_1 and T_2. Let c_a be the average cost of computing *equal*(x, y) for nodes x and y with label a. Then Algorithm *Match* takes time $O(n_a^2 c_a)$ to match nodes with label a. Thus, the total time taken by the algorithm is proportional to $\sum_{a \in L} n_a^2 c_a$, where L is the set of all labels that appear in T_1 or T_2.

To simplify our analysis, let us assume that L is made up two disjoint subsets of labels—P, the set of labels of leaf nodes, and Q, the set of labels of internal nodes. Further, let us assume that all leaf node comparisons have the same average cost, that is, $c_a = c$ for all $a \in P$. Let n be the total number of leaf nodes in T_1 and T_2. Then matching leaf nodes takes time $O(n^2 c)$. For an internal node label $b \in Q$, computing *equal*(x, y) for nodes x and y with label b requires us to intersect the leaf nodes they contain, which takes time proportional to $\min(|x|, |y|)$. If we assume that, on average, $|x| = n/n_b$ for nodes x with label b, then we may approximate c_b by n/n_b, and so matching nodes with label b takes time $O(n_b n)$. Thus, the total time taken by Algorithm *Match* is proportional to

$$n^2 c + n \sum_{b \in Q} n_b.$$

If we denote by m the total number of internal nodes in T_1 and T_2, then $m = \sum_{b \in Q} n_b$, and so the running time of Algorithm *Match* is $O(n^2 c + mn)$.

For a label a, let $d_a = n_a - lcs_a$. Then Algorithm *FastMatch* takes time proportional to $(n_a d_a + d_a^2) c_a$ to match nodes with label a. Let us make the same assumptions as in the analysis of Algorithm *Match*. Then matching leaf nodes takes time that is proportional to $(nd + d^2)c$, where $d = \sum_{a \in P} d_a$. For internal nodes with label b, let us once again assume that $c_b = n/n_b$. Now, remembering that $d_b \leq n_b$ and so $d_b^2 c_b \leq n d_b$, the time taken to match nodes with label b is proportional to $2nd_b$. Hence the total time taken by Algorithm *FastMatch* is proportional to

$$(nd + d^2)c + \sum_{b \in Q} 2nd_b.$$

Now let e be the weighted edit distance between trees T_1 and T_2, as defined in Section 5.3. It is clear that for any label b, we have $d_b \leq e$. Hence the running time of Algorithm *FastMatch* is bounded by

$$(ne + e^2)c + 2lne$$

where $l = |Q|$ is the number of labels of internal nodes in T_1 and T_2.

C Proofs

Lemma C.1 For sequences S_1 and S_2 and an equality function *equal* such that each element in S_1 is equal to exactly one element in S_2 and vice versa, the minimum number of moves of elements of S_1 required to align the elements of S_1 and S_2 is $|S_1| - |LCS(S_1, S_2)|$.

Proof. Suppose we can use fewer moves. Then consider the elements of S_1 that were not moved and their "partners" in S_2. They would form a common subsequence longer than $|LCS(S_1, S_2)|$, a contradiction. □

Theorem C.2 Algorithm *FastMatch* computes the minimum cost edit script that conforms to the given matching M, and it does so in time $O(ND)$ where N is the number of nodes in the two trees and D is the number of misaligned nodes.

Proof. (Sketch) We first show that the edit script E that is generated transforms T_1 to T_2 and conforms to M. The proof is in two stages.

In the first stage we show that at the end of the breadth-first traversal of T_2, the subtree of T_1 corresponding to only its matched nodes (under M') is isomorphic to T_2. The proof is by induction on the number of nodes visited so far by the breadth-first search. The induction hypothesis is the following: Consider the subtree T_2^s of T_2 that contains only nodes that have already been visited by the breadth-first search. Let T_1^s be the subtree of T_1 that contains only partners of the nodes in T_2^s. Then T_1^s is isomorphic to T_2^s. Moreover, every node in T_2^s is matched to some node in T_1^s in M'. The details of the induction are straightforward and are omitted.

In the second stage we show that the post-order traversal of T_1 deletes all the unmatched nodes in T_1, so that T_1 becomes isomorphic to T_2. The only problem we may face is that some node that we wish to delete has children and so the deletion is not a legal operation. Suppose some unmatched nodes in T_1 are not deleted. Let x be a "lowest" such node in T_1, i.e., a node that occurs before all other such nodes in the post-order numbering. Then it follows from the first part of the proof that x does not have any children in T_1. Hence x could have been deleted during the post-order traversal of T_1, a contradiction.

Thus E transforms T_1 to T_2. It is also clear that E conforms to M because E never deletes any nodes that are matched by M. We also note that the inductive proof used in the first stage shows that the claims made by the statements marked with a (*) in Algorithm *EditScript* are indeed correct.

We now show that E is a minimum cost edit script. Any edit script conforming to M must contain at least:

- one insert operation corresponding to each unmatched node in T_2;
- one delete operation corresponding to each unmatched node in T_1; and
- one move operation corresponding to each pair of matched nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$ (call these *inter-parent moves*).

It is clear that Algorithm *EditScript* generates precisely the above inserts, deletes, and inter-parent moves. All that remains is to show that the algorithm also generates the fewest possible *intra-parent* moves (moves that change the relative ordering of siblings). Such moves are generated only in Function *AlignChildren*. That the minimum possible number of such moves is generated is an immediate consequence of Lemma C.1. Hence E is a minimum cost edit script.

Finally, we analyze the running time of the algorithm. Other than in Function *AlignChildren*, the breadth-first search and post-order traversal perform a constant amount of work for each node in T_1 and T_2. Let $|x|$ denote the number of children of node x. For matched nodes $w \in T_1$ and $x \in T_2$, let $d(x, w)$ denote the number of misaligned children of x and w. Then Function *AlignChildren* aligns the children of w and x in time $O((|w| + |x|)d(w, x))$. Hence the total running time is $O(ND)$. $\square$

Lemma 5.1 For matchings M and M' that satisfy Matching Criterion 1 if $M \subseteq M'$ then M is not better than M'.

Proof. (Sketch) For matchings M and M' satisfying the value constraint, the cost of moving and then updating a node is no more than the cost of deleting and inserting a node. Suppose M' is obtained from M by adding to M the match (x, y). Then any edit script conforming to M will contain operations that delete the node x and insert another node corresponding to y, whereas an edit script conforming to M' can replace the insertion and deletion by a move and an update and be no more expensive. □

Lemma C.3 Suppose T_1 and T_2 satisfy the acyclicity condition for labels and Assumption 3 holds. For any internal node $x \in T_1$, there is at most one internal node $y \in T_2$ such that the pair (x, y) satisfies the match threshold constraint. Similarly, for any internal node $y \in T_2$, there is at most one internal node $x \in T_1$ such that the pair (x, y) satisfies the match threshold constraint.

Proof. Suppose that node $x \in T_1$ has two "partners" y and z in T_2 satisfying the match threshold constraint. Then we must have

$$\frac{|common(x, y)|}{\max(|x|, |y|)} > t$$

and

$$\frac{|common(x, z)|}{\max(|x|, |z|)} > t.$$

The acyclicity condition implies that y and z can have no common descendents, so we must have

$$|common(x, y)| + |common(x, z)| > 2t|x|$$

which is impossible since $t \geq 1/2$. A symmetric argument holds, reversing T_1 and T_2. □

Theorem 5.2 Suppose T_1 and T_2 satisfy the acyclicity condition for labels and Assumption 3 holds. There is a unique maximal matching M of the nodes of the two trees T_1 and T_2 that satisfies the value constraint and the match threshold constraint.

Proof. Follows from Lemma 5.1 and Lemma C.3. □