Meaningful Change Detection in Structured Data*

Sudarshan S. Chawathe Hector Garcia-Molina

Computer Science Department, Stanford University, California 94305-9040

{chaw,hector}@cs.stanford.edu

Abstract

Detecting changes by comparing data snapshots is an important requirement for difference queries, active databases, and version and configuration management. In this paper we focus on detecting meaningful changes in hierarchically structured data, such as nested-object data. This is a much more challenging problem than the corresponding one for relational or flat-file data. In order to describe changes better, we base our work not just on the traditional "atomic" insert, delete, update operations, but also on operations that move an entire sub-tree of nodes, and that copy an entire sub-tree. This allows us to describe changes in a semantically more meaningful way. Since this change detection problem is $\mathcal{NP}$-hard, in this paper we present a heuristic change detection algorithm that yields close to "minimal" descriptions of the changes, and that has fewer restrictions than previous algorithms. Our algorithm is based on transforming the change detection problem to a problem of computing a minimum-cost edge cover of a bipartite graph. We study the quality of the solution produced by our algorithm, as well as the running time, both analytically and experimentally.

1 Introduction

Detection of changes between data structures is an important function in many applications. For example, in the World Wide Web an analyst may be interested in knowing how a competitor's site has changed since the last time visited. This may be achieved by saving a snapshot of the previous HTML pages at the site (something that most browsers do for efficiency anyway). In a CAD design environment, an engineer may wish to understand the differences between two related but concurrently developed chip designs. In a distributed file system, an administrator may need to detect differences between two mirror file systems that became partitioned and independently modified. In a warehousing environment, the changes at a site need to be identified so that a materialized view can be incrementally maintained.

In this paper we present an efficient algorithm, MH-DIFF, for *meaningful* change detection between two *hierarchically* structured data snapshots, or *trees*. The key word here is meaningful (the "M" in the name). That is, our goal is to portray the changes between two trees in a succinct and descriptive way. As is commonly done, we portray the changes as an *edit script* that gives the sequence of *operations* needed to transform one tree into another. However, in this paper we use a

*This work was supported by the Air Force Wright Laboratory Aeronautical Systems Center under DARPA Contract F33615-93-1-1339, by the Department of the Air Force Rome Laboratories under DARPA Contract F30602-95-C-0119, and by an equipment grant from IBM Corporation.

richer set of operations than has ever been used before, and this leads, we believe, to much higher quality edit scripts.

In particular, we use *move* and *copy* operations, in addition to the more traditional insert, delete, and update operations. Thus, if a substructure (e.g., a section of text, a shift register) is moved to another location, our algorithm will report it as a single operation. If the substructure is copied (e.g., a second shift register is added which is identical to one already in the circuit), then our algorithm will identify it as such. Traditional change detection algorithms would report such changes as sequences of inserts and deletes (or simply inserts in the case of a copy), which do not convey the true meaning of the change.

Note that detecting moves and copies becomes more important if the moved or copied subtree is large. For instance, if we are comparing file systems, and a large directory with thousands of files is mounted elsewhere, we clearly do not wish to report the change as thousands of file deletes followed by thousands of file creations. Also note that to detect moves and copies, it is essential that our algorithm understand the *structure* as well as the content of the data. Thus, our algorithm cannot treat the data as "flat" information, e.g., as files with records or relations with tuples. This means that techniques developed for flat change detection [Mye86, LGM96] are not applicable here.

Algorithm MH-DIFF has two additional important features:

- It does not rely on the existence of node (atomic object) identifiers that can match nodes in one tree to nodes in the other. In many applications such identifiers do not exist. For instance, sentences and paragraphs in text documents do not come with unique identifiers attached. Even when the nodes are stored in a database system (e.g., circuit components), we may be comparing copies with the same content but different identifiers. Thus, for full generality, MH-DIFF does not assume unique identifiers that span the two trees, and instead compares the contents of nodes to determine if they are related. (If the trees have such identifiers, MH-DIFF could easily take advantage of them, but we do not discuss that here.)

- Algorithm MH-DIFF is based on a fairly flexible cost model. Each operation in the repertoire is given a user-defined fixed cost, except for the update operation, whose cost is determined by a user-provided function that compares the values of two nodes. This gives end users great latitude in saying what types of edit scripts are preferable for an application.

There is a good reason why difference algorithms with the features we have described here have not been developed earlier, even though they are clearly desirable. The reason is the inherent complexity of the problem; one can show that the problem is $\mathcal{NP}$-hard.[1] Algorithm MH-DIFF provides a heuristic solution, which is based on transforming the problem to the "edge cover domain." That is, instead of working with edit scripts, the algorithm works with edge covers that represent how one set of nodes match another set. For this, the costs of the edit operations are translated into costs on the edges of the cover.

In an earlier paper of ours [CRGMW96] we studied a much simpler version of the change detection problem. In that work we did not consider copy operations, we assumed that the number of duplicates of a node was very limited, we assumed ordered trees, and we assumed that nodes had "tags" that reflect the structural constraints on the input trees. (For example, nodes were tagged as say "paragraphs" or "sections," making it easier to match nodes.) All these restrictions made it much simpler to find a minimum-cost edit script, and indeed we developed an efficient

[1] By reduction from the "exact cover by three-sets" problem.

algorithm that found the minimum-cost script. On the other hand, here we drop these limitations, and introduce copy operations. This leads to an algorithm that is very different from the one in [CRGMW96], yielding a heuristic solution in at worst $O(n^3)$ time, where n is the number of nodes, but most often in roughly $O(n^2)$ time. In Section 7 we compare in more detail MH-DIFF to our earlier work, as well as to other work on change detection.

In summary, the contributions of this paper are:

- We present a change detection framework that for the first time includes move and copy operations. We also define a flexible cost model for edit operations.
- We present MH-DIFF, an efficient algorithm for computing meaningful edit-scripts that are very close to the minimal cost edit script.
- We present preliminary experimental results showing how close to optimal the MH-DIFF solutions are. We also experimentally evaluate the key parameter that determines the running time of MH-DIFF in practice.

2 Model and Problem Definition

We use rooted, labeled trees as our model for structured data. These are trees in which each node n has a label $l(n)$ that is chosen from an arbitrary domain $\mathcal{L}$. The problem of snapshot change detection in structured data is thus the problem of finding a way to edit the tree representation of one snapshot to that of the other. We denote a tree T by its nodes N, the parent function p, and the labeling function l, and write $T = (N, p, l)$. The children of a node $n \in N$ are denoted by $C(n)$.

We begin by defining the tree edit operations that we consider. Since there are many ways to transform one tree to another using these edit operations, we define a cost model for these edit operations, and then define the problem of finding a minimum-cost edit script that transforms one tree to another.

2.1 Edit Operations and Edit Scripts

In the following, we will assume that an edit operation e is applied to $T_1 = (N_1, p_1, l_1)$, and produces the tree $T_2 = (N_2, p_2, l_2)$. We write this as $T_1 \xrightarrow{e} T_2$. We consider the following six edit operations:

Insertion: Intuitively, an insertion operation creates a new tree node with a given label, and places it at a given position in the tree. The position of the new node n in the tree is specified by giving its parent node p and a subset C of the children of p. The result of this operation is that n is a child of p, and the nodes C, that were originally children of p, are now children of the newly inserted node n.

Formally, an insertion operation is denoted by $\text{INS}(n, v, p, C)$, where n is the (unique) identifier of the new node, v is the label of the new node, $p \in N_1$ is the node that is to be the parent of n, and $C \subseteq C(p)$ is the set of nodes that are to be the children of n. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup \{n\}$, $p_2(n) = p$, $p_2(c) = n, \forall c \in C$, $p_2(c) = p_1(c), \forall c \in N_1 - C$, $l_2(n) = v$, and $l_2(m) = l_1(m), \forall m \in N_1$. Due to space constraints, we describe the remaining edit operations only informally below; the formal definitions are in Appendix A.

Deletion: This operation is the inverse of the insertion operation. Intuitively, DEL(n) causes n to disappear from the tree; the children of n are now the children of the (old) parent of n. The root of the tree cannot be deleted.

Update: The operation UPD(n, v) changes the label of the node n to v.

Move: A move operation MOV(n, p) moves the subtree rooted at n to another position in the tree. The new position is specified by giving the new parent of the node, p. The root cannot be moved.

Copy: A copy operation CPY(m, p) copies the subtree rooted at n to a another position. The new position is specified by giving the node p that is to be the parent of the new copy. The root cannot be copied.

Glue: This operation is the inverse of a copy operation. Given two nodes n_1 and n_2 such that the subtrees rooted at n_1 and n_2 are isomorphic, GLU(n_1, n_2) causes the subtree rooted at n_1 to disappear. (It is conceptually "united" with the subtree rooted at n_2.) The root cannot be glued. Although the GLU operation may seem unusual, note that it is a natural choice for an edit operation given the existence of the CPY operation. As we will see in Example 2.1, inverting an edit script containing a CPY operations results in an edit script with a GLU operation. This symmetry in the structure of edit operations is useful in the design of our algorithms.

In addition to the above tree edit operations, one may wish to consider operations such as a *subtree delete* operation that deletes all nodes in a given subtree. Similarly, one could define a *subtree merge* operation that merges two or more subtrees. We do not consider such more complex edit operations in this paper, but note that some of these operations, (e.g., subtree deletes) may be detected by post-processing the output of our algorithm.

We define an *edit script* to be a sequence of zero or more edit operations that can be applied in the order in which they occur in the sequence. That is, given a tree T_0, a sequence of edit operations $\mathcal{E} = e_1, e_2, \ldots, e_k$ is an edit script if there exist trees T_i, $1 \leq i \leq k$ such that $T_{i-1} \xrightarrow{e_i} T_i$, $1 \leq i \leq k$. We say that the edit script $\mathcal{E}$ transforms T_0 to T_k, and write $T_0 \xrightarrow{\mathcal{E}} T_k$.

Example 2.1 Consider the tree T_1 depicted in Figure 1. We represent the identifier of each node by the number inside the circle representing the node. The label of each node is depicted to the right of the node. Thus, the root of the tree T_1 has an identifier 1, and a label a. Figure 1 shows how T_1 is transformed by applying the edit script to $\mathcal{E}_1 = (\text{INS}(g, 1, 7, 9), \text{MOV}(2, 6), \text{CPY}(6, 1))$ T_1. Similarly, if we start with the tree T_2 in the figure, the edit script $\mathcal{E}_2 = (\text{GLU}(12, 6), \text{MOV}(2, 1), \text{DEL}(11))$ transforms it back to T_1. We write $T_1 \xrightarrow{\mathcal{E}_1} T_2$, and $T_2 \xrightarrow{\mathcal{E}_2} T_1$.

When an edit script is applied to tree, as in Example 2.1, the node identifiers in the initial and final state of the tree determine a mapping between the nodes in the two states. Note however, that in an instance of a change detection problem, we are given two trees, without any correspondence between their node identifiers. That is, in a change detection problem involving the trees T_1 and T_2 of Figure 1, the node identifiers of T_2 would be unrelated to those of T_1. We will discuss this issue further in Section 3.

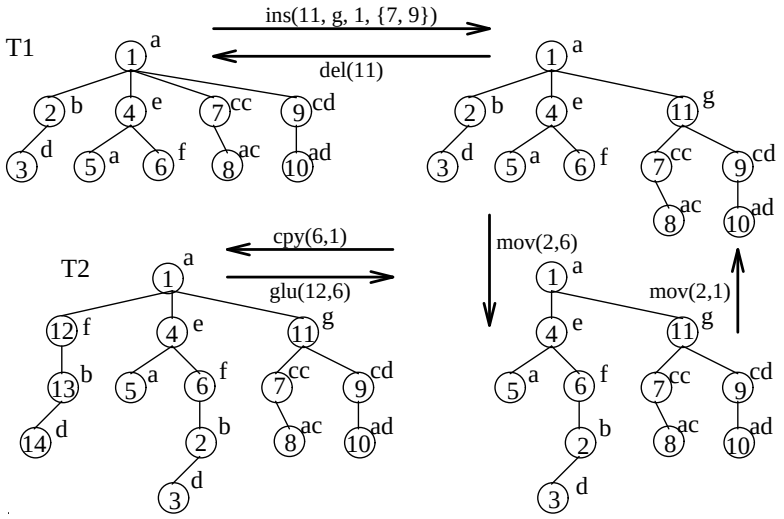

Figure 1: Edit operations on labeled trees

2.2 Cost Model

Given a pair of trees, there are, in general, several edit scripts that transform one tree to the other. For example, there is the trivial edit script that deletes all the nodes of one tree and then inserts all the nodes of the second tree. There are many other edit scripts that, informally, do more work than seems necessary. Formally, we would like to find an edit script that is "minimal" in the sense that it does no more work that what is absolutely required. To this end, we define a cost model for edit operations and edit scripts.

There are two major criteria for choosing a cost model. Firstly, the cost model should accurately capture the domain characteristics of the data being considered. For example, if we are comparing the schematics for two printed-circuit boards, we may prefer an edit script that has as few inserts as possible, and instead describes changes with moves and copies of the old components. However, if we are comparing text documents, we may prefer to see a paragraph as a new insertion, rather than a description of how it was assembled from bits and pieces of sentences from the old document. Secondly, the cost model should be simple to specify, and should require little effort from the user. For example, a cost model that requires the user to specify dozens of parameters is not desirable by this criterion, even though it may accurately model the domain.

Another issue is the trade-off between generality of the cost model and difficulty in computing a minimum-cost edit script. For example, a very general cost model would have a user-specified function to determine the cost of each edit operation, based on the type of the edit operation, as well as the particular nodes on which it operates. However, such a model is not amenable to the design of efficient algorithms for computing the minimum-cost edit script, since it does not permit us to reason about the relative costs of the possible edit operations.

With the above criteria in mind, we propose a simple cost model in which the cost of an insert, delete, move, copy, and glue operation is given by constants, c_i, c_d, c_m, c_c, and c_g, respectively. Furthermore, given the symmetry between INS and DEL, and CPY and GLU, it is reasonable to use $c_i = c_d$, and $c_c = c_g$. Since, intuitively, a MOV operation causes a smaller change than either CPY

or GLU , it is also reasonable to use $c_m < c_c$. Note, however, that our algorithms do not depend on such relationships between the cost parameters. The cost of an update operation depends on the old and new values of the label being updated; that is, $c(\text{UPD}(n, v)) = c_u(v_0, v)$, where v_0 is the old label of n, and c_u is a domain-dependent function that returns a non-negative real number.

Finally, the *cost of an edit script* $\mathcal{E}$, denoted by $c(\mathcal{E})$, is defined as the sum of the costs of the edit operations in $\mathcal{E}$. That is, $c(\mathcal{E}) = \sum_{d \in \mathcal{E}} c(d)$.

Problem Statement: Given two rooted, labeled trees T_1 and T_2, find an edit script $\mathcal{E}$ such that $\mathcal{E}$ transforms T_1 to a tree that is isomorphic to T_2, and such that for every edit script $\mathcal{E}'$ with this property, $C(\mathcal{E}') \geq C(\mathcal{E})$.

3 Method Overview

In this Section, we present an overview of algorithm MH-DIFF for computing a minimum-cost edit script between two trees. We present our algorithm informally using a running example; the details are deferred to later sections.

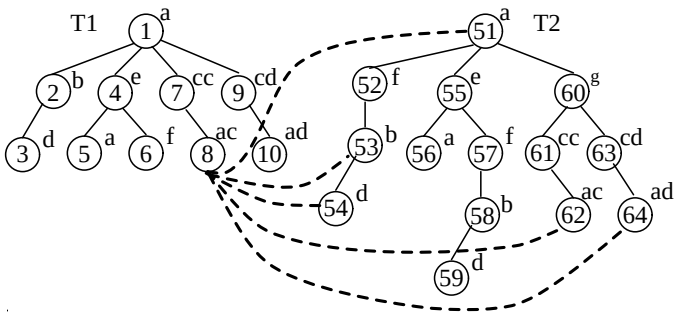

Figure 2: The trees for the running example in Section 3.

Consider the two trees depicted in Figure 2. We would like to find a minimum-cost edit script that transforms tree T_1 into tree T_2. The reader may observe that these trees are isomorphic to the initial and final trees from Example 2.1 in Section 2. Note, however, that there is no correspondence between the node identifiers of T_1 and T_2 in Figure 2. This is because in Example 2.1 we applied a known edit script to a tree, transforming it to another tree in the process, whereas in this section, we are trying to find an edit script, given two trees with no information on the relationship between their nodes. Therefore, our first step consists of finding a correspondence between the nodes of the two given trees.

For example, consider the node 8 in Figure 2. We want to find the node in T_2 that corresponds to this node in T_1. The dashed lines in Figure 2 represent some of the possibilities. Intuitively, we can see that matching the node 8 to the node 51 does not seem like a good idea, since not only do the labels of the two nodes differ, but the two nodes also have very different locations in their respective trees; node 8 is a leaf node, while node 51 is the root node. Similarly, we may intuitively argue that matching node 8 to node 62 seems promising, since they are both leaf nodes and their labels match. However, note that matching a nodes based simply on their labels ignores the structure of the trees, and thus is not, in general, the best choice. We make this intuitive notion

of a correspondence between nodes more precise below.

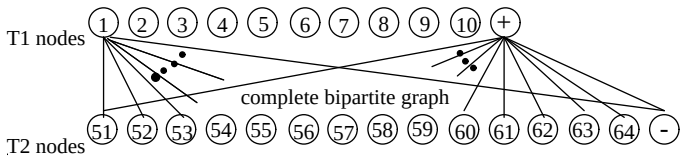

Figure 3: The Induced Graph for the trees in Figure 2

3.1 The Induced Graph

Consider the complete bipartite graph B depicted in Figure 3, consisting of the nodes of T_1 at the top, and the nodes of T_2 at the bottom, plus the special nodes $\oplus$ and $\ominus$. (For clarity, not all edges of the graph are shown in Figure 3.) We call B the *induced graph* of T_1 and T_2. The dashed lines in Figure 2 correspond to the edges of the induced graph. Intuitively, we would like to find a subset K of the edges of B that tells us the correspondence between the nodes of T_1 and T_2. If an edge connects a node $m \in T_1$ to a node $n \in T_2$, it means that n was "derived" from m. (For example, n may be a copy of m.) We say m is *matched* to n. A node matched to the special node $\oplus$ indicates that it was inserted, and a node matched to $\ominus$ indicates that it was deleted. Note that this matching between nodes need not be one-to-one; a node may be matched to more than one other nodes. (For example, referring to Figures 2 and 3, node 6 may be matched to both node 54 and node 59.) The only restriction is that a node be matched to at least one other node. Thus, finding the correspondence between the nodes of two trees consists essentially of finding an edge cover[2] of their induced graph.

The induced graph has a large number of edge covers (this number being exponential in the number of nodes). However, we may intuitively observe that most of these possible edge covers of B are undesirable. For example, and edge cover that maps all nodes in T_1 to $\ominus$, and all nodes in T_2 to $\oplus$ seems like a bad choice, since it corresponds to deleting all the nodes of T_1 and then inserting all the nodes of T_2. We will define the correspondence between an edge cover of an induced graph and an edit script for the underlying trees formally in Section 4.3, where we also describe how to compute an edit script corresponding to an edge cover. For now, we simply note that, given an edge cover of the induced graph, we can compute a corresponding edit script for the underlying trees. Hence, we would like to select an edge cover of the induced graph that corresponds to a minimum-cost edit script.

3.2 Pruning the Induced Graph

We noted earlier that many of the potential edge covers of the induced graph are undesirable because they correspond to expensive and undesirable edit scripts. Intuitively, we may therefore expect a substantial number of the edges of the induced graph to be extraneous. Our next step, therefore, consists of removing (pruning) as many of these extraneous edges as possible from the induced

[2] An edge cover of a graph is a subset K of the edges of the graph such that any node in the graph is incident on at least one edge in S.

graph, by using some *pruning rules*. The pruning rules that we use are *conservative*, meaning that they remove only those edges that we can be sure are not needed by a minimum-cost edit script. We discuss pruning rules in detail in Section 5.3, presenting only a simple example here.

As an example of the action of a simple pruning rule, consider the edge $e_1 = [5, 54]$ in Figure 3, representing the correspondence between nodes 5 and 54 in Figure 2. Suppose that the cost $c_U(a, d)$ of updating the label a of node 5 to the label d of node 54 is 3 units. Furthermore, let the cost of inserting a node and deleting a node be 1 unit each. Then we can safely prune the edge $[5, 54]$ because, intuitively, given any edge cover K_1 that includes the edge e_1, we can generate another edge cover that excludes e_1, and that corresponds to an edit script that is at least as good as the one corresponding to K_1. As an illustration of such pruning, consider the edge cover $K_2 = K_1 - \{e\} \cup \{[5, \ominus], [\oplus, 54]\}$. This edge cover corresponds to an edit script that deletes the node 5, and inserts the node 54. These two operations cost a total of 2 units, which is less than the cost of the update operation suggested by the edge e in edge cover K_1. We therefore conclude that the edge $[5, 54]$ in our running example may safely be pruned. In Section 5.3 we present Pruning Rule 2, which is a generalization of this example.

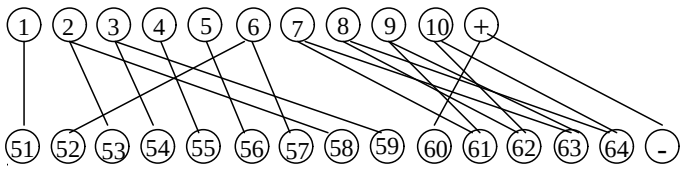

Figure 4: The induced graph of Figure 3 after pruning

3.3 Finding an Edge Cover

By applying the pruning rules to the induced graph of our running example (Section 5.3), say we obtain the *pruned induced graph* depicted in Figure 4. Although the pruned induced graph typically has far fewer edges than the original induced graph does, it typically still contains more edges than needed to form an edge cover. In Section 4.3 we will see that we need only consider edge covers that are *minimal*; that is, edge covers that are not proper supersets of another edge cover. In other words, we would like to remove from the pruned induced graph those edges that are not needed to cover nodes. For example, in the pruned induced graph shown in Figure 4, having all four of the edges $[7, 61]$, $[7, 63]$, $[9, 61]$, and $[9, 63]$ is unnecessary; we may remove either $[7, 63]$ and $[9, 61]$; or $[7, 61]$ and $[9, 63]$. However, it is not possible to decide a priori which of these options is the better one; that is, it is not obvious which choice would lead to an edit script of lower cost. With pruning, on the other hand, there was no doubt that certain edges could be removed.

One way to decide among these options is to enumerate all possible minimal edge covers of the pruned induced graph, find the edit script corresponding to each one (using the method described later in Section 4.3), and to pick the one with the least cost. However, given the exponentially large number of edge covers, this is obviously not an efficient algorithm. To compute an optimal edge cover efficiently, we need to be able to determine how much each edge in the edge cover contributes to the total cost of an edit script corresponding to an edge cover containing it. That is, we need to distribute the cost of the edit script corresponding to an edge cover over the individual edges of the edge cover. Once we have a cost defined for each edge in the pruned induced graph, we can find a

minimum-cost edge cover using standard techniques based on reducing the edge cover problem to a weighted matching problem [PS82, Law76]. For example, if the edges $[7,61]$, $[7,63]$, $[9,61]$, and $[9,63]$, have costs 0, 1.3, 0.2, and 2.4, respectively, then we generate an edge cover that includes $[7,61]$ and $[9,61]$, and excludes $[7,63]$ and $[9,61]$.

Note, however, that such a reduction of the edit script problem to an edge cover (and thus, weighted matching) problem cannot be exact, given the hardness of the edit script problem.[3] Indeed, our method of assigning costs to edges of the induced graph (Section 5.1) is only approximate, and thus the minimum-cost edge cover is not guaranteed to produce the best solution for the edit script problem.

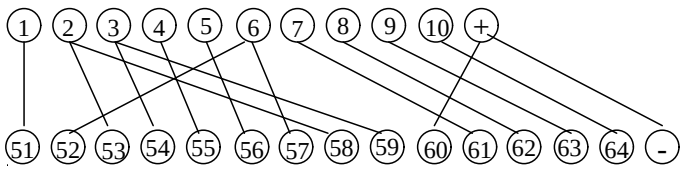

Figure 5: A minimum-cost edge cover of the induced graph in Figure 4

3.4 Generating the Edit Script

Returning to the pruned induced graph of our running example, let us assume that we have gone through the process of determining the cost of each edge, and have computed a minimum-cost edge cover according to these costs, obtaining the edge cover depicted in Figure 5. Our next step consists of using this edge cover to compute an edit script that transforms the tree T_1 to the tree T_2. We do this in two steps. First, we tag each edge of the edge cover with an annotation that represents the edit operation suggested by that edge. During this annotation process, we also add some ordering constraints between the annotations on different edges. Once we have annotated all edges in this manner, we topologically sort the annotations (based on the ordering constraints added by the annotation algorithm) to generate the final edit script. The annotation algorithm is described in Section 4.3. We illustrate some of the ideas used by the algorithm by considering its action on an edge in the edge cover for our running example.

Consider the edge $e_1 = [6,52]$ of the edge cover in Figure 5. In Figure 6, we depict this edge in relation to the original trees. We also depict the edges $[4,55]$ and $[6,57]$, from the edge cover. (The edge cover edges are shown as dashed lines in Figure 6; for clarity we do not show all of them.) We observe that there is one other edge in the edge cover that is incident on node 6, viz. $[6,57]$, suggesting that the node 6 was copied either directly, or indirectly (due to one of its ancestors being copied). Furthermore, we note that the parent (node 4) of node 6 is matched to the parent (node 55) of node 57 (i.e., the edge [4, 55] exists in the edge cover), while the parent of node 52 is *not* matched to the parent of node 6. This matching of the parents suggests that node 52 is the original instance of node 6, while node 57 is the copy. We record this fact by annotating the edge $[6,52]$ with a CPY mark, and the edge $[6,57]$ with a NIL mark (indicating that no edit operation is involved in the matching suggested by this edge).

Proceeding thusly, we annotate all the edges in the edge cover of our running example, to

[3] unless $\mathcal{P} = \mathcal{NP}$, since we are considering a polynomial-time reduction.

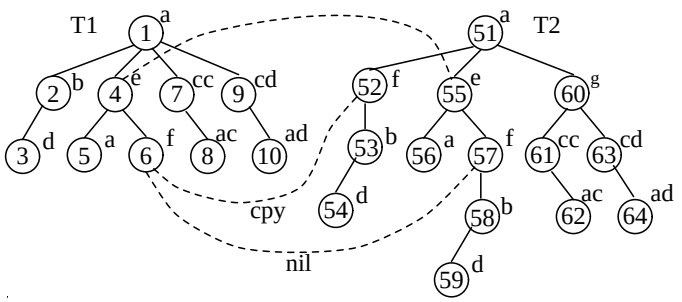

Figure 6: Annotating edges in the edge cover of Figure 5

obtain the annotated edge cover depicted in Figure 7, which shows only the edges with non-nil annotations, for clarity. Our annotation algorithm also discovers an ordering constraint that requires the MOV operation to precede the CPY operation. As the final step, we generate the edit operation corresponding to each annotation, and topologically sort these operations based on the ordering constraint, to yield the required edit script. In our example, one possible ordering of the edit operations that satisfies the ordering constraint is $(\text{INS}(g, 1, 7, 9), \text{MOV}(2, 6), \text{CPY}(6, 1))$. We see that this edit script is identical to the one in Example 2.1, which happens to be a minimum cost edit script for our example. There is another ordering of the above edit operations that satisfies the ordering constraint: $(\text{MOV}(2, 6), \text{CPY}(6, 1), \text{INS}(g, 1, 7, 9))$. Both edit scripts have the same final effect, and have the same cost. In general, all edit scripts that satisfy the ordering constraints among the annotations have the same overall effect and the same cost.

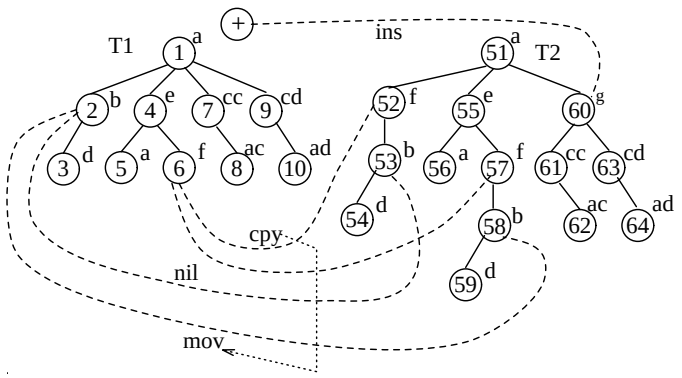

Figure 7: Annotated edges of the edge cover of Figure 5

For the above example MH-DIFF produces a minimum-cost edit script, but it may sometimes not find one with globally minimum cost. In Section 6 we evaluate how often this happens and we briefly discuss how one could perform additional searching in the neighborhood of the script found by MH-DIFF .

This concludes the overview of MH-DIFF. To summarize, the process consists of constructing an induced graph from the input trees, pruning the induced graph, finding a minimum-cost edge cover of the pruned induced graph, annotating this edge cover to generate edit operations, and finally,

ordering these edit operations to obtain an edit script. In the following sections, we describe these phases in detail. For ease of presentation, we present these phases in a different order than the order in which they are performed. In particular, in Section 4, we begin by formally defining the correspondence between and edit script and an edge cover of the induced graph. In that section, we also describe the method for generating an edit script from an edge cover of the induced graph. In Section 5, we describe how the cost of an edit script is distributed over the edges of the corresponding edge cover of the induced graph. In that section, we also describe how this cost function is approximated by deriving upper and lower bounds on the cost of an edge of the induced graph, and how these bounds are used to prune the induced graph. Since finding a minimum-cost edge cover for a bipartite graph with fixed edge costs is a problem that has been previously studied in the literature [PS82, Law76], we do not present the details in this paper.

4 Edge Covers and Edit Scripts

In this section, we describe algorithm *Annotate*, which generates an edit script between two trees, given an edge cover of their induced graph. Before we can describe this algorithm, we need to understand the relationship between an edit scripts between two trees and edge covers of their induced graph. Therefore, we first define the edge cover induced by an edit script. That is, we describe how, given an edit script between two trees, we generate an edge cover of the induced graph. (Note that this process is the reverse of the process the algorithm *Annotate* performs. However, a definition of this reverse process is needed for the description of the algorithm.) Next, we study some important properties of edit scripts and describe how they translate to properties of the induced edge covers. Finally, we present the algorithm *Annotate*, which uses these properties to generate a minimum-cost edit script from a given edge cover.

4.1 Edge Cover Induced by an Edit Script

In Section 3, we introduced the graph induced by two trees T_1 and T_2 as the complete bipartite graph $B = (U, V, U \times V)$, with $U = N_1 \cup \{\oplus\}$ and $V = N_2 \cup \{\ominus\}$ (where N_1 and N_2 are the nodes of T_1 and T_2, respectively). Let $\mathcal{E}$ be an edit script that transforms T_1 to T_2; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. We now define the edge cover $K(\mathcal{E})$ induced by $\mathcal{E}$. Intuitively, we obtain $K(\mathcal{E})$ as follows. Create a copy T_3 of T_1, and introduce an edge between each node in T_1 and its copy in T_3. Apply the edit script to T_3, moving, copying, etc. the end-points of the edges with the nodes they are attached to as nodes are moved, copied, etc. Thus, when an a node $n \in T_3$ is copied, producing node n', any edge $[m, n]$ is split to produce an new edge $[m, n']$. The other edit operations are handled analogously. Furthermore, an edge between the special nodes $\oplus$ and $\ominus$ is added initially, and removed when it is no longer needed to cover either $\oplus$ or $\ominus$. Due to space limitations, the formal definition of the edge cover induced by an edit script is relegated to Appendix B. Here, we present an example that illustrates the definition of the edge cover induced by an edit script.

Example 4.1 Consider the edit script from Example 2.1, and the initial tree T_1 from Figure 1. As described above, our first step consists of creating a copy T_3 of T_1, and adding an edge between each node of T_1 and its counterpart in T_3. We also add the special nodes $\oplus$ and $\ominus$, along with an edge connecting them. The result of this step is depicted in Figure 8. For clarity in presentation, the edges between the nodes of T_1 and their counterparts in T_3 are not shown in Figure 8; instead,

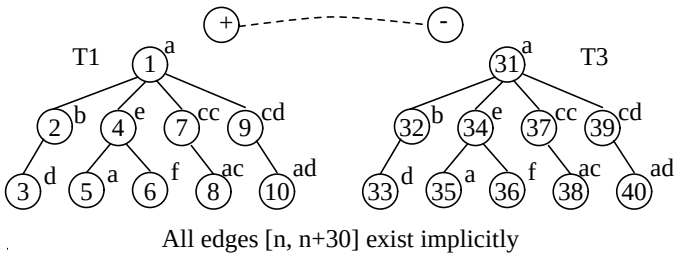

Figure 8: Example 4.1: the edge cover before the action of any edit operations

we encode these edges using the node identifiers of T_1 and T_2. That is, as indicated in the figure, imagine an edge $[n, n+30], \forall n = 1 \ldots 10$.

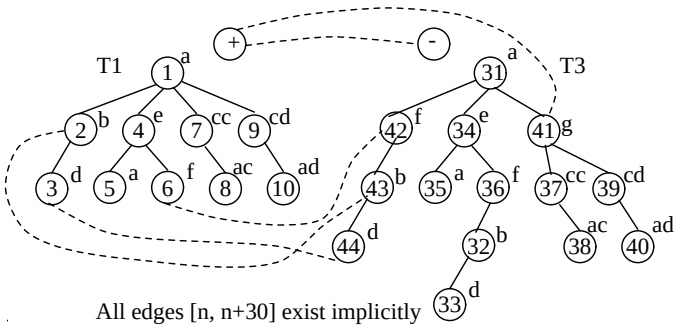

Figure 9: Example 4.1: the final induced edge cover

Our next step consists of applying the edit script from Example 2.1 to the tree T_3. To enable this application of the edit script for T_1 to T_3, we change the node identifiers in the edit script from the identifiers of the nodes of T_1 to those of T_2, obtaining $\mathcal{E}_1 = (\text{INS}(41, g, 31, \{37, 39\}), \text{MOV}(32, 36), \text{CPY}(36, 31))$. As a result of the INS operation, a node with identifier 41 and label g is inserted as a child of node 31, and nodes 37 and 39 are made its children. In addition, we add an edge $[\oplus, 41]$ to the induced edge cover. Next, consider the action of the MOV operation, which moves node 32 to become a child of node 36. This operation does not add any new edges to the edge cover. (The existing edges $[2, 32]$ and $[3, 33]$ continue to exist.) Finally, the CPY operation creates a copy of the subtree rooted at node 36, and inserts this copy as a child of node 31. In addition, the edges $[6, 42]$, $[2, 43]$, and $[3, 44]$ are added to the edge cover. The result is depicted in Figure 9, (which also omits edges $[n, n+30], \forall n = 1 \ldots 10$). Note that the transformed tree T_3 is now isomorphic to the tree T_2 in Example 2.1, so that essentially, we now have an edge cover of the induced graph of T_1 and T_2.

4.2 Using Edge Covers

The goal of using an edge cover is that it should capture the essential aspects of an edit script; that is, no important information should be lost in going from an edit script to the edge cover induced by it. However, there are certain edit scripts for which this property does not hold. For example, consider an edit script $\mathcal{E}_2$ that inserts a node p as the parent of ten siblings (children of the same

parent) $n_1, \ldots, n_{10}$, then moves p to another location in the tree, and finally deletes p. The node p is absent from both the initial tree and the final tree. Therefore, an edge cover of the initial and final trees contains no record of the temporary insertion of node p. Thus, we have lost some information in going from $\mathcal{E}_2$ to the edge cover.

Is the fact that our edge covers cannot capture edit scripts like $\mathcal{E}_2$ a problem? On the one hand, $\mathcal{E}_2$ could be the minimum cost edit script MH-DIFF is trying to find. For example, say that insert, delete, and move operations all cost one unit. The cost of $\mathcal{E}_2$ would then be the cost of one insert, plus the cost of one move, plus the cost of one delete, for a total cost of 3. If we do not use the "bulk move trick" that $\mathcal{E}_2$ uses, we need to move each of $n_1, \ldots, n_{10}$ individually, for a cost of 10. Thus, $\mathcal{E}_2$ could be the minimum cost edit script, and if we rule it out, then MH-DIFF would miss it.

On the other hand, scripts like $\mathcal{E}_2$ do not represent transformations that are meaningful or intuitive to an end user. In other words, if a user saw $\mathcal{E}_2$, he would not understand why node p was inserted, since it really has no function in his application. True, the costs provided by the user are intended to describe the desirability of edit operations, but if we abuse these numbers we can end up with "tricky" scripts like $\mathcal{E}_2$ that are more confusing than helpful.

Our decision for MH-DIFF is to rule out "tricky" scripts like $\mathcal{E}_2$, even if we miss some low-cost scripts. Because of this restriction, our edge covers are indeed able to capture all edit scripts of interest. The two properties below describe precisely the scripts we are considering for MH-DIFF.

Property 1 Any given node is operated on by at most one structure-changing operation (INS, DEL, MOV, CPY, and GLU).

This property states that once a node is inserted, it is not moved, deleted, copied, or glued. Similarly, a node may be moved at most once. Note, however, that this restriction applies only to the nodes directly operated upon, not to the nodes in their subtrees (which are operated upon indirectly). For example, if a node is moved, the nodes in its subtree (which get moved with it) are not subject to this restriction, and can be moved, copied, deleted, etc.

Property 2 A node that is either the source or the target of an indirect CPY operation (that is, a node that is copied as a result of a CPY operation on one of its ancestors) is neither the source nor the target of a (direct or indirect) GLU operation.

One of the important consequences of Properties 1 and 2 is that we can restrict our attention to *minimal* edge covers. Define a *minimal edge cover* to be a an edge cover K such that no proper subset of K is an edge cover. We then have the following Lemma, proved in Appendix D:

Lemma 4.1 If $\mathcal{E}$ is an edit script satisfying Properties 1 and 2, then the edge cover $K(\mathcal{E})$ induced by $\mathcal{E}$ is a minimal edge cover.

Minimal edge covers of a graph have the following property that is useful in the description of our algorithm in Section 4.3:

Lemma 4.2 If K is a minimal edge cover of a graph, K does not contain any path of length three.

Thus, the edges of any minimal edge cover can be partitioned such that all edges in each partition are incident on a common node; that is, they have a "star" configuration. For a bipartite graph,

For each edge $e = [m, n] \in K$ not already annotated, do:
  Let $M = \{m' \in T_1^+ : [m', n] \in K\}$
    $N = \{n' \in T_2^+ : [m, n'] \in K\}$
  Case 1: $|M| = 1 \wedge |N| = 1$
    Case 1.1: $M = \{\oplus\},\ N = \{n\},\ n \neq \ominus$
      $A(\oplus, n) \leftarrow$ INS
    Case 1.2: $M = \{m\},\ m \neq \oplus,\ N = \{\ominus\}$
      $A(m, \ominus) \leftarrow$ DEL
    Case 1.3: $M = \{\oplus\},\ N = \{\ominus\}$
      $A(\oplus, \ominus) \leftarrow$ NIL
    Case 1.4: $M = \{m\},\ m \neq \oplus,\ N = \{n\},\ n \neq \ominus$
      if $[p(m), p(n)] \in K$ then $A(m, n) \leftarrow$ NIL
      else $A(m, n) \leftarrow$ MOV
  Case 2: $|M| = 1 \wedge |N| > 1$
    Case 2.1: $M = \{m\} \wedge m \neq \oplus$.
      See Figure 16, Appendix D
    Case 2.2: $M = \{\oplus\}$.
      See Figure 17, Appendix D.
  Case 3: $|M| > 1 \wedge |N| = 1$
    Analogous to Case 2.
  Case 4: $|M| > 1 \wedge |N| > 1$
    Not possible because K is minimal; see Lemma 4.2.
For each edge $e = [m, n] \in K$, do:
  If $l(m) \neq l(n)$ then $A(m, n) \leftarrow A(m, n)$.UPD

Figure 10: Algorithm *Annotate*

we call these partitions of an edge cover *flowers*. The common node of the edges in each flower we call the *base*, and the node of each edge that is not a base, we call a *petal*. This terminology is useful for explaining the algorithm *Annotate* in Section 4.3.

4.3 Generating an Edit Script from an Edge Cover

We now describe how, given a minimal edge cover K of the graph induced by trees T_1 and T_2, we compute a minimum-cost edit script corresponding to this edge cover. Our method has two steps. In the first step, we mark each edge in the given edge cover with an *annotation* that describes the edit operations (if any) mandated by that edge. During this step, we will also discover the ordering constraints over the edit operations represented by these annotations. The second step consists of generating the edit operation corresponding to each annotation, and ordering them in a way that satisfies the ordering constraints discovered in the first step.

Algorithm *Annotate*

Input: A minimal edge cover K of the bipartite graph $B(T_1, T_2)$ induced by trees T_1 and T_2.

Output: An annotated edge cover $K_a = \{[m, n, l] : [m, n] \in K \wedge l \in A^+\}$, where
$A^+ = \{$NIL, INS, DEL, UPD, MOV, CPY, GLU, MOV.UPD, CPY.UPD, GLU.UPD$\}$. These annotations rep-

resent the edit operations with the corresponding names, while the compound annotations (e.g., MOV.UPD) represent two edit operations (e.g., a MOV operation and an UPD operation). Also, a set of ordering constraints D between the annotations, such that ordering the edit operations represented by the annotations subject to the ordering constraints D produces an edit script $\mathcal{E}$ such that $K(\mathcal{E}) = K$, and such that $K(\mathcal{E}') = K \Rightarrow c(\mathcal{E}) \leq c(\mathcal{E}')$.

Method: Let $T_1^+ = T_1 \cup \oplus$, and let $T_2^+ = T_2 \cup \ominus$. We consider each edge $e = [m, n]$ of the edge cover K in turn, and consider the different cases based on the degrees of m and n, as suggested by the pseudo-code in Figure 10. Consider Case 1 of the algorithm, which corresponds to an edge $e = [m, n]$ such that there is no other edge in K incident on m and n. If m is the special node $\oplus$, this edge suggests that n is inserted, since it does not correspond to any real node in T_1. We therefore annotate e with INS. The sub-case where n is the special node $\ominus$ is analogous. The only interesting sub-case is when both m and n are regular tree nodes. In this case, if the parents of m and n do not match (i.e., there is no edge between them in K), we need to move m to the proper parent.

Consider now Case 2.1 of the algorithm, which is the case of an edge $[m, n]$ such that there are a number of other edges incident on m. Clearly, all but one of the nodes that match m are produced as a result of copy operations. Naively, we may be therefore be tempted to annotate all but one of the edges incident on m with CPY annotations. However, note that we a node may be copied both directly (by a copy operation acting on it) and indirectly, by virtue of being (at some point during the execution of the edit script) in the subtree of some other node that is copied. We call copies obtained indirectly in such a manner "free" copies. Furthermore, note that the *only* way in which we can get a copy of a node m without directly copying m is by copying some node a such that, at the time of the copy, m is in the subtree of a.

Another key observation is the following: If n and n' are two nodes matched to m, and if neither n nor n' is copied *directly*, then either $p(m)$ is matched to two or more nodes in T_2, or $p(n)$ and $p(n')$ are both matched to the same node m' in T_1 (or both). If $m' = p(m)$, we get the copies of m for free because some ancestor of m is copied to some ancestor of n and some ancestor of n'. If $m' \neq p(m)$, we get a free copy of m by moving m to m' before m' is copied. Note this argument applies to the case where we have more than two copies of m too. In general, therefore, we group the copies n_i of m by the node to which their parents match. We call such groups *copy-flowers*.

Each such copy-flower potentially generates a number of free copies (equal to the number of nodes in the flower). However, in the case where $m' \neq p(m)$, we need to move a copy of m from some other location to m'. Copies of m for this purpose are obtained from free copies generated by an ancestor of m being copied. When such free copies are in short supply, it is prudent to allocate them to the larger copy-flowers, since that generates a greater number of free copies. The only remaining complexity is the effect of insertions. If the node m' above is the special node $\oplus$, we need to continue upwards in the tree, looking for a non-$\oplus$ ancestor in order to get a free copy. (Recall that inserted nodes may not be copied.) The pseudo-code in Figure 16 in Appendix C systematically explores all the above possibilities, and arrives at the best possible way of annotating the set of edges incident on m.

Case 2.2 of the algorithm is quite similar to Case 2.1, the only difference being that the node m is now the special node $\oplus$. Finally, Case 3 is completely analogous to Case 2. Due to space constraints, we do not discuss these cases in detail in this paper. Instead, we illustrate the operation of Case 2 on our running example from Section 3.

Example 4.2 Consider the running example from Section 3, with the edge cover depicted in

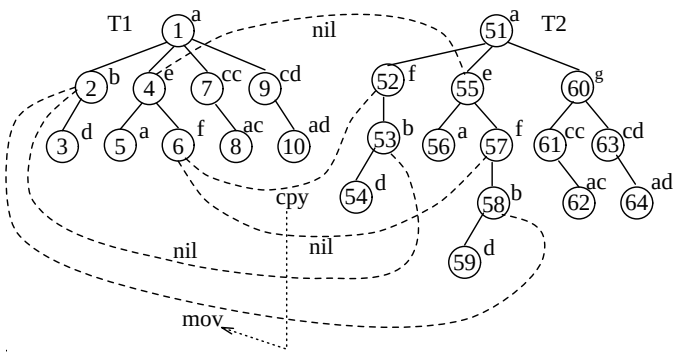

Figure 11: Action of Case 2.1 of Algorithm *Annotate* for Example 4.2

Figure 5. In Section 3, we saw how the CPY annotation in Figure 6 is determined. To illustrate Case 2.1, let us now consider annotating the edge $[2, 53]$, depicted in Figure 11. Consulting the edge cover in Figure 5, we note that there is one other edge, $[2, 58]$, incident on the node 2; thus, Case 2.1 of algorithm *Annotate* is applicable. We note that the parent (node 51) of node 53, and the parent (node 57) of node 58 match the same node (node 6) in T_1. We therefore group nodes 53 and 58 together, and look for a "spare" copy (the set S in Figure 16) of node 2 that we can move to under node 6. We discover that the default copy of node 2 (viz., the node that would have appeared as a child of the root in tree T_2) is not present, and is thus available as a spare. We therefore resolve to move it to node 6 by annotating the edge $[2, 58]$ with a MOV annotation. The second edge, $[2, 53]$, is annotated with NIL, since this copy of node 2 will be made for free when node 6 is copied. This, of course, assumes that the CPY operation occurs after the MOV operation, which is recorded by a dependency from the CPY annotation to the MOV annotation (indicated by the dotted arrow in Figure 11).

5 Finding the Edge Cover

In this section we describe how MH-DIFF finds a minimal edge cover of the induced graph. The resulting cover will serve as input to algorithm *Annotate* (Section 4). Our goal is to find not just any minimal edge cover, but one that corresponds to a minimum-cost edit script. Let us call such an minimal edge cover the *target cover*.

Consider an edge e in our pruned induced graph. To get to the target cover, MH-DIFF must decide whether e should be included in the cover. To reach this decision, it would be nice if MH-DIFF knew the "cost" of e. That is, if e remains in the target cover, then it would be annotated (by algorithm *Annotate*) with some operation of a given cost, and we could say that this is the cost of e. Unfortunately, we have a "chicken and the egg problem" here: *A*nnotate cannot run until we have the target cover, and we cannot get the target cover until we know the costs it will imply. To break the impasse, our approach uses the following idea:

Instead of trying to compute the actual cost of e, we compute an upper and lower bound to this cost. These bounds can be computed without the knowledge of which other edges are included in the target cover, and serve two purposes: Firstly, they allow us to design pruning rules that are used to (conservatively) eliminate unnecessary edges from the induced graph. Secondly, after

pruning, the bounds can guide our search for the target cover.

As an enhancement, we actually use a variation on the cost of edge e suggested above. The following example shows that simply "charging" each annotation to the edge it is on is not entirely "fair." We are given a tree T_1 containing two nodes, n_1 and n_2 with the same label l. Furthermore n_1 has children n_{11} and n_{12} with labels a and b, respectively, and n_2 has children n_{21} and n_{22} with labels c and d, respectively. Suppose T_2 is a logical copy of T_1. (That is, T_1 and T_2 are isomorphic.) Consider an edge cover that matches each node in T_1 to its copy in T_2 except that it "cross matches" n_1 and n_2 across the trees, as shown in Figure 12. Given this edge cover, algorithm *Annotate* will produce a move operation for each of the nodes n_{11}, n_{12}, n_{21}, and n_{22}. However, these move operations were caused not by the mismatching of the nodes n_{11}, n_{12}, n_{21}, or n_{22}, but instead, by the mismatching of n_1 and n_2. Therefore it would be intuitively more fair to charge these move operations to the edges responsible for the mismatch, viz. $[n_1, n_2']$ and $[n_2, n_1']$. To achieve this, we use the following scheme: If e is annotated with INS, DEL, or UPD in the target cover, we do charge e for this operation. However, if e is annotated by MOV, CPY, or GLU, then the *parent* of e, and not e is charged. We call the edge costs computed in such a fashion *fair costs*.

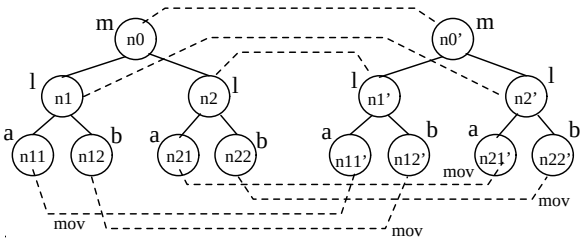

Figure 12: Distributing edge costs fairly

In summary, MH-DIFF first computes upper and lower bounds for the fair cost of each edge in the pruned induced graph. These bounds are then used to prune edges in the induced graph, and finally to search for the target cover. We begin by defining the fair cost of an edge below.

5.1 An Edge-wise Cost Function

Let K be an *annotated* minimal edge cover. For an edge $e \in K$, if the annotation on e is MOV, CPY, or GLU, let $c_x(e)$ denote the cost of that operation. (Recall that, given Properties 1 and 2, there can be at most one of these three annotations on a given edge.) If e is annotated with INS, DEL, or UPD, then let $c_s(e)$ denote the cost of the operation. Furthermore, let $E(m)$ be the set of edges in K that are incident on m, that is, $E(m) = \{[m, n] \in K\}$. Let $C(m)$ be the set of the children of m. We then define the *fair* cost of each edge $[m, n] \in K$ as follows:

$$
\begin{aligned}
c_K([m,n]) &= c_s(m,n) \\
&+ \frac{1}{2|E(m)|} \sum_{m' \in C(m)} \sum_{[m',n'] \in K} c_x([m',n']) \\
&+ \frac{1}{2|E(n)|} \sum_{n' \in C(n)} \sum_{[m',n'] \in K} c_x([m',n']) \qquad (1)
\end{aligned}
$$

Note that this cost depends on K, and thus is not a function of e alone. The following lemma, proved in Appendix D, states that the above scheme of distributing the cost of an edge cover over its component edges is a sound one; that is, adding up the cost edge-wise yields the overall cost of the edge cover.

Lemma 5.1 If K is an annotated, minimal edge cover of the graph induced by two trees, then $c(K) = \sum_{e \in K} c_K(e)$.

5.2 Bounds on Edge Costs

Although Lemma 5.1 suggests a method of distributing the cost of an annotated edge cover (and thus an edit script) over the component edges, the cost of each edge depends on the other edges present in the edge cover, and is thus not directly useful for computing a minimum-cost edge cover. However, we use that distribution scheme to derive upper and lower bounds on the fair cost $c_K(e)$ of an edge e over all minimal edge covers K.

Intuitively, given that the cost of any UPD annotation on an edge is charged to that edge (by Equation 1), a simple choice for the lower bound on the cost of an edge $[m, n]$ is simply the cost $c_u(m, n)$ of updating the label m to that of n. However, we can do a little better. In some cases, selecting an edge $[m, n]$ (as part of the edge cover being constructed) may *force* some of the children m' of m to be moved to n. In particular, this happens for those children of m' for which there is no edge that could possibly match m' to a child of n. We call such moves *forced moves*. In cases where we can determine a forced move exists, the cost of a MOV is added to the lower bound cost. However, according to Equation 1 not all the cost of a forced move goes to edge $[m, n]$. In the worst case, the number of edges incident on m, $|E(m)|$, is large, leaving $[m, n]$ with an insignificant contribution. However, if $|E(m)|$ is greater than 1, we know by Lemma 4.2 that $|E(n)| = 1$, so forced moves on the n side would contribute to $[m, n]$. Thus, we may add the minimum of the second and the third terms in Equation 1 to the lower bound function.

Formally, let E be the set of edges in the induced graph of T_1 and T_2.[4] We define the *forced move cost*, $c_{mf}(m', n)$ of a node $m' \in T_1$ with respect to another node $n \in T_2$ as follows: $c_{mf}(m', n) = c_m$, if $\nexists n' \in C(n)$ such that $[m', n'] \in E$, and 0 otherwise. (The cost $c_{mf}(m, n')$ is defined analogously.) We then define the *lower bound fair cost*, c_{lb}, of an edge as follows:

$$c_{lb}([m, n]) = c_u(m, n) + \frac{1}{2} \min \left\{ \sum_{m' \in C(m)} c_{mf}(m', n), \sum_{n' \in C(n)} c_{mf}(m, n') \right\} \tag{2}$$

To help us compute the upper bound, let us now define a *conditional move cost*, $c_{m?}$. Intuitively, $c_{m?}(m', n)$ costs one MOV cost unless there is a partner of m' that is a child of n. Formally, $c_{m?}(m', n) = 0$, if $\exists n' \in C(n)$ such that $[m', n'] \in E$, and c_m otherwise. The cost $c_{m?}(n', m)$ is defined analogously. Furthermore, define $c_w(m, n) = c_u(m, n)$ if m and n are regular nodes, 0 if $(m = \oplus) \wedge (n = \ominus)$, c_i if$(m = \oplus) \wedge (n \neq \ominus)$, and c_d if $(m \neq \oplus) \wedge (n = \ominus)$.

[4] As we will see later, although E initially includes all edges in the complete bipartite graph, the pruning of edges results in successive reduction of the size of E.

Using reasoning similar to the one used for deriving the lower bound cost above, we arrive at the following definition for the *upper bound fair cost*, c_{ub}, of an edge:

$$\begin{aligned} c_{ub}([m,n]) &= c_w(m,n) \\ &+ \frac{1}{2}\sum_{m' \in C(m)} (c_c(|E(m')|-1) + c_{m?}(m',n)) \\ &+ \frac{1}{2}\sum_{n' \in C(n)} (c_g(|E(n')|-1) + c_{n?}(n',m)) \end{aligned} \tag{3}$$

Note that both $c_{ub}(e)$ and $c_{lb}(e)$ can be computed by MH-DIFF without knowing the target cover. Furthermore, the following lemma, proved in Appendix D, states that the above definitions of $c_{ub}(e)$ and $c_{lb}(e)$, are upper and lower bounds, respectively, on the fair cost contribution $c_K(e)$ of edge e to *any* minimal edge cover K that contains e.

Lemma 5.2 Let $B = (U,V,E)$ be the bipartite graph induced by trees T_1 and T_2. Let $B' = (U,V,E')$, where $E' \subseteq E$. Let $\mathcal{K}$ denote the collection of all minimal edge covers of B'. We then have the following inequalities:

$$c_{lb}(e) \leq \min_{K \in \mathcal{K}} c_K(e) \quad \text{and} \quad c_{ub}(e) \geq \max_{K \in \mathcal{K}} c_K(e)$$

5.3 Pruning Rules

We now use the upper and lower bound functions for the cost of an edge as defined above to introduce the pruning rules we use to reduce the size of the induced graph of the two trees being compared. Let $e_1 = [m,n]$ be any edge in the induced graph, as shown in Figure 13. Let e_2 be any edge incident on m, and let e_3 be any edge incident on n. Intuitively, our first pruning rules tries to remove edges with a lower bound cost that is so high that it is preferable to match each of its nodes using some other edges, given the existence of such edges with a suitably low upper bound cost.

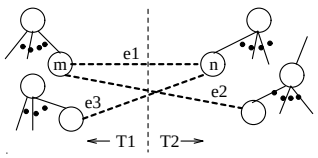

Figure 13: Applying pruning rules

Pruning Rule 1 Let $C_t = \max\{c_m, c_c, c_g\}$. If $c_{lb}(e_1) \geq c_{ub}(e_2) + c_{ub}(e_3) + 2C_t$ then prune e_1.

Example 5.1 To illustrate this rule, consider a tree T_1 containing, among others, two childless nodes 1 (label f) and 2 (label g). Similarly, T_2 contains childless nodes 3 (label g) and 4 (label f), among others. Say the costs c_m, c_c, and c_g are one unit each, while the update costs are $c_u(f,g) = 3$, and $c_u(f,f) = c_u(g,g) = 0$. Let us now consider if edge $e_1 = [1,3]$ can be pruned

because edges $e_2 = [1,4]$ and $e_3 = [2,3]$ exist. Since the nodes have no children, it is easy to compute $c_{lb}(e_1) = c_u(f,g) = 3$, $c_{ub}(e_2) = c_u(f,f) = 0$, and $c_{ub}(e_3) = c_u(g,g) = 0$. Since $C_t = 1$, we see that Pruning Rule 1 holds and e_1 can be safely removed. The intuition is that in the worst case we can replace e_1 by edges e_2 and e_3. Using the latter edges could introduce at most the costs $c_{ub}(e_2)$ and $c_{ub}(e_3)$, plus the cost of two MOV, CPY, or GLU operations. The last factor can arise for instance if node 2 ends up being matched not just to node 3 but to another node in T_2. This means that node 2 needs to be copied, which would not have been necessary if we had kept edge e_1 and not used e_2. Similarly, the removal of edge e_1 may cause an extra glue operation for node 4. However, even in this worst case scenario, the costs would be less than the cost of updating the label of node 1 to that of node 2, so we can safely remove the $[1,2]$ edge.

Our second pruning rule (already illustrated in Section 3) states that if it is less expensive to delete a node and insert another, we do not need to consider matching the two nodes to each other. More precisely, we state the following:

Pruning Rule 2 If $c_{lb}(e_1) \geq c_d(m) + c_i(n)$ then prune e_1.

Note that the above pruning rules are simpler to apply if we let e_2 and e_3 be the minimum-cost edge incident on m and n, respectively. The following lemma, proved in Appendix D, tells us that the pruning rules are conservative:

Lemma 5.3 Let E_p be the set of edges pruned by repeated application of Pruning Rules 1 and 2. Let K_1 be any minimal edge cover of the graph B. There exists a minimal edge cover K_2 such that (1) $K_2 \cap E_p = \emptyset$, and (2) $C(K_2) \leq C(K_1)$.

The pruning phase of our algorithm consists of repeatedly applying Pruning Rules 1 and 2. Note that the absence of edges raises the lower bound function, and lowers the upper bound function, thus possibly causing more edges to get pruned. Our algorithm updates the cost bounds for the edges affected by the pruning of an edge whenever the edge is pruned. By maintaining the appropriate data structures, such a cost-update step after an edge is pruned can be performed in $O(logn)$ time, where n is the number of nodes in the induced graph.

5.4 Computing a Min-Cost Edge Cover

After application of the pruning rules described above, we obtain a pruned induced graph, containing a (typically small) subset of the edges in the original induced graph. In favorable cases, the remaining edges contain only one minimal edge cover. However, typically, there may be several minimal edge covers possible for the pruned induced graph. We now describe how we select one of these minimal edge covers.

We first *approximate* the fair cost of every edge e that remains after pruning by its lower bound $e_{lb}(e)$. (We could have also use the upper bound, or an average of both bounds, since this is just an estimate.) Then, given these constant estimated costs, we compute a minimum-cost edge cover by reducing the edge cover problem to a bipartite weighted matching problem, as suggested in [PS82]. Since the weighted matching problem can be solved using standard techniques, we do not present the details in this paper, noting only that given a bipartite graph with n nodes and e edges, the weighted matching problem can be solved in time $O(ne)$. For our application, e is the number of edges that remain in the induced graph after pruning.

6 Implementation and Performance

In this section, we describe our implementation of MH-DIFF, and discuss its analytical and empirical performance. Figure 14 depicts the overall architecture of our implementation, with rectangles representing the modules (numbered, for reference) of the program, and other shapes representing data. Given two trees T_1 and T_2 as input, Module 1 constructs the induced graph (Section 3.1). This induced graph is next pruned (Module 2) using the pruning rules of Section 5.3 to give the pruned induced graph. In Module 2, the update cost for each edge in the induced graph is computed using the domain-dependent comparison function for node labels (Section 2.2). The next three modules together compute a minimum-cost edge cover of the pruned induced graph using the reduction of the edge cover problem to a weighted matching problem [PS82]. That is, the pruned induced graph is first translated (by Module 3) into an instance of a weighted matching problem. This weighted matching problem is solved using a package (Module 4) [Rot] based on standard techniques [PS82]. The output of the weighted matching solver is a minimum-cost matching, which is translated by Module 5 into K_0, a minimum-cost edge cover of the pruned induced graph. Next, Module 6 uses the minimum-cost edge cover computed, to produce an annotated edge cover, with edge annotations representing edit operations (Section 4.3), and a set of ordering constraints (dependencies). Finally, Module 7 performs a topological sort of the annotations based on the dependencies, resulting in the desired edit script that transforms T_1 to T_2.

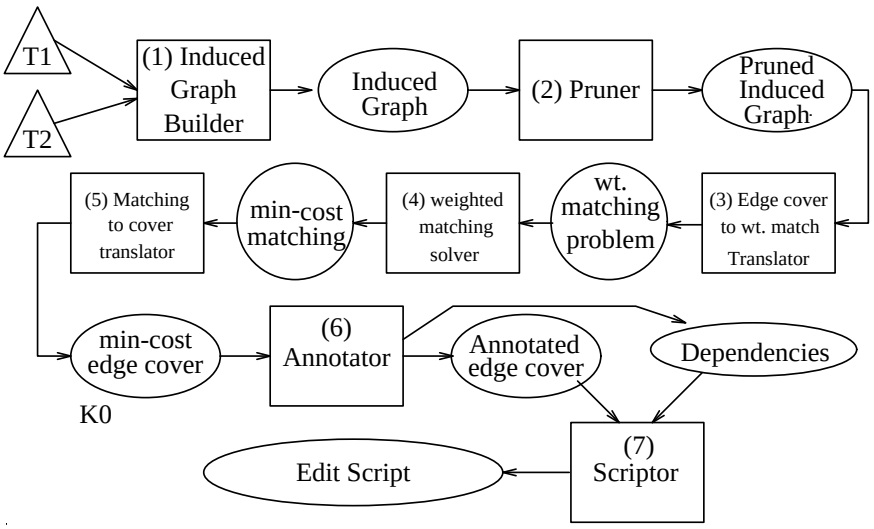

Figure 14: System Architecture

Recall that since we use a heuristic cost function to compute a minimum-cost edge cover, the edge cover produced by our program, and hence the edit script may not be the optimal one. We have also implemented a simple search module that starts with minimum-cost edge cover K_0 (see Figure 14) computed by our program and explores its neighborhood of minimal edge covers in an effort to find a better solution. The search proceeds by first exploring minimal edge covers that contain only one edge not in K_0. Next, we explore minimal edge covers containing two edges not in K_0, and so on. The intuition is that we expect the optimal solution to be "close" to the initial solution K_0. Although, in the worst case, such an exploration may be extremely time-consuming,

note that as a result of pruning edges, the search space is typically much smaller than the worst case. Due to space constraints, we do not describe the details of this search phase in this paper.

We have used our implementation to compute the differences between query results as part of the *Tsimmis* [CGMH+94] and C^3 [WU95] projects at Stanford. These projects use the OEM data model, which is a simple labeled-object model to represent tree-structured query results. In particular, we have run our system on the output of *Tsimmis* queries over a bibliographic information source that contains information about database-related publications in a format similar to BibTeX. Since the data in this information source is mainly textual, we treat all labels as strings. For the domain-dependent label-update cost function, we use a weighted character-frequency histogram difference scheme that compares strings based on the number of occurrences of each character of the alphabet in them. For example, consider comparing the labels "foobar" and "crowbar." The character-frequency histograms are, respectively, $(a:1, b:1, f:1, o:2, r:1)$ and $(a:1, b:1, c:1, o:1, r:2, w:1)$. The difference histogram is $(c:-1, f:1, o:1, r:-1, w:-1)$. Adding up the magnitudes of the differences gives us 5, which we then normalize by the total number of characters in the strings (13), and scale by a parameter (currently 5), to get the update cost $(5/13) * 5 = 1.9$.

Let us now analyze the running time of our program. Let n be the total number of nodes in both input trees T_1 and T_2. Constructing the induced graph (Module 1, in Figure 14) involves building complete bipartite graph with $O(n)$ nodes on each side. We also evaluate the domain-dependent label-comparison function for each pair of nodes, and store this cost on the corresponding edge. Thus, building the induced graph requires time $O(kn^2)$, where k is the cost of the domain-dependent comparison function. Next, consider the pruning phase (Module 2). By maintaining a priority queue (based on edge costs) of edges incident on each node of the induced graph, the test to determine whether an edge may be pruned can be performed in constant time. If the edge is pruned, removing it from the induced graph requires constant time, while removing it from the priority queues at each of its nodes requires $O(logn)$ time. When an edge $[m, n]$ is pruned, we also record the changes to the costs $c_{m?}(m, p(n))$, $c_{m?}(n, p(m))$, $c_{mf}(m, p(n))$, and $c_{mf}(n, p(m))$, which can be done in constant time. Thus, pruning an edge requires $O(logn)$ time. Since at most $O(n^2)$ are pruned, the total worst case cost of the pruning phase is $O(n^2 logn)$. Let e be the number of edges that remain in the induced graph after pruning. The minimum-cost edge cover is computed in time $O(ne)$ by Modules 3, 4, and 5. The annotation of the minimum-cost edge cover can be done in $O(n)$ time by Module 6. (Note that the number of edges in a minimal edge cover is always $O(n)$.) Finally, note that we have $O(1)$ ordering constraints per annotation, and $O(1)$ annotations per edge of the edge cover, so that ordering the edit operations corresponding to the annotations can be done in $O(n)$ time by Module 7 to produce the final edit script.

The number of edges that remain in the induced graph after pruning (denoted by e above) is an important metric for three main reasons. Firstly, as seen above, a lower number of edges results in faster execution of the minimum-cost edge cover algorithm. Secondly, a smaller number of edges decreases the possibility of finding a suboptimal edge cover, since there are fewer choices that need to be made by the algorithm. Thirdly, having a smaller number of edges in the induced graph drastically reduces the size of the space of candidate minimal edge covers that the search module needs to explore.

Given the importance of the metric e, we have conducted a number of experiments to study the relationship between e and n. We start with four "input" trees representing actual results of varying sizes from our *Tsimmis* system. For each input tree, we generate a batch of "output" trees by applying a number of random edits. The number of random edits is either 10% or 20% of the number of nodes in the input tree. Then for each output tree, we run MH-DIFF on it and its original

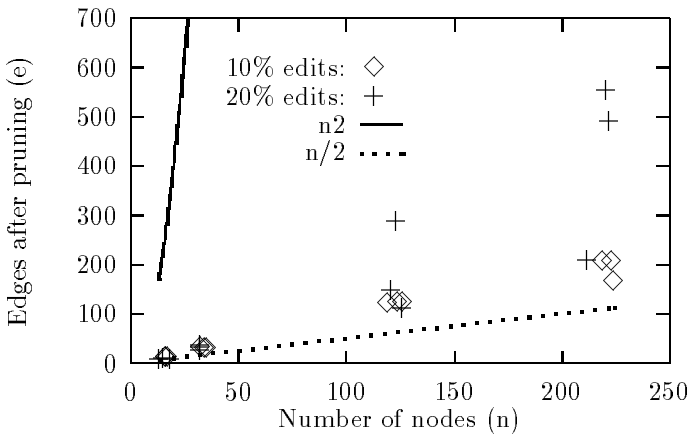

Figure 15: Number of unpruned edges as a function of the number of nodes

input tree. The results are summarized by the graph in Figure 15. The horizontal axis indicates the total number of nodes in the two trees being compared (and hence, in the induced graph). The vertical axis indicates the number of edges that remain after pruning the induced graph. Note that the ideal case (best possible pruning) corresponds to $e = \lceil n/2 \rceil$, since we need at least $\lceil n/2 \rceil$ edges to cover n nodes, whereas the worst case is $e = n^2$ (no pruning at all). For comparison, we have also plotted $e = n/2$ and $e = n^2$ on the graph in Figure 15. We observe that the relationship between e and n is close to linear, and that the observed values of e are much closer to $n/2$ than to n^2.

Note that in Figure 15 we have plotted the results for two different values of d, the percentage of random edit operations applied to the input tree. We see that, for a given value of n, a higher value of d results in a higher value of e, in general. We note that some points with a higher d value seem to have a lower value of e than the general trend. This is because applying d random edits is not the same as having the input and output trees separated by d edits, due to the possibility of redundant edit operations. Thus, some data points, even though they were obtained by applying d random edits, actually correspond to fewer changes in the tree.

We have also studied the quality of the initial solution produced by MH-DIFF. In particular, we are interested in finding out in what fraction of cases our method produces suboptimal initial solutions, and by how much the cost of the suboptimal solution exceeds that of the optimal. Given the exponential (in e) size of the search space of minimal edge covers of the induced graph, it is not feasible to try exhaustive searches on large datasets. However, we have exhaustively searched the space of minimal edge covers, and corresponding edit scripts, for smaller datasets. We ran 50 experiments, starting with an input tree T_1 derived as in the experiments for e above, and using 6 randomly generated edit operations to generate an output tree. We searched the space of minimal edge covers of the pruned induced graph exhaustively for these cases, and found that the MH-DIFF initial solution differed from the minimum-cost one in only 2 cases out of 50. That is, in 96% of the cases MH-DIFF found the minimum cost edit script, and of course it did this in much less time than the exhaustive method. In the two cases where MH-DIFF missed, the resulting script cost about 15% more that the minimum cost possible.

7 Related Work

The general problem of detecting changes from snapshots of data has been studied before from different angles. For example, [WF74] defines a string-to-string correction problem as the problem of finding the best sequence of insert, delete, and update operations that transform one string to another. The problem is developed further in [Wag75], which adds the "swap" operation to the list of edit operations. These papers also introduce the structure of a "trace" or a matching between the characters of the strings being compared as a useful tool for computing an edit script. A simpler change detection problem for strings, using only insertions and deletions as edit operations has been studied extensively [Mye86, WMG90]. The idea of a *longest common subsequence* replaces the idea of a trace in this simpler problem. A variant of the algorithm presented in [Mye86] for computing the longest common subsequence is implemented in the *gnudiff* [HHS+] program. All these algorithms work with strings, that is, with flat-file, or relational data, and are not suitable for computing changes in structured data.

In [ZS89, SZ90], the authors define a change detection problem for *ordered* trees, using insertion, deletion, and label-update as the edit operations, observing its added difficulty compared to the equivalent problem for strings; they also present an efficient dynamic-programming based algorithm to solve that problem. A proof of the $\mathcal{NP}$-hardness of a similar change detection problem (using insertion, deletion, and label-update) for *unordered* trees is presented in [ZWS95], which also presents an algorithm for a restricted version of the change detection problem. An important assumption made by the algorithms in [ZS89, SZ90, ZWS95] is that the cost of updating any label to any other label is always less than the cost of deleting a node with the old label and inserting a node with the new label. While this restriction is reasonable for some domains, it does not always lead to intuitive results. For example, consider two trees with the same structure, but completely different labels on the nodes (e.g., two trees representing different query results, but with a similar structure). Assuming the cost of label update is always lower than the cost of the corresponding insertion and deletion will result in an edit script that simply updates all the labels in the trees. While this is technically sound, it is not the semantically desirable result for this example.

In [CRGMW96] we defined a variant of the change detection problem for ordered trees, using subtree *moves* as an edit operation in addition to insertions, deletions, and updates, and presented an efficient algorithm for solving it. That algorithm uses domain characteristics to find a solution efficiently. A major drawback of the algorithm in [CRGMW96] is that it assumes that the number of duplicates (or near duplicates) in the labels found in the input trees is very small. Another drawback of of the algorithm in [CRGMW96] is that it assumes each node of the input trees has a special tag that describes its semantics. (For example, an ordered tree representing a document may have tags "paragraph," "section," etc.) Furthermore, that algorithm assumes the existence of a total order $<_t$ over these tags such that a node with tag t_1 cannot be the child of a node with tag t_2 unless $t_1 \leq t_2$. While these assumptions are reasonable in a text comparison scenario, there are many domains in which they do not hold.

The work presented in this paper differs from previous work in several important ways. Firstly, we detect the change detection problem for unordered trees, which is inherently harder than the similar problem for ordered trees. Secondly, we consider a rich set of edit operations, including copy and move operations, that make the edit script computed more meaningful and intuitively usable. Furthermore, we do not assume that the nodes of the input trees are "tagged" in a manner required by the algorithm in [CRGMW96], nor do we assume the absence of duplicates (or near duplicates) in the labels of the nodes in the input trees. Finally, we do not assume that the cost of

updating any label to any other label is always less than the cost of deletion and insertion.

8 Conclusion

We have described the need for computing semantically meaningful changes in structured data. We have introduced operations such as subtree copy and subtree move that allow us to describe changes to structured data more meaningfully than is possible by using only the traditional insert, delete, and update operations. We have formally defined the problem of computing a minimum-cost edit script, consisting of these operations, between two trees. To solve this problem, we have presented an algorithm that is based on representing an edit script between two trees as an edge cover of a bipartite graph induced by the trees. We have also studied the the performance of our algorithm both analytically and empirically. The experimental results, although preliminary, are very encouraging.

References

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[CRGMW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[HHS+] M. Haertel, D. Hayes, R. Stallman, L. Tower, P. Eggert., and W. Davison. The GNU diff program. Texinfo system documentation. Available through anonymous ftp from `prep.ai.mit.edu`.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[LGM96] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB Conference*, India, September 1996.

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

[PS82] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice-Hall, 1982.

[Rot] E. Rothberg. The *wmatch* program for finding a maximum-weight matching for undirected graphs. Live OR collection. Available at URL: `http://www.orsoc.org.uk/home.html`.

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

[Wag75] R. Wagner. On the complexity of the extended string-to-string correction problem. In *Seventh ACM Symposium on the Theory of Computation*, 1975.

[WF74] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of the Association of Computing Machinery*, 21(1):168–173, January 1974.

[WMG90] S. Wu, U. Manber, and G.Myers. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, September 1990.

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

[ZWS95] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.

A Definitions of Edit Operations

In this section, we present the formal definitions of the edit operations discussed in Section 2.1.

- An insertion operation is denoted by $\text{INS}(n, v, p, C)$, where n is the (unique) identifier of the new node, v is the label of the new node, $p \in N_1$ is the node that is to be the parent of n, and $C \subseteq C(p)$ is the set of nodes that are to be the children of n. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup \{n\}$, $p_2(n) = p$, $p_2(c) = n, \forall c \in C$, $p_2(c) = p_1(c), \forall c \in N_1 - C$, $l_2(n) = v$, and $l_2(m) = l_1(m), \forall m \in N_1$.

- A deletion operation is denoted by $\text{DEL}(n)$, where $n \in N_1$ and n is not the root of T_1. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$ with $N_2 = N_1 - \{n\}$, $p_2(c) = p_1(n), \forall c \in C(n)$, $p_2(c) = p_1(c) \forall c \in N_2 - C(n)$, and $l_2(m) = l_1(m), \forall m \in N_2$.

- An update operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\text{UPD}(n, v)$, where $n \in N_1$, and produces $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $p_2 = p_1$, $l_2(n) = v$, and $l_2(m) = l_1(m), \forall m \in N_2 - \{n\}$.

- A move operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\text{MOV}(n, p)$, where $n, p \in N_1$, and p is not in the subtree rooted at n.[5] The resulting tree is $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $l_2 = l_1$, $p_2(n) = p$, and $p_2(c) = p_1(c), \forall c \in N_2 - \{n\}$.

- A copy operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\text{CPY}(n, p)$, where $n, p \in N_1$, and n is not the root. Let $T_3 = (N_3, p_3, l_3)$ be a new tree that is isomorphic to the subtree of T_1 rooted at n, and let n' be the root of T_3. The result of the copy operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup N_3$, $l_2(m) = l_1(m), \forall m \in N_1$, $l_2(m) = l_3(m), \forall c \in N_3$, $p_2(n') = p$, $p_2(m) = p_1(m), \forall m \in N_1$, and $p_2(m) = p_3(m), \forall m \in N_3$.

- A glue operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by $\text{GLU}(n_1, n_2)$. Let T_3 be the subtree rooted at n_1, and let $T_4 = (N_4, p_4, l_4)$ be the subtree rooted at n_2. The precondition of this GLU operation is that T_4 is isomorphic to $T_3 - T_4$. The result of the glue operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 - N_4$, $p_2(c) = p_1(c), \forall c \in N_2$, and $l_2(c) = l_1(c), \forall c \in N_2$.

B Defining the Edge Cover Induced by an Edit Script

Let $\mathcal{E}$ be an edit script that transforms T_1 to T_2; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. We now define $K(\mathcal{E})$, the edge cover (of the induced graph of T_1 and T_2) induced by $\mathcal{E}$. Let T_3 be a tree that is isomorphic to T_1, with f being the isomorphism. Thus, $f : T_1 \rightarrow T_2$ is a one-to-one, onto function that preserves the parent-child and label relationships defining labeled trees. More precisely, $\mathit{label}(f(m)) = \mathit{label}(m)$, and $\mathit{parent}(f(m)) = f(\mathit{parent}(m))$ for all nodes $m \in T_1$. Let us extend f to $T_1 \cup \{\oplus\}$ and $T_2 \cup \{\ominus\}$ by defining $f(\oplus) = \ominus$). We will now define how, given the edit script $\mathcal{E}$, we derive a mapping $g(\mathcal{E})$, called the mapping induced by $\mathcal{E}$, from the isomorphism f. We will see that the mapping g is an onto mapping from T_1 to T_2, and is thus isomorphic to an edge cover of the induced graph B.

Base case: If the edit script is empty, that is if $\mathcal{E} = ()$, then $g = f$.

[5] This restriction is necessary to disallow moving a subtree to a node in the same subtree, since the resulting structure would not be a tree.

Inductive case: The edit script is non-empty. Let d be the last edit operation in the edit script $\mathcal{E}$; that is, $\mathcal{E} = \mathcal{E}'.d$ for some edit script $\mathcal{E}'$. Let T_2' be the tree script obtained by applying $\mathcal{E}$ to T_1; that is, $T_1 \xrightarrow{\mathcal{E}} T_2$. Let g' be (inductively) the mapping induced by $\mathcal{E}'$; that is $g' = g(\mathcal{E}')$. We have the following cases, based on the last edit operation d. (Recall the formal definitions of the edit operations from Section 2.)

Case 1: d is an update operation. Then $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 2: d is an insert operation $\textsc{ins}(n, l, p, C)$. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup \{(\oplus, n)\}$.

Case 3: d is a delete operation $\textsc{del}(n)$. If $n \in T_1$, then $g(\mathcal{E}) = g(\mathcal{E}') \cup \{(n, \ominus)\}$, else $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 4: d is a move operation $\textsc{mov}(n_1, n_2)$. Then $g(\mathcal{E}) = g(\mathcal{E}')$.

Case 5: d is a copy operation $\textsc{cpy}(n_1, n_2)$. Let t_1 be the subtree rooted at n_1, and let t_1' be the subtree isomorphic to t_1 that is created as a result of this copy operation. Let h be the isomorphism between t_1 and t_1'. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup h$.

Case 6: d is a glue operation $\textsc{glu}(n_1, n_2)$. Let t_1 be the subtree rooted at n_1, and let t_2 be the subtree (isomorphic to t_1) rooted at n_2. (Recall that the subtree t_1 disappears as a result of this glue operation, being "united" with the subtree t_2.) Let h be the isomorphism between t_1 and t_2. Let $h' = (n, g(\mathcal{E}')(n)) \forall n \in t_1$. Then $g(\mathcal{E}) = g(\mathcal{E}') \cup h - h'$.

Finally, if the $\oplus$ node and the $\ominus$ node are both mapped to more than one node, we remove $[\oplus, \ominus]$ from the mapping. Now observe that after performing the operations indicated above for all the edit operations in $\mathcal{E}$, T_3 is transformed to a tree that is isomorphic to T_2 (by the definition of $\mathcal{E}$), so that the mapping $g(\mathcal{E})$ may be viewed as an onto mapping between T_1 and T_2. An onto mapping between the nodes of T_1 and T_2 is isomorphic to an edge cover of the bipartite graph induced by T_1 and T_2; thus $g(\mathcal{E})$ defines the edge cover induced by an edit script.

C Pseudo-code for the Algorithm Annotate

Recall our description of algorithm *Annotate* in Section 4.3. Figures 16 and 17 present the pseudo-code for Case 2.1 and Case2.2, respectively, of the algorithm.

D Proofs

Lemma 4.1 If $\mathcal{E}$ is an edit script satisfying Properties 1 and 2, then the edge cover $K(\mathcal{E})$ induced by $\mathcal{E}$ is a minimal edge cover.

Proof: (outline) We use an inductive argument that reflects the inductive definition of the edge cover induced by an edit script in Section 4.1. The edge cover induced by an empty edit script is minimal, since it is a one-to-one mapping between the trees. The only operations that cause more than one edge to become incident on a given node (other than the special nodes $\oplus$ and $\ominus$) are CPY and GLU . Given Property 2, a node n belonging to an intermediate tree T_3' produced by an edit script that is matched to more than one node in the original tree T_1 cannot be copied, thus avoiding the creation of a three-path. An analogous argument applies for the case of a node being glued.

Case 2.1: $M = \{m\},\ m \neq \oplus,\ |N| > 1$

Partition $N = N_1 \cup N_2 \cup N_3$ as follows:

$N_1 = \{n \in N : [p(m), p(n)] \in K\}$

$N_2 = \{n \in N : \exists m' \in T_1 \ni [m', p(n)] \in K\}$

$N_3 = \{n \in N : [\oplus, p(n)] \in K\}$

Partition $N_1 = \cup_{j=1}^{k_1} N_{1j}$ such that

$\forall j = 1 \ldots k_1,\ n_1, n_2 \in N_{1j} \Rightarrow p(n_1) = p(n_2)$

$\forall j = 1 \ldots k_1$, do:

Pick $n^* \in N_{1j}$

$A(m, n^*) \leftarrow$ NIL

$\forall n \in N_{1j}, n \neq n^*$, do:

$A(m, n) \leftarrow$ CPY

Partition $N_2 = \cup_{j=1}^{k_2} N_{2j}$ as follows:

$\forall j = 1 \ldots k_2,\ n_1, n_2 \in N_{2j} \Rightarrow \exists m' \in T_1 \ni$

$[m', p(n_1)], [m', p(n_2)] \in K$

Partition $N_3 = \cup_{j=1}^{k_3} N_{3j}$ as follows:

$\forall j = 1 \ldots k_3,\ n_1, n_2 \in N_{3j} \Rightarrow \exists m' \in T_1, l \in Z^+ \ni$

$\forall i = 1 \ldots l-1,\ [\oplus, p^i(n_1)], [\oplus', p^i(n_2)] \in K,\ \wedge$

$[m', p^l(n_1)], [m', p^l(n_2)] \in K$

Let $P_{pm} = \{n \in T_2 : [p(m), n] \in K\}$

$S = P_{pm} - \{p(n) : n \in N\}$

$C =$ the $|S|$ largest sets from $N_{ij},\ i = 1, 2,\ j = 1 \ldots k_i$

$\forall N_{ij}$, do:

Pick $n^* \in N_{ij}$

if $N_{ij} \in C$, then $A(m, n^*) \leftarrow$ MOV

else $A(m, n^*) \leftarrow$ CPY

$\forall n \in N_{ij}, n \neq n^*$, do:

$A(m, n) \leftarrow$ NIL

$D = D \cup \{[m, n^*, \{\text{MOV}, \text{CPY}\}] \rightarrow n\}$

Figure 16: Case 2.1 of the annotation algorithm

Case 2.2: $M = \{\oplus\} \wedge |N| > 1$
  Let $N_L = \{n' \in N : l(n') = l(n)\}$
  If $|N_L| = 1$ then
    $A(\oplus, n) \leftarrow$ INS
  else
    Partition $N_L = \cup_{j=1}^{k_L} N_{Lj} \ni$
      /* similar to N_3 in case 2.1 */
      $\forall j = 1 \ldots k_L,\ n_1, n_2 \in N_{Lj} \Rightarrow \exists l \in Z^+ \ni$
        $\forall i = 1 \ldots l,\ [\oplus, p^i(n_1)], [\oplus', p^i(n_2)] \in K,$
  $\forall N_{Lj}$, do:
    Pick $n^* \in NLj$
    $A(\oplus, n^*) \leftarrow$ INS
    $\forall n' \in N_{Lj}, n' \neq n^*$, do:
      $A(\oplus, n') \leftarrow$ NIL
      $D = D \cup \{[\oplus, n^*, \text{INS}] \rightarrow n'$

Figure 17: Case 2.2 of the annotation algorithm

For the case of edges incident on the $\oplus$ and $\ominus$ nodes, we note that the possibility of a three-path is avoided by the definition of the induced edge cover because it removes the edge $[\oplus, \ominus]$ whenever there are multiple edges incident on both $\oplus$ and $\ominus$. □

Lemma 4.2 If K is a minimal edge cover of a graph, K does not contain any path of length three.

Proof: Suppose n_1, n_2, n_3, n_4 is a path in K. (That is, $[n_i, n_{i+1}] \in K, i = 1 \ldots 3$.) Then $K - \{[n_2, n_3]\}$ is an edge cover contradicting the minimality of K. □

Lemma 5.1 If K is an annotated, minimal edge cover of the graph induced by two trees, then $c(K) = \sum_{e \in K} c_K(e)$.

Proof: By accounting. Recall that the cost $c(K)$ of an annotated edit script is the sum of the costs of the annotations in K (where the cost of each annotation is equal to the cost of the edit operation it represents). Each annotation in K is on some edge $e \in K$. If the annotation is an UPD , it is charged (by $c_K(e)$) to the edge e itself. For other annotations, each node of e is charged for half the cost of the annotation. Furthermore, the cost of each node is distributed evenly over all edges $e' \in K$ incident on its parent. Since the special edge between the (dummy) roots of the two trees being considered is never annotated (without loss of generality), it follows that the two methods of accounting for the cost of an annotated edge cover are equivalent. □

Lemma 5.2 Let $B = (U, V, E)$ be the bipartite graph induced by trees T_1 and T_2. Let $B' = (U, V, E')$, where $E' \subseteq E$. Let $\mathcal{K}$ denote the collection of all minimal edge covers of B'. We then have the following inequalities:

$$c_{lb}(e) \leq \min_{K \in \mathcal{K}} c_K(e) \quad \text{and} \quad c_{ub}(e) \geq \max_{K \in \mathcal{K}} c_K(e)$$

Proof: (outline) Given an edge $[m, n]$ in a minimal edge cover K, the upper bound cost function assumes the worst possible case. In particular, it assumes that, for each child m' of m, a cost of c_c and c_g, respectively, is incurred for all but one edges incident on m'; the remaining edge is assumed

to incur a cost c_m for a move.[6] The only exception is when there is an edge $[m', n']$ for some child n' of n; such an edge clearly does not involve a move, and therefore contributes 0 units to the cost. An analogous worst-case scenario is assumed for each child n' of n. Furthermore, the cost of $[m, n]$ is highest when $|E(m)| = |E(n)| = 1$, which is what the upper bound function assumes, resulting in the overall upper bound.

Similarly, the lower bound function assumes the best possible case for each child m' of m. In particular, it assumes that no cost is incurred on behalf of m' except in those cases where matching m to n would force a child m' to be moved; in such a case, a cost contribution of c_m is added. Furthermore, note that the cost of an edge $[m, n]$ is lower as $E(m)$ and $E(n)$ are bigger. However, since K is restricted to be a minimal edge cover, at least one of $E(m)$ and $E(n)$ must be a singleton set (containing just the edge $[m, n]$), or else there would be a path of length three in K, contradicting Lemma 4.2. Therefore, the cost of $[m, n]$ includes at least the lower of the two costs propagated from each of m, and n. Since this is precisely what the lower bound function defines c_{lb} to be, we see that the inequality for c_{lb} holds. □

Lemma 5.3 Let E_p be the set of edges pruned by repeated application of Pruning Rules 1 and 2. Let K_1 be any minimal edge cover of the graph B. There exists a minimal edge cover K_2 such that (1) $K_2 \cap E_p = \emptyset$, and (2) $C(K_2) \leq C(K_1)$.

Proof: The proof is by induction on the cardinality of E_p. When $|E_p| = 0$, the lemma is trivially true. Now assume that the lemma is true whenever $|E_p| \leq k$, for any $k \geq 0$. We will show that the lemma is also true when $|E_p| = k+1$. Each (successful) application of a pruning rule adds one edge to E_p. Consider the edge e_1 that was pruned last. Using the induction hypothesis for $E'_p = E_p - \{e_1\}$, we can generate an edge cover K'_1 such that (1) $K'_1 \cup E'_p = \emptyset$, and (2) $C(K'_1) \leq C(K_1)$.

If K'_1 does not contain e_1, let $K_2 = K'_1$. If K'_1 contains e_1, we modify K'_1 to obtain K_2 as follows. If e_1 was pruned using Pruning Rule 1, then let $K_2 = K'_1 - \{e_1\} \cup \{e_2, e_3\}$, where e_2 and e_3 are the edges used in the application of Pruning Rule 1. Else, e_1 was pruned using Pruning Rule 2; in this case, let $K_2 = K'_1 - \{e_1\} \cup \{[n_1, \ominus], [\oplus, n_2]\}$, where $e_1 = [n_1, n_2]$.

Clearly, $K_2 \cup E_p = \emptyset$. Since K'_1 is an edge cover of B, and since the only nodes that could be possibly exposed as a result of removing e_1 from K'_1 (namely, n_1 and n_2) are covered by the edges added to K'_1 to obtain K_2, it follows that K_2 is also an edge cover of B. From the definition of the pruning rules, and Lemma 5.2 we see that $C(K_2) \leq C(K'_1) \leq C(K_1)$. □

[6] Recall that we assume that CPY and GLU both cost more than a MOV .