Tree-to-tree Correction for Document Trees
Technical Report 95-372*

David T. Barnard†
Gwen Clarke
Nicholas Duncan

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada

David.Barnard@QueensU.CA

January 1995

Abstract

Documents can be represented as ordered labelled trees. Finding the editing distance between documents is a particular case of the general problem for trees. We give a detailed survey of previous results, presenting them in a single notation to elucidate their commonalities. We then discuss two ways of extending these results—first, by changing the set of primitive editing operations used by existing algorithms and, second, by post-processing the output of the algorithms to recognize patterns of change significant to documents. Finally, we provide extensions of the first type. Our algorithm allows subtree operations but is otherwise similar to that of Zhang and Shasha.

*This is a corrected and expanded version of Technical Report 91-315.

†This report was completed during a sabbatical at INRIA (Institute National de Recherche en Informatique et en Automatique) in Rocquencourt, France.

Contents

1 Introduction **3**

2 Background **5**

2.1 String-to-String Correction: Wagner and Fischer 5

2.2 Tree-to-Tree Correction: Selkow 5

2.3 Tree-to-Tree Correction: Tai 8

2.4 Tree-to-Tree Correction: Zhang and Shasha 10

3 Extending the Tree-to-Tree Correction Problem **14**

3.1 Subtree Insertion and Deletion 15

3.2 Swapping of Subtrees 20

3.3 Swapping and Editing of the Subtrees 21

3.4 Comparing the Three New Algorithms 22

4 Determining the Cost of Operations **22**

5 Conclusion **26**

A Programs for Object-to-Object Correction **29**

A.1 String-to-String Correction: Wagner and Fischer 29

A.2 Tree-to-Tree Correction: Selkow 31

A.3 Tai's Tree-to-Tree Algorithm 35

A.4 Zhang and Shasha's Tree-to-Tree Algorithm 38

A.5 Subtree Insertion and Deletion 41

A.6 Swapping Subtrees 43

1 Introduction

Documents can be represented not just as strings of characters in a file, but as structures with a hierarchical arrangement of text and non-text nodes. The non-text nodes are labelled with category names such as "paragraph", "section" and so on. Representing documents this way is a natural consequence of using the Standard Generalized Markup Language (SGML) to encode the content and form of documents [5]. Such representations can form the basis for text archives [2]. Even documents that are not simple hierarchies can be represented this way [3].

Finding the minimum difference between two of these documents is analogous to the string-to-string and tree-to-tree correction problems examined by other researchers [12, 19, 20, 21, 22, 23]. Such documents are trees with labelled nodes in which the left to right ordering of the offspring of a node is significant. Any piece of text in the document can be thought of as a single labelled node, all leaf nodes are text and any node with children is a structural or non-text node (see Figure 1).

An algorithm to find the minimum difference between two structured documents could be used in several different ways.

- *Editing or Co-authoring*: When an author presents a modified version of a document to an editor or co-author, the two versions could be compared to isolate only the changed components. A sophisticated display mechanism could highlight the differences. This would be a powerful editing tool.
- *Querying*: If documents are stored in an archive using a structured representation, a query against the archive could also be a tree. The document (or document fragments) that satisfy the query would be those that are closest to the query in edit distance.
- *Storing*: Documents are often published in several versions. A "document control system" might be modelled on a source code control system to provide help in managing versions. One of the characteristics of such a system should be that it stores only the differences between versions of a documents without having to store more than one complete document.

The general approach to edit distance problems (for strings or trees) has been to define a sequence of primitive operations that can be applied to one object to produce another, and to define the distance between two objects as a function computed on a sequence of such operations. In simple cases it can be sufficient to determine the length of the sequence. More realistically and more generally, each operation is assigned a *cost* that represents the difficulty of making that change to the object. The cost could be thought of as the perceived unlikelihood of the change having arisen at random in whatever process produced the changed object.

For string-to-string editing the operations most frequently considered are

- *insert* a character,
- *delete* a character, and
- *change* one character into another.

When considering trees this set of operations needs to be generalized to include others that deal with the structure of the tree. The primitive operations that can be used to measure tree-to-tree edit distance include:

- *insertTree*: Add an entire subtree (possibly one leaf node) to the target tree.
- *deleteTree*: Remove an entire subtree (possibly one leaf node) from the original tree.
- *insert*: Add an internal node to the target tree.

- *delete*: Remove an internal node from the given tree.
- *change*: Relabel a node in the given tree with the label of a node in the target tree.
- *swap*: Swap subtrees rooted at adjacent siblings.
- *swap with editing*: Swap subtrees rooted at adjacent siblings and edit the subtrees in the given tree to the corresponding subtrees in the target tree.

Assigning infinite cost to any operation will result in its elimination from the set of usable operations. For example, if $change(x, y)$ is assigned infinite cost for all nodes x in the original tree and all nodes y in the target tree, then a particular $change(x, y)$ operation will become more expensive than a $delete(x)$ followed by an $insert(y)$. The cost function need not return the same value for each operation of a particular type, because it can examine each individual operation and take into account properties of the nodes involved. For instance, the cost of deleting an internal node could depend on the number of children which must be re-attached to its parent.

Our algorithms extend previous work in the area, much of which is listed at the end of this paper. Some of the work referenced was not directly relevant because it focussed on structures other than the ones in which we are interested, like binary trees or undirected, acyclic graphs. Section 2 describes the relevant background to our work. Section 3 is a discussion of how this previous work might be extended to deal more realistically with trees that represent documents. We present our first extension of the Zhang and Shasha algorithm which allows insertion and deletion at the tree level. We also consider two ways of implementing a swap operation in the extended Zhang and Shasha algorithm. We consider the application of the algorithm with tree insertion, deletion, and swapping to structured documents. An appendix contains the text of programs that implement the algorithms (except for the last extension described).

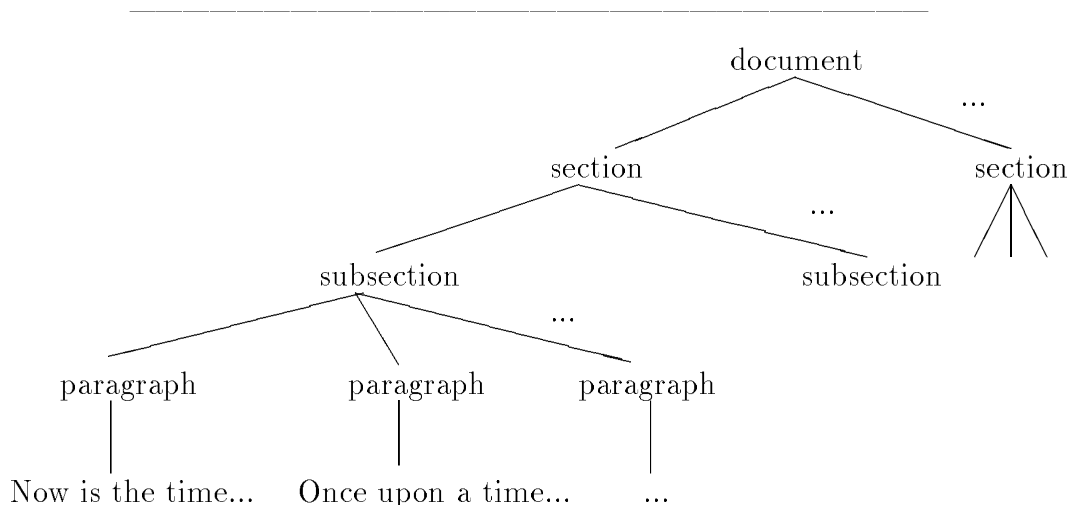

Figure 1: Representing Text as a Tree

2 Background

2.1 String-to-String Correction: Wagner and Fischer

The string-to-string correction problem is that of determining the distance between two strings as measured by the number of edit operations needed to transform one string into the other. There are actually two problems here: determining the minimum cost of a solution and producing a list of operations which, when performed on string S, would produce string S' with the minimum cost. The edit operations normally used are:

- *change* one character to another,
- *insert* a character, and
- *delete* a character.

This problem has been studied by several researchers; the algorithm of Wagner and Fischer [22] illustrates the approach.

Edit operations are written as $S_i \rightarrow S'_j$ where S and S' are the original and target strings, respectively, and S_i refers to the ith character of the string S. The symbol Λ represents the empty string, so $a \rightarrow \Lambda$ represents the deletion of character a and $\Lambda \rightarrow b$ represents the insertion of character b. A sequence of operations necessary to transform S into S' can be written as a collection of ordered pairs called a *trace*. Each pair (i, j) indicates that operation $S_i \rightarrow S'_j$ is one of the operations needed. If there is no pair with i in the first position then node S_i is to be deleted. Insertions are indicated in a similar manner. A trace is thus equivalent to a sequence of operations. Each edit operation has an associated cost function, γ, which assigns a non-negative real number, $\gamma(a \rightarrow b)$, to the transformation, $a \rightarrow b$.

The algorithm is presented in Figure 2. This is the first of several algorithms to be presented. These algorithms illustrate the various methods used to solve either the string-to-string or tree-to-tree correction problems. The notation has been altered from their original presentations in the literature in order to make clear what each algorithm shares with the others and for ease of reading. Each algorithm is accompanied by a brief description of the notational conventions used. The conventions introduced for one algorithm are considered to be in use in those presented later, unless a new convention is stated.

Given a pair of strings, S and S', to compare, with respective lengths n and m, the algorithm sets up an $(n+1) \times (m+1)$ matrix, D, where each entry, $D[i, j]$, corresponds to the distance between the substring of S composed of its first i characters and the substring of S' composed of its first j characters. It is easy to see that the mth entry in the 0th row would contain the cost of inserting all the characters in S' and the nth entry in the 0th column would contain the cost of deleting all the characters of S. The entry $D[n, m]$ is the minimum distance between the two entire strings. The cost in, time and space, of computing this result is $O(nm)$.

To solve the second problem, that is, to determine a sequence of edit operations which will accomplish the minimum cost conversion, it is only necessary to "follow the path of least resistance" through the distance array, recording the moves that are made. An $O(n+m)$ algorithm to do this is included in Figure 3. The algorithm produces a list of operations which is guaranteed to be a minimal cost solution (perhaps not the only one). Alternatively, we could extend the algorithm in Appendix 2 by keeping track of the operations used to form the least-cost edit sequence as the cost itself is calculated. This would require expanding each matrix entry to also record one operation.

2.2 Tree-to-Tree Correction: Selkow

Selkow extended the string-to-string problem solution to finding the tree-to-tree edit distance [12]. His solution used the basic operations *relabel*, *delete* and *insert*, but node deletions and insertions could only be

Notational conventions are used throughout:

Λ	the empty string or the empty tree
$\gamma(a \rightarrow b)$	cost of edit operation $a \rightarrow b$
n	length of the original string, S, or total number of nodes in the original tree, T
m	length of the target string, S', or total number of nodes in the target tree, T'
D	array of edit distances between string S and string S' or tree T and tree T'
$D[i,j]$	the cumulative edit distance between $S_{1..i}$ and $S'_{1..j}$ or between $T_{1..i}$ and $T'_{1..j}$
$\|\|$	list catenation operator

```
D[0,0] = 0
for i = 1 to n do
    D[i,0] = D[i-1,0] + γ(S_i → Λ)
for j = 1 to m do
    D[0,j] = D[0,j-1] + γ(Λ → S'_j)
for i = 1 to n do
    for j = 1 to m do
        D[i,j] = min(D[i-1,j-1] + γ(S_i → S'_j),
            D[i-1,j] + γ(S_i → Λ),
            D[i,j-1] + γ(Λ → S'_j))
```

Figure 2: The Wagner and Fischer Algorithm

```
L = nil
i = n
j = m
while i ≠ 0 and j ≠ 0 do
    if D[i,j] = D[i-1,j] + γ(S_i → Λ) then
        L = L || (S_i → Λ)
        i = i - 1
    else if D[i,j] = D[i,j-1] + γ(Λ → S'_j) then
        L = L || (Λ → S'_j)
        j = j - 1
    else
        L = L || (S_i → S'_j)
        i = i - 1
        j = j - 1
```

Figure 3: Extracting the Operations From Wagner and Fischer

Additional notational conventions:

c	the number of children from T
c'	the number of children from T'
T_i	in this algorithm only: the ith child (counting from the left) of T, not the ith node of the tree in some overall labelling
$\lambda(T)$	the label of the node at the root of T
$ins(\Lambda \rightarrow T'_x)$	the cost to insert the subtree rooted at x into T'
$del(T_x \rightarrow \Lambda)$	the cost to delete the subtree rooted at x within T
CostType	some predefined type (probably integer)

```
edit(T, T'): CostType
   D[0,0] = γ(λ(T) → λ(T'))
   for i = 1 to c do
      D[i,0] = D[i−1,0] + del(T_i → Λ)
   for j = 1 to c' do
      D[0,j] = D[0,j−1] + ins(Λ → T'_j)
   for i = 1 to c do
      for j = 1 to c' do
         D[i,j] = min(D[i−1,j−1] + edit(T_i, T'_j), D[i−1,j] + del(T_i → Λ), D[i,j−1] + ins(Λ → T'_j))
   return(D[c,c'])
```

Figure 4: Selkow's Tree-to-Tree Algorithm

done at the leaves of the trees. Only if the entire subtree rooted at a node was deleted could the node be deleted and, similarly, only nodes without descendants (i.e. , leaf nodes) can be inserted. Given an input tree, A, and a target tree, B, Selkow's algorithm takes the cost of changing the root of A to that of B and adds to this the cost of changing the subtrees of A, $A_1, A_2, \ldots, A_n$, to the subtrees of B, $B_1, B_2, \ldots, B_m$.

Selkow's approach is recursive. The first invocation of the algorithm is given the root nodes of two trees, the original and the target trees ; it then invokes the algorithm on each child of one root paired with each child of the other root. At each pair of nodes, the algorithm sets up a temporary $n \times m$ cost matrix, where n is the number of children of the node from the original tree and m is the number of children of the node from the target tree. Entry (i, j) in the matrix is then computed by taking the minimum cost from three approaches available to edit $A_1, A_2, \ldots, A_i$ into $B_1, B_2, \ldots, B_j$.

- The first approach is to add the cost to edit $A_1, A_2, \ldots, A_i$ into $B_1, B_2, \ldots, B_{j-1}$ and the cost to insert B_j.
- The second method is to add the cost to edit $A_1, A_2, \ldots, A_{i-1}$ into $B_1, B_2, \ldots, B_j$ and the cost to delete A_i.
- The last of these costs is calculated by taking the cost to edit $A_1, A_2, \ldots, A_{i-1}$ into $B_1, B_2, \ldots, B_{j-1}$ and recursively examining the cost of editing A_i into B_j.

Selkow's algorithm is given in Figure 4.

A problem with the recursive approach is that the distance array is a local, temporary data structure. The intermediate costs necessary for determining the actual list of operations are thus gone once the algorithm

Notational conventions are:

CostType	a pair with a field c, for cost, and another l, for list of operations.
D	an array of size $(n+1) \times (m+1)$ of type CostType.

$\mathbf{edit}(T, T')$: CostType
  $D[0,0] = (\gamma(\lambda(T) \rightarrow \lambda(T')),\ (\lambda(T) \rightarrow \lambda(T')))$
  for $i = 1$ to c do
    $D[i,0] = (D[i-1,0].c + del(T_i \rightarrow \Lambda),\ D[i-1,0].l \parallel T_i \rightarrow \Lambda)$
  for $j = 1$ to c' do
    $D[0,j] = (D[0,j-1].c + ins(\Lambda \rightarrow T'_j),\ D[0,j-1].l \parallel \Lambda \rightarrow T'_j)$
  for $i = 1$ to c do
    for $j = 1$ to c' do
      $change = \mathbf{edit}(T_i, T'_j)$
      $delete = D[i-1,j].c + del(T_i \rightarrow \Lambda)$
      $insert = D[i,j-1].c + ins(\Lambda \rightarrow T'_j)$
      if $D[i-1,j-1].c + change.c \leq delete$ and $(D[i-1,j-1].c + change.c \leq insert)$ then
        $D[i,j] = (D[i-1,j-1].c + change.c,\ D[i-1,j-1].l \parallel change.l)$
      else if $delete \leq insert$ then
        $D[i,j] = (delete,\ D[i-1,j].l \parallel (T_i \rightarrow \Lambda))$
      else
        $D[i,j] = (insert,\ D[i,j-1].l \parallel (\Lambda \rightarrow T'_j))$
  return$(D[c,c'])$

Figure 5: Creating an Operation List with Selkow's Algorithm

terminates. We cannot, therefore, use an algorithm like that presented as an extension of the string-to-string edit algorithm, to determine the list of operations making up the minimal-cost edit. However, it is not difficult to keep track of the operation chosen as having lowest cost at each stage by returning a richer data structure than the one used in the present algorithm. Figure 5 shows how to do this; it is the analog of the alternate approach suggested under the discussion of the string-to-string problem.

The time complexity of Selkow's algorithm is $O(nmd)$, where n and m are the maximum number of children from any node in each of the trees and d is the maximum depth of the trees.

Restricting insertions and deletions to leaf nodes makes the algorithm very simple, but does not accurately represent our intuition about which set of primitive operations is reasonable for tree editing or, especially, document editing. For example, the cost of merging two internal nodes would be the cost of deleting every descendant of one of them, plus the cost of inserting all the same nodes under the other, plus the cost of changing one internal node. It would be more realistic in this case to delete and insert only the nodes, not the subtrees of which they are the roots. Nevertheless, deletion and insertion of entire subtrees are useful operations and we will restore them into our set of primitive operations once we have examined some other tree-to-tree editing distance solutions.

2.3 Tree-to-Tree Correction: Tai

Tai [20] solves the tree-to-tree correction problem by a method which is, again, very similar to the methods used for the string problem. However, unlike Selkow's algorithm, Tai's uses a dynamic programming

approach without recursion. The primitive operations used are

- *change* one node label into another,
- *delete* a node, and *insert* a node.

The insertions attach some or all of the children of the parent of the node being inserted as the children of the node being inserted. Similarly, all of the children of a deleted node are re-attached to the parent of that node. The exact method used to attach only some of the children to the new node in an insertion is not important in determining the minimum edit distance as long as there is some accurate way of determining the cost of insertions and as long as we are assured that only consecutive subsequences of children are attached.

In adapting the string algorithm to trees, Tai used a preorder traversal to name each node uniquely. A preorder traversal is one in which the first node visited is the root, then all subtrees are traversed in preorder from left to right. Tai assumes that the root of the tree remains unchanged during editing.

During the traversal each node is numbered, so that node 1 is always the root of the tree and node i is the node that was visited after node $i-1$. $T(x:y)$ denotes the nodes and their associated edges between nodes $T_x, T_{x+1}, \ldots, T_y$ and T_x is an ancestor of T_y. Tai then uses the idea of traces from the string-to-string algorithm to catalog which operations are used to transform T to T'. They are called *mappings* here but they are similar to traces in the Wagner and Fischer algorithm [22]. The pairs in a mapping may imply a change operation on node T_i to produce node T'_j if they are labelled differently, or a simple connection between the nodes if they have the same label. If, for any node $a \in T$, there is no pair in a mapping with the first integer representing a, then this is equivalent to deleting a. If no pair exists with a second integer representing $b \in T'$ in a mapping, then b is considered to be inserted into T'.

The biggest problem with this approach is that, unlike the situation with strings, preserving structural information about the tree is necessary. This means that while in the string case one always knew that if $S_{i..i+h} = S'_{j..j+h}$ then the distance between $S_{1..i-1}$ and $S'_{1..j-1}$ is the same as the distance between $S_{1..i+h}$ and $S'_{1..j+h}$, the analogous relationship in trees does not hold. See Figure 6 for a simple case where the least-cost mapping $((1,1),(2,2))$ from $T(1:2)$ to $T'(1:3)$ cannot be extended to $((1,1),(2,2),(3,4))$ as a mapping from $T'(1:4)$; this is because T_2 is an ancestor of T_3 but T'_2 is not an ancestor of T'_4. The least-cost mapping in this case is $((1,1),(2,3),(3,4))$ assuming that the cost of deleting, inserting or changing a character is the same. Because of this problem, conditions are imposed on mappings to ensure that the structure of both trees is preserved by the mapping.

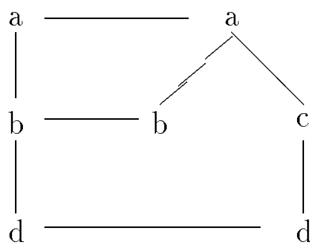

Figure 6: Example of an Illegal Mapping

If we examine the mechanics of Tai's algorithm we can see that, as in the string-to-string algorithm, a distance matrix, D, is used and for each location in the matrix, $D[i,j]$, the cost of deleting T_i, the cost of

Notational conventions are:

$p(x)$	the immediate parent of x
$p^n(x)$	the immediate parent of $p^{n-1}(x)$
$p^0(x)$	x
T_i	in this algorithm only: the ith node of T, where the tree is labelled in preorder
INF	the maximum possible edit cost (i.e., deleting T then inserting T')
E	an array for keeping track of the minimum cost of change paths.
CC	an array for keeping track of the minimum cost of changing node T_i into node T_j (including accumulated costs from all of their descendants and left siblings).

```
compute minimum cost of all possible change paths
for i = 1 to n do
  for j = 1 to m do
    for u = i, p(i), p^2(i), ..., 1 do
      for s = u, p(u), p^2(u), ..., 1 do
        for v = j, p(j), p^2(j), ..., 1 do
          for t = v, p(v), p^2(v), ..., 1 do
            if s = u = i and t = v = j then
              E[s, u, i, t, v, j] = γ(T_i → T'_j)
            else if s = u = i or t < v = j then
              E[s, u, i, t, v, j] = E[s, u, i, t, p(j), j − 1] + γ(Λ → T'_j)
            else if s < u = i or t = v = j then
              E[s, u, i, t, v, j] = E[s, p(i), i − 1, t, v, j] + γ(T_i → Λ)
            else E[s, u, i, t, v, j] = min(E[s, x, i, t, v, j], E[s, u, i, t, y, j],
                E[s, u, x − 1, t, v, y − 1] + E[x, x, i, y, y, j]);
Where if u = p^n(i) then x = p^{n−1}(i) in T and if v = p^n(j) then y = p^{n−1}(j) in T'
```

Figure 7: Tai's Tree-to-Tree Algorithm: Step 1

inserting T'_j, and the cost of mapping T_i to T'_j, are assessed and the minimum selected. Also as in the string case, a list of operations necessary to ensure a minimum-cost transformation of T into T' is obtainable after the algorithm is terminated. Since the structure of the trees must be preserved, there is some extra work (compared to the string problem) required to ensure that a legal minimum cost mapping is being achieved. Steps 1 and 2 of the algorithm are needed to ensure that no illegal mappings are formed. The first step forms a six-dimensional matrix, E, which records the cost of transforming each subtree of T into each subtree of T'. The minimum cost is selected in the CC array in the second step. Tai's algorithm has time and space complexity in $O(nmd^2d'^2)$, where d is the depth of the tree T and d' is the depth of T'. Tai's algorithm is given in Figures 7, 8 and 9. A similar algorithm using a postorder traversal of the tree and having a better time complexity, and much better space complexity, is given in [23]. The next section discusses that algorithm.

2.4 Tree-to-Tree Correction: Zhang and Shasha

The algorithm due to Zhang and Shasha [19, 23] uses the same basic definitions and assumptions as that of Tai, including the same three primitive operations. They number the trees to be compared in a different way, however, using a postorder traversal instead of preorder. A postorder traversal visits the nodes of a

```
compute minimum change cost between each pair of nodes
CC[1,1] = 0
for i = 2 to n do
   CC[i, 1] = i
for j = 2 to m do
   CC[1, j] = j
for i = 2 to n do
   for j = 2 to m do
      CC[i, j] = INF
      for s = i, p(i), p^2(i), ..., 1 do
         for t = j, p(j), p^2(j), ..., 1 do
            CC[i, j] = min(CC[i, j], CC[s, t] + E[s, p(i), i − 1, t, p(j), j − 1] - γ(T_s → T'_t)
      CC[i, j] = CC[i, j] + γ(T_i → T'_j)
```

Figure 8: Tai's Tree-to-Tree Algorithm : Step 2

```
compute tree-to-tree distance
D[1, 1] = 0
for i = 2 to n do
   D[i, 1] = D[i − 1, 1] + γ(T_i → Λ)
for j = 2 to m do
   D[1, j] = D[1, j − 1] + γ(Λ → T'_j)
for i = 2 to n do
   for j = 2 to m do
      D[i, j] = min(CC[i, j], D[i − 1, j] + γ(T_i → Λ), D[i, j − 1] + γ(Λ → T'_j))
```

Figure 9: Tai's Tree-to-Tree Algorithm : Step 3

tree starting with the leftmost leaf descendant of the root and proceeding to the leftmost descendant of the right sibling of that leaf, the right sibling(s), then the parent of the leaf and so on up the tree to the root. The last node visited will always be the root.

With Tai's algorithm it was necessary, in effect, to keep track of multiple solutions in order to backtrack around illegal ones once the lower levels were reached. With Zhang and Shasha's algorithm, the minimum cost mappings of all the descendants of each node have been computed before the node is encountered, so the least-cost mapping can be selected right away. This is accomplished by keeping track of the *keyroots* of a tree. Keyroots are defined to be the root of the tree plus all nodes which have a left sibling. Determining the keyroots of a tree in advance allows the algorithm to separate the concepts of tree distance and forest distance. The distance between two nodes when considered in the context of their left siblings in the trees T and T' is their *forest distance*. The distance between two nodes considered separately from their siblings and ancestors (but not from their descendants) is their *tree distance*. For example, in Figure 10, we can see that the forest distance between node b in T and node b in T' is 6 units while the tree distance is 3 units (if each change operation costs 1 unit). On the other hand, the forest distance and tree distance between nodes a and q are both 3 units, because neither has any left siblings.

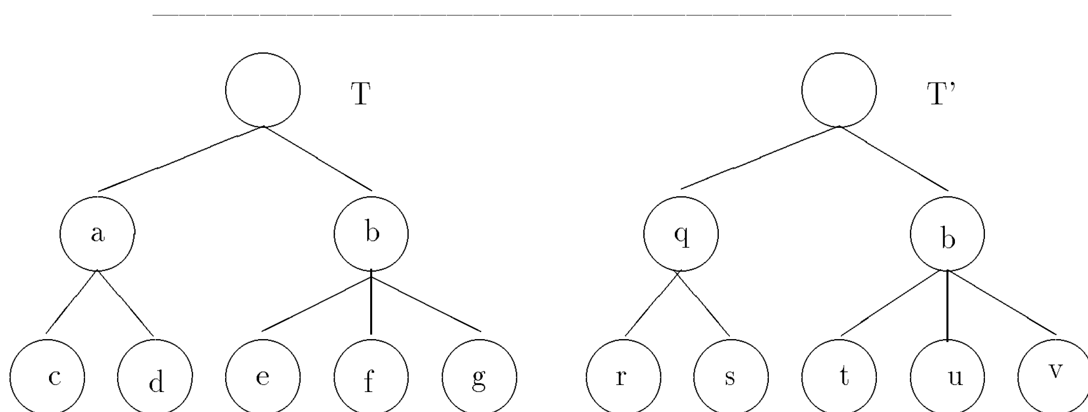

Figure 10: Tree distance vs. Forest distance

For each node, the calculation to determine the minimum cost mapping from the node to a node in the other tree (the tree distance) depends only on mapping the nodes and their children. To determine the minimum cost mapping of a node, then, one must know the minimum cost mapping from all the keyroots amongst its children, plus the cost of its leftmost child (the forest distance of its rightmost child). The algorithm thus proceeds through the nodes determining mappings from all leaf keyroots first, then all keyroots at the next higher level, and so on to the root. Since the nodes are numbered in postorder, this is a natural way to proceed. The algorithm employs a temporary forest distance array for each pair of keyroots. If a node is not a keyroot, one could determine a separate forest distance for it, but it would be exactly equal to its tree distance since there are no left siblings, so this can be determined when the parent is determined. See Figure 11 for the algorithm. Extracting the set of operations necessary to transform T into T' at a cost no greater than $D[n, m]$ can be done with techniques similar to those presented above.

Zhang and Sasha's solution to the problem has a time complexity in O($nmdd'$) and a space complexity, depending on how the temporary arrays are implemented, in O(nm). An example which illustrates this algorithm and compares it to our solution is given below. This work has been extended by the inclusion of more general pattern matching, and the development of programs that implement the algorithms [13, 14, 15, 16, 17, 18].

Notational conventions are:

T_i	in this algorithm only: the ith node of T, labelled in post order
$l(i)$	the leftmost leaf descendant of the subtree rooted at i
$K(T)$	the keyroots of tree T, $K(T) = \{k \in T \mid \neg\exists k' > k \text{ with } l(k') = l(k)\}$
$\emptyset$	a null tree
$FD[T[i,i'],T'[j,j']]$	the forest distance from nodes i to i' in T to nodes j to j' in T', if $i \geq i'$ then $T[i,i'] = \emptyset$

```
compute l( ), K(T), and K(T')
for each x ∈ K(T) do
  for each y ∈ K(T') do
    FD[∅,∅] = 0
    for i = l(x) to x do
      FD[T[l(x),i],∅] = FD[T[l(x),i−1],∅] + γ(T_i → Λ)
    for j = l(y) to y do
      FD[∅,T'[l(y),j]] = FD[∅,T'[l(y),j−1]] + γ(Λ → T'_j)
    for i = l(x) to x do
      for j = l(y) to y do
        m = min(FD[T[l(x),i−1],T'[l(y),j]] + γ(T_i → Λ),
                FD[T[l(x),i],T'[l(y),j−1]] + γ(Λ → T'_j))
        if l(i) = l(x) and l(j) = l(y) then
          D[i,j] = min(m, FD[T[l(x),i−1],T'[l(y),j−1]] + γ(T_i → T'_j))
          FD[T[l(x),i],T'[l(y),j]] = D[i,j] /* Note */
        else
          FD[T[l(x),i],T'[l(y),j]] = min(m, FD[T[l(x),l(i)−1],T'[l(y),l(j)−1]] + D[i,j])
```

Note: This required assignment was omitted from the version of the algorithm given in [23].

Figure 11: Zhang and Shasha's Tree-to-Tree Algorithm

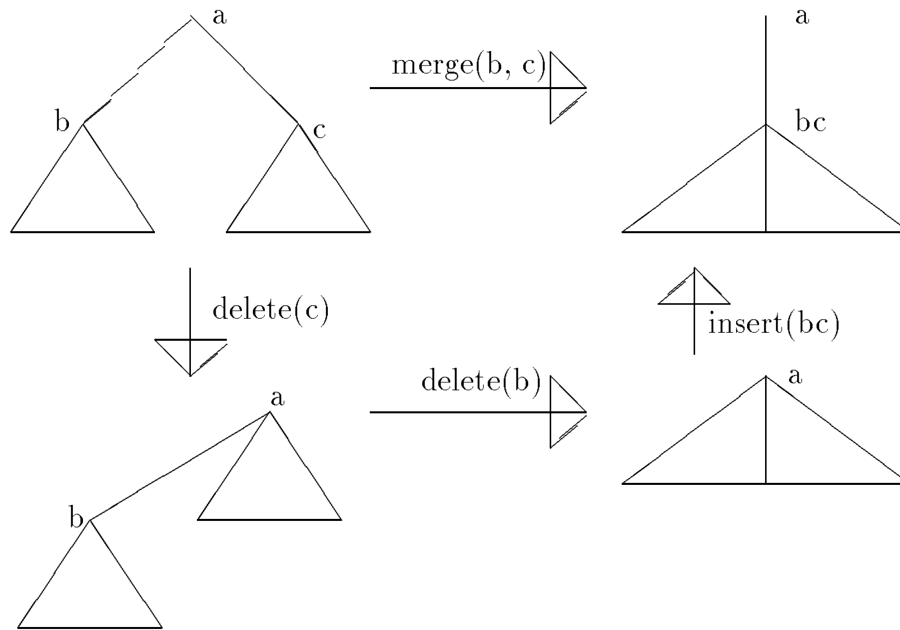

Figure 12: A Merge Operation

3 Extending the Tree-to-Tree Correction Problem

The solutions to the tree-to-tree edit problem presented above lack an intuitive applicability to document editing. When editing documents most people perform operations that are more complex than the primitive operations we have seen in the algorithms of the previous section. Many operations that seem primitive when editing documents are, in fact, combinations of these operations. Examples of operations which are commonly performed when editing documents are:

- merge two sections,
- split one section into two,
- permute sections of text under an unchanging encompassing structure, and
- move one section of text to another position.

One approach is to consider these operations as primitive and look for them in the same way as the other primitive operations already identified. This seems impractical, however, because it would make the inner loop of the computation much more complex.

A simpler way to account for these kinds of operations would be to use a post-processing approach, where the set of primitive operations is not extended but subsequent processing tries to identify operations that are combinations of one or more primitives. Two deletes followed by an insert could be a merge, for example. See Figure 12 for an illustration of a merge. The complex operations should be intuitively more acceptable in the context of editing documents than the primitive operations.

Notational conventions and definitions:

$l(i)$	the leftmost leaf descendant of the subtree rooted at i.
K(T)	$\{k \mid \neg\exists k' > k$ with $l(k') = l(k)\}$
deleteTree(a)	the cost of performing a deleteTree operation on node a.
insertTree(a)	the cost of performing an insertTree operation on node a.

```
for each x ∈ K(T) do
   for each y ∈ K(T') do
      FD[l(x) − 1, l(y) − 1] = 0
      for i = l(x) to x do
         FD[i, l(y) − 1] = FD[l(i) − 1, l(y) − 1] + deleteTree(T_i)
      for j = l(y) to y do
         FD[l(x) − 1, j] = FD[l(x) − 1, l(j) − 1] + insertTree(T'_j)
      for i = l(x) to x do
         for j = l(y) to y do
            m = min(FD[i − 1, j] + γ(T_i → Λ),
                  FD[i, j − 1] + γ(Λ → T'_j),
                  FD[l(i) − 1, j] + deleteTree(T_i),
                  FD[i, l(j) − 1] + insertTree(T'_j))
            if l(i) = l(x) and l(j) = l(y) then
               D[i, j] = min(m, FD[i − 1, j − 1] + γ(T_i → T'_j))
               FD[i, j] = D[i, j]
            else
               FD[i, j] = min(m, FD[l(i) − 1, l(j) − 1] + D[i, j])
```

Figure 13: Extending with Insertion and Deletion of Subtrees

Neither the set of primitives used by Zhang and Shasha nor that used by Selkow, however, seem rich enough to allow this kind of postprocessing. The merge example, given above, relies on delete and insert operations which are restricted to single nodes, but if we still wish to find any permute operations that apparently have occurred, we must have delete and insert operations on a subtree level. We propose to add the two primitive operations deleteTree and insertTree, which are defined to be the same as Selkow's delete and insert, to the set of primitive operations given by Zhang and Shasha. The base algorithm for dealing with *deleteTree* and *insertTree* is given below. Following that, we propose to add the primitive operation *swap* which in the context of document editing would seem more natural than the deletion of a subtree followed by the insertion of that same subtree or a very similar one.

3.1 Subtree Insertion and Deletion

This section will present the extensions to the Zhang and Shasha algorithm which are necessary to add the two primitive operations *deleteTree* and *insertTree*. An example of the operation of both algorithms on the same tree is then given. The revised algorithm is presented in Figure 13.

As can be seen, our algorithm is very similar to that of Figure 11. But there are significant differences caused by the addition of *deleteTree* and *insertTree*. Lemma 3 in the Zhang and Shasha paper includes

these two statements:

$$\text{forestdist}(T_1[l(i_1)..i], \emptyset) = \text{forestdist}(T_1[l(i_1)..i-1], \emptyset) + \gamma(T_1[i] \rightarrow \Lambda)$$

$$\text{forestdist}(\emptyset, T_2[l(j_1)..j]) = \text{forestdist}(\emptyset, T_2[l(j_1)..j-1]) + \gamma(\Lambda \rightarrow T_2[j])$$

With our additions of deleteTree and insertTree, these are changed to:

$$\text{forestdist}(T_1[l(i_1)..i], \emptyset) = \text{forestdist}(T_1[l(i_1)..l(i)-1], \emptyset) + \text{deleteTree}(T_1[i])$$

$$\text{forestdist}(\emptyset, T_2[l(j_1)..j]) = \text{forestdist}(\emptyset, T_2[l(j_1)..l(j)-1]) + \text{insertTree}(T_2[j])$$

The calculation of the distance corresponds to the cost of deleting or inserting the subtree rooted at i or j, respectively, and repeating the process with all subtrees rooted at the siblings. This change in lemma 3 causes the change in the initialization phase of the algorithm.

When D is being initialized, the basic operations can no longer be *delete* and *insert*, since those operations can no longer be performed on leaf nodes. The maximum possible cost of the entire transformation is therefore the cost of performing a *deleteTree* operation on the root of T, then performing an *insertTree* operation on the root of T'. For each pair of keyroots, the forest distance array (FD) is now initialized by calculating the cost of deleting and inserting each subtree on a cumulative basis, rather than deleting and inserting each node.

With the new primitive operations, we must add two new cases.

$$FD[l(i)-1, j] + \text{deleteTree}(T_i)$$

$$FD[i, l(j)-1] + \text{insertTree}(T'_j)$$

At any pair of nodes, T_i and T'_j, the minimum cost may now be that yielded by one of five operations plus the appropriate accumulated cost. The costs are the same as in the previous algorithm for the three shared operations, but the accumulated cost of the two new subtree operations depends on the cost up to the root of the previous sibling subtree (if any). The root of the previous subtree is $T_{l(i)-1}$ for T_i and $T'_{l(j)-1}$ for T'_j since the tree is numbered in postorder.

Consider any pair (i, j) where $i \in K(T)$ and $j \in K(T')$, then:

1. Just before the calculation $D(i, j)$, all distances $D(i_1, j_1)$ are available such that i_1 is not on the path from i to $\mathrm{l}(i)$ but is in the subtree of tree(i) and j is not on the path from j to $\mathrm{l}(j)$ but is in the subtree of tree(j).

2. After the calculation of $D(i, j)$ all values $D(i_1, j_1)$ are available.

Condition 1 always holds: Consider i_1 where $\mathrm{l}(i_1) \neq \mathrm{l}(i)$. Choose the lowest ancestor of i_1, i'_1 such that $i'_1 \in K(T)$. Because a node can be its own ancestor and because of the properties of a keyroot, we know that $\mathrm{l}(i_1) = \mathrm{l}(i'_1)$, so $i'_1 \neq i$. Both i'_1 and $i \in K(T)$ so $i'_1 \leq i$, therefore $i_1 < i$. The same holds for j_1. Therefore because the keyroots are stored in ascending order in $K(T)$ and $K(T')$, $D(i'_1, j'_1)$ will always be computed before $D(i, j)$ and $D(i_1, j_1)$ is available after calculating $D(i'_1, j'_1)$.

Based on the truth of the first condition and the if condition in the algorithm, ($\mathrm{l}(i) = \mathrm{l}(x)$ and $\mathrm{l}(j) = \mathrm{l}(y)$), condition 2 holds.

The complexity is the same as that of the Zhang and Shasha algorithm, since all of the changes take place inside the loops already there. The time complexity is $\mathrm{O}(nmdd')$ and the space complexity is $\mathrm{O}(nm)$.

We will now give an example using two tree comparison algorithms to find the minimum edit distance between two trees. First we will examine the operation of the Zhang and Shasha algorithm. We will then compare this to our extended algorithm.

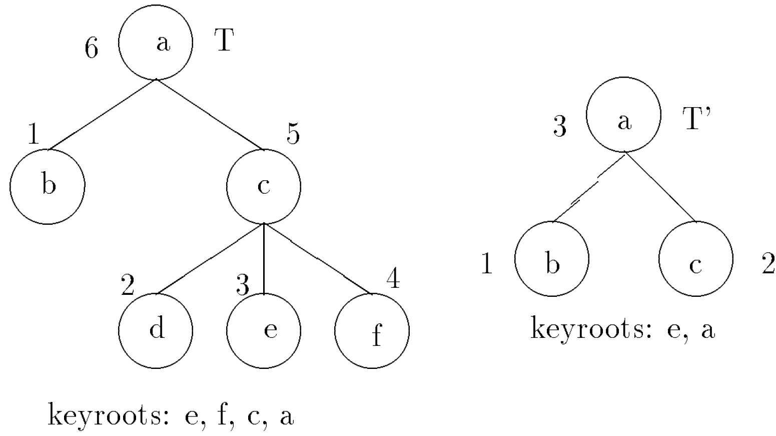

Figure 14: Convert T into T'

The two trees to be compared are shown in Figure 14. Note that the letters inside the circles representing the nodes are to be taken as the labels of the nodes and that the numbers beside the nodes are their postorder numbering. In the case of structured documents the node labels would be longer strings, which would add some complexity to the explanation without altering the functioning of the algorithm. We will refer to nodes by either their labels or their postorder numbering when discussing the algorithms, but the algorithms themselves use only the postorder numbering. Also given in the diagram is the set of keyroots of each tree.

The first assumption we must make in order to demonstrate the algorithms has to do with the relative costs of the various operations. For this example, we will assume that all three of the operations allowed by the Zhang and Shasha algorithm have equal cost for all nodes. Arbitrarily we will set that cost to 1 unit. The purpose in setting the costs in such a simple way is so that the interaction of the various costs will not obscure the functioning of the algorithm. Both algorithms will function with any costs assigned to their operations, including costs which vary depending on which node is being considered.

D Array

	1	2	3
1			
2			
3		0	
4			
5			
6			

FD Array

	$\emptyset$	$T'[2,2]$
$\emptyset$	0	1
$T[3,3]$	1	0

Figure 15: Converting node e to node e

In Figure 15 we see the first iteration of the algorithm, determining the cost of transforming the tree rooted in node e of tree T to the tree rooted in node e of tree T'. Since these two nodes have the same label, the cost is that of converting label e to label e, which is to say 0. Once the FD cost is determined, the D array is updated, since node e is on the path from node e to node e in both trees.

D Array

	1	2	3
1			
2			
3	1	0	2
4			
5			
6			

FD Array

	$\emptyset$	$T'[1,1]$	$T'[1,2]$	$T'[1,3]$
$\emptyset$	0	1	2	3
$T[3,3]$	1	1	1	2

Figure 16: Converting node e to node a

Comparing the subtree rooted at e in tree T to the subtree rooted in a in tree T' (which is actually the entire tree T') is the next stage of the algorithm. This is slightly more complex and results in two slots of D being filled. Since the cost of converting node e to node b is 1, that is the number put in the respective slot of D. D[3,3] gets a cost of 2 since we could obtain T' from node e by inserting nodes b and a.

D Array

	1	2	3
1			
2			
3	1	0	2
4	1	1	3
5			
6			

FD Array

	$\emptyset$	$T'[2,2]$
$\emptyset$	0	1
$T[4,4]$	1	1

FD Array

	$\emptyset$	$T'[1,1]$	$T'[1,2]$	$T'[1,3]$
$\emptyset$	0	1	2	3
$T[4,4]$	1	1	2	3

Figure 17: Converting node f to nodes e and a

In Figure 17, we see the costs of converting the leaf f to the leaf e and then to the entire tree T'. The greater cost of creating T' from f than from e is because f must be changed into another node label whereas e can be left the same as it is in T'.

The next two iterations compare the subtree rooted at c with the two keyroots of T'. Note that the leaf d is compared also because it is on the path from c to $l(c) = d$. It costs 3 units to convert the subtree rooted at c into T' because we must change b to d, e to e, c to a and delete f. In this case there is only one way to obtain the lowest cost transformation, but this does not, in general, have to be true.

Finally, in Figure 19, we see the cost of converting all of T into T'. Again, any nodes which are on the path to the leftmost leaf descendant of the root (in this case, b) are transfered from FD at the same time as their ancestor. The final cost is the same as it was when converting node c to node e (D[5,2]), since we can

D Array

	1	2	3
1			
2	1	1	3
3	1	0	2
4	1	1	3
5	4	3	3
6			

FD Array

	$\emptyset$	$T'[2,2]$
$\emptyset$	0	1
$T[2,2]$	1	1
$T[2,3]$	2	1
$T[2,4]$	3	2
$T[2,5]$	4	3

FD Array

	$\emptyset$	$T'[1,1]$	$T'[1,2]$	$T'[1,3]$
$\emptyset$	0	1	2	3
$T[2,2]$	1	1	2	3
$T[2,3]$	2	2	1	2
$T[2,4]$	3	3	2	3
$T[2,5]$	4	4	3	3

Figure 18: Converting node c to nodes e and a

D Array

	1	2	3
1	0	1	2
2	1	1	3
3	1	0	2
4	1	1	3
5	4	3	3
6	5	5	3

FD Array

	$\emptyset$	$T'[2,2]$
$\emptyset$	0	1
$T[1,1]$	1	1
$T[1,2]$	2	2
$T[1,3]$	3	2
$T[1,4]$	4	3
$T[1,5]$	5	4
$T[1,6]$	6	5

FD Array

	$\emptyset$	$T'[1,1]$	$T'[1,2]$	$T'[1,3]$
$\emptyset$	0	1	2	3
$T[1,1]$	1	0	1	2
$T[1,2]$	2	1	1	2
$T[1,3]$	3	2	1	2
$T[1,4]$	4	3	2	3
$T[1,5]$	5	4	3	4
$T[1,6]$	6	5	4	3

Figure 19: Converting node a to nodes e and a

simply convert a to a, b to b, delete c and d, convert e to e and delete f. Using the three operations allowed in this algorithm, this is the best possible conversion, given the relative operation cost assumption above.

Looking now at the new algorithm, we must assign costs to two additional operations. The simplest assignment is to again give each of the five operations a cost of 1 unit. This would result in a strong bias in favor of the two operations deleteTree and insertTree, however, and does not seem to be intuitively correct. We have therefore assigned a cost of k, where k is the number of nodes in the subtree being inserted, to the operation insertTree. The cost of deleteTree we will leave at 1, regardless of the size of the subtree being deleted.

The costs of converting the leaf nodes to other leaf nodes is the same as in the previous algorithm, so Figures 20 to 22 are very similar to Figures 15 to 17. This need not be the case, since the operations now being performed at the leaf level are deleteTree, insertTree and change instead of insert, delete and change. Since in this example the cost of deleting a leaf or inserting a leaf is the same regardless of which set of operations is used, the cumulative costs are also the same. The effects of adding deleteTree to our set of primitive operations on subtrees become quite clear in the next few iterations, however.

In Figure 23, we see that since deleting the entire subtree rooted at c is now possible, it is cheaper to do that and insert node e into T' than to delete the three nodes necessary to move e up to become a child of a. The cost in slot D[5,2] reflects that fact.

D Array

	1	2	3
1			
2			
3		0	
4			
5			
6			

FD Array

	$l(2)-1$	2
$l(3)-1$	0	1
3	1	0

Figure 20: Converting node e to node e: our algorithm

D Array

	1	2	3
1			
2			
3	1	0	2
4			
5			
6			

FD Array

	$l(3)-1$	1	2	3
$l(3)-1$	0	1	2	3
3	1	1	1	2

Figure 21: Converting node e to node a: our algorithm

As before, adding the two nodes a and b to the comparison does not change the cost since equivalent nodes exist in both trees. So the minimum cost of changing T into T' is 2.

The two algorithms presented are quite similar, with the only significant difference being the addition of the two new operations. This example shows that the addition of deleteTree to the set of primitive operations can result in significant differences in the minimum cost set of operations discovered. The operation insertTree has little effect on the example, but if the cost of operations were to be changed, then insertTree could also result in different minimum cost sets being discovered.

3.2 Swapping of Subtrees

This section first presents the extensions to the algorithm introduced in subsection 4.1 which are necessary to add the primitive operation *swap*, where the subtrees to be swapped are adjacent siblings and the subtrees are not edited. Subsequently swap is developed to edit subtrees that are swapped. A comparison of the two swapping algorithms to that of subsection 4.1 is then discussed.

Only swapping of adjacent siblings is examined. The first swapping algorithm only swaps subtrees if the target tree has equivalent subtrees in the swapped position. The revised algorithm is shown in Figure 25.

D Array

	1	2	3
1			
2			
3	1	0	2
4	1	1	3
5			
6			

FD Array

	$l(2)-1$	2
$l(4)-1$	0	1
4	1	1

FD Array

	$l(3)-1$	1	2	3
$l(4)-1$	0	1	2	3
4	1	1	2	3

Figure 22: Converting node f to nodes e and a: our algorithm

D Array

	1	2	3
1			
2	1	1	3
3	1	0	2
4	1	1	3
5	2	2	3
6			

FD Array

	$l(2)-1$	2
$l(5)-1$	0	1
2	1	1
3	2	1
4	3	2
5	1	2

FD Array

	$l(3)-1$	1	2	3
$l(5)-1$	0	1	2	3
2	1	1	2	3
3	2	2	1	2
4	3	3	2	3
5	1	2	3	3

Figure 23: Converting node c to nodes e and a: our algorithm

The first algorithm to introduce the use of a swap operation considers swapping adjacent if their corresponding subtrees are equivalent. *swap* looks at the subtree tooted at T_i and its sibling's subtree rooted at $T_{l(i)-1}$ and compares them to the subtree rooted at T'_j and its sibling's subtree rooted at $T'_{l(j)-1}$. T_i must be equivalent to $T`_{l(j)-1}$ and $T_{l(i)-1}$ equivalent to T'_j in order for *swap* to be a candidate for inclusion on a minimum path.

The algorithm for calculating tree distance is very similar to that of section 4.1. But there is a significant difference caused by the addition of *swap*. The operation is added to the calculation of $FD[i,j]$, where T_i and T'_j are keyroots and $l(i) \neq l(x)$ and $l(j) \neq l(y)$. The new primitive operation is a new case added in order to cover all possible mappings that yield $FD[i,j]$.

$$FD[l(l(i)-1)-1, l(l(j)-1)-1] + \text{swap}(i,j)$$

Notice that the root of the previous subtree is $T_{l(l(i)-1)-1}$ for T_i and $T'_{l(l(j)-1)-1}$ for T'_j since the tree is numbered in postorder.

3.3 Swapping and Editing of the Subtrees

The second algorithm to implement a swap operation still only considers swapping adjacent siblings, but allows editing the swapped subtrees. T_i is edited to $T'_{l(j)-1}$ and similarly $T_{l(i)-1}$ is edited to be equivalent

D Array	1	2	3
1	0	1	2
2	1	1	3
3	1	0	2
4	1	1	3
5	2	2	3
6	2	2	2

FD Array	$l(2)-1$	2
$l(6)-1$	0	1
1	1	1
2	2	2
3	3	2
4	4	3
5	2	3
6	1	2

FD Array	$l(3)-1$	1	2	3
$l(6)-1$	0	1	2	3
1	1	0	1	2
2	2	1	1	2
3	3	2	1	2
4	4	3	2	3
5	2	1	2	3
6	1	2	3	2

Figure 24: Converting node a to nodes e and a: our algorithm

to T'_j. The algorithm for calculating tree distance remains the same as that given above. The definition of the operation thus becomes:

$swap(i,j)$ — the cost of performing a swap between subtrees rooted at T_i and $T_{l(i)-1}$ + tree distance between T_i and $T'_{l(j)-1}$ + tree distance between $T_{l(i)-1}$ and T'_j

However, swap and edit become mutually recursive and, therefore, the calculation of complexity must be revisited. Every keyroot except the root calls swap which calls edit. Therefore, the time complexity becomes $O(n^2m^2dd')$. An implementation is possible that does not increase the space complexity from $O(nm)$. Termination is guaranteed because when *edit* is called with two leaves *swap* is not called from within *edit*.

3.4 Comparing the Three New Algorithms

The example given in Figure 26 demonstrates the difference between the algorithms. We will assume that a *deleteTree* operation costs 1, an *insertTree* operation has a cost equal to the number of nodes in the tree and that a *swap* operation has cost 1.

The algorithm without *swap* results in a tree distance of 4 because the subtree rooted at b is deleted and then inserted as the middle sibling of a. In contrast to this, incorporating a simple *swap* operation results in a tree distance of 1. When $x = 9$ and $y = 9$, $FD[5,5]$ utilizes the *swap* as its minimum available approach.

Figure 27 highlights the advantage of adding editing of swapped subtrees. Again the actual cost of swapping two subtrees is considered to be 1. The algorithm with a simple *swap* results in a tree distance of 3 because the subtrees rooted at b are not equivalent and, therefore, when $x = 4$ and $y = 5$, $FD[3,4]$ does not utilize the *swap* operation. Instead the node c in T is deleted and the nodes c and e in T' are inserted. However, the algorithm using *swap* with editing assigns $FD[3,4] = 2$, when $x = 4$ and $y = 5$, because the *swap* costs 1 and the tree distance between the subtree rooted at b in T to that rooted at b in T' is 1.

4 Determining the Cost of Operations

All of the algorithms for solving tree-to-tree or string-to-string correction problems require accurate costing of the operations in order to be useful. As was pointed out in the introduction, establishing a sufficiently high cost for the change operation in the string-to-string case means it is never used in any minimum cost

Notational conventions :

$swap(i, j)$	if $(T_{l(i)-1} = T_j)$ and $(T_i = T_{l(j)-1})$ the cost of performing a swap between subtrees rooted at T_i and $T_{l(i)-1}$

for each $x \in K(T)$ do
  for each $y \in K(T')$ do
    $FD[l(x)-1, l(y)-1] = 0$
    for $i = l(x)$ to x do
      $FD[i, l(y)-1] = FD[l(i)-1, l(y)-1] + \text{deleteTree}(T_i)$
    for $j = l(y)$ to y do
      $FD[l(x)-1, j] = FD[l(x)-1, l(j)-1] + \text{insertTree}(T'_j)$
    for $i = l(x)$ to x do
      for $j = l(y)$ to y do
        $m = \min(FD[i-1, j] + \gamma(T_i \to \Lambda),$
            $FD[i, j-1] + \gamma(\Lambda \to T'_j),$
            $FD[l(i)-1, j] + \text{deleteTree}(T_i),$
            $FD[i, l(j)-1] + \text{insertTree}(T'_j))$
        if $l(i) = l(x)$ and $l(j) = l(y)$ then
          $D[i, j] = \min(m, FD[i-1, j-1] + \gamma(T_i \to T'_j))$
          $FD[i, j] = D[i, j]$
        else if $(i \in K(T)$ and $j \in K(T'))$
          $FD[i, j] = \min(m, FD[l(i)-1, l(j)-1] + D[i, j], FD[l(l(i)-1)-1, l(l(j)-1)-1] + \text{swap}(i,j))$
        else
          $FD[i, j] = \min(m, FD[l(i)-1, l(j)-1] + D[i, j])$

Figure 25: Extending with Swapping of Subtrees

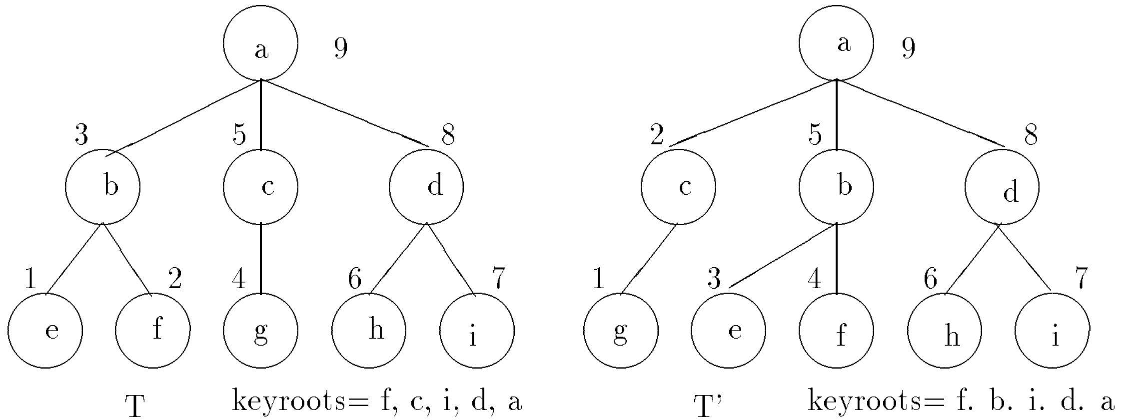

Figure 26: Convert T into T'

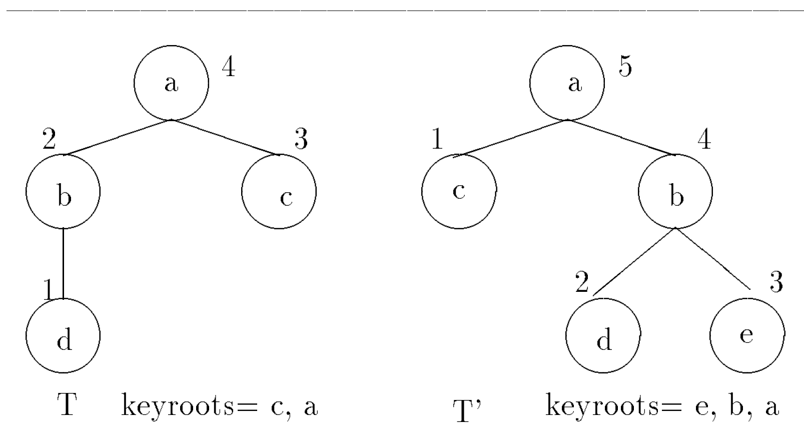

Figure 27: Convert T into T

solution and all changes are accomplished by using insert and delete. With trees, we have a more difficult problem. In the string case it is somewhat reasonable to assume that each operation has a fixed cost, regardless of which characters were being operated on, since characters can be regarded as atomic in some sense.

This would be the case if the strings were considered to be related by some random process, such as the introduction of noise from some communication channel. However, the assumption is no longer true if the strings are considered to be related to each other in a more sophisticated way. For example, one string might be the result of a beginning typist attempting to type the other string. In this case patterns of key arrangement on a keyboard might be significant, with some errors (depressing a key adjacent to the desired one) being more likely than others (depressing a key further away).

In the tree case, the assumption of atomicity certainly does not hold true, and the cost of an operation may vary depending on which node is being deleted, inserted or changed because each node may have a multi-character label and may have several children. This is especially true of structured documents, where a node contianing only text—and thus only a label without any subordinate structure—may contain as much as an entire paragraph of text or as little as a single letter. It is important to understand the context of our algorithms in order to make clear the relationship between cost of operations and various implementation factors, primarily storage considerations.

In Figure 28 we see an example of a case where different costs assigned to different edit operations would result in very different least-cost editing sequences. Assume that deleting and inserting nodes costs exactly 1 unit per child of the node and that changing a node label costs 1 unit. If tree T is transformed into tree T' by deleting node b, the cost is 3 units. If T is changed into T' by deleting a and changing b to a, the cost is 2 units. If we assume that the cost of deleting any node is the same as the cost of deleting any other node, of course, the cost of deleting b is 1 unit, and the cost of deleting a and then changing b to a, is 2 units.

To determine the cost of the operations accurately, we will have to know what data structure is being used. A case can be made for keeping the costs of insertions and deletions constant regardless of which node is concerned. If a tree is stored as a series of special characters indicating structural information interspersed with node labels (possibly text), as might be the case for a Lisp function, for example, then deletion (insertion) is a matter of removing (adding) a node label and some fixed number of special characters. In a Lisp function this would be removing (adding) one set of brackets and the first atom in the brackets. If, however, the tree is stored as a set of vertices and a set of edges, or if it is stored with each node having a

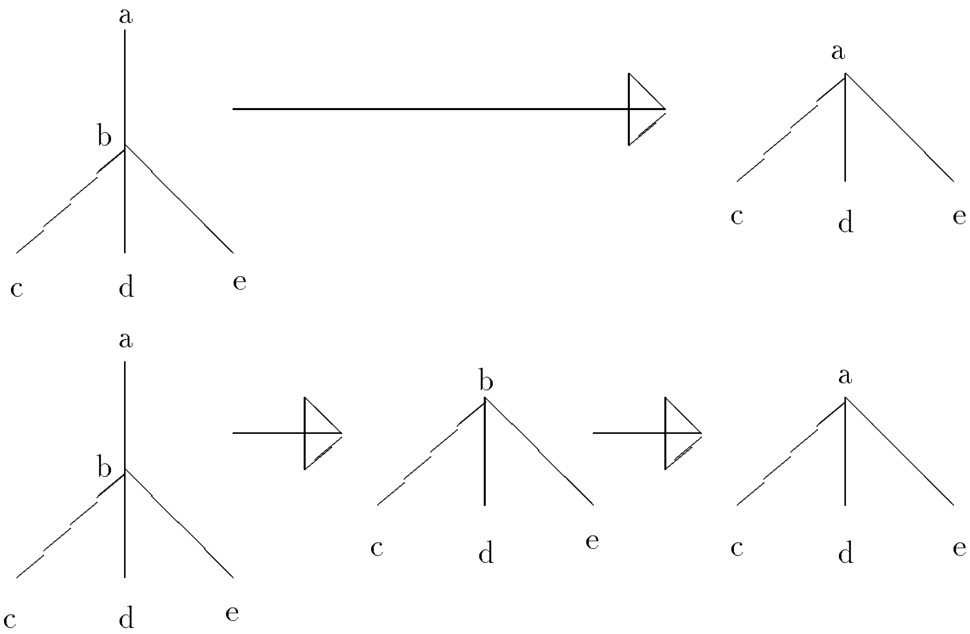

Figure 28: Different Costs for Same Transformation

label and a set of pointers to children of that node, then this assumption does not hold. In either of these cases, the cost of inserting or deleting a node would vary according to the number of children that it had. If we have a definite idea of the data structure to be used to represent the tree, determining the cost of insertions and deletions of internal nodes is relatively straightforward.

Similarly, the cost of a deleteTree operation is, under one representation of trees, a deletion of everything between two special characters, and therefore node-independent. If the tree in question is represented as labelled nodes with pointers to their children, deleteTree is again no problem, it costs exactly whatever it costs to de-allocate memory. If a tree is represented as a set of vertices and a set of edges, however, then determining the cost of deleteTree becomes a very expensive operation in itself, since the entirety of both structures might need to be scanned for each structural node in the original tree.

It is less intuitively easy to establish an acceptable cost for any version of insertTree. We can say, though, that using any of the data structures suggested to represent trees, it should probably have cost proportional to the size of the subtree being inserted since, under any storage method for trees, the nodes being inserted must come from somewhere.

In the case of the swap operation, if the tree representation is as labeled nodes with pointers to their children, the cost is that of changing two pointers. In a representation of a tree such as that in a Lisp function then a swap is similar to a tree deletion followed by an insertion. For a tree represented by a set of vertices and a set of edges, both structures would be scanned in their entirety, similar to the $deleteTree$ situation.

Changing leaf node labels can be accurately modelled by a constant value or by finding the string-to-string edit distance between them. This could be done in a pre-processing stage and the results tabulated in an $n \times m$ array in $O(nmll')$ where l is the longest node label of T and l' is the longest of T'. The internal node labels could be strings (such as the SGML tag <chapter>) or they could be parse tokens of some type. If they are strings, then they can be treated exactly the same as leaf nodes. If not, the distance between them depends entirely on the way that the parse tokens can be compared.

5 Conclusion

String-to-string correction can be generalized to trees, and ultimately to structured documents. There are several algorithms for dealing with tree-to-tree correction. To extend these to structured documents, several approaches are possible. Two obvious ones are to add more appropriate primitives to the sets used by the existing algorithms, or to post-process the output of existing algorithms. We have presented a small advance on the Zhang and Shasha algorithm that is of the first type.

It remains to consider a richer set of document editing primitives, and to integrate them into a single algorithm using the two approaches discussed. In addition, it will be important to have a single cost metric reflecting both structural transformations and value transformations, and to extend these algorithms with considerations of value transformations (string-to-string corrections) at the leaves.

Acknowledgements

Doug Hamilton contributed the description of the algorithm. Henk Meijer and Selim Akl read and commented on drafts of an earlier version of the paper.

References

[1] Lloyd Allison and Trevor I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23:305–310, December 1986.

[2] David T. Barnard and Ian A. Macleod. Maestro working paper 0: An archive of structured texts. Technical Report 89-262, Department of Computing and Information Science, Queen's University, November 14 1989.

[3] D.T. Barnard, L. Burnard, J.-P. Gaspart, L. Price, C.M. Sperberg-McQueen, and N. Varile. Hierarchical encoding of text: Technical problems and sgml solutions. *Computers and the Humanities*, 1995 (accepted 1994). Invited paper in special issue dealing with the Text Encoding Initiative.

[4] Karel Culik II and Derick Wood. A note on some tree similarity measures. *Information Processing Letters*, 15(1):39–42, August 1982.

[5] International Organization for Standardization. Geneva/New York. *Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*, ISO 8879:155, 1986.

[6] R. Lowrance and R.A. Wagner. The extended string-to-string correction problem. *Journal of the ACM*, 22(2):177–183, April 1975.

[7] Jean Pallo. Enumerating, ranking and unranking binary trees. *The Computer Journal*, 29(2):171–175, 1986.

[8] Jean Pallo. On the rotational distance in the lattice of binary trees. *Information Processing Letters*, 25:369–373, July 1987.

[9] D.F. Robinson. Comparison of labelled trees with valency three. *Journal of Combinatorial Theory*, 11:105–119, 1971.

[10] D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.

[11] David Sankoff and Joseph B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

[12] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[13] D. Shasha, K. Jeong, T-L. Wang, and K. Zhang. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(2), 1994.

[14] D. Shasha and T.-L. Wang. New techniques for best match retrieval. *Transactions on Office Information Systems*, 8(2):140–158, 1990.

[15] D. Shasha, T-L. Wang, K. Zhang, and F. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics*, 24(2), 1994.

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

[17] D. Shasha, K. Zhang, and R. Statman. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.

[18] D. Shasha, K. Zhang, and T-L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, (to appear).

[19] Dennis Shasha and Kaizhong Zhang. Fast parallel algorithms for the unit cost editing distance between trees. Technical report, Courant Institute of Mathematical Sciences, NYU, 1989. Draft Version.

[20] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–423, 1979.

[21] R.A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pages 218–223, 1975.

[22] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[23] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, December 1989.

A Programs for Object-to-Object Correction

We present programs in Turing implementing the algorithms in the paper.

A.1 String-to-String Correction: Wagner and Fischer

```
% file Tgamma.t
function gamma(p, q: char): int
    if p = q then
        result 0
    else
        result 1
    end if
end gamma
```

```
% file Ta1_1.t
include "Tgamma.t"

const null := " "

var d: array 0 .. 50, 0 .. 50 of int
var s, t: string

put "Input source and target strings:"
get s, t                                                    10
put s
put t

const n := length(s)
const m := length(t)

put "Lengths are ", n, " and ", m

d(0,0) := 0
for i: 1 .. n                                               20
    d(i, 0) := d(i-1, 0) + gamma(s(i), null)
end for
for j: 1 .. m
    d(0, j) := d(0, j-1) + gamma(null, t(j))
end for
for i: 1 .. n
    for j: 1 .. m
        d(i, j) := min( d(i-1, j-1) + gamma(s(i), t(j)),
            min(d(i-1, j) + gamma(s(i), null),
            d(i, j-1) + gamma(null, t(j))))                30
    end for
end for
put "Distance is ", d(n, m)
```

To create an operation list, these additional program fragments are appended to the program already shown.

```
% file Trepchg.t
% c at position i to d at position j with cost v
function repChange(c: char, i: int, d: char, j, v: int): string
    if c = d then
        result ""
    else
        result "[" + c + "|" + intstr(i) + "," + d + "|" + intstr(j) + ":" + intstr(v) + "]"
    end if
end repChange
```

```
include "Ta1_1.t"
include "Trepchg.t"

% extract the operation list working backwards
var l := ""
var i := n
var j := m
loop
    exit when i = 0 and j = 0
    if d(i, j) = d(i-1, j) + gamma(s(i), null) then                                10
        l := repChange(s(i), i, null, j, gamma(s(i), null)) + l
        i -= 1
    elsif d(i, j) = d(i, j-1) + gamma(null, t(j)) then
        l := repChange(null, i, t(j), j, gamma(null, t(j))) + l
        j -= 1
    else
        l := repChange(s(i), i, t(j), j, gamma(s(i), t(j))) + l
        i -= 1
        j -= 1
    end if                                                                          20
end loop
put l
```

A.2 Tree-to-Tree Correction: Selkow

```
% file Tsize.t
function treeSize(n: ^node): int
    if n = nil then
        result 0
    else
        var s := 0
        for i: 1 .. n -> fanout
            s += treeSize(n -> child(i))
        end for
        result s + 1 % count the node itself                          10
    end if
end treeSize
```

```
% file Ta2_1defs.t
type costType: int

const maxFanout := 10
const dSize := maxFanout + 1

type node:
    record
        name: char
        fanout: 0 .. maxFanout                                        10
        child: array 1 .. maxFanout of ^node
    end record
```

```
% Ta2_1mkprt.t
procedure makeTree(s: string, var i: int, var n: ^node)
    new n
    n -> name := s(i+1) % skip the opening bracket
    n -> fanout := 0
    i += 2
    loop
        exit when s(i) = ")"
        n -> fanout += 1
        makeTree(s, i, n -> child(n -> fanout))                       10
    end loop
    i += 1 % skip the closing bracket
end makeTree

procedure printTree(n: ^node, indent: int)
    const blanks := "                         "
    put blanks(1 .. indent), n -> name
    for i: 1 .. n -> fanout
        printTree(n -> child(i), indent+1)
    end for                                                           20
end printTree
```

```
% file Ta2_1costs.t
function deleteTree(n: ^node): costType
    result 1
end deleteTree

function insertTree(n: ^node): costType
    result treeSize(n)
end insertTree
```

```
% file Ta2_1edit.t
function edit(n, m: ^node): costType
    const p := n -> fanout
    const q := m -> fanout
```

```
    var d: array 0 .. dSize, 0 .. dSize of costType
    d(0, 0) := gamma(n -> name, m -> name)
    for i: 1 .. p
        d(i, 0) := d(i-1, 0) + deleteTree(n -> child(i))
    end for
    for j: 1 .. q                                                          10
        d(0, j) := d(0, j-1) + insertTree(m -> child(j))
    end for
    for i: 1 .. p
        for j: 1 .. q
            d(i, j) := min(d(i-1, j-1) + edit(n -> child(i), m -> child(j)),
                min(d(i-1, j) + deleteTree(n -> child(i)),
                d(i, j-1) + insertTree(m -> child(j))))
        end for
    end for
    result d(p, q)                                                         20
end edit
```

```
% algorithm of Selkow
include "Tgamma.t"
include "Ta2_1defs.t"
include "Ta2_1mkprt.t"
include "Tsize.t"
include "Ta2_1costs.t"
include "Ta2_1edit.t"

var t1, t2: ^node
put "Examples of legal trees are (a(b(c))) and (a(b)(c))."                 10
put "Node labels are single characters.  No blanks are allowed."
put "Enter source and target:"
var s1, s2: string
get s1, s2
var p := 1
makeTree(s1, p, t1)
p := 1
makeTree(s2, p, t2)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)                                                           20
put "Target is of size ", treeSize(t2)
printTree(t2, 1)
put "Distance is: ", edit(t1, t2)
```

This variation creates an operation list for the minimal distance edit.

```
% file Ta2_2defs.t
type costType:
    record
        c: int
        l: string
    end record
const maxFanout := 10
const dSize := maxFanout + 1
type node:
    record                                                          10
        name: char
        number: int
        fanout: 0 .. maxFanout
        child: array 1 .. maxFanout of ^node
    end record
```

```
% file Ta2_2mkprt.t
procedure makeTree(s: string, var i: int, var n: ^node, var nodes: int)
    new n
    n -> name := s(i+1) % skip the opening bracket
    n -> fanout := 0
    n -> number := nodes
    nodes += 1
    i += 2
    loop
        exit when s(i) = ")"                                        10
        n -> fanout += 1
        makeTree(s, i, n -> child(n -> fanout), nodes)
    end loop
    i += 1 % skip the closing bracket
end makeTree

procedure printTree(n: ^node, indent: int)
    const blanks := "                    "
    put blanks(1 .. indent), n -> name, "[", n -> number, "]"
    for i: 1 .. n -> fanout                                         20
        printTree(n -> child(i), indent+1)
    end for
end printTree
```

```
% file Ta2_2costs.t
function deleteTree(n: ^node): costType
    var r: costType
    r.c := 1; r.l := ""
    result r
end deleteTree

function insertTree(n: ^node): costType
    var r: costType
    r.c := treeSize(n); r.l := ""                                   10
    result r
end insertTree
```

```
% file Ta2_2edit.t
function edit(n, m: ^node): costType
    const a := n -> number
    const b := m -> number
    const p := n -> fanout
    const q := m -> fanout
    var d: array 0 .. dSize, 0 .. dSize of costType
    var g := gamma(n -> name, m -> name)
```

```
    d(0, 0).c := g
    d(0, 0).l := repChange(n -> name, a, m -> name, b, g)                                10
    for i: 1 .. p
        const nci := n -> child(i)
        const del := deleteTree(nci).c
        d(i, 0).c := d(i-1, 0).c + del
        d(i, 0).l := repChange(nci -> name, nci -> number, "T", b, del) + d(i-1, 0).l
    end for
    for j: 1 .. q
        const mcj := m -> child(j)
        const ins := insertTree(mcj).c
        d(0, j).c := d(0, j-1).c + ins                                                   20
        d(0, j).l := repChange("T", a, mcj -> name, mcj -> number, ins) + d(0, j-1).l
    end for
    for i: 1 .. p
        for j: 1 .. q
            const nci := n -> child(i)
            const mcj := m -> child(j)
            const change := edit(nci, mcj)
            const costToChange := d(i-1, j-1).c + change.c
            const del := deleteTree(nci).c
            const costToDelete := d(i-1, j).c + del                                      30
            const ins := insertTree(mcj).c
            const costToInsert := d(i, j-1).c + ins
            if costToChange <= costToDelete and costToChange <= costToInsert then
                d(i, j).c := costToChange
                d(i, j).l := d(i-1, j-1).l + change.l
            elsif costToDelete <= costToInsert then
                d(i, j).c := costToDelete
                d(i, j).l := repChange(nci -> name, nci -> number, "T", b, del) + d(i-1, j).l
            else
                d(i, j).c := costToInsert                                                40
                d(i, j).l := repChange("T", a, mcj -> name, mcj -> number, ins) + d(i, j-1).l
            end if
        end for
    end for
    result d(p, q)
end edit
```

```
% algorithm of Selkow
include "Tgamma.t"
include "Ta2_2defs.t"
include "Ta2_2mkprt.t"
include "Trepchg.t"
include "Tsize.t"
include "Ta2_2costs.t"
include "Ta2_2edit.t"
var t1, t2: ^node
put "Examples of legal trees are (a(b(c))) and (a(b)(c))."                              10
put "Node labels are single characters.  No blanks are allowed."
put "Enter source and target:"
var s1, s2: string
get s1, s2
var p := 1; var nodes := 1
makeTree(s1, p, t1, nodes)
p := 1; nodes := 1
makeTree(s2, p, t2, nodes)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)                                                                         20
put "Target is of size ", treeSize(t2)
printTree(t2, 1)
const e := edit(t1, t2)
put "Distance is: ", e.c
put "with operations: ", e.l
```

A.3 Tai's Tree-to-Tree Algorithm

```
% file Ta3defs.t
type costType: int
const infinity := 999999
const null := " "

const maxFanout := 10
const dSize := maxFanout + 1

type node:
    record
        name: char
        number: int
        fanout: 0 .. maxFanout
        child: array 1 .. maxFanout of ^node
        parent: ^node
    end record

const maxNodes := 10
const maxDepth := 10

type nodeList: array 1 .. maxNodes of ^node

var e: array 1.. maxNodes, 1.. maxNodes, 1.. maxDepth, 1.. maxDepth, 1.. maxDepth, 1.. maxDepth of costType
var cc, d: array 1 .. maxNodes, 1 .. maxNodes of costType
```

```
% file Ta3mkprt.t
procedure linkItUp(s: string, var i: int, var n: ^node, p: ^node, var nodes: int, var nl: nodeList)
    new n
    n -> name := s(i+1) % skip the opening bracket
    n -> number := nodes
    nl(nodes) := n
    n -> fanout := 0
    n -> parent := p
    nodes += 1
    i += 2
    loop
        exit when s(i) = ")"
        n -> fanout += 1
        linkItUp(s, i, n -> child(n -> fanout), n, nodes, nl)
    end loop
    i += 1 % skip the closing bracket
end linkItUp

procedure makeTree(s: string, var n: ^node, var nl: nodeList)
    var nodes := 1
    var p := 1 % index into specification string
    linkItUp(s, p, n, nil, nodes, nl)
end makeTree

procedure printTree(n: ^node, indent: int)
    const blanks := "                    "
    put blanks(1 .. indent), n -> name, "[", n -> number, ">" ..
    if n -> parent = nil then
        put "*]"
    else
        put n -> parent -> number, "]"
    end if
    for i: 1 .. n -> fanout
        printTree(n -> child(i), indent+1)
    end for
end printTree
```

```
% file Ta3step1.t
```

```
function parentOf(i: int, nl:nodeList): int
    if i = 1 then
        result 0
    else
        result nl(i) -> parent -> number
    end if
end parentOf

procedure setValue(s, u, i, t, v, j, x, y: int, t1, t2: ^node, nl1, nl2: nodeList)        10
    if (s = u and u = i) and (t = v and v = j) then
        e(s, u, i, t, v, j) := gamma(nl1(i) -> name, nl2(j) -> name)
    elsif (s = u and u = i) or (t < v and v = j) then
        e(s, u, i, t, v, j) := e(s, u, i, t, parentOf(j, nl2), j-1) + gamma(" ", nl2(j) -> name)
    elsif (s < u and u = i) or (t = v and v = j) then
        e(s, u, i, t, v, j) := e(s, parentOf(i, nl1), i-1, t, v, j) + gamma(nl1(i) -> name, " ")
    else
        e(s, u, i, t, v, j) := min(e(s, x, i, t, v, j), min(e(s, u, i, t, y, j),
            e(s, u, x-1, t, v, y-1) + e(x, x, i, y, y, j)))
    end if        20
end setValue

procedure step1(t1: ^node, nl1: nodeList, t2: ^node, nl2: nodeList)
    var x := -1
    var y := -1
    for i: 1 .. treeSize(t1)
        for j: 1 .. treeSize(t2)
            var u := i
            loop
                exit when u = 0        30
                var s := u
                loop
                    exit when s = 0
                    var v := j
                    loop
                        exit when v = 0
                        var t := v
                        loop
                            exit when t = 0
                            setValue(s, u, i, t, v, j, x, y, t1, t2, nl1, nl2)        40
                            t := parentOf(t, nl2)
                        end loop
                        y := v
                        v := parentOf(v, nl2)
                    end loop
                    s := parentOf(s, nl1)
                end loop
                x := u
                u := parentOf(u, nl1)
            end loop        50
        end for
    end for
end step1
```

```
% file Ta3step2.t
procedure step2(t1, t2: ^node, nl1, nl2: nodeList)
    cc(1, 1) := 0
    for i: 2 .. treeSize(t1)
        cc(i, 1) := i
    end for
    for j: 2 .. treeSize(t2)
        cc(1, j) := j
    end for
    for i: 2 .. treeSize(t1)        10
        for j: 2 .. treeSize(t2)
            cc(i, j) := infinity
            var s := parentOf(i, nl1)
```

```
            loop
                var t := parentOf(j, nl2)
                loop
                    cc(i, j) := min(cc(i, j), cc(s, t) + e(s, parentOf(i, nl1), i−1, t, parentOf(j, nl2), j−1)
                        − gamma(nl1(s) -> name, nl2(t) -> name))
                    exit when t = 1
                    t := parentOf(t, nl2)                                                          20
                end loop
                exit when s = 1
                s := parentOf(s, nl1)
            end loop
            cc(i, j) := cc(i, j) + gamma(nl1(i) -> name, nl2(j) -> name)
        end for
    end for
end step2
```

```
% file Ta3step3.t
procedure step3
    d(1, 1) := 0
    for i: 2 .. treeSize(t1)
        d(i, 1) := d(i−1, 1) + gamma(nl1(i) -> name, null)
    end for
    for j: 2 .. treeSize(t2)
        d(1, j) := d(1, j−1) + gamma(null, nl2(j) -> name)
    end for
    for i: 2 .. treeSize(t1)                                                                       10
        for j: 2 .. treeSize(t2)
            d(i, j) := min(cc(i, j), min(d(i−1, j) + gamma(nl1(i) -> name, null),
                d(i, j−1) + gamma(null, nl2(j) -> name)))
        end for
    end for
end step3
```

```
% algorithm of Tai
include "Tgamma.t"
include "Ta3defs.t"
include "Ta3mkprt.t"
include "Tsize.t"
include "Ta3step1.t"
include "Ta3step2.t"

var t1, t2: ^node
put "Examples of legal trees are (a(b(c))) and (a(b)(c))."                                         10
put "Node labels are single characters.  No blanks are allowed."
put "Enter source and target:"
var s1, s2: string
get s1, s2
var nl1, nl2: nodeList
makeTree(s1, t1, nl1)
makeTree(s2, t2, nl2)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)
put "Target is of size ", treeSize(t2)                                                             20
printTree(t2, 1)

step1(t1, nl1, t2, nl2)
step2(t1, t2, nl1, nl2)
include "Ta3step3.t"
step3
put "Distance is: ", d(treeSize(t1), treeSize(t2))
```

A.4 Zhang and Shasha's Tree-to-Tree Algorithm

```
% file Ta4defs.t
type costType: int
const infinity := 999999
const null := " "

const maxFanout := 10
const dSize := maxFanout + 1

type node:
    record
        name: char
        number, leftmost: int
        keyroot: boolean
        fanout: 0 .. maxFanout
        child: array 1 .. maxFanout of ^node
        parent: ^node
    end record

const maxNodes := 10
const fdSize := maxNodes + 1
const maxDepth := 4

type nodeList: array 1..maxNodes of ^node

var fd: array 0 .. fdSize, 0 .. fdSize, 0 .. fdSize, 0 .. fdSize of costType
var d: array 1 .. maxNodes, 1 .. maxNodes of costType
```

```
% file Ta4mkprt.t
procedure linkItUp(s: string, var i: int, var n: ^node, p: ^node)
    new n
    n -> name := s(i+1) % skip the opening bracket
    n -> fanout := 0
    n -> parent := p
    i += 2
    loop
        exit when s(i) = ")"
        n -> fanout += 1
        linkItUp(s, i, n -> child(n -> fanout), n)
    end loop
    i += 1 % skip the closing bracket
end linkItUp

procedure postorder(var n: ^node, var nodes: int, var nl: nodeList, makeKeyroot: boolean)
    if n not = nil then
        n -> leftmost := nodes
        n -> keyroot := makeKeyroot
        for i: 1 .. n -> fanout
            postorder(n -> child(i), nodes, nl, i not= 1)
        end for
        n -> number := nodes
        nl(nodes) := n
        nodes += 1
    end if
end postorder

procedure makeTree(s: string, var n: ^node, var nl: nodeList)
    var nodes := 1
    var p := 1 % index into specification string
    linkItUp(s, p, n, nil)
    postorder(n, nodes, nl, true)
end makeTree

function repNode(n: ^node): string
    var r := n -> name + "[" + intstr(n -> number) + ">"
```

```
    if n -> parent = nil then
        r += "*]"
    else                                                                    40
        r += intstr(n -> parent -> number) + "]"
    end if
    if n -> keyroot then
        r += "*"
    else
        r += "-"
    end if
    result r + intstr(n -> leftmost)
end repNode
                                                                            50
procedure printTree(n: ^node, indent: int)
    const blanks := "                              "
    put blanks(1 .. indent), repNode(n)
    for i: 1 .. n -> fanout
        printTree(n -> child(i), indent+1)
    end for
end printTree
```

```
% file Ta4edit.t
function access(l, m, n, o: int): costType
    var lprime := l
    var mprime := m
    var nprime := n
    var oprime := o
    if l > m then
        lprime := 0
        mprime := 0
    end if                                                                  10
    if n > o then
        nprime := 0
        oprime := 0
    end if
    result fd(lprime, mprime, nprime, oprime)
end access

for x: 1 .. treeSize(t1)
    if nl1(x) -> keyroot then
        const lx := nl1(x) -> leftmost                                      20
        for y: 1 .. treeSize(t2)
            if nl2(y) -> keyroot then
                const ly := nl2(y) -> leftmost
                fd(0, 0, 0, 0) := 0
                for i: lx .. x
                    fd(lx, i, 0, 0) := access(lx, i−1, 0, 0) + gamma(nl1(i) -> name, null)
                end for
                for j: ly .. y
                    fd(0, 0, ly, j) := access(0, 0, ly, j−1) + gamma(null, nl2(j) -> name)
                end for                                                     30
                for i: lx .. x
                    for j: ly .. y
                        var part := min(access(lx, i−1, ly, j) + gamma(nl1(i) -> name, null),
                                access(lx, i, ly, j−1) + gamma(null, nl2(j) -> name))
                        if nl1(i) -> leftmost = lx and nl2(j) -> leftmost = ly then
                            fd(lx, i, ly, j) := min(part,
                                access(lx, i−1, ly, j−1) + gamma(nl1(i) -> name, nl2(j) -> name))
                            d(i, j) := fd(lx, i, ly, j)
                        else
                            fd(lx, i, ly, j) := min(part,                   40
                                access(lx, nl1(i) -> leftmost − 1, ly, nl2(j) -> leftmost − 1) + d(i, j))
                        end if
                    end for
                end for
            end if
```

```
        end for
    end if
end for
```

```
% file Trdtree.t
put "Examples of legal trees are (a(b(c))) and (a(b)(c))."
put "Node labels are single characters.  No blanks are allowed."
put "Enter source and target:"
var s1, s2: string
get s1, s2
put s1, skip, s2
```

```
include "Tgamma.t"
include "Ta4defs.t"
include "Tsize.t"

include "Ta4mkprt.t"

include "Trdtree.t"
var t1, t2: ^node
var nl1, nl2: nodeList
makeTree(s1, t1, nl1)                                              10
makeTree(s2, t2, nl2)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)
put "Target is of size ", treeSize(t2)
printTree(t2, 1)
include "Ta4edit.t"
put "Distance is: ", d(treeSize(t1), treeSize(t2))
```

A.5 Subtree Insertion and Deletion

```
% file N1defs.t
type costType: int
const infinity := 999999
const null := " "

const maxFanout := 10
const dSize := maxFanout + 1

type node:
    record                                                         10
        name: char
        number, leftmost: int
        keyroot: boolean
        fanout: 0 .. maxFanout
        child: array 1 .. maxFanout of ^node
        parent: ^node
    end record

const maxNodes := 10
const fdSize := maxNodes + 1                                       20
const maxDepth := 4

type nodeList: array 1..maxNodes of ^node

var fd: array 0 .. fdSize, 0 .. fdSize of costType
var d: array 1 .. maxNodes, 1 .. maxNodes of costType
```

```
% file N1edit.t
for x: 1 .. treeSize(t1)
    if nl1(x) -> keyroot then
        const lx := nl1(x) -> leftmost
        for y: 1 .. treeSize(t2)
            if nl2(y) -> keyroot then
                const ly := nl2(y) -> leftmost
                fd(lx - 1, ly - 1) := 0
                for i: lx .. x
                    fd(i, ly - 1) := fd(nl1(i) -> leftmost - 1, ly - 1) + deleteTree(nl1(i))          10
                end for
                for j: ly .. y
                    fd(lx - 1, j) := fd(lx - 1, nl2(j) -> leftmost - 1) + insertTree(nl2(j))
                end for
                for i: lx .. x
                    for j: ly .. y
                        const li := nl1(i) -> leftmost
                        const lj := nl2(j) -> leftmost
                        var part := min(fd(i - 1, j) + gamma(nl1(i) -> name, null),
                            min(fd(i, j - 1) + gamma(null, nl2(j) -> name),                           20
                            min(fd(li - 1, j) + deleteTree(nl1(i)),
                            fd(i, lj - 1) + insertTree(nl2(j)))))
                        if li = lx and lj = ly then
                            d(i, j) := min(part, fd(i - 1, j - 1) + gamma(nl1(i) -> name, nl2(j) -> name))
                            fd(i, j) := d(i, j)
                        else
                            fd(i, j) := min(part, fd(li - 1, lj - 1) + d(i, j))
                        end if
                    end for
                end for                                                                               30
            end if
        end for
    end if
end for
```

```
% file N1.t
```

```
include "Tgamma.t"
include "N1defs.t"
include "Tsize.t"

include "Ta4mkprt.t"

include "Trdtree.t"
var t1, t2: ^node
var nl1, nl2: nodeList                                                              10
makeTree(s1, t1, nl1)
makeTree(s2, t2, nl2)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)
put "Target is of size ", treeSize(t2)
printTree(t2, 1)
include "Ta2_1costs.t"
include "N1edit.t"
put "Distance is: ", d(treeSize(t1), treeSize(t2))
```

A.6 Swapping Subtrees

```
% file N2swap.t
function equalTrees(p, q: ^node): boolean
    if p = nil then
        result q = nil
    elsif q = nil then
        result false
    elsif p -> name not = q -> name or p -> fanout not = q -> fanout then
        result false
    else
        var i := 1                                                        10
        loop
            exit when i > p -> fanout
            if not equalTrees(p -> child(i), q -> child(i)) then
                result false
            end if
            i += 1
        end loop
        result true
    end if
end equalTrees                                                            20

function swap(i, sibOfi, j, sibOfj: ^node) : costType
    if i -> parent not = nil and j -> parent not = nil
        and equalTrees(i, sibOfj) and equalTrees(sibOfi, j) then
        result 1
    else
        result infinity
    end if
end swap
```

```
% file N2edit.t
for i: 0..treeSize(t1)
    for j: 0..treeSize(t2)
        fd(i, j) := infinity
    end for
end for
d(treeSize(t1), treeSize(t2)) := 0
for x: 1 .. treeSize(t1)
    if nl1(x) -> keyroot then
        const lx := nl1(x) -> leftmost                                    10
        for y: 1 .. treeSize(t2)
            if nl2(y) -> keyroot then
                const ly := nl2(y) -> leftmost
                fd(lx - 1, ly - 1) := 0
                for i: lx .. x
                    fd(i, ly - 1) := fd(nl1(i) -> leftmost - 1, ly - 1) + deleteTree(nl1(i))
                end for
                for j: ly .. y
                    fd(lx - 1, j) := fd(lx - 1, nl2(j) -> leftmost - 1) + insertTree(nl2(j))
                end for                                                   20
                for i: lx .. x
                    for j: ly .. y
                        const li := nl1(i) -> leftmost
                        const lj := nl2(j) -> leftmost
                        fd(i, j) := min(fd(i - 1, j) + gamma(nl1(i) -> name, null),
                            min(fd(i, j - 1) + gamma(null, nl2(j) -> name),
                            min(fd(li - 1, j) + deleteTree(nl1(i)),
                            fd(i, lj - 1) + insertTree(nl2(j)))))
                        if li = lx and lj = ly then
                            fd(i, j) := min(fd(i, j), fd(i - 1, j - 1) + gamma(nl1(i) -> name, nl2(j) -> name)) 30
                            d(i, j) := fd(i, j)
                        else
                            fd(i, j) := min(fd(i, j), fd(li - 1, lj - 1) + d(i, j))
                            if nl1(i) -> keyroot and nl1(i) -> parent not= nil
```

```
                                          and nl2(j) -> keyroot and nl2(j) -> parent not= nil then
                                          fd(i, j) := min(fd(i, j),
                                              fd(nl1(li - 1) -> leftmost - 1, nl2(lj - 1) -> leftmost - 1)
                                              + swap (nl1(i), nl1(li - 1), nl2(j), nl2(lj - 1)))
                                      end if
                                  end if                                                                    40
                              end for
                          end for
                      end if
                  end for
              end if
end for
```

```
% file N2.t
include "Tgamma.t"
include "N1defs.t"
include "Tsize.t"

include "Ta4mkprt.t"

include "Trdtree.t"
var t1, t2: ^node
var nl1, nl2: nodeList                                                                                     10
makeTree(s1, t1, nl1)
makeTree(s2, t2, nl2)
put "Source is of size ", treeSize(t1)
printTree(t1, 1)
put "Target is of size ", treeSize(t2)
printTree(t2, 1)
include "Ta2_1costs.t"
include "N2swap.t"
include "N2edit.t"
put "Distance is: ", d(treeSize(t1), treeSize(t2))                                                        20
```