SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS*

KAIZHONG ZHANG† AND DENNIS SHASHA‡

Abstract. Ordered labeled trees are trees in which the left-to-right order among siblings is significant. The distance between two ordered trees is considered to be the weighted number of edit operations (insert, delete, and modify) to transform one tree to another. The problem of approximate tree matching is also considered. Specifically, algorithms are designed to answer the following kinds of questions:

1. What is the distance between two trees?
2. What is the minimum distance between $T_1$ and $T_2$ when zero or more subtrees can be removed from $T_2$?
3. Let the pruning of a tree at node $n$ mean removing all the descendants of node $n$. The analogous question for prunings as for subtrees is answered.

A dynamic programming algorithm is presented to solve the three questions in sequential time $O(|T_1| \times |T_2| \times \min(\text{depth}(T_1), \text{leaves}(T_1)) \times \min(\text{depth}(T_2), \text{leaves}(T_2)))$ and space $O(|T_1| \times |T_2|)$ compared with $O(|T_1| \times |T_2| \times (\text{depth}(T_1))^3 \times (\text{depth}(T_2))^3)$ for the best previous published algorithm due to Tai [J. Assoc. Comput. Mach., 26 (1979), pp. 422-433]. Further, the algorithm presented here can be parallelized to give time $O(|T_1| + |T_2|)$.

Key words. trees, editing distance, parallel algorithm, dynamic programming, pattern recognition

AMS(MOS) subject classifications. 68P05, 68Q25, 68Q20, 68R10

1. Motivation.

1.1. Applications. Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. As such they can represent grammar parses, image descriptions, and many other phenomena. Comparing such trees is a way to compare scenes, parses, and so on.

As an example, consider the secondary structure comparison problem for RNA. Because RNA is a single strand of nucleotides, it folds back onto itself into a shape that is topologically a tree (called its secondary structure). Each node of this tree contains several nucleotides. Nodes have colorful labels such as “bulge” and “hairpin.” Various researchers [ALKBO], [BSSBWD], [DD] have observed that the secondary structure influences translation rates (from RNA to proteins). Because different sequences can produce similar secondary structures [DA], [SK], comparisons among secondary structures are necessary to understanding the comparative functionality of different RNAs. Previous methods for comparing multiple secondary structures of RNA molecules represent the tree structures as parenthesized strings [S88]. These have been recently converted to using our tree distance algorithms.

Currently we are implementing a package containing algorithms described in this paper and some other related algorithms. A preliminary version of the package is being used at the National Cancer Institute for the RNA comparison problem.

1.2. Algorithmic approach. The tree distance problem is harder than the string distance problem. Intuitively, here is why. In the string case, if $S_1[i] = S_2[j]$, then the
distance between \(S_1[1..i-1]\) and \(S_2[1..j-1]\) is the same as between \(S_1[1..i]\) and \(S_2[1..j]\). The main difficulty in the tree case is that preserving ancestor relationships in the mapping between trees prevents the analogous implication from holding.

By introducing the distance between ordered forests and careful elimination of certain subtree-to-subtree distance calculations we are able to improve the time and space of best previous published algorithm [T]. Note that the improvement of space for this problem is extremely important in practical applications.

Besides improving on the time and space of the best previous algorithm [T], our algorithm is far simpler to understand and to implement. In style, it resembles algorithms for computing the distance between strings. In fact, the string distance algorithm is a special case of our algorithm when the input is a string.

2. Definitions.

2.1. Edit operations and editing distance between trees. Let us consider three kinds of operations. Changing node \(n\) means changing the label on \(n\). Deleting a node \(n\) means making the children of \(n\) become the children of the parent of \(n\) and then removing \(n\). Inserting is the complement of delete. This means that inserting \(n\) as the child of \(n'\) will make \(n\) the parent of a consecutive subsequence of the current children of \(n'\). Figs. 1–3 illustrate these editing operations.

\[\begin{align*}
T_1 & \quad (a \rightarrow b) \quad T_2 \\
\quad (a \rightarrow b) & \quad \Rightarrow \\
\quad (b \rightarrow \Lambda) & \quad \Rightarrow \\
\quad (\Lambda \rightarrow b) & \quad \Rightarrow
\end{align*}\]

Fig. 1

Fig. 2

Fig. 3
(1) Change. To change one node label to another.
(2) Delete. To delete a node. (All children of the deleted node $b$ become children
of the parent $a$.)
(3) Insert. To insert a node. (A consecutive sequence of siblings among the children
of $a$ become the children of $b$.)

Following [WF] and [T], we represent an edit operation as a pair $(a, b) \neq (\Lambda, \Lambda)$,
sometimes written as $a \to b$, where $a$ is either $\Lambda$ or a label of a node in tree $T_1$ and $b$
is either $\Lambda$ or a label of a node in tree $T_2$. We call $a \to b$ a change operation if $a \neq \Lambda$
and $b \neq \Lambda$; a delete operation if $b = \Lambda$; and an insert operation if $a = \Lambda$. Since many
nodes may have the same label, this notation is potentially ambiguous. It could be
made precise by identifying the nodes as well as their labels. However, in this paper,
which node is meant will always be clear from the context.

Let $S$ be a sequence $s_1, \ldots, s_k$ of edit operations. An $S$-derivation from $A$ to $B$
is a sequence of trees $A_0, \ldots, A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \to A_i$ via $s_i$
for $1 \leq i \leq k$.

Let $\gamma$ be a cost function that assigns to each edit operation $a \to b$ a nonnegative
real number $\gamma(a \to b)$. This cost can be different for different nodes, so it can be used
to give greater weights to, for example, the higher nodes in a tree than to lower nodes.

We constrain $\gamma$ to be a distance metric. That is,

(i) $\gamma(a \to b) \geq 0$; $\gamma(a \to a) = 0$
(ii) $\gamma(a \to b) = \gamma(b \to a)$; and
(iii) $\gamma(a \to c) \leq \gamma(a \to b) + \gamma(b \to c)$.

We extend $\gamma$ to the sequence $S$ by letting $\gamma(S) = \sum_{i=1}^{i=|S|} \gamma(s_i)$. Formally the distance
between $T_1$ and $T_2$ is defined as follows:

$\delta(T_1, T_2) = \min \{ \gamma(S) | S \text{ is an edit operation sequence taking } T_1 \text{ to } T_2 \}$. 

The definition of $\gamma$ makes $\delta$ a distance metric also.

2.2. Mapping. Let $T_1$ and $T_2$ be two trees with $N_1$ and $N_2$ nodes, respectively.
Suppose that we have an ordering for each tree, then $T[i]$ means the $i$th node of tree
$T$ in the given ordering.

The edit operations give rise to a mapping that is a graphical specification of what
edit operations apply to each node in the two trees (or two ordered forests). The mapping in Fig. 4 shows a way to transform $T_1$ to $T_2$. It corresponds to the sequence
delete (node with label $c$), insert (node with label $c$)).

![Fig. 4](image-url)
Consider the diagram of a mapping in Fig. 4. A dotted line from $T_1[i]$ to $T_2[j]$ indicates that $T_1[i]$ should be changed to $T_2[j]$ if $T_1[i] \neq T_2[j]$, or that $T_1[i]$ remains unchanged if $T_1[i] = T_2[j]$. The nodes of $T_1$ not touched by a dotted line are to be deleted and the nodes of $T_2$ not touched are to be inserted. The mapping shows a way to transform $T_1$ to $T_2$.

Formally we define a triple $(M, T_1, T_2)$ to be a mapping from $T_1$ to $T_2$, where $M$ is any set of pair of integers $(i, j)$ satisfying:

1. $1 \leq i \leq N_1$, $1 \leq j \leq N_2$;
2. For any pair of $(i_1, j_1)$ and $(i_2, j_2)$ in $M$,
   a. $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one),
   b. $T_1[i_1]$ is to the left of $T_1[i_2]$ if and only if $T_2[j_1]$ is to the left of $T_2[j_2]$ (sibling order preserved),
   c. $T_1[i_1]$ is an ancestor of $T_1[i_2]$ if and only if $T_2[j_1]$ is an ancestor of $T_2[j_2]$ (ancestor order preserved).

We will use $M$ instead of $(M, T_1, T_2)$ if there is no confusion. Let $M$ be a mapping from $T_1$ to $T_2$. Let $I$ and $J$ be the sets of nodes in $T_1$ and $T_2$, respectively, not touched by any line in $M$. Then we can define the cost of $M$:

$$\gamma(M) = \sum_{(i,j) \in M} \gamma(T_1[i] \rightarrow T_2[j]) + \sum_{i \in I} \gamma(T_1[i] \rightarrow \Lambda) + \sum_{j \in J} \gamma(\Lambda \rightarrow T_2[j]).$$

Mappings can be composed. Let $M_1$ be a mapping from $T_1$ to $T_2$ and let $M_2$ be a mapping from $T_2$ to $T_3$. Define

$$M_1 \circ M_2 = \{(i, j) \mid \exists k \text{ s.t. } (i, k) \in M_1 \text{ and } (k, j) \in M_2\}.$$

Lemma 1. (1) $M_1 \circ M_2$ is a mapping.
(2) $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2)$.

Proof. Case (1) follows from the definition of mapping.
(2) Let $M_1$ be the mapping from $T_1$ to $T_2$. Let $M_2$ be the mapping from $T_2$ to $T_3$. Let $M_1 \circ M_2$ be the composed mapping from $T_1$ to $T_3$ and let $I$ and $J$ be the corresponding deletion and insertion sets. Three general situations occur. $(i, j) \in M_1 \circ M_2$, $i \in I$, or $j \in J$. In each case this corresponds to an editing operation $\gamma(x \rightarrow y)$ where $x$ and $y$ may be nodes or may be $\Lambda$. In all such cases, the triangle inequality on the distance metric $\gamma$ ensures that $\gamma(x \rightarrow y) \leq \gamma(x \rightarrow z) + \gamma(z \rightarrow y)$. □

The relation between a mapping and a sequence of edit operation is as follows.

Lemma 2. Given $S$, a sequence $s_1, \cdots, s_k$ of edit operations from $T_1$ to $T_2$, there exists a mapping $M$ from $T_1$ to $T_2$ such that $\gamma(M) \leq \gamma(S)$. Conversely, for any mapping $M$, there exists a sequence of editing operations such that $\gamma(S) = \gamma(M)$.

Proof. The first part can be proved by induction on $k$. The base case is $k = 1$. This case holds, because any single editing operation preserves the ancestor and sibling relationships in the mapping. In the general case, let $S_1$ be the sequence $s_1, \cdots, s_{k-1}$ of edit operations. There exist a mapping $M_1$ such that $\gamma(M_1) \leq \gamma(S_1)$. Let $M_2$ be the mapping for $s_k$. From Lemma 1, we have that

$$\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2) \leq \gamma(S).$$

To construct the sequence of editing operations, simply perform all the deletes indicated by the mapping (i.e., all nodes in $T_1$ having no lines attached to them are deleted), then all relabellings, then all inserts. □

\footnote{Note that our definition of mapping is different from the definition in [T]. We believe that our definition is more natural because it does not depend on any traversal ordering of the tree.}
Hence, $\delta(T_1, T_2) = \min \{\gamma(M)|M\text{ is a mapping from } T_1 \text{ to } T_2\}$.

There has been previous work on this problem. Tai [T] gave the best published algorithm for the problem. [Z83] is an improvement of [T], giving better sequential time and space than [T]. Our new algorithm is much simpler than [T] and [Z83], gives better time and space than both of them, and extends to related problems. The algorithm of Lu [L] does not solve this problem for trees of more than two levels.

3. A simple new algorithm. This algorithm, unlike [T], [L], and [Z83], will, in its intermediate steps, consider the distance between two ordered forests. At first sight one may think that this will complicate the work, but it will in fact make matters easier.

We use a postorder numbering of the nodes in the trees. In the postordering, $T_1[1..i]$ and $T_2[1..j]$ will generally be forests as in Fig. 5. (The edges are those in the subgraph of the tree induced by the vertices.) Fortunately, the definition of mapping for ordered forests is the same as for trees.

3.1. Notation. Let $T[i]$ be the $i$th node in the tree according to the left-to-right postorder numbering. $l(i)$ is the number of the leftmost leaf descendant of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $l(i) = i$. The parent of $T[i]$ is denoted $p(i)$. We define $p^0(i) = i$, $p^1(i) = p(i)$, $p^2(i) = p(p(i))$, and so on. Let $\text{anc}(i) = \{p^k(i) | 0 \leq k \leq \text{depth}(i)\}$.

$T[i..j]$ is the ordered subforest of $T$ induced by the nodes numbered $i$ to $j$ inclusive (Fig. 5). If $i > j$, then $T[i..j] = \emptyset$. $T[1..i]$ will be referred to as forest$(i)$, when the tree $T$ referred to is clear. $T[l(i)..i]$ will be referred to as tree$(i)$. Size$(i)$ is the number of nodes in tree$(i)$.

The distance between $T[i'..i]$ and $T[j'..j]$ is denoted forestdist$(T[i'..i], T[j'..j])$ or forestdist$(i'..i, j'..j)$ if the context is clear. We use a more abbreviated notation for certain special cases. The distance between $T[1..i]$ and $T[1..j]$ is sometimes denoted forestdist$(i, j)$. The distance between the subtree rooted at $i$ and the subtree rooted at $j$ is sometimes denoted treedist$(i, j)$.

3.2. New algorithm. We first present three lemmas and then give our new algorithm.

Recall that $\text{anc}(i) = \{p^k(i) | 0 \leq k \leq \text{depth}(i)\}$.

Lemma 3. (i) forestdist$(\emptyset, \emptyset) = 0$. 

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

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

where \( i_1 \in \text{anc}(i) \) and \( j_1 \in \text{anc}(j) \).

**Proof.** Case (i) requires no edit operation. In (ii) and (iii), the distances correspond to the cost of deleting or inserting the nodes in \( T_i[l(i_1) \ldots i] \) and \( T_2[l(j_1) \ldots j] \), respectively.

**Lemma 4.** Let \( i_1 \in \text{anc}(i) \) and \( j_1 \in \text{anc}(j) \). Then

\[
\text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) = \min \left\{ \begin{array}{l}
\text{forestdist}(l(i_1) \ldots i-1, l(j_1) \ldots j) + \gamma(T_i[i] \rightarrow \Lambda), \\
\text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j-1) + \gamma(\Lambda \rightarrow T_2[j]), \\
\text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) + \gamma(T_i[i] \rightarrow T_2[j]), \\
\end{array} \right.
\]

Proof. We compute \( \text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) \) for \( l(i_1) \leq i \leq i_1 \) and \( l(j_1) \leq j \leq j_1 \). We are trying to find a minimum-cost map \( M \) between \( \text{forest}(l(i_1) \ldots i) \) and \( \text{forest}(l(j_1) \ldots j) \). The map can be extended to \( T_i[i] \) and \( T_2[j] \) in three ways.

1. \( T_i[i] \) is not touched by a line in \( M \). Then \( (i, \Lambda) \in M \). So, \( \text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) = \text{forestdist}(l(i_1) \ldots i-1, l(j_1) \ldots j) + \gamma(T_i[i] \rightarrow \Lambda) \).

2. \( T_2[j] \) is not touched by a line in \( M \). Then \( (\Lambda, j) \in M \). So, \( \text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) = \text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j-1) + \gamma(\Lambda \rightarrow T_2[j]) \).

3. \( T_i[i] \) and \( T_2[j] \) are both touched by lines in \( M \). Then \( (i, j) \in M \). Here is why. Suppose \( (i, k) \) and \( (h, j) \) are in \( M \). If \( l(i_1) \leq h \leq l(i)-1 \), then \( i \) is to the right of \( h \) so \( k \) must be to the right of \( j \) by the sibling condition on mappings. This is impossible in \( \text{forest}(l(j_1) \ldots j) \). Similarly, if \( i \) is a proper ancestor of \( h \), then \( k \) must be a proper ancestor of \( j \) by the ancestor condition on mappings. This too is impossible. So, \( h = i \).

By symmetry, \( k = j \) and \( (i, j) \in M \).

Now, by the ancestor condition on mapping, any node in the subtree rooted at \( T_i[i] \) can only be touched by a node in the subtree rooted at \( T_2[j] \). Hence,

\[
\text{forestdist}(l(i_1) \ldots i, l(j_1) \ldots j) = \text{forestdist}(l(i_1) \ldots i-1, l(j_1) \ldots j-1) + \gamma(T_i[i] \rightarrow T_2[j]).
\]

Figure 6 shows the situation.

---

**Fig. 6.** Case (3) of Lemma 4.
Since these three cases express all the possible mappings yielding forest-dist(l(iₙ)...i, l(jₙ)...j), we take the minimum of these three costs. Thus,

\[
\text{forest-dist}(l(i₁)...i, l(j₁)...j) = \min \left\{ \begin{array}{c}
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j) + \gamma(T₁[i] \rightarrow \Lambda), \\
\text{forest-dist}(l(i₁)...i, l(j₁)...j-1) + \gamma(\Lambda \rightarrow T₂[j]), \\
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j-1) + \gamma(T₁[i] \rightarrow T₂[j]).
\end{array} \right.
\]

**Lemma 5.** Let \( i₁ \in \text{anc}(i) \) and \( j₁ \in \text{anc}(j) \). Then

(1) If \( l(i) = l(i₁) \) and \( l(j) = l(j₁) \)

\[
\text{forest-dist}(l(i₁)...i, l(j₁)...j) = \min \left\{ \begin{array}{c}
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j) + \gamma(T₁[i] \rightarrow \Lambda), \\
\text{forest-dist}(l(i₁)...i, l(j₁)...j-1) + \gamma(\Lambda \rightarrow T₂[j]), \\
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j-1) + \gamma(T₁[i] \rightarrow T₂[j]).
\end{array} \right.
\]

(2) If \( l(i) \neq l(i₁) \) or \( l(j) \neq l(j₁) \) (i.e., otherwise)

\[
\text{forest-dist}(l(i₁)...i, l(j₁)...j) = \min \left\{ \begin{array}{c}
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j) + \gamma(T₁[i] \rightarrow \Lambda), \\
\text{forest-dist}(l(i₁)...i, l(j₁)...j-1) + \gamma(\Lambda \rightarrow T₂[j]), \\
\text{forest-dist}(l(i₁)...i-1, l(j₁)...j-1) + \gamma(T₁[i] \rightarrow T₂[j]), \\
\text{treedist}(i, j).
\end{array} \right.
\]

**Proof.** By Lemma 4, if \( l(i) = l(i₁) \) and \( l(j) = l(j₁) \) then, since forest-dist(l(i₁)...i, l(j₁)...j) = forest-dist(∅, ∅) = 0, (1) follows immediately.

Because the distance is the cost of a minimal cost mapping, we know forest-dist(l(i₁)...i, l(j₁)...j) ≤ forest-dist(l(i₁)...i-1, l(j₁)...j-1) + treedist(i, j) since the latter formula represents a particular (and therefore possibly suboptimal) mapping of forest(l(i₁)...i) to forest(l(j₁)...j). For the same reason, treedist(i, j) ≤ forest-dist(l(i₁)...i-1, l(j₁)...j-1) + γ(T₁[i] → T₂[j]). Lemma 4 and these two inequalities imply that the substituting of treedist(i, j) for forest-dist(l(i₁)...i-1, l(j₁)...j-1) in (2) is correct. (See Fig. 7.)

**Lemma 5** has three important implications:

- First, the formulas it yields suggest that we can use a dynamic programming style algorithm to solve the tree distance problem.

- Second, from (2) of Lemma 5 we observe that to compute treedist(i₁, j₁) we need in advance almost all values of treedist(i, j) where i is the root of a subtree containing i₁ and j₁ is the root of a subtree containing j. This suggests a bottom-up procedure for computing all subtree pairs.

- Third, from (1) in Lemma 5 we can observe that when i₁ is in the path from l(i₁) to i, and j₁ is in the path from l(j₁) to j, we do not need to compute treedist(i, j) separately. These subtree distances can be obtained as a byproduct of computing treedist(i₁, j₁).

These implications lead to the following definition and then our new algorithm. Let us define the set \( LR\text{-keyroots} \) of tree \( T \) as follows:

\[ LR\text{-keyroots}(T) = \{ k \mid \text{there exists no } k' > k \text{ such that } l(k) = l(k') \}. \]

That is, if \( k \) is in \( LR\text{-keyroots}(T) \) then either \( k \) is the root of \( T \) or \( l(k) \neq l(p(k)) \), i.e., \( k \) has a left sibling. Intuitively, this set will be the roots of all the subtrees of tree \( T \) that need separate computations.

Consider trees \( T₁ \) and \( T₂ \) in Fig. 4. From the above definition we can see that \( LR\text{-keyroots}(T₁) = \{ 3, 5, 6 \} \) and \( LR\text{-keyroots}(T₂) = \{ 2, 5, 6 \} \).
It is easy to see that there is a linear time algorithm to compute the function \( l(\quad) \) and the set \( LR_{\text{keyroots}} \). We can also assume that the result is in array \( l \) and \( LR_{\text{keyroots}} \). Furthermore, in array \( LR_{\text{keyroots}} \) the order of the elements is in increasing order.

We are now ready to give our new simple algorithm.

**Input:** Tree \( T_1 \) and \( T_2 \).

**Output:** \( Tree_{\text{dist}}(i, j) \), where \( 1 \leq i \leq |T_1| \) and \( 1 \leq j \leq |T_2| \).

**Preprocessing**

(To compute \( l(\quad) \), \( LR_{\text{keyroots}1} \) and \( LR_{\text{keyroots}2} \))

**Main loop**

for \( i' := 1 \) to \( |LR_{\text{keyroots}}(T_1)| \)

for \( j' := 1 \) to \( |LR_{\text{keyroots}}(T_2)| \)

\[ i = LR_{\text{keyroots}1}[i']; \]

\[ j = LR_{\text{keyroots}2}[j']; \]

Compute \( treedist(i, j) \).

We use dynamic programming to compute \( treedist(i, j) \). The forestdist values computed and used here are put in a temporary array that is freed once the corresponding treedist is computed. The treedist values are put in the permanent treedist array.
\( \text{forestdist}(\emptyset, \emptyset) = 0; \)

for \( i := l(i) \) to \( i \)

\( \text{forestdist}(T_i[l(i) \ldots i], \emptyset) = \text{forestdist}(T_i[l(i) \ldots i - 1], \emptyset) + \gamma(T_i[i] \rightarrow \Lambda) \)

for \( j := l(j) \) to \( j \)

\( \text{forestdist}(\emptyset, T_j[l(j) \ldots j], \emptyset) = \text{forestdist}(\emptyset, T_j[l(j) \ldots j - 1], \emptyset) + \gamma(\Lambda \rightarrow T_j[j]) \)

for \( i := l(i) \) to \( i \)

for \( j := l(j) \) to \( j \)

if \( l(i) = l(i) \) and \( l(j) = l(j) \)

\( \text{forestdist}(T_i[l(i) \ldots i], T_j[l(j) \ldots j]) = \min \{ \)

\( \text{forestdist}(T_i[l(i) \ldots i - 1], T_j[l(j) \ldots j]) + \gamma(T_i[i] \rightarrow \Lambda), \)

\( \text{forestdist}(T_i[l(i) \ldots i], T_j[l(j) \ldots j - 1]) + \gamma(\Lambda \rightarrow T_j[j]), \)

\( \text{forestdist}(T_i[l(i) \ldots i - 1], T_j[l(j) \ldots j - 1]) + \gamma(T_i[i] \rightarrow T_j[j]) \} \)

\( \text{treedist}(i, j) = \text{forestdist}(T_i[l(i) \ldots i], T_j[l(j) \ldots j]) / \) put in permanent array */

else

\( \text{forestdist}(T_i[l(i) \ldots i], T_j[l(j) \ldots j]) = \min \{ \)

\( \text{forestdist}(T_i[l(i) \ldots i - 1], T_j[l(j) \ldots j]) + \gamma(T_i[i] \rightarrow \Lambda), \)

\( \text{forestdist}(T_i[l(i) \ldots i], T_j[l(j) \ldots j - 1]) + \gamma(\Lambda \rightarrow T_j[j]), \)

\( \text{forestdist}(T_i[l(i) \ldots i - 1], T_j[l(j) \ldots j - 1]) + \text{treedist}(i, j) \} \)

\textbf{Theorem 1.} \textit{The basic algorithm is correct.}

\textit{Proof.} We will prove that for any pair \( (i, j) \) such that \( i \in LR\_\text{keyroots}(T_i) \) and \( j \in LR\_\text{keyroots}(T_j) \), the following invariants holds.

\begin{align*}
\text{tree\_dist}(3, 2) & \quad \text{tree\_dist}(3, 5) & \quad \text{tree\_dist}(3, 6) \\
0 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 1 & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 5 \\
2 & 2 & 1 & 2 & 2 & 2 & 2 & 4 & 5 & 6 \\
\text{tree\_dist}(5, 2) & \quad \text{tree\_dist}(5, 5) & \quad \text{tree\_dist}(5, 6) \\
0 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 1 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\text{tree\_dist}(6, 2) & \quad \text{tree\_dist}(6, 5) & \quad \text{tree\_dist}(6, 6) \\
0 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 1 & 1 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\
2 & 2 & 2 & 2 & 1 & 0 & 1 & 2 & 3 & 4 \\
3 & 3 & 3 & 3 & 2 & 1 & 2 & 3 & 4 & 5 \\
4 & 4 & 4 & 4 & 3 & 2 & 1 & 2 & 3 & 4 \\
5 & 4 & 5 & 4 & 5 & 4 & 3 & 2 & 3 & 2 \\
6 & 5 & 6 & 5 & 6 & 5 & 4 & 3 & 3 & 2 \\
\end{align*}

\textit{FIG. 8. The result of computation for } \( T_1 \text{ and } T_2 \) \textit{in Fig. 4.}
(1) Immediately before the computation of $\text{treedist}(i, j)$, all distances $\text{treedist}(i_l, j_l)$, where $l(i) \leq i \leq i$ and $l(j) \leq j \leq j$ and either $l(i) \neq l(i_l)$ or $l(j) \neq l(j_l)$, are available. In other words, $\text{treedist}(i, j)$ is available if $i$ is in the subtree of $\text{tree}(i)$ but not in the path from $l(i)$ to $i$ and $j$ is in the subtree of $\text{tree}(j)$ but not in the path from $l(j)$ to $j$.

(2) Immediately after the computation of $\text{treedist}(i, j)$, all distances $\text{treedist}(i, j_l)$, where $l(i) < i < i$ and $l(j) < j < j$ are available.

We first show that if (1) is true then (2) is true. From Lemma 5 we know that all required subtree-to-subtree distances are available. (We need all $\text{treedist}(i_l, j_l)$ such that $l(i) < i < i$ and $l(j) < j < j$ and either $l(i) \neq l(i_l)$ or $l(j) \neq l(j_l)$, and by (1) all these distances are available.) We compute each $\text{treedist}(i_l, j_l)$, where $l(i_l) = l(i)$ and $l(j_l) = l(j)$ in the if part and add it to the permanent treedist array. So, (2) holds.

Let us show that (1) always holds. Suppose $l(i_l) \neq l(i)$. Let $i_l'$ be the lowest ancestor of $i_l$ such that $i_l' \in \text{LR_keyroots}(T_1)$. Since $l(i_l') = l(i_l)$, $i_l' < i$. So $i_l' < i$. Let $j_l'$ be the lowest ancestor of $j_l$ such that $j_l' \in \text{LR_keyroots}(T_2)$. Since $j \in \text{LR_keyroots}(T_2)$, $j < j$. Hence $i_l' + j_l' < i + j$. This means that $\text{treedist}(i_l', j_l')$ will have already been computed before $\text{treedist}(i_l, j_l)$ because in the main loop $\text{LR_keyroots}(T_1)$ and $\text{LR_keyroots}(T_2)$ are in increasing order. Hence $\text{treedist}(i_l, j_l)$ is available after the computation of $\text{treedist}(i_l', j_l')$. □

As an example, consider tree $T_1$ and $T_2$ in Fig. 4. For simplicity, assume that all insert, delete, and change (of labels) operations will cost one. Figure 8 shows the result of applying our new algorithm to $T_1$ and $T_2$. The matrix below $\text{tree_dist}(i, j)$ is the result of temporary array produced by the computation of $\text{tree_dist}(i, j)$. (Out of 36 possible $\text{tree_dist}$ arrays, only nine—those corresponding to pairs of keyroots—are explicitly computed.) The matrix below $\text{treedist}$ is the final result. The value in the lower right corner (2) is the distance between $T_1$ and $T_2$.

4. Some aspects of our algorithm.


**Lemma 6.** $|\text{LR_keyroots}(T)| \leq |\text{leaves}(T)|$.

**Proof.** We will prove that for any $i, j \in \text{LR_keyroots}(T)$, $l(i) \neq l(j)$.

Let $i, j \in \text{LR_keyroots}(T)$ and $i < j$. If $l(i) = l(j)$ from $i < j$ we know that $i$ is in the path from $l(j)$ to $j$. By the definition of $l(j)$, $i$ has no left_sibling. This contradicts the assertion that $i \in \text{LR_keyroots}(T)$. Hence each leaf is the leftmost descendant at at most one member of $\text{LR_keyroots}(T)$. So, $|\text{LR_keyroots}(T)| \leq |\text{leaves}(T)|$. □

Because not all subtree-to-subtree distances need be computed, the number of such calculation a node participates in is less than its depth. Instead, it is the node's **collapsed depth**:

$$\text{LR_collapse}(i) = |\text{anc}(i) \cap \text{LR_keyroots}(T)|.$$  

We define the collapsed depth of tree $T$ as follows:

$$\text{LR_collapse}(T) = \max \text{LR_collapse}(i).$$

By the definition and Lemma 6 we can see that $\text{LR_collapse}(i) \leq \min (\text{depth}(T), \text{leaves}(T))$ for $1 \leq i \leq |T|$. Hence $\text{LR_collapse}(T) \leq \min (\text{depth}(T), \text{leaves}(T))$.

**Lemma 7.**

$$\sum_{i=1}^{i=|\text{LR_keyroots}(T)|} \text{Size}(i) = \sum_{j=1}^{j=N_1} |\text{LR_collapse}(j)|.$$
Proof. Consider when node \( j \) is counted in the first summation: in the subtrees corresponding to each of its ancestors that is in \( LR\_keyroots(T) \). By the definition of \( LR\_colldepth() \), \( j \) is counted \( LR\_colldepth(j) \) times.

**Theorem 2.** The time complexity is \( O(|T_1| \times |T_2| \times \min(\text{depth}(T_1), \text{leaves}(T_1)) \times \min(\text{depth}(T_2), \text{leaves}(T_2))) \). The space complexity is \( O(|T_1| \times |T_2|) \).

Proof. Let us consider the space complexity first. We use a permanent array for \( treedist \) and a temporary array for \( forestdist \). Each of these two arrays requires space \( O(|T_1| \times |T_2|) \).

Consider the time complexity of our algorithm. The preprocessing takes linear time. The subtree distance dynamic programming algorithm takes \( \text{Size}(i) \times \text{Size}(j) \) for the subtree rooted at \( T_1[i] \) and the subtree rooted at \( T_2[j] \). We have a main loop that calls this subroutine several times. So the time is:

\[
\sum_{i=1}^{\text{LR}_\text{keyroots}(|T_1|)} \sum_{j=1}^{\text{LR}_\text{keyroots}(|T_2|)} \text{Size}(i) \times \text{Size}(j)
\]

By Lemma 6, the above equals

\[
\sum_{i=1}^{N_1} \text{LR}_\text{colldepth}(i) \times \sum_{j=1}^{N_2} \text{LR}_\text{colldepth}(j).
\]

This is less than

\[
|T_1| \times |T_2| \times \text{LR}_\text{colldepth}(T_1) \times \text{LR}_\text{colldepth}(T_2).
\]

By the definition of \( \text{LR}_\text{colldepth} \), we have that the time complexity is

\[
O(|T_1| \times |T_2| \times \min(\text{depth}(T_1), \text{leaves}(T_1)) \times \min(\text{depth}(T_2), \text{leaves}(T_2))). \quad \Box
\]

These time and space complexities are an improvement over the \( O(|T_1| \times |T_2| \times \text{depth}(T_1)^2 \times \text{depth}(T_2)^2) \) time and space complexity of [T].

Note. If we use a right-to-left postorder numbering for tree nodes and define similar functions \( r(i), RL\_\text{keyroots}(T) \) and \( RL\_\text{colldepth}(i) \), we can have the same result as above. The complexity will be \( \sum_{i=1}^{N_1} \text{LR}_\text{colldepth}(i) \times \sum_{j=1}^{N_2} \text{LR}_\text{colldepth}(j) \).

Clearly, using the left-to-right or right-to-left postorder numberings give the same worst-case time complexity. However, in practice it may be beneficial to choose the ordering that gives the lower of the following two products: \( \sum_{i=1}^{N_1} \text{LR}_\text{colldepth}(i) \times \sum_{j=1}^{N_2} \text{LR}_\text{colldepth}(j) \) and \( \sum_{i=1}^{N_1} \text{RL}_\text{colldepth}(i) \times \sum_{j=1}^{N_2} \text{RL}_\text{colldepth}(j) \).

4.2. Mapping. It is natural to ask for a mapping that yields the distance computed. Also given two trees, we may ask, what is the largest common substructure of these two trees? This is analogous to the longest common substring problem for strings. We can find the mapping in the same time and space complexity as finding the distance, although we do not give the details here. The mapping is produced by our toolkit.

4.3. Parallel implementation. A straightforward transformation of our algorithm to a parallel one yields an algorithm with time complexity \( O(N_1 + N_2) \) whereas [T] and [Z83] have time complexity \( O((N_1 + N_2) \times (\text{depth}(T_1) + \text{depth}(T_2))) \). Our algorithm uses \( O(\min(|T_1|, |T_2|) \times \text{leaves}(T_1) \times \text{leaves}(T_2)) \) processors.\(^2\)

\(^2\) Actually, by controlling the starting point of each \( treedist \) computation more carefully, we can reduce the processor bound to \( O(\min(|T_1|, |T_2|) \times \min(\text{depth}(T_1), \text{leaves}(T_1)) \times \min(\text{depth}(T_2), \text{leaves}(T_2))) \). The algorithm is more complicated however.
The algorithm computes in “waves” for all subtree pairs tree(i) and tree(j), where \(i \in LR\_keyroots(T_1)\) and \(j \in LR\_keyroots(T_2)\), simultaneously. We start at wave 0. At wave \(k\), for each such subtree pair tree(i) and tree(j), compute forest-dist(l(i) \(i\) l, l(j) \(j\)), where \((i - l(i)) + (j - l(j)) = k\).

We now present the parallel algorithm in detail. (When the PARBEGIN-PAREND construct surrounds one or more for loops, it means that every setting of the iterators in the enclosed for loops can be executed in parallel. The semantics are those of the sequential program ignoring this construct.)

In the algorithm \(dist[i, j]\) is the array for the computation of treedist(i, j). Therefore \(dist[i, j][p, q]\) is the distance forestdist(l(i) \(l\), l(j) \(q\)) and is the \(p, q\)th member of the array computing treedist(i, j).

**Algorithm Parallel Distance.**

begin
PARBEGIN
for \(i' := 1\) to \(|LR\_keyroots(T_1)|\)
for \(j' := 1\) to \(|LR\_keyroots(T_2)|\)
\(i := LR\_keyroots1[i']\)
\(j := LR\_keyroots2[j']\)
\(dist[i, j][l(i) - 1, l(j) - 1] := 0\) /* initializes temporary array for each tree dist */
PAREND
for \(k := 0\) to \(N - 1\)
PARBEGIN
for \(i' := 1\) to \(|LR\_keyroots(T_1)|\)
for \(j' := 1\) to \(|LR\_keyroots(T_2)|\)
\(i := LR\_keyroots1[i']\)
\(j := LR\_keyroots2[j']\)
\(dist[i, j][l(i) + k, l(j) - 1] := dist[i, j][l(i) + k - 1, l(j) - 1] + \gamma(T_1[l[i] + k] \rightarrow \Lambda)\)
PAREND
for \(k := 0\) to \(M - 1\)
PARBEGIN
for \(i' := 1\) to \(|LR\_keyroots(T_1)|\)
for \(j' := 1\) to \(|LR\_keyroots(T_2)|\)
\(i := LR\_keyroots1[i']\)
\(j := LR\_keyroots2[j']\)
\(dist[i, j][l(i) - 1, l(j) + k] := dist[i, j][l(i) - 1, l(j) + k - 1] + \gamma(\Lambda \rightarrow T_1[l[j] + k])\)
PAREND
for \(k := 0\) to \(N + M - 2\)
PARBEGIN
for \(i' := 1\) to \(|LR\_keyroots(T_1)|\)
for \(j' := 1\) to \(|LR\_keyroots(T_2)|\)
\(i := LR\_keyroots1[i']\)
\(j := LR\_keyroots2[j']\)
for \(i, j, i'l, j'l\) satisfying \(i'l \leq l(i) + j'l - l(j) = k\) and \(l(i) \leq i \leq i'l \leq j \leq j'l \leq j\)
if \(l(i) = l(i')\) and \(l(j) = l(j')\) then
\(dist[i, j][i', j'] := \min\{\)
\(dist[i, j][i - 1, j] + \gamma(T_1[i] \rightarrow \Lambda)\)
\(dist[i, j][i, j + 1] + \gamma(T_1[1] \rightarrow \Lambda)\)
\(dist[i, j][i, j] + \gamma(T_1[i] \rightarrow \Lambda)\)
\}
$$\text{dist}(i,j)[i_1,j_1] + \gamma(\Lambda \rightarrow T_2[j_1])$$

$$\text{dist}(i,j)[i_1-1,j_1-1] + \gamma(T_1[i_1] \rightarrow T_2[j_1])$$

$$\text{treedist}(i,j) := \text{dist}(i,j)[i_1,j_1]$$

else

$$\text{dist}(i,j)[i_1,j_1] := \min \{$$

$$\text{dist}(i,j)[i_1-1,j_1] + \gamma(T_1[i_1] \rightarrow \Lambda)$$

$$\text{dist}(i,j)[i_1,j_1-1] + \gamma(\Lambda \rightarrow T_2[j_1])$$

$$\text{dist}(i,j)[l(i_1)-1,l(j_1)-1] + \text{treedist}(i,j_1)$$

$$\}$$

end

It is easy to see that in the above algorithm all the terms, except $\text{treedist}(i_1, j_1)$, are available whenever needed. We now show that $\text{treedist}(i_1, j_1)$ is available whenever we use it. Our argument is similar to the one we used in the sequential case.

Note that we compute all terms such that $(i_1 - l(i_1)) + (j_1 - l(j_1)) = k$ together. During that computation, all terms such that $(i_1 - l(i_1)) + (j_1 - l(j_1)) < k$ are available. So, when we need item $\text{treedist}(i_1, j_1)$, either $l(i_1) > l(i_1)$ or $l(j_1) > l(j_1)$. Let $i_2$ be the lowest ancestor of $i_1$ such that $i_2 \in LR_keyroots(T_1)$. Let $j_2$ be the lowest ancestor of $j_1$ such that $j_2 \in LR_keyroots(T_2)$. Since $l(i_1) = l(i_2)$ and $l(j_1) = l(j_2)$ we know either $l(i_2) > l(i)$ or $l(j_2) > l(j)$. Therefore, $(i_1 - l(i_2)) + (j_1 - l(j_2)) < (i_1 - l(i)) + (j_1 - l(j)) = k$. Hence $\text{treedist}(i_1, j_1)$ was already computed in the computation of $\text{dist}(i_1, j_2)$ and put into the permanent tree distance array. This settles correctness.

**Theorem 3.** The Parallel Distance Algorithm has time complexity $O(|T_1| + |T_2|)$.

**Proof.** By simple analysis of the for loop. □

### 4.4. From trees to strings

Strings are an important special case of trees. This algorithm is a generalization of the natural dynamic programming algorithms on strings in two senses: time complexity and algorithmic style.

First, we consider the time complexity. Since a string has only one leaf, applying our algorithms to strings yields a time complexity of $O(|T_1| \times |T_2|)$. This is the same as that of the best available algorithm for the general problem of string distance.

Second, we consider the algorithm itself. For a string $S$, $LR_keyroots(S) = \{\text{root}\}$. So the main loop will only have one iteration. In the dynamic programming subroutine, since $l(i) = 1$, we will never come to the case $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$. So if we change $i$ to $|T_1|$, $j$ to $|T_2|$, $l(i)$ to one, $l(j)$ to one, delete the main loop and delete the case where $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, we will have exactly the string distance algorithm.

### 5. The general technique applied to approximate tree matching

Many problems in strings can be solved with dynamic programming. Similarly, our algorithm not only applies to tree distance but also provides a way to do dynamic programming for a variety of tree problems with the same time complexity. In this section we show how to apply this general paradigm to approximate tree matching.

### 5.1. Algorithm template

Here is the general form of the algorithm (assuming a left-to-right postorder traversal):

**preprocessing**

**main loop**

for $i' := 1$ to $|LR_keyroots(T_i)|$

for $j' := 1$ to $|LR_keyroots(T_j)|$

$i = LR_keyroots1[i']$

$j = LR_keyroots2[j']$
compute $Tree_D(i, j)$;

subroutine for $Tree_D(i, j)$

empty_initialization

for $i_1 := l(i)$ to $i$

left_initialization

for $j_1 := l(j)$ to $j$

right_initialization

for $i_1 := l(i)$ to $i$

for $j_1 := l(j)$ to $j$

if $l(i_1) = l(i)$ and $l(j_1) = l(j)$ then

general_if_computation

$Tree_D(i_1, j_1) = Forest_D(T_1[l(i) \ldots i_1], T_2[l(j) \ldots j_1]);$

else

general_else_computation

5.2. Approximate tree matching. We first consider approximate string matching [S80], [U83], [U85], [LV]. We will then give two natural generalizations of approximate string matching to approximate tree matching. This will also be a generalization of the exact tree matching algorithm as found in Hoffmann and O'Donnell [HO].

The approximate string matching problem is the following. Given two strings $STEXT$ and $SPAT$, the problem is to compute, for each $i$, $SD[i, SPAT] = \min_j \{D(STEXT[j..i], SPAT)\}$, where $1 \leq j \leq i+1$ and $D$ is the string distance metric. In other words, the problem is to compute, for each $i$, the minimum number of editing operations between the “pattern” string $SPAT[1..|PAT|]$ and the “text string” $STEXT[1..i]$ where any prefix can be removed from $STEXT[1..i]$. (Intuitively, the algorithm finds the “occurrence” in $TEXT$ that most closely matches $PAT$.)

To extend this problem to trees, we must generalize the notion of removing a prefix. For us, a prefix will mean a collection of subtrees.

We first define two operations at a node.

Removing at node $T[i]$ means removing the subtree rooted at $T[i]$. In other words, delete $T[l(i) \ldots i]$. (See Fig. 9.)

Pruning at node $T[i]$ means removing all the descendants of $T[i]$. In other words, delete $T[l(i) \ldots i - 1]$. (Thus, a pruning never eliminates the entire tree.) (See Fig. 10.)

![Fig. 9. Remove subtree rooted at T[8].](image-url)
Assume an ordering for tree $T$. Define a subtree set $S(T)$ as follows: $S(T)$ is a set of numbers satisfying

1. $i \in S(T)$ implies that $1 \leq i \leq |T|
2. i, j \in S(T)$ implies that neither is an ancestor of the other.

Define $R(T, S(T))$ to be the tree $T$ with removing at all nodes in $S(T)$. Define $P(T, S(T))$ to be the tree $T$ with pruning at all nodes in $S(T)$.

Now we can give the definition of approximate tree matching. Given tree $T$ and $PAT$, for each $i$, we want to calculate

$$DR(T[l(i)\ldots i, PAT) = \min_{S} \{treedist(R(T[l(i)\ldots i]), S(T[l(i)\ldots i])), PAT\}).$$

$$DP(T[l(i)\ldots i, PAT) = \min_{S} \{treedist(P(T[l(i)\ldots i]), S(T[l(i)\ldots i])), PAT\}).$$

The minimum here is over all possible subtree sets $S(T[l(i)\ldots i])$. We consider each generalization in turn.

5.2.1. Remove any number of subtrees from TEXT tree. The problem is as follows. Given trees $T_1$ and $T_2$, we want to know what is the minimum distance between $T_1[l(i)\ldots i]$ and $T_2$ when zero or more subtrees can be removed from $T_1[l(i)\ldots i]$.

Let $F_{-DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_i])$ denote the minimum distance between forest $T_1[l(i)\ldots i]$ and $T_2[l(j)\ldots j_i]$ with zero or more subtrees removed from $T_1[l(i)\ldots i]$. Let $T_{-DR}(i, j)$ denote the minimum distance between tree $T_1[l(i)\ldots i]$ and $T_2[l(j)\ldots j]$ with zero or more subtrees removed from $T_1[l(i)\ldots i]$. We write the algorithm in the form suggested by the algorithm template.

**Algorithm Subtree Removal.**

empty_initialization:

$F_{-DR}(\emptyset, \emptyset) = 0$

left_initialization:

$F_{-DR}(T_1[l(i)\ldots i_1], \emptyset) = 0$

right_initialization:

$F_{-DR}(\emptyset, T_2[l(j)\ldots j_i]) = F_{-DR}(\emptyset, T_2[l(j)\ldots j_i-1]) + \gamma(A \rightarrow T_2[j_i])$

general_if_computation

/* applies if $l(i_1) = l(i)$ and $l(j_1) = l(j)*/$

$F_{-DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_i]) = \min \{ $
\[ F_{DR}(\emptyset, T_2[l(j)\ldots j_1]), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i_1] \rightarrow \Lambda), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow \Lambda), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow T_2[j_1]), \]
/* put the derived treedist in the permanent array, as specified by template */

general_else_computation
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) = \min \{ \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow \Lambda), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow T_2[j_1]), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow T_2[j_1]) \}
/* put the derived treedist in the permanent array, as specified by template */

**Lemma 8.** Algorithm Subtree Removal is correct.

**Proof.** First we show that the initialization is correct. The \texttt{empty\_initialization} and the \texttt{right\_initialization} are the same as in the tree distance algorithm. The \texttt{left\_initialization} \( F_{DR}(T_1[l(i)\ldots i_1], \emptyset) = 0 \) is correct, because we can remove all of \( T_1[l(i)\ldots i_1] \).

For the general term \( F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) \), we ask first whether or not the subtree \( T_1[l(i)\ldots i_1] \) is removed. If it is removed, then the distance should be \( F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) \). Otherwise, consider the mapping between \( T_1[l(i)\ldots i_1] \) and \( T_2[l(j)\ldots j_1] \) after we perform an optimal removal of subtrees of \( T_1[l(i)\ldots i_1] \). Now we have the same three cases as in Lemma 4. Hence the general expression should be the minimum of these four terms:

\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) = \min \{ \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow \Lambda), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow T_2[j_1]), \]
\[ F_{DR}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1] + \gamma(T_i[i] \rightarrow T_2[j_1]) + T_{DR}(i_1, j_1)) \]
As in Lemma 5, this specializes to the \texttt{general\_if\_computation} and the \texttt{general\_else\_computation} given in the algorithm. \( \square \)

**5.2.2. Prune at any number of nodes from the TEXT tree.** Given trees \( T_1 \) and \( T_2 \), we want to know what is the minimum distance between \( T_1[l(i)\ldots i] \) and \( T_2[l(j)\ldots j] \) when there have been zero or more prunings at nodes of \( T_1[l(i)\ldots i] \).

Let \( F_{DP}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) \) denote the minimum distance between forest \( T_1[l(i)\ldots i_1] \) and \( T_2[l(j)\ldots j_1] \) with zero or more pruning from \( T_1[l(i)\ldots i_1] \). Let \( T_{DP}(i, j) \) denote the minimum distance between tree \( T_1[l(i)\ldots i] \) and \( T_2[l(j)\ldots j] \) with zero or more prunings from \( T_1[l(i)\ldots i] \). The following initialization and general term computation steps will give us an algorithm to solve our problem.

**Algorithm Prunings.**

**empty\_initialization:**
\[ F_{DP}(\emptyset, \emptyset) = 0 \]
**left\_initialization:**
\[ F_{DP}(T_1[l(i)\ldots i_1], \emptyset = F_{DP}(T_1[l(i)\ldots i_1], \emptyset) + \gamma(T_i[i] \rightarrow \Lambda) \]
**right\_initialization:**
\[ F_{DP}(\emptyset, T_2[l(j)\ldots j_1]) = F_{DP}(\emptyset, T_2[l(j)\ldots j_1]) + \gamma(\Lambda \rightarrow T_2[j_1]) \]
**general\_if\_computation**
/* applies if \( l(i) = l(i) \) and \( l(j) = l(j) */
\[ F_{DP}(T_1[l(i)\ldots i_1], T_2[l(j)\ldots j_1]) = \min \{ \]


TREE EDITING DISTANCE

\[ F_{\text{DP}}(\emptyset, T_1[l(i)\ldots j_i - 1]) + \gamma(T_1[i_i] \rightarrow T_2[j_i]) \]
\[ F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda) \]
\[ F_{\text{DP}}(T_1[(i)\ldots j_i], T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_2[j_j]) \]
\[ F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(T_1[i_i] \rightarrow \Lambda) \]
\[ F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_2[j_j]) \]

/* put the derived treeedit in the permanent array, as specified by template */

general_else_computation
\[ F_{\text{DP}}(T_1[(i)\ldots i_i], T_2[(j)\ldots j_j]) = \min \{
F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda),
F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(T_1[i_i] \rightarrow \Lambda),
F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_2[j_j]),
F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_2[j_j])\} \]

LEMMA 9. Algorithm Prunings is correct.

Proof. First we show that the initialization is correct. The empty_initialization and the right_initialization are the same as in the tree distance algorithm. For left_initialization, the best we can do for tree \( T_1[(i)\ldots i_i] \) is to prune at \( T_1[i_i] \). Therefore
\[ F_{\text{DP}}(T_1[(i)\ldots i_i], \emptyset) = F_{\text{DP}}(T_1[(i)\ldots i_i - 1], \emptyset) + \gamma(T_1[i_i] \rightarrow \Lambda) \]
Thus the left_initialization is correct.

For the general term \( F_{\text{DP}}(T_1[(i)\ldots i_i], T_2[(j)\ldots j_j]) \), we have the following similar three cases.

1. \( T_1[i_i] \) is not touched by a line of \( M \).
   - (1a) (without pruning) \( F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda) \)
   - (1b) (with pruning) \( F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + \gamma(T_1[i_i] \rightarrow \Lambda) \)
2. \( T_2[j_j] \) is not touched by a line of \( M \). Since we only prune from \( T_1 \), there is only one case here:
   - \( F_{\text{DP}}(T_1[(i)\ldots i_i], T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_1[i_i]) \)
3. both \( T_1[i_i] \) and \( T_2[j_j] \) are touched by lines of \( M \).
   - (3a) (without pruning)
     \[ F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow T_2[j_j]) \]
   - (3b) (with pruning) \( F_{\text{DP}}(T_1[(i)\ldots i_i - 1], T_2[(j)\ldots j_j - 1]) + F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow T_2[j_j]) \)

If \( l(i) = l(i_i) \) and \( l(j) = l(j_j) \), consider cases (1b) and (3b). Case (1b) becomes \( F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda) \). Case (3b) becomes \( F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow T_2[j_j]) \). Now from the right_initialization we know that
\[ F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda) \]
\[ \equiv F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j - 1]) + \gamma(\Lambda \rightarrow T_2[j_j]) + \gamma(T_1[i_i] \rightarrow \Lambda) \]
\[ \equiv F_{\text{DP}}(\emptyset, T_2[(j)\ldots j_j - 1]) + \gamma(T_1[i_i] \rightarrow T_2[j_j]). \]

So the distance given by case (1b) \( \equiv \) the distance from (3b). The proposed general_if_computation is therefore correct where the first term handles two cases.

If \( l(i) \neq l(i_i) \) or \( l(j) \neq l(j_j) \), consider case (3). As in Lemma 6, cases (3a) and (3b) can be replaced by \( F_{\text{DP}}(T_1[(i)\ldots l(i_i) - 1], T_2[(j)\ldots l(j_j) - 1]) + T_{DP}(i_i, j_j) \). The proposed general_else_computation is therefore correct. Hence algorithm pruning is correct. \( \square \)
6. Conclusion. We present a simple dynamic programming algorithm for finding the editing distance between ordered labelled trees. Our algorithm

(1) Has better time and space complexity than any in the literature;
(2) Is efficiently parallelizable; and
(3) Is generalizable with the same time complexity to approximate tree matching problems.

We have implemented these algorithms as a toolkit that has already been used at the National Cancer Institute.

Acknowledgments. We thank Bob Hummel for helpful discussions and the referees for valuable comments.

REFERENCES


---

3 In a separate result obtained with the help of Rick Statman, we find that the editing distance between unordered labelled trees (i.e., where the sibling order is insignificant) is NP-complete. The reduction is from exact cover by 3-sets [Z89].