JOURNAL OF ALGORITHMS 11, 581–621 (1990)

Fast Algorithms for the Unit Cost Editing Distance between Trees*

DENNIS SHASHA AND KAIZHONG ZHANG

Courant Institute of Mathematical Sciences, New York University, New York, New York 10012

Received August 5, 1988; revised July 27, 1989

CONTENTS

1. PROBLEM

Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. Given two different trees, a natural question to ask is how similar or different they are. We consider the distance between two trees to be the minimum number of edit operations (insert, delete, and modify) necessary to transform one tree to another.

*This work was partially supported by the National Science Foundation under Grant DCR8501611 and by the Office of Naval Research under Grant N00014-85-K-0046.

We present three algorithms to find the distance. The first algorithm is a simple dynamic programming algorithm based on a postorder traversal whose complexity improves upon the best previously published algorithm due to Tai [Tai79].[1] The second and third algorithms are optimal speedup parallel algorithms based on the applications of suffix trees to the comparison problem. The cost of executing these algorithms is a monotonic increasing function of the distance between the two trees.

Results. Let trees T_1 and T_2 have numbers of levels L_1 and L_2, respectively. Let k be the actual distance between T_1 and T_2. Let N be $\min(|T_1|, |T_2|)$. The asymptotic running times (assuming a concurrent-read concurrent-write parallel random access machine) are:

Algorithm	Time	Processors
Tai	$\|T_1\| \times \|T_2\| \times L_1^2 \times L_2^2$	1
Alg0 (basic)	$\|T_1\| \times \|T_2\| \times L_1 \times L_2$	1
Alg0 parallel	$\|T_1\| + \|T_2\|$	$\|T_1\| \times \|T_2\| \times \min(\|L_1\|, \|L_2\|)$
Alg1 (simple)	$k^2 \times N \times \min(\|L_1\|, \|L_2\|)$	1
Alg2 parallel	$k \times \log(k) \times \log(N)$	$k^2 \times N$
Alg3 parallel	$(k^2 \times \log(k)) + \log(N)$	$k^2 \times N$

Actually, we have shown in [ZS89] that for special trees, the complexity of our basic algorithm is much lower. In particular L_1 can be replaced by the number of leaves in T_1 and L_2 can be similarly replaced in the complexity expressions.

Application approach. We are applying these algorithms to comparing tree descriptions of spatial curves, secondary structures of RNA, and sentence parses.

The RNA problem is of the greatest immediate interest to us since researchers at the National Cancer Institute plan to use these algorithms. 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 [ALKBO87, BSSBWD87, DD87] have observed that the secondary structure influences translation rates (from RNA to proteins). Because different sequences can produce similar secondary structures [DA82, SK76], comparisons among secondary structures are necessary to understanding the comparative functionality of different RNAs.

Existing methods for comparing multiple secondary structures of RNA molecules represent the tree structures as parenthesized strings [S88]. A

[1]A paper by Lu [L79] solves a degenerate case (two-level trees) of this problem.

multiple string alignment program clusters these tree representations according to a string distance editing metric. This heuristic measure of tree similarity does a good job of clustering similar trees and subtrees. However, it is easy to show examples where string distance suggests that T_1 and T_2 are closer than T_1 and T_3 whereas the reverse is true [Z89].

Algorithmic significance. We use the Ukkonen [U83] technique of computing in waves along the center diagonals of the distance array. At the beginning of stage k, all distances up to $k - 1$ have been computed. Stage k then computes in parallel all distances up to k. We use suffix trees, inspired by [LV86], to perform this computation fast. But, whereas Landau and Vishkin apply suffix trees to comparing strings we apply suffix trees to comparing trees. That is, we map each of the two trees T_1 and T_2 to strings (each string is a traversal order where each node is associated with the number of its children), construct suffix trees from these strings, and then use the suffix trees to infer that portions of the T_1 are identical to portions of T_2. This leaves some subtle problems.

In the string case, if $S_1[i \dots i + h] = S_2[j \dots j + h]$, then the distance between $S_1[1 \dots i - 1]$ and $S_2[1 \dots j - 1]$ is the same as that between $S_1[1 \dots i + h]$ and $S_2[1 \dots j + h]$. The main difficulty in the tree case is that preserving ancestor relationships in the mapping between trees prevents the analogous implication from holding. In addition, to compute the distance between two forests at stage k sometimes requires knowing whether two contained subtrees are distance k apart. We overcome these problems by exploiting the relationship between identical subforests and tree-to-tree mappings (Section 4.4).

2. Edit Operations

Our distance metric for trees is a generalization of the editing distance between sequences. The edit operations are relabel, delete, and insert. Relabeling 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. Insert 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'. Figures 1, 2, and 3, illustrate these editing operations.

As mentioned above, Tai [Tai79] gives the previous best published algorithms with time complexity $O(|T_1| \times |T_2| \times L_1^2 \times L_2^2)$. Recently, [ZS89] introduced an extremely simple postorder traversal dynamic programming algorithm with complexity $O(|T_1| \times |T_2| \times L_1 \times L_2)$. (Both algorithms can handle non-unit edit costs.)

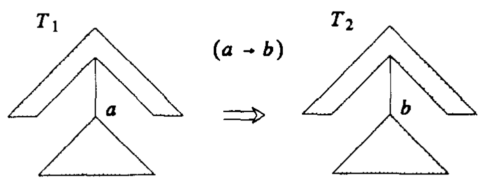

FIG. 1. Relabeling to change one node label to another.

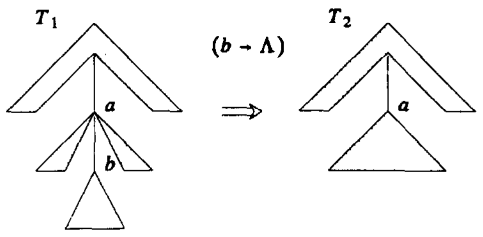

FIG. 2. Deletion of a node. (All children of the deleted node b become children of the parent a.)

The simplicity of [ZS89] led us to the solution of the following related problems:

1 (Approximate distance, flavor 2). What is the minimum distance between T_1 and T_2', where T_2' is T_2 with an arbitrary number of subtrees removed?

2 (Approximate tree matching). What is the minimum distance between T_1 to any t' such that t is a subtree of T_2 and t' is t with an arbitrary number of subtrees replaced by their roots?

These problems can be solved in the same time as the original problem.

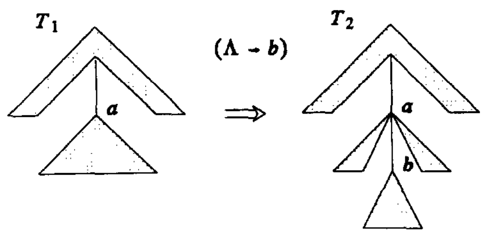

FIG. 3. Insertion of a node. (A consecutive sequence of siblings among the children of a become the children of b.)

In this paper, we solve the specialized problem of distance when the cost of relabeling, the cost of insertion, and the cost of deletion are all 1. We can do so faster as the table above shows.

2.1. *Definitions*

We represent an edit operation [Tai79, ZS89] as a pair $(a, b) \neq (\Lambda, \Lambda)$, sometimes written $a \rightarrow b$. We call $a \rightarrow b$ a relabeling operation if $a \neq \Lambda$ and $b \neq \Lambda$; a delete operation if $b = \Lambda$; and an insert operation if $a = \Lambda$. 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 $A_{i-1} \rightarrow A_i$ via s_i for $1 \leq i \leq k$.

This definition of editing operation is really a shorthand for the specification. Here is the specification in full detail. Consider a single editing operation, e.g., one that transforms A_{i-1} to A_i. If it is a relabeling operation, we specify the position in A_{i-1}. The same holds for a delete operation. An insert operation is a little more complicated: we must specify the parent p of the node n to be inserted and which consecutive subsequence of children of p will be the children of n. If that consecutive subsequence is empty, then we need to specify the position of n among the children of p. However, we will continue to use our shorthand because these other specifications will be clear from the mapping structure defined below.

For the purposes of this paper, the cost of any editing operation $a \rightarrow b$, denoted $\gamma(a \rightarrow b)$, is 1 if $a \neq b$ and 0 otherwise. Actually, these results generalize slightly: the algorithm can handle the case where all costs are integral, all insert and delete costs are the same, and the triangle inequality should be satisfied.

The cost of a sequence is simply the summation of the costs of each editing operation in the sequence. The *distance* between T_1 and T_2 is simply the minimum cost sequence taking T_1 to T_2. Our problem is to find the distance.

2.2. *Mappings*

The edit operations correspond to a *mapping* which 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 d), insert (node with label d)).

Formally a mapping from T_1 to T_2 is a triple (M, T_1, T_2), where M is any set of pair of integers (i, j) satisfying the following conditions (see

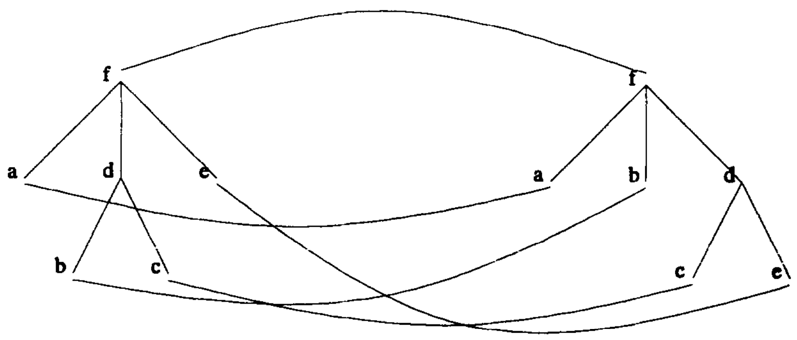

FIG. 4. Edit sequence: $((p \leftarrow V), (V \leftarrow p))$.

Fig. 5):

(1) $1 \le i \le |T_1|, 1 \le j \le |T_2|$;

(2) For any pairs (i_1, j_1) and (i_2, j_2) in M,

(a) (one-to-one) $i_1 = i_2$ iff $j_1 = j_2$

(b) (ancestor) $T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor of $T_2[j_2]$

(c) (sibling) $T_1[i_1]$ is to the left of $T_1[i_2]$ iff $T_2[j_1]$ is to the left of $T_2[j_2]$.

We use M instead of (M, T_1, T_2) if there is no confusion. The cost of M, denoted $\gamma(M)$, is the number of nodes to be inserted (i.e., those in T_2 that are not touched by a mapping line) plus the number to be deleted (i.e.,

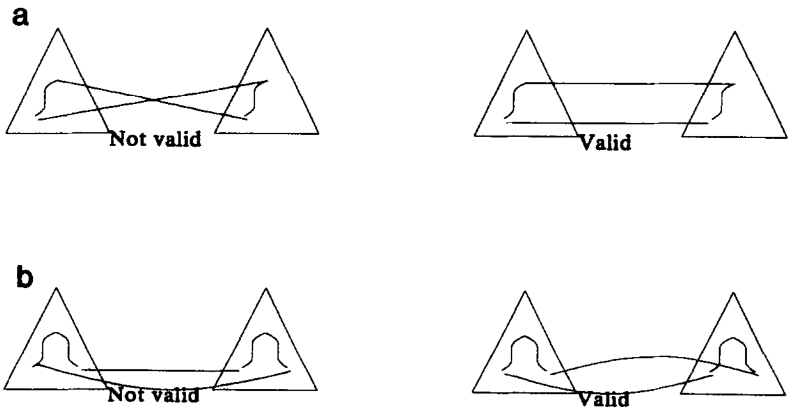

FIG. 5. Mapping rules: (a) Mapping must preserve ancestor descendant relationship; (b) mapping must preserve sibling order.

those in T_1 not touched by a line) plus the number relabeled (i.e., those pairs of nodes related by mapping lines with differing labels).

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

The following two lemmas are proved in [ZS89].

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

The second lemma says that the best mapping corresponds to the best sequence of edit operations. Intuitively, the sequence can be formed from the deletes induced by the mapping followed by the inserts and relabelings.

LEMMA 2. *Given S, a sequence $s_1, \ldots, s_k$ 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 $\gamma(S) = \gamma(M)$.*

Hence, $\delta(T_1, T_2) = \min\{\gamma(M) \mid M$ is a mapping from T_1 and $T_2\}$.

The equivalence between mappings and editing sequences simplifies later proofs by making the definition of distances constructive rather than operational.

2.3. *Left-to-Right Postorder Traversal Notation—The Default*

Let $T[i]$ be the ith node in the tree according to the left-to-right postorder numbering (our default traversal order). $l(i)$ is the number of the leftmost leaf descendent of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $l(i) = i$. The number of children of $T[i]$ is denoted $nc(i)$. The parent of $T[i]$ is denoted $p(i)$. We define $p^0(i) = i$, $p^1(i) = p$, $p^2(i) = p(p^1(i))$, and so on. Let $anc(i) = \{p^k(i) \mid 0 \leq k \leq depth(i)\}$.

$T[i \ldots j]$ is the ordered subforest of T induced by the nodes numbered i to j inclusive (Fig. 6). If $i > j$, then $T[i \ldots j] = \emptyset$. $T[1 \ldots i]$ will be referred to as *forest*(i), when the tree referred to is clear. $T[l(i) \ldots i]$ will be referred to as *tree*(i). *Size*(i) is the number of nodes in *tree*(i). If $T[i]$ and $T[i']$ are nodes, then $lca(i, i')$ is the postorder number of the least common ancestor of those two nodes. If $T[i]$ is an ancestor of $T[i']$ then the *proper child of* $T[i]$ *with respect to* $T[i']$ is the child of $T[i]$ on the path between $T[i']$ and $T[i]$.

The distance between $T_1[i' \ldots i]$ and $T_2[j' \ldots j]$ is denoted

$$dist(T_1[i' \ldots i], T_2[j' \ldots j]) \quad \text{or } dist(i' \ldots i, j' \ldots j)$$

if the context is clear. We use a more abbreviated notation for certain

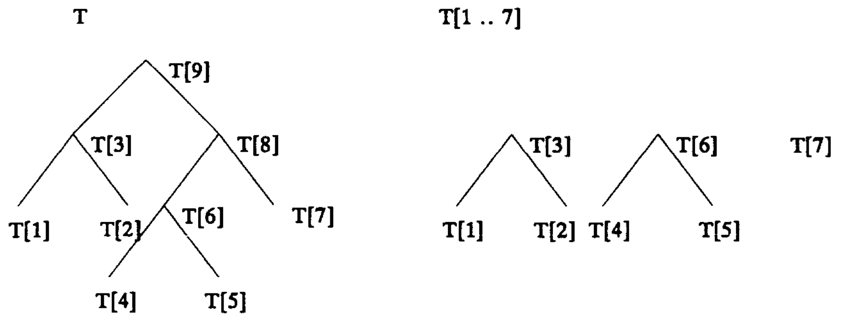

FIG. 6. Left-to-right postorder numbering.

special cases. The distance between $T_1[1 \ldots i]$ and $T_2[1 \ldots j]$ 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. BASIC ALGORITHM

We compute *forestdist*(i, j) for $1 \le i \le |T_1|$ and $1 \le j \le |T_2|$. Let M be a minimum-cost map between *forest*(i) and *forest*(j). The distance is the minimum of these three cases:

(1) $T_1[i]$ is not touched by a line in M. So, *forestdist*$(i, j) =$ *forestdist*$(i - 1, j) + 1$.

(2) $T_2[j]$ is not touched by a line in M. So, *forestdist*$(i, j) =$ *forestdist*$(i, j - 1) + 1$.

(3) $T_1[i]$ and $T_2[j]$ are touched by lines in M (see Fig. 7). By the ancestor and sibling conditions on mappings, (i, j) must be in M. By the

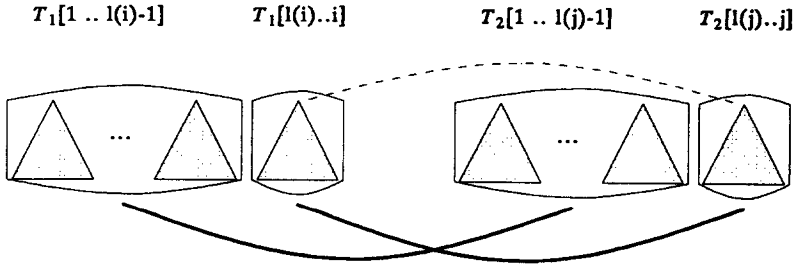

FIG. 7. Case 3 holds when (i, j) is in mapping.

ancestor condition on mapping, any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. Hence, $forestdist(i, j) = forestdist(l(i) - 1, l(j) - 1) + dist(T1[l(i) \dots i - 1], T2[l(j) \dots j - 1]) + \gamma(T[i] \to T[j])$.

Since these three cases express all the possible mappings yielding *forestdist*(i, j), we take the minimum of these three costs. Thus,

$$forestdist(i,j) = \min\begin{cases} forestdist(i-1,j)+1 \\ forestdist(i,j-1)+1 \\ forestdist((l(i)-1),(l(j)-1)) \\ \quad +dist(l(i)\dots i-1, l(j)\dots j-1) \\ \quad +\gamma(T[i] \to T[j]). \end{cases}$$

When either $l(i) \neq$ leftmost child of T_1 or $l(j) \neq$ leftmost child of T_2, we can use the equation $forestdist(i, j) = forestdist(l(i) - 1, l(j) - 1) + treedist(i, j)$. The reason is that $treedist(i, j)$ is bounded by the following two inequalities: $treedist(i, j) \leq dist(T_1[l(i) \dots i - 1], T_2[l(j) \dots j - 1]) + dist(T_1[i], T_2[j])$ and $forestdist(i, j) \leq forestdist(l(i) - 1, l(j) - 1) + treedist(i, j)$, or

$$forestdist(i,j) = \min\begin{cases} forestdist(i-1,j)+1 \\ forestdist(i,j-1)+1 \\ forestdist((l(i)-1),(l(j)-1)) + treedist(i,j). \end{cases}$$

These three cases specify a step of a simple dynamic programming algorithm. Because of case (3), subtree-to-subtree distances may be required. We explain the simple dynamic programming algorithm that comes out of this explanation below. [ZS89] shows the detailed program and also shows the improvement for special trees.

BASIC ALGORITHM 0. Allocate an array called the treedist array consisting of $|T_1| \times |T_2|$ cells. Cell (i, j) will hold *treedist*(i, j). When computing the (i, j)th entry, the algorithm has already computed all entries (i', j') such that $i' \leq i$ and $j' \leq j$ and strict inequality holds for at least one of these.

To compute each *treedist*(i, j), we use a temporary dynamic programming array. Since the postorder numbers of all nodes in $tree_1$ are less than or equal to i and similarly for $tree_2$, the array already has all treedistance values necessary for that computation. (That is why the postorder traversal is a better traversal order to use than the preorder traversal that Tai uses.) So, computing each cell of this temporary array takes constant time.

Computing the entire temporary array for *treedist*(i, j) requires time $O(\mathit{size}(i) \times \mathit{size}(j))$. Once *treedist*$(i, j)$ has been computed, the temporary array can be discarded.

Therefore to compute all subtrees requires time

$$O\left(\sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_1|} \mathit{size}(i) \times \mathit{size}(j)\right).$$

The contribution of a given node n in tree T_1 to the summation $\sum_{i=1}^{|T_1|} \mathit{size}(i)$ is $\mathit{depth}(n) + 1$. So,

$$\sum_{i=1}^{|T_1|} \mathit{size}(i) = \sum_{i=1}^{|T_1|} (1 + \mathit{depth}(i)) \leq \sum_{i=1}^{|T_1|} L_1 = |T_1| \times L_1.$$

The total time is therefore $O(|T_1| \times |T_2| \times L_1 \times L_2)$. Since the temporary matrices never exceed the size of the treedist array and since there is never more than one temporary array, the total space is $O(|T_1| \times |T_2|)$.

In the parallel case, we compute in waves for all subtree pairs simultaneously. We start at step 0. At step p, for each $t_1 = \mathit{tree}_1(i)$ and $t_2 = \mathit{tree}_2(j)$, compute $\mathit{dist}(T_1[l(i) \ldots i'], T_2[l(j) \ldots j'])$ such that $(i' - l(i)) + (j' - l(j)) = p$.

Thus, p denotes the counter-diagonal. It reaches its maximum value when i and j are the roots of the two trees. For that array the time is $O(T_1 + T_2)$.

Each wave has $\min(\mathit{size}(t_1), \mathit{size}(t_2))$ elements and can be calculated in parallel by that many processors. So the number of processors is

$$\sum_{t_1 \in T_1} \sum_{t_2 \in T_2} \min(\mathit{size}(t_1), \mathit{size}(t_2)) \leq |T_1|\,|T_2| \min(L_1, L_2),$$

where we convert the sum of sizes of subtrees to the number of levels as in the analysis of the sequential algorithm.

For any subtree pair, the processors must have access to all previously calculated forest-to-forest distances for that subtree pair as well as previously calculated tree-to-tree distance pairs. So, the space required is $O(|T_1| \times |T_2| \times L_1 \times L_2)$.

4. A Simple Improved Algorithm

Two ideas lead to an improvement of the basic algorithm. We start with the simple one.

Deciding whether two trees differ by at most k requires less work than determining what the distance between two trees is (e.g., there is no need to compute *treedist*(i, j), where $|i - j| > k$).

Even if we do not know the distance, we can use the following approach:

```
for $j := 0$ to $\lceil \log_2(|T_1| + |T_2|) \rceil$
if $T_1$ and $T_2$ are within distance $2^j$
    then print the distance and exit
end for
```

4.1. *Relevance—Computing for Short Distances First*

DEFINITION. Let the *treedistance triple* $(tree_1(i), tree_2(j), k)$ be shorthand for the question "Is the distance between $tree_1(i)$ and $tree_2(j)$ less than or equal to k and if so what is it?" for non-negative integer k.

DEFINITION. For all i, j, if $k \leq k'$, then we say that $(tree_1(i), tree_2(j), k')$ *covers* $(tree_1(i), tree_2(j), k)$.

Intuitively, answering $(tree_1(i), tree_2(j), k')$ gives us the answer to $(tree_1(i), tree_2(j), k)$.

We will restrict the set of treedistances calculated to those that are relevant. Intuitively, $(tree_1(i'), tree_2(j'), k')$ is relevant to $(tree_1(i), tree_2(j), k)$ if a mapping from $tree_1(i')$ to $tree_2(j')$ with distance k' could be a submapping of a mapping from $tree_1(i)$ to $tree_2(j)$ with distance k. Formally,

DEFINITION. $(tree_1(i'), tree_2(j'), k')$ is *relevant* to $(tree_1(i), tree_2(j), k)$ if

(1) $l(i) \leq i' \leq i$ and $l(j) \leq j' \leq j$.

(2) $|(i' - l(i)) - (j' - l(j))| \leq k$.

(3) $k' \leq k - |(l(i') - l(i)) - (l(j') - l(j))|$.

The first condition says that $tree_1(i')$ is a subtree of $tree_1(i)$ and similarly for $tree_2(j')$. If the second condition were false, then *dist*$(l(i) \ldots i', l(j) \ldots j')$ would necessarily exceed k. The difference in the third condition is the maximum distance of any mapping between $tree_1(i')$ and $tree_2(j')$ that would be a submapping of a mapping between $tree_1(i)$ and $tree_2(j)$ with distance k.

LEMMA 4.1 (RELEVANCE). *If* $(tree_1(i'), tree_2(j'), k')$ *is relevant to* $(tree_1(i), tree_2(j), k)$ *and* $(tree_1(i''), tree_2(j''), k'')$ *is relevant to* $(tree_1(i'), tree_2(j'), k')$, *then* $(tree_1(i''), tree_2(j''), k'')$ *is relevant to* $(tree_1(i), tree_2(j), k)$.

Proof. Condition 1 is obvious.

Condition 2. We want to show that $|(i'' - l(i)) - (j'' - l(j))| \leq k$. Let us manipulate the left-hand side.

$$\begin{aligned}
&|(i'' - l(i)) - (j'' - l(j))| \\
&\quad = |(l(i') - l(i')) + (l(j') - l(j')) \\
&\qquad + (i'' - l(i)) - (j'' - l(j))| \\
&\quad \leq |(i'' - l(i')) - (j'' - l(j'))| + |(l(i') - l(i)) - (l(j') - l(j))|,
\end{aligned}$$

by the triangle inequality.

But this expression is bounded from above by $k' + (k - k') = k$.

Condition 3. By definition, $k^1 \leq k - |(l(i') - l(i)) - (l(j') - l(j))|$. Also, $k'' \leq k' - |(l(i'')) - l(i') - (l(j'') - l(j'))|$.

By the triangle inequality, $|(l(i'') - l(i)) - (l(j'') - l(j))| \leq |(l(i') - l(i)) - (l(j') - l(j))| + |(l(i'') - l(')) - (l(j'') - l(j'))|$. So, $k'' \leq (k - |(l(i') - l(i)) - (l(j') - l(j))|) - |(l(i'') - l(i')) - (l(j'') - l(j'))| \leq k - |(l(i'') - l(i)) - (l(j'') - l(j))|$. □

Definition. $allow(k, i, j) = k - |l(i) - l(j)|$.

Corollary 1. *If* $(tree_1(i'), tree_2(j'), h)$ *is relevant to* (T_1, T_2, k), *then* $0 \leq h \leq allow(k, i', j')$ *and* $|i' - j'| \leq k$.

Proof. From the definition of $allow(k, i, j)$ and relevance. □

Suppose that at some stage of the algorithm, we are only interested in (T_1, T_2, p). Lemma 3 and its corollary permit us to restrict our attention to only relevant subtrees. This saves us space and processors.

4.2. *A Simple Algorithm*

Let us now consider the algorithm to solve the problem of (T_1, T_2, K); i.e., if the distance between T_1 and T_2 is within K, then return the distance, else fail.

In order to solve the problem of (T_1, T_2, K) we only need to solve $(tree_1(i), tree_2(j), K - |l(i) - l(j)|)$ such that $|i - j| \leq K$ and $|l(i) - l(j)| \leq K$. The reason is that this set of triples is relevant to (T_1, T_2, K) and covers all the relevant triples. Hence from Lemma 3 we can observe the following:

(1) We only need to compute $treedist(i, j)$ such that $|i - j| \leq K$ and $|l(i) - l(j)| \leq K$.

(2) In the temporary array for compute $treedist(i, j)$ we only need to consider center diagonals less than $K - |l(i) - l(j)|$.

The above observations lead to an algorithm which is better than the basic algorithm especially in the case where K is small.

```
Preprocessing
(To compute $l()$)

Main loop (to compute $treedist(T_1, T_2, K)$)
  for $i = 1$ to $|T_1|$
    for $j = \max(i - K, 1)$ to $\min(|T_2|, i + K)$
      if $|l(i) - l(j)| \leq K$
        Compute $treedist(i, j, K - |l(i) - l(j)|)$;
```

We also use dynamic programming to compute *treedist*(i, j, k). 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.

The computation of *treedist*(i, j, k).

```
$forestdist(\emptyset, \emptyset) = 0$;

for $i_1 := l(i)$ to $l(i) + k$
  $forestdist(T_1[l(i) \ldots i_1], \emptyset) = i_1 - l(i) + 1$

for $j_1 := l(j)$ to $l(j) + k$
  $forestdist(\emptyset, T_2[l(j) \ldots j_1]) = j_1 - l(j) + 1$

for $i_1 := l(i)$ to $i$
  for $j_1 := \max(i_1 - k, l(j))$ to $\min(j, j_1 + k)$
    if $l(i_1) = l(i)$ and $l(j_1) = l(j)$ then
      $forestdist(T_1[l(i) \ldots i_1], T_2[l(j) \ldots j_1]) = \min\{$
        $forestdist(T_1[l(i) \ldots i_1 - 1], T_2[l(j) \ldots j_1]) + 1,$
        $forestdist(T_1[l(i) \ldots i_1], T_2[l(j) \ldots j_1 - 1]) + 1,$
        $forestdist(T_1[l(i) \ldots i_1 - 1], T_2[l(j) \ldots j_1 - 1]) + \gamma(T_1[i_1] \to$
          $T_2[j_1])\}$
  else
      $forestdist(T_1[l(i) \ldots i_1], T_2[l(j) \ldots j_1]) = \min\{$
        $forestdist(T_1[l(i) \ldots i_1 - 1], T_2[l(j) \ldots j_1]) + 1,$
        $forestdist(T_1[l(i) \ldots i_1], T_2[l(j) \ldots j_1 - 1]) + 1,$
        $forestdist(T_1[l(i) \ldots l(i_1) - 1], T_2[l(j) \ldots l(j_1) - 1]) +$
$treedist(i_1, j_1))\}$
```

LEMMA 4.2. *The time complexity of this simple algorithm is* $O(K^2 \min(T_1, T_2) \min(L_1, L_2))$.

Proof. For the computation of *treedist*(i, j), the time will be bounded by $K \times \min(size(i), size(j))$, since we only compute about $2 \times (K - |l(i) - l(j)|)$ diagonals. For each $tree_1(i)$ in T_1, *treedist*(i, l) will be computed for at most $2 \times K + 1$ values of l, i.e., those that satisfy the condition $|i - l| \leq K$. Hence the time will be at most $K^2 size(i)$ for each $tree_1(i)$.

The total time will be bounded by $K^2 \times T_1 \times L_1$. Symmetrically the total time will also be bounded by $K^2 \times T_2 \times L_2$. So we know that the time complexity will be bounded by the following:

(1) $K^2 \times T_1 \times L_1$.

(2) $K^2 \times T_2 \times L_2$.

(3) $T_1 \times T_2 \times L_1 \times L_2$.

If $T_1 \leq T_2$ and $L_1 \leq L_2$, then from (1) we know that the time complexity is $O(K^2 \min(T_1, T_2)\min(L_1, L_2))$.

If $T_1 \leq T_2$ and $L_2 \leq L_1$, then there are two cases. The first case is that $K \leq T_1$, since $T_2 \leq T_1 + K$ (otherwise the distance between the two trees must be greater than K), $T_2 \leq 2 \times T_1$. From (2) we know that time is $K^2 \times T_2 \times L_2$. So it will be $2 \times k^2 \times T_1 \times L_2$. Hence the time complexity is $O(K^2 \min(T_1, T_2)\min(L_1, L_2))$. The second case is that $K \geq T_1$, since $T_2 \leq T_1 + k, T_2 \leq 2 \times K$. From (3) we know that the time is $T_1 \times T_2 \times L_1 \times L_2$. So it will be less than $T_1 \times 2K \times K \times L_2$. Hence the time complexity is $O(K^2 \min(T_1, T_2)\min(L_1, L_2))$.

The other situation of $T_2 \leq T_1$ and $L_2 \leq L_1$ ($T_2 \leq T_1$ and $L_1 \leq L_2$) can be proved similarly. Hence we prove that the time complexity of the simple algorithm is $O(K^2 \min(T_1, T_2)\min(L_1, L_2))$. □

5. Improving the Simple Algorithm—The Second Improvement

5.1. *Strategies and Inspiration*

5.1.1. *Basic Strategies*

The following idea improves the simple algorithm proposed above. If a portion of the two trees (or of two subtrees) is the same, then it should be possible to speed up the algorithm over this portion. This will be the difficult improvement to apply. For inspiration, we look at a modified form of the Landau–Vishkin string algorithm.

5.1.2. *Inspiration*: *Landau–Vishkin Algorithm*

In the following discussion, diagonal d corresponds to the set of distances $\{stringdist(i, j) | j - i = d\}$. (The name diagonal comes from the distance array in the straightforward dynamic programming algorithm for string editing [U83].)

The basic algorithm of [LV86] can be modified to yield

```
for $p := 1$ to $|S_2|$ do
 for diagonals $d$ between $-p$ and $p$ inclusive pardo
  compute maximum row $i$ in $d$ such that $stringdist(i, i + d) \leq p$
  exit program when $stringdist(|S_1|, |S_2|)$ is computed
```

Here is the computation for a given diagonal at stage p:

(1) Find a row i in diagonal d with value p (consult diagonals $d - 1$ and $d + 1$ for this).

(2) Jump to $i + h$ if h is the maximum value such that $S_1[i \ldots i + h] = S_2[i + d \ldots i + d + h]$.

Both steps can be done in constant time, where step (2) uses a suffix tree. So the whole algorithm takes $O(k)$ time, where k is the actual distance between the two strings.

5.1.3. *Problems in Applying this Approach to Trees*

Problem 1. We would like to use suffix trees based on some traversal order, but a traversal order on labels alone is insufficient as Fig. 8a shows. On the other hand, it is well known [K73] that any traversal (we use a left-to-right postorder traversal) in which each label is associated with the number of its children is sufficient to specify the tree. We will call that traversal SLR_T.

Problem 2. Identical traversals with children are not necessary. That is, $forestdist(i, j) = forestdist(i + h, j + h)$ is possible even though $SLR_{T_1}[i + 1 \ldots i + h] \neq SLR_{T_2}[j + 1 \ldots j + h]$. See Fig. 8b.

Problem 3. Identical traversals with children are not sufficient. That is, $forestdist(i, j) < forestdist(i + h, j + h)$ is possible even though $SLR_{T_1}[i + 1 \ldots i + h] = SLR_{T_2}[j + 1 \ldots j + h]$. See Fig. 8c.

So, what are these traversals good for? Here is one way. If the single node labeled e in Fig. 8b or in 8c were replaced by a tree (or even a forest) of size r, then in both cases $forestdist(3, 2) = forestdist(3 + r, 2 + r)$ and this would be discovered by establishing that $SLR_{T_1}[3 + 1 \ldots 3 + r] = SLR_{T_2}[2 + 1 \ldots 2 + r]$.

5.1.4. *Lemmas to Achieve Second Improvement*

Here is a road map through the lemmas.

1. Lemma 5 shows that the dynamic programming array for the tree distance, like the one for strings, is monotonically non-decreasing along its diagonals. In particular, the distance array of size $|T_1| \times |T_2|$ can be

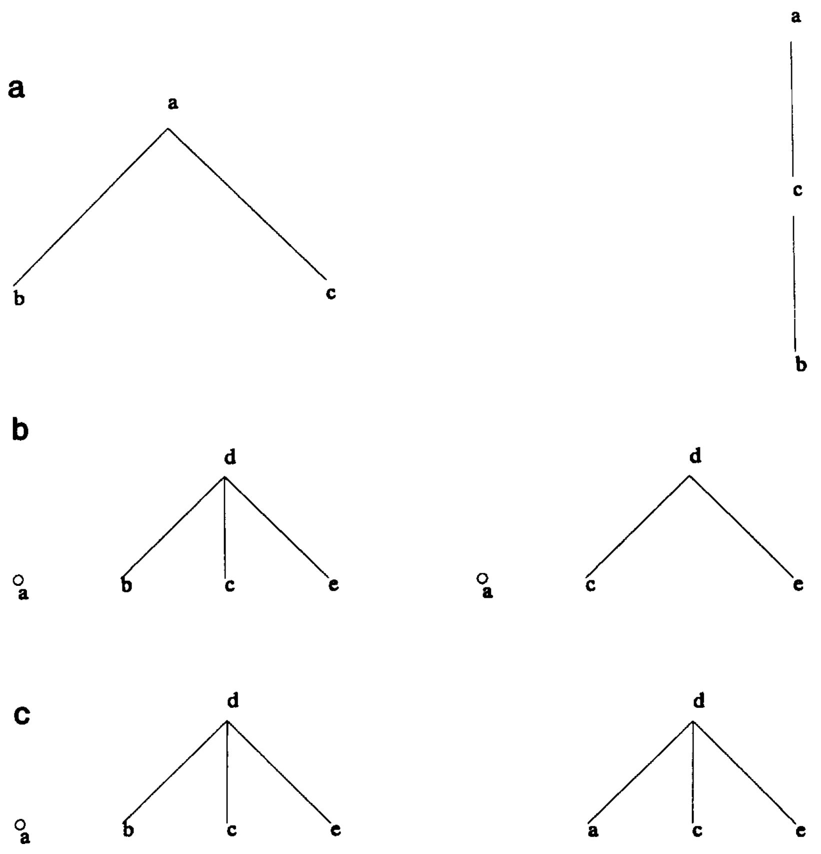

FIG. 8. (a) Different trees may have the same postorder traversal. (b) Label with number of children seems unnecessary (*forestdist*(3, 2) = *forestdist*(5, 4)). (c) Label with number of children seems not sufficient (*forestdist*(3, 2) ≠ *forestdist*(5, 4)).

encoded into an array with only $O(k^2)$ elements if k is the distance between T_1 and T_2.

2. Lemmas 6 to 8 give assurance that using equal portions of two trees or forests in certain generic situations is not a mistake.

3. We then consider various traversal orderings of trees, where the nodes are represented by their label and the number of their children.

4. Lemmas 9 to 11 give conditions for using such encoded traversal orderings.

5. Finally, we define two predicates up and down, computable in constant time (assuming that certain data structures on individual trees

have been created in a preprocessing step) that uses the ideas of the previous points. Up and down are used in our algorithms.

5.2. *Monotonicity*

LEMMA 5 (Monotonicity). *The distance values along the diagonal of any of the distance arrays are monotonically non-decreasing. That is,* $\mathit{forestdist}(i-1, j-1) \leq \mathit{forestdist}(i, j)$, *for* $i, j \geq 1$.

Proof. By induction on $i + j$.

Base case. By definition of insert weight and delete weight, $\mathit{forestdist}(0, i) = \mathit{forestdist}(i, 0) = i$, for any i. Therefore, $\mathit{forestdist}(0,0) = 0$. So $\mathit{forestdist}(0,0)$.

Inductive step. By our discussion of the basic step, there are three ways to handle $T_1[i]$ and $T_2[j]$ in the computation of $\mathit{forestdist}(i, j)$:

$$\mathit{forestdist}(i, j)$$
$$= \min\begin{cases} \mathit{forestdist}(i-1, j) + 1 \\ \mathit{forestdist}(i, j-1) + 1 \\ \mathit{dist}(T_1[l(i)\ldots i-1], T_2[l(j)\ldots j-1]) \\ +\mathit{forestdist}((l(i)-1),(l(j)-1)) + \gamma(T_1[i] \to T_2[j]). \end{cases}$$

The first and second terms are symmetric, so we address them by considering the case that $\mathit{forestdist}(i, j) = \mathit{forestdist}(i-1, j) + 1$.

If $i = 1$ then $\mathit{forestdist}(i, j) > \mathit{forestdist}(i-1, j) = \mathit{forestdist}(0, j) = j$ and $\mathit{forestdist}(i-1, j-1) = \mathit{forestdist}(0, j-1) = j-1$. So, $\mathit{forestdist}(i-1, j-1) \leq \mathit{forestdist}(i, j)$. Otherwise by the distance calculation $\mathit{forestdist}(i-1, j-1) \leq \mathit{forestdist}(i-2, j-1) + 1$. By induction hypothesis, $\mathit{forestdist}(i-2, j-1) + 1 \leq \mathit{forestdist}(i-1, j) + 1$. So, by transitivity, $\mathit{forestdist}(i-1, j-1) \leq \mathit{forestdist}(i, j)$.

If the third term applies, observe that the sum $\mathit{forestdist}(l(i)\ldots i-1,\ l(j)\ \ldots\ j-1) + \mathit{forestdist}((l(i)-1),(l(j)-1)) \geq \mathit{forestdist}(i-1, j-1)$. Since $\gamma(T_1[i] \to T_2[j])$ is non-negative, we then have $\mathit{forestdist}(i-1, j-1) \leq \mathit{forestdist}(i, j)$. □

After some stage in the algorithm, we will have located the largest row values with distance value p in each diagonal. In the next stage, the algorithm finds the first row i in diagonal d with value $p + 1$ and such that the neighboring positions in diagonals $d - 1$ and $d + 1$ have values $p + 1$ or greater. That row in diagonal d corresponds to position (i, j) (where $j = i + d$). We then try to find the greatest q such that $\mathit{forestdist}(i, j) = \mathit{forestdist}(i + q, j + q)$.

By monotonicity, for each r, $1 \leq r \leq q$, the values from diagonals $d - 1$ and $d + 1$ are irrelevant since $forestdist(i - 1, j) \geq p + 1$ and $forestdist(i, j - 1) \geq p + 1$. That means we only need to find a way to "continue" along diagonal d from position (i, j), running along identical parts of the two trees.

5.3. *Proper Forests and Quarantined Subtrees*

Because of the problems identified in Section 5.1.3, we must be careful to identify just when the suffix tree helps us. This section identifies two important (and intuitive) cases.

DEFINITION. Given forest F consisting of the subtrees rooted at $t_1, \ldots, t_p$, we say that $F[l(t_i) \ldots t_j]$ where $1 \leq i \leq j \leq p$ is a *proper subforest* of F. Intuitively, a proper subforest of F is a consecutive subsequence of the trees of F.

LEMMA 6 (One-sided proper subforest). *Suppose* $F[1 \ldots m]$ (*in Fig.* 9, F) *and* $F'[1 \ldots n]$ (F') *are ordered forests,* $F[1 \ldots x]$ (f_1) *is a proper subforest of* F, *and* $F'[1 \ldots y](f_1')$ *is a proper subforest of* F'.

(i) *If* $F[x + 1 \ldots m]$ (f_2) *and* $F'[y + 1 \ldots n]$ (f_2') *are identical, then* $dist(1 \ldots m, 1 \ldots n) = dist(1 \ldots x, 1 \ldots y)$.

(ii) *If* $F[1 \ldots x]$ *and* $F'[1 \ldots y]$ *are identical, then* $dist(1 \ldots m, 1 \ldots n) = dist(x + 1 \ldots m, y + 1 \ldots n)$.

Proof. (i) Let $left(F) = F[1 \ldots x]$ and $left(F') = F'[1 \ldots y]$. Let $right(F) = F[x + 1 \ldots m]$ and $right(F') = F'[y + 1 \ldots n]$.

(1) Since $right(F)$ and $right(F')$ are identical, $m - x = n - y$. From Lemma 5 (monotonic), $dist(1 \ldots m, 1 \ldots n) \geq dist(1 \ldots x, 1 \ldots y)$.

(2) We construct a particular mapping between F and F' as follows: We map $left(F)$ to $left(F')$ using the best mapping between them. We map $right(F)$ to $right(F')$. Since $right(F)$ (resp. $right(F')$) is a proper subforest of F (resp. F'), this gives a mapping.

In the above mapping, $dist(1 \ldots m, 1 \ldots n) \leq dist(1 \ldots x, 1 \ldots y) + dist(x + 1 \ldots m, y + 1 \ldots n)$. Since $right(F)$ and $right(F')$ are identical, $dist(x + 1, \ldots, m, y + 1, \ldots, n) = 0$. Hence, $dist(1 \ldots m, 1 \ldots n) \leq dist(1 \ldots x, 1 \ldots y)$, by (1) and (2). Hence, $dist(1 \ldots m, 1 \ldots n) = dist(1 \ldots x, 1 \ldots y)$.

(ii) This proof is symmetric to (i). Just consider the left–right mirror images of F and F'. □

LEMMA 7 (Two-sided proper subforest). *Suppose* $F[1 \ldots m]$ (*in Fig.* 9, F) *and* $F'[1 \ldots n](F')$ *are ordered forests,* $F[1 \ldots x](f_1)$ *is a proper subfor-*

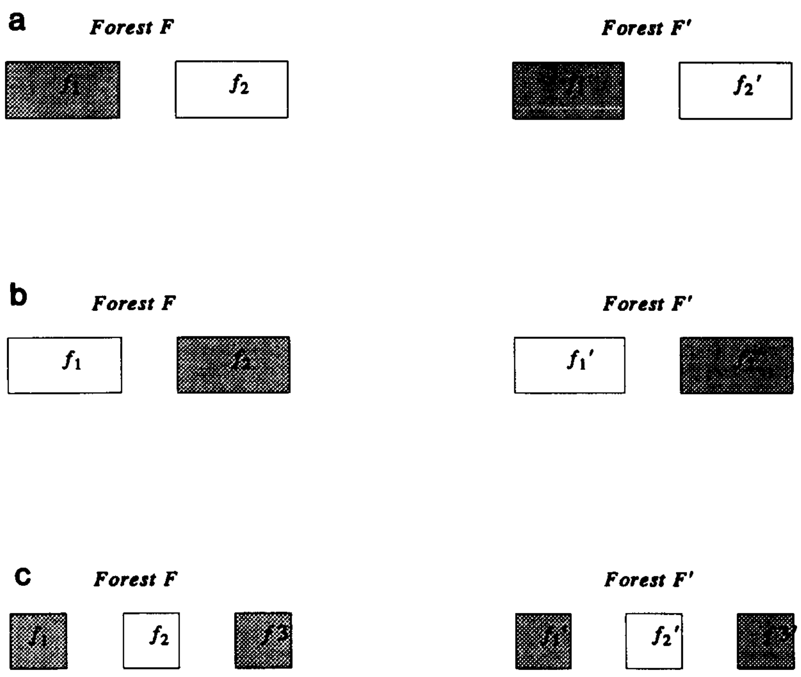

FIG. 9. Proper subforest lemmas. (a) If $dist(f_1, f_1') = 0$ then $dist(F, F') = dist(f_2, f_2')$. (b) If $dist(f_2, f_2') = 0$ then $dist(F, F') = dist(f_1, f_1')$. (c) If $dist(f_1, f_1') = 0$ and $dist(f3, f3') = 0$ then $dist(F, F') = dist(f_2, f_2')$.

est of F, $F[x + 1, \ldots, y - 1](f_2)$ is a proper subforest of F, $F'[1 \ldots x](f_1')$ is a proper subforest of F', and $F'[x + 1 \ldots z - 1](f_2')$ is a proper subforest of F'. If $F[1 \ldots x]$ is identical to $F'[1 \ldots x]$ and $F[y \ldots m](f3)$ is identical to $F'[z \ldots n](f3')$, then $dist(1 \ldots m, 1 \ldots n) = dist(x + 1 \ldots y - 1, x + 1 \ldots z - 1)$.

Proof. By (i) in Lemma 6, $dist(1 \ldots m, 1 \ldots n) = dist(1 \ldots y - 1, 1 \ldots z - 1)$. By (ii) in Lemma 6, $dist(1 \ldots m, 1 \ldots n) = dist(1 \ldots y - 1, 1 \ldots z - 1) = dist(x + 1 \ldots y - 1, x + 1 \ldots z - 1)$. □

We now propose the analogous property for trees.

DEFINITION. Suppose $T_1[1 \ldots m]$ and $T_2[1 \ldots n]$ are ordered trees, $T_1[l(i) \ldots i]$ (also denoted $tree_1(i)$) is a subtree of T_1 and $T_2[l(j) \ldots j]$ (also denoted $tree_2(j)$) is a subtree of T_2. We say that *the only difference between T_1 and T_2 is between $tree_1(i)$ and $tree_2(j)$* if replacing both $tree_1(i)$ and $tree_2(j)$ by a single node with the same label makes T_1 and T_2 identical.

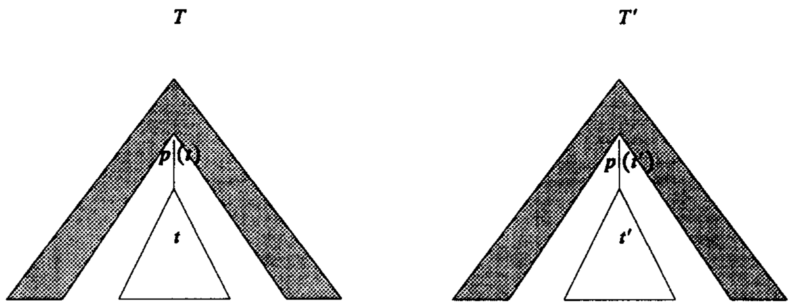

FIG. 10. Quarantined subtree lemma. If shaded parts of the two trees are the same, then *dist*$(T, T') =$ *dist*(t, t').

LEMMA 8 (Quarantined subtree). *If the only difference between* $T[1 \dots m]$ *(in Fig.* 10, T*) and* $T'[1 \dots n](T')$ *is between* $T[l(i) \dots i](t)$ *and* $T'[l(j) \dots j](t')$, *then treedist*$(m, n) =$ *treedist*(i, j).

Proof. If $i = m$ and $j = n$, then the lemma is trivial. From the condition of the lemma, it is clear that $i = m$ and $j \neq n$ (or $i \neq m$ and $j = n$) is not possible. So we consider the case $i \neq m$ and $j \neq n$.

(1) Since only *tree*$_1(i)$ and *tree*$_2(j)$ differ, $m - i = n - j$. Hence from Lemma 5 (Monotonicity), we know *dist*$(1 \dots m, 1 \dots n) \geq$ *dist*$(1 \dots i, 1 \dots j)$. Since $T_1[1 \dots l(i) - 1]$ and $T_2[1 \dots l(j) - 1]$ are identical and $T_1[l(i) \dots i]$ is a proper subforest of $T_1[1 \dots i]$ and $T_2[l(j) \dots j]$ is a proper subforest of $T_2[1 \dots j]$, from Lemma 7, we know *forestdist*$(i, j) =$ *dist*$(1 \dots i, 1 \dots j) =$ *dist*$(l(i) \dots i, l(j) \dots j) =$ *treedist*(i, j). Hence *forestdist*$(m, n) =$ *dist*$(1 \dots m, 1 \dots n) \geq$ *dist*$(l(i) \dots i, l(j) \dots j) =$ *treedist*(i, j).

(2) Recall that replacing both $T_1[l(i) \dots i]$ and $T_2[l(j) \dots j]$ by a single node with the same label causes T_1 and T_2 to be identical. So, we known that the following is a mapping for $T_1[1 \dots m]$ and $T_2[1 \dots n]$. Use the best mapping to map $T_1[l(i) \dots i]$ to $T_2[l(j) \dots j]$. Map the identical parts of T_1 and T_2 to each other. By construction, this does not violate the conditions of the mappings. Hence *forestdist*$(m, n) =$ *dist*$(1 \dots m, 1 \dots n) \leq$ *dist*$(l(i) \dots i, l(j) \dots j) =$ *treedist*(i, j).

Combining (1) and (2) gives the conclusion. □

5.4. *Traversal Orderings and Continuation*

For any tree, we consider four traversal orderings: left-to-right and right-to-left postorder and preorder. We also consider the suffix trees corresponding to these orders such that each element consists of a label of each node in order and the number of children of that node. (For tree T, these are called SLR_T and SRL_T for the postorder left-to-right and right-to-left traversals and $SLRP_T$ and $SRLP_T$ for the corresponding preorder traversals.) By default, we use the left-to-right postorder number as the identifier of each tree node. This ensures that our previous notation continues to make sense.

Given tree node i according to the left-to-right postorder traversal ordering, then

$lr(i)$ is the left-to-right postorder number of node i and so equals i.

$rl(i)$ is the right-to-left postorder number of node i.

$lrp(i)$ is the left-to-right preorder number of node i.

$rlp(i)$ is the right-to-left preorder number of node i.

To describe consecutive subsequences of nodes according to some traversal, we use the notation $T[traversaltype(number1 \dots number2)]$, where the traversal type is *LR*(*left-to-right postorder*), *RL*(*right-to-left postorder*), *LRP*(*left-to-right preorder*), *or RLP*(*right-to-left preorder*). *Number*1 and *number*2 are numbers in the given traversal type. As a special case, we use $T[traversaltype(number)]$ to replace $T[traversaltype(number \dots number)]$.

EXAMPLE. In Fig. 6, the nodes are numbered based on a left-to-right postorder traversal order; however, $T[LRP(3 \dots 7)]$ is the induced forest on the third through seventh nodes in the left-to-right preorder traversal. So, the forest consists of the nodes $T[1]$, $T[2]$, $T[8]$, $T[6]$, and $T[4]$. On the other hand, $T[LRP(lrp(4) \dots lrp(7))]$ is the induced forest on the seventh ($lrp(4) = 7$) through the fifth ($lrp(8) = 5$) nodes in the left-to-right preorder traversal. This is empty. (Note that $lrp(4) = 7$, because node 4 in the left-to-right postorder traversal is the seventh node in the left-to-right preorder traversal.) End of example.

One Notational Convenience. $T[LR(lr(i) \dots lr(j))]$ is the induced forest in the left-to-right postorder traversal with numbers between i and j. That is, it is the induced forest on $T[i]$, $T[i+1], \dots, T[j]$. Since this ordering is the default, we sometimes write this as $T[i \dots j]$.

Fact. The following four representations uniquely represent a tree (see [K73, Vol. 1, p. 350]):

(1) The string derived from the left-to-right postorder of the tree such that element i of the string contains both the label and the number of children of the node whose left-to-right postorder number is i. We use $SLR_T[1 \dots n]$ to represent this string.

(2) The analogous string derived from the right-to-left postorder of the tree. We use $SRL_T[1 \dots n]$ to represent this string. In this expression $SRL_T[i]$ is the node whose right-to-left postorder number is i.

(3) The analogous string derived from the left-to-right preorder of the tree. We use $SLRP_T[1 \dots n]$ to represent this string.

(4) The analogous string derived from the right-to-left preorder of the tree. We use $SRLP_T[1 \dots n]$ to represent this string.

The next lemmas in this section have intuitive statements and rather technical proofs.

LEMMA 9 (Postorder). *Proposition Postorder. If* $SLR_{T_1}[i \dots i + q] = SLR_{T_2}[j \dots j + q]$, *then* $T_1[i \dots i + q]$ *and* $T_2[j \dots j + q]$ *are identical.*

Proof. If $q = 0$, the lemma is clearly true. If $nc(i + q' + 1) = nc(j + q' + 1) = 0$, then $T_1[i + q' + 1]$ and $T_2[j + q' + 1]$ are leaves, so the conclusion is obvious. (Recall that $nc(i)$ is the number of children of i.) In the postorder traversal, $T_1[i + q' + 1]$ is the parent of the rightmost $nc(i + q' + 1)$ trees in $T_1[i \dots i + q']$ (or all the trees in that ordered forest if there are not that many in the forest). Similarly, for $T_2[j + q' + 1]$. By induction hypothesis the trees from the two ordered forests must be identical. Conclusion follows. □

LEMMA 10 (Preorder). *If in the preorder traversal (left-to-right or right-to-left)* $SLRP_{T_1}[i \dots i + d] = SLRP_{T_2}[j \dots j + d]$, *then the induced subgraphs* $T_1[LRP(i \dots i + d)] = T_2[LRP(j \dots j + d)]$. *If* $SLRP_{T_1}[i] \neq SLRP_{T_2}[j]$, *then* $tree_1(LRP(i)) \neq tree_2(LRP(j))$.

Proof. By Lemma 9 and the fact that if tree T has n nodes, then the left-to-right postorder node i is the same as right-to-left preorder node $n + 1 - i$. Therefore $T[LR(i \dots i + d)] = T[RLP(n + 1 - (i + d) \dots n + 1 - i)]$. A similar relationship holds between the right-to-left postorder and left-to-right preorder. □

LEMMA 11 (Must include). *Suppose* $forestdist(i, j) = k$. *Suppose further that any mapping between* $T_1[1 \dots i]$ *and* $T_2[1 \dots j]$ *with cost* k *must include* (i, j) *and finally suppose that* $forestdist(i + q, j + q) = k$. *Then*

1. *For* s, $0 \leq s \leq q$, $(i + s, j + s)$ *must be in the mapping between* $forest(i + q)$ *and* $forest(j + q)$ *with cost* k *and* $T_1[i + s] = T_2[j + s]$.

2. *For any* s *such that* $p^s(i) \leq i + q, (p^s(i), p^s(j))$ *is in the mapping between* $forest(i + q)$ *and* $forest(j + q)$ *with cost* k.

3. *For any* $q' \leq q$ *if* $T_1[i + q']$ *is not an ancestor of* $T_1[i]$, *then* $T_2[j + q']$ *is not an ancestor of* $T_2[j]$ *and* $nc(i + q') = nc(j + q')$.

4. *If* h, $h \leq q$, *is the greatest value such that* $T_1[i + h]$ *is an ancestor of* $T_1[i]$, *then* $T_1[i + h + 1 \dots i + q]$ *is identical to* $T_2[j + h + 1 \dots j + q]$.

Proof. Proof of 1. By the supposition, $forestdist(i - 1, j) \geq k$ and $forestdist(i, j - 1) \geq k$ (otherwise (i, j) would not have to be in the best mapping between *forest*(i) and *forest*(j)). Therefore the lemma holds for $s = 0$.

Suppose the lemma holds up to some value r. By Lemma 5 (Monotonicity), *forestdist*$(i + r - 1, j + r) \geq k$ and *forestdist*$(i + r, j + r - 1) \geq k$. So, $(i + r + 1, j + r + 1)$ must be in the best mapping between *forest*$(i + r + 1)$ and *forest*$(j + r + 1)$ yielding distance k. If we remove $(i + r + 1, j + r + 1)$ from that mapping we have a mapping between *forest*$(i + r)$ and *forest*$(j + r)$. The distance for that mapping is still k. By induction, for all s, $0 \leq s \leq r$, $(i + s, j + s)$ must be in this mapping. If $T_1[i + s] \neq T_2[j + s]$, then *forestdist*$(i + s, j + s) \geq k + 1$.

Remark. Before proving 2 and 3, we first note that for any $q' \leq q$, $T_1[i + q']$ is an ancestor of $T_1[i]$ iff $T_2[j + q']$ is an ancestor of $T_2[j]$. This holds since from 1 we know that (i, j) and $(i + q', j + q')$ are in the best mapping so the conclusion follows from the ancestor condition on mappings.

Proof of (2). Suppose 2 is not true. Let t be the smallest value such that $p^t(i) \leq i + q$ and $(p^t(i), p^t(j))$ is not in the best mapping. From the remark, we know that there is a $t' > t$ such that $(p^t(i), p^{t'}(j))$ is in the best mapping. Symmetrically, there must be a $t'' > t$ such that $(p^{t''}(i), p^t(j))$ is in the best mapping. This violates the ancestor conditions on mappings.

Proof of 3. By the remark, we know that if $T_1[i + q']$ is not an ancestor of $T_1[i]$ then $T_2[j + q']$ is not an ancestor of $T_2[j]$. Suppose 3 is not true; let t be the smallest value, $t \leq q$, such that $T_1[i + t]$ is not an ancestor of $T_2[j]$, $T_2[j + t]$ is not an ancestor of $T_2[j]$, and $nc(i + t) \neq nc(j + t)$. Let $h < t$ be the greatest value such that $T_1[i + h]$ is an ancestor of $T_1[i]$. By definition of t and h, we know that $SLR_{T_1}[i + h + 1 \ldots i + t - 1] = SLR_{T_2}[j + h + 1 \ldots j + t - 1]$. By Lemma 9 (Postorder) we know that $T_1[i + h + 1 \ldots i + t - 1]$ is identical to $T_2[j + h + 1 \ldots j + t - 1]$. Since $(i + t, j + t)$ is in the best mapping and since neither $T_1[i + t]$ is an ancestor of $T_1[i]$ nor $T_2[j + t]$ is an ancestor of $T_2[j]$, $nc(i + t) \neq nc(j + t)$ implies that a child of one of them, say $T_1[i + t]$, maps to a non-child of the other. This violates the ancestor condition on mappings.

Proof of 4. By 1, 3, and Lemma 9 (Postorder). □

Lemma 11 shows what *forestdist*$(i, j) =$ *forestdist*$(i + q, j + q)$ implies. Lemma 12 shows how to obtain that condition.

LEMMA 12 (Continuation condition). *Suppose forestdist*$(i, j) = p$, *any mapping between* $T_1[1 \ldots i]$ *and* $T_2[1 \ldots j]$ *with cost* p *must include* (i, j), *and forestdist*$(i + q - 1, j + q - 1) = p$. *Then forestdist*$(i + q, j + q) = p$

if

1. $T_1[i+q] = T_2[j+q]$; *and*

2. a. $T_1[i+q] \in anc(T_1[i])$ *and* $T_2[j+q] \in anc(T_2[j])$ *and* $treedist(i+q, j+q) = p - forestdist(l(i+q)-1, l(j+q)-1)$, *or*

   b. $T_1[i+q] \notin anc(T_1[i])$ *and* $T_2[j+q] \notin anc(T_2[j])$ *and* $nc(i+q) = nc(j+q)$.

Proof. There are two possibilities. If cases 1 and 2a hold, then $p = treedist(i+q, j+q) + forestdist(l(i+q)-1, l(j+q)-1)$. So, $forestdist(i+q, j+q) = p$ by definition of distance.

If 1 and 2b hold, then by Lemma 11 (Must include), point 4 (using the definition of h from that point), $T_1[i+h+1 \ldots i+q-1]$ is identical to $T_2[j+h+1 \ldots j+q-1]$. Therefore, we can use the distance p mapping from $forest(i+q-1)$ to $forest(j+q-1)$. By 2b, adding $(i+q, j+q)$ will preserve the ancestor condition on mappings. By 1, the addition will not increase the cost.

If $T_1[i+q] \notin anc(T_1[i])$ and $T_2[j+q] \in anc(T_2[j])$ or $T_1[i+q] \in anc(T_1[i])$ and $T_2[j+q] \notin anc(T_2[j])$, then $(i+q, j+q)$ and (i, q) would still have to be in the mapping by Lemma 11 (Must include), point 1, but this violates the ancestor condition on mappings, a contradiction. □

5.5 *Up and Down*

We now define predicates that will be computed using suffix trees. *Up* is used in algorithm 3 as identified in the table in the Introduction (the algorithm called "repeated hops" later). *Down* is used in Algorithm 2 as identified in that table (the algorithm called "binary search" later). We present *up* first because it is simpler to understand.

DEFINITION. Given trees T_1 and T_2 and a pair of subtrees $(tree_1(i), tree_2(j))$, define $up(i,j)$ to be a pair of subtrees rooted at (s,t) satisfying the following (Fig. 11):

(1) $tree_1(i)$ is a subtree of $tree_1(s)$ and $tree_2(j)$ is a subtree of $tree_2(t)$.

(2) $(tree_1(s), tree_2(t))$ is the largest subtree pair (equivalently, s and t are the greatest ancestors of i and j) such that the only difference between them is between $tree_1(i)$ and $tree_2(j)$. That is, if $tree_1(i)$ and $tree_2(j)$ are both replaced by a node with the same label, then $tree_1(s)$ would be identical to $tree_2(t)$.

Note that $up(i,j)$ is unique, because $level(s) - level(i) = level(t) - level(j)$, where $level(x)$ is the level of node x in the given tree. The

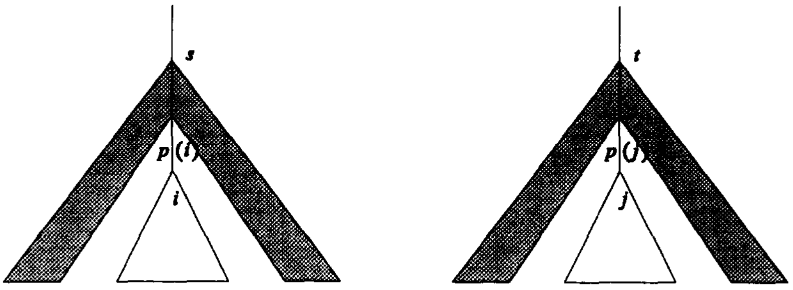

FIG. 11. If $up(i, j) = (s, t)$, the shaded parts of the two trees are the same.

following algorithm computes *up*:

(1) Find r such that $SLR_{T_1}[i + 1 \ldots i + r - 1] = SLR_{T_2}[j + 1 \ldots j + r - 1]$ and $SLR_{T_1}[i + r] \neq SLR_{T_2}[j + r]$.

(2) Find q such that $SLR_{T_1}[rl(i) + 1 \ldots rl(i) + q - 1] = SRL_{T_2}[rl(j) + 1 \ldots rl(j) + q - 1]$ and $SLR_{T_1}[rl(i) + q] \neq SRL_{T_2}[rl(j) + q]$.

(3) Find the least common ancestor sr of $T_1[LR(lr(i))]$ ($= T_1[i]$) and $T_1[LR(lr(i + r))]$. Find the least common ancestor sl of $T_1[RL(rl(i))]$ ($= T_1[i]$) and $T_1[RL(rl(i) + q)]$.

(4) Find the least common ancestor tr of $T_2[LR(lr(j))]$ ($= T_2[j]$) and $T_2[LR(lr(j + r)]$. Find the least common ancestor tl of $T_2[RL(rl(j))]$ ($= T_2[j]$) and $T_2[RL(rl(j) + q)]$.

(5) Let s be the proper child of the lower of sl and sr with respect to i. Let t be the proper child of the lower of tl and tr with respect to j.

(6) $up\ (i, j) = (s, t)$.

LEMMA 13 (Up). *This algorithm correctly computes up.*

Proof. By Lemma 9 (Postorder), point (1) implies that the induced subgraphs $T_1[LR(lr(i) + 1 \ldots lr(i) + r - 1)]$ and $T_2[LR(lr(j) + 1 \ldots lr(j) + r - 1)]$ are identical. However, since $SLR_{T_1}[i + r] \neq SLR_{T_2}[j + r]$, (sr, tr) cannot be $up(i, j)$, since there must be some difference between $tree_1(sr)$ and $tree_2(tr)$ besides the differences between $tree_1(i)$ and $tree_2(j)$. Now, let sr' be the proper child of sr with respect to i and tr' be the proper child of tr with respect to j. $sr' \leq i + r - 1$ and $tr' \leq j + r - 1$, so $T_1[i + 1 \ldots sr']$ is identical to $T_2[j + 1 \ldots tr']$. The same argument goes for the right-to-left postorder traversal. Define sl' and tl' similarly to sr' and tr'. Since s is just the lower of sr' and sl' and t is the lower of tr' and tl', $T_1[LR(lr(i) + 1 \ldots lr(s))]$ is the same as $T_2[LR(lr(j) + 1 \ldots lr(t))]$ and $T_1[RL(rl(i) + 1 \ldots rl(s))]$ is the same as $T_2[RL(rl(j) + 1 \ldots rl(t))]$. □

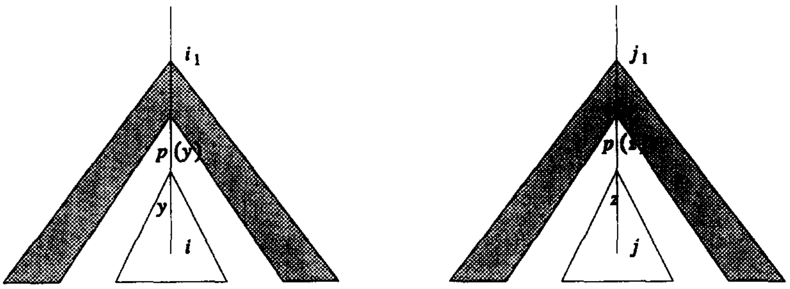

FIG. 12. If $down(i_1, j_1, i, j) = (y, z)$ the shaded parts of the two trees are the same.

The function down is symmetric to up. The extra two arguments in its definition result from the fact that there is a unique root, but there are many leaves in a given tree.

DEFINITION. Given trees T_1 and T_2 and subtree pair $(tree_1(i_1), tree_2(j_1))$ which contains $(tree_1(i), tree_2(j))$ respectively as subtrees, $down(i_1, j_1, i, j)$ is a pair of subtrees $(tree_1(y), tree_2(z))$ satisfying the following (Fig. 12):

(1) $tree_1(i)$ is a subtree of $tree_1(y)$ which is a subtree of $tree_1(i_1)$. $tree_2(j)$ is a subtree of $tree_2(z)$ which is a subtree of $tree_2(j_1)$.

(2) The pair (y, z) are the lowest ancestors of (i, j) such that $(tree_1(y), tree_2(z))$ satisfies condition (1) and the only difference between $tree_1(i_1)$ and $tree_2(j_1)$ is between $tree_1(y)$ and $tree_2(z)$.

Just as for up, $down(i_1, j_1, i, j)$ is unique, since $level(i_1) - level(y) = level(j_1) - level(z)$. Here is how to find $down(i_1, j_1, i, j)$.

(1) Find q such that $SLRP_{T_1}[lrp(i_1 \ldots lrp(i_1) + q] = SLRP_{T_2}[lrp(j_1) + q]$ and $SLRP_{T_1}[lrp(i_1) + q + 1] \neq SLRP_{T_2}[lrp(j_1) + q + 1]$. Let $q' := \min(q + 1, lrp(i) - lrp(i_1), lrp(j) - lrp(j_1))$.

(2) Find r such that $SLRP_{T_1}[rlp(i_1) \ldots rlp(i_1) + r] = SRLP_{T_2}[rlp(j_1) \ldots rlp(j_1) + r]$ and $SRLP_{T_1}[rlp\ (i_1) + r + 1] \neq SRLP_{T_2}[rlp(j_1) + r + 1]$. Let $r' := \min(r + 1, rlp(i) - rlp(i_1), rlp(j) - rlp(j_1))$.

(3) Let $sl :=$ the least common ancestor of $T_1[LR(lr(i))]$ $(= T1[i])$ and $T_1[LRP(lrp(i_1) + q')]$. Let $sr :=$ the least common ancestor of $T_1[i]$ and $T_1[RLP(rlp(i_1) + r')]$.

(4) Let $tr :=$ the least common ancestor of $T_2[j]$ and $T_2[LRP(lrp(j_1) + q')]$. Let $tl :=$ the least common ancestor of $T_2[j]$ and $T_2[RLP(rlp(j_1) + r')]$.

(5) Let y be the highest of sl and sr. Let z be the highest of tl and tr.

(6) $down(i_1, j_1, i, j) = (y, z)$.

LEMMA 14 (Down). *The algorithm correctly computes down.*

Proof. After step (1), we know that $T_1[LRP(lrp(i_1)\ldots lrp(i_1) + q')] = T_2[LRP(lrp(j_1)\ldots lrp(j_1) + q')]$. After step (2), we know that $T_1[RLP(rlp(i_1)\ldots rlp(i_1) + r')] = T_2[RLP(rlp(j_1)\ldots rlp(j_1) + r')]$.

Since $lrp(sl) \le lrp(i_1) + q' + 1$, we know that $T_1[LRP(lrp(i_1)\ldots lrp(sl) - 1)]$ is identical to $T_2[LRP(lrp(j_1)\ldots lrp(j_1) + (lrp(sl) - (1 + lrp(i_1))))]$. Similarly, we know that $T_1[RLP(rlp(i_1)\ldots rlp(sr) - 1)]$ is identical to $T_2[RLP(rlp(j_1)\ldots rlp(j_1) + (rlp(sr) - (1 + rlp(i_1))))]$.

Note that $T_1[LRP(lrp(i_1)\ldots lrp(i_1) + q')]$ maps to a portion of T_2 all of whose nodes are either on the path from j to j_1 or to the left of that path. Also, $lrp(sl) - 1 \le lrp(i_1) + q'$. So, $T_1[LRP(lrp(i_1)\ldots lrp(sl) - 1)]$ maps to nodes that are either on the path from j to j_1 or to the left of that path in T_2. Similarly, $T_1[RLP(rlp(i_1)\ldots rlp(sr) - 1)]$ maps to nodes that are either on the path from j to j_1 or to the right of that path in T_2.

Since s is the least common ancestor of sl and sr, there must be two nodes tl' and tr' such that $T_1[LRP(lrp(i_1)\ldots lrp(s) - 1)]$ is identical to $T_2[LRP(lrp(j_1)\ldots lrp(tl') - 1)]$ and $T_1[RLP(rlp(i_1)\ldots rlp(s) - 1)]$ is identical to $T_2[RLP(rlp(j_1)\ldots rlp(tr') - 1)]$. (Remark that this implies that $level(i_1) - level(s) = level(j_1) - level(tl') = level(j_1) - level(tr')$.)

CLAIM. $T_2[tl'] = T_2[tr']$.

Proof of claim. Suppose not. Let x be the least common ancestor of tl' and tr' in tree T_2. By construction, tl' is on or to the left of the path from j to j_1 and tr' is on or to the right, so x must be on the path from j to j_1. Because of the remark before the claim, $level(tl') = level(tr')$, so tl' and tr' must not be related by the ancestor relationship. This implies that x must be distinct from both of them.

Therefore $T_2[x]$ maps to a node in $T_1[LRP(lrp(i_1)\ldots lrp(s) - 1)]$ and in $T_1[RLP(rlp(i_1)\ldots rlp(s) - 1)]$. The node must be the same node $T_1[y]$. (Reason. $level(j_1) - level(x) = level(i_1) - level(y)$.) Let xl' be the proper child of x with respect to tl' and xr' be the proper child of x with respect to tr'. Let y' be the proper child of y with respect to i. Then, in the left-to-right preorder traversal y' maps to xl' and in the right-to-left preorder traversal, y' maps to xr'. There have to be the same number of left siblings of xl' as of y' and the same number of right siblings of xr' and y'; therefore x and y would have different numbers of children, a contradiction. End of proof of claim.

Now, tl' ($= tr'$) must be an ancestor (perhaps improper) of t, by the definition of t. By a symmetric construction, we can define sl' and sr' in tree T_1. It must be that $sl' = sr'$ and that sl' is an ancestor of s. By the remark before the claim, $level(i_1) - level(s) = level(j_1) - level(tl') \leq level(j_1) - level(t) = level(i_1) - level(sl') \leq level(i_1) - level(s)$. So, $s = sl'$ and $t = tl'$. The conclusion follows. □

5.6. *Preprocessing*

LEMMA 15. *With $O(n)$ sequential preprocessing time and $O(\log n)$ parallel time, we can construct the following from a set of parent–child and sibling edges*:

0. *All traversal orderings, the number of children of each node and $l(i)$* [TV85].

1. *A suffix tree for a string* [AILSV88]. *In our case the string will be two postorder and two preorder traversals (left-to-right and right-to-left in each case) with the numbers of children associated with each node.*

2. *A data structure for constant-time computation of the least common ancestor of any two nodes in a tree* [SV88].

3. *A data structure for constant-time computation of the proper child of any node n with respect to descendant m (by a similar method to* [SV88]).

COROLLARY. *Up and down can be computed in constant time using these suffix trees and data structures.*

6. ALGORITHMS

6.1. *Encoding of Distance Array*

In these algorithms, we do not maintain $forestdist(i, j)$ for all i, j. Instead, we encode these values using an auxiliary array called the *f-array*. $f(d, p)$ is the row number r such that the intersection between diagonal d and row $r + 1$ holds a number that is greater than p, but the intersection between d and row r holds a number that is less than or equal to p in the distance array. Diagonal d in the distance array is the set of values $forestdist(i, j)$ such that $j - i = d$. (Every different subtree pair that is evaluated has an analogous construct, called its g-array, but for notational simplicity, we will only discuss the f-array concerned with the main tree pair, T_1 and T_2.)

Suppose we want to determine $forestdist(i, j)$ given the f-array. We proceed by binary search as follows:

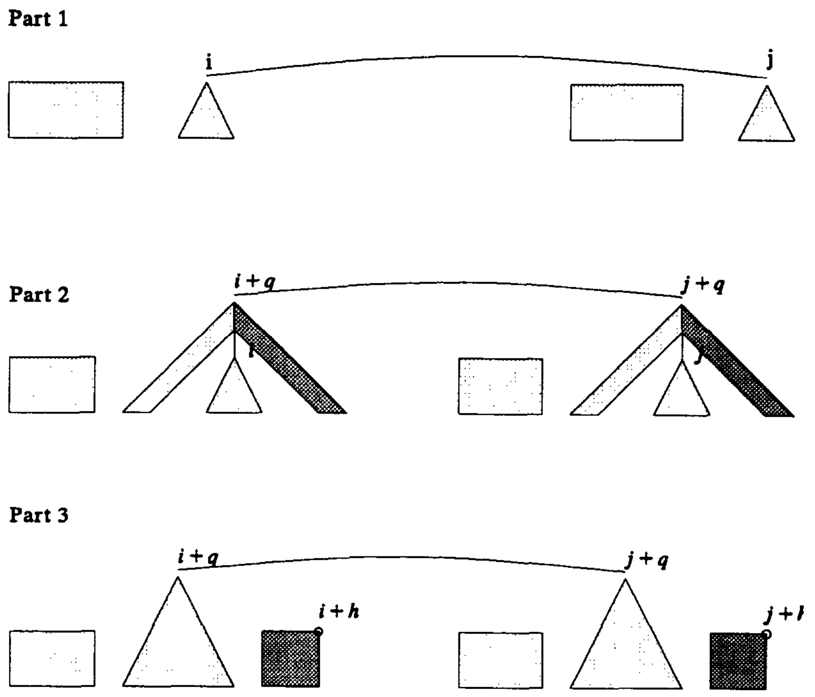

FIG. 13. Three parts to the basic jump.

Let $d = j - i$. Perform a binary search in row d of the f-array to find the smallest p such that $i \leq f(d, p)$. Then $forestdist(i, j) = p$. Since the f-array can never have more than k columns when evaluating (T_1, T_2, k), the time to access this array is $O(\log k)$ time.

6.2. *One Stage of the Algorithm*

The two algorithms we are about to introduce execute in parallel to evaluate (T_1, T_2, p) for some distance value p. (Recall that (T_1, T_2, p) gives the distance if that distance is no greater than p, otherwise it returns false.) These are optimal speedup parallel algorithms, which is why we do not present their sequential counterparts.

Both algorithms consist of stages. At each stage k, the algorithm determines for each diagonal d the maximum i such that $forestdist(i, i + d) \leq k$. We describe here the three parts that make up the kth stage along one diagonal d. See Fig. 13.

Part 1 finds an i and $j = i + d$ such that $forestdist(i, j) = k$ and (i, j) must be in any mapping where this is true. If no such i exists then stage k

is over for this diagonal. Part 2 determines the maximum ancestors $T_1[i+q]$ (of $T_1[i]$) and $T_2[j+q]$ (of $T_2[j]$) such that $\mathit{forestdist}(i+q, j+q) = k$. Part 3 then determines the maximum h such that $\mathit{forestdist}(i+h, j+h) = k$, using a left-to-right postorder suffix tree. The algorithms only differ in their implementation of part 2.

When this stage begins $f(d', k-1)$ is known for $-(k-1) \le d' \le k-1$. Also, for any tree pair i, j such that $|i-j| \le p$ (the relevant ones), we know the analogous quantity $g(i,j,d,h)$. That is, the maximum row value r such that $\mathit{dist}(l(i)\dots r, l(j)\dots r + d(l(i) - l(j))) \le h$, where $h = \min(k-1, \mathit{allow}(p,i,j))$. For convenience, we only consider the step in computing the jump in main distance array, i.e., the temporary array constructed when computing the distance between all of T_1 and all of T_2. Before presenting the algorithms, we explain the "check if" construct and the "unknown" test.

Remark 1. The algorithm makes use of statements of the form

$$\text{check if } \mathit{treedist}(x,y) = k - \mathit{forestdist}(l(x)-1, l(y)-1).$$

In these cases, we will know that $\mathit{forestdist}(l(x)-1, l(y)-1) > 0$. Since the algorithm is in stage k, either $\mathit{treedist}(x,y)$ will be unknown (and therefore $\ge k$) rendering the expression false or $\mathit{treedist}(x,y)$ will be known and the expression can be evaluated directly.

Remark 2. The algorithm also makes use of statements of the form

$$\text{if } \mathit{forestdist}(l(t+1)-1, (t+1+d)-1) \text{ is unknown so far,}$$

This implies that $\mathit{forestdist}(l(t+1)-1, l(t+1+d)-1) \ge k$. If, in addition, $\mathit{forestdist}(t, t+d) = k-1$, then $(l(t+1)-1, l(t+1+d)-1)$ is not in diagonal d by Lemma 5 (Monotonicity). (Otherwise, $\mathit{forestdist}(l(t+1)-1, l(t+1+d)-1) \le k$.) So, $t - l(t+1) \ne t + d - l(t+1+d)$. Hence, $\mathit{treedist}(t+1, t+d+1) > 0$. Therefore, $\mathit{treedist}(t+1, t+d+1) + \mathit{forestdist}(l(t+1)-1, l(t+1+d)-1) > k$.

For a given k and each diagonal d, do the following:

Part 1. Find (i,j) in diagonal d such that $\mathit{forestdist}(i,j) = k$ and (i,j) must be in the best mapping between $\mathit{forest}(i)$ and $\mathit{forest}(j)$ (Fig. 14).

Step 1. From the three diagonals, find the initial value,

$$t := \max(f(d,k-1), f(d+1,k-1)+1, f(d-1,k-1))$$

Step 2. We know that $\mathit{forestdist}(t, t+d) \le k$. Check if $(t+1, t+d+1)$ is in the best mapping and $\mathit{forestdist}(t+1, t+d+1) = k$.

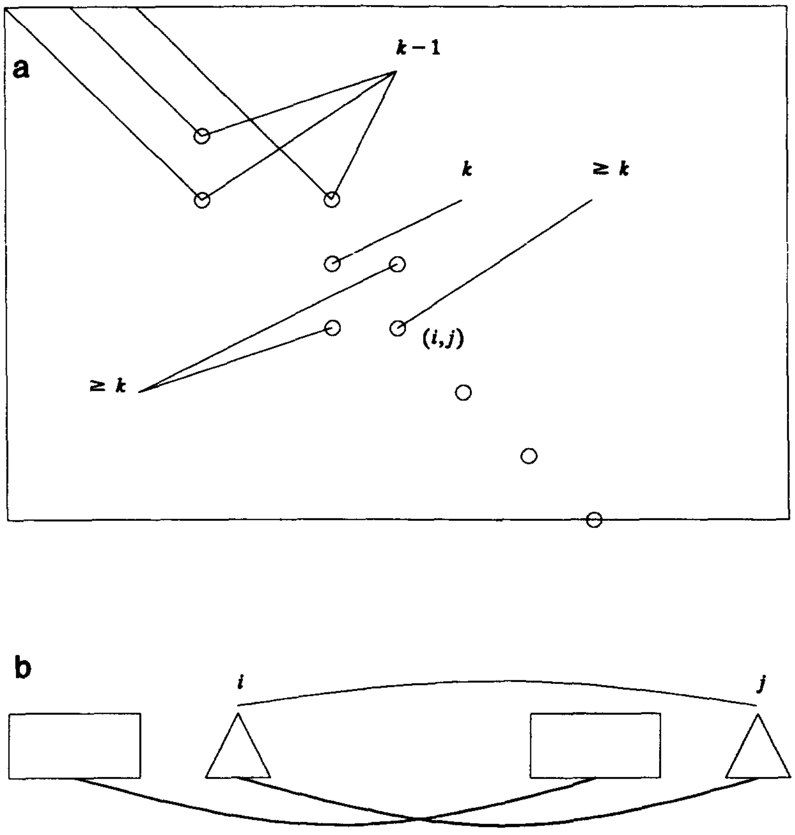

FIG. 14. Part 1 of the three basic parts to the jump; *forestdist*$(i, j) = k$ iff the best mapping is of form (b).

2a. If *forestdist*$(l(t+1)-1, l(t+1+d)-1)$ is unknown, $f(d,k) = t$

2b. If *forestdist*$(l(t+1)-1, l(t+1+d)-1) > 0$, check if *treedist*$(t+1, t+1+d) = k -$ *forestdist*$(l(t+1)-1, l(t+1+d)-1)$.

if not, then $f(d,k) = t$ so exit parts 1, 2, and 3

if so, then *forestdist*$(t+1, t+1+d) = k$ and $(t+1, t+1+d)$ must be in best mapping

2c. If *forestdist*$(l(t+1)-1, l(t+1+d)-1) = 0$,

then *forestdist*$(l(t+1)\ldots t, l(t+1+d)\ldots t+d) =$ *forestdist*$(t, t+d)$ by Lemma 7 (Proper subforest). Therefore *forestdist*$(t+1, t+d+1) =$ *treedist*$(t+1, t+d+1)$.

If $forestdist(t, t + d) = k - 1$ or $forestdist(t, t + d) = k$ and $T_1[t + 1] = T_2[t + 1 + d]$
then $forestdist(t + 1, t + 1 + d) = k$ and $(t + 1, t + 1 + d)$ must be in best mapping
else $f(d, k) = t$ so exit parts 1, 2, and 3.

Part 2. Continue from $(t + 1, t + 1 + d)$, provided $forestdist(t + 1, t + d + 1) = k$. (For notational convenience, $i = t + 1$ and the $j = t + d + 1$.) In this step, we find the largest r such that $p^r(i)$ and $p^r(j)$ are in diagonal d and $forestdist(p^r(i), p^r(j)) = k$. Note that $forestdist(p^r(i), p^r(j))) = k$ if and only if $forestdist(l(p^r(i)) - 1, l(p^r(j)) - 1)) + treedist(p^r(i), p^r(j)) = k$ by Lemma 11 (Must include). There are two methods for this part: binary search or bottom up search.

Method one—Binary search. The difficult ideas of this method are in the procedure down-probe. The outer loop does a binary search among the ancestors of (i, j) in diagonal $j - i$ and calls down-probe to see if a given pair of such ancestors (i_1, j_1) has the property that $forestdist(i_1, j_1) = k$. If so, then the outer loop restarts the search from (i_1, j_1).

Outer loop for one diagonal at stage k. (In the calculation of the distances between subtrees of T_1 and T_2, $high_1$ and $high_2$ are set to the roots of those subtrees.)

```
Let $mindiff := \min(high_1 - i, high_2 - j)$
loop
$x := i + mindiff/2$, rounding up.
$i_1 := lca(x, i)$;
$j_1 := j + (i_1 - i)$;
(So, $(i_1, j_1)$ is in the same diagonal as $(i, j)$.)
if $j_1$ is not an ancestor of $j$
        then $forestdist(i_1, j_1) > k$ by Lemma 11 (Must include) and ances-
        tor condition on mappings
        else use the $Down\text{-}probe(i, j, i_1, j_1)$ algorithm to determine whether
        $forestdist(i_1, j_1) = k$ (see below).
end if
if $forestdist(i_1, j_1) = k$
        then $i := i_1$; $j := j_1$;
        else $high_1 := x$
end if
$mindiff := high_1 - i$;
        exit if $high_1 = i$ or $high_1 = i + 1$ through two iterations in this loop
        end loop
```

Each time through the loop we halve the difference $high_1 - i$. So, at most, we require $\log \min(|T_1|, |T_2|)$ calls to down-probe for any diagonal at this part.

Function down-probe(i, j, i_1, j_1). Given (i_1, j_1) in diagonal d such that i_1 is ancestor of i, j_1 is ancestor of j and $level(i_1) - level(i) = level(j_1) - level(j)$, this function determines whether $forestdist(i_1, j_1) = k$.

Recall that (i_1, j_1) must be in any mapping such that $forestdist(i_1, j_1) = k$, since (i, j) must be in any such mapping and by Lemma 11 (Must include). Therefore $forestdist(i_1, j_1) = forestdist(l(i_1) - 1, l(j_1) - 1) + treedist(i_1, j_1)$.

Case 1. If $forestdist(l(i_1) - 1, l(j_1) - 1)$ is unknown, then $forestdist(i_1, j_1) > k$. (See Remark 2.)

```
Case 2. If $forestdist(l(i_1) - 1, l(j_1) - 1) > 0$ then
check if $treedist(i_1, j_1) = k - forestdist(l(i_1) - 1, l(j_1) - 1)$;
    if not, then $forestdist(i_1, j_1) > k$
    if so, then $forestdist(i_1, j_1) = k$
Case 3. If $forestdist(l(i_1) - 1, l(j_1) - 1) = 0$ then find $down(i_1, j_1, i, j)$
= $(y, z)$

    3a. if $y = i$ and $z = j$,
        then $forestdist(y, z) = k$ and $forestdist(i_1, j_1) = k$
    3b. if $y \neq i$ and $z \neq j$
        find the proper child $y_1$ of $y$ and the proper child $z_1$ of $z$;
        use the left-to-right postorder suffix tree to see if
        $SLR_{T_1}[y_1 + 1 \ldots y - 1] = SLR_{T_2}[z_1 + 1 \ldots z - 1]$ and $T_1[y] = T_2[z]$.
      1. If not, then $forestdist(y, z) > k$ so $forestdist(i_1, j_1) > k$. (By definition of down, $(i_1, j_1)$ and $(y, z)$ are in the same diagonal. So, by Lemma 5 (Monotonicity), $forestdist(y, z) > k$ implies that $forestdist(i_1, j_1) > k$.)
      2. Suppose so. That is, $SLR_{T_1}[y_1 + 1 \ldots y - 1] = SLR_{T_2}[z_1 + 1 \ldots z - 1]$ and $T_1[y] = T2[z]$.

if $forestdist(l(y_1) - 1, l(z_1) - 1)$ is unknown
    then $forestdist(y_1, z_1) > k$ so $forestdist(i_1, j_1) > k$ by the monotonicity lemma.
    else (we know that $forestdist(l(y_1) - 1, l(z_1) - 1) > 0$ by definition of down.)
check if $treedist(y_1, z_1) = k - forestdist(l(y_1) - 1, l(z_1) - 1)$.
  if not then $forestdist(y_1, z_1) > k$, so $forestdist(i_1, j_1) > k$
  if so, then $forestdist(y_1, z_1) = k$ by definition of distance
      so, $forestdist(y, z) = k$ by the left-to-right poster traversal
      so, $forestdist(i_1, j_1) = k$ by definition of down and the quarantined subtree lemma.
```

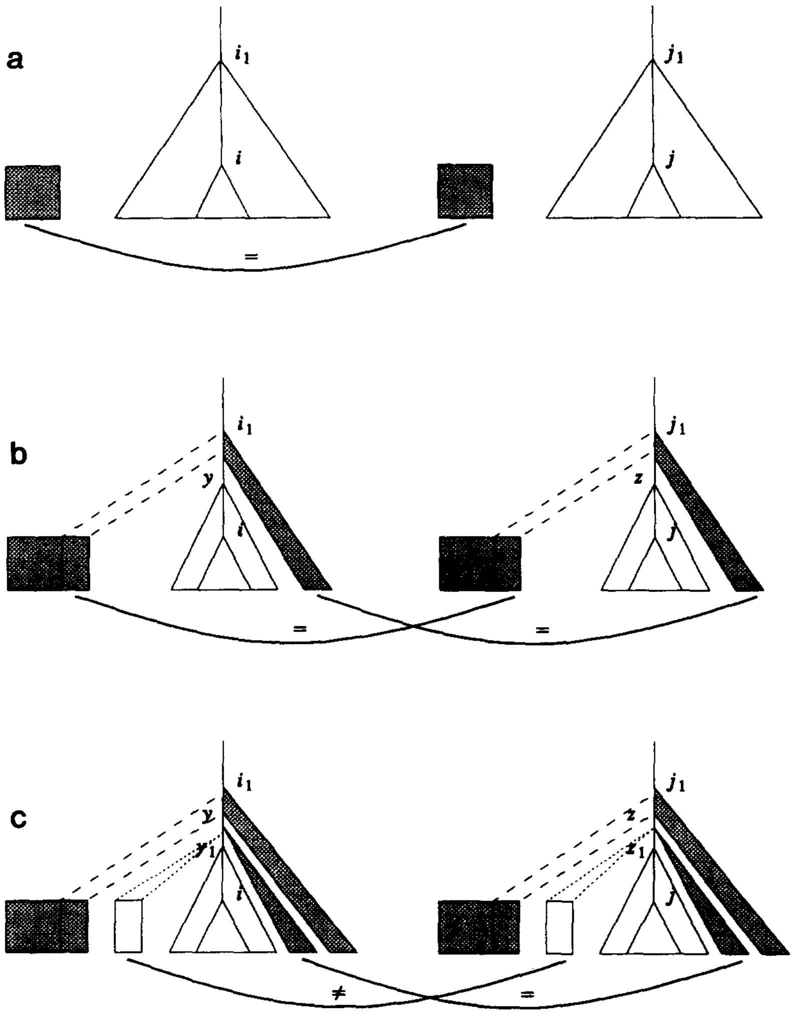

FIG. 15. Hard case for binary search approach: (a) $forestdist(l(i)-1, l(j)-1)=0$; (b) $down(i_1, j_1, i, j)=(y, z)$; (c) $y=p(y_1)$ and $z=p(z_1)$.

The hard case is the very last one (depicted in Fig. 15). In that case,

(0) $forestdist(l(i_1)-1, l(j_1)-1)=0$, $forestdist(i, j)=k$ and (i, j) must be in the mapping with cost k from $forest(i)$ to $forest(j)$.

(1) $down(i_1, j_1, i, j)=(y, z)$.

(2) y_1 is the proper child of y with respect to i.

(3) z_1 is the proper child of z with respect to j.

(4) $T_1[y_1 + 1 \dots y] = T_2[z_1 + 1 \dots z]$.

then either $forestdist(y_1, z_1) > k$ or $forestdist(i_1, j_1) = k$.

Proof. By condition (4), $y - y_1 = z - z_1$. So, by condition (1), (y, z) and (y_1, z_1) are in diagonal d. By Lemma 11 (Must include), $forestdist(y_1, z_1) = k$ implies that $forestdist(y_1, z_1) = treedist(y_1, z_1) + forestdist(l(y_1) - 1, l(z_1) - 1) = k$. If this sum is greater than k, then $forestdist(y_1, z_1) > k$.

Otherwise, we can assume $forestdist(y_1, z_1) = k$. From (4) and (1), we know that there is a non-zero distance between $T_1[l(y) \dots l(y_1) - 1]$ and $T_2[l(z) \dots l(z_1) - 1]$. By (4), (y_1, z_1) is in the same diagonal as (y, z), which in turn is in the same diagonal as (i_1, j_1) by condition (1). Now, $forestdist(l(y) - 1, l(z) - 1) = 0$ by conditions (0) and (1). So, by Lemma 7 (Proper subforest), $forestdist(y, z) = treedist(y, z)$. This implies, by the quarantined subtree lemma, that $forestdist(i_1, j_1) = treedist(y, z)$.

By conditions (0) and (4), $treedist(y, z) = treedist(y_1, z_1) + dist(l(y) \dots l(y_1) - 1, l(z) \dots l(z_1) - 1) = treedist(y_1, z_1) + forestdist(l(y_1) - 1, l(z_1) - 1)$ by Lemma 7 (Proper subforest). This is equal to $forestdist(y_1, z_1)$, since (y_1, z_1) must be in the mapping by Lemma 11 (Must include). So, if $k = forestdist(y_1, z_1)$ then $treedist(y, z) = forestdist(i_1, j_1) = k$ as well. □

Method two—Repeated hops. Given (i_1, j_1) in diagonal d such that i_1 is ancestor of i, j_1 is ancestor of j and $level(i_1) - level(i) = level(j_1) - level(j)$, we want to determine whether $forestdist(i_1, j_1) = k$. This is a bottom-up approach in which the loop may be repeated k times for a given diagonal at stage k as we show in Lemma 17. This method does not require suffix trees for preorder traversals.

As in method one, by Lemma 11 (Must include), (i_1, j_1) must be in any mapping such that $forestdist(i_1, j_1) = k$. So, $forestdist(i_1, j_1) = forestdist(l(i_1) - 1, l(j_1) - 1) + treedist(i_1, j_1)$.

When we say $return(x, y)$ in this method, we mean that (x, y) are the ancestors of (i, j) that Part 2 should return, so it implies an exit of the loop.

loop
find $up(i, j) = (i_2, j_2)$. From Lemma 7 (Proper subforest) and Lemma 12 (Continuation condition), we know that $forestdist(i_2, j_2) = k$.

```
Let $i_1 = p(i_2)$ and $j_1 = p(j_2)$.
Use the left-to-right postorder suffix tree to see if
  $SLR_{T_1}[i_2 + 1 \ldots i_1 - 1] = SLR_{T_2}[j_2 + 1 \ldots j_1 - 1]$ and $T_1[i_1] = T2[j_1]$.
  1. If not, then $forestdist(i_1, j_1) > k$ so $return(i_2, j_2)$
  2. If so, check
    2a. If $forestdist(l(i_1) - 1, l(j_1) - 1)$ is unknown, then
        $forestdist(i_1, j_1) > k$ so $return(i_2, j_2)$.
    2b. If $forestdist(l(i_1) - 1, l(j_1) - 1) > 0$.
        check if $treedist(i_1, j_1) = k - forestdist(l(i_1) - 1, l(j_1) - 1)$;
        if not, then $forestdist(i_1, j_1) > k$ so $return(i_2, j_2)$
        if so, then $forestdist(i_1, j_1) = k$, $i := i_1$, $j := j_1$.
    2c. If $forestdist(l(i_1) - 1, l(j_1) - 1) = 0$ then
        $forestdist(i_1, j_1) = k$, $i := i_1$, $j := j_1$.
end loop.
```

The last case (2c) is the subtle one.

LEMMA 16. *If*

0. $up(i, j) = (i_2, j_2)$, $i_1 = p(i_2)$, $j_1 = p(j_2)$.

1. $forestdist(i, j) = k$ and (i, j) must be in the mapping with cost k from $forest(i)$ to $forest(j)$.

2. $SLR_{T_1}[i_2 + 1 \ldots i_1 - 1] = SLR_{T_2}[j_2 + 1 \ldots j_1 - 1]$ and $T_1[i_1] = T2[j_1]$.

3. $forestdist(l(i_1) - 1, l(j_1) - 1) = 0$.

then $forestdist(i_1, j_1) = k$ (see Fig. 16).

Proof. By Lemma 8 (Quarantined subtree) and condition 0, $treedist(i_2, j_2) = treedist(i, j)$. By condition 1, $k = forestdist(i, j) = treedist(i, j) + forestdist(l(i) - 1, l(j) - 1)$. By Lemma 7 (Proper subforest) and condition 0, $forestdist(l(i) - 1, l(j) - 1) = forestdist(l(i_2) - 1, l(j_2) - 1)$. Putting this together, $k = treedist(i_2, j_2) + forestdist(l(i_2) - 1, l(j_2) - 1)$. So, $forestdist(i_2, j_2) = k$.

By condition 3 and Lemma 7 (Proper subforest), $forestdist(l(i_2) - 1, l(j_2) - 1) = dist(l(i_1) \ldots l(i_2) - 1, l(j_1) \ldots l(j_2) - 1)$. By condition 2, $treedist(i_1, j_1) = treedist(i_2, j_2) + dist(l(i_1) \ldots l(i_2) - 1, l(j_1) \ldots l(j_2) - 1)$. This sums to k by the argument of the last paragraph. By condition 3 and the Lemma 7 (Proper subforest), this implies that $forestdist(i_1, j_1) = k$. □

Part 3. At this point we have found the largest r such that $p^r(i)$ and $p^r(j)$ are in diagonal d and $forestdist(p^r(i), p^r(j)) = k$. Now, we use a left-to-right suffix tree to find the $f(d, k)$.

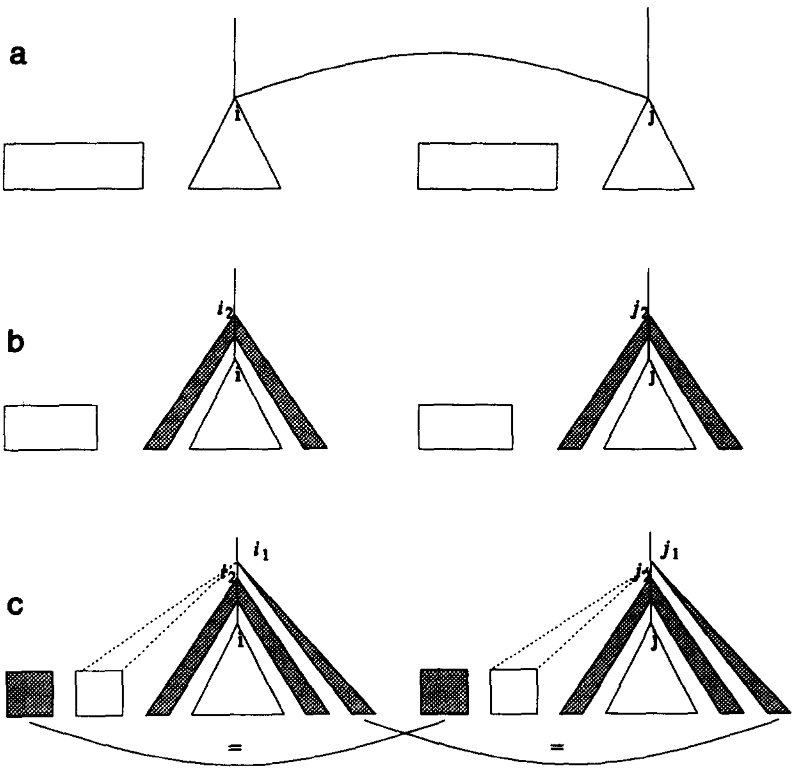

FIG. 16. Hard case for bottom-up approach: (a) (i, j) must in best mapping; (b) $up(i, j) = (i_2, j_2)$; (c) $p(i_2) = i_1$ and $p(j_2) = j_1$.

Let $i = p^r(i)$ and $j = p^r(j)$. Let h be such that $SLR_{T_1}[i \ldots i + h] = SLR_{T_2}[j \ldots j + h]$ and $SLR_{T_1}[i + h] \neq SLR_{T_2}[j + h]$. So, $f(d, k) = i + h$.

Parts 1 and 3 take constant time. So, the only question is how long part 2 takes. In the case of the first method, the $\log_2 N$ probes are shown with the algorithm for the outer loop. Part 2 of the second method is still at issue.

LEMMA 17. *The loop of part* 2 *of the second method will be repeated at most k times.*

Proof. We will show that *treedist*(i, j) will be increased by at least one in the course of each execution in which *forestdist*$(i_1, j_1) = k$. Clearly, this can happen at most k times. Since $up(i, j) = (i_2, j_2)$, we know that

$treedist(i_2, j_2) = treedist(i, j)$ by Lemma 8 (Quarantined subtree). Since $i_1 = p(i_2)$ and $j_1 = p(i_2)$, and $SLR_{T_1}[i_2 + 1 \ldots i_1 - 1] = SLR_{T_2}[j_2 + 1 \ldots j_1 - 1]$ and $T_1[i_1] = T_2[j_1]$, we know that $dist(l(i_1) \ldots l(i_2) - 1, l(j_1) \ldots l(j_2) - 1) > 0$. By Lemma 11 (Must include), (i_2, j_2) and (i_1, j_1) must both be in the mapping between $forest(i_1)$ and $forest(j_1)$, therefore $T_1[l(i_1) \ldots l(i_2) - 1]$ is mapped to $T_2[l(j_1) \ldots l(j_2) - 1]$. So, $treedist(i_1, j_1) \geq dist(l(i_1) \ldots l(i_2) - 1, l(j_1) \ldots l(j_2) - 1) + treedist(i_2, j_2) > treedist(i, j)$. □

7. Overall Resource Analysis of Algorithms 2 and 3

The algorithm has the following structure:

```
for $s := 0$ to $\log_2 N$
    if $(T_1, T_2, 2^s)$ is answered affirmatively then exit and report distance
end for
```

So, at any given stage in this algorithm, we are asking a question of the form (T_1, T_2, p); i.e., give the distance between T_1 and T_2 if no greater than p else return false.

Proposition Resources. Suppose a (T_1, T_2, p) requires $T(p)$ resources (time, space, processors) to answer, where $T(p)/p$ is a nondecreasing function. Then if the final distance is k, the algorithm requires $O(T(2k))$ resources.

Proof. Suppose $2^{c-1} < k \leq 2^c$. Then the total time for this algorithm is $\Sigma_{s=0}^{c} T(2^s) \leq \Sigma_{s=0}^{c}(1/2^s)T(2^c)$, since $T(2^s) \leq 2^s \times T(2^c)/2^c$. This sum is bounded from above by $2 \times T(2^c)$. This shows that the algorithm requires $O(T(2k))$ resources. □

Fact. For any of the complexity measures we derive $T(2k) = cT(k)$ (though the constant c differs). So, we simply address the problem of computing (T_1, T_2, k). Let N be $\min(|T_1|, |T_2|)$.

Time complexity. The two algorithms can be characterized as follows for the problem of computing (T_1, T_2, k). The binary search algorithm requires at most $O(\log N)$ probes. Each one requires $O(\log k)$ time to determine forest distances from the f-array. For each diagonal this must be done at most k times. So, the total time is $k \log k \log N$.

The repeated hop algorithm of Method Two requires at most k probes each time the distance is increased by Lemma 17. This happens k times along the diagonal, so the total time is $k^2 \log k$. Thus, Method Two should be used if $k \leq \log N$.

The time complexity to construct the suffix tree is $O(\log(|T_1| + |T_2|))$, because the alphabet is polynomial in the size of the two trees. Since $O(\log(|T_1| + |T_2|)) \leq O(\log(2 \times N + k)) \leq O(\log(N) + \log(k))$, the time complexity to construct the suffix tree is $\leq O(\log(N) + \log(k))$. Hence the time complexity for binary search algorithm is $O(k \log(k)\log(N))$ and the time complexity for repeated hop algorithm is $O(k^2 \log(k) + \log(N))$.

For purposes of processor and space complexity, we must count the tree distances that are relevant. For a given k, we only need to compute $(tree_1(i), tree_2(j), allow(k, i, j))$ such that $allow(k, i, j) \geq 0$ and such that $|i - j| \leq k$.

Here is why. When computing $(tree_1(i), tree_2(j), h)$, the algorithm only calls $(tree_1(i'), trees2(j'), h')$ if the latter is relevant to the former. Therefore the only treedistance triples ever called are relevant to (T_1, T_2, k). Hence the only triples called are of the form $(tree_1(i), tree_2(j), h)$, where $h < allow(k, i, j)$, $allow(k, i, j) \geq 0$, and such that $|i - j| \leq k$. Since $(tree_1(i), tree_2(j), h)$ is covered by $(tree_1(i), tree_2(j), allow(k, i, j))$, the above is enough.

Processor complexity. There are $O(k \times N)$ relevant subtree-to-subtree distances. In each subtree-to-subtree distance computation, only $2k + 1$ diagonals (at most) are of interest. Each diagonal needs a processor. So $O(k^2N)$ processors are enough. (This assumes that whenever we want a processor, we just use it and more processors can be added at will. In the real world of few processors, a run time queue simulates this situation.) The processor complexity for construct suffix tree is $O(2 \times N + k)$. Hence the processor complexity is $O(k^2N)$.

Space complexity. Each tree needs space k^2. There are $k \times N$ trees. So $O(k^3N)$ space is enough. For the construction of suffix trees, we need $O((2N + k)^2)$ space, by [AILSV88]. Hence the space complexity is $O(k^3N + N^2)$.

Preprocessing costs. Preprocessing costs are dominated by the cost of constructing the suffix trees. Those costs were described in the last two paragraphs.

8. Conclusion

We have presented some fast algorithms for comparing two trees. The algorithms use several suffix trees and introduce hopping and binary search techniques as methods for speeding up dynamic programming. We have implemented the simple algorithm as a toolkit which is available from the authors.

Acknowledgments

The authors thank Gadi M. Landau, Richard Cole, Robert Hummel, and Micha Sharir for their helpful comments. The authors also thank the anonymous referees for their many helpful suggestions.

References

[AILSV88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica* **3** (1988), 347–365.

[ALKBO87] S. Alluvia, H. Locker-Giladi, S. Koby, O. Ben-Nun, and A. B. Oppenheim, RNase III stimulates the translations of the cIII gene of bacteriophage lambda, *Proc. Nat. Acad. Sci. U.S.A.* **85**, (1987), 1–5.

[BSSBWD87] B. Berkout, B. F. Schmidt, A. Strien, J. Boom, J. Westrenen, and J. Duin, Lysis gene on bacteriophage MS2 is activated by translation termination at the overlapping coat gene, *Proc. Nat. Acad. Sci. U.S.A.* **195** (1987), 517–524.

[CS85] M. T. Chen and J. Seiferas, Efficient and elegant subword tree construction, *in* "Combinatorial Algorithms on Words," A. Apostolico and Z. Galil, Ed., NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, pp. 97–107, Springer-Verlag, New York/Berlin, 1985.

[DA82] N. Delihas and J. Anderson, Generalized structures of 5s ribonsomal RNA's, *Nucleic Acid Res.* 10 (1982), p. 7323.

[DD87] I. C. Deckman and D. E. Draper, S4-alpha mRNA translation regulation complex, *J. Mol. Biol.* **196** (1987), 323–332.

[G84] Z. Galil, Optimal parallel algorithms for string matching, *in* "Proceedings, 16th ACM Symposium on Theory of Computing 1984," pp. 240–248; *Inform. and Control* **67** (1985), 144–157.

[GG86a] Z. Galil and R. Giancarlo, Improved string matching with k mismatches, *SIGACT News* **17**, No. 4 (1986), 52–54.

[GG86b] Z. Galil and R. Giancarlo, Parallel string matching with k mismatches, *Theoret. Comput. Sci.* **51**, 341–348.

[GS83] Z. Galil and J. I. Seiferas, Time-space-optimal string matching, *J. Comput. System Sci.* **26** (1983), 280–294.

[HT84] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13**, No. 2 (1984), 338–355.

[Knuth] D. E. Knuth, "The Art of Computer Programming," Vol. I, Addison-Wesley, Reading, MA, 1974.

[L79] S. Y. Lu, A tree-to-tree distance and its application to cluster analysis, *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-1**, No. 2 (1979), 219–224.

[LV86] G. M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching, *in* "Proceedings 18th ACM Symposium on Theory of Computing, 1986," pp. 220–230.

[S88] B. A. Shapiro, An algorithm for comparing multiple RNA secondary structures, *Comput. Appl. Biosci.* in press; Image Processing Section, Frederick Cancer Research Facility, Bldg. 469, Room 150, Frederick, MD 21701.

[SK76] J. L. Sussman and S. H. Kim, Three dimensional structure of a transfer RNA in two crystal forms, *Science* **192** (1976), 853.

[SK83] D. Sankoff and J. B. Kruskal (Ed.), "Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison," Addison-Wesley, Reading, MA, 1983.

[SV88] B. Schieber and U. Vishkin, "Parallel Computation of Lowest Common Ancestor in Trees," NYU Computer Science Technical Report.

[Tai79] K.-C. Tai, The tree-to-tree correction problem *J. Assoc. Comput. Mach.* **26** (1979), 422–433.

[TV85] R. E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, No. 4, 1985.

[U83] E. Ukkonen, On approximate string matching, *in* "Proceedings, Int. Conf. Found. Comput. Theory, Lecture Notes in Computer Science," Vol. 158, pp. 487–495, Springer-Verlag, New York/Berlin, 1983.

[ZS89] K. Zhang and D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, *SIAM J. Comput.* **18**, No. 6, (1989), 1245–1262.

[Z89] K. Zhang, "The Editing Distance between Trees: Algorithms and Applications," Ph.D. thesis, Courant Institute of Mathematical Sciences, New York University, 1989.