Fast Parallel Algorithms for the Unit Cost Editing Distance Between Trees (extended abstract)

Dennis Shasha, shasha@nyu.edu
Kaizhong Zhang, zhang@nyu.acf8
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012 U.S.A.†

1. Problem Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. 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.

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 (T79 in JACM). The second and third algorithms are parallel algorithms based on the application 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
Alg1	$\|T_1\| \times \|T_2\| \times L_1 \times L_2$	1
Alg1 parallel	$\|T_1\| + \|T_2\|$	$\|T_1\| \times \|T_2\| \times min(L_1, L_2)$
Alg2 parallel	$k \times log(k) \times log(N)$	$k^2 \times N$
Alg3 parallel	$(k^2 \times log(k)) + log(N)$	$k^2 \times N$

† This work was partially supported by the Office of Naval Research under grant number N00014-85-K-0046.

Application Significance 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 some of these algorithms has been used by researchers at the National Cancer Institute. 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, BP87] 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 RNA's.

Existing methods for comparing the secondary structures of two RNA's take a traversal ordering of the two trees and discover the string edit distance between the orderings [S88]. That is unsatisfactory since a traversal ordering does not uniquely specify a tree. The tree edit distance is clearly a better metric.

For all the applications, differences are most significant for small values of k, since trees that differ by more than a certain threshhold are for practical purposes simply different.

Algorithmic Significance We use the Ukkonen [U83] idea of computing in waves along the center diagonals of the distance matrix. At the beginning of stage k, all the distances up to $k-1$ have been computed. Stage k then computes in parallel all the

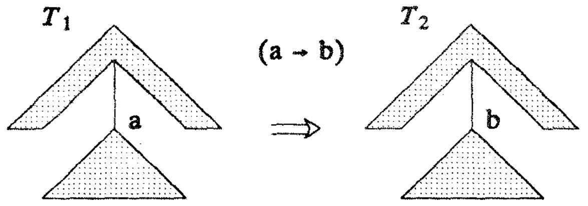


Figure 1. Relabeling

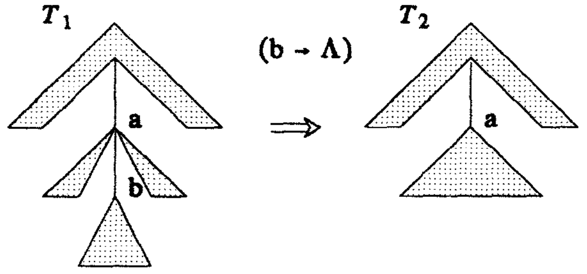


Figure 2. Deletion

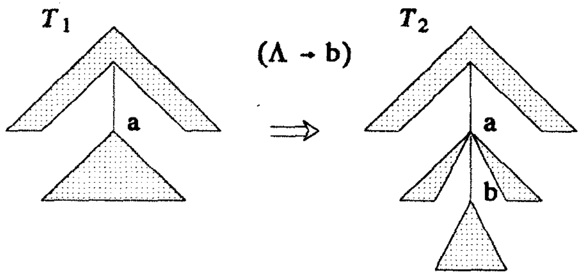


Figure 3. Insertion

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 $S_1[i..i+h]=S_2[j..j+h]$, then the distance between $S_1[1..i-1]$ and $S_2[1..j-1]$ is the same as between $S_1[1..i+h]$ and $S_2[1..j+h]$. 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 5).

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.

We represent an edit operation [T79, ZS87] 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 $1 \leq i \leq k$.

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. By extension, the cost of a sequence is simply the length of 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.

2.1. 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 Figure 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)).

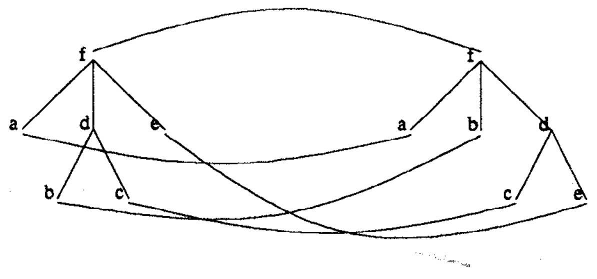


Figure 4. Edit Sequence

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 Figure 5):

(1) $1 \le i \le N_1$, $1 \le j \le N_2$;
(2) For any pair of (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. 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).

Lemma 1: 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 $\gamma(M) \le \gamma(S)$. 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 $T_2\}$

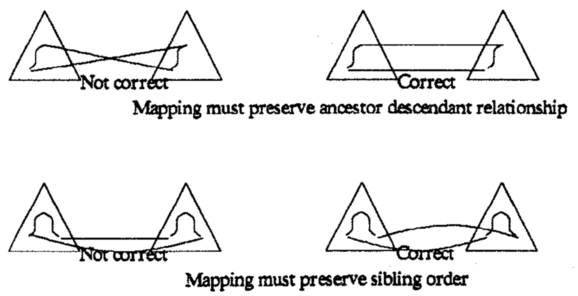

Not correct
Correct
Mapping must preserve ancestor descendant relationship
Not correct
Correct
Mapping must preserve sibling order

Figure 5. Mapping rules

2.2. 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 descendant of the subtree rooted at T[i]. When T[i] is a leaf, $l(i) = i$.

T[i..j] is the ordered subforest of *T* induced by the nodes numbered i to j inclusive (Figure 6). *T*[1..i] will be referred to as *forest(i),* when the tree 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_1[i'..i] and T_2[j'..j] is denoted dist(T_1[i'..i], T_2[j'..j]) or dist(i'..i, j'..j) if the context is clear. We use a more abbreviated notation for certain special cases. The distance between T_1[1 .. i] and T_2[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)*.

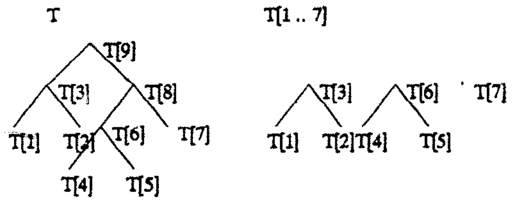


Figure 6. Postorder T[1 .. 7] = forest(7)

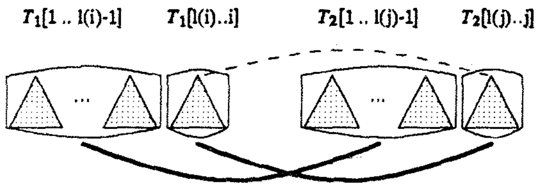


Figure 7. Case 3. holds when (i,j) is in mapping

3. Basic Algorithm

We compute forestdist(i,j) for $1 \le i \le N_1$ and $1 \le j \le N_2$. 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 (Figure 7). By the ancestor and sibling conditions on mappings, (i,j) must be in M. By the 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, $forestdist(i,j) = forestdist(l(i)-1, l(j)-1) + dist(T_1[l(i)..i-1], T_2[l(j)..j-1]) + \gamma(T_1[i] \to T_2[j])$. When either $l(i) \neq$ leftmost child of T_1 or $l(j) \neq$ leftmost child of T_2, we can use the equation $forestdist(i,j) = forestdist(l(i)-1, l(j)-1) + treedist(i,j)$.

These three cases specify a step of a simple dynamic programming algorithm. Because of case 3, any subtree-to-subtree distance may be required. So, the time complexity is $O(\sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_2|} size(i) \times size(j)) = O(|T_1| \times |T_2| \times L_1 \times L_2)$.

4. Improving the simple algorithm

4.1. Review of Landau-Vishkin algorithm

In the following discussion, diagonal d corresponds to the the set of distances {stringdist(i,j) | i − j = d}. (The name diagonal comes from the distance matrix in the naive dynamic programming algorithm.)

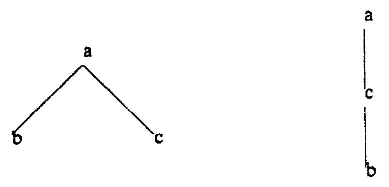


Figure 8a. Different Trees May have the Same Postorder Traversal (Here, bca)

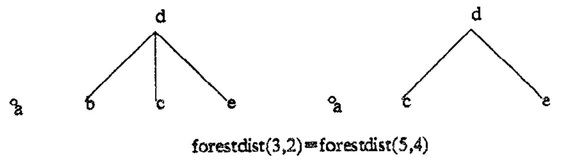


Figure 8b. Label with Number of Children Seems not Necessary (Even though d has a different number of children, editing operation is "delete b")

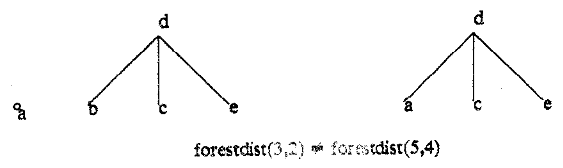


Figure 8c. Label with Number of Children Seems not Sufficient (even though both have traversal with children sequence {c,0}, {e,0}, {d,3})

The basic algorithm of [LV86] is

```
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 $S_1[i..i+h]=S_2[i+d..i+d+h]$.

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.

4.2. 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 Figure 8a shows. On the other hand, it is well known [Knuth vol. 1, p. 350] 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.

Problem 2: Identical traversals with children are not necessary. That is, forestdist(i,j) = forestdist(i+h, j+h) is possible even though $SLR_1[i+1..i+h] \neq SLR_2[j+1..j+h]$. See Figure 8b.

Problem 3: Identical traversals with children are not sufficient. That is, forestdist(i,j) < forestdist(i+h, j+h) is possible even though $SLR_1[i+1..i+h]=SLR_2[j+1..j+h]$. See Figure 8c.

So, what are these traversals good for? -- we hear you cry. Well, if the single node labeled *e* in Figure 8b or in 8c were replaced by a tree (or even forest) of size *r*, then in both cases $forestdist(3,2)=forestdist(3+r,2+r)$ and this would be discovered by establishing that $SLR_1[3+1..3+r]=SLR_2[2+1..2+r]$.

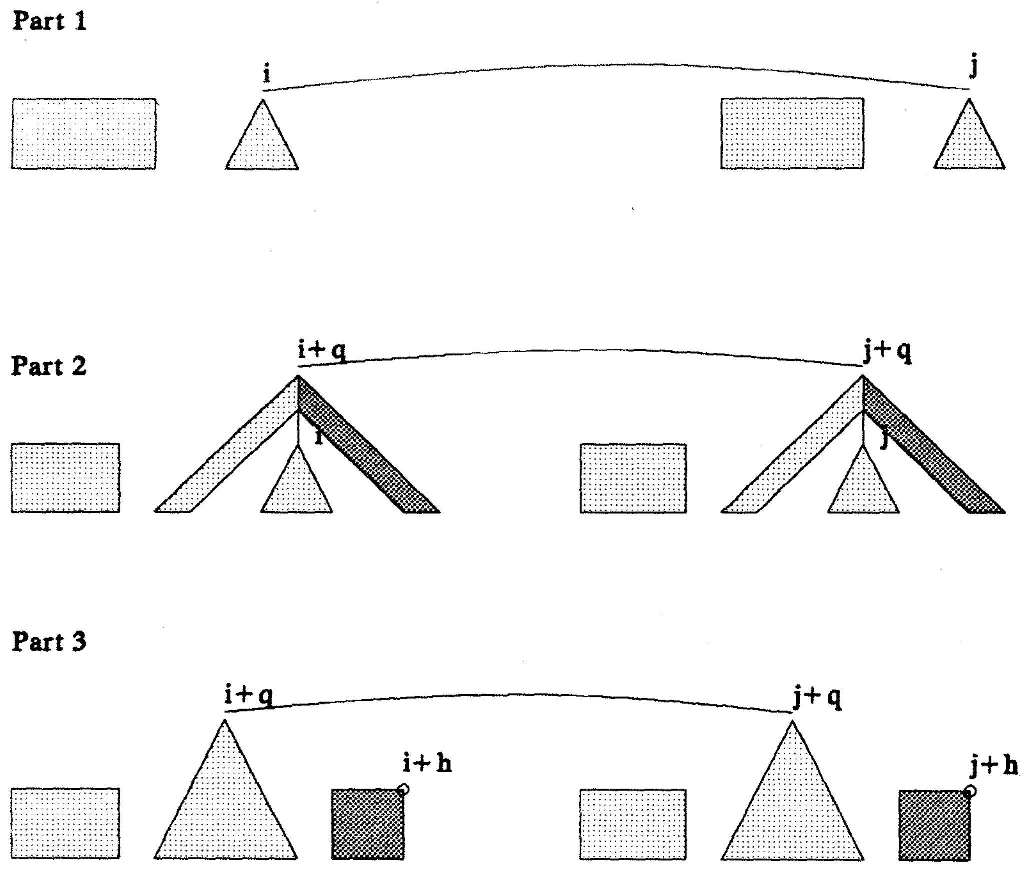


Figure 9. Three Parts to Basic Jump
(goal is to find largest h usch that forest(i+h, j+h)=forest(i,j))

4.3. Overview of Our improved algorithm

Having discussed these problems, we will now see how our algorithm deals with them. Figure 9 shows the three parts of the basic jump along one diagonal. Parts I and III are analogous to the string case, whereas part II requires special attention. Our algorithms differ in how they perform part II. We present only algorithm 3's approach, because algorithm 2, which uses binary search in part II is slightly more complex.

Part I finds the first i such that $(i,i+d)$ must be in any mapping such that $forestdist(i,i+d)=k$. In that case, we let $j=i+d$. If no such i exists then stage k is over for this diagonal.

Part II determines the maximum ancestors $T_1[i+q]$ (of $T_1[i]$) and $T_2[j+q]$ (of $T_2[j]$) such that $forestdist(i+q,j+q)=k$.

Part III then determines the maximum h such that $forestdist(i+h,j+h)=k$, using a left-to-right postorder suffix tree.

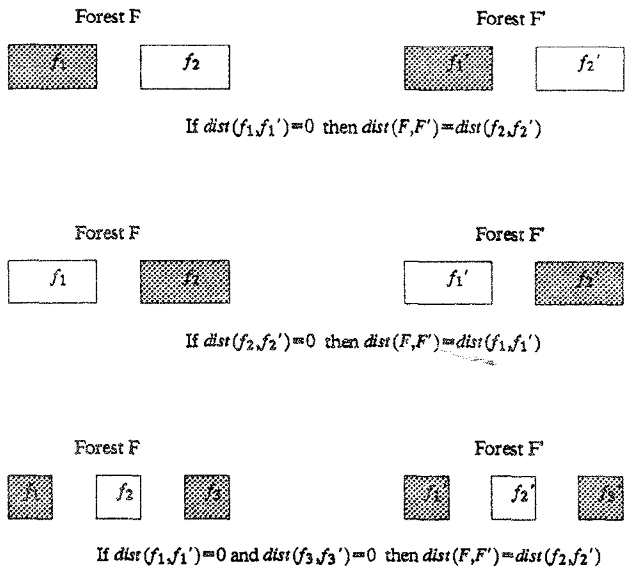

Forest F
Forest F'
f_1
f_2
f_1'
f_2'
If $dist(f_1,f_1')=0$ then $dist(F,F')=dist(f_2,f_2')$
If $dist(f_2,f_2')=0$ then $dist(F,F')=dist(f_1,f_1')$
f_3
f_3'
If $dist(f_1,f_1')=0$ and $dist(f_3,f_3')=0$ then $dist(F,F')=dist(f_2,f_2')$

Figure 10. Proper Forest Lemmas

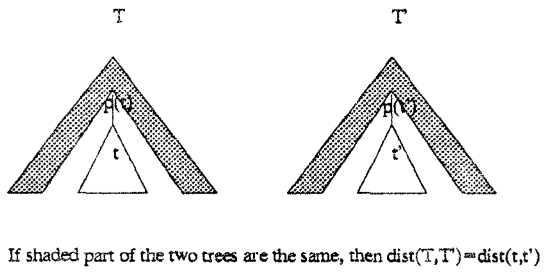

T
T'
t
t'
If shaded part of the two trees are the same, then dist(T,T')=dist(t,t')

Figure 11. Quarantined Subtree Lemma

4.3.1. Doing part II

One particularly difficult problem in part II is that determining that $forestdist(i_1,j_1)=k$ may require knowing that $treedist(i_1,j_1)=k$. Our ability to determine that fact without waiting depends on the following definitions and lemmas.

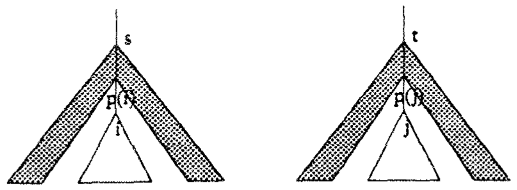
Definition: Given forest F, we say that F[i..j] is a *proper forest* of F if the subgraph induced by the nodes i through j in the post-order numbering of F has the following property: if n is in F[i..j], then all children of n in the tree F are in F[i..j].

Lemma 2 (two sided proper forest): Suppose $F_1[1..m]$ and $F_2[1..n]$ are ordered forests, $F_1[1..x]$ is a proper forest of F_1, $F_1[x+1..y-1]$ is a proper forest of F_1, $F_2[1..x]$ is a proper forest of F_2. and $F_2[x+1..z-1]$ is a proper forest of F_2. If $F_1[1..x]$ is identical to $F_2[1..x]$ and $F_1[y..m]$ is identical to $F_2[z..n]$. then dist(1..m, 1..n) = dist(x+1..y-1, x+1..z-1). (See Figure 10.) □

We now propose the analogous property for trees.

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

Lemma 3 (Quarantined Subtree) If the only difference between T_1 and T_2 is between tree(i) and tree(j) then $dist(1..m,1..n)=dist(l(i)..i,l(j)..j)$. (Figure 11.)



up(i,j)=(s,t) shaded part of the two trees are the same

Figure 12. up(i,j)

5. Algorithm

To do part II fast, we use a predicate *up* obtained by one application of suffix trees on the left-to-right and right-to-left postorder traversals.

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 (Figure 12):

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). □

Figure 13(b) shows the application of up(i,j). Notice that the right-to-left postorder suffix tree will examine forests to the left of $tree_1$(i) and $tree_2$(j) as well as ancestors of *i* and *j*.

5.1. Bottom up algorithm for part II

Part I has established the condition in Figure 13(a).

Start Find up(i,j)=(i_2,j_2). Using the proper forest and quarantined subtree lemmas, we know that $forestdist(i_2,j_2)=k$. Let $i_1=parent(i_2)$ and $j_1=parent(j_2)$. The hard question is whether $forestdist(i_1,j_1)=k$.

This is only possible if $T_1[i_2+1..i_1-1]=T_2[j_2+1..j_1-1]$ and $T_1[i_1]=T_2[j_1]$. (That can be determined by one application of a suffix tree.) Otherwise (i_2,j_2) is the answer to part II.

If so, there are three cases.

(1) If $forestdist(l(i_1)-1,l(j_1)-1)$ has not been computed up to stage k − 1, then $forestdist(i_1,j_1)>k$ so (i_2,j_2) is the answer to part II.

(2) If $forestdist(l(i_1)-1,l(j_1)-1)>0$, then we can check whether $treedist(i_1,j_1)=k-forestdist(l(i_1)-1,l(j_1)-1)$. If not, then $forestdist(i_1,j_1)>k$ so (i_2,j_2) is the answer to part II. If so, then $forestdist(i_1,j_1)=k$ and go to *Start* setting $i=i_1$ and $j=j_1$.

(3) If $forestdist(l(i_1)-1,l(j_1)-1)=0$ then $forestdist(i_1,j_1)=k$, so go to *Start* setting $i=i_1$ and $j=j_1$.

Case 1 holds because $treedist(i_1,j_1)>0$ by definition of predicate up. Case 2 poses no difficulty either except as far as running time is concerned. In both case 2 and case 3 if $forestdist(i_1,j_1)=k$, then $treedist(i_1,j_1)>treedist(i,j)$, so we can return to *Start* only at most *k* times at stage *k*.

The third case is quite subtle, involving all the machinery we have presented so far. It may help the reader to refer to Figure 13(c).

Lemma 4 (bottom up hop): 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. $T_1[i_2+1..i_1-1]=T_2[j_2+1..j_1-1]$ and $T_1[i_1]=T_2[j_1]$.
3. $forestdist(l(i_1)-1,l(j_1)-1)=0$.

then $forestdist(i_1,j_1)=k$.

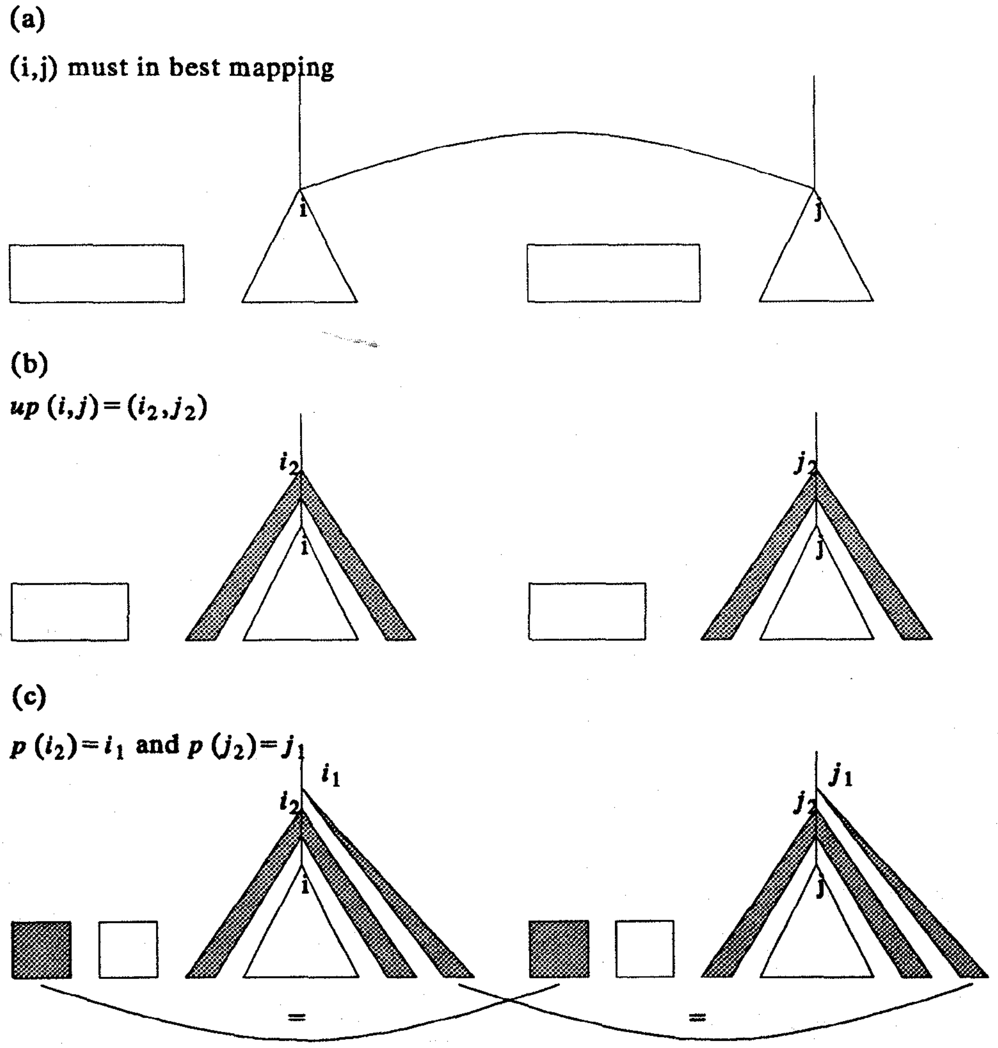


Figure 13. Hard Case for Bottom-up Approach

Proof: By lemma 3 (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 2 (proper forest) 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 2 (proper forest), $forestdist(l(i_2)-1,l(j_2)-1)=dist(l(i_1)..l(i_2)-1,l(j_1)..l(j_2)-1)$. By condition 2, $treedist(i_1,j_1)=treedist(i_2,j_2)+dist(l(i_1)..l(i_2)-1,l(j_1)..l(j_2)-1)=k$. By condition 3, this implies that $forestdist(i_1,j_1)=k$. □

5.2. Timing Analysis

As mentioned above, we may return to *Start* at most k times at stage k. There are only k stages if the final distance between the two trees is k. The extra factor of log k as shown in the first table is due to the fact that we do not store all of forestdist(i,j). Instead, we just store points where the distance changes in an array f. $f(d,p)$ is the maximum row number r such that the intersection between diagonal d and row $r+1$ holds a number that is greater than p, but row r holds a number that is less than or equal to p in the distance array. To determine forestdist(i,j), we use binary search, requiring $O(log\ k)$ time.

So, the total time is $O(k^2 \log k)$. Finally, the additive $O(log\ N)$ factor is required to build the suffix tree.

5.3. Processors

To determine whether two T_1 and T_2 are distance k apart or less, hereafter called the ***within k distance problem*** it is only necessary to evaluate treedist(i,j) if $|i-j| \leq k$. There are $k \times N$ such subtree pairs. It is only necessary to evaluate the $2k+1$ center diagonals of each subtree pair. So, $O(k^2 \times N)$ processors are needed for that problem.

We reach that processor bound for the full problem by the simple trick of evaluating the ***within k distance problem*** for successive powers of two. This less than doubles the time complexity and achieves the desired processor bound.

The full paper with the two method of calculating part II and all the proofs can be obtained from the authors.

6. References

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

[BSSBWD87] Berkout, B. Schmidt, B. F. Strien, A., Boom J., Westrenen, J., Duin, J., (1987), "Lysis gene of bateriophage MS2 is activated by translation termination at the overlapping coat gene." *Proc. Natl. Acad. Sci.* USA, 195, pp.517-524.

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

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

[Knuth] D. E. Knuth, *The Art of Computer Programming, vol. I* Addison-Wesley, Reading, Mass.

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

[LSV87] G.M. Landau, B. Schieber, and U. Vishkin, "Parallel construction of a suffix tree," *Proc. 14th ICALP*, Lecture Notes in Computer Science 267, Springer-Verlag, 1987, pp. 314-325.

[S88] Shapiro, B. A., "An algorithm for comparing multiple RNA secondary structures" to appear in Computer Applications in Biology, manuscript from Image processing section, Frederick Cancer Research Facility, building 469, room 150, Frederick Maryland 21701.

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

[SV88] B. Schieber and U. Vishkin, "Parallel computation of lowest common ancestor in trees," NYU Computer Science Technical Report.

[T79] Kuo-Chung Tai, "The tree-to-tree correction problem" JACM 26, pp. 422-433, 1979.

[TV85] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm" SIAM J. Computing, vol. 14, no. 4, November 1985

[U83] E. Ukkonen, "On approximate string matching," Proc. Int. Conf. Found. Comp. Theory, Lecture Notes in Computer Science 158, Spring-Verlag, 1983, pp. 487-495

[ZS87] K. Zhang and D. Shasha, "On the editing distance between trees and related problems" Ultracomputer Note 122, NYU C.S TR 310, August 1987

[ZS88] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems" Accepted by SIAM J. Computing