J Comb Optim (2014) 27:778–797
DOI 10.1007/s10878-012-9552-1

An improved algorithm for tree edit distance with applications for RNA secondary structure comparison

Shihyen Chen · Kaizhong Zhang

Published online: 12 October 2012

Abstract An ordered labeled tree is a tree in which the nodes are labeled and the left-to-right order among siblings is relevant. The edit distance between two ordered labeled trees is the minimum cost of transforming one tree into the other through a sequence of edit operations. We present techniques for speeding up the tree edit distance computation which are applicable to a family of algorithms based on closely related recursion strategies. These techniques aim to reduce repetitious steps in the original algorithms by exploring certain structural features in the tree. When these features exist in a large portion of the tree, the speedup due to our techniques would be significant. Viable examples for application include RNA secondary structure comparison and structured text comparison.

Keywords Tree edit distance · RNA secondary structure comparison · Structured text comparison

1 Introduction

An ordered labeled tree is a tree in which the nodes are labeled and the left-to-right order among siblings is significant. Trees can represent many phenomena, such as grammar parses, image descriptions and structured texts, to name a few. In many applications where trees are useful representations of objects, the need for comparing trees frequently arises.

S. Chen · K. Zhang (✉)
Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7
e-mail: kzhang@csd.uwo.ca

S. Chen
e-mail: shihyen_c@yahoo.ca

The tree edit distance metric was introduced by Tai as a generalization of the string editing problem (Tai 1979; Wagner and Fischer 1974). Given two trees T_1 and T_2, the tree edit distance between T_1 and T_2 is the minimum cost to transform one tree into the other by a sequence of edit operations. Tai gave an algorithm with a time complexity $\mathcal{O}(|T_1||T_2|\prod_{i=1}^{2} depth^2(T_i))$. Since then, a number of improved algorithms have been developed. Bille (2005) presented a survey on the tree edit distance algorithms. Our focus in this paper is on a family of algorithms that are based on closely related dynamic programming approaches. These algorithms were developed by Zhang and Shasha (1989), Klein (1998), and Demaine et al. (2009), with time complexities being $\mathcal{O}(|T_1||T_2|\prod_{i=1}^{2} \min\{depth(T_i), leaves(T_i)\})$, $\mathcal{O}(|T_1|^2|T_2|\log|T_2|)$, and $\mathcal{O}(|T_1|^2|T_2|(1+\log\frac{|T_2|}{|T_1|}))$, respectively. Dulucq and Touzet introduced the notion of *decomposition strategy* as a general framework for these algorithms (Dulucq and Touzet 2005). The algorithm by Demaine et al. has the best worst-case time complexity which also represents the optimality for decomposition strategy algorithms.

In this paper, we present techniques for speeding up the computation based on a strategy called *right decomposition*. The principles of these techniques are general enough to be used with other decomposition strategy algorithms. The idea is based on compression of nodes satisfying certain structural relations thereby reducing repetitious steps encountered by the original algorithms. Our techniques would in practice speed up all of the decomposition strategy algorithms in the presence of the related structural features.

The rest of the paper is organized as follows. In Sect. 2, we define notations to be used throughout the paper. We describe the key concepts underlying the decomposition strategy algorithms. In Sect. 3, we define the compressible parts of the tree and the associated operations and representations. In Sect. 4 and Sect. 5, we present techniques leading to improved algorithms. In Sect. 6, we provide experimental result for the speed improvement using RNA secondary structure comparison as an example.

2 Preliminaries

2.1 Notations and definitions

Given a tree T, we denote by $r(T)$ its root and $t[i]$ the ith node in the left-to-right postorder numbering. The subtree rooted at $t[i]$ is denoted by $T[i]$. The index of the leftmost leaf of $T[i]$ is denoted by $l(i)$. We denote by $T[i, j]$ the part of the tree which includes the nodes consecutively numbered from i to j. The structure obtained by removing $t[i]$ from $T[i]$ is denoted by $F[i]$, i.e., $F[i] = T[l(i), i-1]$. Note that $F[i]$ represents a forest if $t[i]$ has multiple children, and a tree which is $T[i-1]$ if $t[i]$ has only one child. When referring to the children of a specific node, we adopt a subscript notation in accordance with the left-to-right sibling order. For example, the children of $t[i]$, from left to right, may be denoted by $t[i_1], t[i_2], \ldots, t[i_k]$. Denote by $F_1 \circ F_2$ the concatenation of F_1 and F_2.

There are three basic edit operations on trees: substitution, insertion and deletion. The substitution operation substitutes a tree node with another one. The insertion

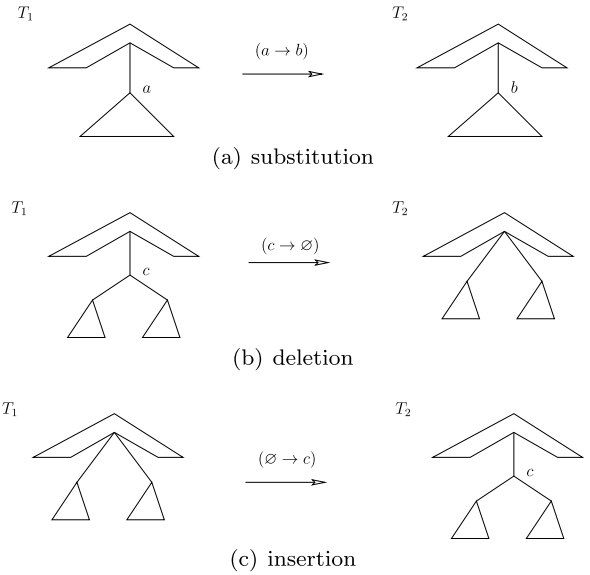


Fig. 1 Tree edit operations

operation inserts a node into a tree. The deletion operation deletes a node from a tree. These operations are displayed in Fig. 1.

We denote by $\delta(t_1, t_2)$ the cost for substituting t_1 with t_2, $\delta(t_1, \varnothing)$ the cost for deleting t_1, and $\delta(\varnothing, t_2)$ the cost for inserting t_2. The edit distance between T_1 and T_2 is denoted by $d(T_1, T_2)$.

2.2 Algorithmic strategies

For the family of algorithms that we consider, the solution for tree edit distance is based on the formula for forest-to-forest edit distance in Eq. (1) as a more general approach for formulating the tree distance solution, since when both forests are composed of one tree (i.e., $(F, G) = (T, T')$) the solution is a tree-to-tree distance as in Eq. (2).

$$d(F, G) = \min \begin{cases} d(F - r(T), G) + \delta(r(T), \varnothing), \\ d(F, G - r(T')) + \delta(\varnothing, r(T')), \\ d(F - T, G - T') + d(T - r(T), T' - r(T')) \\ \quad + \delta(r(T), r(T')) \end{cases}. \tag{1}$$

$$d\big(T, T'\big) = \min \begin{cases} d(T - r(T), T') + \delta(r(T), \varnothing), \\ d(T, T' - r(T')) + \delta(\varnothing, r(T')), \\ d(T - r(T), T' - r(T')) + \delta(r(T), r(T')) \end{cases}. \tag{2}$$

The recursion in Eq. (1) takes on two possible directions:

- right decomposition where both $r(T)$ and $r(T')$ are rightmost roots,
- left decomposition where both $r(T)$ and $r(T')$ are leftmost roots.

The algorithmic framework is based on dynamic programming which calls for a bottom-up enumeration scheme associated with some strategy that specifies the

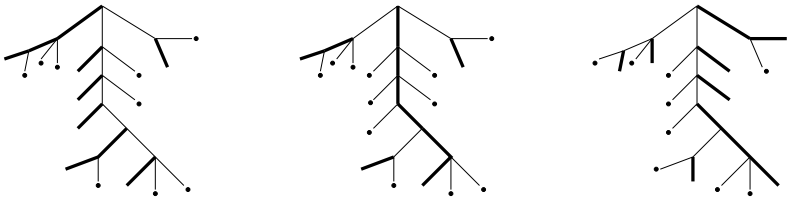


Fig. 2 The paths which guide the bottom-up enumeration of tree nodes in a dynamic program

recursion directions in Eq. (1). For a given subtree, the enumeration starts at a leaf and ends at the root where the leaf and the root are connected by a unique path with respect to which a postorder enumeration for the tree nodes is defined. Recursively starting at the root of the tree, a set of paths can be obtained into which the tree is decomposed. We may define three types of paths: leftmost paths, rightmost paths, and heavy paths. A strategy based on right decomposition is associated with leftmost paths. A strategy based on left decomposition is associated with rightmost paths. A strategy that recurses on both directions is associated with heavy paths. These paths are illustrated in Fig. 2 in thick lines. Each path guides the enumeration of nodes for a subtree in a dynamic programming distance computation. With leftmost paths, the enumeration starts at the leftmost leaf and follows the left-to-right postorder ending at the root. The enumeration based on rightmost paths is defined symmetrically to the case for leftmost paths. With heavy paths, the starting leaf belongs to a path where each subtree rooted on this path is the largest subtree among its sibling subtrees. The postorder defined by heavy path is left-to-right and right-to-left intermittently with respect to the heavy path.

Zhang and Shasha's algorithm decomposes both trees into leftmost paths. This yields two sets of subtrees. Each subtree-subtree pair between the two sets requires a separate distance computation. In each distance computation, the distances for subtree pairs with both roots on the leftmost paths can be obtained in the process of the same computation. Klein's algorithm applies the technique of heavy-path decomposition (Sleator and Tarjan 1983; Harel and Tarjan 1984) to the larger tree and enumerates all sub-forests in the smaller tree. Demaine et al. improved upon this algorithm by recursively decomposing the larger of two subtrees for which the distance is to be computed. Their algorithm returns the top-level heavy path in the larger subtree associated with each recursive call, as opposed to Klein's strategy which returns all the heavy paths in the initially larger tree. This makes it feasible to apply decomposition to both trees.

3 Compression of tree nodes

In this section, we define compressible parts of a tree.

Definition 1 (Maximal non-branching path) A path in a tree is a non-branching path if both the postorder tree traversal and pre-order tree traversal visit the nodes on the

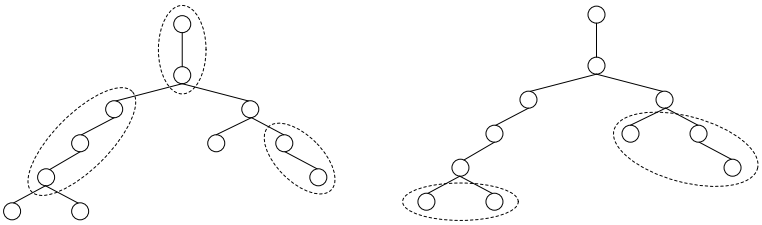


Fig. 3 Compressible parts: Each compressible path and compressible sequence is enclosed by *dashed* lines

path in consecutive order. A non-branching path is maximal if no other non-branching path contains it.

We consider two types of compressible parts as follows.

Definition 2 (Compressible path) A compressible path is a maximal non-branching path.

Definition 3 (Compressible sequence) A compressible sequence is a maximal contiguous sequence of sibling subtrees consisting of non-branching paths or leaves where the length of the sequence is ≥ 2.

Figure 3 gives an example of compressible parts in a tree.

We define two types of compression: path compression and sequence compression. Path compression replaces a compressible path by a single node. Sequence compression replaces a compressible sequence by a single node. We denote by T^p the tree obtained from T after path compression, and T^s the tree obtained from T^p after sequence compression. The size of a compressed node in T^p is the number of nodes in T that are compressed into this node. The size of a compressed node in T^s is the number of nodes in T^p that are compressed into this node.

We define a position mapping between compressed nodes and compressible paths by two functions α and β. For a compressed node $t^p[i]$ in T^p, $\alpha(i)$ and $\beta(i)$ are, respectively, the highest and lowest numbered positions on the path in T from which $t^p[i]$ is compressed. For generality, the functions can be trivially applied to uncompressed nodes where $\alpha(i) = \beta(i)$. Figure 4 shows an example of this mapping. The mapping between a compressed node in T^s and its corresponding sequence in T^p is defined similarly.

4 Computation with path compression

Our presentation of techniques is based on right decomposition. The principles of these techniques, however, are applicable to the other decomposition strategies as we will explain later.

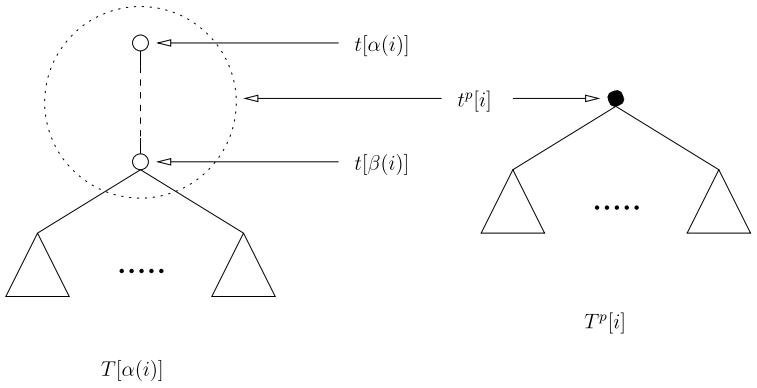


Fig. 4 Mapping between a compressed node (*right*) and a compressible path (*left*)

In computing the tree-to-tree distance in a bottom-up order, we enumerate the nodes through the forests contained in the trees. During this process, we encounter subtree-to-subtree distances which have been computed in separate computations and their values have been stored. Consider the following case. Given a pair of subtrees $(T_1[i], T_2[j])$, suppose that the top portion of each subtree is a maximal non-branching path. That is, $(t_1[i-k], t_1[i-k+1], \ldots, t_1[i])$ and $(t_2[j-l], t_2[j-l+1], \ldots, t_2[j])$, for some k and l, are maximal non-branching paths. Note that enumerating the nodes along these paths is necessary only for the computation of $d(T_1[i], T_2[j])$. Once the value of $d(T_1[i], T_2[j])$ is obtained, in subsequent computations for forest-to-forest distances of the form $d(F \circ T_1[i], G \circ T_2[j])$ where $d(F \circ F_1[i-k], G \circ F_2[j-l])$ has been evaluated, it suffices to consider only the following cases: $d(F \circ F_1[i-k], G \circ T_2[j]) + \sum_{u=i-k}^{i} \delta(t_1[u], \varnothing)$, $d(F \circ T_1[i], G \circ F_2[j-l]) + \sum_{v=j-l}^{j} \delta(\varnothing, t_2[v])$, and $d(F, G) + d(T_1[i], T_2[j])$, which can be done in constant time since the values for $\sum_{u=i-k}^{i} \delta(t_1[u], \varnothing)$ and $\sum_{v=j-l}^{j} \delta(\varnothing, t_2[v])$ can be preprocessed and stored. This observation forms the basis of the new algorithm where the paths $(t_1[i-k], t_1[i-k+1], \ldots, t_1[i])$ and $(t_2[j-l], t_2[j-l+1], \ldots, t_2[j])$ are effectively compressed in the above situation for the computation of $d(F \circ T_1[i], G \circ T_2[j])$.

The new algorithm with path compression is based on the following recurrence formulae.

Lemma 1

1. $d(\varnothing, \varnothing) = 0$.
2. $\forall i \in T_1^p$ *and* $\forall i'$ *such that* $l(i) \le i' \le i$,

$$d\big(T_1^p\big[l(i), i'\big], \varnothing\big) = d\big(T_1^p\big[l(i), i'-1\big], \varnothing\big) + \delta\big(t_1^p\big[i'\big], \varnothing\big).$$

3. $\forall j \in T_2^p$ *and* $\forall j'$ *such that* $l(j) \le j' \le j$,

$$d\big(\varnothing, T_2^p\big[l(j), j'\big]\big) = d\big(\varnothing, T_2^p\big[l(j), j'-1\big]\big) + \delta\big(\varnothing, t_2^p\big[j'\big]\big).$$

Proof In case 1, the edit distance between two identical trees, which are empty trees in this case, is zero. In case 2 and case 3, since one of the trees is empty there is only one way to write the last edit step, which must be a deletion or insertion, respectively. □

Lemma 2 *$\forall(i, j) \in (T_1^p, T_2^p)$ and $\forall(i', j')$ such that $l(i) \le i' \le i$ and $l(j) \le j' \le j$, if $l(i') = l(i)$ and $l(j') = l(j)$, then $d(T_1^p[l(i), i'], T_2^p[l(j), j']) = d(T_1^p[i'], T_2^p[j'])$ which is computed according to Lemmas 3 to 6; otherwise,*

$$d\big(T_1^p\big[l(i), i'\big], T_2^p\big[l(j), j'\big]\big) = \min \left\{ \begin{array}{l} d(T_1^p[l(i), i' - 1], T_2^p[l(j), j']) + \delta(t_1^p[i'], \varnothing), \\ d(T_1^p[l(i), i'], T_2^p[l(j), j' - 1]) + \delta(\varnothing, t_2^p[j']), \\ d(T_1^p[l(i), l(i') - 1], T_2^p[l(j), l(j') - 1]) + d(T_1^p[i'], T_2^p[j']) \end{array} \right\}.$$

Proof We are concerned with the distance between every pair of subtrees in T_1^p and T_2^p. For each such pair, the solution is built in a bottom-up order starting at the leftmost leaf node of each subtree, based on the postorder numbering. The partial solution may be a tree-to-tree distance or a forest-to-forest distance. In the case of tree-to-tree distance, which is identified by the condition $l(i') = l(i)$ and $l(j') = l(j)$, the original algorithm (Zhang and Shasha 1989) would use the formula in Eq. (2). However, for compressed trees the formula in Eq. (2) does not capture all the possible cases unless both tree roots are of size 1. Since the roots may be of size greater than 1, to correctly obtain this tree-to-tree distance, we must refer to the original trees for the corresponding compressible paths. This part of the computation is handled by the formulae in Lemmas 3 to 6. In the case of forest-to-forest distance, consider the roots of the rightmost trees in the compressed forests, and we have the following possible situations:

– both roots are of sizes greater than 1,
– one root is of size greater than 1 and the other root is of size 1, and
– both roots are of size 1.

We show for the first situation, where both roots are of sizes greater than 1, that the given formula in the lemma for forest-to-forest distance holds. The same argument applies for the other two situations which are special cases of the first one. With both roots of sizes greater than 1, we have two compressible paths which we denote by P_1 and P_2. Consider the suffix of the optimal forest-to-forest edit script, we may categorize all the possibilities in terms of the existence of node-to-node substitutions between P_1 and P_2, which leads to the following three cases:

1. the suffix ends with deletion of entire P_1,
2. the suffix ends with insertion of entire P_2,
3. the suffix contains at least one node-to-node substitution between P_1 and P_2, requiring the two rightmost trees to be matched with each other.

Hence, the formula holds. □

Note The main difference between the new formula as presented in Lemma 2 and the formula used in the original algorithm is in the way tree-to-tree distance is computed. This is due to the fact that the new algorithm is applied to compressed trees. The presence of the compressed roots makes it necessary to modify the procedure for computing the tree-to-tree distance.

We show how to compute $d(T_1^p[i'], T_2^p[j'])$ in the following lemmas.

Lemma 3 *$\forall u \in T_1$ such that $\beta(i') \le u \le \alpha(i')$ for some $(i', j') \in (T_1^p, T_2^p)$,*

$$d\big(T_1[u], F_2\big[\beta\big(j'\big)\big]\big) = \min_{j'_1 \le q \le j'_l} \left\{ \begin{array}{l} d(F_1[u], F_2[\beta(j')]) + \delta(t_1[u], \varnothing), \\ d(T_1[u], T_2[\alpha(q)]) - d(\varnothing, T_2[\alpha(q)]) + d(\varnothing, F_2[\beta(j')]) \end{array} \right\}.$$

Proof The optimal edit script for $d(T_1[u], F_2[\beta(j')])$ would end in one of the following situations:

- deletion of $t_1[u]$,
- insertion of $t_2[\beta(j') - 1]$, or
- substitution of $t_1[u]$ with $t_2[\beta(j') - 1]$.

We show that by considering the following two cases, it is sufficient to cover all the above situations. In the first case, $t_1[u]$ is deleted and the rest part $F_1[u]$ is matched to $F_2[\beta(j')]$. This case covers the first situation. In the second case, $t_1[u]$ is matched to a node somewhere in $F_2[\beta(j')]$. This case effectively covers the other two situations where if $t_1[u]$ is matched to the rightmost root in $F_2[\beta(j')]$, i.e., $t_2[\beta(j') - 1]$, it leads to the third situation; and, if $t_1[u]$ is matched to any other node in $F_2[\beta(j')]$, it leads to the second situation with $t_2[\beta(j') - 1]$ inserted. The matter for the second case therefore is to find a subtree in $F_2[\beta(j')]$ to be matched to $T_1[u]$ so as to minimize the distance between $T_1[u]$ and $F_2[\beta(j')]$ under such matching constraint. This can be done by considering the set of conversions where exactly one tree in $F_2[\beta(j')]$ is matched to $T_1[u]$ while the remainder of $F_2[\beta(j')]$ is deleted. The minimum in this set is retained for the second case. □

Note The above lemma computes the edit distance between the tree $T_1[u]$ and the forest $F_2[\beta(j')]$. For the base case where $u = \beta(i')$, the term $d(F_1[u], F_2[\beta(j')]) = d(F_1^p[i'], F_2^p[j']) = d(T_1^p[l(i'), i' - 1], T_2^p[l(j'), j' - 1])$ would have been obtained according to Lemma 2. When $u > \beta(i')$, $d(F_1[u], F_2[\beta(j')]) = d(T_1[u - 1], F_2[\beta(j')])$ would have been obtained from the same recursion in this lemma. The term $d(T_1[u], T_2[\alpha(q)])$ would have been obtained according to Lemma 5, which is to be shown below.

Lemma 4 *$\forall v \in T_2$ such that $\beta(j') \le v \le \alpha(j')$ for some $(i', j') \in (T_1^p, T_2^p)$,*

$$d\big(F_1\big[\beta\big(i'\big)\big], T_2[v]\big) = \min_{i'_1 \le p \le i'_k} \left\{ \begin{array}{l} d(F_1[\beta(i')], F_2[v]) + \delta(\varnothing, t_2[v]), \\ d(T_1[\alpha(p)], T_2[v]) - d(T_1[\alpha(p)], \varnothing) + d(F_1[\beta(i')], \varnothing) \end{array} \right\}.$$

Proof This is the symmetric case to Lemma 3. □

Lemma 5 $\forall (u, v) \in (T_1, T_2)$ *such that* $\beta(i') \leq u \leq \alpha(i')$ *and* $\beta(j') \leq v \leq \alpha(j')$ *for some* $(i', j') \in (T_1^p, T_2^p)$,

$$d\big(T_1[u], T_2[v]\big) = \min \left\{ \begin{array}{l} d(F_1[u], T_2[v]) + \delta(t_1[u], \varnothing), \\ d(T_1[u], F_2[v]) + \delta(\varnothing, t_2[v]), \\ d(F_1[u], F_2[v]) + \delta(t_1[u], t_2[v]) \end{array} \right\}.$$

Proof The optimal edit script for $d(T_1[u], T_2[v])$ would end in one of the following steps: (1) the root of $T_1[u]$, i.e. $t_1[u]$, is in a deletion, (2) the root of $T_2[v]$, i.e. $t_2[v]$, is in an insertion, or (3) the two roots $t_1[u]$ and $t_2[v]$ are in a substitution. □

Note For the base case where $u = \beta(i')$ and $v = \beta(j')$, $d(T_1[\beta(i')], T_2[\beta(j')])$ depends on the following terms: $d(F_1[\beta(i')], T_2[\beta(j')])$, $d(T_1[\beta(i')], F_2[\beta(j')])$, and $d(F_1^p[i'], F_2^p[j'])$, which would have been obtained from the recursions in Lemmas 4, 3 and 2, respectively. When $u = \beta(i')$ and $v > \beta(j')$, $d(T_1[u], T_2[v])$ depends on $d(F_1[u], T_2[v])$, $d(T_1[u], T_2[v-1])$ and $d(F_1[u], T_2[v-1])$, where $d(T_1[u], T_2[v-1])$ would have been obtained from the same recursion, and the other two distances from that in Lemma 4. The case for $u > \beta(i')$ and $v = \beta(j')$ is analyzed similarly. When $u > \beta(i')$ and $v > \beta(j')$, $d(T_1[u], T_2[v])$ depends on $d(T_1[u-1], T_2[v])$, $d(T_1[u], T_2[v-1])$ and $d(T_1[u-1], T_2[v-1])$ all of which would have been obtained from the same recursion. We need to ensure that the following values are retained: $d(T_1[u], T_2[\alpha(j')])$ $\forall u \in \{\beta(i'), \ldots, \alpha(i')\}$ and $d(T_1[\alpha(i')], T_2[v])$ $\forall v \in \{\beta(j'), \ldots, \alpha(j')\}$. The reason is that they will be needed for computations in Lemmas 3 and 4 as these values correspond to the terms of the following forms: $d(T_1[u], T_2[\alpha(q)])$ and $d(T_1[\alpha(p)], T_2[v])$.

Lemma 6 $\forall (i', j') \in (T_1^p, T_2^p)$, $d(T_1^p[i'], T_2^p[j']) = d(T_1[\alpha(i')], T_2[\alpha(j')])$.

Proof The lemma holds since by definition $T_1^p[i']$ represents $T_1[\alpha(i')]$ and $T_2^p[j']$ represents $T_2[\alpha(j')]$. □

The above lemmas lead to a dynamic programming algorithm. Along the way, we need space for both computation and storage purposes. The major space requirement is described as follows:

- D_f: a table where computations for forest-to-forest distances are carried out based on Lemmas 1 and 2,
- D_r: a table where computations for tree-to-tree distances based on Lemmas 3, 4, and 5 are carried out,
- D_t: a table where tree-to-tree distances are stored based on Lemma 6.

In the above tables, an entry in table D at the intersection of the ith row and jth column is denoted by $D[i, j]$. In addition, two linear arrays A_1 and A_2 are used for the purpose of retaining the following distances as mentioned in the note following Lemma 5: $d(T_1[u], T_2[\alpha(j')])$ $\forall u \in \{\beta(i'), \ldots, \alpha(i')\}$ and $d(T_1[\alpha(i')], T_2[v])$

input : (T_1, T_2)
output: $d(T_1[i], T_2[j])$, where $1 \le i \le |T_1|$, $1 \le j \le |T_2|$

1 preprocessing to construct (T_1^p, T_2^p), and compute $l()$, key roots, $\alpha()$ and $\beta()$;
2 sort $(keyroots(T_1^p), keyroots(T_2^p))$ in increasing order into arrays (K_1, K_2) ;
3 **for** $i' \leftarrow 1$ **to** $|keyroots(T_1^p)|$ **do** `/* computes subtree-to-subtree distance` $d(T_1^p[K_1[i']], T_2^p[K_2[j']])$ `*/`
4 **for** $j' \leftarrow 1$ **to** $|keyroots(T_2^p)|$ **do**
5 *SubtreeDistance*$(K_1[i'], K_2[j'])$;

Fig. 5 The new algorithm for computing tree-to-tree distance

$\forall v \in \{\beta(j'), \ldots, \alpha(j')\}$. The cells holding these distances in the dynamic programming table may be overlapped with the areas for boundary condition initialization in subsequent computations which will overwrite the existing distances in these cells. The linear arrays are used to temporarily transport these values out of the table in these situations so as to retain these values.

We define a set of nodes called *key roots* as

$$\text{keyroots}(T) = \{t \in T \mid t \text{ is the root of } T \text{ or has a left sibling}\}.$$

For each pair of key roots $(t_1[i], t_2[j])$, the distance $d(T_1[i], T_2[j])$ requires a separate dynamic programming computation. The new algorithm first sorts the two sets of key roots in compressed trees in increasing order and iterates through each pair of key roots and computes the subtree-to-subtree distances according to the lemmas presented above. The distance between the two trees is obtained in the last iteration. The main loop is presented in Fig. 5 which iterates through each pair of key roots. Note that the representation is based on the compressed trees whereas the counterpart of the main loop in the original algorithm is based on the uncompressed trees. Within each iteration, the relevant subtree-to-subtree distance is computed by the procedure *SubtreeDistance* which is presented in Fig. 6. The counterpart of this procedure in the original algorithm is presented in Fig. 7. Comparing Figs. 6 and 7, the main difference between the two procedures, apart from the trivial notational differences due to the newly introduced compressed representation, is illustrated in the computation for subtree-to-subtree distances.

Lines 1 to 3 in *SubtreeDistance* correspond to Lemma 1 where boundary values are initialized. The loop block starting at line 4 corresponds to Lemma 2. The conditional test on line 6 checks if the current entry corresponds to a pair of subtrees, the distance of which is computed by the corresponding block. Otherwise, the entry corresponds to forest-to-forest distance and is handled by the else-block. The blocks starting at lines 10, 12, 14 and 17 correspond to, respectively, Lemmas 3, 4, 5 and 6. On lines 7 and 8 the values in the boundary cells are transported out to make room for current boundary initialization, as they are needed in subsequent computations. On lines 18 and 19, these values are transported back to their original cells.

We now analyze the time complexity for the improved algorithm. We say that a node $t[i]$ is under a key root $t[k]$ if $t[i] \in T[k]$.

1 $D_f[l(i)-1, l(j)-1] \leftarrow 0$; /* Lemma 1 */
2 **for** $i' \leftarrow l(i)$ **to** i **do** $D_f[i', l(j)-1] \leftarrow D_f[i'-1, l(j)-1] + \delta(t_1^p[i'], \varnothing)$;
3 **for** $j' \leftarrow l(j)$ **to** j **do** $D_f[l(i)-1, j'] \leftarrow D_f[l(i)-1, j'-1] + \delta(\varnothing, t_2^p[j'])$;
4 **for** $i' \leftarrow l(i)$ **to** i **do** /* Lemma 2 */
5 **for** $j' \leftarrow l(j)$ **to** j **do**
6 **if** $l(i') = l(i)$ ***and*** $l(j') = l(j)$ **then** /* computes subtree-to-subtree distance $d(T_1^p[i'], T_2^p[j'])$ */
7 **for** $u \leftarrow \beta(i') - 1$ **to** $\alpha(i')$ **do** $A_1[u] \leftarrow D_r[u, \beta(j') - 1]$;
8 **for** $v \leftarrow \beta(j')$ **to** $\alpha(j')$ **do** $A_2[v] \leftarrow D_r[\beta(i') - 1, v]$;
9 $D_r[\beta(i') - 1, \beta(j') - 1] \leftarrow D_f[i' - 1, j' - 1]$;
10 **for** $u \leftarrow \beta(i')$ **to** $\alpha(i')$ **do** /* Lemma 3 */
11 $$D_r[u, \beta(j') - 1] \leftarrow \min \left\{ \begin{array}{l} D_r[u-1, \beta(j') - 1] + \delta(t_1[u], \varnothing), \\ \min_{j'_1 \le q \le j'_l} \{D_r[u, \alpha(q)] - \delta(\varnothing, T_2[\alpha(q)])\} \\ \quad + \delta(\varnothing, F_2[\beta(j')]) \end{array} \right\} ;$$
12 **for** $v \leftarrow \beta(j')$ **to** $\alpha(j')$ **do** /* Lemma 4 */
13 $$D_r[\beta(i') - 1, v] \leftarrow \min \left\{ \begin{array}{l} D_r[\beta(i') - 1, v - 1] + \delta(\varnothing, t_2[v]), \\ \min_{i'_1 \le p \le i'_k} \{D_r[\alpha(p), v] - \delta(T_1[\alpha(p)], \varnothing)\} \\ \quad + \delta(F_1[\beta(i')], \varnothing) \end{array} \right\} ;$$
14 **for** $u \leftarrow \beta(i')$ **to** $\alpha(i')$ **do** /* Lemma 5 */
15 **for** $v \leftarrow \beta(j')$ **to** $\alpha(j')$ **do**
16 $$D_r[u, v] \leftarrow \min \left\{ \begin{array}{l} D_r[u-1, v] + \delta(t_1[u], \varnothing), \\ D_r[u, v-1] + \delta(\varnothing, t_2[v]), \\ D_r[u-1, v-1] + \delta(t_1[u], t_2[v]) \end{array} \right\} ;$$
17 $D_t[i', j'] \leftarrow D_f[i', j'] \leftarrow D_r[\alpha(i'), \alpha(j')]$; /* Lemma 6 */
18 **for** $u \leftarrow \beta(i') - 1$ **to** $\alpha(i')$ **do** $D_r[u, \beta(j') - 1] \leftarrow A_1[u]$;
19 **for** $v \leftarrow \beta(j')$ **to** $\alpha(j')$ **do** $D_r[\beta(i') - 1, v] \leftarrow A_2[v]$;
20 **else** /* computes forest-to-forest distance $d(T_1^p[l(i), i'], T_2^p[l(j), j'])$ */
21 $$D_f[i', j'] \leftarrow \min \left\{ \begin{array}{l} D_f[i'-1, j'] + \delta(t_1^p[i'], \varnothing), \\ D_f[i', j'-1] + \delta(\varnothing, t_2^p[j']), \\ D_f[l(i')-1, l(j')-1] + D_t[i', j'] \end{array} \right\} ;$$

Fig. 6 *SubtreeDistance*(i, j)

Lemma 7 (Zhang and Shasha 1989) *The number of key roots a node in T is under is at most* $\min\{depth(T), leaves(T)\}$.

Theorem 1 *$d(T_1, T_2)$ can be computed in $\mathcal{O}(|T_1||T_2| + \prod_{i=1}^{2} |T_i^p| \min\{depth(T_i^p), leaves(T_i^p)\})$ time and $\mathcal{O}(|T_1||T_2|)$ space.*

Proof The time complexity depends on the following computations:

1. subtree distances $d(T_1^p[i], T_2^p[j])$ based on Lemmas 3, 4 and 5,
2. forest distances.

1 $D_f[l(i)-1, l(j)-1] \leftarrow 0$;
2 **for** $i' \leftarrow l(i)$ **to** i **do** $D_f[i', l(j)-1] \leftarrow D_f[i'-1, l(j)-1] + \delta(t_1[i'], \varnothing)$;
3 **for** $j' \leftarrow l(j)$ **to** j **do** $D_f[l(i)-1, j'] \leftarrow D_f[l(i)-1, j'-1]] + \delta(\varnothing, t_2[j'])$;
4 **for** $i' \leftarrow l(i)$ **to** i **do**
5 **for** $j' \leftarrow l(j)$ **to** j **do**
6 **if** $l(i') = l(i)$ ***and*** $l(j') = l(j)$ **then** `/* computes subtree-to-subtree distance` $d(T_1[i'], T_2[j'])$ `*/`
7 $D_f[i', j'] \leftarrow \min \left\{ \begin{array}{l} D_f[i'-1, j'] + \delta(t_1[i'], \varnothing), \\ D_f[i', j'-1] + \delta(\varnothing, t_2[j']), \\ D_f[i'-1], j'-1] + \delta(t_1[i'], t_2[j']) \end{array} \right\}$;
8 $D_t[i', j'] \leftarrow D_f[i', j']$;
9 **else** `/* computes forest-to-forest distance` $d(T_1[l(i), i'], T_2[l(j), j'])$ `*/`
10 $D_f[i', j'] \leftarrow \min \left\{ \begin{array}{l} D_f[i'-1, j'] + \delta(t_1[i'], \varnothing), \\ D_f[i', j'-1] + \delta(\varnothing, t_2[j']), \\ D_f[l(i')-1, l(j')-1] + D_t[i', j'] \end{array} \right\}$;

Fig. 7 The original procedure for computing subtree-to-subtree distance $d(T_1[i], T_2[j])$

Each subtree distance is computed only once when the algorithm encounters it the first time. The running time depends on the sizes and degrees of the compressed nodes, since for each compressed node the algorithm enumerates the nodes on the corresponding compressible path as well as the children of the compressed node. Let the numbers of compressed nodes in T_1^p and T_2^p be N_1 and N_2, respectively. Let $size(i)$ be the size of the ith compressed node. The running time due to subtree distances is bounded by $\sum_{i=1}^{N_1} \sum_{j=1}^{N_2} (size_1(i) degree_2(j) + degree_1(i) size_2(j) + size_1(i) size_2(j)) = \mathcal{O}(|T_1||T_2|)$.

For the complexity due to forest distances, by Lemma 7 each pair of nodes in (T_1^p, T_2^p) participate in at most $\prod_{i=1}^{2} \min\{depth(T_i^p), leaves(T_i^p)\}$ separate dynamic programming subtree distance computations. Therefore, the computation for forest distances takes $\mathcal{O}(\prod_{i=1}^{2} |T_i^p| \min\{depth(T_i^p), leaves(T_i^p)\})$ time. Hence, the time complexity follows. It is clear that the data structures used consumes a quadratic space. □

Since a compressible path is non-branching, there is only one way a dynamic program may recurse along this path regardless which strategy is used. Consequently, the techniques related to path compression can be used with all the decomposition strategy algorithms.

Corollary 1 *For a decomposition strategy algorithm $\mathcal{A}$, $d(T_1, T_2)$ can be computed in $\mathcal{O}(|T_1||T_2| + \mathcal{T}(\mathcal{A}, |T_1^p|, |T_2^p|))$ time where $\mathcal{T}(\mathcal{A}, |T_1^p|, |T_2^p|)$ is the time complexity of a decomposition strategy algorithm $\mathcal{A}$ applied to $(|T_1^p|, |T_2^p|)$.*

5 Computation with sequence compression

In this section, we present additional techniques for speeding up the computation, making use of certain properties related to compressible sequences. These properties allow us to relate the computation of forest-to-forest distances in the presence of compressible sequences to the problem of finding row minima in a totally monotone matrix which can be solved efficiently.

5.1 A transformation of the problem

The techniques to be presented are closely related to the method of finding row minima in a totally monotone matrix. A matrix $A_{m\times n}$ is totally monotone if $\forall\ a < b$ and $c < d$, $A_{a,c} \geq A_{a,d} \Rightarrow A_{b,c} \geq A_{b,d}$. Denote by lp_i the leftmost position with the minimum value in the ith row. The following inequality holds as a consequence of total monotonicity: $lp_1 \leq lp_2 \leq \cdots \leq lp_n$.

Aggarwal et al. proposed a linear time solution for the problem. The next lemma states the result.

Lemma 8 (Aggarwal et al. 1987) *The problem of finding row minima in a totally monotone matrix $A_{m\times n}$ can be solved in $\mathcal{O}(m+n)$ time using $\mathcal{O}(\max\{m,n\})$ space.*

We consider the computation of forest-to-forest distances of the form $d(F \circ h_1[1,l_1], G \circ h_2[1,l_2])$ where $h_1[1,l_1]$ and $h_2[1,l_2]$ are compressible sequences. In the course of computing these distances, instead of evaluating all the $l_1 l_2$ terms corresponding to $d(F \circ h_1[1,i], G \circ h_2[1,j])$ for $1 \leq i \leq l_1$ and $1 \leq j \leq l_2$, we evaluate only $l_1 + l_2 - 1$ terms corresponding to $d(F \circ h_1[1,i], G \circ h_2[1,l_2])$ for $1 \leq i \leq l_1$ and $d(F \circ h_1[1,l_1], G \circ h_2[1,i])$ for $1 \leq i \leq l_2$. This is feasible with the total monotonicity among relevant terms on which the evaluation of the above terms depends. Computing only these entries instead of the entire block in the dynamic programming table is sufficient for the computation to continue beyond the boundary. The omitted computation for the rest of the block results in additional time saving. The situation is depicted in Fig. 8. The next lemma states how these terms are evaluated.

Lemma 9 $\forall i \in \{1,\ldots,l_1\}$,

$$d\big(F \circ h_1[1,i], G \circ h_2[1,l_2]\big) = \min_{\substack{0\leq p\leq i\\ 0\leq q\leq l_2}} \left\{ \begin{array}{l} d(F \circ h_1[1,p], G) + d(h_1[p+1,i], h_2[1,l_2]), \\ d(F, G \circ h_2[1,q]) + d(h_1[1,i], h_2[q+1,l_2]) \end{array} \right\}. \tag{3}$$

$\forall i \in \{1,\ldots,l_2\}$,

$$d\big(F \circ h_1[1,l_1], G \circ h_2[1,i]\big) = \min_{\substack{0\leq p\leq l_1\\ 0\leq q\leq i}} \left\{ \begin{array}{l} d(F \circ h_1[1,p], G) + d(h_1[p+1,l_1], h_2[1,i]), \\ d(F, G \circ h_2[1,q]) + d(h_1[1,l_1], h_2[q+1,i]) \end{array} \right\}. \tag{4}$$

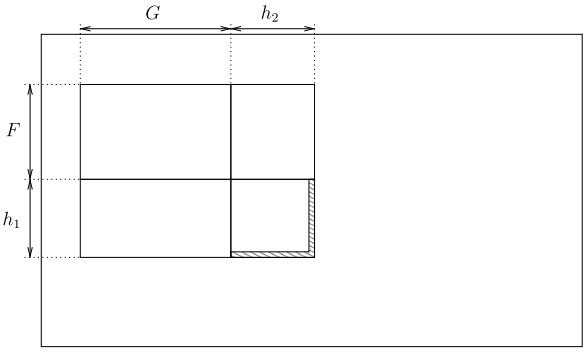


Fig. 8 Dynamic programming table for $d(F \circ h_1, G \circ h_2)$. For the computation in the *lower right block*, it is sufficient to compute only the entries on the boundary in the *shaded part* with total monotonicity

Proof We give the proof for Eq. (3). Equation (4) is the symmetrical case and the proof is similar.

Let $\langle h_1[p+1], h_2[q+1]\rangle$, $0 \le p \le i$ and $0 \le q \le l_2$, be the leftmost substitution step between $h_1[1,i]$ and $h_2[1,l_2]$ in the optimal edit script. Since there does not exist any substitution between $h_1[1,p]$ and $h_2[1,q]$, there are three cases to consider:

1. $h_1[1,p]$ is entirely deleted and $h_2[1,q]$ is entirely inserted: $d(F \circ h_1[1,i], G \circ h_2[1,l_2]) = d(F \circ h_1[1,p], G) + d(h_1[p+1,i], h_2[1,l_2]) = d(F, G \circ h_2[1,q]) + d(h_1[1,i], h_2[q+1,l_2])$.
2. There exists substitution between $h_1[1,p]$ and G, and $h_2[1,q]$ is entirely inserted: $d(F \circ h_1[1,i], G \circ h_2[1,l_2]) = d(F \circ h_1[1,p], G) + d(h_1[p+1,i], h_2[1,l_2])$.
3. $h_1[1,p]$ is entirely deleted, and there exists substitution between $h_2[1,q]$ and F: $d(F \circ h_1[1,i], G \circ h_2[1,l_2]) = d(F, G \circ h_2[1,q]) + d(h_1[1,i], h_2[q+1,l_2])$.

The lemma follows from the above cases. □

The next lemma states the total monotonicity relations satisfied by the distances in Lemma 9.

Lemma 10 *Let*

$$\begin{aligned}
A_{ij} &= d\big(F \circ h_1[1,j], G\big) + d\big(h_1[j+1,i], h_2[1,l_2]\big),\\
B_{ij} &= d\big(F, G \circ h_2[1,j]\big) + d\big(h_1[1,i], h_2[j+1,l_2]\big),\\
C_{ij} &= d\big(F, G \circ h_2[1,j]\big) + d\big(h_1[1,l_1], h_2[j+1,i]\big),\\
D_{ij} &= d\big(F \circ h_1[1,j], G\big) + d\big(h_1[j+1,l_1], h_2[1,i]\big).
\end{aligned}$$

The following relations hold:

$$\begin{aligned}
&A_{ij} + A_{i'j'} \le A_{ij'} + A_{i'j}, \quad 0 \le j \le j' \le i \le i' \le l_1.\\
&B_{ij} + B_{i'j'} \ge B_{ij'} + B_{i'j}, \quad 1 \le i \le i' \le l_1,\ 0 \le j \le j' \le l_2.\\
&C_{ij} + C_{i'j'} \le C_{ij'} + C_{i'j}, \quad 0 \le j \le j' \le i \le i' \le l_2.\\
&D_{ij} + D_{i'j'} \ge D_{ij'} + D_{i'j}, \quad 1 \le i \le i' \le l_2,\ 0 \le j \le j' \le l_1.
\end{aligned}$$

Proof We give the proof for $A_{ij} + A_{i'j'} \le A_{ij'} + A_{i'j}$. The other relations are symmetrical cases and the proofs are similar.

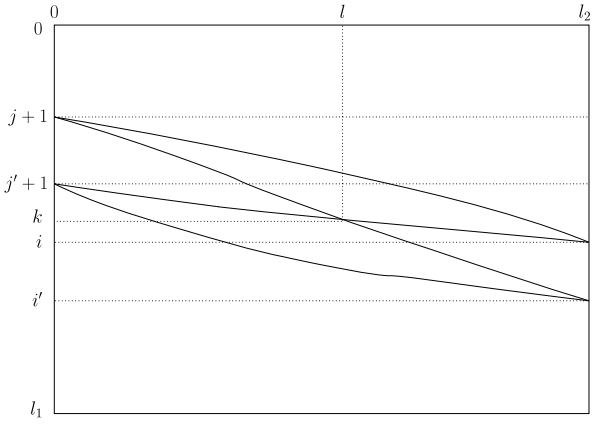


Fig. 9 An example for the proof of Lemma 10

Define $d_{ij} = d(h_1[j+1, i], h_2[1, l_2])$, then $A_{ij} = d(F \circ h_1[1, j], G) + d_{ij}$. Since $d(F \circ h_1[1, j], G) + d(F \circ h_1[1, j'], G)$ is included in both $A_{ij} + A_{i'j'}$ and $A_{ij'} + A_{i'j}$, it suffices to show $d_{ij} + d_{i'j'} \leq d_{ij'} + d_{i'j}$. Since $[j', i] \subseteq [j, i']$, the edit paths for $d_{ij'}$ and $d_{i'j}$ must meet at some point (see Fig. 9). Therefore, $\exists(k, l)$ where $j' \leq k \leq i$ and $0 \leq l \leq l_2$, such that

$$d_{ij'} = d\big(h_1\big[j'+1, k\big], h_2[1, l]\big) + d\big(h_1[k+1, i], h_2[l+1, l_2]\big),$$
$$d_{i'j} = d\big(h_1[j+1, k], h_2[1, l]\big) + d\big(h_1\big[k+1, i'\big], h_2[l+1, l_2]\big).$$

Since

$$d_{ij} \leq d\big(h_1[j+1, k], h_2[1, l]\big) + d\big(h_1[k+1, i], h_2[l+1, l_2]\big),$$
$$d_{i'j'} \leq d\big(h_1\big[j'+1, k\big], h_2[1, l]\big) + d\big(h_1\big[k+1, i'\big], h_2[l+1, l_2]\big),$$

it follows that $d_{ij} + d_{i'j'} \leq d_{ij'} + d_{i'j}$, hence $A_{ij} + A_{i'j'} \leq A_{ij'} + A_{i'j}$. □

5.2 Algorithmic techniques

With the properties established in the previous section, we present techniques which make use of these properties to speed up the computation.

For convenience, we group the distances in Lemma 10 into sets as follows. Denote by $\mathcal{J}$ the union of sets of the following forms:

- $\{d(F \circ h_1[1, i], G \circ h_2[1, l_2]) \mid 1 \leq i \leq l_1\}$,
- $\{d(F \circ h_1[1, l_1], G \circ h_2[1, i]) \mid 1 \leq i \leq l_2\}$.

Denote by $\mathcal{I}$ the union of sets of the following forms:

- $\{d(F \circ h_1[1, i], G) \mid 1 \leq i \leq l_1\}$,
- $\{d(F, G \circ h_2[1, i]) \mid 1 \leq i \leq l_2\}$.

Denote by $\mathcal{H}$ the union of sets of the following forms:

- $\{d(h_1[j+1, i], h_2[1, l_2]) \mid 1 \leq i \leq l_1, 0 \leq j \leq i\}$,
- $\{d(h_1[1, i], h_2[j+1, l_2]) \mid 1 \leq i \leq l_1, 0 \leq j \leq l_2\}$,

– $\{d(h_1[1, l_1], h_2[j+1, i]) \mid 1 \le i \le l_2, 0 \le j \le i\}$,
– $\{d(h_1[j+1, l_1], h_2[1, i]) \mid 1 \le i \le l_2, 0 \le j \le l_1\}$.

In the above three sets, we need to compute $\mathcal{H}$ only once and store the values which will be used repeatedly for computing $\mathcal{I}$ and $\mathcal{J}$. Based on Lemma 9, the distances in $\mathcal{J}$ may be obtained from the distances in $\mathcal{I}$ and $\mathcal{H}$. Prior to the computation of $\mathcal{J}$ the distances in $\mathcal{I}$ would have been computed and their values are available in the same dynamic programming table. The task of evaluating the distances in $\mathcal{H}$ is equivalent to evaluating edit distances between subsequences of two sequences, which can be handled efficiently with a divide-and-conquer scheme. The next lemma states the result.

Lemma 11 (Apostolico et al. 1990) *Given $h_1[1, l_1]$ and $h_2[1, l_2]$ where $d(h_1[l], h_2[l'])$ is known for all $l \in \{1, \ldots, l_1\}$ and $l' \in \{1, \ldots, l_2\}$, the distances in $\mathcal{H}$ can be evaluated in $\mathcal{O}(\max\{l_1^2 \log l_1, l_2^2 \log l_2\})$ time and $\mathcal{O}(l_1 l_2)$ space.*

Proof We associate with $(h_1[1, l_1], h_2[1, l_2])$ a grid $G_{(l_1+1)\times(l_2+1)}$. On this grid, directed edges are drawn such that for any vertex $v(i, j)$, the only allowed outgoing edges are to $v(i, j+1)$, $v(i+1, j)$ and $v(i+1, j+1)$. Each edge is assigned a cost: the cost for $e((i, j), (i, j+1))$ is $\delta(\varnothing, h_2[j+1])$, the cost for $e((i, j), (i+1, j))$ is $\delta(h_1[i+1], \varnothing)$, and the cost for $e((i, j), (i+1, j+1))$ is $d(h_1[i+1], h_2[j+1])$. To each of the terms we need to evaluate corresponds a minimum-cost path which starts at the left or top border and ends at the bottom or right border. The task of evaluating the terms is reduced to finding their corresponding minimum-cost paths. The problem is recursively solved for the four sub-grids of equal size of which the solutions are combined to form the whole solution. The combine step relies on finding positions on the common border between adjacent sub-grids whereby the minimum-cost paths passing through the border are constructed by connecting the minimum-cost paths incident to these positions in each sub-grid. The order of such positions for a fixed start point on the left or top border of a sub-grid is an instance of total monotonicity with respect to the linear order of the end points on the bottom and right border of the adjacent sub-grid. Let $l = \max\{l_1, l_2\}$. The combine step takes $\mathcal{O}(l^2)$ time. The total running time satisfies the recurrence relation: $T(l) \le 4T(l/2) + kl^2$. Hence, the time complexity is $\mathcal{O}(l^2 \log l)$. □

The main result in this section is given in the next lemma which shows how to compute the terms in $\mathcal{J}$ within a desired time bound, given that some preprocessing is done efficiently as shown in the previous lemma.

Lemma 12 *Given $(F \circ h_1[1, l_1], G \circ h_2[1, l_2])$, the distances $d(F \circ h_1[1, i], G \circ h_2[1, l_2])$ $\forall i \in \{1, \ldots, l_1\}$ and $d(F \circ h_1[1, l_1], G \circ h_2[1, i])$ $\forall i \in \{1, \ldots, l_2\}$ can be obtained in $\mathcal{O}(l_1 + l_2)$ time from $\mathcal{H}$ and $\mathcal{I}$.*

Proof From Lemma 10, it is straightforward to verify that a matrix with entries defined as follows is a totally monotone matrix.

$$A_{i,j} = \begin{cases} d(F \circ h_1[1,j], G) + d(h_1[j+1,i], h_2[1,l_2]), & \text{if } 0 \le j \le i, \\ A_{i,i}, & \text{if } j > i. \end{cases}$$

$$B_{i,j} = d\big(F, G \circ h_2[1, l_2 - j]\big) + d\big(h_1[1,i], h_2[l_2 - j + 1, l_2]\big).$$

$$C_{i,j} = \begin{cases} d(F, G \circ h_2[1,j]) + d(h_1[1,l_1], h_2[j+1,i]), & \text{if } 0 \le j \le i, \\ C_{i,i}, & \text{if } j > i. \end{cases}$$

$$D_{i,j} = d\big(F \circ h_1[1, l_1 - j], G\big) + d\big(h_1[l_1 - j + 1, l_1], h_2[1,i]\big).$$

To compute $d(F \circ h_1[1,i], G \circ h_2[1,l_2])$ $\forall i \in \{1,\ldots,l_1\}$ and $d(F \circ h_1[1,l_1], G \circ h_2[1,i])$ $\forall i \in \{1,\ldots,l_2\}$, we compute the row minima for the above matrices, store the row minima of A, B, C, and D in arrays R_1, R_2, R_3, and R_4, respectively, where the ith cell in each array stores the minimum in the ith row of the corresponding matrix, then obtain $d(F \circ h_1[1,i], G \circ h_2[1,l_2])$ and $d(F \circ h_1[1,l_1], G \circ h_2[1,i])$ as $\min\{R_1[i], R_2[i]\}$ and $\min\{R_3[i], R_4[i]\}$, respectively. By Lemma 8, this takes $\mathcal{O}(l_1 + l_2)$ time. □

We have shown the situation in which it is feasible to omit the work for portions of the computation table that is normally required. Although this incurs slightly extra work in preprocessing, it is worth the trade-off as these table entries would repeatedly associate with many different pairs of forests.

The techniques presented in this section would be useful in situations where the trees contain a large number of leaves. One such case may be text comparison where the texts can be represented by trees.

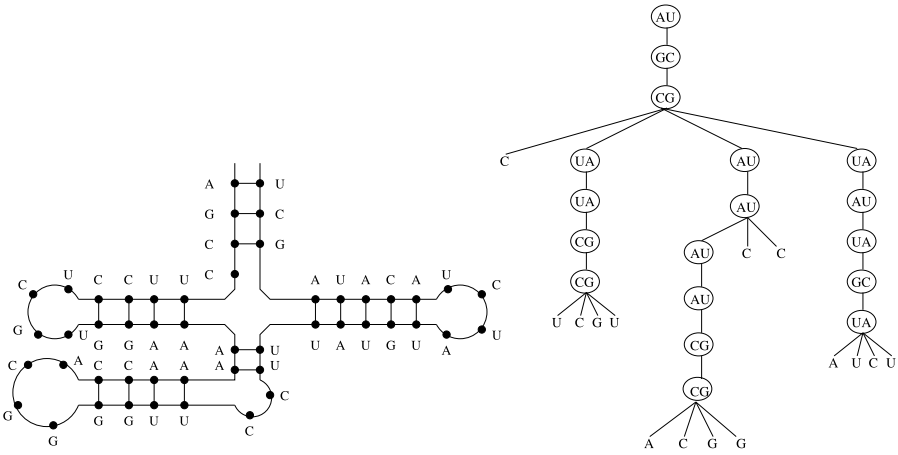
6 Applications

We give an example for RNA secondary structure comparison where path compression is applied to speed up the computation.

RNA is an important molecule that performs a wide range of functions in biological systems. The secondary structure of RNA plays an important role in its functions (Moore 1999). RNA has recently become the center of much attention due to its catalytic properties which lead to an increasing interest in obtaining its structural information. In many biological problems involving RNA, it is required to compare the structures of the molecules. The presumption is that, to a preserved biological function there corresponds a preserved molecular structure. Therefore, discovering structural similarity can help reveal valuable information regarding biological functions. As such, the ability of comparing RNA structures is of fundamental importance in many research tasks involving RNA molecules.

The RNA structure consists of a single strand of nucleotides (abbreviated as A, C, G and U) which folds back onto itself by means of hydrogen bonding between distant complementary nucleotides, giving rise to the secondary structure. The secondary structure of an RNA molecule can be represented by a tree, as illustrated by the example in Fig. 10. Therefore, the problem of comparing RNA secondary structures can be treated as a tree editing problem.

Represented as a tree, a pair of interacting nucleotides in an RNA secondary structure corresponds to an internal node in the tree. When a number of such pairs are



(a) An RNA secondary structure (b) A tree representation

Fig. 10 The secondary structure of RNA and the corresponding tree representation

Table 1 Tree sizes for RNA molecules

	RNA	length	n	n^p	$\frac{n^p}{n}$
1	D. Radiodurans	486	333	207	0.622
2	R. Rubrum	429	297	192	0.646
3	P. Gingivalis	398	280	184	0.657
4	P. Marinus	387	271	179	0.661
5	K. Pneumoniae	383	269	180	0.669
6	Y. Pestis	377	265	178	0.672
7	S. Marcescens	378	266	179	0.673
8	B. Anthracis	408	293	199	0.679

stacked up, they form a frequently occurring local structure called *stem*, which is represented by a maximal non-branching path in the tree.

The fraction of size due to stems is a relatively consistent feature in RNA molecules. For most RNA molecules, about $\frac{1}{3}$ of the size is due to stems when each interacting pair of nucleotides are counted as a single unit. This feature is illustrated in Table 1 where n denotes the size of the input tree T and n^p the size of the path-compressed tree T^p where each stem is represented by a single node.

The original algorithm requires $\mathcal{O}(|T_1||T_2|)$ space for computing the forest distances. When $\frac{2}{3}$ of the tree nodes are used, the portion of the space where the computation takes place is proportionally reduced to $(\frac{2}{3})^2$ while the density of workload remains approximately unchanged in that space, which would roughly lead to a 50 % improvement of running time. The speedup applies to all of the decomposition strategy algorithms since the above situation holds regardless which algorithm

Table 2 New running time as a fraction of the original running time for the leftmost-paths algorithm, applied to the RNA molecules in Table 1

	1	2	3	4	5	6	7	8
1	0.524							
2	0.528	0.547						
3	0.534	0.545	0.553					
4	0.539	0.534	0.536	0.539				
5	0.529	0.533	0.535	0.536	0.539			
6	0.537	0.541	0.543	0.543	0.545	0.548		
7	0.546	0.549	0.550	0.551	0.552	0.554	0.556	
8	0.555	0.555	0.555	0.554	0.553	0.554	0.555	0.556

is used. Table 2 presents the experimental results on the molecules in Table 1, which confirms the above estimate.

7 Conclusions

In this paper, we consider the problem of computing the edit distance between two ordered and labeled trees. We have presented techniques based on path compression and sequence compression for speeding up a family of tree edit distance algorithms based on closely related decomposition strategies. The speedup is particularly significant for situations where the structural features we have explored are present in a large portion of the tree. We presented experimental result showing significant speed improvement using RNA secondary structure comparison as an example.

Acknowledgements We thank Zhifeng Lin for obtaining the experimental data. This research was supported by Natural Science and Engineering Research Council of Canada.

References

Aggarwal A, Klawe MM, Moran S, Shor P, Wilber R (1987) Geometric applications of a matrix-searching algorithm. Algorithmica 2:195–208

Apostolico A, Atallah MJ, Larmore LL, McFaddin HS (1990) Efficient parallel algorithms for string editing and related problems. SIAM J Comput 19(5):968–988

Bille P (2005) A survey on tree edit distance and related problems. Theor Comput Sci 337:217–239

Demaine ED, Mozes S, Rossman B, Weimann O (2009) An optimal decomposition algorithm for tree edit distance. ACM Trans Algorithms 6(1):2:1–2:19

Dulucq S, Touzet H (2005) Decomposition algorithms for the tree edit distance problem. J Discrete Algorithms 3:448–471

Harel D, Tarjan RE (1984) Fast algorithms for finding nearest common ancestors. SIAM J Comput 13(2):338–355

Klein PN (1998) Computing the edit-distance between unrooted ordered trees. In: Proceedings of the 6th European symposium on algorithms (ESA), pp 91–102

Moore PB (1999) Structural motifs in RNA. Annu Rev Biochem 68:287–300

Sleator DD, Tarjan RE (1983) A data structure for dynamic trees. J Comput Syst Sci 26:362–391

Tai K (1979) The tree-to-tree correction problem. J ACM 26(3):422–433
Wagner RA, Fischer MJ (1974) The string-to-string correction problem. J ACM 21(1):168–173
Zhang K, Shasha D (1989) Simple fast algorithms for the editing distance between trees and related problems. SIAM J Comput 18(6):1245–1262