Efficient Computation of the Tree Edit Distance

MATEUSZ PAWLIK and NIKOLAUS AUGSTEN, University of Salzburg

We consider the classical tree edit distance between ordered labelled trees, which is defined as the minimum-cost sequence of node edit operations that transform one tree into another. The state-of-the-art solutions for the tree edit distance are not satisfactory. The main competitors in the field either have optimal worst-case complexity, but the worst case happens frequently, or they are very efficient for some tree shapes, but degenerate for others. This leads to unpredictable and often infeasible runtimes. There is no obvious way to choose between the algorithms.

In this paper we present RTED, a robust tree edit distance algorithm. The asymptotic complexity of our algorithm is smaller or equal to the complexity of the best competitors for any input instance, i.e., our algorithm is both efficient and worst-case optimal. This is achieved by computing a dynamic decomposition strategy that depends on the input trees. RTED is shown to be optimal among all algorithms that use LRH (Left-Right-Heavy) strategies, which include RTED and the fastest tree edit distance algorithms presented in literature. In our experiments on synthetic and real world data we empirically evaluate our solution and compare it to the state of the art.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*Pattern Matching*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Trees*; H.2.4 [**Database Management**]: Systems—*Query Processing*; I.5.3 [**Pattern Recognition**]: Clustering—*Similarity Measures*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: approximate matching, tree edit distance, similarity search

ACM Reference Format:
ACM Trans. Datab. Syst. V, N, Article A (January YYYY), 40 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Tree structured data appear in many applications, for example, XML documents are often represented as ordered labelled trees. An interesting query computes the difference between two trees. This is useful when dealing with different versions of a tree (for example, when synchronizing file directories or archiving websites), or to find pairs of similar trees (for example, for record linkage or data mining). The standard approach to tree differences is the tree edit distance, which computes the minimum-cost sequence of node edit operations that transform one tree into another. The tree edit distance has been applied successfully in a wide range of applications, such as bioinformatics [Ma et al. 2002; Heumann and Wittum 2009; Akutsu 2010], image analysis [Bellando and Kothari 1999], pattern recognition [Klein et al. 2000], melody recognition [Habrard et al. 2008], natural language processing [Lin et al. 2010], information extraction [de Castro Reis et al. 2004; Kim et al. 2007], or document retrieval [Kamali and Tompa 2013], and has received considerable attention from the database community [Chawathe 1999; Guha et al. 2002; Cobena et al. 2002; Lee et al. 2004; Garofalakis and Kumar 2005; Dalamagas et al. 2006; Augsten et al. 2010a; Cohen 2013; Korn et al. 2013].

The tree edit distance problem has a recursive solution that decomposes the trees into smaller subtrees and subforest. The best known algorithms are dynamic programming implementations of this recursive solution. Only two of these algorithms achieve $O(n^2)$ space complexity, where n is the number of tree nodes: the classical algorithm by Zhang and Shasha [1989], which runs in $O(n^4)$ time, and the algorithm by Demaine et al. [2009], which runs in $O(n^3)$ time. Demaine's algorithm was shown to be worst-case optimal, i.e., no implementation of the recursive solution can improve over cubic runtime for the worst case. The runtime complexity is given by the number of subproblems that must be solved.

The runtime behaviour of the two space-efficient algorithms heavily depends on the tree shapes, and it is hard to choose between the algorithms. Zhang's algorithm runs efficiently in $O(n^2 \log^2 n)$ time for trees with depth $O(\log n)$, but it runs into the $O(n^4)$ worst case for some other tree shapes. Demaine's algorithm is better in the worst case, but unfortunately the $O(n^3)$ worst case happens frequently, also for tree shapes for which Zhang's algorithm is almost quadratic. The runtime complexity can vary by more than a polynomial degree, and the choice of the wrong algorithm leads to a prohibitive runtime for tree pairs that could be computed efficiently otherwise. There is no easy way to predict the runtime behaviour of the algorithms for a specific pair of trees, and this problem has not been addressed in literature.

In this paper we develop RTED, a new algorithm for the tree edit distance. RTED is robust, i.e., independently of the tree shape the number of subproblems that RTED computes is at most as high as the number of subproblems the best competitor must compute. In many cases RTED beats the competitors and is still efficient when they run into the worst case. RTED requires $O(n^2)$ space as the most space-efficient competitors and its runtime complexity of $O(n^3)$ in the worst case is optimal.

The key to our solution is our dynamic decomposition strategy. A decomposition strategy recursively decomposes the input trees into subforests by removing nodes. At each recursive step, either the leftmost or the rightmost root node must be removed. Different choices lead to different numbers of subproblems and thus to different runtime complexities. Zhang's algorithm always removes from the right, Demaine removes the largest subtree in a subforest last, after removing all nodes to the left and then all nodes to the right of that subtree. RTED dynamically chooses one of the above strategies at each recursive step, and we show that the choice is optimal.

We develop a recursive cost formula for different strategies and present an algorithm which computes the optimal strategy in $O(n^2)$ time and space. The computation of the strategy does not increase space or runtime complexity of the tree edit distance algorithm. Our experimental evaluation confirms the analytic results and shows that the time used to compute the strategy is small compared to the overall runtime, and the percentage decreases with the tree size. Summarizing, the contributions of this paper are the following:

— We introduce the class of *path strategies* to decompose the input trees and the general tree edit distance algorithm, GTED, which implements any path strategy in $O(n^2)$ space. We identify LRH, the smallest subclass of path strategies that covers all tree edit distance algorithms presented in literature.
— We introduce the concept of *single-path functions*, which are at the core of all tree edit distance algorithms and process left, right, or heavy paths of a tree. We analyse the single-path functions of the existing algorithms and discuss their limitations. Most notably, we address the issue of subproblem indexing, which is crucial for the efficient retrieval of intermediate results. This problem has not been addressed in literature

before, and a straightforward approach requires an index of size $O(n^2)$. We provide a solution of size $O(n)$ based on a new forest indexing scheme.
— We analyse and compare the single-path functions presented in literature and find that none of them is suited for the efficient processing of general LRH strategies. We develop a new, versatile single-path function which processes LRH strategies efficiently, overcoming the problems of the previous solutions.
— We present RTED, our robust tree edit distance algorithm. For any tree pair, the number of subproblems computed by RTED is at most the number of subproblems computed by any previous LRH algorithm. The optimal path strategy is computed in $O(n^2)$ time and space, and does not increase the overall space or runtime complexity. Empirical evaluations show that the strategy computation takes only a small percentage of the overall runtime.
— We empirically evaluate RTED and compare it to the state-of-the-art algorithms. To the best of our knowledge, this is the first experimental evaluation of the state of the art in computing the tree edit distance.

The rest of the article is organized as follows. Section 2 provides background material. Section 3 generalizes previously proposed solutions for the tree edit distance and introduces the GTED algorithm. Section 4 defines the problem. In Section 5 we analyse path strategies for GTED and introduce the robust tree edit distance algorithm RTED in Section 6. In Section 7 we introduce the general single-path function and discuss the related index structures in Section 8. Section 9 surveys related work. We experimentally evaluate our solution in Section 10 and conclude in Section 11.

2. NOTATION AND BACKGROUND

We introduce our notation and recap the basic concepts of the tree edit distance computation.

2.1. Notation

A *tree* F is a directed, acyclic, connected graph with nodes $N(F)$ and edges $E(F) \subseteq N(F) \times N(F)$, where each node has at most one incoming edge. A *forest* F is a graph in which each connected component is a tree; each tree is also a forest. Each node has a *label*, which is not necessarily unique within the tree.

In an edge (v, w), node v is the *parent* and w is the *child*, $p(w) = v$. A node with no parent is the *root* node, a node without children is a *leaf*. Children of the same node are *siblings*. A node x is an ancestor of node v iff $x = p(v)$ or x is an ancestor of $p(v)$; x is a *descendant* of v iff v is an ancestor of x. $r_L(F)$ and $r_R(F)$ are the leftmost and rightmost root nodes in forest F, respectively; if F is a tree, then $r(F) = r_L(F) = r_R(F)$.

A *subforest* of a tree F is a graph with nodes $N' \subseteq N(F)$ and edges $E' = \{(v, w) : (v, w) \in E(F), v \in N', w \in N'\}$. F_v is the *subtree rooted in node* v of F iff F_v is a subforest of F and $N(F_v) = \{x : x = v$ or x is a descendant of v in $F\}$. A *path* γ in F is a connected subforest of F in which each node has at most one child.

The nodes of a forest F are strictly and totally ordered such that (a) $v < w$ for any edge $(v, w) \in E(F)$, and (b) for any two nodes f, g, if $f < g$ and f is not an ancestor of g, then $f' < g$ for all descendants f' of f. The tree traversal that visits all nodes in ascending order is the *left-to-right preorder* traversal. The *right-to-left preorder* visits the root node first and recursively traverses the subtrees rooted in the children of the root node in right-to-left preorder; the children are visited in descending node order. The *reverse left-to-right preorder* (*reverse right-to-left preorder*)[1] visits the nodes in descending left-

[1]Reverse right-to-left preorder is also known as "postorder".

to-right preorder (descending right-to-left preorder). A node v is *to the left* (right) of w iff $v < w$ ($v > w$) and v is not an ancestor (v is not a descendant) of w.

We use the following short notation: By $|F| = |N(F)|$ we denote the size of F, we write $v \in F$ for $v \in N(F)$, and denote the empty forest with $\emptyset$. $F - v$ is the forest obtained from F by removing node v and all edges at v. By $F - F_v$ we denote the forest obtained from F by removing subtree F_v. $F - \gamma$ is the set of subtrees of tree F obtained by removing path γ from F: $F - \gamma = \{F_v : v \notin \gamma \wedge p(v) \in \gamma\}$. $\mathit{left}(F, v)$ is the set of nodes in tree F to the left of node v, $\mathit{right}(F, v)$ is the set of nodes in F to the right of v, $\mathit{left}(\emptyset, v) = \mathit{right}(\emptyset, v) = \emptyset$.

In our graphical representation of trees and forests we omit the direction of the edges and show parent nodes above their children. The children of a node are shown from left to right in increasing node order.

Example 2.1. The nodes of tree F in Figure 1 are $N(F) = \{a, b, c, d, e, f, g, h, i, j, k, l, m\}$, the edges are $E(F) = \{(a, b), (b, c), (b, d), (d, e), (a, f), (f, g), (f, h), (h, i), (i, j), (i, k), (a, l), (l, m)\}$. The root of F is $r(F) = a$, and $|F| = 13$. F_b with nodes $N(F_b) = \{b, c, d, e\}$ and edges $E(F_b) = \{(b, c), (b, d), (d, e)\}$ is a subtree of F. $F_b - b$ is the subforest of F_b with $N(F_b - b) = \{c, d, e\}$ and $E(F_b - b) = \{(d, e)\}$, $r_L(F_b - b) = c$, $r_R(F_b - b) = d$. Let γ be the path from a to j (marked with a thick line). Then, $F - \gamma = \{F_b, F_g, F_k, F_l\}$. The left subscript of a node is its left-to-right and the right subscript its right-to-left preorder number. The reverse left-to-right preorder of the nodes in F is $m, l, k, j, i, h, g, f, e, d, c, b, a$, the reverse right-to-left preorder is $c, e, d, b, g, j, k, i, h, f, m, l, a$. $\mathit{left}(F_a, h) = \{g, e, d, c, b\}$, $\mathit{right}(F_a, h) = \{m, l\}$, $\mathit{left}(F_f, h) = \{g\}$, $\mathit{right}(F_f, h) = \emptyset$.

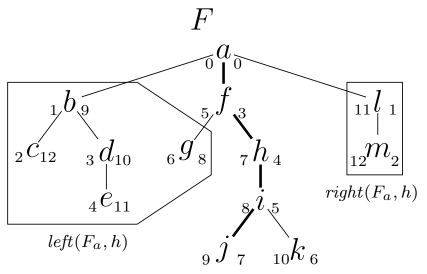


Fig. 1. Tree F for Example 2.1.

2.2. Recursive Solution for the Tree Edit Distance

The tree edit distance, $\delta(F, G)$, is defined as the minimum-cost sequence of node edit operations that transform F into G. We use the standard edit operations [Zhang and Shasha 1989; Klein 1998; Demaine et al. 2009]: *delete* a node and connect its children to its parent maintaining the order; *insert* a new node between an existing node v, and a consecutive subsequence of v's children; *rename* the label of a node (see Figure 2). The costs are $c_d(v)$ for deletion, $c_i(v)$ for insertion, and $c_r(v, w)$ for renaming v to w.

The tree edit distance has the recursive solution shown in Figure 3 [Zhang and Shasha 1989]. The distance between two forests F and G is computed in constant time from the solutions of the following four smaller subproblems: (1) $\delta(F - v, G)$, (2) $\delta(F, G - w)$, (3) $\delta(F_v - v, G_w - w)$, (4) $\delta(F - F_v, G - G_w)$. The nodes v and w are either both the leftmost ($v = r_L(F)$, $w = r_L(G)$) or both the rightmost ($v = r_R(F)$, $w = r_R(G)$) root

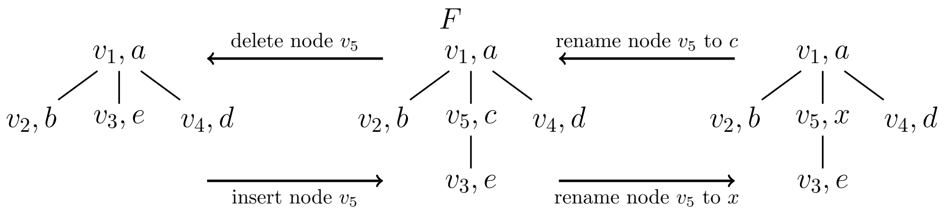


Fig. 2. Example of edit operations.

nodes of the respective forests. The subproblems that result from recursively decomposing F and G are called the *relevant subproblems*. The overall number of relevant subproblems depends on the choice of the nodes v and w at each recursive step.

$$\begin{aligned}
&\delta(\varnothing,\varnothing) = 0,\\
&\delta(F,\varnothing) = \delta(F-v,\varnothing) + c_d(v),\\
&\delta(\varnothing,G) = \delta(\varnothing,G-w) + c_i(w),\\
&\delta(F,G) = \min\begin{cases}\delta(F-v,G) + c_d(v) & (1)\\ \delta(F,G-w) + c_i(w) & (2)\\ \delta(F_v-v,G_w-w) + \delta(F-F_v,G-G_w) + c_r(v,w) & (3),(4)\end{cases}
\end{aligned}$$

Fig. 3. Recursive Formula for the Tree Edit Distance.

2.3. Dynamic Programming Algorithms

The fastest algorithms for the tree edit distance are dynamic programming implementations of the recursive solution. Since each subproblem is computed in constant time from other subproblems, the runtime complexity of these algorithms is equal to the number of different relevant subproblems they produce. Decomposing a tree with the recursive formula in Figure 3 can result in a quadratic number of subforests. Thus, the space complexity of a straightforward algorithm, which stores the distance between all pairs of subforests of two trees F and G, is $O(|F|^2|G|^2)$.

Fortunately, the required storage space can be reduced to $O(|F||G|)$ by computing the subproblems bottom-up and reusing space. To achieve this goal, the quadratic space algorithms restrict the choice of v and w in each recursive step. Two solutions have been presented in literature. The algorithm by Zhang and Shasha [1989] always chooses the same direction (both v and w are always rightmost root nodes). Demaine et al. [2009] switch between leftmost and rightmost root nodes depending on a predefined path.

While the recursive formula is symmetric, i.e., $\delta(F,G)$ and $\delta(G,F)$ produce the same set of subproblems, this does not necessarily hold for its dynamic programming implementations. The bottom-up strategies used in these algorithms compute and store subproblems for later use. When the direction is allowed to change, the subproblems that are required later are hard to predict and only a subset of the precomputed subproblems is actually used later. Thus, in addition to the direction, a decomposition strategy must also choose the order of the parameters.

3. GENERALIZATION OF THE TREE EDIT DISTANCE ALGORITHMS

In this section we introduce the GTED (general tree edit distance) algorithm, which abstracts from the specific strategies used by different approaches and generalizes most of the existing tree edit distance algorithms.

3.1. Relevant Subforests and Subtrees

The subforests that result from decomposing a tree with the recursive formula in Figure 3 are called the *relevant subforests*. The set of all subforests that can result from any decomposition is called the *full decomposition*. It results from repeated removal of the rightmost or leftmost root node. An example is shown in Figure 4(d).

Definition 3.1. The *full decomposition* of a tree F, $\mathcal{A}(F)$, is the set of all subforests of F obtained by recursively removing the leftmost and rightmost root nodes, $r_L(F)$ and $r_R(F)$, from F and the resulting subforests:

$$\begin{aligned}&\mathcal{A}(\varnothing) = \varnothing\\ &\mathcal{A}(F) = \{F\} \cup \mathcal{A}(F - r_L(F)) \cup \mathcal{A}(F - r_R(F))\end{aligned}$$

Next, we show how to decompose a tree into subtrees and subforest based on a so-called *root-leaf path*, a path that connects the root node of the tree to one of its leaves. The set of all root-leaf paths of F is denoted as $\gamma^*(F)$. The *left path*, $\gamma^L(F)$, the *right path*, $\gamma^R(F)$, and the *heavy path*, $\gamma^H(F)$, recursively connect a parent to its leftmost child, its rightmost child, or to the child which roots the largest subtree, respectively. In case of multiple subtrees of the same size, the heavy path connects to the root of the rightmost largest subtree. An *inner path* is a path that is neither left nor right. Decomposing trees with root-leaf paths is essential for the path strategies discussed in the next subsection.

The *relevant subtrees* of tree F for some root-leaf path γ are all subtrees that result from removing γ from F, i.e., all subtrees of F that are connected to a node on path γ.

Definition 3.2. The set of *relevant subtrees* of a tree F with respect to a root-leaf path $\gamma \in \gamma^*(F)$ is defined as $F - \gamma$.

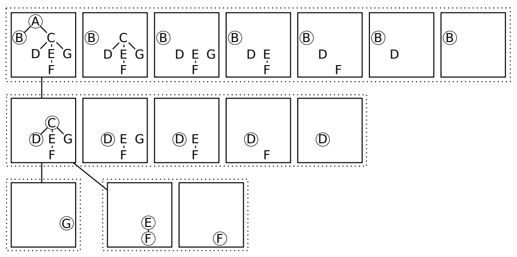
The *relevant subforests* of tree F for some root-leaf path are F itself and all subforests obtained by removing nodes in the following order: (1) remove the root of F and stop if the resulting forest is empty, (2) remove the rightmost root node in the resulting forest until the rightmost root node is on the root-leaf path, (3) remove the leftmost root node until the leftmost root node is on the root-leaf path, (4) recursively repeat the procedure for the resulting subtree.

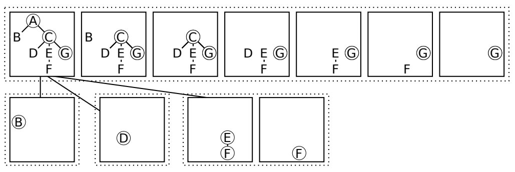
Definition 3.3. The set of *relevant subforests* of a tree F with respect to a root-leaf path $\gamma \in \gamma^*(F)$ is recursively defined as

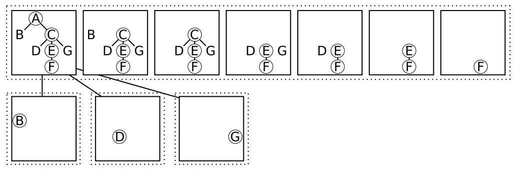
$$\begin{aligned}&\mathcal{F}(\varnothing,\gamma) = \varnothing\\ &\mathcal{F}(F,\gamma) = \{F\} \cup \begin{cases}\mathcal{F}(F - r_L(F),\gamma) & \text{if } r_R(F) \in \gamma\\ \mathcal{F}(F - r_R(F),\gamma) & \text{otherwise}\end{cases}\end{aligned}$$

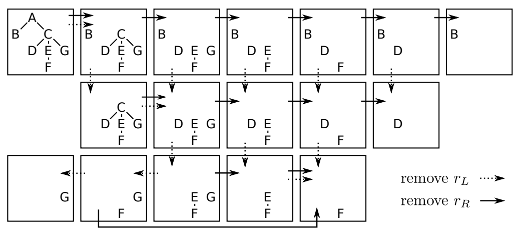
Example 3.4. The leftmost tree in each dotted box in Figure 4 is a relevant subtree. Each relevant subtree has a root-leaf path and the nodes on the root-leaf path are circled. The other forests in the box are the relevant subforests with respect to the root-leaf path. The solid lines connect trees and their relevant subtrees.

Recursive path decomposition. So far, we have considered relevant subtrees and subforests with respect to a single root-leaf path. If we assume that also for each of the


(a) Left path decomposition (15 relevant subforests).


(b) Right path decomposition (11 relevant subforests).


(c) Heavy path decomposition (10 relevant subforests).



(d) Full decomposition (17 subforests).

Fig. 4. Relevant subtrees and subforests.

resulting relevant subtrees of a tree F a root-leaf path is defined, we can recursively continue the decomposition as follows: (1) produce all relevant subforest of F with respect to some root-leaf path γ, (2) recursively apply this procedure to all relevant subtrees of F with respect to γ.

The set of paths that is used to recursively decompose F is called *path partitioning* of F and is denoted as $\Gamma(F)$. A path partitioning is a set of non-overlapping paths that all end in a leaf node and cover the tree. We define the left path partitioning of F to be $\Gamma^L(F) = \{\gamma^L(F)\} \cup \bigcup_{F_v \in F - \gamma^L(F)} \Gamma^L(F_v)$; the right path partitioning, $\Gamma^R(F)$, is defined analogously.

The set of relevant subforests that result from recursively decomposing tree F with the path partitioning Γ is

$$\mathcal{F}(F,\Gamma) = \mathcal{F}(F,\gamma_F) \cup \bigcup_{F' \in F - \gamma_F} \mathcal{F}(F',\Gamma), \tag{1}$$

where γ_F is the root-leaf path of F in Γ, i.e., the only element of $\gamma^*(F) \cap \Gamma$. The set of relevant subtrees that results from recursively decomposing tree F with the path partitioning Γ is

$$\mathcal{T}(F,\Gamma) = \{F\} \cup \bigcup_{F' \in F - \gamma_F} \mathcal{T}(F',\Gamma) \tag{2}$$

3.2. Path Strategies

A *decomposition strategy* chooses between leftmost and rightmost root node at each recursive step, resulting in a set of relevant subforests, which is a subset of the full decomposition. We define a class of decomposition strategies that uses root-leaf paths to decompose trees.

Definition 3.5. A *path strategy* S for two trees F and G is a mapping between pairs of subtrees from the two trees to a root-leaf path in one of the subtrees such that:

— (F, G) is mapped to one of the paths in $\gamma^*(F) \cup \gamma^*(G)$,
— if (F_v, G_w) is mapped to a path $\gamma \in \gamma^*(F_v)$, then all pairs $(F', G_w), F' \in F_v - \gamma$, are mapped to a path in $\gamma^*(F') \cup \gamma^*(G_w)$,
— if (F_v, G_w) is mapped to a path $\gamma \in \gamma^*(G_w)$, then all pairs $(F_v, G'), G' \in G_w - \gamma$ are mapped to a path in $\gamma^*(F_v) \cup \gamma^*(G')$.

An *LRH strategy* is a path strategy in which subtree pairs are mapped to left, right, or heavy paths.

A decomposition strategy determines the choice of left vs. right at each step in the recursive tree edit distance solution in Figure 3. A simple algorithm that uses a path strategy colours the nodes on the path and proceeds as follows: Initially, all nodes are white. If both F and G are trees with white root nodes, colour the nodes on path $S(F, G)$ in black, all other nodes in F and G are white. Let $B \in \{F, G\}$ be the tree/forest with a coloured root node b. If the rightmost root of B, $r_R(B)$, is black, then $v = r_L(F)$ and $w = r_L(G)$ in the recursive formula, otherwise $v = r_R(F)$ and $w = r_R(G)$.

3.3. Single-Path Functions

Space-efficient implementations of the tree edit distance use a bottom-up approach, which computes distances between smaller pairs of subtrees first. By carefully ordering the subtree computations, storage space can be reused without violating the preconditions for later subtree computations. The quadratic-space algorithms presented in literature are based on a distance matrix and a quadratic space function which fills the distance matrix.

The *distance matrix* D stores the distances between pairs of subtrees of F and G without their root nodes, i.e., $\delta(F_v - v, G_w - w)$, $v \in F$, $w \in G$ (cf. Figure 3). We represent D as a set of triples (F_v, G_w, d), where $d = \delta(F_v - v, G_w - w)$ is the tree edit distance between $F_v - v$ and $G_w - w$, $v \in F$, $w \in G$; the transposed distance matrix is defined as $D^T = \{(G_w, F_v, d) : (F_v, G_w, d) \in D\}$.

A *single-path function*, $\Delta(F, G, \gamma, D)$, computes distances between the subtrees of two trees F and G for a given root-leaf path $\gamma \in \gamma^*(F)$. More specifically, the single-path function between two trees F and G computes the distance between all subtrees of G and all subtrees of F that are rooted in the nodes of the root-leaf path γ. The precondition is that the tree distances (without root node) between all relevant subtrees of F with respect to γ and all subtrees of G are stored in a distance matrix D_{in}, i.e., $\{(F_v, G_w, d) : F_v \in F - \gamma, w \in G, d = \delta(F_v - v, G_w - w)\} \subseteq D_{in}$, which is part of the input. The output is $D_{out} = D_{in} \cup \{(F_v, G_w, d) : v \in \gamma_F, w \in G, d = \delta(F_v - v, G_w - w)\}$.

3.4. GTED: A General Algorithm for Path Strategies

The *general tree edit distance algorithm* (GTED, Algorithm 1) computes the tree edit distance for any path strategy S in quadratic space. The input is a pair of trees, F and G, the strategy S, and the distance matrix D, which initially is empty. The algorithm fills the distance matrix with the distances between all pairs of subtrees without root nodes from F and G (including $F - r(F)$ and $G - r(G)$).

GTED works as follows. For the pair of input trees (F, G), the root-leaf path γ is looked up in the strategy S. If γ is a path in the left-hand tree F, then

— GTED is recursively called for tree G and every relevant subtree of tree F with respect to path γ,
— a single-path function is called for the pair of input trees F, G and the path γ.

If γ is a path in the right-hand tree G, then GTED is called with the trees swapped. Conceptually, this requires to transpose the strategy S and the distance matrix D; finally, also the output must be transposed to get the original order of F and G in the distance matrix. In the implementation, the transposition is a flag that indicates the tree order.

The space complexity of GTED is quadratic. The size of the strategy is of at most $|F||G|$ because it maps unique pairs of subtrees to paths; the distance matrix is of the size $|F||G|$. The best single-path functions require $O(|F||G|)$ space and release the memory when they terminate. Only one single-path function is running at any time. The runtime complexity of the entire GTED algorithm depends on the strategy and can widely vary between $O(|F||G|)$ and $O(|F|^2|G|^2)$.

Algorithm 1: GTED(F, G, S, D)

```
1 $\gamma \leftarrow S(F, G)$
2 if $\gamma \in \gamma^*(F)$ then
3     foreach $F' \in F - \gamma$ do
4         $D \leftarrow D \cup$ GTED($F', G, S, D$)
5     $D \leftarrow D \cup \Delta(F, G, \gamma, D)$
6 else $D \leftarrow D \cup$ (GTED($G, F, S^T, D^T$))$^T$
7 return $D$
```

3.5. Generalization of Previous Work

Our general tree edit distance algorithm, GTED, generalizes much of the previous work on the tree edit distance, i.e., most algorithms presented in literature are equivalent to GTED with a specific path strategy and single-path function. Table I summarizes the details of how to turn GTED into the algorithms of Zhang and Shasha [1989], Klein [1998], and Demaine et al. [2009].

Table I. State-of-the-art algorithms.

Algorithm	Path strategy	Single-path function	Runtime
Zhang and Shasha [1989][2]	$\gamma^L(F_v)$ / $\gamma^R(F_v)$	Δ^L/ Δ^R	$O(n^4)$
Klein [1998][3]	$\gamma^H(F_v)$	Δ^A	$O(n^3 \log n)$
Demaine et al. [2009]	$\gamma^H(F_v)$ if $\|F_v\| \geq \|G_w\|$ $\gamma^H(G_w)$ otherwise	Δ^A	$O(n^3)$

Path Strategies. For each subtree pair (F_v, G_w), $v \in F$, $w \in G$, in the strategy, Table I lists the path to which (F_v, G_w) is mapped. Zhang and Shasha [1989] map all subtree pairs to the left path, $\gamma^L(F_v)$, which results in an algorithm with runtime $O(n^4)$ in the worst case. The symmetric strategy maps pairs of subtrees to the right path, $\gamma^R(F_v)$, with the same runtime complexity. Klein [1998] maps pairs of subtrees to the heavy path, $\gamma^H(F_v)$, achieving $O(n^3 \log n)$ runtime.

These strategies use only the root-leaf paths in F for the decomposition. Demaine et al. [2009] use both trees and map pairs of subtrees (F_v, G_w) to $\gamma^H(F_v)$ if $|F_v| \geq |G_w|$ and to $\gamma^H(G_w)$, otherwise. Thus, in each recursive step, the larger tree is decomposed using the heavy path. The resulting algorithm has $O(n^3)$ runtime.

[1]Zhang and Shasha [1989] discuss only a strategy which maps subtree pairs to left paths and introduce the Δ^L single-path function. Right path strategy and Δ^R are symmetric, and the extension is straightforward.
[3]We do not discuss the single-path function by Klein because its space complexity is $O(n^3)$. We focus on the quadratic-space single-path functions and use Δ^A to implement Klein's strategy.

Single-Path Functions. Three single-path functions with quadratic space complexity have been proposed in literature. Δ^A by Demaine et al. [2009] is called "compute period" in their paper and it is executed for every node on the path. Δ^L (and Δ^R) are adapted from the "tree distance function" by Zhang and Shasha [1989]. They differ along the following dimensions, which are relevant for an efficient implementation of RTED. We summarize the differences in Table II.

Table II. Key properties of different single-path functions.

	Δ^L	Δ^R	Δ^A
Path type	left	right	any
Cost left path	$\|F\|\|\mathcal{F}(G,\Gamma^L(G))\|$	—	$\|F\|\|\mathcal{A}(G)\|$
Cost right path	—	$\|F\|\|\mathcal{F}(G,\Gamma^R(G))\|$	$\|F\|\|\mathcal{A}(G)\|$
Cost heavy path	—	—	$\|F\|\|\mathcal{A}(G)\|$
Size of data structures	$\|F\|\|G\|$	$\|F\|\|G\|$	$\|F\|\|G\| + 2\|G\|^2$
Maximum fanout	unlimited	unlimited	2
Indexing (path type, space complexity)	left only $O(1)$	right only $O(1)$	any $O(\|G\|^2)$

Path type - the type of path (left, right, and any path) that can be processed. Δ^L and Δ^R are usable for left and right paths, respectively. They cannot process heavy paths, which is required for LRH strategies. Δ^A can process any path type, but left and right paths are processed very inefficiently, as will be discussed below.

Cost - the number of subproblems that must be computed to process one path. We discuss the costs of single-path functions in Section 5. The cost of Δ^L is $|F||\mathcal{F}(G,\Gamma^L(G))|$, the cost of Δ^R is $|F||\mathcal{F}(G,\Gamma^R(G))|$. This is the best known solution for left resp. right paths. The cost of Δ^A is $|F||\mathcal{A}(G)|$ disregarding the path type. This makes Δ^A inefficient for left and right paths since $\mathcal{F}(G,\Gamma^L(G))$ and $\mathcal{F}(G,\Gamma^R(G))$ are (typically small) subsets of $\mathcal{A}(G)$.

Size of data structures - the number and dimensions of quadratic size data structures that must be allocated concurrently. In addition to the quadratic size distance matrix D, which is maintained by the GTED algorithm and is an input to all single-path functions, the following data structures are required: Δ^L and Δ^R need a single matrix of size $|F||G|$. Δ^A needs a matrix of size $|F||G|$ and two matrices of size $|G|^2$. Thus, for $|F| = |G| = n$ (equally sized trees) the main memory requirement is n^2 for $\Delta^{L/R}$ vs. $3n^2$ for Δ^A.

Maximum fanout - maximum fanout that the single-path function can process. Δ^L and Δ^R can process trees of any fanout. Δ^A can process only trees with a maximum fanout of 2. Demaine et al. [2009] argue that an extension to an arbitrary fanout is possible without changing the asymptotic complexity, but no algorithm is given. The extension turns out to be tricky since the direction of processing nodes can change for each node on the path.

Indexing - indexing scheme used to identify subproblems for storage and retrieval of the intermediate results. An efficient technique for indexing subproblems is crucial since the dynamic programming algorithms for single-path functions must store and access intermediate results in constant time. Δ^L and Δ^R use an efficient technique for indexing subproblems, which does not require any additional storage structures. Unfortunately, this indexing scheme is not expressive enough for heavy paths. To process heavy paths in Δ^A, Demaine et al. devise a different indexing scheme. Unfortunately, each subproblem is identified by two different indexes depending on the processing direction, and translations between the indexes are frequently required within a single run of Δ^A. This problem has not been discussed in literature before. A solution that precomputes a translation matrix between the two indexes increases the space requirements by a matrix of size $|G|^2$. A detailed discussion of indexing schemes can be found in Section 7.

4. PROBLEM DEFINITION

A dynamic programming algorithm that implements the recursive solution of the tree edit distance must choose a direction (left or right) and the order of the input forests at each recursive step. The choices at each step form a strategy, which determines the overall number of subproblems that must be computed.

In this paper we introduce the class of *path strategies* (cf. Section 3). Path strategies can be expressed by a set of non-overlapping paths that connect tree nodes to leaves. The choice at each recursive step depends on the position of the paths. The class of path strategies is of particular interest since only for this class quadratic space solutions are known. A *path algorithm* computes the tree edit distance using a path strategy. An *LRH strategy* is a path strategy which uses only left, right, and heavy paths. LRH strategies are sufficient to express strategies with optimal asymptotic complexity. An algorithm based on an LRH strategy is an *LRH algorithm*. The most efficient tree edit distance algorithms presented in literature [Zhang and Shasha 1989; Klein 1998; Demaine et al. 2009] fall into the class of LRH algorithms.

The state of the art in path algorithms is not satisfactory. All previously proposed path algorithms degenerate, i.e., they run into their worst case although a better path strategy exists. Our experiments in Section 10 confirm that the difference can be of a polynomial degree, leading to highly varying and often infeasible runtimes. For example, the algorithm by Zhang and Shasha runs in $O(n^4)$ time for right branch (Figure 16(b)) and zig-zag trees (Figure 16(d)) for which the strategy by Demaine et al. requires $O(n^3)$ time. On the other hand, the strategy by Zhang and Shasha is optimal for left branch (Figure 16(a)) and full binary trees with $O(n^2 \log^2 n)$ runtime, whereas the solutions by Demaine et al. needs $O(n^3)$ time for these tree shapes.

The goal of this paper is to develop a new path algorithm which combines the features of previously proposed algorithms (worst case guarantees for time and space) and in addition is robust, i.e., the algorithm adapts to the input instances and applies an appropriate strategy. More specifically, the algorithm should have the following properties:

— *space-efficient:* the space complexity should be $O(n^2)$, which is the best known complexity for a tree edit distance algorithm;
— *optimal runtime:* the runtime complexity should be $O(n^3)$, which has been shown to be optimal among all possible strategies for the recursive formula in Figure 3 [Demaine et al. 2009];
— *efficient single-path function:* (a) any path type (left, right, and inner path) of arbitrary-fanout trees should be supported; (b) the cost for a path should be at most the cost of the best existing single-path functions (Δ^L, Δ^R, Δ^A) suitable for that path; (c) the translation matrix for heavy paths increases the memory requirement by 50% and should be avoided; (d) the indexing scheme should uniquely identify a subforest, the representation should be independent of the direction in which the nodes are currently processed in the algorithm, and the numeric identifiers should be dense such that the rows and columns in the memoization table can be addressed;
— *robust:* the algorithm should use the optimal LRH strategy, which ensures the minimum distance computation cost for any input disregarding the tree shape.

Our solution is the RTED algorithm, which combines the *optimal LRH strategy* with the *general single-path function*. We show that, for any instance, the number of subproblems computed by RTED is smaller or equal to the number of subproblems computed by all previously proposed LRH algorithms.

5. COST OF PATH STRATEGIES

In this section we count the number of relevant subproblems that must be computed by different single-path functions and the overall GTED algorithm. We develop a cost formula, which counts the number of relevant subproblems of the optimal LRH strategy for any pair of input trees.

5.1. Relevant Subproblems

A *relevant subproblem* is a pair of relevant subforest, for which a distance must be computed during the recursive tree edit distance evaluation. The number of relevant subproblems depends on the decomposition strategy and determines the runtime complexity of the respective algorithm. The set of relevant subproblems is always a subset of $\mathcal{A}(F) \times \mathcal{A}(G)$.

5.2. Cost of the Single-Path Functions

We count the number of relevant subproblems which the single-path functions, Δ^L, Δ^R, and Δ^A, compute for a pair of trees, F and G, and a given path $\gamma \in \gamma^*(F)$. We need to derive the number of subforests in the full decomposition of a tree, the decomposition with a single path, and the recursive decomposition with a set of paths.

LEMMA 5.1. *The number of the subforests in the full decomposition of tree F is $|\mathcal{A}(F)| = \frac{|F|(|F|+3)}{2} - \sum_{v \in F} |F_v|$.*

PROOF. Proof by [Dulucq and Touzet 2005]. □

LEMMA 5.2. *The number of relevant subforests of tree F with respect to a root-leaf path $\gamma \in \gamma^*(F)$ is equal to the number of nodes in F, $|\mathcal{F}(F,\gamma)| = |F|$.*

PROOF. We show $|\mathcal{F}(F,\gamma)| = |F|$ for the general case when F is a forest and γ is a root-leaf path in one of the trees (connected components) of F. Then, $|\mathcal{F}(F,\gamma)| = |F|$ is also true for F being a tree. The proof is by induction on the size of F. *Base case:* For $|F| = 1$, $|\mathcal{F}(F,\gamma)| = |F|$ holds due to Definition 3.3 of $\mathcal{F}(F,\gamma)$. *Inductive hypothesis:* We assume, that $|\mathcal{F}(F,\gamma)| = |F|$ holds for all forests F_k of size k. Then, for a forest F_{k+1} of size $k+1$ with Definition 3.3: $|\mathcal{F}(F_{k+1},\gamma)| = |\{F_{k+1}\} \cup \mathcal{F}(F_{k+1} - v,\gamma)| = 1 + |\mathcal{F}(F_{k+1} - v,\gamma)| = 1 + k$ since $|F_{k+1} - v| = k$. □

With the results of Lemma 5.1 and 5.2, the cardinalities of both $\mathcal{A}(F)$ and $\mathcal{F}(F,\gamma)$ are independent of the path used to decompose tree F and can be computed in linear time in a single tree traversal for F and all its subtrees.

LEMMA 5.3. *The number of relevant subforests produced by a recursive path decomposition of tree F with path partitioning Γ is the sum of the sizes of all the relevant subtrees in the recursive decomposition:*

$$|\mathcal{F}(F,\Gamma)| = \sum_{F' \in \mathcal{T}(F,\Gamma)} |F'|$$

PROOF. Follows from the definition of $\mathcal{F}(F,\Gamma)$ in Equation (1) and Lemma 5.2. □

Example 5.4. Figure 4 shows different recursive path decompositions. All path decompositions in the figure produce different numbers of relevant subforests, and they all produce less subforests than the full decomposition in Figure 4(d).

The next lemma counts the number of relevant subproblems that the different single-path functions must compute.

LEMMA 5.5. *The number of relevant subproblems computed by the single-path functions Δ^A, Δ^L, and Δ^R, for a pair of trees F and G is as follows (D is the distance matrix):*

$\Delta^A(F, G, \gamma, D)$, $\gamma \in \gamma^*(F)$	:	$\|F\|\|\mathcal{A}(G)\|$
$\Delta^L(F, G, \gamma^L(F), D)$	:	$\|F\|\|\mathcal{F}(G, \Gamma^L(G))\|$
$\Delta^R(F, G, \gamma^R(F), D)$	:	$\|F\|\|\mathcal{F}(G, \Gamma^R(G))\|$

PROOF. Demaine et al. [2009] show that the number of subproblems computed by $\Delta^A(F, G, \gamma, D)$, $\gamma \in \gamma^*(F)$, is $|\mathcal{F}(F, \gamma)||\mathcal{A}(G)|$. Zhang and Shasha [1989] show the number of relevant subproblems for $\Delta^L(F, G, \gamma^L(F), D)$ to be $|\mathcal{F}(F, \gamma^L(F))||\mathcal{F}(G, \Gamma^L(G))|$; $|\mathcal{F}(F, \gamma^L(F))| = |F|$ follows from Lemma 5.2. The same rationale holds for Δ^R. □

5.3. Cost of the Optimal LRH Strategy

With the results in Section 5.2 we can compute the cost of the GTED algorithm for any path strategy, i.e., the number of relevant subproblems that GTED must compute for a given strategy. The overall cost of a strategy is computed by decomposing a tree into its relevant subtrees according to the paths in the strategy and summing up the costs for executing a single-path function for each pair of relevant subtrees.

We compute the cost of the *optimal* LRH strategy given the single-path functions Δ^L, Δ^R, and Δ^A. This is achieved by an exhaustive search in the space of all possible LRH strategies for a given pair of trees. At each recursive step there are six choices. Either F is decomposed by the left, right, or heavy path, or G is decomposed. For each of the six choices, the cost of the relevant subtrees that result from the decomposition must be explored. The optimal strategy is computed by the formula in Figure 5, which chooses the minimum cost at each recursive step. The cost formula counts the exact number of the relevant subproblems for the optimal LRH strategy between two trees.

$$\mathrm{cost_{LRH}}(F, G) = \min \begin{cases} |F||\mathcal{A}(G)| + \sum_{F' \in F - \gamma^H(F)} \mathrm{cost_{LRH}}(F', G) \\ |G||\mathcal{A}(F)| + \sum_{G' \in G - \gamma^H(G)} \mathrm{cost_{LRH}}(G', F) \\ |F||\mathcal{F}(G, \Gamma^L(G))| + \sum_{F' \in F - \gamma^L(F)} \mathrm{cost_{LRH}}(F', G) \\ |G||\mathcal{F}(F, \Gamma^L(F))| + \sum_{G' \in G - \gamma^L(G)} \mathrm{cost_{LRH}}(G', F) \\ |F||\mathcal{F}(G, \Gamma^R(G))| + \sum_{F' \in F - \gamma^R(F)} \mathrm{cost_{LRH}}(F', G) \\ |G||\mathcal{F}(F, \Gamma^R(F))| + \sum_{G' \in G - \gamma^R(G)} \mathrm{cost_{LRH}}(G', F) \end{cases}$$

Fig. 5. Cost of the optimal LRH strategy.

THEOREM 5.6. *Given the single-path functions Δ^L, Δ^R, and Δ^A, the cost formula in Figure 5 computes the cost of the optimal LRH strategy.*

PROOF. Proof by induction over the structures of F and G. *Base case:* both, F and G, consist of a single node and $\text{cost}_{\text{LRH}}(F,G) = 1$. *Inductive hypothesis:* $\text{cost}_{\text{LRH}}(F',G)$ and $\text{cost}_{\text{LRH}}(F,G')$ for all relevant subtrees F' and G' with respect to the left, right, and heavy path of F and G are optimal. We show that the optimality of $\text{cost}_{\text{LRH}}(F,G)$ follows. There are six possible paths for decomposing F and G: left, right, or heavy in either F or G. If γ is a path in F, the distance of G to all relevant subtrees of F with respect to γ must be computed, which can be done with an optimal cost of $\sum_{F' \in F-\gamma} \text{cost}_{\text{LRH}}(F',G)$ (inductive hypothesis). Further, depending on the path type of γ, a single-path function must be called: Δ^L for the left path, Δ^R for the right path, and Δ^A for the heavy path. The costs follow from Lemma 5.5 and are added to the cost sum for the relevant subtrees. Calling Δ^A for left or right paths cannot lead to smaller cost since $\mathcal{F}(F,\Gamma^L(F)) \subseteq \mathcal{A}(F)$ and $\mathcal{F}(F,\Gamma^R(F)) \subseteq \mathcal{A}(F)$. Similar rationale holds if γ is a path in G. The cost formula in Figure 5 takes the minimum of the six possible costs. $\square$

The single-path function, Δ^A, which is used for heavy paths in GTED, is not optimal since it computes subproblems that are not used later. It is, however, the only known algorithm for heavy paths that runs in $O(n^2)$ space. Substituting Δ^A with another single-path function requires changing the respective costs in the cost formula.

6. RTED: ROBUST TREE EDIT DISTANCE ALGORITHM

The *robust tree edit distance algorithm*, RTED, computes the optimal LRH strategy for two trees. The optimal strategy is executed with the GTED algorithm presented in Section 3 and the efficient *general signle-path function* introduced in Section 7.

The optimal LRH strategy for two trees, F and G, is computed with the cost formula in Figure 5. Since an LRH strategy maps each pair of relevant subtrees to at most one path, the strategy can be stored in a matrix of size $|F||G|$, the *strategy matrix*, which we initialize with empty paths. After the strategy computation, the strategy matrix maps any pair of subtrees to a root-leaf path in one of the subtrees, thus the LRH strategy is a subset of the strategy matrix (cf. Definition 3.5).

The key challenge is to compute the optimal strategy efficiently. The exhaustive exploration of the search space of exponential size is obviously prohibitive. We observe, however, that the cost is computed only between pairs of subtrees of F and G, and there is only a quadratic number of such subtree pairs. This suggests a memoization algorithm, which stores the intermediate results and traverses identical branches of the search tree only once.

6.1. Baseline Algorithm for Optimal LRH Strategy

The baseline algorithm for computing the optimal LRH strategy implements the cost formula in Figure 5 and traces back the optimal strategy. It uses memoization to avoid computing the strategy for identical pairs of subtrees more than once. The intermediate results are stored in a *memoization matrix* of size $|F||G|$, and the optimal strategy is stored in the strategy matrix. At each recursive step, four actions are performed for F and G:

— If the cost for F and G is in the memoization matrix, return the cost and ignore the following steps.
— Compute the costs for G (F) and the relevant subtrees of F (G) w.r.t. the left, right, and heavy paths; sum up the values and compute the cost for each path.
— Store the path with smallest cost as the entry for F and G in the strategy matrix.
— Store the cost for F and G in the memoization matrix.

THEOREM 6.1. *The runtime complexity of the baseline algorithm for two trees F and G is bound by $O(n^3)$, $n = \max(|F|, |G|)$, and the bound is tight.*

PROOF. The runtime of the baseline algorithm is determined by the number of sums that must be computed. We proceed in three steps: (1) count the number of the summations, (2) show the $O(n^3)$ upper bound for the complexity, (3) give an instance for which this bound is tight.

(1) *Summation count*: The cost for a subtree pair (F_v, G_w) is the minimum of six values (cf. Figure 5). Computing each value requires $|F - \gamma|$ summations: The cost of the single-path function (product) is computed in constant time since the factors can be precomputed in $O(|F| + |G|)$ time with Lemmas 5.1 and 5.3; adding the costs for the relevant subtrees w.r.t. path γ requires $|F - \gamma| - 1$ summations. Since we store the results, the cost for each pair (F_v, G_w), $v \in F$, $w \in G$, must be computed at most once. For some subtree pairs no cost might be computed, for example, when the root node of one of the subtrees is the only child of its parent. Overall, this leads to an upper bound for the number of summations:

$$\begin{aligned}\#sums \leq \sum_{v \in F, w \in G} (&|F_v - \gamma^L(F_v)| + |F_v - \gamma^R(F_v)| + |F_v - \gamma^H(F_v)| + \\ &|G_w - \gamma^L(G_w)| + |G_w - \gamma^R(G_w)| + |G_w - \gamma^H(G_w)|)\end{aligned} \tag{3}$$

(2) *Upper bound*: The number of relevant subtrees of F_v with respect to $\gamma \in \gamma^*(F_v)$ is limited by the number of leaf nodes of F_v, $|F_v - \gamma| \leq |l(F_v)| - 1$. Substituting in (3) we get a new (looser) upper bound, $\#sums \leq \sum_{v \in F, w \in G}(3(|l(F_v)| - 1) + 3(|l(G_w)| - 1))$. There are at most $|F||G|$ different pairs of subtrees, $|l(F_v)| \leq |F|$, $|l(G_w)| \leq |G|$, thus $\#sums \leq |F||G|(3|F| + 3|G|) = O(n^3)$.

(3) *Tightness of the bound*: We show that for some instances the runtime of the baseline algorithm is $\Omega(n^3)$. Let F be a left branch tree (cf. Figure 16(a)) and G a right branch tree (cf. Figure 16(b)). For a subtree F_v rooted in a non-leaf node v of F, it holds that $|F_v - \gamma^L(F_v)| = \frac{|F_v|-1}{2}$, $|F_v - \gamma^H(F_v)| = \frac{|F_v|-1}{2}$, $|F_v - \gamma^R(F_v)| = 1$; if v is a leaf, then $|F_v - \gamma| = 0$ for any path. Similarly, for subtree G_w and a non-leaf node w, $|G_w - \gamma^L(G_w)| = 1$, $|G_w - \gamma^H(G_w)| = \frac{|G_w|-1}{2}$, $|G_w - \gamma^R(G_w)| = \frac{|G_w|-1}{2}$; $|G_w - \gamma| = 0$ if w is a leaf. For F and G, every pair of subtrees occurs during the computation of the baseline algorithm, thus the right-hand term in (3) is the exact number of summations. By substituting in (3) we get:

$$\begin{aligned}\#sums = &\sum_{v \in F \setminus l(F), w \in G \setminus l(G)} \Big(\frac{|F_v| - 1}{2} + 1 + \frac{|F_v| - 1}{2} + \\ &\qquad 1 + \frac{|G_w| - 1}{2} + \frac{|G_w| - 1}{2}\Big) + \\ &\sum_{v \in l(F), w \in G \setminus l(G)} \Big(1 + \frac{|G_w| - 1}{2} + \frac{|G_w| - 1}{2}\Big) + \\ &\sum_{v \in F \setminus l(F), w \in l(G)} \Big(\frac{|F_v| - 1}{2} + 1 + \frac{|F_v| - 1}{2}\Big)\end{aligned}$$

$l(F)$ and $l(G)$ are leaf nodes, $F \setminus l(f)$ and $G \setminus l(G)$ are the non-leaf nodes of F and G, respectively. With $|l(F)| = (|F| + 1)/2$, $|l(G)| = (|G| + 1)/2$, we get $\#sums = \Omega(\frac{|F|}{2}\frac{|G|}{2}(|F| + |G|) + \frac{|F|}{2}\frac{|G|}{2}|G| + \frac{|F|}{2}\frac{|G|}{2}|F|) = \Omega(n^3)$. □

Algorithm 2: OptLRHStrategy(F, G)

```
1  $L_v, R_v, H_v$ : arrays of size $|F||G|$
2  $L_w, R_w, H_w$ : arrays of size $|G|$
3  for $v = |F|$ to $1$ in reverse left-to-right preorder do
4      for $w = |G|$ to $1$ in reverse left-to-right preorder do
5          if $w$ is leaf then $L_w[w] \leftarrow R_w[w] \leftarrow H_w[w] \leftarrow 0$
6          if $v$ is leaf then $L_v[v,w] \leftarrow R_v[v,w] \leftarrow H_v[v,w] \leftarrow 0$
7          $C \leftarrow \{(|F_v||\mathcal{A}(G_w)| + H_v[v,w], \gamma^H(F_v)),$
8                 $(|G_w||\mathcal{A}(F_v)| + H_w[w], \gamma^H(G_w)),$
9                 $(|F_v||\mathcal{F}(G_w, \Gamma^L(G))| + L_v[v,w], \gamma^L(F_v)),$
10                $(|G_w||\mathcal{F}(F_v, \Gamma^L(F))| + L_w[w], \gamma^L(G_w)),$
11                $(|F_v||\mathcal{F}(G_w, \Gamma^R(G))| + R_v[v,w], \gamma^R(F_v)),$
12                $(|G_w||\mathcal{F}(F_v, \Gamma^R(F))| + R_w[w], \gamma^R(G_w))\}$
13         $(c_{min}, \gamma_{min}) \leftarrow (c, \gamma)$ such that $(c,\gamma) \in C$ and $c = \min\{c' : (c', \gamma') \in C\}$
14         $STR[v,w] \leftarrow \gamma_{min}$
15         if $v$ is not root then
16             $L_v[p(v),w] \mathrel{+}= \begin{cases} L_v[v,w] & \text{if } v \in \gamma^L(F_{p(v)}) \\ c_{min} & \text{otherwise} \end{cases}$
17             $R_v[p(v),w] \mathrel{+}= \begin{cases} R_v[v,w] & \text{if } v \in \gamma^R(F_{p(v)}) \\ c_{min} & \text{otherwise} \end{cases}$
18             $H_v[p(v),w] \mathrel{+}= \begin{cases} H_v[v,w] & \text{if } v \in \gamma^H(F_{p(v)}) \\ c_{min} & \text{otherwise} \end{cases}$
19         if $w$ is not root then
20             $L_w[p(w)] \mathrel{+}= \begin{cases} L_w[w] & \text{if } w \in \gamma^L(G_{p(w)}) \\ c_{min} & \text{otherwise} \end{cases}$
21             $R_w[p(w)] \mathrel{+}= \begin{cases} R_w[w] & \text{if } w \in \gamma^R(G_{p(w)}) \\ c_{min} & \text{otherwise} \end{cases}$
22             $H_w[p(w)] \mathrel{+}= \begin{cases} H_w[w] & \text{if } w \in \gamma^H(G_{p(w)}) \\ c_{min} & \text{otherwise} \end{cases}$
23 return $STR$
```

6.2. OptLRHStrategy: Efficient Algorithm for Optimal LRH Strategy

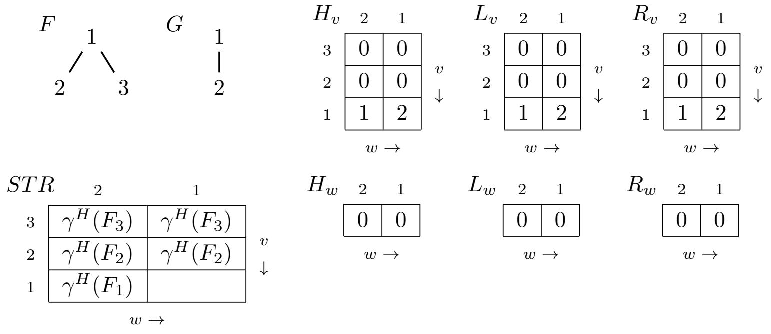
The runtime complexity of the baseline algorithm is $O(n^3)$. While this is clearly a major improvement over the naive exponential solution, it is unfortunately not enough in our application. The runtime complexity for computing the strategy must not be higher than the complexity of the optimal strategy for GTED. The optimal GTED strategy is often better than cubic, e.g., $O(n^2 \log^2 n)$ for trees of depth $\log(n)$.

In this section we introduce OptLRHStrategy (Algorithm 2), a dynamic-programming algorithm which computes the optimal LRH strategy for GTED in $O(n^2)$ time. Similar to the base line algorithm, a strategy matrix STR of quadratic size is used to store the best path at each recursive step, resulting in the best overall strategy.

Different from the base line algorithm, we do not store costs between individual pairs of relevant subtrees. Instead, we maintain and incrementally update the cost sums of the relevant subtrees (summations over the relevant subtrees in the cost formula). We do not sum up the same cost multiple times and thus reduce the runtime. The cost sums are stored in *cost matrices*: L_v, R_v, H_v of size $|F||G|$ store a cost sum for each pair (F_v, G_w), $v \in F$, $w \in G$, for the left, right, and heavy path in F_v, respectively; for example, $L_v[F_v, G_w] = \sum_{F' \in F_v - \gamma^L(F_v)} \text{cost}_{\text{LRH}}(F', G_w)$ (cf. Figure 5). L_w, R_w, H_w of size $|G|$ store the cost sums between all relevant subtrees of G w.r.t. the corresponding path and a specific subtree F_v; for example, $L_w[G_w] = \sum_{G' \in G_w - \gamma^L(G_w)} \text{cost}_{\text{LRH}}(G', F_v)$.

Algorithm 2 iterates over every pair of subtrees (F_v, G_w) in reverse left-to-right preorder of the nodes $v \in F$, $w \in G$. In this ordering we start with the last node in the left-to-right preorder. Such ordering ensures that children are processed before their parents. The cost sum for a leaf node is zero, because leaves do not have relevant subtrees (lines 5–6). The values in the cost matrices are used to compute the costs of the pair (F_v, G_w) for each of the six possible paths in the cost formula (lines 7–12). For each result a pair $(cost, path)$ is stored in the temporary set C. The minimum cost in C is assigned to c_{min}, the respective path γ_{min} is stored in the strategy matrix STR (lines 13–14). Finally, the cost sums for the subtree pairs $(F_{p(v)}, G_w)$ and $(F_v, G_{p(w)})$ are updated, where $p(v)$ and $p(w)$ are the parents of v and w, respectively. The update depends on whether v and w belong to the same path as their parent (lines 15–22).

Example 6.2. We use Algorithm 2 to compute the optimal strategy for two trees F and G, $N(F) = \{1,2,3\}$, $E(F) = \{(1,2),(1,3)\}$, $N(G) = \{1,2\}$, $E(G) = \{(1,2)\}$; for simplicity, node IDs and labels are identical and correspond to the left-to-right preorder position in the tree. Figure 6 shows the cost matrices and the strategy matrix before the last node pair, $v = 3$, $w = 2$, is processed. Rows and columns are labelled with node IDs, e.g., the cost sum for the subtree pair (F_1, G_2) w.r.t. the heavy path in F_1 is $H_v[1,2] = 1$. We compute the missing value in the strategy matrix STR: Neither v nor w are leaves; with $|G_w| = |\mathcal{A}(G_w)| = |\mathcal{F}(G_w, \Gamma_L(G))| = |\mathcal{F}(G_w, \Gamma_R(G))| = 2$, $|F_v| = 3$, $|\mathcal{A}(F_v)| = |\mathcal{F}(F_v, \Gamma_L(F))| = |\mathcal{F}(F_v, \Gamma_R(F))| = 4$ we compute $C = \{(3{\cdot}2{+}2, \gamma^H(F_1))$, $(2{\cdot}4{+}0, \gamma^H(G_1))$, $(3{\cdot}2{+}2, \gamma^L(F_1))$, $(2{\cdot}4{+}0, \gamma^L(G_1))$, $(3{\cdot}2{+}2, \gamma^R(F_1))$, $(2{\cdot}4{+}0, \gamma^R(G_1))\} =$ $\{(8, \gamma^H(F_1))$, $(8, \gamma^H(G_1))$, $(8, \gamma^L(F_1))$, $(8, \gamma^L(G_1))$, $(8, \gamma^R(F_1))$, $(8, \gamma^R(G_1))\}$. In the example, all costs are identical, and we arbitrarily pick $(c_{min}, \gamma_{min}) = (8, \gamma^H(F_1))$ as the minimum; the missing value in the strategy matrix is $\gamma^H(F_1)$. Since both v and w are roots, the algorithm terminates and returns the optimal strategy.



$$\begin{array}{c|cc} H_v & 2 & 1 \\ \hline 3 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 2 \end{array} \quad \begin{array}{c|cc} L_v & 2 & 1 \\ \hline 3 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 2 \end{array} \quad \begin{array}{c|cc} R_v & 2 & 1 \\ \hline 3 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 2 \end{array}$$

$$\begin{array}{c|cc} STR & 2 & 1 \\ \hline 3 & \gamma^H(F_3) & \gamma^H(F_3) \\ 2 & \gamma^H(F_2) & \gamma^H(F_2) \\ 1 & \gamma^H(F_1) & \end{array} \quad \begin{array}{c|cc} H_w & 2 & 1 \\ \hline & 0 & 0 \end{array} \quad \begin{array}{c|cc} L_w & 2 & 1 \\ \hline & 0 & 0 \end{array} \quad \begin{array}{c|cc} R_w & 2 & 1 \\ \hline & 0 & 0 \end{array}$$

Fig. 6. Cost matrices and the strategy matrix.

THEOREM 6.3. *OptLRHStrategy (Algorithm 2) is correct.*

PROOF. The strategy matrix STR maps every pair of subtrees to a root-leaf path and thus is a superset of a path strategy according to Definition 3.5. Next, we show by induction that the cost matrices store the correct values according to the cost formula. *Base case*: For a pair of leaf nodes (v, w) the cost matrices are zero; this is correct due to $F_v - \gamma_{F_v} = G_w - \gamma_{G_w} = \emptyset$ for any $\gamma_{F_v} \in \gamma^*(F_v)$ and $\gamma_{G_w} \in \gamma^*(G_w)$. *Inductive hypothesis*: For all pairs of children (v, w) of two nodes, $f = p(v)$ and $g = p(w)$, the values in the cost matrices are correct. We show that, after processing all children, the values for f and

g are correct. This implies the correctness of the overall algorithm since the nodes are processed in reverse left-to-right preorder, i.e., all children are processed before their parents.

We consider two cases for node v (for w analogous reasoning holds): (1) node v lies on the same left (right, heavy) root-leaf path γ_{F_f} as its parent, (2) nodes v does not lie on the same left (right, heavy) root-leaf path γ_{F_f} as its parent. *Case 1*: F_v is not a relevant subtree of F_f with respect to γ_{F_f}. The cost already stored for v is the sum of the costs for every relevant subtree $F' \in F_v - \gamma_{F_v}$, i.e., a part of the sum $\sum_{F' \in F_f - \gamma_{F_f}} \text{cost}(F', G_w)$. We increment the value in the cost matrix of γ_{F_f} with the cost already stored for v. *Case 2*: $F_v \in F_f - \gamma_{F_f}$ is the root of some relevant subtree of F_f with respect to γ_{F_f}. The cost of v is an element in the cost sum for f, $\sum_{F' \in F_f - \gamma_{F_f}} \text{cost}(F', G_w)$. We add the cost value of v (c_{min}) to the cost entry of f. □

THEOREM 6.4. *The time and space complexities of OptLRHStrategy (Algorithm 2) are $O(n^2)$, $n = \max(|F|, |G|)$.*

PROOF. Algorithm 2 iterates over all pairs of subtrees (F_v, G_w), $v \in F$, $w \in G$, thus the innermost loop is executed $|F||G|$ times. In the inner loop we do a constant number of array lookups and sums. The factors of the six products in lines 7–12 are precomputed in $O(|F| + |G|)$ time and space using the lemmas in Section 5.2. We use four matrices of size $|F||G|$ and three matrices of size $|G|$, thus the overall complexity is $O(|F||G|)$ in time and space. □

7. THE GENERAL SINGLE-PATH FUNCTION

The core of all tree edit distance algorithms is the single-path function, which processes one path of the strategy. In Section 3 we analyse the single-path functions Δ^L, Δ^R, and Δ^A, and show that they are not suitable for implementing RTED efficiently. In Section 4 we identify requirements for an efficient single-path function: (a) any path type of arbitrary-fanout trees should be supported; (b) a path should be processed with the minimum cost among the cost of Δ^L, Δ^R, and Δ^A for that path type; (c) translation matrices should be avoided; (d) an efficient indexing scheme of subproblems should be used to guarantee constant time access.

The first two requirements can be satisfied by calling Δ^L for left paths, Δ^R for right paths, and Δ^A for all other paths.[4] This, however, does not solve issues (c) and (d).

In this section we present the *general single-path function*, which satisfies all the requirements identified above. We first analyse the existing indexing schemes for subproblems and show their limitations. The indexing is crucial for the efficient storage and retrieval of intermediate results. We introduce a novel forest representation, *root encoding*, which allows us to uniquely identify all required subforests. The indexing scheme based on our root encoding solves the problems of the previously proposed single-path functions. We order the subproblems to achieve a quadratic-space dynamic programm. Our subproblem ordering extends the forest enumerations by Demaine et al. [2009] and guarantees optimal runtime for left and right paths. Overall, the number of subproblems processed by our general single-path function is minimal among all known single-path functions.

7.1. Existing Indexing Schemes

A subproblem is a pair of subforests. Δ^L and Δ^R index subforests as postorder prefixes of a specific subtree. Since the memoization table used for $\Delta^{L/R}$ only stores subprob-

[4]In order to do so, Δ^L and Δ^R must be modified to store intermediate results for subtrees *without* root node. Zhang and Shasha [1989] store intermediate results for subtrees *with* root node.

lems for a specific subtree pair from F and G at any point in time, a subproblem is uniquely identified by two postorder prefix lengths, which are used as row resp. column numbers to access the memoization table. Unfortunately, this indexing scheme is not expressive enough for heavy paths.

Δ^A indexes a subforest of tree G as a quadruple (G_v, p, ld, rd), where G_v is a subtree of G, p is the precedence (left-before-right or right-before-left), ld is the number of left deletions, and rd is the number of right deletions. If the precedence is left-before-right, the subforest (G_v, p, ld, rd) is given by first removing the leftmost root node ld times from G_v (or the resulting subforest), followed by rd deletions of the rightmost root node; otherwise right deletions precede left deletions. The precedence matters, i.e., doing left deletions before right deletions leads to a different subforest than doing right deletions first. As an example consider forest G_{l_1,r_1} (the black nodes in the left-hand tree) in Figure 7. To obtain G_{l_1,r_1} from the initial tree, we need 2 left and 3 right deletions if p = left-before-right, and 4 right plus 1 left deletions otherwise.

The problem with the indexing scheme of Δ^A is that there is no simple way to translate left-before-right to right-before-left indexes. This translation is required each time the precedence changes: the intermediate results of the previous steps must be retrieved (old precedence) and new results must be stored (new precedence). Δ^A must switch the precedence depending on whether nodes to the left or to the right of the path are being processed, i.e., the precedence potentially changes for each node on the path. This problem is not addressed in literature. A straightforward approach reorders the memoization tables each time the precedence changes so that the subproblem distances can be accessed with another index. This, however, requires $2|\gamma||G|^2$ operations for a single run of Δ^A in the worst case (where $|\gamma|$ is the length of the path), and additional $|G|^2$ space for temporarily storing the values while reordering. A better approach computes a translation matrix of size $|G|^2$, where G is the input tree to the single-path function that is *not* decomposed with a path. The translation matrix is computed by enumerating all subforests from the full decomposition of G, $\mathcal{A}(G)$. Unfortunately, also this solution is not satisfactory. The translation matrix must be computed for each call of the single-path function and increases the memory requirements of Δ^A by 50% to $|F||G| + 2|G|^2$.

We develop a new indexing scheme which uses a data structure of linear size to access the intermediate results in constant time. The only overhead for the runtime is in the index precomputation which is done in linear time for the entire run of the RTED algorithm.

7.2. Root Encoding

We present the *root encoding*, a novel indexing scheme for subforests. The root encoding uniquely identifies all subforests that can result from decomposing two trees using the recursive formula in Figure 3, i.e., all subforests required during the computation of the single-path function for any path type.

Definition 7.1. Let the *leftmost root node* l_F and the *rightmost root node* r_F be two nodes of tree F, $l_F \leq r_F$. The *root encoding* F_{l_F,r_F} defines a subforest of F with nodes $N(F_{l_F,r_F}) = \{l_F, r_F\} \cup \{x : x \in F, x$ succeeds l_F in left-to-right preorder *and* x succeeds r_F in right-to-left preorder$\}$ and edges $E(F_{l_F,r_F}) = \{(v,w) \in E(F) : v \in F_{l_F,r_F} \wedge w \in F_{l_F,r_F}\}$.

In contrast to the indexing scheme used by Δ^A [Demaine et al. 2009], our root encoding uniquely identifies a subforest: the node set defined by l_F and r_F is unique, and no deletion order needs to be considered.

Example 7.2. Figure 7 shows two subforests of a tree G (black nodes). Leftmost and rightmost root nodes are marked with arrows. The left subscript of a node is its left-to-

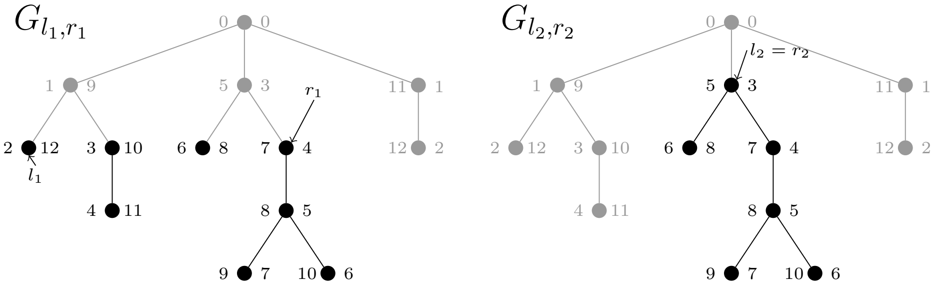


Fig. 7. Example subforests of tree G and their root encoding representation.

right and the right subscript its right-to-left preorder number. Consider the subforest G_{l_1,r_1}. The left-to-right preorder of l_1 is 2 and the right-to-left preorder of r_1 is 4. G_{l_1,r_1} is obtained from the tree G by removing nodes 0, 1, 5, 11, and 12 in left-to-right preorder, and all adjacent edges.

7.3. Algorithm for the General Single-Path Function

We present our algorithm for the *general single-path function* Δ^G, Algorithm 3. The input to the algorithm are two trees, F and G, a root-leaf path $\gamma \in \gamma^*(F)$ of tree F, and the distance matrix D. At the input, D must store the tree distances (without root node) between all relevant subtrees of F with respect to γ and all subtrees of G. The single-path function fills new values into the distance matrix. The new values are the tree distances (without root node) between all subtrees rooted at a node of path γ and all subtrees of G (cf. Section 3.3).

Algorithm 3 uses dynamic programming and stores intermediate results in three memoization tables, S , T , and Q. We focus on producing the subproblems in such an order that all distance results required in a later step have already been computed in a previous step. The details of storing and retrieving values from the memoization tables and the distance matrix are discussed in Section 8.

The subproblems are produced in nested loops, which are labelled with upper-case letters in Algorithm 3 and are nested as follows: $A(B(C(D)), B'(C'(D')))$. A relevant subproblem is of the form $(F_{l_v,r_v}, G_{l_w,r_w})$, i.e., it consists of two subforests. The subforests are defined by four nodes l_v, r_v, l_w, r_w. Each of the nested loops advances one of these nodes. In the innermost loop (D or D') the distance result for the subproblem defined by the current values of l_v, r_v, l_w, and r_w is computed.

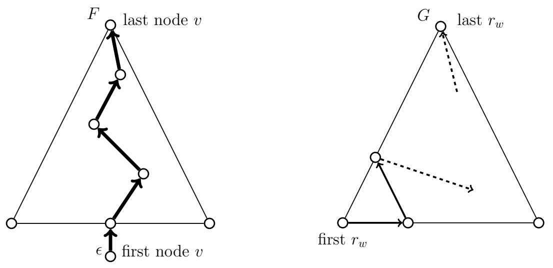
The subforests F_{l_v,r_v} of the left-hand tree are defined by the index variables of loops A and C (or A and C'), the subforests G_{l_w,r_w} are defined by the loops B and D (or B' and D'). Intuitively, loop C deals with subforests of F that have no nodes to the right of path γ, whereas loop C' deals with subforests that have also nodes to the right of γ.

Loop A is the outermost loop and iterates bottom-up over the nodes of path γ (cf. Figure 8(a)). The iteration starts with a dummy leaf node ϵ that is appended to the leaf node of path γ. The dummy node prevents the (non-dummy) leaf from being treated as a special case. The index variable v of loop A defines sets of nodes over which loops C and C' must iterate.

Loop C iterates over the nodes to the left of path γ, in particular, over the nodes $\mathit{left}(F_{p(v)}, v)$ for a given node v, where v is the index variable of loop A (cf. Figure 9(a)). Loop C defines the leftmost root node l_v for subforest F_{l_v,r_v}; the rightmost root node $r_v = v$ is defined by loop A.

Algorithm 3: $\Delta^G(F, G, \gamma, D)$

```
initialize memoization tables $S, T$, and $Q$   // see Section 8 for details
add dummy node $\epsilon$ to the leaf of path $\gamma$
foreach node $v \in F$ on the path $\gamma$ from $\epsilon$ to root do   // loop $A$
    $G' \leftarrow G$
    $l_v^{last} \leftarrow \{\}$
    $l'_v \leftarrow v$
    if $\gamma$ is right or inner then
        foreach node $r_w \in G$ in reverse right-to-left preorder do   // loop $B$
            if $\gamma$ is right then
                $l_v^{last} \leftarrow \{p(v)\}$
                if $r_w$ is rightmost child of its parent $p(r_w)$ then $G' \leftarrow G_{p(r_w)}$ else $G' \leftarrow \emptyset$
            $r_v \leftarrow v$
            foreach node $l_v \in left(F_{p(v)}, v) \cup l_v^{last}$ in reverse left-to-right preorder do   // loop $C$
                if $l_v = p(v)$ then $r_v \leftarrow p(v)$
                foreach node $l_w \in \{r_w\} \cup left(G', r_w)$ in reverse left-to-right preorder do   // loop $D$
                    compute $\delta_L(F_{l_v,r_v}, G_{l_w,r_w})$ and store in $S$   // see Section 8 for details
                $l'_v \leftarrow l_v$
            copy values from $S$ to $T$, $Q$, and $D$   // see Section 8 for details
    if $\gamma$ is left or inner then
        foreach node $l_w \in G$ in reverse left-to-right preorder do   // loop $B'$
            if $\gamma$ is left then
                if $l_w$ is leftmost child of its parent $p(l_w)$ then $G' \leftarrow G_{p(l_w)}$else $G' \leftarrow \emptyset$
            $l_v \leftarrow l'_v$
            foreach node $r_v \in right(F_{p(v)}, v) \cup \{p(v)\}$ in reverse right-to-left preorder do   // loop $C'$
                if $r_v = p(v)$ then $l_v \leftarrow p(v)$
                foreach node $r_w \in \{l_w\} \cup right(G', l_w)$ in reverse right-to-left preorder do   // loop $D'$
                    compute $\delta_R(F_{l_v,r_v}, G_{l_w,r_w})$ and store in $S$   // see Section 8 for details
            copy values from $S$ to $T$, $Q$, and $D$   // see Section 8 for details
return $D$
```

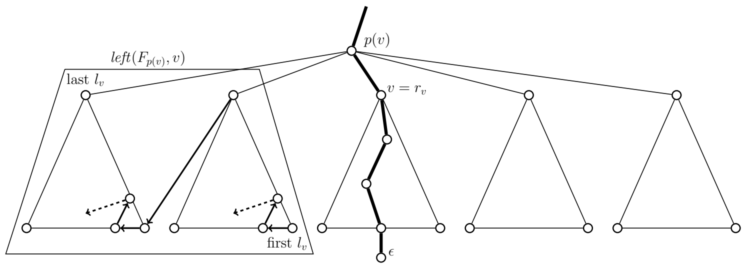


(a) loop A (b) loop B

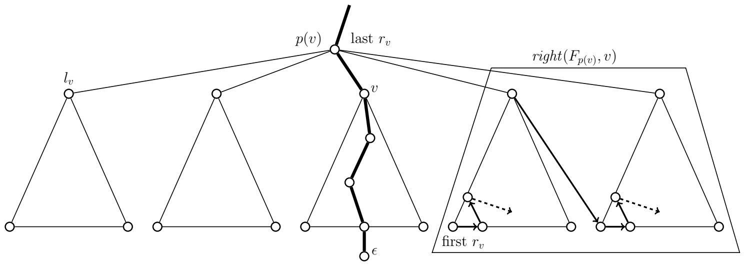
Fig. 8. The order of processing nodes in loops A and B.

Loop C' is symmetric to loop C and iterates over the nodes $right(F_{p(v)}, v) \cup \{p(v)\}$ (Figure 9(b)). Loop C' defines the rightmost root node r_v of subforest F_{l_v,r_v}; the leftmost root node l_v is the leftmost child of $p(v)$. In Figure 9 the first and the last values of node l_v in loop C resp. r_v in loop C' are marked.

If γ is a right path, the subtree of F rooted in $p(v)$ is treated in loop C, i.e., $l_v = p(v)$ (line 10) and $r_v = p(v)$ (line 14), and otherwise in loop C'.

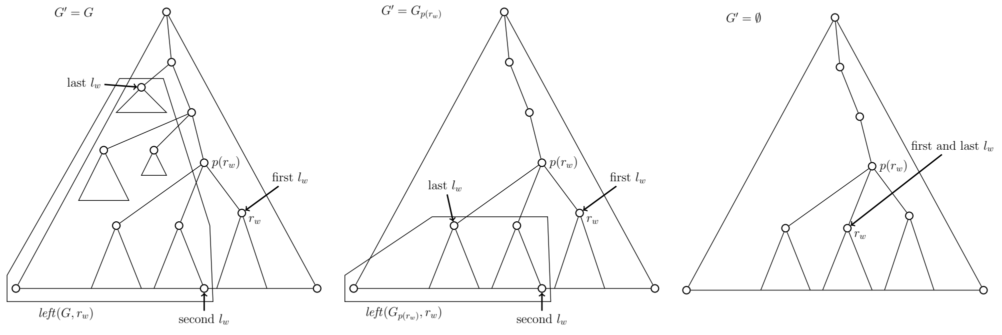


(a) Loop C.



(b) Loop C'.

Fig. 9. The order of processing nodes in loop C and C'.



(a) $G' = G$ (b) $G' = G_{p(r_w)}$ (c) $G' = \varnothing$

Fig. 10. The order of processing nodes in loop D

Loops B and D define subforests of the right-hand tree G. The subforests have the form G_{l_w,r_w}. Node r_w is the index variable of loop B which iterates over all nodes of tree G in reverse right-to-left preorder (cf. Figure 8(b)). Node l_w is the index variable of loop D and iterates over r_w and all nodes to the left of r_w in a specific subtree G' of G.

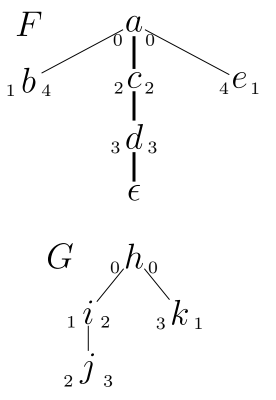
Subtree G' controls the nodes that must be considered in loop D. The definition of G' depends on the type of path γ (right or inner path) and the position of node r_w in G:

— *Inner path:* $G' = G$. This case is illustrated in Figure 10(a) and treated in line 4 in the algorithm. In the figure, the first, second, and last position of the index variable l_w are marked.
— *Right path:* If r_w is the rightmost child of its parent $p(r_w)$, then $G' = G_{p(r_w)}$ (cf. Figure 10(b)), otherwise $G' = \varnothing$ (cf. Figure 10(c)). See line 11 in the algorithm.

Left paths are treated in the loops B' and D', which are symmetric to B and D. Together, these loops produce a full decomposition, $\mathcal{A}(G)$, of tree G if γ is an inner path. If γ is a left (right) path, only the relevant subforests that result from recursively decomposing tree G with a left (right) path partitioning, $\mathcal{F}(G, \Gamma^L(G))$ ($\mathcal{F}(G, \Gamma^R(G))$), are produced.

This is the key feature of our algorithm: for each of the path types (left, right, inner), Δ^G produces exactly the same subforests that are produced by the respective specialized single-path function ($\Delta^L, \Delta^R, \Delta^A$), thus keeping the number of subproblems low. We prove this property in Section 7.4.

Example 7.3. We run the general single-path function, Δ^G (Algorithm 3), on trees F and G, and path γ (bold lines) in Figure 11. The right-hand part of the figure shows the progress of the algorithm: the subproblems that are computed and their order, the innermost loop (D or D') in which a particular subproblem is computed.



order	subproblem	loop
1	$(F_{d,d}, G_{k,k})$	D'
2	$(F_{d,d}, G_{j,j})$	D'
3	$(F_{d,d}, G_{j,k})$	D'
4	$(F_{d,d}, G_{i,i})$	D'
5	$(F_{d,d}, G_{i,k})$	D'
6	$(F_{d,d}, G_{h,h})$	D'
7	$(F_{c,c}, G_{k,k})$	D'
8	$(F_{c,c}, G_{j,j})$	D'
9	$(F_{c,c}, G_{j,k})$	D'
10	$(F_{c,c}, G_{i,i})$	D'
11	$(F_{c,c}, G_{i,k})$	D'
12	$(F_{c,c}, G_{h,h})$	D'
13	$(F_{b,c}, G_{j,j})$	D
14	$(F_{b,c}, G_{i,i})$	D
15	$(F_{b,c}, G_{k,k})$	D
16	$(F_{b,c}, G_{j,k})$	D
17	$(F_{b,c}, G_{i,k})$	D
18	$(F_{b,c}, G_{h,h})$	D
19	$(F_{b,e}, G_{k,k})$	D'
20	$(F_{a,a}, G_{k,k})$	D'
21	$(F_{b,e}, G_{j,j})$	D'
22	$(F_{b,e}, G_{j,k})$	D'
23	$(F_{a,a}, G_{j,j})$	D'
24	$(F_{a,a}, G_{j,k})$	D'
25	$(F_{b,e}, G_{i,i})$	D'
26	$(F_{b,e}, G_{i,k})$	D'
27	$(F_{a,a}, G_{i,i})$	D'
28	$(F_{a,a}, G_{i,k})$	D'
29	$(F_{b,e}, G_{h,h})$	D'
30	$(F_{a,a}, G_{h,h})$	D'

Fig. 11. Example run of Δ^G for trees F, G, and the path γ (bold line).

7.4. Correctness of the General Single-Path Function

We analyze our algorithm for the general single-path function, Δ^G (Algorithm 3), and we prove that it produces the minimal number of subproblems among the single-path functions Δ^L, Δ^R, and Δ^A. The proofs are based on the observation that the subproblems produced by a single-path function are the cartesian product of the subforests produced for the left-hand tree F and the right-hand tree G (cf. Section 5.2). We proceed in three steps.

(1) We treat the relevant subforests of F and G separately and show that Algorithm 3 produces exactly the same relevant subforests as the single-path functions Δ^L, Δ^R, and Δ^A for the respective path type (Lemmas 7.4-7.6).
(2) We show that Algorithm 3 computes the cartesian product of the relevant subforests in F and G for a given path (Lemma 7.7).

(3) We conclude that Algorithm 3 produces the same relevant subproblems as the best single-path function for the given path, and hence the number of subproblems is minimal among all single-path functions (Theorem 7.8).

LEMMA 7.4. *The set of relevant subforests of tree F produced by Algorithm 3 for a given path γ is $\mathcal{F}(F,\gamma)$.*

PROOF. First, we show by induction that each relevant subforest of tree F produced by Algorithm 3 is an element of $\mathcal{F}(F,\gamma)$. Second, we count the number of relevant subforests and show that the count is equal to $|\mathcal{F}(F,\gamma)|$.

Let $v_0, v_1, \dots, v_n$ be the nodes in F on path γ such that $v_0 = r(F)$, $v_i = p(v_{i+1})$, $0 \le i < n$, and $v_n = \epsilon$, where ϵ is a dummy node. Algorithm 3 produces the following three types of relevant subforests:

(a) F_{v_i}, $i < n$, which are the subtrees of F rooted in the nodes on path γ (line 16 if γ is right, line 27 otherwise);
(b) F_{l_v,r_v}, where $r_v \in \mathit{right}(F_{v_i}, v_{i+1})$ and l_v is the leftmost child of v_i (line 27);
(c) F_{l_v,r_v}, where $r_v = v_{i+1}$ and $l_v \in \mathit{left}(F_{v_i}, v_{i+1})$ (line 16).

By Definition 3.3, F_{v_0} is in $\mathcal{F}(F,\gamma)$. Further, if F_{v_i} ($i < n-1$) is in $\mathcal{F}(F,\gamma)$, then also the respective forests in (b) and (c), and $F_{v_{i+1}}$ are in $\mathcal{F}(F,\gamma)$: The forests in (b) are obtained from F_{v_i} by recursively deleting the rightmost root node r_v until r_v is on path γ; the resulting forest (l_v is leftmost child of v_i, $r_v = v_{i+1}$) is the largest subforest in (c); the other subforests in (c) are obtained by recursively deleting the leftmost root node l_v until l_v is on the path, i.e., $l_v = v_{i+1}$; the resulting subforest is $F_{v_{i+1}}$.

We next count all relevant subforests of F produced by Algorithm 3 and show that the count is $|F|$ ($|F| = |\mathcal{F}(F,\gamma)|$, cf. Lemma 5.2). Each subtree F_{v_i}, $0 \le i < n$, counts as one relevant subforest.

$$1 + |\mathit{left}(F_{v_0}, v_1)| + |\mathit{right}(F_{v_0}, v_1)| + 1 + \dots + |\mathit{left}(F_{v_{n-2}}, v_{n-1})| + |\mathit{right}(F_{v_{n-2}}, v_{n-1})| + 1$$
$$= n + \sum_{i=0}^{n-2} |\mathit{left}(F_{v_i}, v_{i+1})| + \sum_{i=0}^{n-2} |\mathit{right}(F_{v_i}, v_{i+1})|$$

$\sum_{i=0}^{n-2} |\mathit{left}(F_{v_i}, v_{i+1})|$ is the number of nodes to the left of path γ, thus

$$n + \sum_{i=0}^{n-2} |\mathit{left}(F_{v_i}, v_{i+1})| + \sum_{i=0}^{n-2} |\mathit{right}(F_{v_i}, v_{i+1})|$$
$$= n + |F| - \sum_{i=0}^{n-2} |\mathit{right}(F_{v_i}, v_{i+1})| - n + \sum_{i=0}^{n-2} |\mathit{right}(F_{v_i}, v_{i+1})| = |F| \quad \square$$

LEMMA 7.5. *The set of relevant subforests of tree G produced by Algorithm 3 is $\mathcal{A}(G)$ if γ is an inner path.*

PROOF. We first show that each of the relevant subforests produced by Algorithm 3 is in $\mathcal{A}(G)$. Second, we count the number of all relevant subforests in G and show that the number is equal to $|\mathcal{A}(G)|$. Algorithm 3 produces the following two types of forests:

(a) G_{l_w,r_w}, where $r_w \in G$ and $l_w \in \mathit{left}(G, r_w) \cup \{r_w\}$ (loop D);
(b) G_{l_w,r_w}, where $l_w \in G$ and $r_w \in \mathit{right}(G, l_w) \cup \{l_w\}$ (loop D').

By Definition 3.1, G is in $\mathcal{A}(G)$. Further, all forests in (a) and (b) are in $\mathcal{A}(G)$: The forests in (a) are obtained from G by a sequence of right deletions (recursively deleting the rightmost root node) followed by a sequence of left deletions; the forests in (b) are obtained from G by a sequence of left deletions followed right deletions.

We now count the number of subforests of tree G produced by Algorithm 3 in loop D. The proof for loop D' is similar. Let $w_1, \dots, w_{|G|}$ be the nodes of G in right-to-left preorder. Then, the number of relevant subforests is

$$1 + |\mathit{left}(G, w_{|G|})| + 1 + |\mathit{left}(G, w_{|G|-1})| + \cdots + 1 + |\mathit{left}(G, w_2)| + 1 + |\mathit{left}(G, w_1)|.$$

Each subtree of G counts as one relevant subforest; $\mathit{left}(G, w_i)$, $1 \le i \le |G|$, is the set of nodes to the left of node w_i, i.e., all nodes in G minus the nodes in G_{w_i} minus all nodes preceding w_i in right-to-left preorder ($i-1$ nodes), thus $|\mathit{left}(G, w_i)| = |G| - |G_{w_i}| - (i-1)$.

Then, the number of subforests is computed as follows.

$$(2 - |G_{w_{|G|}}|) + (3 - |G_{w_{|G|-1}}|) + \cdots + (|G| - |G_{w_2}|) + (|G| + 1 - |G_{w_1}|)$$

$$= 2 + 3 + \cdots + |G| + |G| + 1 - \sum_{i=1}^{|G|} |G_{w_i}| = \frac{(|G|+3)|G|}{2} - \sum_{i=1}^{|G|} |G_{w_i}| = |\mathcal{A}(G)| \quad \text{by Lemma 5.1} \quad \square$$

LEMMA 7.6. *The set of relevant subforests of tree G produced by Algorithm 3 for a given path γ is $\mathcal{F}(G, \Gamma^L(G))$ if γ is the left path and $\mathcal{F}(G, \Gamma^R(G))$ if γ is the right path.*

PROOF. *Left path:* We first show that each subforest of tree G produced by Algorithm 3 is in $\mathcal{F}(G, \Gamma^L(G))$; second, we show that the number of such subforests is $|\mathcal{F}(G, \Gamma^L(G))|$.

Each node of G lies on some (only one) $\gamma \in \Gamma^L(G)$, and $\Gamma^L(G)$ covers the entire tree G. Let $r(\gamma)$ be the root node of path γ, i.e., a node $a \in \gamma$ such that $p(a) \notin \gamma$. Then, the set of the root nodes of all paths in $\Gamma^L(G)$ is defined as $RN = \{w \in G : \exists_{\gamma \in \Gamma^L(G)}(w = r(\gamma))\}$. Due to the definition of relevant subtrees resulting from decomposing a tree with a path partitioning (cf. Equation 2), $RN = \{w \in G : G_w \in \mathcal{T}(G, \Gamma^L(G))\}$. Due to Equations 1 and 2, and the definition of a path partitioning (cf. Section 3.1) the following holds:

$$\mathcal{F}(G, \Gamma^L(G)) = \bigcup_{w \in RN} \mathcal{F}(G_w, \gamma^L(G_w))$$

We show that for any subtree G_w, $w \in RN$, and the nodes on $\gamma^L(G_w)$ the following holds: the subforests produced by Algorithm 3 are elements of $\mathcal{F}(G_w, \gamma^L(G_w))$, thus they are elements of $\mathcal{F}(G, \Gamma^L(G))$. Let $w_0, w_1, \dots, w_n$ be the nodes on path $\gamma^L(G_w)$, where $w_0 = w$, $w_i = p(w_{i+1})$, $0 \le i < n$, w_n is the leaf node of $\gamma^L(G_w)$. Algorithm 3 produces the following two types of relevant subforests (line 27):

(a) G_{w_i}, $0 \le i \le n$, i.e., the subtrees of G_w rooted on the nodes in $\gamma^L(G_w)$;
(b) G_{w_{i+1}, r_w}, where $r_w \in \mathit{right}(G_{w_i}, w_{i+1})$, $0 \le i < n$ (cf. line 22).

By Definition 3.3, G_{w_0} is in $\mathcal{F}(G_w, \gamma^L(G_w))$. Further, if G_{w_i} ($i < n$) is in $\mathcal{F}(G_w, \gamma^L(G_w))$, then also the respective forests in (b) and $G_{w_{i+1}}$ are in $\mathcal{F}(G_w, \gamma^L(G_w))$: The forests in (b) are obtained from G_{w_i} by recursively deleting the rightmost root node r_w until r_w is on path $\gamma^L(G_w)$; the resulting subforest is $G_{w_{i+1}}$.

We next show that the number of relevant subforests produced by Algorithm 3 is equal to the number of subforests in $\mathcal{F}(G, \Gamma^L(G))$. The number of relevant subforests is computed as follows (each subtree of G is a relevant subforest):

$$\sum_{w \in RN} 1 + \sum_{w' \notin RN} (1 + |\mathit{right}(G_{p(w')}, w')|) \qquad \text{(each } w' \text{ lies on some path } \gamma^L(G_w))$$

$$= \sum_{w \in RN} (1 + \sum_{w' \in \gamma^L(G_w) \wedge w' \notin RN} (1 + |\mathit{right}(G_{p(w')}, w')|)) \qquad \text{(by Definition 3.3 for a left path)}$$

$$= \sum_{w \in RN} |\mathcal{F}(G_w, \gamma^L(G_w))| = |\mathcal{F}(G, \Gamma^L(G))|$$

Right path: analogous reasoning. □

LEMMA 7.7. *The subproblems produced by Algorithm 3 for a given path type (left, right, inner) are equal to the subproblems produced by the corresponding single-path function Δ^L, Δ^R, and Δ^A.*

$$\textit{relevantSubproblems}(F,G,\gamma)=\begin{cases}\mathcal{F}(F,\gamma)\times\mathcal{F}(G,\Gamma^L(G)) & \textit{if } \gamma \textit{ is left path}\\ \mathcal{F}(F,\gamma)\times\mathcal{F}(G,\Gamma^R(G)) & \textit{if } \gamma \textit{ is right path}\\ \mathcal{F}(F,\gamma)\times\mathcal{A}(G) & \textit{otherwise}\end{cases}$$

PROOF. By Lemmas 7.4-7.6 for a given path type, Algorithm 3 produces the same subforests of the trees F and G as the respective single-path function. We need to show that the subproblems are computed as the cartesian product of the relevant subforests in F and G. Right paths: loops A and C produce all relevant subforests of F. Loops B and D produce all relevant subforests of G. In line 16 loops C and D pair all relevant subforests of F and G into relevant subproblems. The reasoning for left and inner paths is similar. □

THEOREM 7.8. *For a given path γ, Algorithm 3 produces the minimal number of subproblems among Δ^L, Δ^R, and Δ^A.*

PROOF. Follows from Lemma 7.7 and the following observations.

— If γ is a left path, it can be processed by either Δ^L or Δ^A. Δ^R cannot process left paths. The cost of Δ^L is smaller than the cost of Δ^A because $\mathcal{F}(F,\gamma)\times\mathcal{F}(G,\Gamma^L(G))\subseteq\mathcal{F}(F,\gamma)\times\mathcal{A}(G)$.
— If γ is a right path, it can be processed by either Δ^R or Δ^A. The cost of Δ^R is smaller than the cost of Δ^A because $\mathcal{F}(F,\gamma)\times\mathcal{F}(G,\Gamma^R(G))\subseteq\mathcal{F}(F,\gamma)\times\mathcal{A}(G)$.
— If γ is an inner path, it can only be processed by Δ^A. Thus, the minimal cost is $|\mathcal{F}(F,\gamma)|\times|\mathcal{A}(G)|$. □

8. EFFICIENT INDEX FOR SUBPROBLEMS

We present a new indexing technique that allows us to store and retrieve distance results for subproblems in constant time. Subproblems are pairs of subforests of the form $(F_{l_F,r_F}, G_{l_G,r_G})$, where l_F, r_F, l_G, r_G are leftmost resp. rightmost root nodes in our root encoding (cf. Section 7.2). The result of a subproblem is the distance between F_{l_F,r_F} and G_{l_G,r_G}. Results for subproblems are stored in memoization tables. We map leftmost and rightmost root nodes to numeric IDs, which are used as row and column numbers for the memoization tables.

An indexing technique for subproblems must perform the following operations in constant time:

— *Store distance results*. The result for a given subproblem must be stored. The number of different subproblems that can appear in a distance computation is $O(n^4)$ (n is the number of tree nodes). The challenge is to limit the space complexity to $O(n^2)$ by reusing storage cells for multiple subproblems.
— *Retrieve distance results*. When the distance for a new subproblem is computed, the results for smaller subproblems must be accessed. We must be able to compute the IDs of the smaller subproblems from the ID of the new subproblem in constant time.
— *Find the next subproblem.* The subproblems must be processed in a specific order to guarantee correctness and quadratic space complexity. We must compute the ID of the next subproblem to be processed from the ID of the current subproblem in

constant time. This is challenging since the IDs depend on the tree structure and are not consecutive numbers.

We map subproblems to numeric IDs, present solutions for storing and retrieving intermediate results, and compute the ID of the next subproblem to be processed.

8.1. Numeric IDs for Subproblems

We introduce a novel numbering of nodes to map subproblems, i.e., pairs of subforests, to numeric identifiers. Each node in a tree is assigned two numeric IDs: its left-to-right and its right-to-left preorder position. Using the root encoding (cf. Section 7.2), we represent a subforest by a pair of integers as follows. Forest F_{l_F,r_F}, is represented as $F_{o^{\rightarrow}(l_F),o^{\leftarrow}(r_F)}$, where

— $o^{\rightarrow}(l_F)$ is the left-to-right preorder position of node l_F in F,
— $o^{\leftarrow}(r_F)$ is the right-to-left preorder position of node r_F in F.

The numbering starts with 0, i.e., $o^{\rightarrow}(r(F)) = o^{\leftarrow}(r(F)) = 0$. A subproblem is a pair of subforests and is identified by four integers.

Example 8.1. Consider the forests in Figure 7. The numbers to the left and to the right of a node are its left-to-right and right-to-left preorder positions, respectively. $o^{\rightarrow}(l_1) = 2$ and $o^{\leftarrow}(r_1) = 4$, thus forest G_{l_1,r_1} is represented as $G_{2,4}$. Forest G_{l_2,r_2} is a tree since $l_2 = r_2$; note however, $o^{\rightarrow}(l_2) \neq o^{\leftarrow}(r_2)$, and tree G_{l_2,r_2} is represented as $G_{5,3}$.

8.2. Storing Intermediate Results

The dynamic-programming implementation of the general single-path function, Δ^G, must store intermediate distance results for subproblems efficiently. We expand lines 16, 18, 27, and 28 of Algorithm 3 to provide details of how the intermediate results are stored in memoization tables. The new lines are shown in Algorithm 4.

Algorithm 4: Storing intermediate results in Δ^G (extends Algorithm 3)

```
     ...
16   $S[l_v, l_w] \leftarrow \delta_L(F_{l_v,r_v}, G_{l_w,r_w})$
     ...
18a  if $r_w$ is rightmost child of $p(r_w)$ then
b        if $v$ is rightmost child of $p(v)$ then
c            $D[p(v), p(r_w)] \leftarrow S$[leftmost child of $p(v)$, leftmost child of $p(r_w)$]
d        foreach node $l_v \in left(F_{p(v)}, v) \cup l_v^{last}$ in reverse left-to-right preorder do
e            $Q[l_v] \leftarrow S[l_v,$ leftmost child of $p(r_w)]$
f    foreach node $l_w \in \{r_w\} \cup left(G', r_w)$ in reverse left-to-right preorder do
g        $T[l_w, r_w] \leftarrow S[l'_v, l_w]$
     ...
27   $S[r_v, r_w] \leftarrow \delta_R(F_{l_v,r_v}, G_{l_w,r_w})$
28a  if $l_w$ is leftmost child of $p(l_w)$ then
b        $D[p(v), p(l_w)] \leftarrow S$[rightmost child of $p(v)$, rightmost child of $p(l_w)$]
c        foreach node $r_v \in right(F_{p(v)}, v) \cup \{p(v)\}$ do
d            $Q[r_v] \leftarrow S[r_v,$ rightmost child of $p(l_w)]$
e    foreach node $r_w \in \{l_w\} \cup right(G', l_w)$ in reverse right-to-left preorder do
f        $T[l_w, r_w] \leftarrow S[p(v), r_w]$
     ...
```

The intermediate results are stored in four memoization tables: D, S, T, and Q of the sizes $|F||G|$, $|F||G|$, $|G|^2$, and $|F|$, respectively. Distance matrix D is an input of Δ^G

and is filled with new distance values, which are returned as the result of the single-path function. New distance results are computed only in the innermost loops and are stored in S (lines 16 and 27). Results that are required for later distance computations are copied to the tables D, T, and Q (lines 18 and 28) at the end of loop B (resp. B'). Copying the values does not change the runtime because only a small subset of values of S is copied. In particular, one value is copied to D (line 18c, 28b), one column of S is copied to Q (lines 18d-e, 28c-d), and one row of S is copied to T (lines 18f-g, 28e-f).

A subproblem is identified by four integers, but our two-dimensional memoization tables are indexed with only two values (row and column number). The two missing values are implicit and are given by the context in Algorithm 4. The implicit values are the same for all subproblems stored at the same time in a specific memoization table. For example, if a distance $\delta_L(F_{l_F,r_F}, G_{l_G,r_G})$ must be stored in a memoization table, the following node IDs are used: $D[o^{\rightarrow}(p(r_F)), o^{\rightarrow}(p(r_G))]$, $S[o^{\rightarrow}(l_F), o^{\rightarrow}(l_G)]$, $T[o^{\rightarrow}(l_G), o^{\leftarrow}(r_G)]$, $Q[o^{\rightarrow}(l_F)]$; the distance $\delta_R(F_{l_F,r_F}, G_{l_G,r_G})$ is stored as follows: $D[o^{\rightarrow}(p(l_F)), o^{\rightarrow}(p(l_G))]$, $S[o^{\leftarrow}(r_F), o^{\leftarrow}(r_G)]$, $T[o^{\rightarrow}(l_G), o^{\leftarrow}(r_G)]$, $Q[o^{\leftarrow}(r_F)]$.

Distance matrix D stores the distance values which are needed across different calls of the single-path function. The subproblems in D are pairs of subtrees without their root nodes (cf. Section 3.3).

We discuss the use of the other memoization tables in loops A, B, C, and D of the algorithm; the discussion for loops B', C', and D' is similar. Table S stores the distances of all the subproblems computed in the two innermost loops C and D for specific nodes in the outer loops. The values in S are overwritten in each iteration of loop B. Tables T and Q store distances which are needed in different calls of loop A and loop B, respectively. T stores the distances between a specific subforest in F and each relevant subforest of G. The subforest in F is either a subtree rooted on the path F_v (line 28) or a subforest $F_{l'_v,v}$, where l'_v is the leftmost child of $p(v)$ (line 18). Q stores the distances between each subforest of F defined in loop C and a specific subforest in G of the form $G_{p(r_w)} - p(r_w)$, i.e., a subtree without root node. After loop A, the distance between the two input trees, F and G, is stored in $T[r(G), r(G)]$, where $r(G)$ is the root of G.

Computing node IDs and verifying conditions must be done in constant time in Algorithm 4. In particular, the following operations are performed. *(1) Find the parent of node v:* The result of the function $p(v)$ is precomputed for all nodes in linear time and space in the tree size. *(2) Find the rightmost (leftmost) child of node v:* The IDs of the leftmost and rightmost children of a node v are $o^{\rightarrow}(v) + 1$ and $o^{\leftarrow}(v) + 1$, respectively. *(3) Check equality of nodes:* Two nodes are equal iff their respective IDs are equal. For example, we verify the condition "r_w *is rightmost child of* $p(r_w)$" by evaluating $o^{\leftarrow}(r_w) = o^{\leftarrow}(p(r_w)) + 1$.

Special case (not shown in Algorithm 4). When v is the only child of $p(v)$, we modify lines 18b and 28b to $D[p(p(v)), p(r_w)] \leftarrow S[p(v), \text{leftmost child of } p(r_w)]$ and $D[p(p(v)), p(l_w)] \leftarrow S[p(v), \text{rightmost child of } p(l_w)]$, respectively.

Our use of the memoization tables is inspired by the single-path function Δ^A, and we keep the naming convention of Demaine et al. [2009]. Note, however, that we store the subproblems differently using the indexing scheme presented in this section, which requires only a linear size index structure to access all required values in constant time.

8.3. Retrieving Intermediate Results

The distances of the relevant subproblems are computed in loops D and D' in Algorithm 3 (lines 16 and 27). We expand the recursive formula of Figure 3 into δ_L for leftmost root nodes and δ_R for rightmost root nodes. δ_L and δ_R are shown in Figure 12. We do not use the distance formulas in a recursive fashion. At each step we ensure that

$$\delta_L(F_{l_v,r_v}, G_{l_w,r_w}) = \min \begin{cases} \delta(F_{l_v,r_v} - l_v, G_{l_w,r_w}) + c_d(l_v) \\ \delta(F_{l_v,r_v}, G_{l_w,r_w} - l_w) + c_i(l_w) \\ \delta(F_{l_v} - l_v, G_{l_w} - l_w) \\ \qquad + \delta(F_{l_v,r_v} - F_{l_v}, G_{l_w,r_w} - G_{l_w}) + c_r(l_v, l_w) \end{cases}$$

$$\delta_R(F_{l_v,r_v}, G_{l_w,r_w}) = \min \begin{cases} \delta(F_{l_v,r_v} - r_v, G_{l_w,r_w}) + c_d(r_v) \\ \delta(F_{l_v,r_v}, G_{l_w,r_w} - r_w) + c_i(r_w) \\ \delta(F_{r_v} - r_v, G_{r_w} - r_w) \\ \qquad + \delta(F_{l_v,r_v} - F_{r_v}, G_{l_w,r_w} - G_{r_w}) + c_r(r_v, r_w) \end{cases}$$

Fig. 12. δ_L and δ_R used to compute the distances of the relevant subproblems in Δ^G.

all distances needed to compute the minimum are available for constant-time retrieval in a memoization table.

Figure 13 shows how to retrieve the required values for δ_L (the procedure for δ_R is similar). The correctness of these rules follows from the analysis in [Demaine et al. 2009]. All node IDs in the figure are variables defined in loop D (where δ_L is computed), except a, b, x, and y. These nodes are unknown and must be computed; they are defined as follows: $a, b \in F$ and $x, y \in G$ such that $F_{a,r_v} = F_{l_v,r_v} - l_v$, $F_{b,r_v} = F_{l_v,r_v} - F_{l_v}$, $G_{x,r_w} = G_{l_w,r_w} - l_w$, $G_{y,r_w} = G_{l_w,r_w} - G_{l_w}$.

Accessing the memoization tables with known IDs is straightforward, e.g., $T[o^{\rightarrow}(l_w), o^{\leftarrow}(r_w)]$, $Q[o^{\rightarrow}(l_w)]$, or $D[o^{\rightarrow}(l_v), o^{\rightarrow}(l_w)]$. The challenge is to compute in constant time (a) the sums in Figure 13, (b) the IDs of the unknown nodes a, b, x, y.

(a) Computing the sums in constant time. When one forest of the required subproblem is empty, then the corresponding distance is equal to the cost of inserting (deleting) the other subforest. This cost, $\sum_{w \in G_{l_w,r_w}} c_i(w)$ ($\sum_{v \in F_{l_v,r_v}} c_d(v)$), is computed incrementally while building the subforests in Algorithm 3. Each time a node is added to a forest we update the cost. The insertion (deletion) cost of each subtree in G (F), $\sum_{w \in G_{l_w}} c_i(w)$ ($\sum_{v \in F_{l_v}} c_d(v)$), is precomputed in linear time and space. Thus, also sum $\sum_{w \in G_{l_w,r_w} - G_{l_w}} c_i(w)$ ($\sum_{v \in F_{l_v,r_v} - F_{l_v}} c_d(v)$) is calculated in constant time since $\sum_{w \in G_{l_w,r_w} - G_{l_w}} c_i(w) = \sum_{w \in G_{l_w,r_w}} c_i(w) - \sum_{w \in G_{l_w}} c_i(w)$ ($\sum_{v \in F_{l_v,r_v} - F_{l_v}} c_d(v) = \sum_{v \in F_{l_v,r_v}} c_d(v) - \sum_{v \in F_{l_v}} c_d(v)$).

(b) Computing the IDs of nodes a, b, x, and y. Nodes a and x (b and y) are the leftmost root nodes of subforests that result from removing the leftmost root node (the leftmost subtree) of another subforest. The IDs of these nodes depend on the structure of the input trees, and in some cases there is no direct way to compute the IDs in constant time.

A straightforward approach precomputes the node IDs. Since the IDs of nodes a, b, x, and y differ for each subforest, the precomputation results in substantial space overhead. To materialise the precomputed IDs, eight matrices of size $|F||G|$ are needed: four to store the IDs of the leftmost root nodes a, b, x, y; and another four to store the IDs of the rightmost root nodes resulting from the rightmost root node/rightmost subtree removal in $\delta_R(F_{l_v,r_v}, G_{l_w,r_w})$.

We propose a constant time and space solution which is based on the observation that the nodes a, b (x, y) have been processed before l_v (l_w) in loop C (loop D), i.e., we leverage the order in which the nodes are processed in our algorithm.

Computing the IDs of nodes a and b. The nodes $l_v \in \mathit{left}(F_{p(v)}, v) \cup l_v^{last}$ processed in loop C have consecutive left-to-right preorder numbers as shown in Figure 14. The

$$\delta(F_{l_v,r_v} - l_v, G_{l_w,r_w}) = \begin{cases} \sum_{w \in G_{l_w,r_w}} c_i(w) & \text{if } F_{l_v,r_v} - l_v = \emptyset \\ T[l_w, r_w] & \text{if } F_{l_v,r_v} - l_v = F_v \\ S[a, l_w] & \text{otherwise} \end{cases}$$

$$\delta(F_{l_v,r_v}, G_{l_w,r_w} - l_w) = \begin{cases} \sum_{v \in F_{l_v,r_v}} c_d(v) & \text{if } G_{l_w,r_w} - l_w = \emptyset \\ Q[l_v] & \text{if } G_{l_w,r_w} \text{ is a tree} \\ S[l_v, x] & \text{otherwise} \end{cases}$$

$$\delta(F_{l_v} - l_v, G_{l_w} - l_w) = \begin{cases} \left(\sum_{w \in G_{l_w}} c_i(w)\right) - c_i(l_w) & \text{if } F_{l_v,r_v} - l_v = \emptyset \\ \left(\sum_{v \in F_{l_v}} c_d(v)\right) - c_d(l_v) & \text{if } G_{l_w,r_w} - l_w = \emptyset \\ D[l_v, l_w] & \text{otherwise} \end{cases}$$

$$\delta(F_{l_v,r_v} - F_{l_v}, G_{l_w,r_w} - G_{l_w}) = \begin{cases} 0 & \text{if } F_{l_v,r_v} - F_{l_v} = \emptyset \wedge G_{l_w,r_w} - G_{l_w} = \emptyset \\ \sum_{w \in G_{l_w,r_w} - G_{l_w}} c_i(w) & \text{if } F_{l_v,r_v} - F_{l_v} = \emptyset \\ \sum_{v \in F_{l_v,r_v} - F_{l_v}} c_d(v) & \text{if } G_{l_w,r_w} - G_{l_w} = \emptyset \\ T[y, r_w] & \text{if } F_{l_v,r_v} - F_{l_v} = F_v \\ S[b, y] & \text{otherwise} \end{cases}$$

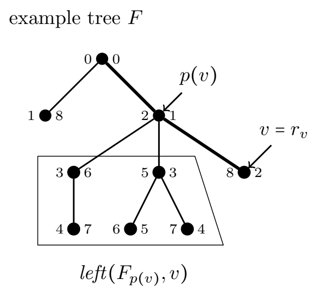
Fig. 13. Rules for obtaining the required distances when computing $\delta_L(F_{l_v,r_v}, G_{l_w,r_w})$ using the formula in Figure 12. Nodes $a, b \in F$ and $x, y \in G$ are such that $F_{a,r_v} = F_{l_v,r_v} - l_v$, $F_{b,r_v} = F_{l_v,r_v} - F_{l_v}$, $G_{x,r_w} = G_{l_w,r_w} - l_w$, $G_{y,r_w} = G_{l_w,r_w} - G_{l_w}$.

leftmost root node a such that $F_{a,r_v} = F_{l_v,r_v} - l_v$ is the node processed just before l_v, and $o^{\rightarrow}(a) = o^{\rightarrow}(l_v) + 1$. Computing the leftmost root node b such that $F_{b,r_v} = F_{l_v,r_v} - F_{l_v}$ is similar. Instead of l_v we remove the entire subtree F_{l_v}. The IDs of all nodes in F_{l_v} are consecutive numbers, and $o^{\rightarrow}(b) = o^{\rightarrow}(l_v) + |F_{l_v}|$.

Example 8.2. In Figure 14 we show the nodes processed in a single run of loop C on an example tree F. The path is marked with a thick line. The left-to-right preorder IDs of the nodes processed in loop C for the given nodes v and $p(v)$, $l_v \in \mathit{left}(F_{p(v)}, v)$, are consecutive numbers $7, 6, 5, 4, 3$. The right side of Figure 14 shows how the IDs of nodes a and b are computed for a given ID of node l_v. For example, consider the forest $F_{5,2}$, where $o^{\rightarrow}(l_v) = 5$, $o^{\leftarrow}(r_v) = 2$. The ID of the node a (b), which is the new leftmost root node after removing the leftmost root node (leftmost subtree) from $F_{5,2}$, is computed by incrementing $o^{\rightarrow}(l_v)$ by 1 (by the size of the subtree rooted in l_v, $|F_{l_v}| = 3$) and results in the forest $F_{6,2}$ ($F_{8,2}$).

Computing the IDs of nodes x and y. The IDs of the nodes processed in loop D, $l_w \in \{r_w\} \cup \mathit{left}(G, r_w)$, are not necessarily consecutive numbers and there may be *gaps* in the numbering. This case is shown on Figure 15: there are two gaps in the left-to-right preorder IDs of the nodes $\{r_w\} \cup \mathit{left}(G, r_w)$: 3-7 and 8-10.

Our solution is to remember the last gap while loop D proceeds. Specifically, we maintain a variable gn, which stores the ID of the node preceding the last gap in the node IDs processed by loop D so far. Let $\{g_1, g_2, \ldots, g_k\}$ be the nodes processed by loop D



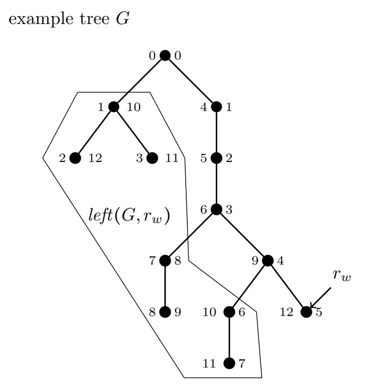
$o^{\rightarrow}(l_v)$	$o^{\rightarrow}(a)$	$o^{\rightarrow}(b)$
7	$7+1=8$	$7+1=8$
6	$6+1=8$	$6+1=8$
5	$5+1=6$	$5+3=8$
4	$4+1=5$	$4+1=5$
3	$3+1=4$	$3+2=5$

Fig. 14. Computing nodes a and b such that $F_{a,r_v} = F_{l_v,r_v} - l_v$ and $F_{b,r_v} = F_{l_v,r_v} - F_{l_v}$, respectively.

so far, such that $o^{\rightarrow}(g_{i-1}) < o^{\rightarrow}(g_i), 1 < i \leq k$. Then, $gn = o^{\rightarrow}(g_i)$, where $g_i \in \{g_1, g_2, \dots, g_k\}$ is the node with the minimal left-to-right preorder ID such that $o^{\rightarrow}(g_i) - o^{\rightarrow}(g_{i-1}) > 1$.

If there is a gap between the IDs of nodes l_w and x/y, then $o^{\rightarrow}(x) = o^{\rightarrow}(y) = gn$. Otherwise, the IDs between l_w and x/y are consecutive numbers and we compute the new IDs like for the nodes a/b, i.e., $o^{\rightarrow}(x) = o^{\rightarrow}(l_w)+1$ and $o^{\rightarrow}(y) = o^{\rightarrow}(l_w)+|G_{l_w}|$. The gap exists if the nodes x/y have not been processed in loop B before the current node r_w, i.e., $o^{\leftarrow}(o^{\rightarrow}(l_w)+1) < o^{\leftarrow}(r_w)$ and $o^{\leftarrow}(o^{\rightarrow}(l_w)+|G_{l_w}|) < o^{\leftarrow}(r_w)$ (where $o^{\leftarrow}(o^{\rightarrow}(v))$ converts left-to-right to right-to-left preorder ID of node v).

Example 8.3. Figure 15 shows the nodes processed in a single run of loop D for example tree G. For the marked node r_w, loop D processes the nodes $\{r_w\} \cup left(G, r_w)$ in reverse left-to-right preorder: $12, 11, 10, 8, 7, 3, 2, 1$. For each node l_w we show the value of gn and the computation of the IDs of x and y. gn is initialized with $o^{\rightarrow}(r_w)$. Consider, for example, forest $G_{7,5}$: the ID of node x is computed by increasing the leftmost root node ID by 1; the ID of node y is computed by changing the leftmost root node ID to the value of gn because $o^{\leftarrow}(o^{\rightarrow}(l_w) + |F_{l_w}|) = 4 < o^{\leftarrow}(r_w) = 5$, i.e., there is a gap between nodes 7 and 10.



$o^{\rightarrow}(l_w)$	gn	$o^{\rightarrow}(x)$	$o^{\rightarrow}(y)$
12	12	—	—
11	12	$11+1=12$	$11+1=12$
10	12	$10+1=11$	$10+2=12$
8	10	$gn=10$	$gn=10$
7	10	$7+1=8$	$gn=10$
3	7	$gn=7$	$gn=7$
2	7	$2+1=3$	$2+1=3$
1	7	$1+1=2$	$gn=7$

Fig. 15. Computing nodes x and y such that $G_{x,r_w} = G_{l_w,r_w} - l_w$ and $G_{y,r_w} = G_{l_w,r_w} - G_{l_w}$, respectively.

8.4. Finding the Next Subproblem

In each iteration of a loop in Algorithm 3 we must find the IDs of the next nodes to be processed. The implementation of loops A and B is straightforward. In loop C

the IDs of the nodes $l_v \in left(F_{p(v)}, v) \cup l_v^{last}$ are consecutive numbers. Given $o^{\rightarrow}(l_v)$, the ID of the next node to be processed is $o^{\rightarrow}(l_v) - 1$. Differently, the IDs of the nodes $l_w \in \{r_w\} \cup left(G, r_w)$ in loop D are not necessarily consecutive, and there may be gaps in the numbering. An example is shown on Figure 15: the IDs of the nodes processed in loop D are $12, 11, 10, 8, 7, 3, 2, 1$ with two gaps (10-8 and 7-3). The gaps depend on the tree structure. Therefore it is challenging to find the ID of the next node in constant time.

Our solution is the following. Let $p, q \in \{r_w\} \cup left(G, r_w)$ such that $o^{\rightarrow}(p) < o^{\rightarrow}(q)$. If there is a gap between the left-to-right preorder IDs of p and q (i.e., if $o^{\rightarrow}(q) - o^{\rightarrow}(p) > 1$) store $o^{\rightarrow}(p)$ for node q. p is the node which has to be processed right after q. We observe that in the case of a gap between $o^{\rightarrow}(p)$ and $o^{\rightarrow}(q)$, p is the first leaf node to the left of q. We introduce an array ln, which for each node $w \in G$ stores the left-to-right preorder of the first leaf node to the left of w. ln is defined as follows.

$$ln[o^{\rightarrow}(w)] = o^{\rightarrow}(w') \text{ , such that } w' \text{ is a leaf}, w' \in left(G, w), o^{\rightarrow}(w') \text{ is maximal}$$

Array ln is of the size $|G|$ and is precomputed in linear time using Algorithm 5. For each input tree to the RTED algorithm, two ln arrays are required: one for left-to-right preorder and one for right-to-left preorder.

Algorithm 5: Ln(G)

```
1 $ln \leftarrow$ an empty array of size $|G|$
2 $currentLeaf \leftarrow -1$        // $ln[o^{\rightarrow}(w)] = -1$ denotes no leaf node to the left of $w$
3 for $w \in G$ in left-to-right preorder do
4     $ln[o^{\rightarrow}(w)] \leftarrow currentLeaf$
5     if $w$ is leaf then $currentLeaf \leftarrow o^{\rightarrow}(w)$
6 return $ln$
```

If there is a gap between the current node l_w and the next node in loop D, then the ID of the next node is $ln[o^{\rightarrow}(l_w)]$. Otherwise, the next node has a consecutive ID and is $o^{\rightarrow}(l_w) - 1$. The gap exists only if the node with ID $o^{\rightarrow}(l_w) - 1$ has not been processed earlier in loop B, i.e., if $o^{\leftarrow}(o^{\rightarrow}(l_w) - 1) < o^{\leftarrow}(r_w)$.

Example 8.4. For the example tree in Figure 15, array ln looks as follows:

array ln

0	1	2	3	4	5	6	7	8	9	10	11	12
−1	−1	−1	2	3	3	3	3	3	8	8	8	11

The first node l_w processed in loop D is $l_w = r_w$ with $o^{\rightarrow}(l_w) = 12$. We compute the IDs of node l_w for the three following iterations of loop D. $o^{\leftarrow}(r_w) = 5$ does not change in loop D. Iteration 1: $o^{\leftarrow}(12 - 1) = 7 \geq 5$, thus the next ID is $12 - 1 = 11$. Iteration 2: $o^{\leftarrow}(11 - 1) = 6 \geq 5$, thus the next ID is $11 - 1 = 10$. Iteration 3: $o^{\leftarrow}(10 - 1) = 4 < 5$, thus the next ID is $ln[o^{\rightarrow}(10)] = 8$.

9. RELATED WORK

The first tree edit distance algorithm for the unrestricted edit model, which is also assumed in our work, was already proposed in 1979 by Tai [1979]. Tai's algorithm runs in $O(m^3n^3)$ time and space for two trees F and G with m and n nodes, respectively. Zhang and Shasha [1989] improve the complexity to $O(m^2n^2)$ time and $O(mn)$ space; for trees with l leaves and depth d the runtime is $O(mn \min(l_F, d_F) \min(l_G, d_G))$, which is much better than $O(m^2n^2)$ for some tree shapes. Klein [1998] uses heavy paths [Sleator and Tarjan 1983] to decompose the larger tree and gets an $O(n^2 m \log m)$ time and space algorithm, $m \geq n$. Demaine et al. [2009] also use heavy paths, but different from Klein they switch the trees such that the larger subtree is decomposed in each recursive step. Their algorithm runs in $O(n^2m(1 + \log \frac{m}{n}))$ time and in $O(mn)$ space, $m \geq n$. Although Demaine's algorithm is worst-case optimal [Demaine et al. 2009], it is slower than Zhang's algorithm for some interesting tree shapes, for example, balanced trees. Dulucq and Touzet [2005] compute a decomposition strategy in the first step, then use the strategy to compute the tree edit distance. They only consider strategies that decompose a single tree and get an algorithm that runs in $O(n^2m \log m)$ time and space.

A preliminary version of this work appeared in [Pawlik and Augsten 2011]. The present work extends [Pawlik and Augsten 2011] in two ways. (1) We analyse the indexing schemes for subproblems presented in literature and show that they are either not expressive enough [Zhang and Shasha 1989] or come with substantial (quadratic) overhead in time and/or space [Demaine et al. 2009]. We propose the root encoding, a new indexing scheme that requires only linear time and space and allows us to access subproblems in constant time. (2) Based on our new indexing scheme we present the *general single-path function*, which, in addition to the indexing problem, addresses the following issues of previously proposed single-path functions: (a) our single-path function works for any fanout (Δ^A is designed for binary trees only), (b) all path types are supported (Δ^L and Δ^R supports only left and right paths, respectively), and (c) left and right paths are processed efficiently (Δ^A is inefficient for these types). To the best of our knowledge, these issues have never been addressed in literature before. We evaluate our approach on both synthetic and real world data and compare it to the solutions of Zhang and Shasha [1989], Klein [1998], and Demaine et al. [2009].

Pawlik and Augsten [2014] modify the strategy computation for RTED [Pawlik and Augsten 2011] and reduce its memory requirements by at least 50%. This improvement is relevant and guarantees that the strategy computation does not use more memory than the actual distance computation. The memory optimization by Pawlik and Augsten [2014] directly applies to the strategy algorithm presented in this paper.

For specific tree shapes or a restricted set of edit operations faster algorithms have been proposed. Chen [2001] presents an $O(mn + l_F^2 n + l_F^{2.5} l_G) = O(m^{2.5} n)$ algorithm based on fast matrix multiplication, which is efficient for some instances, e.g., when one of the trees has few leaves. Chen and Zhang [2007] present an efficient algorithm for trees with long chains. By contracting the chains they reduce the size of tree F (G) from m to $\tilde{m}$ (n to $\tilde{n}$) and achieve runtime $O(mn + \tilde{m}^2\tilde{n}^2)$. Chawathe [1999] restricts the delete operation to leaf nodes. An external memory algorithm that reduces the tree edit distance problem to a well-studied shortest path problem is proposed. Other variants of the tree edit distance are discussed in a survey by Bille [2005]. Our algorithm adapts to any tree shape and we assume the unrestricted edit model.

Tree edit distance variants are also used for change detection in hierarchical data. Lee et al. [2004] and Chawathe et al. [1996] match tree nodes and compute a distance in $O(ne)$ time, where e is the distance between the trees. Cobéna et al. [2002] take advantage of element IDs in XML documents, which cannot be generally assumed. The X-Diff algorithm by Wang et al. [2003] allows leaf and subtree insertion and deletion, and node renaming. In order to achieve $O(n^2 f \log f)$ time complexity for trees with n nodes and maximum fanout f, only nodes with the same path to the root are matched. Finis et al. [2013] develop the RWS-Diff algorithm (Random Walk Similarity Diff) to compute an approximation of the edit distance in $O(n \log n)$ time. RWS-Diff works with both ordered and unordered trees and supports edit operations on nodes and subtrees. No guarantee about the quality of the approximation is provided.

Lower and upper bounds of the tree edit distance have been studied. Guha et al. [2002] propose a lower bound based on the string edit distance between serialized trees and an upper bound based on a restricted tree edit distance variant. Yang et al. [2005] decompose trees into so-called binary branches and derive a lower bound from the number of non-matching binary branches between two trees. Similarly, Augsten et al. [2010b] decompose the trees into pq-grams and provide a lower bound for an edit distance that gives higher weight to nodes with many children. The bounds have more efficient algorithms than the exact tree edit distance that we compute in this paper. Bounds are useful to prune exact distance computations when trees are matched with a similarity threshold. There is no straightforward way to build bounds into the dynamic programming algorithms for the exact tree edit distance to improve its performance.

The TASM (top-k approximate subtree matching) algorithm by Augsten et al. [2010a] identifies the top-k subtrees in a data tree with the smallest edit distances from a given query tree. TASM prunes distance computations for large subtrees of the data tree and achieves a space complexity that is independent of the data size. The pruning makes use of the top-k guarantee, which is not given in our scenario.

Garofalakis and Kumar [2005] embed the tree edit distance with subtree move as an additional edit operation into a numeric vector space equipped with the standard L_1 distance norm. They compute an efficient approximation of the tree edit distance with asymptotic approximation guarantees. In our work, we compute the exact tree edit distance.

In this work we assume ordered trees. For unordered trees the problem is NP-hard [Zhang et al. 1992].

10. EXPERIMENTS

We empirically evaluate our RTED algorithm on both real world and synthetic datasets and compare it to the fastest algorithms proposed in literature: the algorithm by Zhang and Shasha [1989] (Zhang-L), which uses only left paths to decompose the trees; the symmetric version of this algorithm that always uses right paths (Zhang-R); Klein's algorithm [1998] (Klein-H), which uses heavy paths in only one tree; the worst-case optimal solution by Demaine et al. [2009] (Demaine-H), which decomposes both trees with heavy paths. We include the improvements proposed by Demaine et al. [2009] into Klein's algorithm such that it runs in quadratic space. All algorithms were implemented as single-thread applications in Java 1.6 and run on a single core of a 64-core AMD Opteron 2.3GHz server. The source code is available online[5].

The Datasets. We test the algorithms on both synthetic and real world data. We generate synthetic trees of six different shapes. The left branch, right branch, and the zigzag tree (Figure 16) are constructed such that the strategies Zhang-L, Zhang-R, and Demaine-H are optimal, respectively; for the full binary tree both Zhang-L and Zhang-R are optimal; the mixed tree shape does not favour any of the algorithms; the random trees vary in depth and fanout (with maximum depth 14 and maximum fanout 6).

We choose three real world datasets with different characteristics. SwissProt[6] is an XML protein sequence database with 50000 medium sized and flat trees (max. depth 4, max. fanout 346, avg. size 187); TreeBank[7] is an XML representation of natural language syntax trees with 56385 small and deep trees (avg. depth 10.4, max. depth 35, avg. size 68). TreeFam[8] stores 16138 phylogenetic trees of animal genes (avg. depth 14, max. depth 158, avg. fanout 2, avg. size 95).

Number of Relevant Subproblems. We first compare the number of relevant subproblems computed by each of the algorithms for different tree shapes. The relevant subproblems are the constant time operations that make up the complexity of the algo-

[5]http://www.inf.unibz.it/dis/projects/tree-edit-distance/
[6]http://www.expasy.ch/sprot/
[7]http://www.cis.upenn.edu/~treebank/
[8]http://www.treefam.org/

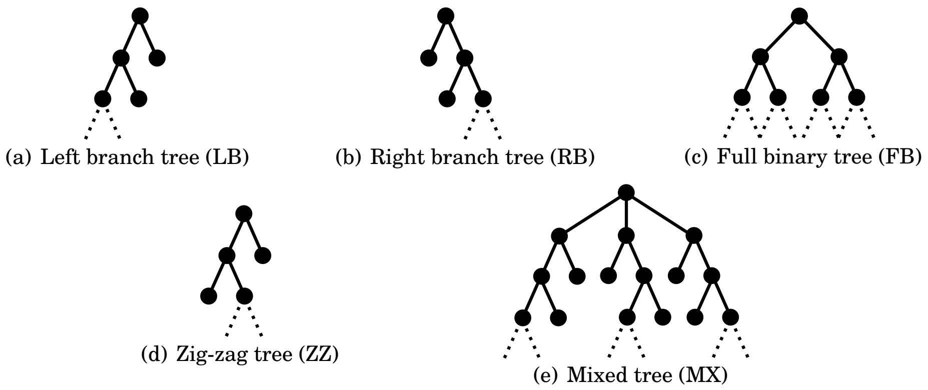
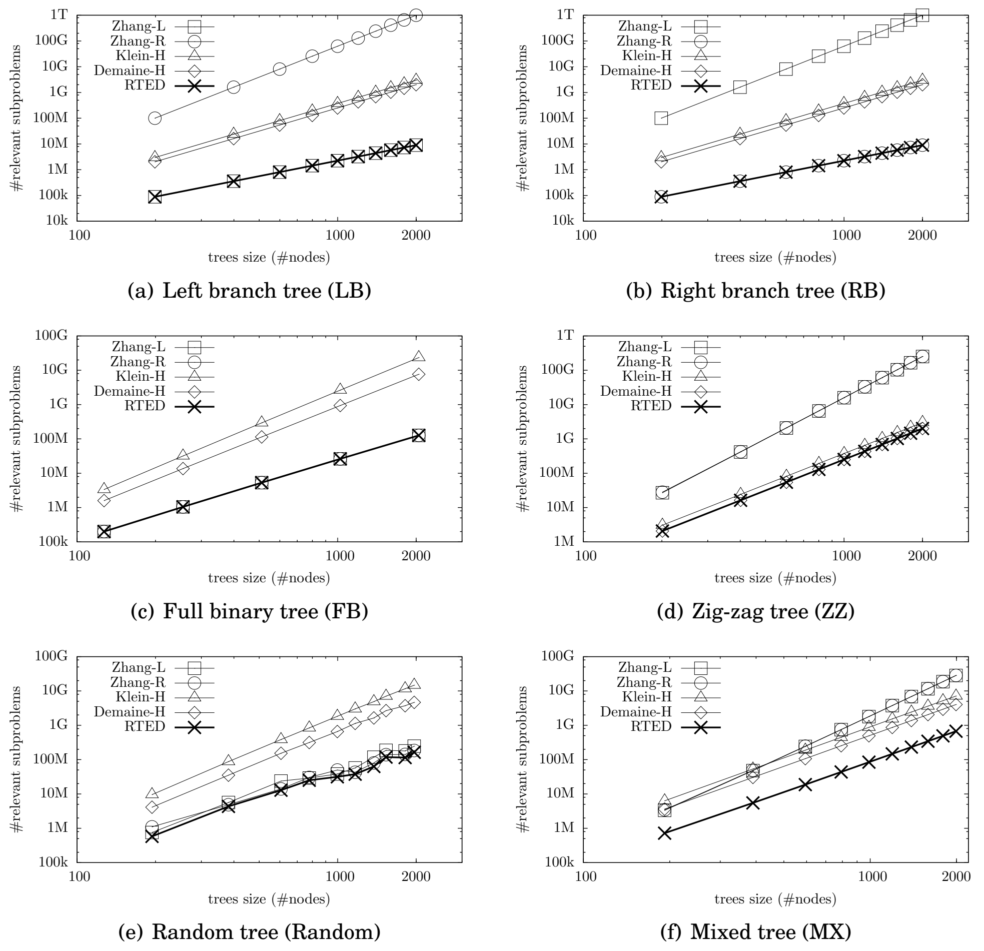

(a) Left branch tree (LB) (b) Right branch tree (RB) (c) Full binary tree (FB) (d) Zig-zag tree (ZZ) (e) Mixed tree (MX)

Fig. 16. Shapes of the synthetic trees.

rithm. We create trees with different shapes (random trees and the five shapes shown in Figure 16) with sizes varying between 200 and 2000 nodes. We count the number of relevant subproblems computed by each of the algorithms for pairs of identical trees. The results are shown in Figure 17. Figure 17 uses log-log scale (where slope k and intercept a of a straight line correspond to the power and constant term of the function $y = ax^k$, respectively). Each of the tested algorithms, except RTED, degenerates for at least one of the tree shapes, i.e., its asymptotic runtime behaviour is higher than necessary. RTED either wins together with the best competitor (left branch, right branch, full binary, zigzag), or is the only winner (random, mixed), which confirms our analytic results. Note that the trees, for which a competitor draws level with RTED, were designed such that the strategy of the respective competitor is optimal. The differences are substantial, for example, for the left branch trees with 2000 nodes (Figure 17(a)) Zhang-R produces 111259 times more relevant subproblems than RTED; for the mixed trees with 2000 nodes, the best competitor of RTED (Demaine-H) does 6 and the worst competitor (Zhang-L) 42 times more computations.

Runtime on Synthetic Data. In Figure 18 (log-log scale) we compare the runtime of the algorithms for different tree shapes. Figures 18(a) and 18(b) show the runtime for left an right branch trees. RTED scales well. Zhang-L is good for left branch tree and runs into its worst case for right branch trees; Zhang-R shows an opposite behaviour. The runtime for the full binary tree is given in Figure 18(c). Zhang-L, Zhang-R, and RTED scale well with the tree size, whereas the runtime of Demaine-H and Klein-H grows fast. This is expected since Demaine-H and Klein-H must compute many more subproblems than the other algorithms (cf. Figure 17(c)). The strategy of Zhang-L and Zhang-R is optimal for the full binary tree such that Zhang-L, Zhang-R, and RTED compute the same number of subproblems. The overall runtime of RTED is higher since RTED pays a small additional cost for computing the strategy. Further, our implementation of Zhang-L and Zhang-R is optimized for the hard-coded strategy, such that the runtime per subproblem is smaller for Zhang-L and Zhang-R than for RTED (by a constant factor below two). For other tree shapes, this advantage of Zhang-L and Zhang-R is outweighed by the smaller number of subproblems that RTED must compute. For the zig-zag trees in Figure 18(d), Zhang-L and Zhang-R are slower than RTED, Klein-H, and Demaine-H. RTED's overhead for the strategy computation is negligible compared to the overall runtime, and RTED is the fastest. For the mixed tree shapes in Figure 18(f), RTED scales very well, while the runtime of all other algorithms grows fast with the tree size.

Scalability of Similarity Join. We generate two datasets of trees, T_{small} and T_{big}. T_{small} consists of 15 trees (three for each of the shapes LB, RB, ZZ, FB, and Random;

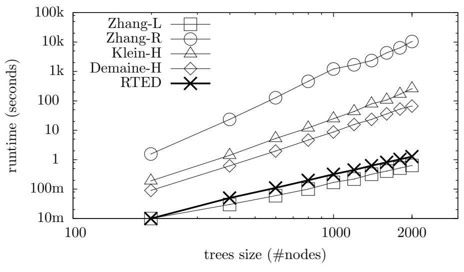


(a) Left branch tree (LB)
(b) Right branch tree (RB)
(c) Full binary tree (FB)
(d) Zig-zag tree (ZZ)
(e) Random tree (Random)
(f) Mixed tree (MX)

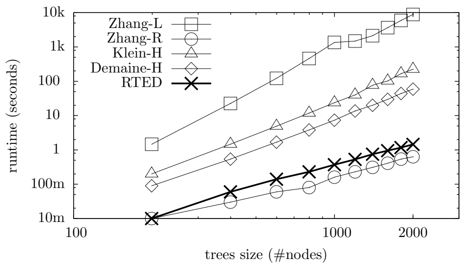
Fig. 17. Number of relevant subproblems for different algorithms and tree shapes (log-log scale).

sizes between 100 and 600 nodes). T_{big} consists of 13 trees (three for each of the shapes LB, RB, ZZ, and Random with sizes between 1600 and 2000 nodes; one FB tree of size 2047). We compute the following joins: $T_{\text{small}} \bowtie T_{\text{small}}$, $T_{\text{big}} \bowtie T_{\text{big}}$, and $T_{\text{small}} \bowtie T_{\text{big}}$, where the join predicate is of the form $\delta(T_1, T_2) < \tau$. Table III shows the runtime (average of three runs) and the number of relevant subproblems computed in the join. RTED widely outperforms all other algorithms. Different from the previous experiment, where we tested on pairs of identical trees, the join computes the distance between all pairs of trees, regardless of their shape. The competitors of RTED degenerate for some pairs of trees with different shapes, leading to high runtimes. For example, both Zhang-L and Zhang-R run into their worst case for pairs of unbalanced trees (LB and RB in our experiment). Unbalanced trees appear frequently in practice, for example, in our phylogenetic dataset. The runtime per relevant subproblem differs between the solutions and is the smallest for Zhang-L and Zhang-R.

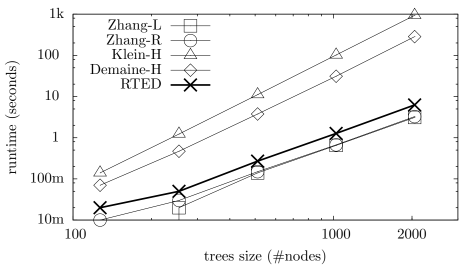
Overhead of Strategy Computation. RTED computes the optimal strategy before the tree edit distance is computed. We measure the overhead that the strategy computation adds to the overall runtime. We run our tests on three datasets: SwissProt,



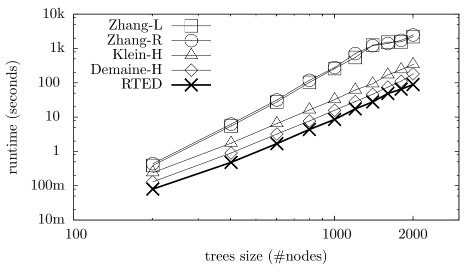
(a) Left branch tree (LB)



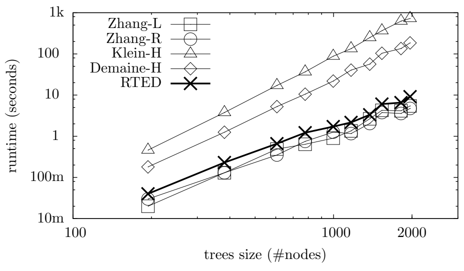
(b) Right branch tree (RB)



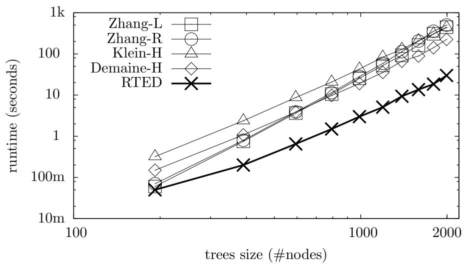
(c) Full binary tree (FB)



(d) Zig-zag tree (ZZ)



(e) Random tree (Random)

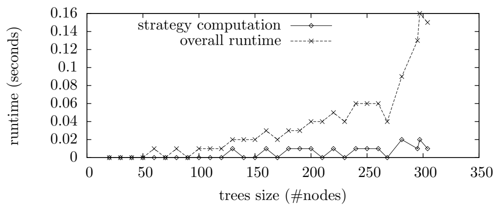


(f) Mixed tree (MX)

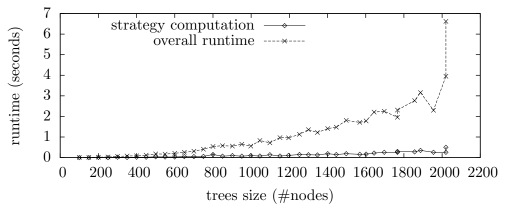
Fig. 18. Runtime of the tree edit distance algorithms for different tree shapes (log-log scale).

Table III. Joining trees with different shapes and sizes.

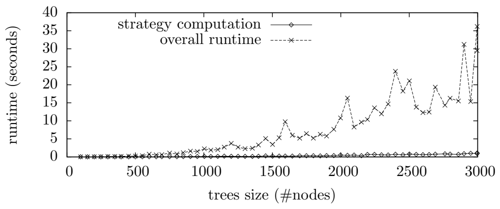
	$T_{\text{small}} \bowtie T_{\text{small}}$		$T_{\text{big}} \bowtie T_{\text{big}}$		$T_{\text{small}} \bowtie T_{\text{big}}$	
Algorithm	Time [sec]	#Rel. sub.	Time [sec]	#Rel. sub.	Time [sec]	#Rel. sub.
Zhang-L	597	$50.57 \cdot 10^9$	183802	$14.00 \cdot 10^{12}$	8905	$84.16 \cdot 10^{10}$
Zhang-R	537	$50.34 \cdot 10^9$	169469	$13.98 \cdot 10^{12}$	9052	$83.91 \cdot 10^{10}$
Klein-H	407	$9.07 \cdot 10^9$	32811	$0.76 \cdot 10^{12}$	2411	$4.78 \cdot 10^{10}$
Demaine-H	177	$4.84 \cdot 10^9$	22204	$0.35 \cdot 10^{12}$	1424	$2.71 \cdot 10^{10}$
RTED	53	$1.59 \cdot 10^9$	4844	$0.10 \cdot 10^{12}$	361	$0.94 \cdot 10^{10}$



(a) TreeBank dataset.



(b) SwissProt dataset.



(c) Synthetic random trees.

Fig. 19. Overhead of the strategy computation in the overall execution time of RTED.

Table IV. Ratio of relevant subproblems computed by RTED w.r.t. the (a) best and (b) worst competitor.

(a) RTED to the best competitor

↓ tree sizes →	<500	500-1000	>1000
<500	94.4%	90.1%	88.2%
500-1000	91.4%	85.6%	84.2%
>1000	89.4%	85.9%	86.9%

(b) RTED to the worst competitor

↓ tree sizes →	<500	500-1000	>1000
<500	30.6%	17.6%	17.3%
500-1000	21.1%	8.9%	7.9%
>1000	18.3%	7.7%	5.6%

TreeBank, and a synthetic data set with random trees that vary in size, fanout, and depth. We pick tree pairs at regular size intervals and compute the tree edit distance. For a given tree size n we pick the two trees in the dataset that are closest to n; the size value used in the graphs is the average size of the two trees. Figure 19 shows the runtime for computing the strategy for a pair of trees and the overall runtime of RTED. The strategy computation scales well and the fraction it takes in the overall runtime decreases with the tree size. The spikes in the overall runtime are due to the different tree shapes, for which more or less efficient strategies exist. The runtime for the strategy computation is independent of the tree shape. These empirical results confirm our analytic findings in Section 6.

Scalability on Real World Data. We measure the scalability of the tree edit distance algorithms for the TreeFam dataset with phylogenetic trees. We partition the dataset by size (less than 500, 500–1000, more than 1000 nodes) and compute the distance between pairs of trees of two partitions. We measure the performance of RTED as the percentage of relevant subtree computations that RTED performs with respect to the best (Table IV(a)) and the worst competitor (Table IV(b)). The best and worst competitors vary between the pairs of partitions. The tables show the results for random samples of size 20 from each partition. RTED always computes less subproblems than all its competitors (84.2% to 94.4% w.r.t. the best competitor, 5.6% to 30.6% w.r.t. the worst competitor). The advantage increases with the tree size. For the partitions with the largest trees, RTED produces 18 times less relevant subproblems than the worst competitor. This experiment shows the relevance of RTED in practical settings. The wrong choice among the competing algorithms for a specific dataset may lead to highly varying runtimes. RTED is robust to different tree shapes and always performs well.

11. CONCLUSION

In this paper we discussed the tree edit distance between ordered labelled trees. We introduced the concept of path strategies, which generalizes most previous approaches, including the best known algorithms for the tree edit distance. Our general tree edit distance algorithm, GTED, executes any path strategy in $O(n^2)$ space. We developed an efficient algorithm for computing the optimal path strategy for GTED. We introduced a novel single-path function, which forms the core of GTED and efficiently processes any type of path; previously proposed single-path functions were either limited to specific paths or were inefficient for some path types. Our single-path function leverages a new index structure, which requires only $O(n)$ space to store and retrieve intermediate results in constant time.

The resulting RTED algorithm runs in $O(n^2)$ space as its best competitors, and its $O(n^3)$ runtime complexity is worst-case optimal. Compared to previous algorithms, RTED is efficient for any tree shape. In particular, we showed that the number of subproblems computed by RTED is at most the number of subproblems that its best competitor must compute. Our empirical evaluation confirmed that RTED is efficient for any input and outperforms the other approaches, especially when the tree shapes within a dataset vary.

ACKNOWLEDGMENT

This work was partially supported by the RARE project funded by the Autonomous Province of Bolzano - South Tyrol, Italy, and by the SyRA project of the Free University of Bozen-Bolzano, Italy.

REFERENCES

Tatsuya Akutsu. 2010. Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics. *IEICE Transactions on Information and Systems* 93-D, 2 (2010), 208–218.

Nikolaus Augsten, Denilson Barbosa, Michael H. Böhlen, and Themis Palpanas. 2010a. TASM: Top-k Approximate Subtree Matching. In *Int. Conf. on Data Engineering (ICDE)*. 353–364.

Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2010b. The *pq*-gram distance between ordered labeled trees. *ACM Transactions on Database Systems (TODS)* 35, 1 (2010).

John Bellando and Ravi Kothari. 1999. Region-Based Modeling and Tree Edit Distance as a Basis for Gesture Recognition. In *Int. Conf. on Image Analysis and Processing (ICIAP)*. 698–703.

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1-3 (2005), 217–239.

Sudarshan S. Chawathe. 1999. Comparing Hierarchical Data in External Memory. In *Int. Conf. on Very Large Data Bases (VLDB)*. 90–101.

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *ACM SIGMOD Int. Conf. on Management of Data*. 493–504.

Shihyen Chen and Kaizhong Zhang. 2007. An Improved Algorithm for Tree Edit Distance Incorporating Structural Linearity. In *Int. Conf. on Computing and Combinatorics (COCOON)*. 482–492.

Weimin Chen. 2001. New Algorithm for Ordered Tree-to-Tree Correction Problem. *J. of Algorithms* 40, 2 (2001), 135–158.

Gregory Cobena, Serge Abiteboul, and Amélie Marian. 2002. Detecting Changes in XML Documents. In *Int. Conf. on Data Engineering (ICDE)*. 41–52.

Sara Cohen. 2013. Indexing for Subtree Similarity-Search Using Edit Distance. In *ACM SIGMOD Int. Conf. on Management of Data*. 49–60.

Theodore Dalamagas, Tao Cheng, Klaas-Jan Winkel, and Timos K. Sellis. 2006. A methodology for clustering XML documents by structure. *Information Systems* 31, 3 (2006), 187–228.

Davi de Castro Reis, Paulo Braz Golgher, Altigran Soares da Silva, and Alberto H. F. Laender. 2004. Automatic web news extraction using tree edit distance. In *Int. World Wide Web Conf. (WWW)*. 502–511.

Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2009. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)* 6, 1 (2009).

Serge Dulucq and Hélène Touzet. 2005. Decomposition algorithms for the tree edit distance problem. *J. of Discrete Algorithms* 3, 2-4 (2005), 448–471.

Jan P. Finis, Martin Raiber, Nikolaus Augsten, Robert Brunel, Alfons Kemper, and Franz Färber. 2013. RWS-Diff: Flexible and Efficient Change Detection in Hierarchical Data. In *ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 339–348.

Minos Garofalakis and Amit Kumar. 2005. XML Stream Processing Using Tree-Edit Distance Embeddings. *ACM Transactions on Database Systems (TODS)* 30, 1 (2005), 279–332.

Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. 2002. Approximate XML joins. In *ACM SIGMOD Int. Conf. on Management of Data*. 287–298.

Amaury Habrard, José Manuel Iñesta Quereda, David Rizo, and Marc Sebban. 2008. Melody Recognition with Learned Edit Distances. In *Joint IAPR Int. Work. on Structural, Syntactic, and Statistical Pattern Recognition (SSPR/SPR)*. 86–96.

Holger Heumann and Gabriel Wittum. 2009. The tree-edit-distance, a measure for quantifying neuronal morphology. *BMC Neuroscience* 10, Suppl 1 (2009), P89.

Shahab Kamali and Frank Wm Tompa. 2013. Retrieving Documents with Mathematical Content. In *ACM SIGIR Int. Conf. on Research and Development in Information Retrieval*. 353–362.

Yeonjung Kim, Jeahyun Park, Taehwan Kim, and Joongmin Choi. 2007. Web Information Extraction by HTML Tree Edit Distance Matching. In *Int. Conf. on Convergence Information Technology (ICCIT)*. 2455–2460.

Philip N. Klein. 1998. Computing the Edit-Distance between Unrooted Ordered Trees. In *European Symposium on Algorithms (ESA)*. 91–102.

Philip N. Klein, Srikanta Tirthapura, Daniel Sharvit, and Benjamin B. Kimia. 2000. A tree-edit-distance algorithm for comparing simple, closed shapes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 696–704.

Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. 2013. On Repairing Structural Problems In Semi-structured Data. *Proceedings of the VLDB Endowment (PVLDB)* 6, 9 (2013), 601–612.

Kyong-Ho Lee, Yoon-Chul Choy, and Sung-Bae Cho. 2004. An Efficient Algorithm to Compute Differences between Structured Documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 16, 8 (2004), 965–979.

Zhiwei Lin, Hui Wang, and Sally I. McClean. 2010. Measuring Tree Similarity for Natural Language Processing Based Information Retrieval. In *Int. Conf. on Applications of Natural Language to Information Systems (NLDB)*. 13–23.

Bin Ma, Lusheng Wang, and Kaizhong Zhang. 2002. Computing similarity between RNA structures. *Theoretical Computer Science* 276, 1-2 (2002), 111–132.

Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: A Robust Algorithm for the Tree Edit Distance. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (2011), 334–345.

Mateusz Pawlik and Nikolaus Augsten. 2014. A Memory-Efficient Tree Edit Distance Algorithm. In *Int. Conf. on Database and Expert Systems Applications (DEXA)*. 196–210.

Daniel Dominic Sleator and Robert Endre Tarjan. 1983. A Data Structure for Dynamic Trees. *J. of Computer and System Sciences* 26, 3 (1983), 362–391.

Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. of the ACM (JACM)* 26, 3 (1979), 422–433.

Yuan Wang, David J. DeWitt, and Jin yi Cai. 2003. X-Diff: An Effective Change Detection Algorithm for XML Documents. In *Int. Conf. on Data Engineering (ICDE)*. 519–530.

Rui Yang, Panos Kalnis, and Anthony K. H. Tung. 2005. Similarity Evaluation on Tree-structured Data. In *ACM SIGMOD Int. Conf. on Management of Data*. 754–765.

Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. on Computing (SICOMP)* 18, 6 (1989), 1245–1262.

Kaizhong Zhang, Richard Statman, and Dennis Shasha. 1992. On the Editing Distance Between Unordered Labeled Trees. *Information Processing Letters* 42, 3 (1992), 133–139.

Received Month Year; revised Month Year; accepted Month Year