Information Systems 56 (2016) 157–173

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Tree edit distance: Robust and memory-efficient

Mateusz Pawlik*, Nikolaus Augsten

University of Salzburg, Department of Computer Sciences, Jakob-Haringer-Str. 2, 5020 Salzburg, Austria

ARTICLE INFO

Article history:
Received 12 August 2014
Received in revised form
25 May 2015
Accepted 2 August 2015
Available online 28 August 2015

Keywords:
Tree edit distance
Similarity search
Approximate matching

ABSTRACT

Hierarchical data are often modelled as trees. An interesting query identifies pairs of similar trees. The standard approach to tree similarity is the tree edit distance, which has successfully been applied in a wide range of applications. In terms of runtime, the state-of-the-art algorithm for the tree edit distance is RTED, which is guaranteed to be fast independent of the tree shape. Unfortunately, this algorithm requires up to twice the memory of its competitors. The memory is quadratic in the tree size and is a bottleneck for the tree edit distance computation.

In this paper we present a new, memory efficient algorithm for the tree edit distance, AP-TED (All Path Tree Edit Distance). Our algorithm runs at least as fast as RTED without trading in memory efficiency. This is achieved by releasing memory early during the first step of the algorithm, which computes a decomposition strategy for the actual distance computation. We show the correctness of our approach and prove an upper bound for the memory usage. The strategy computed by AP-TED is optimal in the class of all-path strategies, which subsumes the class of LRH strategies used in RTED. We further present the AP-TED$^+$ algorithm, which requires less computational effort for very small subtrees and improves the runtime of the distance computation. Our experimental evaluation confirms the low memory requirements and the runtime efficiency of our approach.

1. Introduction

Data with hierarchical dependencies are often modelled as trees. Tree data appear in many applications, ranging from hierarchical data formats like JSON or XML to merger trees in astrophysics [33]. An interesting query computes the similarity between two trees. The standard measure for tree similarity is the tree edit distance, which is defined as the minimum-cost sequence of node edit operations that transform one tree into another. The tree edit distance has been successfully applied in bioinformatics (e.g., to find similarities between RNA secondary structures [1,29], neuronal cells [21], or glycan structures [3]), in image analysis [7], pattern recognition [25], melody recognition [19], natural language processing [28], information extraction [12,23], and document retrieval [22], and has received considerable attention from the database community [5,8–11,16–18,26,27].

* Corresponding author. Tel.: +43 66280446348.
E-mail addresses: mateusz.pawlik@sbg.ac.at (M. Pawlik), nikolaus.augsten@sbg.ac.at (N. Augsten).

http://dx.doi.org/10.1016/j.is.2015.08.004
0306-4379/

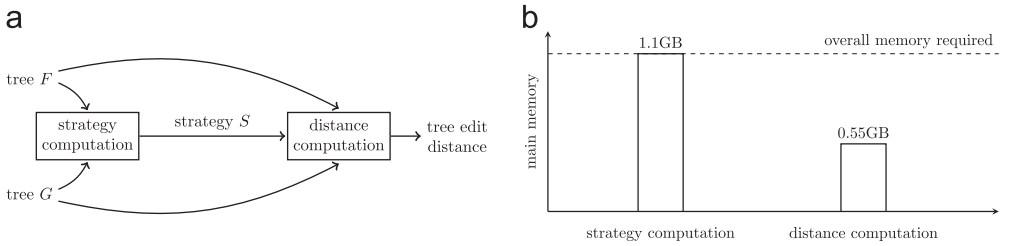


Fig. 1. Strategy computation requires more memory than actual distance computation. (a) Two-step algorithm for tree edit distance and (b) strategy vs. distance computation.

The fastest algorithms for the tree edit distance (TED) decompose the input trees into smaller subtrees and use dynamic programming to build the overall solution from the subtree solutions. The key difference between various TED algorithms is the decomposition strategy, which has a major impact on the runtime. Early attempts to compute TED [13,24,37] use a hard-coded strategy, which disregards or only partially considers the shape of the input trees. This may lead to very poor strategies and asymptotic runtime differences of up to a polynomial degree. The most recent development is the Robust Tree Edit Distance (RTED) algorithm [30], which operates in two steps (cf. Fig. 1(a)). In the first step, a decomposition strategy is computed. The strategy adapts to the input trees and is shown to be optimal among all previously proposed strategies. The actual distance computation is done in the second step, which executes the strategy.

In terms of runtime, the overhead for the strategy computation in RTED is small compared to the gain due to the better strategy. Unfortunately, this does not hold for the main memory consumption. Fig. 1(b) shows the memory usage for two example trees (perfect binary trees) of 8191 nodes: the strategy computation requires 1.1 GB of RAM, while the execution of the strategy (i.e., the actual distance computation) requires only 0.55 GB. Thus, for large instances, the strategy computation is the bottleneck and the fallback is a hard-coded strategy. This is undesirable since the gain of a good strategy grows with the instance size. Reducing the memory requirements of the strategy computation affects the maximum tree size that can be processed. This is crucial especially for large trees like abstract syntax trees of source code repositories [15,20] (Emacs: $>$10k nodes and MythTV: $>$50k nodes) or merger trees in astrophysics[1] [33].

In this paper we propose the AP-TED algorithm, which solves the memory problem of the strategy computation. This is achieved by computing the strategy bottom-up using dynamic programming and releasing part of the memorization tables early. We prove that our algorithm requires at most 1/3 of the memory that is needed by RTED's strategy computation [30]. As a result, the memory cost of the strategy computation is never above the cost of the distance computation. Our extensive experimental evaluation on various tree shapes, which require very different strategies, confirms our analytic memory bound and shows that our algorithm is often much better than its theoretical upper bound. For some tree shapes, it even runs in linear space, while the RTED strategy algorithm always requires quadratic space.

In addition to reducing the memory usage, AP-TED computes the optimum in a larger class of strategies than RTED. Strategies are expressed by root-leaf paths that guide the decomposition of the input trees. A path decomposes a tree into subtrees by deleting nodes and edges on a root-leaf path. Each resulting subtree is recursively decomposed by a new root-leaf path. RTED computes the optimal LRH strategy. An LRH strategy considers only left, right, and heavy paths. The left (right) root-leaf path connects each parent with its first (last) child; the heavy path connects the parent with the rightmost child that roots the largest subtree. AP-TED considers *all* root-leaf paths and is not limited to left, right, and heavy paths. Thus, our strategy is at least as good as the strategies used by RTED. To the best of our knowledge, this is the first algorithm to compute the optimal all-path strategy. The runtime complexity of our strategy algorithm is $O(n^2)$ as for the RTED strategy. This result is surprising since in each recursive step we need to consider a linear number of paths compared to only three paths (left, right, and heavy) in the RTED strategy. Our empirical evaluation suggests that in practice our strategy algorithm is even slightly faster than the RTED strategy algorithm since it allocates less memory.

On the distance computation side, we observe that a large number of subproblems that result from the tree decompositions are very small trees with one or two nodes only. We show that a significant boost can be achieved by treating these cases separately. We introduce the AP-TED$^+$ algorithm, which leverages that fact and achieves runtime improvements of more than 50% in some cases.

Summarizing, the contributions of this paper are the following:

- *Memory efficiency*. We substantially reduce the memory requirements w.r.t. previous strategy computation algorithms by traversing the trees bottom-up and systematically releasing memory early. The resulting AP-TED algorithm always consumes less memory for the strategy computation than for the actual distance computation and thus breaks the

[1] Accessible at http://www.mpa-garching.mpg.de/millennium/.

bottleneck of previous algorithms. (We show the correctness of our approach and prove an upper bound for the memory usage.)

- *Optimal all-path strategy*. The decomposition strategy used by AP-TED is optimal in the class of all-path strategies. This class generalizes LRH strategies and contains all strategies of previous TED algorithms. Although our strategy algorithm must consider more paths, it is as efficient as the strategy algorithm in RTED (quadratic in the input size).
- *New single-path functions*. We develop AP-TED$^+$, which leverages two new single-path functions to compute the distance of subtree pairs when one of the subtrees is small. This case occurs frequently during the decomposition process. Our new single-path functions run in linear time and at most linear space, which substantially improves over the single-path functions Δ^L, Δ^R, and Δ^I used in RTED [30]. To take full advantage of the new functions, we integrate them into the strategy computation to obtain better strategies. Our experiments confirm the significant runtime improvement.

The paper is structured as follows. Section 2 sets the stage for our discussion of strategy algorithms. In Section 3 we define the problem, and we present our AP-TED algorithm in Section 4. The memory efficient implementation of the strategy computation in AP-TED is discussed in Section 5. The AP-TED$^+$ algorithm is presented in Section 6. We treat related work in Section 7, experimentally evaluate our solution in Section 8, and conclude in Section 9.

2. Notation and background

2.1. Notation

We follow the notation of [30] when possible. 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. Each node has a *label*, which is not necessarily unique within the tree. The nodes of a tree 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 a descendant of g, then $f < g'$ for all descendants g' of g. The tree traversal that visits all nodes in ascending order is the *postorder* traversal.

In an edge (v,w), node v is the *parent* and w is the *child*, $p(w) = v$. A node with no parent is a *root* node, a node without children is a *leaf*.

F_v is the *subtree rooted in node* v of F iff F_v is a tree, $N(F_v) = \{x: x = v$ or x is a descendant of v in $F\}$, and $E(F_v) \subseteq E(F)$. A *path* γ in F is a subtree of F in which each node has at most one child. By $\gamma^*(F)$ we denote the set of all root-leaf paths in F. The *left path* $\gamma^L(F) \in \gamma^*(F)$ recursively connects a parent to its leftmost child. Similarly, the *right path* $\gamma^R(F) \in \gamma^*(F)$ connects a parent to its rightmost child. The *heavy path* connects a parent to the child which roots the rightmost largest subtree. An *inner path* is a path in $\gamma^*(F)$ which is neither left nor right.

We use the following short notation: By $|F| = |N(F)|$ we denote the size of F, we write $v \in F$ for $v \in N(F)$. $F - \gamma$ is the set of *relevant subtrees* of tree F with respect to the path γ obtained by removing path γ from F: $F - \gamma = \{F_v: \exists x(x \in \gamma \wedge v \notin \gamma \wedge p(x) = p(v))\}$.

Example 1. The nodes of tree F in Fig. 2 are $N(F) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, the edges are $E(F) = \{(v_{13}, v_4), (v_{13}, v_{10}), (v_{13}, v_{12}), (v_4, v_1), (v_4, v_3), (v_3, v_2), (v_{10}, v_5), (v_{10}, v_9), (v_9, v_8), (v_8, v_6), (v_8, v_7), (v_{12}, v_{11})\}$, the node labels are shown in italics in the figure. The root of F is v_{13}, and $|F| = 13$. F_{v_8} with nodes $N(F_{v_8}) = \{v_6, v_7, v_8\}$ and edges $E(F_{v_8}) = \{(v_8, v_6), (v_8, v_7)\}$ is a subtree of F. The path γ that connects v_{13} and v_7 is the heavy path in F, and $F - \gamma = \{F_{v_4}, F_{v_5}, F_{v_6}, F_{v_{12}}\}$. $\gamma^L(F)$ is the path from v_{13} to v_1 and $\gamma^R(F)$ is the path from v_{13} to v_{11}. The paths from v_{13} to v_2, v_5, v_6, and v_7, respectively, are the inner paths. The postorder traversal visits the nodes of F in the following order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}$.

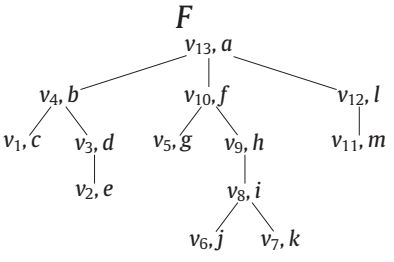


Fig. 2. Example tree.

2.2. Strategy and distance computation in RTED

The tree edit distance is the minimum-cost sequence of node edit operations that transforms tree F into G. The standard edit operations are: delete a node, insert a node, and rename the label of a node. The state-of-the-art algorithms compute the tree edit distance by implementing a well-known recursive solution [34] using dynamic programming [13,30,34,37]. They devise so-called *decomposition strategies* to decompose the input trees into smaller subtree pairs for which the distance is computed first. The results of smaller subproblems are used to compute the distances of larger problems. Pawlik and Augsten [30] introduce *path strategies* as a subset of all possible decomposition strategies and propose a two-step process for the TED computation: (a) strategy computation and (b) distance computation (cf. Fig. 1(a)).

(a) In the first step, a path strategy is computed, which maps each pair of subtrees (F_v, G_w) of two input trees F and G to a root-leaf path in either F_v or G_w. RTED considers LRH strategies which allow left, right, and heavy paths only. The resulting strategy is optimal in the sense that it minimizes the number of subproblems (i.e., the number of distances) that must be computed. In this paper we consider *all-path strategies* which, in contrast to LRH strategies, do not have limitation on the path type.

Definition 1. An *all-path strategy* S for two trees F and G maps each pair of subtrees (F_v, G_w), $v \in F$, $w \in G$, to a root-leaf path γ in one of the subtrees, such that $\gamma \in \gamma^*(F_v) \cup \gamma^*(G_w)$ [30].

(b) In the second step, the tree edit distance is computed. First, the input trees are decomposed into *relevant subtree* pairs according to the strategy. Then, the corresponding distances are computed using so-called *single-path functions*. The single-path function updates a *distance matrix* D, which stores distances between subtree pairs and is filled during the distance computation [30].

Definition 2. Given are two subtrees F_v, G_w, a path $\gamma \in \gamma^*(F_v)$, and the distances between all relevant subtrees of F_v with respect to γ and all subtrees of G_w, i.e., between each pair $(F_{v'}, G_{w'})$: $F_{v'} \in F_v - \gamma \wedge w' \in G_w$. A *single-path function* $\Delta(F_v, G_w)$ computes the distances between all subtrees rooted on path γ in F_v and every subtree of G_w, i.e., between all pairs $(F_{v'}, G_{w'})$: $v' \in \gamma \wedge w' \in G_w$.

Existing algorithms for single-path functions differ in the path type which they can process (Δ^L: left path, Δ^R: right path, and Δ^I: inner path), their cost (number of subproblems that are computed), and memory usage [30,32]. Depending on the strategy, different single-path functions must be used, resulting in a different overall cost for computing the tree edit distance. Although Δ^I can processes any inner path, RTED considers only heavy paths. In this paper, we always assume the improved version of Δ^I as presented in [32], which requires less memory than the version in [30].

3. Problem definition

As outlined in previous sections, the path strategies introduced by Pawlik and Augsten [30] generalize all state-of-the-art algorithms for computing the tree edit distance. They consider the class of LRH strategies and show optimality. However, LRH strategies limit the paths to be left, right, or heavy. We observe that allowing *all* paths leads to less expensive strategies. Another drawback of the RTED algorithm is the fact that the computation of the optimal strategy requires more space than executing the strategy, i.e., computing the actual tree edit distance. This limits the maximum size of the tree instances which can be processed in given memory. We moreover observe that many subtrees resulting from the optimal strategy are small. This is true for any input. The single-path functions used in RTED are inefficient in such cases and better solutions should be devised.

Based on the issues in RTED, our goal is an algorithm with the following properties:

- *Memory efficient.* The strategy computation should not be a main memory bottleneck. The optimal strategy must be computed *and* stored within the memory bounds of the distance computation.
- *Allowing all paths in the strategy.* The strategies should not be limited to only left, right, and heavy paths. The algorithm for the optimal strategy should consider all paths and run in $O(n^2)$ time.
- *Fast for small subtrees.* Executing the expensive single-path functions of RTED for small subtrees is a significant overhead for the entire algorithm. Whenever possible, the distances of small subtrees should be computed efficiently.

4. AP-TED algorithm

Until now, only LRH strategies have been considered in literature [13,24,30,37]. They are limited to left, right and heavy paths only. LRH strategies are only a fraction of all possible path strategies. There may exist non-LRH path strategies that lead to better solutions. In principle, all possible path strategies must be checked for the best result. In this section we present AP-TED, a new algorithm that computes the tree edit distance with the optimal all-path strategy. The core of AP-TED is a new algorithm to compute the optimal all-path strategy efficiently in small memory. Then, similar to RTED, the strategy

is executed using single-path functions. We develop a recursive formula that computes the cost of the optimal *all-path strategy* and an algorithm which implements the formula efficiently. In Section 5 we present techniques to improve the memory requirements of the strategy computation in AP-TED.

4.1. Cost of the optimal all-path strategy

The tree edit distance is computed by executing a sequence of so-called single-path functions [30]. The sequence is defined by the strategy. The cost of computing the distance is the cost sum of executing all single-path functions specified by the strategy. Depending on the path type (left, right, and inner), different single-path functions must be used. Each single-path function has a different cost. To compute the distance with the lowest cost, all possible path strategies must be checked.

The cost formula in Fig. 3 performs an exhaustive search in the space of all possible path strategies. At each recursive step, either F or G is decomposed with a path from $\gamma^*(F)$ or $\gamma^*(G)$, respectively. For each of the choices, the cost of the resulting relevant subtrees must be computed to find the strategy with the minimal cost.

Given a path, the cost is the sum of the optimal costs for the relevant subtrees with respect to the path plus the cost of executing the single-path function. For each path type we use the single-path function with the lowest processing cost (number of subproblems which must be computed). For left and right paths we use the costs of Δ^L and Δ^R, respectively. Let $\gamma^I(F) = \{\gamma \in \gamma^*(F)\colon \gamma \neq \gamma^L(F) \wedge \gamma \neq \gamma^R(F)\}$ be the set of all *inner paths* of tree F. For inner paths we use the cost of Δ^I [30,32]. We denote the cost of a single-path function by $|\Delta^{L/R/I}|$.

Theorem 1. *Given the single path functions Δ^L, Δ^R, and Δ^I, the cost formula in Fig. 3 computes the cost of the optimal all-path strategy.*

Proof. The proof is by induction. *Base case*: Both trees are single nodes: $|\Delta(F,G)| = 1$ for all path types [30] and $\text{cost}(F,G) = 1$. *Inductive hypothesis*: $\text{cost}(F',G)\colon F' \in F-\gamma$, $\gamma \in \gamma^*(F)$ and $\text{cost}(G',F)\colon G' \in G-\gamma$, $\gamma \in \gamma^*(G)$ are optimal. The cost for a path γ in F consists of the execution cost for the respective single path function ($|\Delta^{L/R/I}(F,G)|$ if γ is a left/right/inner path) and the cost sum of the relevant subtrees $F' \in F-\gamma$. The cost computation for a path in G is analogous. All paths in F and G are considered, and we pick the path with the minimum cost, thus $\text{cost}(F,G)$ is optimal.□

4.2. Strategy computation in AP-TED

In this section we present the AP-TED strategy algorithm (Algorithm 1), which computes the optimal all-path strategy. The algorithm implements the all-path cost formula in Fig. 3 and is as efficient as the RTED strategy algorithm in terms of runtime, but requires significantly less memory. Our early deallocation technique is implemented in lines 6, 7, and 38, and is described in detail in Section 5.

For the all-path strategy, similarly to the RTED strategy, we compute the optimal path for each pair of subtrees. Different from [30], however, at each step we have to evaluate a linear number of paths in both trees instead of only left, right, and heavy. This makes the computation of the optimal all-path strategy much more challenging. A straightforward extension of the RTED strategy algorithm, which evaluates a linear number of paths for each pair of subtrees, requires $O(n^3)$ time. This is prohibitive because the strategy computation must not increase the overall complexity of the tree edit distance computation, which is $O(n^2)$ in the best case.

$$\text{cost}(F,G) = \min \begin{cases} |\Delta^L(F,G)| + \sum_{F' \in F-\gamma^L(F)} \text{cost}(F',G) \\ |\Delta^R(F,G)| + \sum_{F' \in F-\gamma^R(F)} \text{cost}(F',G) \\ |\Delta^I(F,G)| + \min_{\gamma' \in \gamma^I(F)} \{\sum_{F' \in F-\gamma'} \text{cost}(F',G)\} \\ |\Delta^L(G,F)| + \sum_{G' \in G-\gamma^L(G)} \text{cost}(G',F) \\ |\Delta^R(G,F)| + \sum_{G' \in G-\gamma^R(G)} \text{cost}(G',F) \\ |\Delta^I(G,F)| + \min_{\gamma' \in \gamma^I(G)} \{\sum_{G' \in G-\gamma'} \text{cost}(G',F)\} \end{cases}$$

Fig. 3. Formula for the cost of the optimal all-path strategy.

Algorithm 1. AP-TED strategy.

Input: trees F and G
L_F, R_F, I_F: arrays of size $|F|$
STR: array of arrays of size $|F||G|$
$L_G, R_G, I_G, Path_w, leafRow$: arrays of size $|G|$
fill each cell of $leafRow$ with 0
for $v \leftarrow 1$ to $|F|$ *in postorder* **do**
  if v *is leaf* **then** $L_F[v] \leftarrow R_F[v] \leftarrow I_F[v] \leftarrow leafRow$
  if *row for* $p(v)$ *is not allocated* **then** allocate a row for $p(v)$ in $L_F,\ R_F,\ I_F$
  for $w \leftarrow 1$ *to* $|G|$ *inpostorder* **do**
    if w *is leaf* **then** $L_G[w] \leftarrow R_G[w] \leftarrow I_G[w] \leftarrow 0$
    $C \leftarrow \{(|\Delta^L(F_v, G_w)| + L_F[v][w], \gamma^L(F_v)),$
      $(|\Delta^I(F_v, G_w)| + I_F[v][w], STR[v][w]),$
      $(|\Delta^R(F_v, G_w)| + R_F[v][w], \gamma^R(F_v)),$
      $(|\Delta^L(G_w, F_v)| + L_G[w], \gamma^L(G_w)),$
      $(|\Delta^I(G_w, F_v)| + I_G[w], Path_w[w]),$
      $(|\Delta^R(G_w, F_v)| + R_G[w], \gamma^R(G_w))\}$
    $(c_{min}, \gamma_{min}) \leftarrow (c, \gamma)$ such that $(c, \gamma) \in C$ and $c = \min\{c' : (c', \gamma') \in C\}$
    if v *is not root* **then**
      $R_F[p(v)][w] \overset{+}{\leftarrow} c_{min}$
      $c_{tmp} \leftarrow -c_{min} + I_F[v][w]$
      if $c_{tmp} < I_F[p(v)][w]$ **then**
        $I_F[p(v)][w] \leftarrow c_{tmp}$
        $STR[p(v)][w] \leftarrow STR[v][w]$
      if $v \in \gamma^R(F_{p(v)})$ **then**
        $I_F[p(v)][w] \overset{+}{\leftarrow} R_F[p(v)][w]$
        $R_F[p(v)][w] \overset{+}{\leftarrow} R_F[v][w] - c_{min}$
      $L_F[p(v)][w] \overset{+}{\leftarrow} \begin{cases} L_F[v][w] & \text{if } v \in \gamma^L(F_{p(v)}) \\ c_{min} & \text{otherwise} \end{cases}$
    if w *is not root* **then**
      $R_G[p(w)] \overset{+}{\leftarrow} c_{min}$
      $c_{tmp} \leftarrow -c_{min} + I_G[w]$
      if $c_{tmp} < I_G[p(w)]$ **then**
        $I_G[p(w)] \leftarrow c_{tmp}$
        $Path_w[p(w)] \leftarrow Path_w[w]$
      if $w \in \gamma^R(G_{p(w)})$ **then**
        $I_G[p(w)] \overset{+}{\leftarrow} R_G[p(w)]$
        $R_G[p(w)] \overset{+}{\leftarrow} R_G[w] - c_{min}$
      $L_G[p(w)] \overset{+}{\leftarrow} \begin{cases} L_G[w] & \text{if } w \in \gamma^L(G_{p(w)}) \\ c_{min} & \text{otherwise} \end{cases}$
    $STR[v, w] \leftarrow \gamma_{min}$
  if v *is not leaf* **then** deallocate row of v in $L_F,\ R_F,\ I_F$
return STR

AP-TED strategy is similar to RTED strategy in that the sums in the cost formula are computed incrementally in a bottom-up fashion and are stored in the cost arrays. The cost arrays of size $|F||G|$ are L_F, R_F, and I_F ($|F||G|$ is the array size without our early deallocation technique). L_F, R_F, and I_F store the final cost values for each pair (F_v, G_w), $v \in F, w \in G$, for left, right, and best inner path in F_v, respectively:

- $L_F[v][w] = \sum_{F' \in F_v - \gamma^L(F_v)} \text{cost}(F', G_w)$,

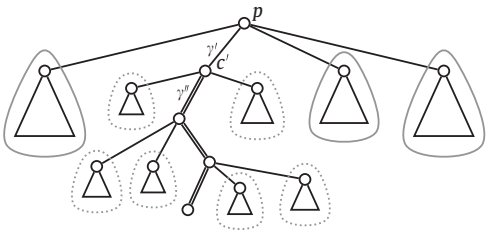


Fig. 4. Subtree F_p, its children subtrees – solid line, and path subtrees – dotted line.

- $R_F[v][w] = \sum_{F' \in F_v - \gamma^R(F_v)} \text{cost}(F', G_w)$,
- $I_F[v][w] = \min_{\gamma' \in \gamma^I(F_v)} \{\sum_{F' \in F_v - \gamma'} \text{cost}(F', G_w)\}$.

The cost arrays of size $|G|$ are L_G, R_G, and I_G. They store the cost sums between all relevant subtrees of G w.r.t. the corresponding path and a specific subtree F_v:

- $L_G[w] = \sum_{G' \in G_w - \gamma^L(G_w)} \text{cost}(G', F_v)$,
- $R_G[w] = \sum_{G' \in G_w - \gamma^R(G_w)} \text{cost}(G', F_v)$,
- $I_G[w] = \min_{\gamma' \in \gamma^I(G_w)} \{\sum_{G' \in G_w - \gamma'} \text{cost}(G', F_v)\}$.

The algorithm iterates over each pair of subtrees (F_v, G_w) in postorder of the nodes; v and w are postorder IDs of nodes in F and G, respectively. The cost sums stored in the cost arrays are used to compute the minimal cost c_{min} and the corresponding path $\gamma_{\min}$ (lines 10–16). The cost sums for the parent nodes, i.e., for the pairs $(F_{p(v)}, G_w)$ and $(F_v, G_{p(w)})$, are updated in the lines 17–36 ($a \overset{+}{\leftarrow} b$ stands for $a \leftarrow a + b$). Finally, the best path, γ_{min}, is stored in the strategy array *STR* (line 37) which encodes the optimal strategy.

The challenge is to implement lines 3 and 6 of the cost formula in Fig. 3, which require to find the best inner path in F and G, respectively. While there is a single heavy path, there can be many inner paths. Our solution is to rewrite lines 3 and 6 of the cost formula such that they can be computed incrementally while processing the children of a node with a constant time operation for each child. To simplify the discussion, we focus on line 3, which computes the best inner path for tree F.

The key to our solution is the following observation. The relevant subtrees for a given path $\gamma' \in \gamma^*(F_p)$ in subtree F_p fall into two disjoint categories of subtrees shown in Fig. 4:

(1) *Children subtrees*. All relevant subtrees rooted in the children of p, except the subtree rooted in child c', which is the child on path γ'.
(2) *Path subtrees*. All relevant subtrees in $F_{c'}$ for path γ'', which is the sub-path of γ', i.e., a root-leaf path of $F_{c'}$.

We distinguish children and path subtrees and rewrite the cost sum as follows:

$$\sum_{F' \in F_p - \gamma'} \text{cost}(F', G) = \overbrace{\underbrace{\sum_{c \in ch(p)} \text{cost}(F_c, G)}^{A_p} - \overbrace{\text{cost}(F_{c'}, G)}^{C_{c'}}}^{\text{cost for children subtrees}} + \overbrace{\underbrace{\sum_{F'' \in F_{c'} - \gamma''} \text{cost}(F'', G)}_{B_{c',\gamma''}}}^{\text{cost for path subtrees}} \tag{1}$$

where p is a node in F, $ch(p)$ denotes the children of p, $\gamma' \in \gamma^*(F_p)$, c' is the child of p on path γ' (i.e., $c' \in ch(p)$ and $c' \in \gamma'$), γ'' is the sub-path of γ' that traverses $F_{c'}$ (i.e., $N(\gamma') \backslash N(\gamma'') = \{p\}$ and $E(\gamma') \backslash E(\gamma'') = \{(p, c')\}$).

Using Eq. (1), we express $R_F[p(v)][w]$ and $I_F[p(v)][w]$ by formulas that are useful to compute these values incrementally. Let $c_r(p)$ denote the rightmost child of p, $A_p = \sum_{c \in ch(p)} \text{cost}(F_c, G)$, $B_{c',\gamma''} = \sum_{F'' \in F_{c'} - \gamma''} \text{cost}(F'', G)$, $C_{c'} = \text{cost}(F_{c'}, G)$:

$$R_F[p(v)][w] = B_{p(v),\gamma^R(F_{p(v)})} = A_{p(v)} - C_{c_r(p(v))} + B_{c_r(p(v)),\gamma^R(F_{c_r(p(v))})} = A_{p(v)} - C_{c_r(p(v))} + R_F[c_r(p(v))][w] \tag{2}$$

$$I_F[p(v)][w] = \min_{\gamma' \in \gamma^I(F_{p(v)})} \{B_{p(v),\gamma'}\} = A_{p(v)} + \min_{c \in ch(p(v))} \{-C_c + \min_{\gamma'' \in \gamma^I(F_c)} \{B_{c,\gamma''}\}\} = A_{p(v)} + \min_{c \in ch(p(v))} \{-C_c + I_F[c][w]\} \tag{3}$$

Eqs. (2) and (3) are used in Algorithm 1 to incrementally compute the cost sums for the parent node $p(v)$ while traversing its children. The postorder traversal of the nodes ensures that we process all children of $p(v)$ before node $p(v)$ itself. The child $c_r(p(v))$ is traversed last. In Eq. (2), we compute the cost sum for the relevant subtrees w.r.t. the right path in $F_{p(v)}$, which is stored in $R_F[p(v)][w]$. We use the previously computed sum stored in $R_F[c_r(p(v))][w]$. $C_{c_r(p(v))}$ is the value c_{min} for the node $c_r(p(v))$, i.e., the minimal cost of computing the distance for the subtree pair $(F_{c_r(p(v))}, G_w)$. The sum $A_{p(v)}$ is computed incrementally by summing up the values c_{min} for each child of $p(v)$ (cf. line 18). $A_{p(v)}$ is stored temporarily in $R_F[p(v)][w]$ until the last child (rightmost child) is processed.

In Eq. (3), for each child of node $p(v)$, we compute the value $c_{tmp} = -C_c + I_F[c][w]$, which is the sum of (a) a negative cost c_{min} for the particular child c and (b) the cost sum of the best inner path in F_c stored in $I_F[c][w]$. While traversing the children of $p(v)$, we compute the minimal value c_{tmp} and store it in $I_F[p(v)][w]$ (cf. lines 20–21). The minimum is found when the last child is traversed, i.e., the node $c_r(p(v))$. Then, $R_F[p(v)][w]$ and $I_F[p(v)][w]$ are updated with the correct cost values w.r.t. Eqs. (2) and (3), respectively (cf. lines 24–25).

While traversing the children, the best inner paths in F_v and G_w seen so far are stored in the strategy array *STR* and in the array $Path_w$, respectively, e.g., $STR[p(v)][w]$ stores the best inner path which corresponds to the minimal cost sum stored in $I_F[p(v)][w]$ (line 22), $Path_w[p(w)]$ stores the best inner path of the minimal cost sum stored in $I_G[p(w)]$ (line 32). After traversing the last child, $STR[p(v)][w]$ and $Path_w[p(w)]$ store the best inner paths corresponding to the minimal cost of computing the distance between the subtrees $(F_{p(v)}, G_w)$ and $(F_v, G_{p(w)})$, respectively. While traversing the nodes $p(v)$ and $p(w)$, the best inner path is compared to the respective left and right paths, and the overall best path is stored in the strategy array *STR* (line 37).

With the above analysis we show the correctness of the following theorems.

Theorem 2. *AP-TED strategy (Algorithm 1) is correct and computes the optimal path strategy.*

Proof. We show by induction that the cost arrays store correct values. *Base case*: For each pair of leaf nodes (v,w) the cost arrays are zero (lines 6 and 9). This is correct due to $F_v - \gamma_F = G_w - \gamma_G = \varnothing$, where $\gamma_F \in \gamma^*(F)$ and $\gamma_G \in \gamma^*(G)$. *Inductive hypothesis*: The values in the cost arrays for all children of node v are correct. We show that after processing all children, the values for v are correct. We show this for the array I_F. The proof for L_F, R_F, L_G, R_G, I_G is similar.

With Eq. (3), for node v and its children $ch(v)$,

$$I_F[v][w] = A_v + \min_{c \in ch(v)} \{-C_c + I_F[c][w]\}.$$

While processing the children of v we compute the summands as follows: (a) A_v is computed by summing up the cost $c_{min} = C_c$ for every child $c \in ch(v)$. (b) The minimal value $c_{tmp} = -C_c + I_F[c][v]$ is computed among all children of v. Due to the induction hypothesis the value $I_F[c][v]$ is correct for every node $c \in ch(v)$. Thus, the value stored in $I_F[v][w]$ is correct.

The strategy array *STR* maps each pair of subtrees to a root-leaf path and thus is a strategy by Definition 1. Only the minimum-cost paths are stored in the strategy array, thus *STR* stores the optimal strategy.□

Theorem 3. *Time and space complexity of the AP-TED strategy algorithm are $O(n^2)$, where $n = \max\{|F|, |G|\}$.*

Proof. Algorithm 1 iterates over each pair of subtrees (F_v, G_w), $v \in F, w \in G$, hence the innermost loop is executed $|F||G|$ times. Inside the innermost loop a constant number of arithmetic operations and array lookups is performed. The costs of the single-path functions in lines 10–15 are precomputed in $O(|F|+|G|)$ time and space. Four arrays of size $|F||G|$ and five arrays of size $|G|$ are used. Thus, the overall time and space complexity is $O(|F||G|)$.□

5. Memory efficiency in AP-TED

The main memory requirement is a bottleneck of the tree edit distance computation. The strategy computation in RTED exceeds the memory needed for executing the strategy. Our AP-TED strategy algorithm reduces the memory usage by at least 2/3 and never uses more memory than the execution of the strategy. We achieve that by decreasing the maximum size of the data structures used for strategy computation.

5.1. Memory analysis

The asymptotic space complexity is quadratic for both computing the strategy and executing it (cf. Fig. 1(a)). However, due to different constants and depending on the strategy, the strategy computation may use twice the memory of the distance computation. This is undesirable since the strategy computation becomes a memory bottleneck of the overall algorithm.

We analyse the memory usage of the strategy and distance computation steps. For the moment we ignore the lines in Algorithm 1 that perform early deallocation (lines 6, 7, and 38). Then, the memory requirement of the strategy computation is dominated by three cost arrays L_F, R_F, I_F, and the strategy array *STR*, which take $4|F||G|$ space in total.

The memory of the distance computation depends on the strategy. The tree edit distance is computed by *single-path functions* which process a single path from the strategy. Different path types require a different amount of memory. Independent of the strategy, a distance matrix of size $|F||G|$ is used.

(a) If only left and/or right paths are used in the strategy, the single path functions require only one array of size $|F||G|$. Thus $2|F||G|$ space is needed in total.
(b) If an inner path is involved in the strategy, an array of size $|F||G|$ and an array of size $|G|^2$ are used, amounting to $2|F||G|+|G|^2$ space in total.

We observe that the distance computation always uses less memory than the strategy. Thus, the strategy computation is the limiting factor for the maximum tree size that can be processed in given memory.

The goal is to use less memory for the strategy computation than for the distance computation. The key observation is that some of the rows in the cost arrays are no longer needed once they have been read. We develop an early deallocation technique to minimize the size of the cost arrays.

5.2. Early deallocation of rows

The AP-TED strategy algorithm uses three cost arrays of quadratic size (L_F, R_F, I_F), where each row corresponds to a node $v \in F$. The outermost loop (line 5) iterates over the nodes in F in postorder. In each iteration only two rows of the cost arrays are accessed: the row of node v is read and the row of its parent $p(v)$ is updated. Once the row of node v has been read it will not be accessed later and thus can be deallocated. In our analysis we count the maximum number of concurrently needed rows.

We observe that a row which corresponds to a leaf node stores only zeros. Thus we store a single read-only row for all leaf nodes in all cost arrays and call it *leafRow* (cf. line 6).

We define a set of four operations that we need on the rows of a cost array.

- read(v)/read(*leafRow*) – read the row of node v/read *leafRow*,
- allocate(v) – allocate a row of node v,
- update(v) – update values in the row of node v,
- deallocate(v) – deallocate the row of node v.

The same operations are synchronously applied to the rows of all cost arrays (L_F, R_F, I_F). To simplify the discussion, we consider a single cost array and sum up in the end.

AP-TED strategy (Algorithm 1) dynamically allocates and deallocates rows in the cost arrays. Lines 6, 7, and 38 implement our technique. At each iteration step, i.e., for each node $v \in F$ in postorder, the following steps are performed on a cost array:

if v is not root of F	
if row for $p(v)$ does not exist: allocate($p(v)$)	(line 7)
read(v)	(lines 10–15)
update($p(v)$)	(lines 17–36)
if v is not a leaf: deallocate(v)	(line 38)

Depending on the position of a node in the tree, a specific sequence of operations must be performed. Fig. 5 shows the possible sequences and assigns them to the respective node types.

Example 2. We study the number of concurrently need rows for the example tree in Fig. 6. The table in the figure shows the operations performed for every node of the example tree and the rows stored in memory after each step. The subscript numbers of the nodes represent their postorder positions. We iterate over the nodes in postorder, i.e., $v_1, v_2, v_3, v_4, v_5, v_6$. Depending on the node type, different operations are performed that lead to the following operation sequences: $v_1 - S1$, $v_2 - S1$, $v_3 - S3$, $v_4 - S2$, $v_5 - S4$, $v_6 - S0$.

The cost array for the example tree has six rows (one for each node). From the table in Fig. 6 we see that a maximum of only three rows (*leafRow*, rows for v_3 and v_4) need to be stored at the same time.

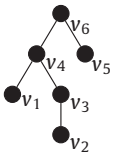
We now consider the general case and count the number of rows that must be kept in memory at the same time. Each of the operation sequences allocates and/or deallocates rows. With $s(v) \in \{S0, S1, S2, S3, S4\}$ we denote the operation sequence required for node v, with $d(S)$, $S \in \{S0, S1, S2, S3, S4\}$, we denote the difference between the number of allocated and deallocated rows by a particular operation sequence S, i.e., $d(S0) = 0$, $d(S1) = 1$, $d(S2) = 0$, $d(S3) = -1$, $d(S4) = 0$. We count the maximum number of concurrently needed rows during the postorder traversal of tree F with the following formula:

$$cnr(F) = 1 + \max_{v \in F} \left\{ \sum_{w \in F, w < v} d(s(w)) \right\}$$

$S0$ = $\langle\rangle$
$S1$ = $\langle$allocate($p(v)$), read(*leafRow*), update($p(v)$)$\rangle$
$S2$ = $\langle$allocate($p(v)$), read(v), update($p(v)$), deallocate(v)$\rangle$
$S3$ = $\langle$read(v), update($p(v)$), deallocate(v)$\rangle$
$S4$ = $\langle$read(*leafRow*), update($p(v)$)$\rangle$

	leaf	non-leaf
leftmost child	$S1$	$S2$
not leftmost child	$S4$	$S3$
root		$S0$

Fig. 5. Row operations depending on the node position.



node	operations	rows in memory
v_1	allocate(v_4), read(*leafRow*), update(v_4)	*leafRow*, v_4
v_2	allocate(v_3), read(*leafRow*), update(v_3)	*leafRow*, v_3, v_4
v_3	read(v_3), update(v_4), deallocate(v_3)	*leafRow*, v_4
v_4	allocate(v_6), read(v_4), update(v_6), deallocate(v_4)	*leafRow*, v_6
v_5	read(*leafRow*), update(v_6)	*leafRow*, v_6
v_6	-	-

Fig. 6. Row operations on an example tree.

The *leafRow* is always kept in memory. In addition, each node contributes with $d(S)$ rows, where S is the operation sequence required by the node. For the tree in Fig. 6 we have $cnr(F) = 1+\max\{1, 1+1, 1+1-1,\ 1+1-1+0, 1+1-1+0+0, 1+1-1+0+0+0\} = 3$.

We observe that only operation sequence $S1$ adds more rows than it deallocates, i.e., $d(S1) > 0$. An upper bound on the number of concurrently needed rows is given by the number of nodes that require operation sequence $S1$. The only type of nodes that falls into operation sequence $S1$ is a leaf node which is the leftmost child of its parent.

Lemma 1. *The maximum number of concurrently needed rows of each cost array is* $\lfloor |F|/2 \rfloor$.

Proof. The maximum number of the concurrently needed rows for a tree F is the number of its leftmost-child leaf nodes, which is bound by $\lfloor |F|/2 \rfloor$: All leftmost-child leaf nodes, must have a different parent. This results in $\lfloor |F|/2 \rfloor$ different parent nodes for $\lfloor |F|/2 \rfloor$ leftmost-child leaf nodes. If $|F|$ is odd, then $\lfloor |F|/2 \rfloor + \lfloor |F|/2 \rfloor = |F| - 1$ and we can add one more leftmost-child leaf to the tree. The new node will either become the child or the left sibling of an existing leftmost-child leaf node. Thus, by adding a new leftmost-child leaf we loose an existing one. If $|F|$ is even, $\lfloor |F|/2 \rfloor + \lfloor |F|/2 \rfloor = |F|$. Thus, the maximum number of the leftmost-child leaf nodes is $\lfloor |F|/2 \rfloor$.□

An example of a tree with $\lfloor |F|/2 \rfloor$ leftmost-child leaf nodes is the right branch tree (a tree symmetric to the left branch tree in Fig. 8(a)).

Lemma 2. *The memory required for the strategy computation in AP-TED is* $2.5|F||G|$, *including the output* (*strategy array STR*).

Proof. By expiring rows, with Lemma 1, the sizes of the cost arrays L_F, R_F, and I_F are reduced from $|F||G|$ to at most $0.5|F||G|$. The strategy array *STR* requires $|F||G|$ memory. The size of *STR* is not reduced since it stores the final strategy after the algorithm terminates. Thus, the AP-TED strategy algorithm requires $2.5|F||G|$ memory in the worst case.□

Using our early deallocation technique, AP-TED requires only $2.5|F||G|$ memory for computing and storing the strategy. Unfortunately, it is still $0.5|F||G|$ above the lower bound for the distance computation. In the following section we show that we can reduce the memory to $2|F||G|$ by traversing the trees in a clever order.

5.3. Tightening the memory upper bound

We reduce the number of concurrently needed rows in the cost matrices to $\lceil |F|/3 \rceil$. Our solution is based on the following observation. The AP-TED strategy algorithm iterates over the nodes in postorder. One can think of a symmetric algorithm which iterates over the nodes in right-to-left postorder.[2] Then, the maximum number of concurrently needed rows is equal to the maximum number of the rightmost-child leaves in a tree (instead of leftmost-child leaves). For the right branch tree, which is the worst case for postorder, the right-to-left postorder algorithm needs only three rows, including the *leafRow*. Thus, the direction of the tree traversal matters. Let $l(F)$ and $r(F)$ be the number of leftmost-child and rightmost-child leaf nodes in F, respectively. We use the following rule: if $l(F) < r(F)$, then use (left-to-right) postorder, otherwise use right-to-left postorder to compute the strategy. We call the order resulting from our rule an *adaptive order*.

We observe that whenever $l(F)$ is high, $r(F)$ is low and vice versa. Moreover, in practice $\min\{l(F), r(F)\}$ is typically much below $|F|/2$. We show that the number of rows is bounded to $\lceil |F|/3 \rceil$.

The first step is to show that for any node $v \in F$ at most the rows for v and its ancestors, i.e., the nodes on the path from v to the root, are concurrently stored in main memory.

Lemma 3. *For any node* $v \in F$ *processed by AP-TED strategy* (*Algorithm* 1), *at most the rows for* v *and its ancestors are concurrently stored in main memory*.

Proof. We show the proof for the postorder traversal. The reasoning for the right-to-left postorder is similar.

Given a node $v \in F$, we divide the nodes of F into four disjoint sets visualized in Fig. 7:

- $L = \{l \in F : l < v \wedge l$ is not descendant of $v\}$,

[2] In right-to-left postorder, children are traversed from right to left before their parents. For example, traversing the tree in Fig. 6 in right-to-left postorder gives the following node sequence: $v_5, v_2, v_3, v_1, v_4, v_6$.

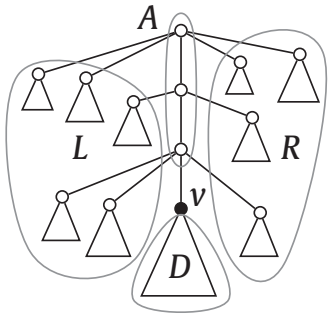


Fig. 7. Node sets L (left), R (right), A (ancestors), and D (descendants) in the proof of Lemma 3.

- $R = \{r \in F : r > v \wedge r$ is not ancestor of $v\}$,
- $A = \{a \in F : a$ is an ancestor of $v\}$,
- $D = \{d \in F : d$ is a descendant of $v\}$.

We show that while traversing node v during the strategy computation, the only nodes for which a row may be allocated are v and the nodes in A.

While node v is traversed, its row must be present in memory. All nodes in L and D precede v in postorder and have already been processed. Thus, their rows have been deallocated and are no longer in memory. The row of a node is first allocated when its leftmost child is processed. The nodes of R and all their descendants (which are also in R) succeed v in postorder and no row is allocated. Also the nodes in A succeed v in postorder and have not been processed. However, some nodes in A may have a leftmost child in $L \cup \{v\}$, thus the respective row may be allocated.□

Lemma 4. *For a given tree F, the strategy computation in AP-TED with adaptive order requires at most $\lceil |F|/3 \rceil$ rows in main memory.*

Proof. The proof is by contradiction. We assume a node v that has $k > \lceil |F|/3 \rceil$ ancestors for which a row must be stored in both the postorder and right-to-left postorder iteration of the tree.

- *Postorder*. If a row is allocated for an ancestor a of v, then a has a leftmost child which is not an ancestor of v. Node v may cause the row allocation for its parent $p(v)$. For the remaining $k-1$ ancestors of v there must be $k-1$ leftmost children.
- *Right-to-left postorder*. Similarly, for $k-1$ ancestors of v there must be $k-1$ rightmost children which are not ancestors of v.

Thus, F has at least $3k-1$ nodes (node v, k ancestors of v, $k-1$ leftmost children, $k-1$ rightmost children) and we get the following contradiction:

$$k > \left\lceil \frac{|F|}{3} \right\rceil \geq \left\lceil \frac{3k-1}{3} \right\rceil = \left\lceil k - \frac{1}{3} \right\rceil = k \square$$

Theorem 4. *Using an adaptive order, the memory required by AP-TED to compute the strategy is bounded by $2|F||G|$, including the output (strategy array).*

Proof. By Lemmas 3 and 4 we know that the maximum number of concurrently needed rows of each cost array is at most $\lceil |F|/3 \rceil$. The strategy array *STR* needs $|F||G|$ memory. This results in $3((|F|/3)|G|) + |F||G| = 2|F||G|$ memory.□

6. AP-TED$^+$ algorithm

The RTED algorithm computes the tree edit distance by executing the single-path functions for the subtree pairs resulting from the strategy. We observe that when one of the input trees in a single-path function is small, the distance can be computed more efficiently than with the existing single-path functions. We address two special cases, which are very frequent and have a high impact on the runtime: one- and two-node trees. We present AP-TED$^+$, a new algorithm that improves over previous algorithms, including AP-TED, in two ways. First, AP-TED$^+$ uses a new, efficient single-path functions for the cases of small subtrees. The new single-path functions significantly reduce memory and runtime compared to previous single-path functions. Second, AP-TED$^+$ leverages the lower costs of the new single-path functions to compute better strategies. AP-TED$^+$ first computes the optimal all path strategy considering the lower costs for small subtrees and then executes the strategy using the new single-path functions for small subtrees and the old single-path functions for all other cases.

6.1. One-node single-path function

Let F and G be the input trees to the TED algorithm, $c_d(a)$, $c_i(b)$, $c_r(a,b)$ the costs of deleting $a \in F$, inserting $b \in G$, and renaming the label of a to the label of b, respectively. Let F' and G' be the subtrees of F and G, such that G' is a one-node tree, $N(G') = \{w\}$, $E(G') = \emptyset$, and $|F'| \geq 1$. We assume that $c_r(v,w) \leq c_d(v) + c_i(w)$ for any node $v \in F'$ (otherwise rename will never contribute to the minimal distance). Then, the distance between F' and G' is computed with the following formula:

$$\delta_{1node}(F', G') = \sum_{v \in F'} c_d(v) + \min_{v \in F'}\{c_r(v,w) - c_d(v)\} \tag{4}$$

The distance between F' and G' is the cost of deleting all nodes in F' except the one for which the rename cost to the label of the node in G' is minimal. The rename cost is added to the final result.

$\delta_{1node}(F', G')$ can be evaluated in a single traversal of the bigger tree and does not require any data structures for storing intermediate results. In order to implement $\delta_{1node}(F', G')$ as a single-path function, we must store the resulting distances of all subtree pairs (F'_v, G'_w), $v \in F'$, $w \in G'$, in distance matrix D [30]. Algorithm 2 implements Eq. (4) as a single-path function.

Algorithm 2. $\Delta^{1node}(F', G')$.

Input: trees F', G', $|G'| = 1$, $N(G') = \{w\}$
1 $costSum \leftarrow 0$
2 **for** $v \in F'$ *in postorder* **do**
3 $\quad costSum \overset{+}{\leftarrow} c_d(v)$
4 $\quad match \leftarrow \min\{match, c_r(v,w) - c_d(v)\}$
5 $\quad D(F'_v, G'_w) \leftarrow costSum - match$

6.2. Two-node single-path function

Let now G' be a two-node tree, $N(G') = \{w_1, w_2\}$, $E(G') = \{(w_2, w_1)\}$. $anc(v)$ is the set of ancestors of node v in F' without v. Then, the distance between F' and G' is computed with Eq. (5). We limit our discussion to settings with the following properties: (a) $|F'| \geq 2$ (otherwise δ_{1node} can be used), (b) $c_r(v,w) \leq c_d(v) + c_i(w)$ for all nodes $v \in F'$ and $w \in G'$ (otherwise rename will never contribute to the minimal distance).

$$\delta_{2node}(F', G') = \left(\sum_{v \in F'} c_d(v)\right) + c_i(w_1) + c_i(w_2) + \min_{v_1 \in F'} \begin{cases} c_r(v_1, w_2) - (c_d(v_1) + c_i(w_2)) \\ c_r(v_1, w_1) - (c_d(v_1) + c_i(w_1)) + \min\limits_{v_2 \in anc(v_1)} \{c_r(v_2, w_2) - (c_d(v_2) + c_i(w_2))\} \end{cases} \tag{5}$$

Eq. (5) substitutes one or two pairs of deletion and insertion with a rename in order to find the minimal distance.

Algorithm 3. $\Delta^{2node}(F', G')$.

Input trees F', G', $|G'| = 2$, $N(G') = \{w_1, w_2\}$, $E(G') = \{(w_2, w_1)\}$
1 $match$: $[1..|F'|][0..2]$: boolean array initialized to false
2 **for** $v \in F'$ *in postorder* **do**
3 $\quad D(F'_v, G'_{w_2}) \leftarrow \max\{|F'_v|, 2\}$
4 $\quad$ **if** $match[v][2]$ **then**
5 $\quad\quad \lfloor D(F'_v, G'_{w_2}) \leftarrow |F'_v| - 2$
6 $\quad$ **else if** $match[v][0] \vee match[v][1]$ **then**
7 $\quad\quad \lfloor D(F'_v, G'_{w_2}) \leftarrow |F'_v| - 1$
8 $\quad$ **else if** v *is leaf* $\wedge (l(v) = l(w_1) \vee l(v) = l(w_2))$ **then**
9 $\quad\quad \lfloor D(F'_v, G'_{w_2}) \leftarrow 1$
10 $\quad D(F'_v, G'_{w_1}) \leftarrow |F'_v|$
11 $\quad$ **if** $match[v][0] \vee l(v) = l(w_1)$ **then**
12 $\quad\quad \lfloor D(F'_v, G'_{w_1}) \leftarrow |F'_v| - 1$
13 $\quad$ **if** v *is not root of* F' **then**
14 $\quad\quad match[p(v)][0] \leftarrow match[v][0] \vee l(v) = l(w_1) \vee match[p(v)][0]$
15 $\quad\quad match[p(v)][1] \leftarrow match[v][1] \vee l(p(v)) = l(w_2) \vee match[p(v)][1]$
16 $\quad\quad match[p(v)][2] \leftarrow match[v][2] \vee match[p(v)][2] \vee ((match[v][0] \vee l(v) = l(w_1)) \wedge l(p(v)) = l(w_2))$

Algorithm 3 implements Eq. (5) for the unit cost model, where for any node $v \in F'$ and $w \in G'$: $c_d(v) = c_i(w) = 1$, $c_r(v,w) = 0$ if the labels of v and w are equal and $c_r(v,w) = 1$ otherwise. Algorithm 3 computes the distance in a single postorder traversal of tree F'. It uses a two-dimensional boolean array *match* with three columns of length $|F'|$ to store the intermediate label matches:

- $match[v][0]$: $l(w_1)$ matches the label of some non-root node in F'_v,
- $match[v][1]$: $l(w_2)$ matches the label of some non-leaf node in F'_v,
- $match[v][2]$: there is a pair of nodes (v_1, v_2) in F'_v such that v_2 is an ancestor of v_1, $l(v_1) = l(w_1)$, and $l(v_2) = l(w_2)$.

For each node $v \in F'$, the subtree distances (F'_v, G'_{w_1}) and (F'_v, G'_{w_2}) are stored in the distance matrix D. Finally, the entries in array *match* for the parent of v are updated (lines 13–14): a label match in subtree F'_v is also valid for $F'_{p(v)}$.

6.3. Better strategy with new single-path functions

In order to take full advantage of one- and two-node single-path functions, they must be considered in the strategy computation. The reason is the difference in the computation costs (the number of computed distances). We first show that our new single-path functions, Δ^{1node} and Δ^{2node}, always have a lower cost than the single-path functions used in RTED (Δ^L, Δ^R, and Δ^I). In the worst case, the costs of $\Delta^{L/R/I}$ is cubic in the tree size, whereas the cost of our functions is always linear. The cost of a single-path function, $|\Delta(F,G)|$, for a pair of trees F and G, is the number of subproblems that must be computed.

Theorem 5. *For any two trees, F and G, such that $|G| \leq 2$, the following holds*:

$$|\Delta^{1/2node}(F,G)| \leq |\Delta^{L/R/I}(F,G)|$$

Proof. Both $\Delta^{1node}(F,G)$ and $\Delta^{2node}(F,G)$ are computed in a single traversal of tree F. For each node in F only a constant number of arithmetic operations and table lookups is performed, then $|\Delta^{1/2node}(F,G)| = |F|$. Depending on the context, for Δ^L, Δ^R, and Δ^I the following lower bounds hold [30]:

(a) $|\Delta^{L/R/I}(F,G)| \geq |F||G| + \sum_{F' \in F-\gamma} |\Delta^{L/R/I}(F',G)| \geq |F|$,
(b) $|\Delta^{L/R/I}(G,F)| \geq |G||F| \geq |F|$.□

The second step is to include the costs of the new functions in our AP-TED strategy algorithm to obtain better strategies. This is straightforward: when the minimal cost is computed in line 10 of Algorithm 1, we check the sizes of the subtrees and use the costs of our new single-path functions if applicable.

7. Related work

Tree edit distance algorithms. The tree edit distance has a recursive solution, which decomposes the input trees into smaller subtrees and subforests. The best known algorithms are dynamic programming implementations of this recursive solution, where small subproblems are computed first. The first tree edit distance algorithm was proposed by Tai [34]. It runs in $O(n^6)$ time and space where n is the number of tree nodes. The runtime complexity is given by the number of subproblems that must be solved. Zhang and Shasha [37] significantly improve the complexity and achieve $O(n^4)$ time and $O(n^2)$ space. Klein [24] uses so-called heavy paths, which guide the dynamic programming solution by ordering the subproblems resulting with $O(n^3 \log n)$ runtime and space complexity. Demaine et al. [13] developed an algorithm which requires $O(n^3)$ time and $O(n^2)$ space. They moreover show that their solution is worst-case optimal. Pawlik and Augsten [30] observe that the previous algorithms are efficient only for specific tree shapes and run into their worst case otherwise. They propose the general framework for computing the tree edit distance shown in Fig. 1(a) and present the RTED algorithm. RTED computes and executes the optimal LRH strategy and runs in $O(n^3)$ time and $O(n^2)$ space. Compared to previous algorithms, it does not run into the worst case if a better strategy exists. AP-TED, the solution presented in this paper, enjoys the same features but in addition (a) requires less memory and (b) executes the optimal all-path strategy, a superclass of LRH strategies. In [32], Pawlik and Augsten further improve RTED and introduce a novel indexing scheme, called root encoding, that stores and retrieves intermediate distance results in constant time. They develop a new single-path function, Δ^A, that combines the features of the previous solutions for left, right, and inner paths [13,30,37]. Thanks to the root encoding, Δ^A requires less memory for inner paths than Δ^I as proposed by Demaine et al. [13]. We leverage these results in our AP-TED algorithm and build the optimizations of Δ^A for inner paths into our single-path function Δ^I.

Path strategies. Each of the state-of-the-art algorithms uses a specific set of paths to decompose trees and order the computation of subproblems in the dynamic programming solution. The algorithms by Zhang and Shasha [37], Klein [24] and Demaine et al. [13] use hard-coded strategies which do not require the strategy computation step. The resulting algorithms are efficient only for specific tree shapes. Dulucq and Touzet [14] compute a decomposition strategy in the first step, then use the strategy to compute the tree edit distance. Since they only consider strategies that decompose a single tree, the resulting algorithm runs in $O(n^3 \log n)$ time and space. On-the-fly strategies that decompose both trees were

introduced by Pawlik and Augsten [30]. The resulting algorithm, RTED, requires $O(n^3)$ time and $O(n^2)$ space, and their strategy is shown to be optimal in the class of LRH strategies, which covers all previous solutions. Unfortunately, the strategy computation in RTED requires more memory than the actual distance computation.

In a preliminary version of this paper [31], we tackle the memory problem of the strategy computation in RTED [30] and prove an $\lceil n/2 \rceil$ upper bound for the number of rows in the cost matrices. This, however, is not enough to keep the strategy, which is of quadratic size, in main memory between the strategy and distance computation steps. In this paper, we improve over [31] as follows. (a) We decrease the upper bound for the number of rows to $\lceil n/3 \rceil$. This allows the strategy to be computed *and* stored in main memory without increasing the overall memory requirements for the distance computation. Thus, we fully solve the memory problem of the strategy computation. (b) We present the AP-TED algorithm to compute TED with the optimal all-path strategy, which subsumes LRH strategies. Our new strategy algorithm runs in $O(n^2)$ time and implements an early deallocation technique to reduce the memory requirements. (c) We introduce the AP-TED$^+$ algorithm with two new single-path functions for small subtrees, which are more efficient in terms of runtime and memory than the single-path functions used in RTED. By considering the new functions in the strategy computation we further improve the decomposition strategy.

Approximations. More efficient approximation algorithms for the tree edit distance have been proposed. Akutsu et al. [2] approximate the tree edit distance with the string edit distance δ_S between the Euler string representation of the input trees. They show that the tree edit distance is between $\delta_S/2$ and $(2h+1)\delta_S$, where h is the maximum tree height. By encoding small subtrees into node labels, they further achieve the lower and upper bounds $\delta_S/6$ and $O(n^{3/4})\delta_S$, respectively. Aratsu et al. [4] compute the string edit distance δ_B between so-called binary tree codes of the input trees and achieve the bounds $\delta_B/2$ and $(h+1)\delta_B+h$. The binary tree code is computed from the binary tree representation by traversing the nodes in postorder and concatenating the labels. Both solutions run in $O(n^2)$ time. Zhang proposes an upper bound, the *constrained* tree edit distance, which runs in $O(n^2)$ time and space [36]. Wang and Zhang [35] improve the space complexity of the constrained tree edit distance to $O(n \log n)$ with a technique similar to our approach for the strategy computation. Guha et al. [18] develop a lower bound algorithm by computing the string edit distance between the preorder and postorder sequences of node labels in $O(n^2)$ time and space. Augsten et al. [6] decompose the trees into *pq*-grams and propose a lower bound algorithm which requires $O(n \log n)$ time and $O(n)$ space. Finis et al. [16] develop the RWS-Diff algorithm (Random Walk Similarity Diff) to compute an approximation of the tree edit distance and an edit mapping in $O(n \log n)$ time.

8. Experiments

In this section we experimentally evaluate AP-TED and AP-TED$^+$ and compare them to RTED [30]. Our empirical evaluation on real-world and synthetic data confirms our analytical results: computing the strategy in AP-TED is as efficient as in RTED, but requires significantly less memory. In particular, the strategy computation requires less memory than the actual tree edit distance computation.

Set-up. All algorithms are implemented as single-thread applications in Java 1.7. We run the experiments on a 4-core Intel i7 3.70 GHz desktop computer with 32 GB RAM.

The datasets. We test the algorithms on both synthetic and real world data. We generate synthetic trees of three different shapes: left branch (LB), zigzag (ZZ) and full binary tree (FB) (Fig. 8). We also generate random trees (random) varying in depth and fanout (with a maximum depth of 15 and a maximum fanout of 6).

We use three real world datasets with different characteristics. SwissProt[3] is an XML protein sequence database with 50 000 medium sized and flat trees (average depth 3.8, maximum depth 4, average fanout 1.8, maximum fanout 346, and average size 187). TreeBank[4] is an XML representation of natural language syntax trees with 56 385 small and deep trees (average depth 10.4, maximum depth 35, average fanout 1.5, maximum fanout 51, and average size 68). TreeFam[5] stores 16 138 phylogenetic trees of animal genes (average depth 14, maximum depth 158, average fanout 2, maximum fanout 3, and average size 95).

Memory measurement. We first measure the memory requirements for the strategy computation and compare it to the memory needed for executing the strategy. We measure only the heap memory allocated by the Java Virtual Machine. The non-heap memory varies between 3.5 MB and 5 MB for all test cases, including the tests on the largest trees.

Fig. 9(a)–(d) shows the memory requirements for different tree shapes and sizes. We use three datasets containing synthetic trees of the same shape (LB, ZZ, and FB) and a dataset with randomly generated trees (random). Each synthetic dataset contains trees with up to 40 000 nodes. The lines represent the memory usage in MB for strategy computation in RTED and AP-TED, and the memory needed for the distance computation. In all cases AP-TED strategy requires significantly less memory than RTED strategy (note the logarithmic scale). RTED uses much more (up to 200%) memory for the strategy computation than for executing the strategy. On the contrary, AP-TED always uses less memory to compute the strategy. In particular, we observe that most of the memory allocated by AP-TED strategy is used to store the strategy.

[3] http://www.expasy.ch/sprot/
[4] http://www.cis.upenn.edu/~treebank/
[5] http://www.treefam.org/

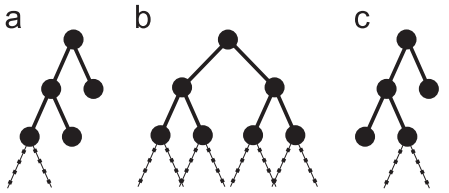


Fig. 8. Shapes of the synthetic trees. (a) Left branch tree (LB), (b) full binary tree (FB) and (c) zig-zag tree (ZZ).

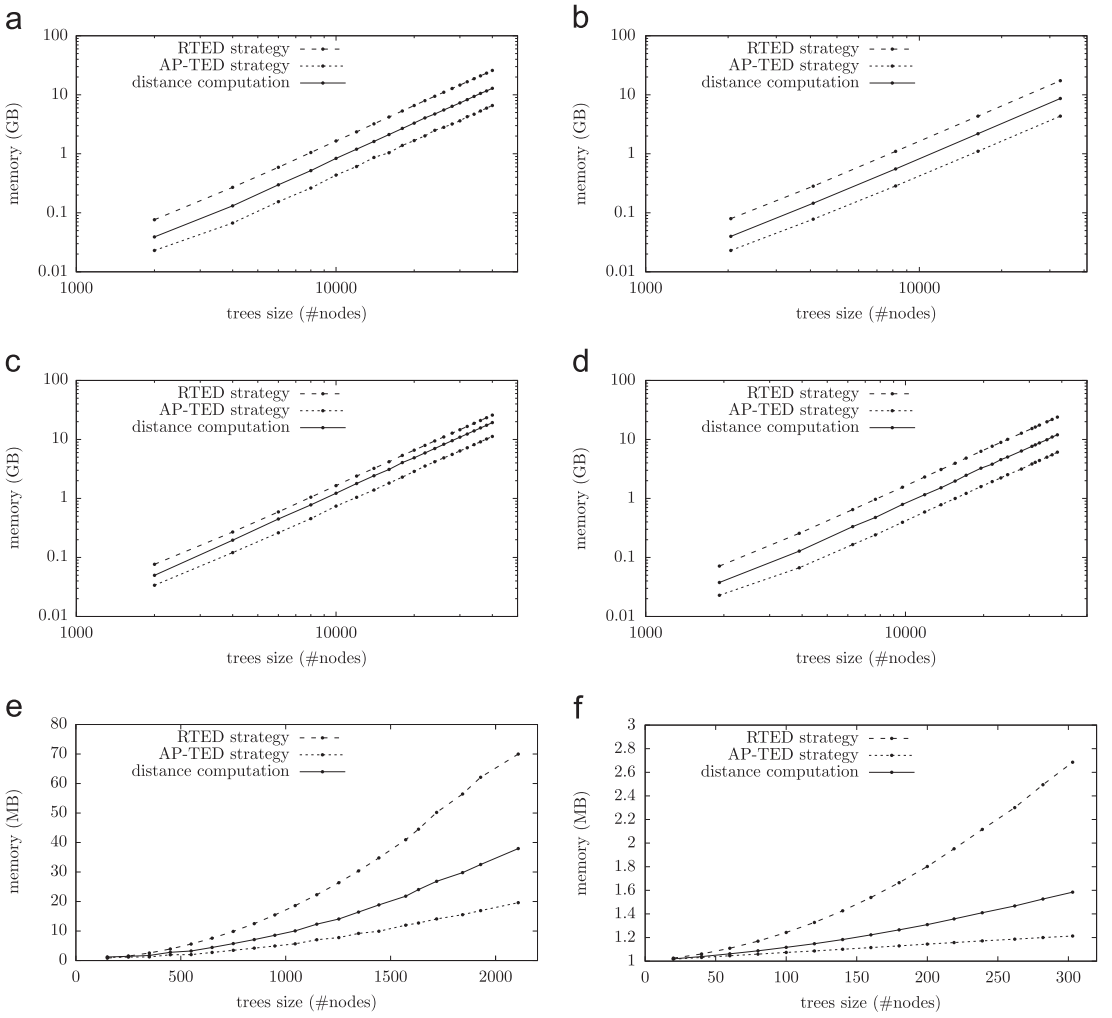


Fig. 9. Memory of strategy computation compared to executing the strategy on synthetic data and real-world datasets (Sprot and TreeBank). (a) Left branch tree (LB), (b) full binary tree (FB), (c) zig-zag tree (ZZ), (d) random tree (random), (e) Sprot, and (f) TreeBank.

We performed a similar experiment on the real-world datasets from Sport and TreeBank (Fig. 9(e) and (f)). We pick pairs of similarly sized trees at regular size intervals. We measure the memory of the strategy computation. The plotted data points correspond to the average size of a tree pair and the memory used in MB. We observe a similar behaviour as in the case of synthetic trees.

We further compute a similarity self join on a sample subset of similar-size trees from the real-world datasets. The join is of the form $D \bowtie_\theta D$, where D is a dataset and θ a tree edit distance predicate. The results are gathered in Table 1. The first three columns show the dataset, the average tree size and the number of trees in the sample. We report the memory for the strategy computation in RTED, in AP-TED, and for the distance computation step. For each of the datasets, the strategy computation with RTED requires more memory than the distance computation, whereas AP-TED always uses less memory.

Number of rows. We count the number of concurrently needed rows in the cost arrays during the strategy computation in AP-TED. We perform an experiment on the real-world datasets. The results are shown in Table 1. For the RTED strategy the

Table 1
Average memory usage (in MB) and runtime (in milliseconds) for different datasets and tree sizes.

Dataset	Avg. size	Trees	RTED strategy			AP-TED strategy			Distance computation	
			rows	Memory	Runtime	rows	Memory	Runtime	Memory	Runtime
treebank-100	98.7	50	98.7	0.57	0.02	5.3	0.53	0.03	0.57	1.43
treebank-200	195.4	50	195.4	1.10	0.07	6.0	0.72	0.09	0.88	6.27
treebank-400	371.3	10	371.3	2.86	2.14	6.3	1.30	2.31	1.77	26.99
sprot-200	211.0	50	211.0	1.21	0.15	3.0	0.70	0.09	0.89	4.92
sprot-400	395.0	50	395.0	3.12	2.25	3.0	1.31	2.16	1.92	15.44
sprot-1000	987.5	20	987.5	16.64	16.95	3.0	5.13	14.94	9.17	123.79
sprot-2000	1960.1	10	1960.1	63.20	64.85	3.0	17.80	55.11	33.10	502.88
treefam-200	197.7	50	197.7	1.16	0.15	9.6	0.74	0.15	0.88	9.86
treefam-400	402.6	50	402.6	3.33	2.52	12.3	1.46	2.37	2.05	55.48
treefam-1000	981.9	20	981.9	16.73	18.33	14.1	5.58	17.33	9.27	453.51

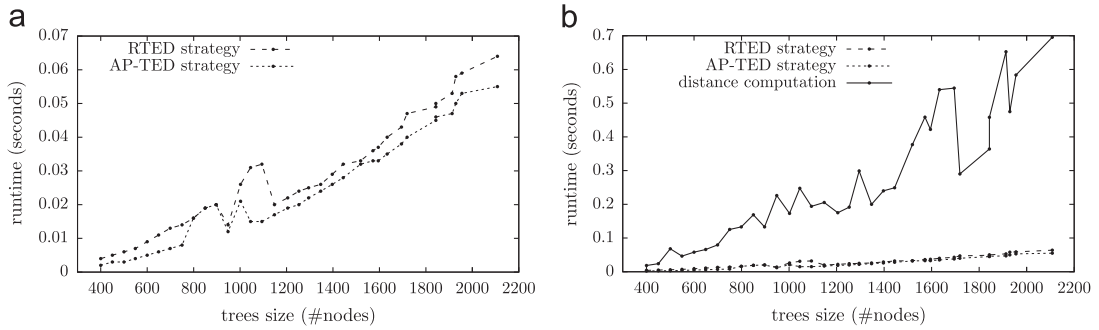


Fig. 10. Runtime difference between RTED strategy, AP-TED strategy, and the overall distance computation. (a) RTED vs. AP-TED strategy and (b) strategy vs. distance computation.

Table 2
Runtime (in seconds) of a similarity self join on different datasets.

Dataset	RTED	AP-TED$^+$
treebank-100	3.87	2.66
treebank-200	17.15	12.42
treebank-400	1.44	1.02
sprot-200	14.24	6.07
sprot-400	45.01	27.56
sprot-1000	56.74	31.54
sprot-2000	59.22	34.04
treefam-200	30.67	25.70
treefam-400	160.44	151.43
treefam-1000	211.00	190.92

number of rows is the size of the left-hand input tree (column 2 – Avg. size). For the AP-TED strategy the number of rows varies from 5% of the tree size for the treebank-100 dataset to 0.15% for the sprot-2000 dataset. Thus, in practice, our memory deallocation technique is much more effective than suggested by the theoretical bound of 33.3%.

Strategy runtime. We measure the time for computing and executing strategies; the results for different datasets are shown in the runtime columns of Table 1. In most cases, AP-TED strategy is slightly faster than RTED strategy. Due to its deallocation mechanism, AP-TED strategy (cf. Section 5) needs to allocate less memory than RTED strategy. The improved runtime performance of AP-TED strategy stems from reusing memory as compared to allocating new memory on the heap. Compared to the distance computation, the strategy requires only a small fraction of the overall time. Fig. 10 shows how the strategy computation scales with the tree size for the Sprot dataset.

Efficiency of the new single-path functions. We measure the efficiency of our new single-path functions, Δ^{1node} and Δ^{2node}. We perform a similarity self join on our real-world datasets (as in the memory experiment). We measure the runtime of the entire join on each dataset separately. The results are shown in Table 2. The second column shows the results for RTED, the third column for our AP-TED$^+$ algorithm. AP-TED$^+$ reduces the runtime from 5% for the treefam-400 dataset to 57% for the sprot-200 dataset.

9. Conclusion

In this paper we develop two new algorithms for the tree edit distance: AP-TED and AP-TED$^+$. The strategy computation is a main memory bottleneck of the state-of-the-art solution, RTED [30]. The memory required for the strategy computation can be twice the memory needed for the actual tree edit distance computation. Our AP-TED strategy algorithm reduces the memory by at least 2/3 compared to the strategy computation in RTED and never uses more memory than the distance computation. The experimental evaluation confirms our theoretical results. Our algorithm computes the optimal all-path strategy which, compared to RTED, does not limit the path type and considers all possible paths. Our AP-TED$^+$ algorithm with new efficient single-path functions reduces the overall runtime of the tree edit distance by up to 57%.

Acknowledgements

This work is partially supported by the SyRA project of the Free University of Bozen-Bolzano, Italy.

References

[1] T. Akutsu, Tree edit distance problems algorithms and applications to bioinformatics, IEICE Trans. Inf. Syst. E 93-D (2) (2010) 208–218.
[2] T. Akutsu, D. Fukagawa, A. Takasu, Approximating tree edit distance through string edit distance, Algorithmica 57 (2) (2010) 325–348.
[3] K.F. Aoki, A. Yamaguchi, Y. Okuno, T. Akutsu, N. Ueda, M. Kanehisa, H. Mamitsuka, Efficient tree-matching methods for accurate carbohydrate database queries, Genome Inform. 14 (2003) 134–143.
[4] T. Aratsu, K. Hirata, T. Kuboyama, Approximating tree edit distance through string edit distance for binary tree codes, Fundam. Inform. 101 (3) (2010) 157–171.
[5] N. Augsten, D. Barbosa, M. Böhlen, T. Palpanas, Efficient top-k approximate subtree matching in small memory, IEEE Trans. Knowl. Data Eng. (TKDE) 23 (8) (2011) 1123–1137.
[6] N. Augsten, M. Böhlen, J. Gamper, The pq-gram distance between ordered labeled trees, ACM Trans. Database Syst. (TODS) 35 (1) .
[7] J. Bellando, R. Kothari, Region-based modeling and tree edit distance as a basis for gesture recognition, in: International Conference on Image Analysis and Processing (ICIAP), 1999.
[8] S.S. Chawathe, Comparing hierarchical data in external memory, in: International Conference on Very Large Data Bases (VLDB), 1999.
[9] G. Cobéna, S. Abiteboul, A. Marian, Detecting changes in xml documents, in: International Conference on Data Engineering (ICDE), 2002.
[10] S. Cohen, Indexing for subtree similarity-search using edit distance, in: ACM SIGMOD International Conference on Management of Data (SIGMOD), 2013, pp. 49–60.
[11] T. Dalamagas, T. Cheng, K.-J. Winkel, T. Sellis, A methodology for clustering xml documents by structure, Inf. Syst. 31 (3) (2006) 187–228.
[12] D. de Castro Reis, P.B. Golgher, A.S. da Silva, A.H.F. Laender, Automatic web news extraction using tree edit distance, in: International World Wide Web Conference (WWW), 2004, pp. 502–511.
[13] E.D. Demaine, S. Mozes, B. Rossman, O. Weimann, An optimal decomposition algorithm for tree edit distance, ACM Trans. Algorithms 6 (1) .
[14] S. Dulucq, H. Touzet, Decomposition algorithms for the tree edit distance problem, J. Discret. Algorithms 3 (2–4) (2005) 448–471.
[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Montperrus, Fine-grained and accurate source code differencing, in: ACM/IEEE International Conference on Automated Software Engineering (ASE), 2014, pp. 313–324.
[16] J.P. Finis, M. Raiber, N. Augsten, R. Brunel, A. Kemper, F. Färber, Rws-diff: flexible and efficient change detection in hierarchical data, in: ACM International Conference on Information and Knowledge Management (CIKM), 2013, pp. 339–348.
[17] M. Garofalakis, A. Kumar, Xml stream processing using tree-edit distance embeddings, ACM Trans. Database Syst. (TODS) 30 (1) (2005) 279–332.
[18] S. Guha, H.V. Jagadish, N. Koudas, D.S.T. Yu, Approximate xml joins, in: ACM SIGMOD International Conference on Management of Data (SIGMOD), 2002.
[19] A. Habrard, J.M.I. Quereda, D. Rizo, M. Sebban, Melody recognition with learned edit distances, in: Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition (SSPR/SPR), 2008, pp. 86–96.
[20] M. Hashimoto, A. Mori, Diff/ts: a tool for fine-grained structural change analysis, in: Working Conference on Reverse Engineering (WCRE), 2008.
[21] H. Heumann, G. Wittum, The tree-edit-distance, a measure for quantifying neuronal morphology, Neuroinformatics 7 (3) (2009) 179–190.
[22] S. Kamali, F.W. Tompa, Retrieving documents with mathematical content, in: ACM SIGIR International Conference on Research and Development in Information Retrieval 2013, pp. 353–362.
[23] Y. Kim, J. Park, T. Kim, J. Choi, Web information extraction by HTML tree edit distance matching, in: International Conference on Convergence Information Technology (ICCIT), 2007, pp. 2455–2460.
[24] P.N. Klein, Computing the edit distance between unrooted ordered trees, in: European Symposium on Algorithms (ESA), 1998, pp. 91–102.
[25] P.N. Klein, S. Tirthapura, D. Sharvit, B.B. Kimia, A tree-edit-distance algorithm for comparing simple, closed shapes, in: ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000.
[26] F. Korn, B. Saha, D. Srivastava, S. Ying, On repairing structural problems in semi-structured data, Proc. VLDB Endow. 6 (9) .
[27] K.-H. Lee, Y.-C. Choy, S.-B. Cho, An efficient algorithm to compute differences between structured documents, IEEE Trans. Knowl. Data Eng. (TKDE) 16 (8) (2004) 965–979.
[28] Z. Lin, H. Wang, S.I. McClean, Measuring tree similarity for natural language processing based information retrieval, in: International Conference on Applications of Natural Language to Information Systems (NLDB), 2010.
[29] B. Ma, L. Wang, K. Zhang, Computing similarity between RNA structures, Theor. Comput. Sci. 276 (1–2) (2002) 111–132.
[30] M. Pawlik, N. Augsten, RTED: a robust algorithm for the tree edit distance, Proc. VLDB Endow. (PVLDB) 5 (4) (2011) 334–345.
[31] M. Pawlik, N. Augsten, A memory-efficient tree edit distance algorithm, in: International Conference on Database and Expert Systems Applications (DEXA), 2014.
[32] M. Pawlik, N. Augsten, Efficient computation of the tree edit distance, ACM Trans. Database Syst. (TODS) 40 (1) . (Article 3).
[33] V. Springel, S.D.M. White, A. Jenkins, C.S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J.A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, F. Pearce, Simulations of the formation, evolution and clustering of galaxies and quasars, Nature 435 (2005) 629–636.
[34] H.-C. Tai, The tree-to-tree correction problem, J. ACM (JACM) 26 (3) (1979) 422–433.
[35] L. Wang, K. Zhang, Space efficient algorithms for ordered tree comparison, Algorithmica 51 (3) (2008) 283–297.
[36] K. Zhang, Algorithms for the constrained editing distance between ordered labeled trees and related problems, Pattern Recognit. 28 (3) (1995) 463–474.
[37] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM J. Comput. (SICOMP) 18 (6) (1989) 1245–1262.