

A Memory-Efficient Tree Edit Distance Algorithm

Mateusz Pawlik and Nikolaus Augsten

University of Salzburg, Austria

{mateusz.pawlik,nikolaus.augsten}@sbg.ac.at

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 for the tree edit distance is the RTED algorithm, which is guaranteed to be fast independently of the tree shape. Unfortunately, this algorithm uses twice the memory of the other, slower algorithms. 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. Our algorithm runs at least as fast as RTED, but requires only half the memory. This is achieved by systematically releasing memory early during the first step of the algorithm, which computes a decomposition strategy and is the main memory bottleneck. We show the correctness of our approach and prove an upper bound for the memory usage. Our empirical evaluation confirms the low memory requirements and shows that in practice our algorithm performs better than the analytic guarantees suggest.

1 Introduction

Data with hierarchical dependencies are often modeled as trees. Tree data appear in many applications, ranging from hierarchical data formats like JSON or XML to merger trees in astrophysics [20]. 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 transforms one tree into another. The tree edit distance has been successfully applied in many applications [1, 2, 14, 18], and has received considerable attention from the database community [3, 5–8, 11–13, 16, 17].

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 the different TED algorithms is the decomposition strategy, which has a major impact on the runtime. Early attempts to compute TED [9, 15, 23] 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 [19], which operates in two steps (cf. Figure 1(a)). In the first step, a decomposition strategy is computed. The strategy adapts to the input trees and is shown to be optimal in the class of LRH strategies, which contains all previously proposed strategies. The actual distance computation is done in the second step, which executes the strategy.

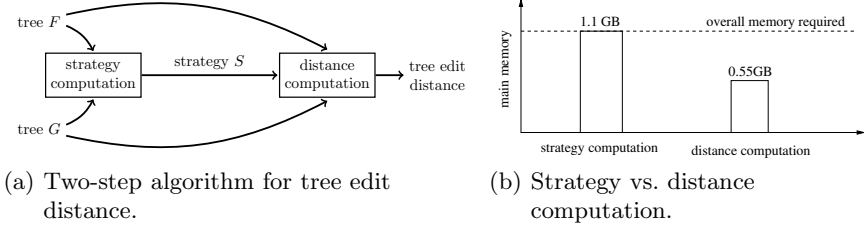


Fig. 1. Strategy computation requires more memory than actual distance computation

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. Figure 1(b) shows the memory usage for two example trees (perfect binary trees) of 8191 nodes: the strategy computation requires 1.1GB of RAM, while the execution of the strategy (i.e., the actual distance computation) requires only 0.55GB. 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.

This paper solves the memory problem of the strategy computation. We present MemoryOptStrategy, a new algorithm for computing an optimal decomposition strategy. We achieve this by computing the strategy bottom-up using dynamic programming and releasing part of the memoization tables early. We prove an upper bound for the memory requirements which is 25% below the best case for the distance computation. Thus, the strategy computation is no longer the main memory bottleneck. In our extensive experimental evaluation on various tree shapes, which require very different strategies, the memory for the strategy computation never exceeds the memory for the distance computation. With respect to the strategy algorithm proposed for RTED [19], we reduce the memory by at least 50%. For some tree shapes our algorithm even runs in linear space, while the RTED strategy algorithm always requires quadratic space. Summarizing, the contributions of this paper are the following:

- *Memory efficiency.* We substantially reduce the memory requirements w.r.t. previous strategy computation algorithms. We develop a new MemOptStrategy algorithm which computes the strategy by traversing the trees bottom-up and systematically releasing rows early. We show the correctness of our approach and prove an upper bound for the memory usage.

- *Shape and size heuristics.* Our MemOptStrategy algorithm requires at most 50% of the memory that the RTED strategy algorithm needs. We observe that the worst case happens for very specific tree shapes only. We devise heuristics which deal with a number of frequent worst-case instances and thus make our solution even more effective in practice.

The paper is structured as follows. Section 2 sets the stage for the discussion of strategy algorithms. In Section 3 we define the problem. Our memory-efficient strategy computation algorithm, MemoryOptStrategy is discussed in Section 4. We treat related work in Section 5, experimentally evaluate our solution in Section 6, and conclude in Section 7.

2 Background

Notation. We follow the notation of [19] 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. 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*. Children of the same node are *siblings*. 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 siblings f, g , if $f < g$, then $f' < g'$ for all descendants f' of f , and $f < g'$ for all descendants g' of g . The tree traversal that visits all nodes in ascending order is the *postorder* traversal.

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

We use the following short notation: $|F| = |N(F)|$ is the size of tree F , we write $v \in F$ for $v \in N(F)$.

Example 1. The nodes of tree F in Figure 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 $r(F) = v_{13}$, 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 postorder traversal visits the nodes of F in the following order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}$.

Strategy Computation. 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, rename the label of a node. The state-of-the-art algorithms compute the tree edit distance by implementing a well-known recursive solution [21] using dynamic programming [9, 19, 21, 23].

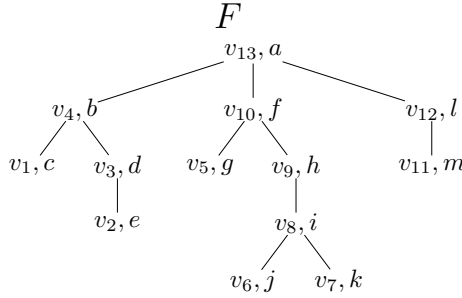


Fig. 2. Example tree

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 bigger problems. Pawlik and Augsten [19] introduce *path strategies* as a subset of all possible decomposition strategies. Together with the *general tree edit distance* algorithm, path strategies generalize all previous approaches. We revisit the concept of path strategies and the computation of the optimal strategy in RTED.

In the first step of RTED (cf. Figure 1(a)) 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 ¹. 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.

The RTEDStrategy algorithm [19] (Algorithm 1) has the following outline. The strategy is computed in a dynamic programming fashion. The algorithm uses six memoization arrays, called *cost arrays*: L_v, R_v, H_v of size $|F||G|$, and L_w, R_w, H_w of size $|G|$. Each cell in a cost array stores the cost of computing the distance for a specific subtree pair and a specific path type. The names of the cost arrays indicate the path type, which may be left (L), right (R), or heavy (H) in the class of LRH strategies considered in [19]. All state-of-the-art strategies fall into the LRH class. The algorithm computes the final result in a bottom-up manner starting with the leaf nodes. The core of the algorithm are two nested loops (lines 3 and 4). They iterate over the nodes of the input trees in postorder, which ensures that a child node is processed before its parent. Inside the innermost loop, first the minimal cost of computing the distance for the particular subtree pair (F_v, G_w) is calculated based on the previously computed distances for the children of nodes v and w (line 7). Second, the path corresponding to that cost is sent to the output (line 8). Finally, the distances for the parent nodes $p(v)$ and $p(w)$ are updated (lines 9 and 10).

¹ RTED considers LRH strategies which allow left, right, and heavy paths. For example, the path in F from v_{13} to v_7 in Figure 2 is a heavy path, i.e., a parent is always connected to its child that roots the largest subtree (the rightmost child in case of ties). The path from v_{13} to v_1 is left, and the path from v_{13} to v_{11} is right.

Since all the operations inside the innermost loop are done in constant time, the time complexity of the algorithm is $O(|F||G|)$. The space complexity is given by the three cost arrays of the size $|F||G|$ and is $O(|F||G|)$.

In this paper we show that the strategy computation in RTED may require twice the memory of the actual distance computation step. We remove that bottleneck by reducing the size of the quadratic cost arrays L_v , R_v , and H_v . We show that our new MemOptStrategy algorithm uses significantly less memory than RT-EDStrategy, and never uses more memory than the distance computation.

Algorithm 1. RTEDStrategy(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  $v \in F$  in postorder do
4   for  $w \in G$  in postorder do
5     if  $v$  is leaf then  $L_v[v, w] \leftarrow R_v[v, w] \leftarrow H_v[v, w] \leftarrow 0$ 
6     if  $w$  is leaf then  $L_w[w] \leftarrow R_w[w] \leftarrow H_w[w] \leftarrow 0$ 
7     compute the minimal cost of distance computation for  $(F_v, G_w)$ 
8     output the path corresponding to the minimal cost for  $(F_v, G_w)$ 
9     if  $v$  is not root then update values for  $L_v[p(v), w], R_v[p(v), w], H_v[p(v), w]$ 
10    if  $w$  is not root then update values for  $L_w[p(w)], R_w[p(w)], H_w[p(w)]$ 
```

3 Problem Definition

As outlined above, the path strategy introduced by Pawlik and Augsten [19] generalizes all state-of-the-art algorithms for computing the tree edit distance. The RTED algorithm picks the optimal strategy. Unfortunately, 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.

In this paper we analyse the memory requirements of strategy and distance computation in the RTED algorithm. We devise a new technique to significantly reduce the memory usage of the strategy computation. We develop the new MemOptStrategy algorithm, which reduces the required memory by at least 50% and thus never requires more memory than the distance computation.

4 Memory Efficient Tree Edit Distance

The main memory requirement is a bottleneck in the tree edit distance computation. The strategy computation in RTED exceeds the memory needed for executing the strategy. Our MemOptStrategy algorithm (Algorithm 2) reduces

the memory usage by at least 50% and in practice never uses more memory than it is required for executing the strategy. We achieve that by decreasing the maximal size of the data structures used for strategy computation.

4.1 Memory Analysis in RTED

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

We analyse the memory usage of RTED. The strategy computation is dominated by three cost arrays L_v, R_v, H_v which take $3|F||G|$ space in total. Each time a strategy path is computed, it is send to the output. However, due to the fact that the entire strategy is required in the decomposition step, it is materialized in a quadratic array of the size $|F||G|$ [19]. Thus, the strategy computation requires $4|F||G|$ memory.

The memory of distance computation depends on the strategy. The tree edit distance is computed by so called *single-path functions* which process a single path from the strategy. Different path types require 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 $2|F||G|$ of space is needed in total.
- b) If a heavy path is involved in the strategy, one array of size $|F||G|$ and one array of size $|G|^2$ are used. Thus the distance computation needs $2|F||G| + |G|^2$ space in total.

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

The goal is to reduce the memory requirements of the strategy computation algorithm to meet the minimum requirements of the distance computation. We observe that some of the rows in the cost arrays are not needed any more after they are read. We develop an early deallocation technique to minimize the size of the cost arrays.

4.2 MemOptStrategy Algorithm

The strategy algorithm in RTED uses three cost arrays of quadratic size (L_v, R_v, H_v), where each row corresponds to a node $v \in F$. The outermost loop (line 3) 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 is 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 4).

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

- $\text{read}(v)$ / $\text{read}(\text{leafRow})$ - read the row of node v / read *leafRow*,
- $\text{allocate}(v)$ - allocate a row of node v ,
- $\text{update}(v)$ - update values in the row of node v ,
- $\text{deallocate}(v)$ - deallocate the row of node v .

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

We present the new MemOptStrategy algorithm, Algorithm 2, which dynamically allocates and deallocates rows in the cost arrays. The lines with the grey background implement our technique. At each iteration step, i.e., for each node $v \in F$ in postorder, MemOptStrategy performs the following steps on a cost array:

```

if  $v$  is not root of  $F$ 
  if row for  $p(v)$  does not exist:  $\text{allocate}(p(v))$  (line 5)
   $\text{read}(v)$  (line 8)
   $\text{update}(p(v))$  (line 10)
  if  $v$  is not a leaf:  $\text{deallocate}(v)$  (line 12)

```

Algorithm 2. MemOptStrategy(F, G)

```

1  $L_v, R_v, H_v$  : arrays of size  $|F|$ 
2  $L_w, R_w, H_w, \text{leafRow}$  : arrays of size  $|G|$ 
3 for  $v \in F$  in postorder do
4   if  $v$  is leaf then  $L_v[v] \leftarrow R_v[v] \leftarrow H_v[v] \leftarrow \text{leafRow}$ 
5   if row for  $p(v)$  is not allocated then allocate a row for  $p(v)$  in  $L_v, R_v, H_v$ 
6   for  $w \in G$  in postorder do
7     if  $w$  is leaf then  $L_w[w] \leftarrow R_w[w] \leftarrow H_w[w] \leftarrow 0$ 
8     compute the minimal cost of distance computation for  $(F_v, G_w)$ 
9     output the path corresponding to the minimal cost for  $(F_v, G_w)$ 
10    if  $v$  is not root then update values for  $L_v[p(v), w], R_v[p(v), w], H_v[p(v), w]$ 
11    if  $w$  is not root then update values for  $L_w[p(w)], R_w[p(w)], H_w[p(w)]$ 
12  if  $v$  is not leaf then deallocate row of  $v$  in  $L_v, R_v, H_v$ 

```

Depending on the position of a node in the tree (root, leaf, leftmost child) a specific sequence of operations must be performed. Figure 3 shows the possible sequences and assigns them to the respective node types.

$S0 = \langle \rangle$		
$S1 = \langle \text{allocate}(p(v)), \text{read}(\text{leafRow}), \text{update}(p(v)) \rangle$		
$S2 = \langle \text{allocate}(p(v)), \text{read}(v), \text{update}(p(v)), \text{deallocate}(v) \rangle$	leftmost child	S1 S2
$S3 = \langle \text{read}(v), \text{update}(p(v)), \text{deallocate}(v) \rangle$	not leftmost child	S4 S3
$S4 = \langle \text{read}(\text{leafRow}), \text{update}(p(v)) \rangle$	root	S0

Fig. 3. Rows operations depending on the node position

Example 2. We study the number of concurrently needed rows for the example tree in Figure 4. 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 position. 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. The nodes trigger the following operation sequences: $v_1 - S1, v_2 - S1, v_3 - S3, v_4 - S2, v_5 - S4, v_6 - S0$.

The cost arrays in RTEDStrategy for the example tree have six rows (one for each node). From the table in Figure 4 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.

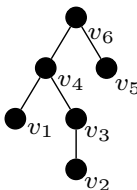
	node operations	rows in memory
	v_1 allocate(v_4), read(<i>leafRow</i>), update(v_4)	<i>leafRow</i> , v_4
	v_2 allocate(v_3), read(<i>leafRow</i>), update(v_3)	<i>leafRow</i> , v_3 , v_4
	v_3 read(v_3), update(v_4), deallocate(v_3)	<i>leafRow</i> , v_4
	v_4 allocate(v_6), read(v_4), update(v_6), deallocate(v_4)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

Fig. 4. Rows operations on an example tree

We now consider the general case and count the number of rows that must be kept in memory concurrently. 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 concurrent rows during the postorder traversal of tree F with the following formula.

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

The *leafRow* is always kept in memory. In addition, each node contributes with $d(S)$ concurrently needed rows, where S is the operation sequence required by

the node. For the tree in Figure 4 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 node that falls into this category 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 at most $\lfloor \frac{|F|}{2} \rfloor$.*

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 $\lfloor \frac{|F|}{2} \rfloor$: Each such node, in order to be a leftmost-child leaf node, must have a different parent. An example of a tree with $\lfloor \frac{|F|}{2} \rfloor$ leftmost-child leaf nodes is the right branch tree (the right branch tree is a symmetric tree to the left branch tree in Figure 5(a)). This results in $\lfloor \frac{|F|}{2} \rfloor$ different parent nodes for $\lfloor \frac{|F|}{2} \rfloor$ leftmost-child leaf nodes. If $|F|$ is odd, then $\lfloor \frac{|F|}{2} \rfloor + \lfloor \frac{|F|}{2} \rfloor = |F| - 1$ and there is one node that we did not count. This node can become a leftmost-child leaf node of either a node which is a leftmost-child leaf or of a node that has a leftmost-child leaf. Then, the old leftmost-child leaf becomes the parent or a right sibling of the new leftmost-child leaf, respectively. If $|F|$ is even, $\lfloor \frac{|F|}{2} \rfloor + \lfloor \frac{|F|}{2} \rfloor = |F|$. Thus, the maximum number of the leftmost-child leaf nodes is $\lfloor \frac{|F|}{2} \rfloor$.

Theorem 1. *The memory required for the strategy computation in MemOptStrategy is $1.5|F||G|$.*

Proof. By expiring rows the sizes of the cost arrays L_v , R_v , and H_v are reduced with Lemma 1 from $|F||G|$ to at most $0.5|F||G|$. Thus, the MemOptStrategy algorithm requires $1.5|F||G|$ memory in the worst case.

4.3 Shape and Size Heuristics

We further devise two heuristic techniques that are easy to apply yet effective in reducing the memory.

Shape. The MemOptStrategy algorithm iterates over the nodes in postorder. One can think of a symmetric algorithm which iterates over the nodes in right-to-left postorder². 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 two rows. Thus, the direction of the tree traversal matters. Let $\#l(F)$ and $\#r(F)$ be the number

² In right-to-left postorder, children are traversed from right to left and before their parents. For example, traversing nodes of the tree in Figure 4 gives the following sequence: $v_5, v_2, v_3, v_1, v_4, v_6$.

of leftmost-child and rightmost-child leaf nodes in F , respectively. The *shape heuristic* says: If $\#l(F) < \#r(F)$, then use postorder, otherwise use right-to-left postorder to compute the strategy.

Size. The length of the rows in the cost array is the size of the right-hand input tree, thus the array size can be reduced by switching the input parameters. The *size heuristic* is as follows: If $|G| \cdot \min\{\#l(F), \#r(F)\} < |F| \cdot \min\{\#l(G), \#r(G)\}$, then the number of rows should depend on F and the length of the rows on G .

5 Related Work

Tree Edit Distance Algorithms. The fastest algorithms for the tree edit distance are dynamic programming implementations of a recursive solution which decomposes the input trees to smaller subtrees and subforests. The runtime complexity is given by the number of subproblems that must be solved. Tai [21] proposes an algorithm that runs in $O(n^6)$ time and space, where n is the number of nodes. Zhang and Shasha [23] improve the complexity to $O(n^4)$ time and $O(n^2)$ space. Klein [15] achieves $O(n^3 \log n)$ runtime and space. Demaine et al. [9] develop an algorithm which requires $O(n^3)$ time and $O(n^2)$ space and show that their solution is worst-case optimal. The most recent development is the RTED algorithm by Pawlik and Augsten [19]. They observe that each of the previous algorithms is efficient only for specific tree shapes and runs into its worst case otherwise. They point out that the runtime difference due to a poor algorithm choice may be of a polynomial degree. They devise the general framework for computing the tree edit distance shown in Figure 1(a).

Path Strategies. Each of the tree edit distance 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 [23], Klein [15], and Demaine et al. [9] use hard-coded strategies which do not require a strategy computation step, but the resulting algorithms are efficient only for specific tree shapes. Dulucq and Touzet [10] compute a decomposition strategy in the first step, then use the strategy to compute the tree edit distance. However, they only consider strategies that decompose a single tree, and the overall algorithm runs in $O(n^3 \log n)$ time and space. More efficient strategies that decompose both trees were introduced by Pawlik and Augsten [19]. Their entire solution requires $O(n^3)$ time and $O(n^2)$ space. The resulting strategy is shown to be optimal in the class of LRH strategies, which cover all previous solutions. However, the strategy computation may need twice as much memory as the distance computation. Our memory-efficient strategy computation algorithm, MemOptStrategy, improves over the strategy computation in RTED [19] and reduces the memory by at least 50%. We never need more memory for the strategy computation than for the tree edit distance computation.

Approximations. More efficient approximation algorithms for the tree edit distance have been proposed. Zhang [22] proposes an upper bound, *constrained* tree edit distance, which is solved in $O(n^2)$ time and space. Guha et al. [13] 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. [4] 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. [11] 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.

6 Experiments

In this section we experimentally evaluate our MemOptStrategy algorithm and compare it to RTED [19]. The experiments on real-world and synthetic data confirm our analytical results. We show that computing the strategy with MemOptStrategy is as efficient as in RTED and requires significantly less memory. The memory requirements of MemOptStrategy are below the memory used to execute the strategy and compute the actual tree edit distance.

Set-up. All algorithms were implemented as single-thread applications in Java 1.7. We run the experiments on a 4-core Intel i7 3.70GHz desktop computer with 8GB RAM memory. We measure main memory and runtime for each of the two steps in the process: the strategy and the distance computation (cf. Figure 1(b)). The output of the strategy computation is a strategy which can be stored in an array of size $|F||G|$ for a pair of trees, F and G . Our memory measurements for the two strategy algorithms also include the space required to store the computed strategy, which we materialize in main memory due to efficiency reasons.

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 (FB) (Figure 5). We also generate random trees (Random) varying in depth and fanout (with a maximum depth of 15 and a maximum fanout of 6).

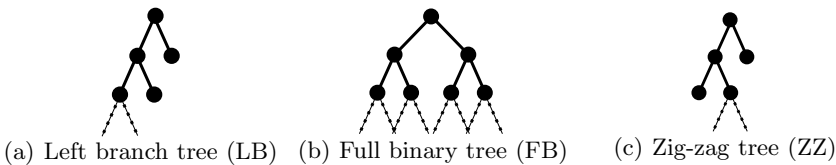


Fig. 5. Shapes of the synthetic trees

We use three real world datasets with different characteristics. SwissProt³ is an XML protein sequence database with 50000 medium sized and flat trees (average depth 3.8, maximum depth 4, average fanout 1.8, maximum fanout 346, average size 187). TreeBank⁴ is an XML representation of natural language syntax trees with 56385 small and deep trees (average depth 10.4, maximum

³ <http://www.expasy.ch/sprot/>

⁴ <http://www.cis.upenn.edu/~treebank/>

depth 35, average fanout 1.5, maximum fanout 51, average size 68). TreeFam⁵ stores 16138 phylogenetic trees of animal genes (average depth 14, maximum depth 158, average fanout 2, maximum fanout 3, average size 95).

Memory measurement. We first measure the memory requirements for 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 is independent of the tree size and varies between 3.5MB and 5MB in all our tests.

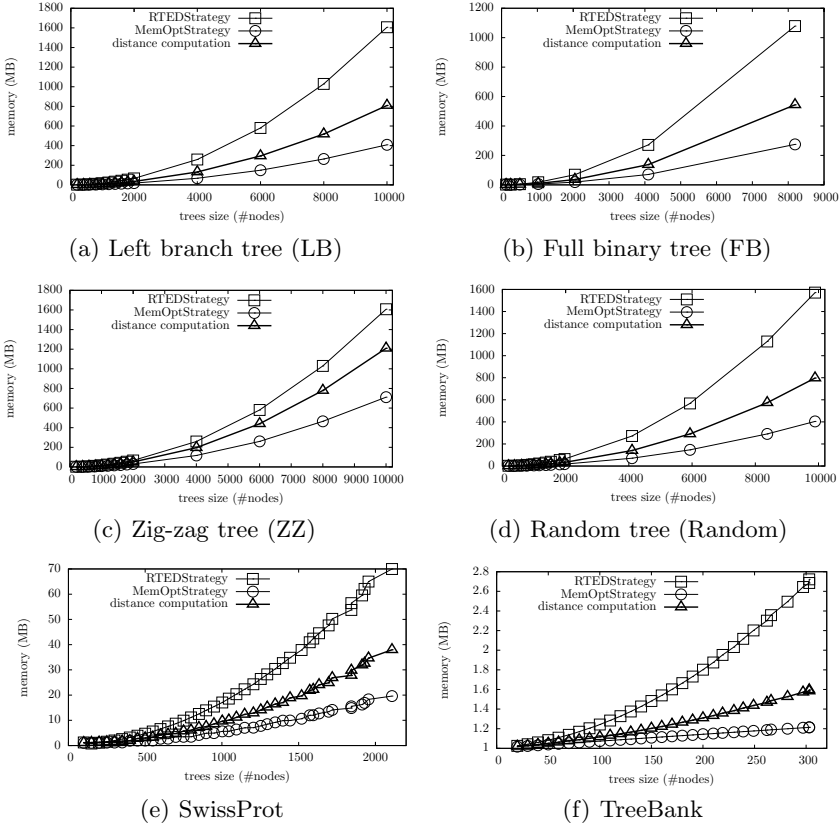


Fig. 6. Memory of strategy computation compared to executing the strategy on synthetic and real-world datasets

Figures 6(a)-6(d) show the memory requirements for different tree shapes and sizes. We use three datasets containing synthetic trees of a specific shape (LB, ZZ, FB) and a dataset with randomly generated trees (Random). Each synthetic dataset contains trees with up to 10000 nodes. The graphs show the memory

⁵ <http://www.treefam.org/>

usage in MB of (a) the RTEDStrategy algorithm, (b) our MemOptStrategy algorithm, and (c) the distance computation step. In all tested scenarios MemOptStrategy requires significantly less memory than RTED. The RTEDStrategy algorithm uses much more (up to 200%) memory than is needed for executing the strategy thus forming a main memory bottleneck. On the contrary, MemOptStrategy always uses less memory than the distance computation. In particular, we observe that most of the memory allocated by MemOptStrategy is used to materialize the output (i.e., the computed strategy) in main memory.

We performed a similar experiment on the real-world data from SwissProt and TreeBank (Figure 6(e) and 6(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 show the used memory in MB. The results for the real-world datasets confirm our findings on synthetic data: MemOptStrategy substantially reduces the memory requirements of the strategy computation and breaks the main memory bottleneck of the RTED strategy computation.

We further compute the pairwise tree edit distance on sampled subsets of similarly-sized trees from the real-world datasets. The results are gathered in Table 1. The first three columns show the name of the dataset, the average tree size, and the number of trees in the sample, respectively. We report the average memory for the strategy computation in RTED, MemOptStrategy, and the distance computation step. For all the datasets, the strategy computation with RTED requires more memory than the distance computation; MemOptStrategy always uses less memory. We reduce the memory usage with respect to RTEDStrategy in all of the test cases: from 7% for TreeBank-100 to 71% for SwissProt-2000, and we achieve a reduction of 51% on average.

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

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

Number of rows. We perform an experiment on our real-world datasets and count the average number of concurrently needed rows in the cost arrays during

the strategy computation with MemOptStrategy. This shows the memory reduction on the cost arrays only and does not consider the materialized strategy. The results are shown in column 6 of Table 1. For RTEDStrategy the number of rows is equal to the number of nodes of the left-hand input tree (column 2 - Avg. size). For MemOptStrategy the number of rows varies from 5% of the tree size for TreeBank-100 to 0.15% for SwissProt-2000, and we obtain a reduction of 97.2% in average. This confirms our analytical results and shows the effectiveness of our approach.

Runtime. We measure the average runtime of computing and executing the strategies. The results for different datasets are shown in the runtime columns of Table 1. They show that MemOptStrategy in most cases is slightly faster than RTEDStrategy. This is due to low memory allocation. Compared to the distance computation, the strategy requires only a small fraction of the overall time. Figure 7 shows how the strategy computation scales with the tree size for the SwissProt dataset. Compared to the distance computation, the strategy computation requires 6% of the overall time on average.

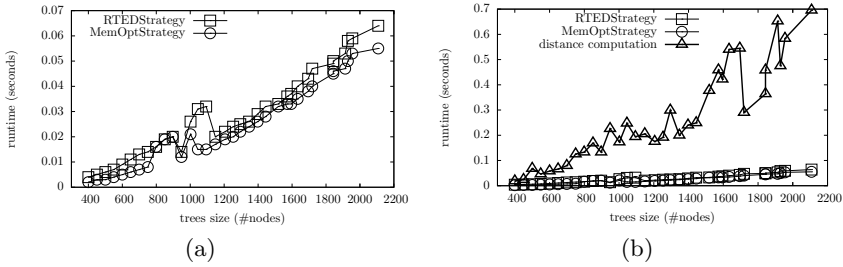


Fig. 7. Runtime difference between RTEDStrategy and MemOptStrategy algorithms (a) compared to the overall time of the distance computation (b)

7 Conclusion

In this paper we developed a new strategy computation algorithm for the tree edit distance. The strategy computation is a main memory bottleneck of the state-of-the-art solution, RTED [19]. The memory required for the strategy computation can be twice the memory needed for the actual tree edit distance computation, thus forming a bottleneck. Our MemOptStrategy algorithm reduces the memory requirements by at least 50% compared to RTED and never uses more memory than the distance computation. Our extensive empirical evaluation on synthetic and real world datasets confirmed our theoretical results.

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

References

1. Akutsu, T.: Tree edit distance problems: Algorithms and applications to bioinformatics. *IEICE Trans. on Inf. Syst.* 93-D(2), 208–218 (2010)
2. Aoki, K.F., Yamaguchi, A., Okuno, Y., Akutsu, T., Ueda, N., Kanehisa, M., Mamitsuka, H.: Efficient tree-matching methods for accurate carbohydrate database queries. *Genome Informatics* 14, 134–143 (2003)
3. Augsten, N., Barbosa, D., Böhlen, M., Palpanas, T.: Efficient top- k approximate subtree matching in small memory. *IEEE TKDE* 23(8), 1123–1137 (2011)
4. Augsten, N., Böhlen, M.H., Gamper, J.: The pq -gram distance between ordered labeled trees. *ACM TODS* 35(1) (2010)
5. Chawathe, S.S.: Comparing hierarchical data in external memory. In: *VLDB*, pp. 90–101 (1999)
6. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: *ICDE*, pp. 41–52 (2002)
7. Cohen, S.: Indexing for subtree similarity-search using edit distance. In: *SIGMOD*, pp. 49–60 (2013)
8. Dalamagas, T., Cheng, T., Winkel, K.-J., Sellis, T.K.: A methodology for clustering XML documents by structure. *Inf. Syst.* 31(3), 187–228 (2006)
9. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. *ACM Trans. on Alg.* 6(1) (2009)
10. Dulucq, S., Touzet, H.: Decomposition algorithms for the tree edit distance problem. *J. Discrete Alg.* 3(2-4), 448–471 (2005)
11. Finis, J.P., Raiber, M., Augsten, N., Brunel, R., Kemper, A., Färber, F.: RWS-Diff: Flexible and efficient change detection in hierarchical data. In: *CIKM*, pp. 339–348 (2013)
12. Garofalakis, M., Kumar, A.: XML stream processing using tree-edit distance embeddings. *ACM TODS* 30(1), 279–332 (2005)
13. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate XML joins. In: *SIGMOD*, pp. 287–298 (2002)
14. Heumann, H., Wittum, G.: The tree-edit-distance, a measure for quantifying neuronal morphology. *BMC Neuroscience* 10(suppl. 1), P89 (2009)
15. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) *ESA 1998*. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998)
16. Korn, F., Saha, B., Srivastava, D., Ying, S.: On repairing structural problems in semi-structured data. *Proceedings of the VLDB Endowment* 6(9) (2013)
17. Lee, K.-H., Choy, Y.-C., Cho, S.-B.: An efficient algorithm to compute differences between structured documents. *IEEE TKDE* 16(8), 965–979 (2004)
18. Lin, Z., Wang, H., McClean, S.: Measuring tree similarity for natural language processing based information retrieval. In: Hopfe, C.J., Rezgui, Y., Métails, E., Preece, A., Li, H. (eds.) *NLDB 2010*. LNCS, vol. 6177, pp. 13–23. Springer, Heidelberg (2010)
19. Pawlik, M., Augsten, N.: RTED: A robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 334–345 (2011)
20. Springel, V., White, S.D.M., Jenkins, A., Frenk, C.S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J.A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., Pearce, F.: Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature* 435 (2005)
21. Tai, K.-C.: The tree-to-tree correction problem. *J. ACM* 26(3), 422–433 (1979)
22. Zhang, K.: Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition* 28(3), 463–474 (1995)
23. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.* 18(6), 1245–1262 (1989)