A New Perspective on the Tree Edit Distance

Stefan Schwarz, Mateusz Pawlik(✉), and Nikolaus Augsten

University of Salzburg, Salzburg, Austria
mateusz.pawlik@sbg.ac.at

Abstract. The tree edit distance (TED), defined as the minimum-cost sequence of node operations that transform one tree into another, is a well-known distance measure for hierarchical data. Thanks to its intuitive definition, TED has found a wide range of diverse applications like software engineering, natural language processing, and bioinformatics. The state-of-the-art algorithms for TED recursively decompose the input trees into smaller subproblems and use dynamic programming to build the result in a bottom-up fashion. The main line of research deals with efficient implementations of a recursive solution introduced by Zhang in the late 1980s. Another more recent recursive solution by Chen found little attention. Its relation to the other TED solutions has never been studied and it has never been empirically tested against its competitors. In this paper we fill the gap and revisit Chen's TED algorithm. We analyse the recursion by Chen and compare it to Zhang's recursion. We show that all subproblems generated by Chen can also origin from Zhang's decomposition. This is interesting since new algorithms that combine the features of both recursive solutions could be developed. Moreover, we revise the runtime complexity of Chen's algorithm and develop a new traversal strategy to reduce its memory complexity. Finally, we provide the first experimental evaluation of Chen's algorithm and identify tree shapes for which Chen's solution is a promising competitor.

1 Introduction

Data featuring hierarchical dependencies are often modelled as trees. Trees appear in many applications, for example, the JSON or XML data formats; human resource hierarchies, enterprise assets, and bills of material in enterprise resource planning; natural language syntax trees; abstract syntax trees of source code; carbohydrates, neuronal cells, RNA secondary structures, and merger trees of galaxies in natural sciences; gestures; shapes; music notes.

When querying tree data, the evaluation of tree similarities is of great interest. A standard measure for the tree similarity, successfully used in numerous applications, is the *tree edit distance* (TED). TED is defined as the minimum-cost sequence of node edit operations that transform one tree into another. In the classical setting [16,18], the edit operations are node deletion, node insertion, and label renaming. In this paper we consider ordered trees in which the sibling order matters. For ordered trees TED can be solved in cubic time, whereas the problem is NP-complete for unordered trees.

C. Beecks et al. (Eds.): SISAP 2017, LNCS 10609, pp. 156–170, 2017.
DOI: 10.1007/978-3-319-68474-1_11

In 1989, Zhang and Shasha proposed a recursive solution for TED [18]. The recursion decomposes trees into smaller subforests. New subforests are generated by either deleting the leftmost or the rightmost root node of a given subforest. A good choice (left or right) is essential for the runtime efficiency of the resulting algorithm. We call *Zhang decomposition* an algorithm that implements Zhang and Shasha's recursive formula.

Most TED algorithms, including the following, are dynamic programming implementations of the Zhang decomposition and differ in the strategy of left vs. right root deletion. Zhang and Shasha's own algorithm [18] runs in $O(n^4)$ time and $O(n^2)$ space for trees with n nodes. Klein [11] proposes an algorithm with $O(n^3 \log n)$ time and space complexity. Demaine et al. [7] further reduce the runtime complexity to $O(n^3)$ ($O(n^2)$ space), which is currently the best known asymptotic bound for TED. The same bounds are achieved by Pawlik and Augsten in their RTED [13] and AP-TED$^+$ [14] algorithms. According to a recent result [2] it is unlikely that a truly subcubic TED solution exists.

Although TED is cubic in the worst case, for many practical instances the runtime is much faster. For example, Zhang and Shasha's algorithm [18] runs in $O(n^2 \log^2 n)$ time for trees with logarithmic depth. Pawlik and Augsten [13] dynamically adapt their decomposition strategy to the tree shape and show that their choice is optimal. They substantially improve the performance for many practically relevant tree shapes. AP-TED$^+$ [14] is a memory and runtime optimized version of RTED and is the state of the art in computing TED.

In this paper we study an algorithm that does not fall into the mainstream category of Zhang decompositions, namely the TED algorithm introduced by Chen in 2001 [6]. Chen proposes an alternative recursive solution for TED and provides a dynamic programming implementation of his recursion. In terms of asymptotic runtime complexity, Chen's algorithm is known to be more efficient than all other algorithms for deep trees with a small number of leaves. Unfortunately, this algorithm has received little attention in literature. In particular, its relation to Zhang decompositions has never been studied. Further, we are not aware of any implementation or empirical evaluation of the algorithm. We revisit Chen's algorithm and make the following contributions:

– We perform the first analytical comparison of the decompositions by Chen and Zhang. Although the decompositions seem very different at the first glance, we show that all subproblems resulting from Chen's recursion can also be generated in a Zhang decomposition. Chen mainly differs in the way solutions for larger subproblems are generated from smaller subproblems. This is an important insight and opens the path to future research that unifies both decompositions into a single, more powerful decomposition.
– We revise the runtime complexity of Chen's algorithm. In the original paper, a significant reduction of the runtime complexity is based on the assumption of a truly subcubic algorithm for the (min,+)-product of quadratic-size matrices. Unfortunately, there is no such algorithm and even its existence remains an open problem. We adjust the asymptotic bounds accordingly and discuss the impact of the change.

– Memory is a major bottleneck in TED computations. We propose a new technique to reduce the memory complexity of Chen's algorithm from $O((n+l^2)\min\{l,d\})$ to $O((n+l^2)\log(n))$ for trees with l leaves and depth d. This is achieved by a smart traversal of the input trees that reduces the size of the intermediate result. Our technique is of practical relevance and is used in our implementation of Chen's algorithm.
– We implement and empirically compare Chen's algorithm to the state-of-the-art TED solutions. We identify tree shapes for which Chen outperforms all Zhang decomposition algorithms both in runtime and the number of intermediate subproblems. To the best of our knowledge, we are the first to implement Chen's algorithm and experimentally evaluate it.

The remaining paper is organised as follows. Section 2 analyses the relationship between Chen's algorithm and Zhang decompositions. In Sects. 3 and 4 we revise the runtime complexity and improve the memory complexity of Chen's algorithm, respectively. We experimentally evaluate Chen's algorithm in Sect. 5. Section 6 draws conclusions and points to future research directions.

2 Chen's Algorithm and Zhang Decompositions

In this section we analyse the relation of Chen's algorithm to the mainstream solutions for TED, namely *Zhang decompositions*. At the first glance, Chen's and Zhang's approaches seem very different and hard to compare. We tackle this problem in three steps: (1) We represent all subforests resulting from Chen's decomposition in the so-called *root encoding*, which was developed by Pawlik and Augsten [13] to index the subforests of Zhang decompositions. (2) We rewrite Chen's recursive formulas using the root encoding and compare them to Zhang's formulas. (3) We develop a Zhang decomposition strategy that always generates a superset of the subproblems resulting from Chen decomposition. These results lead to the important conclusion that Chen and Zhang decompositions can be combined into a single new decomposition strategy. This is a new insight that may lead to new, more powerful algorithms in the future. We refer to the end of this section for more details.

Trees, forests and nodes. A *tree* F is a directed, acyclic, connected graph with *labeled* nodes $N(F)$ and edges $E(F) \subseteq N(F) \times N(F)$, where each node has at most one incoming edge. A *forest* F is a graph in which each connected component is a tree; each tree is also a forest. We write $v \in F$ for $v \in N(F)$. 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*. A node x is an ancestor of node v iff $x = p(v)$ or x is an ancestor of $p(v)$; x is a *descendant* of v iff v is an ancestor of x. A *subforest* of a tree F is a forest with nodes $N' \subseteq N(F)$ and edges $E' = \{(v, w) : (v, w) \in E(F), v \in N', w \in N'\}$. F_v is the *subtree rooted in node* v of F iff F_v is a subforest of F and $N(F_v) = \{x : x = v \text{ or } x \text{ is a descendant of } v \text{ in } F\}$.

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

Example 1. In tree F in Fig. 1, the left (right) subscript of a node is its left-to-right (right-to-left) preorder number.

2.1 Representing Relevant Subproblems

All TED algorithms are based on some recursive solution that decomposes the input trees into smaller subtrees and subforests. Distances for larger tree parts are computed from the distances between smaller ones. A pair of subtrees or subforests that appears in a recursive decomposition is called a *relevant subproblem*. To store and retrieve the distance results for relevant subproblems they must be uniquely identified. Pawlik and Augsten [13] developed the *root encoding* to index all relevant subproblems that can appear in a Zhang decomposition.

Definition 1 (Root Encoding). *[13] Let the leftmost root node l_F and the rightmost root node r_F be two nodes of tree F, $l_F \leq r_F$. The root encoding F_{l_F,r_F} defines a subforest of F with nodes $N(F_{l_F,r_F}) = \{l_F, r_F\} \cup \{x : x \in F$, x succeeds l_F in left-to-right preorder and x succeeds r_F in right-to-left preorder$\}$ and edges $E(F_{l_F,r_F}) = \{(v, w) \in E(F) : v \in F_{l_F,r_F} \wedge w \in F_{l_F,r_F}\}$.*

Example 2. In tree F in Fig. 1(a), subforest $F_{b,j}$ in root encoding (black nodes) is obtained from F by removing all predecessors of b in left-to-right preorder and all predecessors of j in right-to-left preorder.

Chen [6] also uses a recursive formula, but the decomposition rules are different from Zhang's rules. The result of Chen's decomposition are subtrees and subforests. The subforests can be of two different types (using the original notation). $\mathscr{G}_F(l', l'')$ is a subforest of tree F composed of all maximum-size subtrees having their leaf nodes between leaves l' and l'' in left-to-right preorder. $\mathscr{F}_F(v[1..p])$ is a subforest of tree F composed of the subtrees rooted in the first p children of node v (left-to-right preorder). Interestingly, all subtrees and subforests in Chen's decomposition are expressible in the root encoding.

Example 3. Subforest $\mathscr{G}_F(c, j)$ in Fig. 1(a) (root encoding $F_{b,j}$) consists of the largest subtrees having all leaves between c and j. $\mathscr{F}_F(a[1..2])$ in Fig. 1(b) (root encoding $F_{b,f}$) consists of the subtrees rooted at the first two children of a.

Theorem 1. *Every subtree and subforest that results from Chen's recursive decomposition can be represented in root encoding.*

Proof. A subtree F_v is represented as $F_{v,v}$ in root encoding. We show that both subforest types, (a) $\mathscr{G}_F(l', l'')$ and (b) $\mathscr{F}_F(v[1..p])$, also have a root encoding.

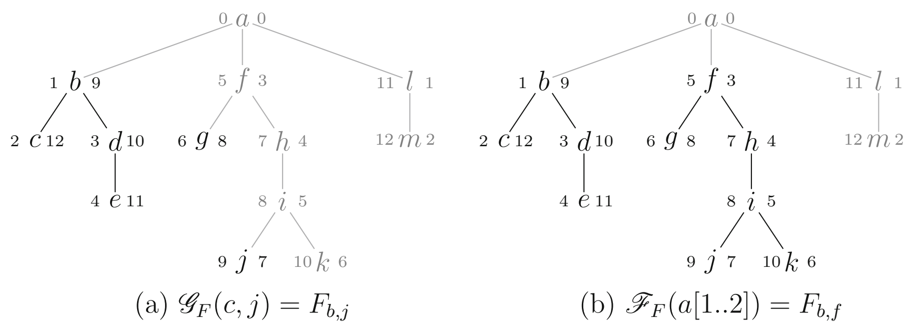

(a) $\mathscr{G}_F(c, j) = F_{b,j}$
(b) $\mathscr{F}_F(a[1..2]) = F_{b,f}$

Fig. 1. Subforests of an example tree F in Chen's and root encodings.

(a): Let a and b be the leftmost and rightmost root nodes of the forest $\mathscr{G}_F(l', l'')$. Then, the leftmost leaf of F_a is l' and the rightmost leaf of F_b is l''. We show that $\mathscr{G}_F(l', l'') = F_{a,b}$. The proof is by contradiction. (i) Assume a node $x \in F_{a,b}$ such that $x \notin \mathscr{G}_F(l', l'')$. Since $x \notin \mathscr{G}_F(l', l'')$, the subtree F_x rooted in x must have a leaf l outside the range l' to l'' (by the definition of $\mathscr{G}_F(l', l'')$), i.e., $l < l'$ or $l > l''$. This, however, is not possible since l' is the leftmost leaf node of F_a and l'' is the rightmost leaf node of F_b. (ii) Assume a node $y \in \mathscr{G}_F(l', l'')$ such that $y \notin F_{a,b}$. Then, by Definition 1, y precedes a in left-to-right preorder or y precedes b in right-to-left preorder. Consider $y < a$ in left-to-right preorder: all nodes that precede a in left-to-right preorder are either to the left of a or are ancestors of a. However, the nodes to the left of a are not in $\mathscr{G}_F(l', l'')$ since they have leaf descendants to the left of l', and ancestors of a are not in $\mathscr{G}_F(l', l'')$ since a is the leftmost root node in $\mathscr{G}_F(l', l'')$. Similar reasoning holds for y and b in right-to-left preorder. Thus y must be in $F_{a,b}$, which contradicts our assumption.

(b): $\mathscr{F}_F(v[1..p])$ is a subforest composed of the subtrees rooted in the first p children of node v. Let $c_1, \ldots, c_p$ be the first p children of node v. Then, according to the definition of the root encoding, $\mathscr{F}_F(v[1..p]) = F_{c_1,c_p}$. c_1 is the leftmost root node and c_p is the rightmost root node of $\mathscr{F}_F(v[1..p])$. Let l_1 be the leftmost leaf of c_1 and l_p be the rightmost leaf of c_p. All nodes in the subtrees rooted at nodes $c_1, \ldots, c_p$ have their left-to-right preorder ids between these of c_1 and l_p, and their right-to-left preorder ids between these of c_p and l_1. Thus, by Definition 1, $\mathscr{F}_F(v[1..p]) = F_{c_1,c_p}$. □

2.2 Comparing Recursions

Thanks to Theorem 1, which allows us to express all subforests of Chen's decomposition in root encoding, we are able to rewrite Chen's recursive formulas with root encoding. This makes them comparable to Zhang's recursion, which also has a root encoding representation.

The tree edit distance between two forests is denoted $\delta(F, G)$. The trivial cases of the recursion are the same for both Chen and Zhang: $\delta(\emptyset, \emptyset) = 0$,

$\delta(F,\emptyset) = \delta(F-v,\emptyset) + c_d(v)$, $\delta(\emptyset,G) = \delta(\emptyset,G-w) + c_i(w)$, where F and G may be forests or trees, and $\emptyset$ denotes an empty forest. $c_d(v)$, $c_i(w)$, $c_r(v,w)$ are the costs of deleting node v, inserting node w, and renaming the label of v to the label of w, respectively. $F-v$ is the forest obtained from F by removing node v and all edges at v. By $F-F_v$ (v is a root node in forest F) we denote the forest obtained from F by removing subtree F_v. Given forest F and its subforest F', $F-F'$ is a forest obtained from F by removing subforest F'.

Zhang. The recursion by Zhang and Shasha [18] distinguishes two cases.

(a) Both F_v and G_w are trees.

$$\delta(F_v,G_w) = \min\begin{cases}\delta(F_v-v,G_w)+c_d(v)\\ \delta(F_v,G_w-w)+c_i(w)\\ \delta(F_v-v,G_w-w)+c_r(v,w)\end{cases} \tag{1}$$

(b) F_{l_F,r_F} is a forest or G_{l_G,r_G} is a forest.

$$\delta(F_{l_F,r_F},G_{l_G,r_G}) = \min\begin{cases}\delta(F_{l_F,r_F}-l_F,G_{l_G,r_G})+c_d(l_F)\\ \delta(F_{l_F,r_F},G_{l_G,r_G}-l_G)+c_i(l_G)\\ \delta(F_{l_F},G_{l_G})+\delta(F_{l_F,r_F}-F_{l_F},G_{l_G,r_G}-G_{l_G})\end{cases} \tag{2}$$

In Eq. 2, instead of removing the leftmost root nodes and their subtrees (l_F, l_G, F_{l_G}, G_{l_G}) we can also remove their rightmost root node counterparts (r_F, r_G, F_{r_G}, G_{r_G}), respectively. The choice of left vs. right in each recursive step has an impact on the total number of subproblems that must be computed.

Chen. The recursion by Chen [6] distinguishes four cases. $\mathit{roots}(F_{l_F,r_F})$ and $\mathit{leaves}(F_{l_F,r_F})$ denote the set of all root resp. leaf nodes in forest F_{l_F,r_F}.

(a) Both F_v and G_w are trees. In this case, Chen's recursion is identical to Eq. 1.
(b) F_{l_F,r_F} is a forest and G_w is a tree.

$$\delta(F_{l_F,r_F},G_w) = \min\begin{cases}\delta(F_{l_F,r_F},G_w-w)+c_i(w)\\ \min\limits_{s\in \mathit{roots}(F_{l_F,r_F})}\{\delta(F_s,G_w)+\delta(F_{l_F,r_F}-F_s,\emptyset)\}\end{cases} \tag{3}$$

(c) F_v is a tree and G_{l_G,r_G} is a forest.

$$\delta(F_v,G_{l_G,r_G}) = \min\begin{cases}\delta(F_v-v,G_{l_G,r_G})+c_d(v)\\ \delta(F_v,G_{l_G,r_G}-G_{r_G})+\delta(\emptyset,G_{r_G})\\ \delta(F_v,G_{r_G})+\delta(\emptyset,G_{l_G,r_G}-G_{r_G})\end{cases} \tag{4}$$

(d) Both F_{l_F,r_F} and G_{l_G,r_G} are forests.

$$\delta(F_{l_F,r_F}, G_{l_G,r_G}) = \\ \min \begin{cases} \delta(F_{l_F,r_F}, G_{l_G,r_G} - G_{r_G}) + \delta(\emptyset, G_{r_G}) \\ \delta(F_{l_F,r_F}, G_{r_G}) + \delta(\emptyset, G_{l_G,r_G} - G_{r_G}) \\ \min\limits_{l' \in leaves(F_{l_F,r_F})} \{\delta(F_{l_F,r'_F}, G_{l_G,r_G} - G_{r_G}) + \delta(F_{l''_F,r_F}, G_{r_G}) \\ \qquad + \delta(F_{l_F,r_F} - F_{l_F,r'_F} - F_{l''_F,r_F}, \emptyset)\} \end{cases} \tag{5}$$

The nodes r'_F and l''_F in Eq. 5 are defined as follows. Let l'' be the next leaf node after l' in F_{l_F,r_F} and $lca(l', l'') \in F$ the lowest common ancestor of the two leaves l' and l'' (not necessarily $lca(l', l'') \in F_{l_F,r_F}$). Then, r'_F (l''_F) is the first descendant of $lca(l', l'')$ in F_{l_F,r_F} that is on the path to l' (l'').

$F_{l_F,r_F} - F_{l_F,r'_F} - F_{l''_F,r_F}$ is a path from $lca(l', l'')$ to a root node (node without a parent) in F_{l_F,r_F} if $lca(l', l'') \in F_{l_F,r_F}$, or it is an empty forest $\emptyset$ otherwise. While this term cannot be expressed in root encoding, the distance in Eq. 5 can be rewritten as follows: $\delta(F_{l_F,r_F} - F_{l_F,r'_F} - F_{l''_F,r_F}, \emptyset) = \delta(F_{l_F,r_F}, \emptyset)$- $\delta(F_{l_F,r'_F}, \emptyset) - \delta(F_{l''_F,r_F}, \emptyset)$. Similarly, $\delta(F_{l_F,r_F} - F_s, \emptyset) = \delta(F_{l_F,r_F}, \emptyset) - \delta(F_s, \emptyset)$ in Eq. 3.

The correctness of Chen's recursion has only been shown for forests F_{l_F,r_F} that are expressible in the form $\mathscr{G}_F(l', l'')$, where l' (l'') is the leftmost (rightmost) leaf descendant of l_F (r_F); and forests G_{l_G,r_G} that are expressible in the form $\mathscr{F}_G(v[1..p])$, where v is the parent of l_G, and r_G is the p-th child of v [6]. Other forests shapes, although they may have root encoding, are not allowed.

Satisfying this restriction and thanks to the unified notation, we can observe that the recursions by Zhang and Chen can be alternated. Since Chen's decomposition is more efficient for some tree shapes, combining the two formulas may lead to better strategies and new, more efficient algorithms.

2.3 Comparing Relevant Subproblems

The choice of left vs. right in Zhang's decomposition has an impact on the number of relevant subproblems that must be computed (cf. Sect. 2.2). This has first been discussed by Dulucq and Touzet [8]. The RTED algorithm by Pawlik and Augsten [12] computes the optimal strategy and guarantees to minimize the number of subproblems in the class of *path decompositions*. Path decompositions constitute a subclass of Zhang decompositions that includes all currently known Zhang decomposition algorithms. We design a path decomposition algorithm *ChenPaths* that mostly resembles that of Chen and show that the subproblems resulting from ChenPaths are a superset of Chen's subproblems. We evaluate the difference in the subproblems count (ChenPaths vs. Chen) in Sect. 5.

A path decomposition algorithm requires two ingredients [13]: a *path strategy* that assigns a root-leaf path to each subtree pair (F_v, G_w) $(v \in F, w \in G)$ and a *single-path function* that is used to reassemble the results for larger subforests from smaller ones (using dynamic programming). A path decomposition algorithm works as follows: (*step1*) For the input trees (F, G), a root-leaf path is looked up in the path strategy. (*step2*) The algorithm is called recursively for each subtree pair resulting from removal of the root-leaf path from the corresponding input tree. (*step3*) A single path function is executed for the input trees. The single-path function decomposes a forest F_{l_F,r_F} such that, if the rightmost root r_F is on the root-leaf path assigned to F, then the leftmost root nodes are used in Eq. 2, otherwise the rightmost root nodes are used. The path choice affects the relevant subproblems resulting from (*step2*) and (*step3*).

We design *ChenPaths* with a path strategy that maps each subtree pair (F_v, G_w) to the left path in F_v and $\varDelta^A$ single-path function [13]. Note that for left paths we could apply $\varDelta^L$ single-path function that results in less subproblems but possibly a subset of Chen's subproblems.

Theorem 2. *The subproblems resulting from ChenPaths algorithm are a superset of the subproblems resulting from Chen's algorithm.*

Proof. As discussed in [13], the subproblems of a path decomposition algorithm are those encountered by all single-path functions executed for subtree pairs resulting from (*step2*). For ChenPaths the subproblems are $\mathcal{F}(F, \varGamma^L(F)) \times \mathcal{A}(G)$, where $\mathcal{F}(F, \varGamma^L(F))$ ($\mathcal{A}(G)$) is the set of all subtrees of F (G) and their subforests obtained by a sequence of rightmost (leftmost and rightmost) root node deletions. The subproblems of Chen's algorithm are $\mathscr{F}(F) \times \mathscr{G}(G)$, where $\mathscr{F}(F)$ is the set of all subtrees of F and their subforests of the form $\mathscr{F}_F(v[1..p])$, and $\mathscr{G}(G)$ is the set of all subtrees and subforests of the form $\mathscr{G}_G(l', l'')$. To show the inclusion of the subproblems it is enough to show the following:

(a) $\mathscr{F}(F) \subseteq \mathcal{F}(F, \varGamma^L(F))$. Every subtree F_v, $v \in F$, is in both sets. Every subforest of the form $\mathscr{F}(v[1..p])$ can be obtained from the subtree F_v by a sequence of rightmost root node deletions that delete root node v and v's children (and all their descendants) from the last child to $p+1$-st. Every subforests obtained this way is in $\mathcal{F}(F, \varGamma^L(F))$.

(b) $\mathscr{G}(G) \subseteq \mathcal{A}(G)$. Every subtree G_w, $w \in G$, is in both sets. Due to Theorem 1, every subforest of the form $\mathscr{G}_G(l', l'')$ can be represented in root encoding. $\mathcal{A}(G)$ is the set of all subforest of G that can be represented in root encoding. □

In this section we showed that there are path decompositions – a subclass of the more general class of Zhang decompositions – that can generate all subproblems of Chen's algorithms. This brings Chen's algorithm even closer to the mainstream TED algorithms. It seems likely that the results of Chen may be used to develop a new single-path function that, together with the results of [14], can be used to reduce the number of subproblems needed for TED algorithms. Furthermore, the cost of such a function can be used to compute the optimal-cost path strategy for a given input instance. See [13] for the input/output requirements of a single-path function and [14] for a discussion on how to leverage costs of new single-path functions for optimal strategies.

3 Revisiting the Runtime Complexity

Chen [6] derives for his algorithm a runtime of $O(n^2 + l^2 n + l^{3.5})$ for two trees with n nodes and l leaves. In his analysis, Chen uses a so called *(min,+)-product* of two $l \times l$ matrices, which has a trivial $O(l^3)$-time solution. In order to achieve the term $l^{3.5}$ in the runtime complexity, the (min,+)-product must be solved in time $O(l^{2.5})$. Without that improvement, the respective term becomes l^4, and the overall runtime complexity of Chen's algorithm is $O(n^2 + l^2 n + l^4)$.

Chen interpreted a result by Fredman [10] towards the existence of an efficient (min,+)-product algorithm that runs in $O(l^{2.5})$. Unfortunately, as recent works point out [3,9], it is still a major open problem whether a truly subcubic algorithm (an $O(n^{3-\epsilon})$-time algorithm for some constant $\epsilon > 0$) exists for the (min,+)-product. Fong et al. [9] analyse the related difficulties, Zwick [19] summarizes (in line with Fredman's discussion [10]) that for every n, a separate program can be constructed that solves the (min,+)-product in $O(n^{2.5})$ time, but the size of that program may be exponential in n. As Fredman points out, these results are primarily of theoretical interest and may be of no practical use.

Summarizing, with the current knowledge on (min,+)-product algorithms, the runtime complexity of Chen's algorithm is $O(n^2 + l^2 n + l^4)$. This is also the complexity of our implementation, which does not use the (min,+)-product improvement. Interestingly, even without that improvement, Chen's algorithm is an important competitor for some tree shapes. We discuss the details in Sect. 5.

4 Reducing the Memory Complexity

In this section we reduce the worst-case space complexity of Chen's algorithm. This is an important contribution for making the algorithm practically relevant.

Chen's algorithm uses dynamic programming, i.e., intermediate results are stored for later reuse. The space complexity of Chen's algorithm is $O((l^2 + n)\min\{l, d\})$ for two trees with n nodes, l leaves, and depth d. The complexity is a product of two terms. The first term, $(l^2 + n)$, is the size of arrays used to store intermediate results. The second term, $\min\{l, d\}$, is the maximum number of such arrays that have to be stored in memory concurrently throughout the algorithm's execution. We observe, that there are tree shapes for which Chen's algorithm requires $\lfloor \frac{n}{2} \rfloor$ arrays, for example, a right branch tree (a vertically mirrored version of the left branch tree in Fig. 2(a)). Then, the space complexity has a tight bound of $O((l^2 + n)n)$, which is worse than $O(n^2)$ achieved by other TED algorithms. In this section, we reduce the number of arrays that must be stored concurrently from $\min\{l, d\}$ to $log_2(n)$.

By thoroughly analysing Chen's algorithm we make a few observations. (a) The algorithm traverses the nodes in one of the input trees, say F, and executes one of two functions. These functions (called by Chen *combine* and *upward*) take arrays with intermediate results as an input and return arrays as an output. (b) Due to internals of the functions combine and upward, the traversal of nodes in F must obey the following rules: children must be traversed before parents, and siblings must be traversed from left to right. These rules resemble the so-called *postorder traversal.* (c) After a node $v \in F$ is traversed, exactly one array has to be kept in memory as a result of executing the necessary functions for v and all its descendants. This array must be kept in memory until the algorithm traverses the right sibling of node v.

Observations (b) and (c) suggest that the number of nodes that cause multiple arrays to be kept in memory concurrently, i.e., the nodes waiting for their right siblings to be traversed, strongly depends on the tree shape. For example, in left branch trees at most one node at a time is waiting for its right sibling, whereas in right branch trees all leaf nodes are waiting for their right siblings until the rightmost leaf node is traversed. Our goal is to minimise the number of such nodes. Our solution is based on the so-called *heavy-light decomposition* [15] which introduces a modification to the postorder traversal in observation (b).

We divide the nodes of a tree F into two disjoint sets: *heavy nodes* and *light nodes.* The root of F is light. For each non-leaf node $v \in F$, the child of v that roots the largest (in the number of descendants) subtree is heavy, and all other children are light. In case of ties, we choose the leftmost child with the largest number of descendants to be heavy. The *heavy-light traversal* is similar to the postorder traversal with one exception: the heavy child is traversed before all other children. The remaining children are traversed from left to right.

Theorem 3. *Using the heavy-light traversal for tree F, the maximum number of nodes that cause an additional array to be kept in memory concurrently is at most $\lceil \log_2 n \rceil$.*

Proof. We modify observation (c) for the heavy-light traversal. An array has to be kept in memory for a heavy node until its immediate left and right light siblings (if any) are traversed. For a light node an array has to be kept in memory until its right light sibling is traversed. Nodes never wait for their heavy siblings because the heavy sibling is traversed first.

Consider a path γ in tree F. The number of arrays that have to be kept in memory concurrently is proportional to the number of light nodes on γ. Let $L(\gamma)$ be all light nodes on path γ, and $W(\gamma)$ be all immediate siblings waiting for nodes in $L(\gamma)$. The array for a node in $W(\gamma)$ must be kept in memory until its sibling in $L(\gamma)$ is traversed. That brings us to the conclusion that the maximum number of arrays that have to be kept in memory concurrently equals the maximum number of light nodes on any path in F.

Let $|F| = |N(F)|$ denote the size of tree F. For any light node v, its heavy sibling has more nodes than v. It holds that $|F_{p(v)}| > 2|F_v|$, and $|F_v| < \frac{|F_{p(v)}|}{2}$. Then, each light node v on a path γ decreases the number of consecutive nodes on γ to be at most $\frac{|F_{p(v)}|}{2}$. Hence, the maximum number of light nodes on any path in F is at most $\lceil \log_2 |F| \rceil$. □

For example, consider left and right branch trees. The heavy-light traversal causes at most one node at a time to wait for its sibling to be traversed. Thus, at most one additional array has to be stored in memory at any time.

With Theorem 3 we reduce the space complexity of Chen's algorithm to $O((l^2+n)\log n)$. For trees with $O(\sqrt{n})$ leaves the complexity becomes $O(n \log n)$. This is remarkable since all other TED algorithms require $O(n^2)$ space independently of the tree shape. So far, space complexities better than $O(n^2)$ were achieved only by approximations (for example, $O(n)$-space pq-gram distance by Augsten et al. [1]), algorithms computing an upper bound for TED (for example, $O(n \log n)$-space constrained tree edit distance by Wang et al. [17]), and algorithms computing the lower bound for TED (for example, $O(n)$-space string edit distance by Chan [4]). Trees with the number of leaves in $O(\sqrt{n})$ are characterised by long node chains, for example, tree representations of RNA secondary structures [5].

5 Experimental Evaluation

In this section we experimentally evaluate Chen's algorithm and compare it to the classical algorithm by Zhang and Shasha (ZS) [18] and the state-of-the-art algorithm AP-TED$^+$ by Pawlik and Augsten [14]. All algorithms were implemented as single-thread applications in Java 1.7. and executed on a single core of a server machine with 8 cores Intel Xeon 2.40 GHz CPUs and 96GB of RAM. The runtime results are averages over three runs.

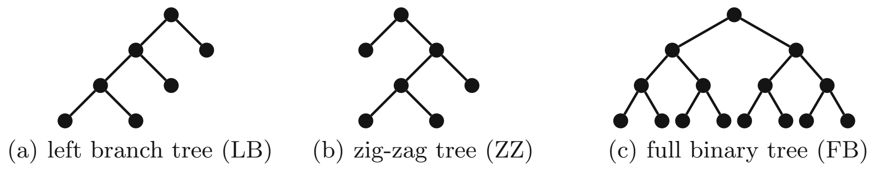
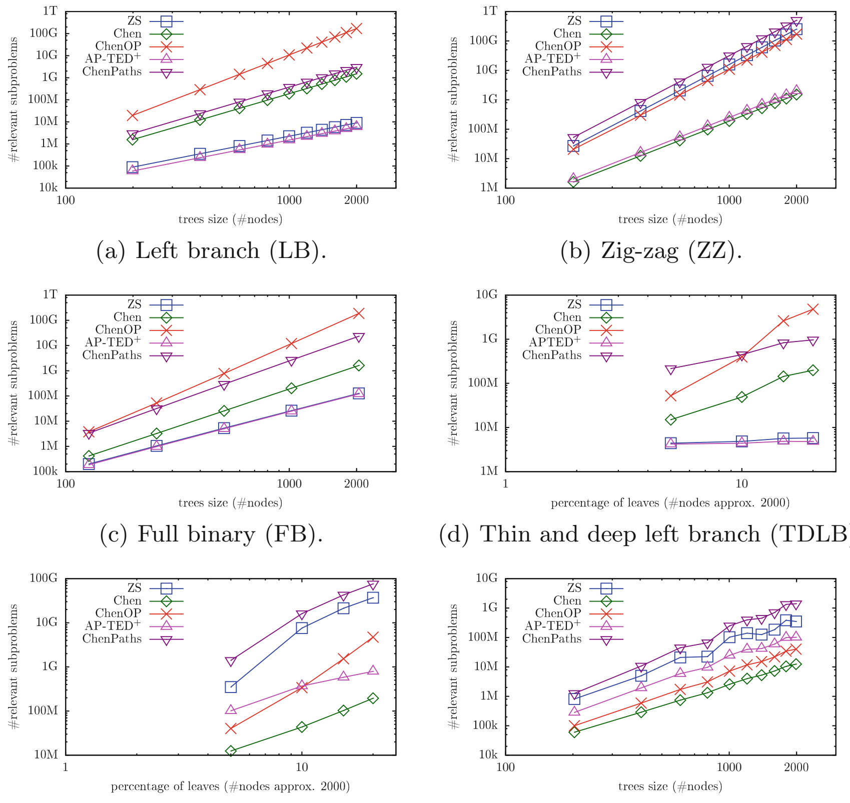

(a) left branch tree (LB) (b) zig-zag tree (ZZ) (c) full binary tree (FB)

Fig. 2. Shapes of the synthetic trees

Implementation. We implemented the original algorithm by Chen without the matrix multiplication extension (cf. Section 3). During the implementation process we discovered some minor bugs in Chen's algorithm that we fixed in our implementation. We further extended the implementation with our new traversal strategy to reduce the memory complexity (cf. Section 4). Our tests (not presented due to space limitations) show that the memory usage reduction is significant, for example, in the case of zig-zag trees we reduce the number of arrays concurrently stored in memory from linear to constant. That translates to a reduction of the memory footprint by one order of magnitude already for small trees with 200 nodes. The improvement ratio grows with the tree size.

Datasets. Similar to Pawlik and Augsten [13], we generated trees of five different shapes and varying sizes. Left branch (LB), zig-zag (ZZ), and full binary trees (FB) are shown in Fig. 2. In addition, we created thin and deep trees which favor Chen's algorithm. Thin and deep left branch trees (TDLB) are obtained from LB trees by inserting node chains (of equal length) to the left child of every node. Thin and deep zig-zag trees (TDZZ) are obtained from long node chains by attaching leaf nodes at random positions (alternating between left and right such that the resulting tree resembles a zig-zag tree). For thin and deep trees, we vary the ratio of leaf nodes from 5% to 20%. It is worth mentioning that LB/TDLB trees are the best-case input for ZS, while the performance of AP-TED$^+$ does not depend on the tree shape.



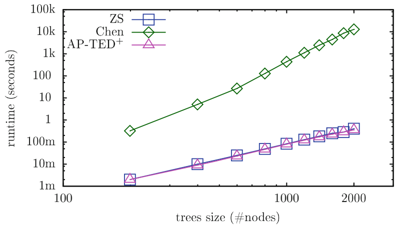
(a) Left branch (LB). (b) Zig-zag (ZZ).

(c) Full binary (FB). (d) Thin and deep left branch (TDLB).

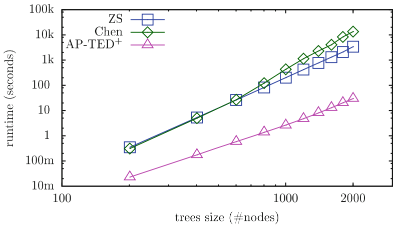
(e) Thin and deep zig-zag (TDZZ). (f) Thin and deep zig-zag, 5% leaves.

Fig. 3. Number of relevant subproblems for different tree shapes.

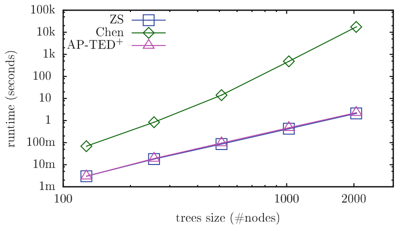
Number of relevant subproblems. The complexity of TED algorithms is proportional to the number of subproblems that an algorithm has to compute. Figure 3 shows the number of subproblems for different tree shapes. For the LB, FB, and TDLB shapes the leaders are AP-TED$^+$ and ZS, while Chen must compute many more subproblems. For the ZZ shape, the winners are Chen and AP-TED$^+$, ZS performs poorly. For TDZZ trees Chen outperforms its competitors. For TDZZ trees with the leaves ratio of 5% the difference is one order of magnitude (Fig. 3(f)). We vary the leaves ratio and observe that Chen results in the smallest number of subproblems for all tested leave ratios between 5% and 20% (Fig. 3(e)). ZS and AP-TED$^+$ require only a constant number of operations for each relevant subproblem, while Chen must evaluate the minimum over a linear number of options (see Eqs. 3 and 5). We count the overall number of elements in the minima and report the result as ChenOP in Fig. 3. Although the number of constant time operations is much larger then the number of subproblems in Chen's algorithm, Chen remains the winner for TDZZ trees with leaves ratio of 5%. With more than 10% leaf ratio Chen looses in favour of AP-TED$^+$, but is better than ZS for all ratios. Additionally, we mark the number of subproblems of ChenPaths introduced in Sect. 2.3. The results confirm that ChenPaths results in more subproblems than Chen and ZS. The latter is caused by the path strategy and single-path function used in ChenPaths.



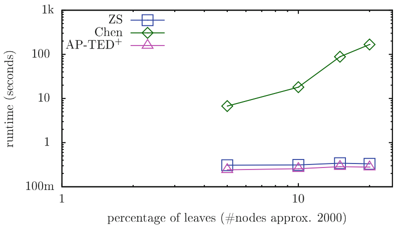
(a) Left branch (LB).



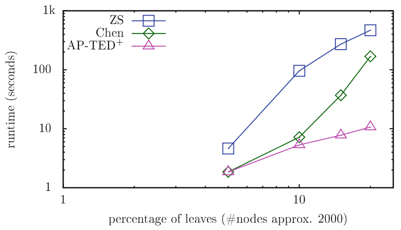
(b) Zig-zag (ZZ).



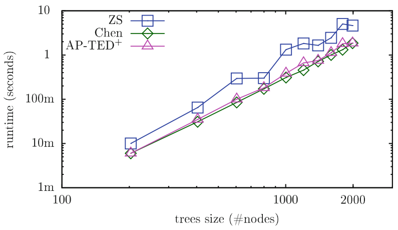
(c) Full binary (FB).



(d) Thin and deep left branch (TDLB).



(e) Thin and deep zig-zag (TDZZ).



(f) Thin and deep zig-zag, 5% leaves.

Fig. 4. Runtime for different tree shapes.

Runtime. We compare the runtime of the algorithms for different tree shapes (Fig. 4). The trend is consistent with the results for the number of subproblems. Chen wins only for TDZZ trees with 5% leaf ratio (Fig. 4(f)); the runtime difference to the runner-up AP-TED$^+$ is marginal. Chen's runtime quickly increases with the leaf ratio.

6 Conclusion

In this paper we analysed and experimentally evaluated the tree edit distance algorithm by Chen [6]. We revised the runtime and improved the space complexity of Chen's algorithm to $O(n \log(n))$ for trees with $O(\sqrt{n})$ leaves. Our experiments showed that Chen beats its competitors for thin and deep zig-zag trees with few leaves. Our analytic results suggest that the recursions of Chen and Zhang can be combined. For the future work, it is interesting to develop new dynamic programming algorithms that can leverage both recursive decompositions. This requires a cost formula for combined Chen and Zhang strategies, and an efficient bottom-up traversal for the dynamic programming implementation of the combined strategy.

Acknowledgments. This work was supported by Austrian Science Fund (FWF): P 29859-N31. We thank Willi Mann and Daniel Kocher for valuable discussions.

References

1. Augsten, N., Böhlen, M., Gamper, J.: The pq-gram distance between ordered labeled trees. ACM Trans. Database Syst. (TODS) **35**(1) (2010)
2. Bringmann, K., Gawrychowski, P., Mozes, S., Weimann, O.: Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). CoRR, abs/1703.08940 (2017)
3. Bringmann, K., Grandoni, F., Saha, B., Williams, V.V.: Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In: IEEE Annual Symposium on Foundations of Computer Science (FOCS), pp. 375–384 (2016)
4. Chan, T.Y.T.: Practical linear space algorithms for computing string-edit distances. In: Huang, D.-S., Li, K., Irwin, G.W. (eds.) ICIC 2006. LNCS, vol. 4115, pp. 504–513. Springer, Heidelberg (2006). doi:10.1007/11816102_54
5. Chen, S., Zhang, K.: An improved algorithm for tree edit distance incorporating structural linearity. In: International Conference on Computing and Combinatorics (COCOON) (2007)
6. Chen, W.: New algorithm for ordered tree-to-tree correction problem. J. Algorithms **40**(2), 135–158 (2001)
7. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. ACM Trans. Algorithms **6**(1) (2009)
8. Dulucq, S., Touzet, H.: Decomposition algorithms for the tree edit distance problem. J. Discrete Algorithms **3**(2–4), 448–471 (2005)
9. Fong, K.C.K., Li, M., Liang, H., Yang, L., Yuan, H.: Average-case complexity of the min-sum matrix product problem. Theoret. Comput. Sci. **609**, 76–86 (2016)

10. Fredman, M.L.: New bounds on the complexity of the shortest path problem. SIAM J. Comput. **5**(1), 83–89 (1976)
11. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Bilardi, G., Italiano, G.F., Pietracaprina, A., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998). doi:10.1007/3-540-68530-8_8
12. Pawlik, M., Augsten, N.: RTED: a robust algorithm for the tree edit distance. Proc. VLDB Endow. (PVLDB) **5**(4), 334–345 (2011)
13. Pawlik, M., Augsten, N.: Efficient computation of the tree edit distance. ACM Trans. Database Syst. (TODS) **40**(1) (2015). Article No. 3
14. Pawlik, M., Augsten, N.: Tree edit distance: robust and memory-efficient. Inf. Syst. **56**, 157–173 (2016)
15. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
16. Tai, H.-C.: The tree-to-tree correction problem. J. ACM (JACM) **26**(3), 422–433 (1979)
17. Wang, L., Zhang, K.: Space efficient algorithms for ordered tree comparison. Algorithmica **51**(3), 283–297 (2008)
18. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6), 1245–1262 (1989)
19. Zwick, U.: All pairs shortest paths using bridging sets and rectangular matrix multiplication. J. ACM (JACM) **49**(3), 289–317 (2002)