Theory Comput Syst (2011) 48: 865–887
DOI 10.1007/s00224-010-9274-7

Algorithms for Finding a Most Similar Subforest

Jesper Jansson · Zeshan Peng

Published online: 8 June 2010

Abstract Given an ordered labeled forest F ("the target forest") and an ordered labeled forest G ("the pattern forest"), *the most similar subforest problem* is to find a subforest F' of F such that the forest edit distance between F' and G is minimum over all possible F'. This problem generalizes several well-studied problems which have important applications in locating patterns in hierarchical structures such as RNA molecules' secondary structures and XML documents. Algorithms for the most similar subforest problem restricted to subforests which are either rooted subtrees or simple substructures exist in the literature; in this article, we show how to solve the most similar subforest problem for two other types of subforests: sibling substructures and closed subforests.

Keywords Approximate pattern matching · Forest edit distance · Simple substructure · Sibling substructure · Closed subforest · Dynamic programming

A preliminary version of this article appeared in *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching* (CPM 2006), volume 4009 of *Lecture Notes in Computer Science*, pp. 377–388, Springer-Verlag, 2006.
J. Jansson funded in part by JSPS (Japan Society for the Promotion of Science), Kyushu University, and the Special Coordination Funds for Promoting Science and Technology.

J. Jansson (✉)
Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
e-mail: Jesper.Jansson@ocha.ac.jp

Z. Peng
Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong, Hong Kong
e-mail: zspeng@cs.hku.hk

1 Introduction

An *ordered labeled tree* is a rooted tree in which the left-to-right ordering among nodes is fixed and each node is labeled by a symbol from a given alphabet. An *ordered labeled forest* is a sequence of ordered labeled trees. Ordered labeled trees and forests are useful data structures for hierarchical data representation; for example, XML documents are basically ordered labeled trees [5] and RNA molecules' secondary structures without pseudoknots can be represented by ordered labeled forests [10, 13, 20]. Below, we refer to ordered labeled trees and ordered labeled forests as *trees* and *forests*, respectively.

This article studies the following problem which we call *the most similar subforest problem*: Given a forest F ("the target forest") and a forest G ("the pattern forest"), find a subforest of F which is the most similar to G, where the *forest edit distance* [16, 21, 25] is used to measure the similarity between forests. The forest edit distance between two forests is the cost of a cheapest possible sequence of so-called relabel, delete, and insert operations on nodes which transforms one forest into the other; see Sect. 2.1 below for formal definitions. There are many ways to define "subforest", so we examine several alternatives and show how to solve all the resulting problems efficiently. Our techniques combine and extend the techniques of [10] and [25].

1.1 Previous Results

This subsection contains a short survey of previous results. For any forest F, denote the number of nodes in F by $|F|$, the depth of F by $dp(F)$, the set of leaves in F by $L(F)$, and the set of all nodes in F by $V(F)$. The subtree of F rooted at a node i is written as $F[i]$, and $p(F)$ is an imaginary parent node of the roots of the trees in F. See Sect. 2.1 for the definitions of the different types of subforests mentioned here: rooted subtrees, simple substructures, sibling substructures, and closed subforests.

Tai [21] gave the first algorithm for computing the forest edit distance $\delta(F, G)$ between two given forests F and G. Later, Zhang and Shasha [25] presented a faster algorithm, Algorithm `ZS-main`, for computing $\delta(F, G)$ in $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space. (Actually, these papers assumed F and G to be *trees*, but it is simple to generalize their methods to forests by letting every forest F contain the imaginary parent node $p(F)$ introduced in Sect. 2.1.) By applying different techniques, Klein [15] obtained an algorithm that computes $\delta(F, G)$ in $O(|F|^2 \cdot |G| \cdot \log |G|)$ time and $O(|F|^2 \cdot |G| \cdot \log |G|)$ space, and more recently, Demaine et al. [7] improved this result to $O(|F|^2 \cdot |G| \cdot (1 + \log(|G|/|F|)))$ time and $O(|F| \cdot |G|)$ space. Chen [4] and Touzet [22] have developed algorithms that are even faster for certain types of inputs. Note, however, that none of the existing algorithms beats all the others in every single case; e.g., Algorithm `ZS-main` is still the fastest when $dp(F)$ and $dp(G)$ are small but slower than the algorithm of Demaine et al. [7] in the worst case.

In short, Algorithm `ZS-main` computes and stores $\delta(F[i], G[j])$ for all $i \in V(F)$ and $j \in V(G)$ using bottom-up dynamic programming. By running `ZS-main` and then selecting a node $i \in V(F)$ which minimizes $\delta(F[i], G[p(G)])$, one can therefore directly find one of the subtrees rooted at a node in F which is the most similar to G, i.e., a most similar *rooted subtree*. However, to identify a most similar *simple substructure*, it is not practical to compute the forest edit distance between G and every one of the possibly exponentially many (in $|F|$) simple substructures of F. To cope with this issue, Zhang and Shasha [25] provided a variant of Algorithm `ZS-main`, henceforth referred to as Algorithm `ZS-simple`, for finding a most similar simple substructure of F to G which runs in $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space. It is outlined in Sect. 3 below.

The problems of finding a most similar *sibling substructure* and a most similar *closed subforest* have not been studied previously. Nevertheless, the algorithm of Klein [15] in its simplified form [2] may be adapted to find a most similar closed subforest, as we briefly discuss next. Following [2], define the *(i, j)-deleted subforest of F* as the forest obtained from F by successively applying the delete operation on the rightmost root j times and then on the leftmost root i times. The simplified version of Klein's algorithm in [2] computes $\delta(F', G)$ for all possible (i, j)-deleted subforests F' of F, so the most similar closed subforest problem can be solved in the same asymptotic complexity by inserting a step to remember the smallest value of $\delta(F', G)$ computed so far for any (i, j)-deleted subforest F' whose leftmost and rightmost roots are siblings; in order to test this last condition in $O(1)$ time, keep a pointer to the leftmost root and a pointer to the rightmost root of each (i, j)-deleted subforest and check if they have the same parent in F in $O(1)$ time. This yields an algorithm for the most similar closed subforest problem running in $O(|F|^2 \cdot |G| \cdot \log |G|)$ time and $O(|F|^2 \cdot |G| \cdot \log |G|)$ space. It seems difficult to extend the algorithm of Demaine et al. [7] in the same way to find most similar closed subforests, though, as it does not compute $\delta(F', G)$ for all possible (i, j)-deleted subforests F' of F.

1.2 Our Contributions

We focus on how to solve the most similar subforest problem where "subforest" means "sibling substructure" or "closed subforest".

Our first new result, an algorithm for finding a most similar sibling substructure (named `Modified_ZS-simple`), is obtained by proving that Zhang and Shasha's [25] Algorithm `ZS-simple` for simple substructures essentially already solves the problem for us, so that only a small modification to their algorithm is necessary. The second new result, an algorithm for efficiently finding a most similar closed subforest (named `Most_similar_csf`), is our main contribution. It is based on a novel recurrence for expressing the value of δ for subforests of F and G of a certain form (Lemma 12). The recurrence was designed to take advantage of the observation that having access to the values of $m(i)$ for all $i \in V(F)$ (see Sect. 2.1 for the definition of $m(i)$), it is easy to check if a given leaf ℓ is a descendant of a given node x in $O(1)$ time (Lemma 2). We believe that this technique can be applied to derive new recurrences for other combinatorial problems involving closed subforests such as *the small-in-large closed subforest similarity problem* [10] (cf. Sect. 7) as well, leading to more efficient dynamic programming-based solutions for such problems.

Table 1 The running times of different algorithms for finding a most similar subforest

Finding a most similar	Algorithm	Time complexity
Rooted subtree	`ZS-main` (Ref. [25])	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{\lvert L(F)\rvert, dp(F)\} \cdot \min\{\lvert L(G)\rvert, dp(G)\})$
Simple substructure	`ZS-simple` (Ref. [25]; see also Sect. 3)	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{\lvert L(F)\rvert, dp(F)\} \cdot \min\{\lvert L(G)\rvert, dp(G)\})$
Sibling substructure	`Modified_ZS-simple` (Sect. 4)	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \min\{\lvert L(F)\rvert, dp(F)\} \cdot \min\{\lvert L(G)\rvert, dp(G)\})$
Closed subforest	`Most_similar_csf` (Sect. 5)	$O(\lvert F\rvert \cdot \lvert G\rvert \cdot \lvert L(F)\rvert \cdot \min\{\lvert L(G)\rvert, dp(G)\})$

The running times of `Modified_ZS-simple` and `Most_similar_csf` are summarized in Table 1, along with the running times of `ZS-main` and `ZS-simple` from [25]. All four algorithms run in $O(|F| \cdot |G|)$ space.

In comparison, `Most_similar_csf` is faster than the simplified version of Klein's algorithm [2, 15] (see Sect. 1.1) when $L(F)$, $L(G)$, or $dp(G)$ is small, and it uses much less space for all types of inputs.

1.3 Organization of Article

The rest of the article is organized as follows. Section 2 contains formal definitions of the most similar subforest problem and the different types of subforests we consider. Section 2 also examines some useful basic properties of the left-to-right postorder numbering of the nodes (Lemmas 1–3), introduces additional notation needed to describe and analyze the algorithms in later sections, and explains the preprocessing done by the algorithms. In Sect. 3, we review Algorithm `ZS-simple` from [25] for finding a most similar simple substructure of F to G. Sections 4 and 5 present our new algorithms for finding a most similar sibling substructure of F to G (Algorithm `Modified_ZS-simple`) and a most similar closed subforest of F to G (Algorithm `Most_similar_csf`), respectively. Next, Sect. 6 mentions applications for the most similar subforest problem. Finally, we discuss open problems in Sect. 7.

2 Preliminaries

2.1 Notation, Terminology, and Problem Definition

Basic notation Let F be any forest. To simplify the presentation, we assume that the roots of the trees in F share an imaginary parent node, denoted by $p(F)$, which is considered to belong to F and which is labeled by a special symbol '$\diamond$' different from all other node labels in F. Let $V(F)$ and $L(F)$ represent the set of all nodes

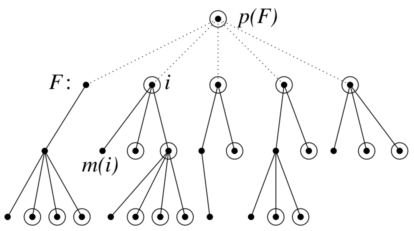

Fig. 1 A forest F consisting of five trees (presented here without node labels). $K(F)$ is the set of circled nodes. According to the definitions, the node marked i has number 14. Also shown in the figure are the imaginary parent node $p(F)$ and the node $m(i)$

belonging to F and the set of leaves in F, respectively. For convenience, denote the number of nodes in F by $|F|$, i.e., $|F| = |V(F)|$. Define deg(F) (the *degree of* F) as the maximum number of children over all nodes in $V(F)$, and $dp(F)$ (the *depth of* F) as the number of edges on the longest path from a root node in F to a leaf of F. For any $i \in V(F)$, denote the parent of i by $p(i)$ and the label of i by $label(i)$.

Any $i_1, i_2 \in V(F)$ are *siblings* if they have the same parent in F. (Thus, throughout this article, every node is defined to be a sibling of itself.) If $i_1, i_2 \in V(F)$ are siblings and $i_1 \neq i_2$ also holds then i_1 and i_2 are called *proper siblings*. Define the *key nodes of* F as the set $K(F) = \{p(F)\} \cup \{i \mid i \in V(F)$ and i has a left proper sibling$\}$. See Fig. 1 for an example.

Enumerate the nodes of F from 1 to $|F|$ in accordance with the order in which they are visited by a left-to-right postorder traversal of F, and associate each node with its number. For any $i_1, i_2 \in V(F)$, define $i_1 : i_2$ as the set of nodes whose numbers are greater than or equal to i_1 and less than or equal to i_2. For any $i_1, i_2 \in V(F)$, define $i_1 \cdot\cdot\, i_2$ as the set of all nodes in $i_1 : i_2$ which are both siblings of i_1 and siblings of i_2 (recall that every node is a sibling of itself). By definition, if i_1 and i_2 are not siblings or if they are siblings such that $i_1 > i_2$ then $i_1 \cdot\cdot\, i_2 = \emptyset$. Next, for any $i \in V(F)$, define $m(i)$ as the smallest numbered node in the subtree consisting of i and all proper descendants of i. Figure 1 illustrates $m(i)$ for a given node i. Finally, for any nodes $x, y \in V(F)$, write $y \preceq x$ if y is a proper descendant of x in F or equal to x, and $y \npreceq x$ otherwise.

From here on, we assume that all forests are labeled by a finite alphabet Σ with $\diamond \in \Sigma$ and that there exists a special blank symbol '$-$' which does not belong to Σ. We also assume that a given function γ assigns a non-negative *cost* to every pair of symbols from $\Sigma \cup \{-\}$. To be precise, we require γ to be a pseudometric of the type $\gamma : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of non-negative real numbers. We also require that $\gamma(-,-) = \infty$, where ∞ represents any sufficiently large real number, and $\gamma(a,\diamond) = \gamma(\diamond,b) = \gamma(\diamond,\diamond) = \gamma(\diamond,-) = \gamma(-,\diamond) = 0$.

Forest edit distance and edit mappings As in [21, 25], first define three *edit operations* on any forest F which may be used to modify the structure and labeling of F:

- *Relabel:* Change the label of a node i in F.
- *Delete:* Remove a node i and the edge between i and $p(i)$ from F, while letting the children of i (if any) become children of $p(i)$ without changing the children's relative left-to-right ordering.
- *Insert:* Insert a new node i with any label into F (the inverse operation of delete). Node i will become a child of an existing node j (where j is allowed to be the

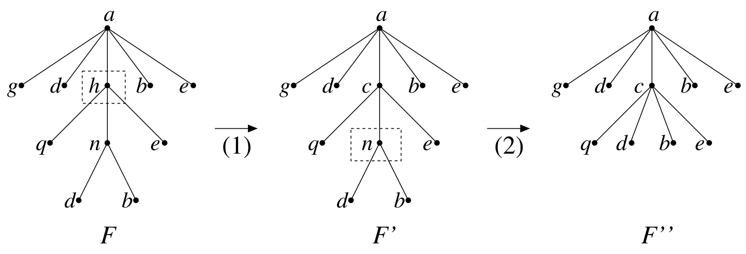

Fig. 2 Two examples of edit operations: (1) Relabeling node 8 in the forest F (whose label is h) with the label c; and (2) deleting node 6 in F' (whose label is n) so that node 8 (the parent of node 6) in F' has four children in the resulting forest F''

imaginary parent node) and the parent of a (possibly empty) consecutive subsequence of children of j.

Refer to Fig. 2 for examples of the relabel and delete operations. Without loss of generality, we do not allow imaginary parent nodes to be relabeled, deleted, or inserted.

The given function γ is overloaded to give the *cost* of any edit operation s (denoted by $\gamma(s)$) as follows: the cost of relabeling any node i by any label $b \in \Sigma$ is $\gamma(label(i), b)$, the cost of deleting any node i is $\gamma(label(i), -)$, and the cost of inserting a new node with label $b \in \Sigma$ is $\gamma(-, b)$. Then, γ is naturally extended to any sequence of edit operations $S = s_1, \ldots, s_k$ by setting $\gamma(S) = \sum_{i=1}^{|S|} \gamma(s_i)$. Lastly, the *forest edit distance* between two forests F and G, denoted by $\delta(F, G)$, is the cost of the cheapest sequence of edit operations which transforms F into G, i.e., $\delta(F, G) = \min\{\gamma(S) \mid S$ is a sequence of edit operations transforming F to $G\}$.

An equivalent formulation of the forest edit distance $\delta(F, G)$ is as the cost of an *optimal edit mapping* between F and G [21, 25]. This formulation has the advantage of being well suited for dynamic programming solutions. Formally, an *edit mapping* M between two forests F and G is a set of pairs of the form (i, j), where $i \in V(F)$ and $j \in V(G)$, such that for any two pairs $(i_1, j_1), (i_2, j_2) \in M$, the following properties are satisfied: (1) $i_1 = i_2$ if and only if $j_1 = j_2$; (2) i_1 is an ancestor of i_2 if and only if j_1 is an ancestor of j_2; and (3) $i_1 < i_2$ if and only if $j_1 < j_2$. For any $(i, j) \in M$, we say that node i is *linked with* node j in M. Define the *left-linked set* as $M_F = \{i \mid \exists(i, j) \in M\}$ and the *left-unlinked set* as $U_F = V(F) \setminus M_F$, and define the *right-linked set* M_G and *right-unlinked set* U_G analogously. Any edit mapping M between F and G induces a sequence of delete and relabel operations on F and G such that the resulting forests F' and G' are identical.[1] Next, for any $i \in V(F)$ and $j \in V(G)$, define $f(i) = label(i)$ and $g(j) = label(j)$, and define the *cost* of an edit mapping M as:

$$\delta(M) = \sum_{(i,j)\in M} \gamma(f(i), g(j)) + \sum_{i\in U_F} \gamma(f(i), -) + \sum_{j\in U_G} \gamma(-, g(j)).$$

[1]More precisely, every $i \in U_F$ means "delete node i from F", every $j \in U_G$ means "delete node j from G", and every $(i, j) \in M$ with $label(i) \neq label(j)$ means "relabel i with the label of j".

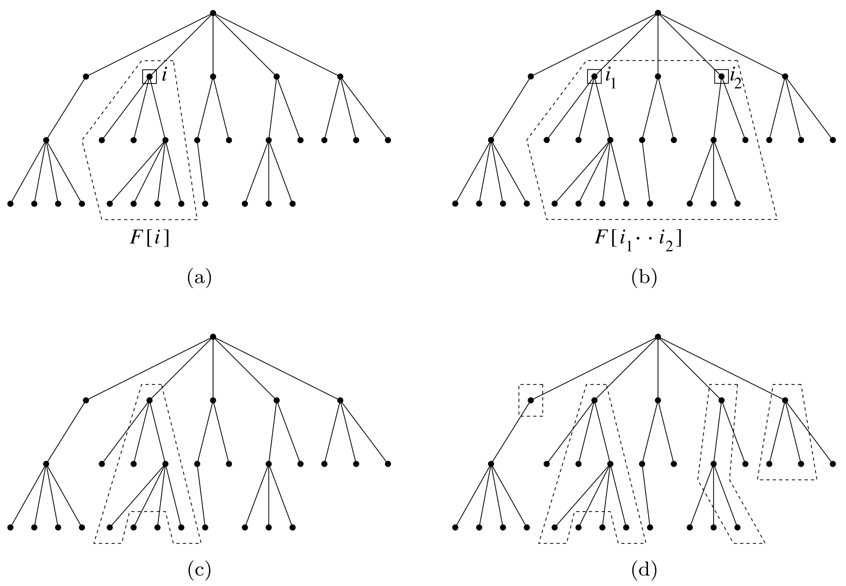

Fig. 3 Various kinds of subforests of a given forest F. (**a**) The subtree of F rooted at a node i, denoted by $F[i]$; (**b**) a closed subforest of F, denoted by $F[i_1 \cdot\cdot i_2]$; (**c**) a simple substructure of F; and (**d**) a sibling substructure of F

Finally, an *optimal edit mapping* between two forests F and G is an edit mapping with the minimum cost: $\min\{\delta(M)\}$ over all possible M. This cost is equal to the forest edit distance $\delta(F, G)$ between F and G [21, 25].

Subforest definitions We consider the following five types of subforests of a forest F. See Fig. 3 for some examples.

- For any node i in F, the *subtree of F rooted at i* is the subtree consisting of i and all proper descendants of i, and is denoted by $F[i]$. ($F[i]$ is also called a *rooted subtree of F*.)
- For any siblings i_1, i_2 in F, the set of subtrees rooted at $i_1 \cdot\cdot i_2$ forms a *closed subforest of F*, denoted by $F[i_1 \cdot\cdot i_2]$.[2]
- A *simple substructure of F* is any connected subgraph of F.
- A *sibling substructure of F* is any set of disjoint simple substructures of F whose roots are siblings (not necessarily consecutive) in F.
- Given any subset S of the nodes in F, *the restriction of F to S*, denoted by $F\|_S$, is defined as the forest obtained from F by deleting all nodes not in S.

Observe that $F[i_1 \cdot\cdot i_1] = F[i_1]$ for any node i_1 in F.

Problem definition Using any one of the above definitions for “subforest”, we say that a *most similar subforest* of a forest F to a forest G is a subforest F' of F that

[2]Closed subforests were introduced by Höchsmann et al. [10].

minimizes $\delta(F', G)$. *The most similar subforest problem* (again, using any one of the above definitions for "subforest") is:

Given two forests F and G, find a most similar subforest of F to G.

2.2 Simple Observations

We start with two simple but useful lemmas:

Lemma 1 *Let F be a forest. Then*:

(a) *For any $i \in V(F)$, $m(i)$ is the leftmost leaf in $F[i]$.*
(b) *$m(p(F))$ is the node with number* 1.
(c) *For any siblings i_1 and i_2 in $V(F)$, $F[i_1 \cdots i_2] = F\|_{m(i_1):i_2}$.*
(d) *For any $i \in V(F)$, $F[i] = F\|_{m(i):i}$.*

Proof

(a) First, suppose that $m(i)$ is not a leaf. Then $m(i)$ has at least one child c. c is also a proper descendant of i, but $c < m(i)$ by the definition of postorder traversal, which leads to a contradiction. Thus, $m(i)$ must be a leaf. Next, suppose there exists some leaf $l \neq m(i)$ such that $l \preceq i$ and l lies to the left of $m(i)$. Then, $l < m(i)$ according to the left-to-right postorder traversal, which is also a contradiction.
(b) Follows immediately from part (a).
(c) Follows from part (a) and the left-to-right postordering of the nodes.
(d) $F[i] = F[i \cdots i] = F\|_{m(i):i}$ by part (c) because i is a sibling of itself. □

Lemma 2 *For any nodes $x, y \in V(F)$, $y \preceq x$ if and only if $m(x) \leq y \leq x$.*

Proof If $y \preceq x$ then $y \leq x$ by the postorder numbering and $m(x) \leq y$ by the definition of $m(x)$. To prove the other direction, note that Lemma 1(a) and the left-to-right postorder numbering imply that the set $\{m(x), \dots, x\}$ is exactly the set of nodes in $F[x]$. Therefore, if $m(x) \leq y \leq x$ then y must belong to $F[x]$, and then y is either a proper descendant of x or equal to x. □

2.3 Nearest Key Node Ancestors

For each node i in a given forest F, define $A(i)$ (*the nearest key node ancestor of i*) by: If $i \in K(F)$ then let $A(i) = i$; otherwise, let $A(i)$ be the nearest ancestor of i which belongs to $K(F)$. Then:

Lemma 3 *For any $i \in V(F)$, $m(i) = m(A(i))$.*

Proof If $i \in K(F)$ then $m(i) = m(A(i))$ is trivially true. If $i \notin K(F)$ then i has no left proper sibling so $m(i) = m(p(i))$ by Lemma 1(a), where $p(i)$ is the parent of i. By induction, $m(i) = m(x)$ for every node x belonging to the path between i and $A(i)$, and in particular, $m(i) = m(A(i))$. □

2.4 The Cut Operation and Definition of Ψ

To describe the algorithms in later sections, we need an additional edit operation called *cut*. It was originally introduced by Zhang and Shasha in [25].[3]

Let F be a forest. For any node i in F, *cutting at node* i means removing the entire subtree $F[i]$ (along with the parent edge of i) from F at cost 0.[4] For any two nodes u and v in F with $u \neq v$, u and v are said to be *consistent* if u is not a proper descendant of v and v is not a proper descendant of u. A set C of nodes from F is *consistent* if every pair of nodes in C is consistent. Denote the set of all consistent sets of nodes in F by $\mathcal{C}(F)$, and for any $C \in \mathcal{C}(F)$, let $F \ominus C$ be the forest obtained from F by cutting all nodes in C. Lemma 4 below follows directly from the definition of a simple substructure.

Lemma 4 *Let i be a node in a forest F. F' is a simple substructure of F rooted at i if and only if $F' = F[i] \ominus C$ for some $C \in \mathcal{C}(F[i])$.*

Next, for any given forests F and G, define:

$$\Psi(F, G) = \min_{C \in \mathcal{C}(F)} \{\delta(F \ominus C, G)\}.$$

Although not explicitly stated by Zhang and Shasha in [25], the following lemma forms the basis for their Algorithm `ZS-simple`.

Lemma 5 *The value $\min_{i \in V(F)}\{\Psi(F[i], G)\}$ is equal to the cost of an optimal edit mapping between G and a most similar simple substructure of F to G.*

Proof For any forest X and any node $i \in V(X)$, let $\mathcal{S}(X)$ denote the set of all simple substructures of X and let $\mathcal{S}^i(X)$ denote the set of simple substructures of X with root node i. To prove the lemma, we will show that $\min_{F' \in \mathcal{S}(F)}\{\delta(F', G)\} = \min_{i \in V(F)} \{\Psi(F[i], G)\}$.

Every simple substructure has a root, which gives $\min_{F' \in \mathcal{S}(F)}\{\delta(F', G)\} = \min_{i \in V(F)} \min_{F' \in \mathcal{S}^i(F)}\{\delta(F', G)\}$. Next, for any fixed $i \in V(F)$, the expression $\min_{F' \in \mathcal{S}^i(F)}\{\delta(F', G)\}$ can be rewritten according to Lemma 4 as $\min_{C \in \mathcal{C}(F[i])}\{\delta(F[i] \ominus C, G)\}$, which equals $\Psi(F[i], G)$ by the definition of Ψ. Thus, we have $\min_{F' \in \mathcal{S}(F)}\{\delta(F', G)\} = \min_{i \in V(F)}\{\Psi(F[i], G)\}$. □

2.5 Preprocessing Step

As a preprocessing step to the algorithms below, we calculate and store the elements of the sets $K(F)$, $K(G)$, and $L(F)$ according to their postorders in three auxiliary arrays `KF`, `KG`, and `LF`. At the same time, $m(i)$ for all $i \in V(F) \cup V(G)$ are also precomputed and stored. More precisely, we first traverse F in left-to-right postorder

[3] In [25], the cut operation is called *removing at a node*.

[4] Observe that the cut operation differs from the delete operation defined in Sect. 2.1 since it removes *all* the nodes in a subtree of F and is for free.

to create two auxiliary arrays KF and LF which list all nodes belonging to $K(F)$ and $L(F)$, respectively, in the order that they are visited. During the traversal, we also compute and store $m(i)$ for all $i \in V(F)$ by utilizing Lemma 1(a). Next, in the same way, we traverse G to obtain an auxiliary array KG storing the nodes from $K(G)$ according to postorder as well as $m(i)$ for all $i \in V(G)$. Clearly, this preprocessing takes $O(|F| + |G|)$ time.

2.6 Traceback Step

The algorithms presented in the following sections compute the *cost* of an optimal edit mapping, but can be modified to also output a corresponding optimal edit mapping as well as a most similar subforest of F to G within the same asymptotic running time and space bounds by applying standard traceback techniques as suggested in, e.g., [25]. For example, to identify a simple substructure F' of F for which $\delta(F', G)$ is optimal (and not just compute the value $\delta(F', G)$), one may use a stronger version of Lemma 5:

Lemma 6 *For any $i^* \in V(F)$ and $C^* \in \mathcal{C}(F[i^*])$ which satisfy the two relations $\{\Psi(F[i^*], G)\} = \min_{i \in V(F)}\{\Psi(F[i], G)\}$ and $\delta(F[i^*] \ominus C^*, G) = \min_{C \in \mathcal{C}(F[i^*])}\{\delta(F[i^*] \ominus C, G)\}$, it holds that $F[i^*] \ominus C^*$ is a most similar simple substructure of F to G.*

Proof Firstly, according to the choice of i^* and C^* and the definition of Ψ, $\min_{i \in V(F)}\{\Psi(F[i], G)\} = \Psi(F[i^*], G)\} = \min_{C \in \mathcal{C}(F[i^*])}\{\delta(F[i^*] \ominus C, G)\} = \delta(F[i^*] \ominus C^*, G)$. Secondly, $\min_{i \in V(F)}\{\Psi(F[i], G)\} = \min_{F' \in \mathcal{S}(F)}\{\delta(F', G)\}$ by Lemma 5, where $\mathcal{S}(F)$ denotes the set of all simple substructures of F. Thus, $\delta(F[i^*] \ominus C^*, G) = \min_{F' \in \mathcal{S}(F)}\{\delta(F', G)\}$, so $F[i^*] \ominus C^*$ is a most similar simple substructure of F to G. □

Analogous versions of Lemma 6 for most similar sibling structures and most similar closed subforests are straightforward.

3 Review of Zhang and Shasha's Algorithm for Finding a Most Similar Simple Substructure

In this section, we review Algorithm ZS-simple by Zhang and Shasha [25] for finding a most similar simple substructure of F to G.

Algorithm ZS-simple computes $\min_{i \in V(F)}\{\Psi(F[i], G)\}$ by dynamic programming. (According to Lemma 5 above, the cost of an optimal edit mapping between G and a most similar simple substructure of F to G is given by this value.) The algorithm is listed in Fig. 4. After performing the preprocessing described in Sect. 2.5, the main loop considers all pairs of indices $i \in K(F)$ and $j \in K(G)$ in bottom-up order, and for each such pair (i, j), it calls a procedure named Compute_Psi to obtain $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for every $x \in V(F[i])$ and $y \in V(G[j])$. (In [25], the procedure Compute_Psi is named SUBTREE REMOVAL.)

```
Main loop:
Input: A target forest $F$ and a pattern forest $G$.
Output: The cost of an optimal edit mapping between $G$ and a most similar simple substructure
  of $F$ to $G$.
1: Preprocessing: Compute the arrays KF and KG and the values of $m(i)$ for all $i \in V(F) \cup V(G)$.
2: $\Psi(\emptyset, \emptyset) := 0$.
3: for $i_1 := 1, \ldots, |K(F)|$ do
4:     for $j_1 := 1, \ldots, |K(G)|$ do
5:         $i :=$ KF[$i_1$]; $j :=$ KG[$j_1$]; Call Compute_Psi($i, j$).
6:     end for
7: end for
8: return $\min_{i \in V(F)} \{\Psi(F[i], G)\}$.
```

```
Procedure Compute_Psi($i, j$):
/* Given nodes $i$ and $j$, computes $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for all $x \in V(F[i])$ and $y \in V(G[j])$. */
1: for $x := m(i), \ldots, i$ do $\Psi(F\|_{m(i):x}, \emptyset) := 0$ end for
2: for $y := m(j), \ldots, j$ do $\Psi(\emptyset, G\|_{m(j):y}) := \Psi(\emptyset, G\|_{m(j):y-1}) + \gamma(-, g(y))$ end for
3: for $x := m(i), \ldots, i$ do
4:     for $y := m(j), \ldots, j$ do
5:         Calculate $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ according to Lemma 8.
6:     end for
7: end for
```

Fig. 4 Algorithm ZS-simple from [25] for finding a most similar simple substructure of F to G

Finally, Algorithm ZS-simple returns the value $\min_{i \in V(F)}\{\Psi(F[i], G)\}$, which satisfies the relation $\min_{i \in V(F)}\{\Psi(F[i], G)\} = \min_{i \in V(F)}\{\Psi(F\|_{m(i):i}, G\|_{1:p(G)})\} = \min_{i \in V(F)}\{\Psi(F\|_{m(i):i}, G\|_{m(p(G)):p(G)})\}$ according to Lemma 1(b) and (d).

Compute_Psi uses Lemmas 7 and 8 below to compute $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ for every $x \in \{m(i), \ldots, i\}$ and $y \in \{m(j), \ldots, j\}$. Intuitively, when treating nodes x and y, if the subtree $F[x]$ is very dissimilar to $G[y]$ then it will be better to cut x (in other words, remove the entire subtree $F[x]$ at once at no additional cost), in which case $F\|_{m(i):x}$ becomes just $F\|_{m(i):m(x)-1}$. On the other hand, if $F[x]$ is similar to $G[y]$ then x and y should be linked, or one of x and y should be deleted, and then the remaining parts of $F[x]$ and $G[y]$ linked.

Lemma 7 $\Psi(\emptyset, \emptyset) = 0$; $\Psi(F, \emptyset) = 0$; $\Psi(\emptyset, G) = \sum_{j \in V(G)} \gamma(-, g(j))$.

Lemma 8 *For any $i \in V(F)$, $j \in V(G)$, $x \in \{m(i), \ldots, i\}$, $y \in \{m(j), \ldots, j\}$, the expression $\Psi(F\|_{m(i):x}, G\|_{m(j):y})$ is equal to the minimum of the following four values*:

$$\begin{cases} \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):y}); \\ \Psi(F\|_{m(i):x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \Psi(F\|_{m(i):x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \begin{cases} \Psi(F\|_{m(i):x-1}, G\|_{m(j):y-1}) \\ \quad + \gamma(f(x), g(y)), & \text{if } m(i) = m(x),\ m(j) = m(y); \\ \Psi(F\|_{m(i):m(x)-1}, G\|_{m(j):m(y)-1}) \\ \quad + \Psi(F[x], G[y]), & \text{otherwise.} \end{cases} \end{cases}$$

For proofs of Lemmas 7 and 8, see Lemma 8 in [25]. The result of [25] regarding simple substructures can be summarized as:

Theorem 1 [25] *Given two forests F and G, Algorithm* `ZS-simple` *correctly computes the forest edit distance between G and a most similar simple substructure of F to G over all simple substructures of F. Furthermore, Algorithm* `ZS-simple` *can be implemented to run in $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space.*

4 An Algorithm for Finding a Most Similar Sibling Substructure

We now give an algorithm for finding a most similar sibling substructure of F to G. It is named Algorithm `Modified_ZS-simple` and is based on Algorithm `ZS-simple` [25] described in Sect. 3 since finding a most similar sibling substructure is closely related to finding a most similar simple substructure, as shown next.

Lemma 9 *Let F' be any sibling substructure of F. F' is a most similar sibling substructure of F to G if and only if $\delta(F', G) = \min_{i \in V(F)}\{\Psi(F\|_{m(i):i-1}, G)\}$.*

Proof The proof is similar to our proof of Lemma 5. For any forest X and any node $i \in V(X)$, let $\mathcal{SB}(X)$ denote the set of all sibling substructures of X and let $\mathcal{SB}^i(X)$ be the set of sibling substructures of X whose component simple substructures' roots are children of the node i. To prove the lemma, we will show that $\min_{F' \in \mathcal{SB}(F)}\{\delta(F', G)\} = \min_{i \in V(F)}\{\Psi(F\|_{m(i):i-1}, G)\}$.

For any given sibling substructure, there exists a single node which is the parent of all of its component simple substructures, i.e., $\min_{F' \in \mathcal{SB}(F)}\{\delta(F', G)\} = \min_{i \in V(F)} \min_{F' \in \mathcal{SB}^i(F)}\{\delta(F', G)\}$. Next, for any non-leaf node $i \in V(F)$, let F_i denote the closed subforest $F[i_1 \cdot \cdot i_q]$, where i_1 and i_q are the leftmost and rightmost children of i, respectively; if i is a leaf then define $F_i = \emptyset$. Then, for any fixed $i \in V(F)$, we may rewrite the expression $\min_{F' \in \mathcal{SB}^i(F)}\{\delta(F', G)\}$ as $\min_{C \in \mathcal{C}(F_i)}\{\delta(F_i \ominus C,\ G)\}$, which equals $\Psi(F_i, G)$ by the definition of Ψ. This shows that $\min_{F' \in \mathcal{SB}(F)}\{\delta(F', G)\} = \min_{i \in V(F)}\{\Psi(F_i, G)\}$. Lastly, it follows from the postorder numbering of $V(F)$ and Lemma 1(d) that for any $i \in V(F)$, the subforest $F\|_{m(i):i-1}$ is the rooted subtree $F[i]$ with the node i removed, i.e., F_i. Therefore, $\min_{F' \in \mathcal{SB}(F)}\{\delta(F', G)\} = \min_{i \in V(F)}\{\Psi(F\|_{m(i):i-1}, G)\}$. □

According to Lemma 9, we can find a most similar sibling substructure of F to G by computing the value $\min_{i \in V(F)}\{\Psi(F\|_{m(i):i-1},\ G)\}$. In contrast, by Lemma 1(d) and Lemma 5, finding a most similar *simple* substructure amounts to computing $\min_{i \in V(F)}\{\Psi(F\|_{m(i):i}, G)\}$. Since Algorithm `ZS-simple` computes $\Psi(F\|_{m(i):i-1}, G)$ before $\Psi(F\|_{m(i):i}, G)$ for every $i \in V(F)$, Lemma 9 makes it very easy for us to find a most similar sibling substructure of F to G. To be precise, we modify Algorithm `ZS-simple` in Fig. 4 to obtain Algorithm `Modified_ZS-simple` as follows: Allocate $O(|F|)$ extra space Z to also store the values of

$\Psi(F\|_{m(i):i-1}, G)$ for all $i \in V(F)$ as they are computed; then, change Step 8 of the main loop to return $\min_{i\in V(F)}\{\Psi(F\|_{m(i):i-1}, G)\}$, found by checking Z, instead. Clearly, the asymptotic time and space complexities for Algorithm `Modified_ZS-simple` are the same as those for Algorithm `ZS-simple`.

For completeness, we now show that the value of $\Psi(F\|_{m(i):i-1}, G)$ for every $i \in V(F)$ is indeed computed, even when $i \notin K(F)$. (This is not proved explicitly in [25].) Let i be any given node in F. By the definition of the procedure `Subtree_Removal`, it holds that for each $k \in K(F)$, Algorithm `ZS-simple` computes $\Psi(F\|_{m(k):x}, G)$ for all $x \in \{m(k), \dots, k\}$. In particular, the algorithm will compute $\Psi(F\|_{m(A(i)):i-1}, G)$ when $k = A(i)$ and $x = i - 1$. By Lemma 3, this is equal to $\Psi(F\|_{m(i):i-1}, G)$. Thus, for any given $i \in V(F)$, Algorithm `ZS-simple` (and therefore also Algorithm `Modified_ZS-simple`) computes $\Psi(F\|_{m(i):i-1}, G)$ at some point during its execution.

We obtain:

Theorem 2 *Given two forests F and G, Algorithm* `Modified_ZS-simple` *correctly computes the forest edit distance between G and a most similar sibling substructure of F to G over all sibling substructures of F in $O(|F|\cdot|G|\cdot\min\{|L(F)|, dp(F)\}\cdot\min\{|L(G)|, dp(G)\})$ time and $O(|F|\cdot|G|)$ space.*

5 An Algorithm for Finding a Most Similar Closed Subforest

Here, we present an efficient algorithm for finding a most similar closed subforest of F to G. Neither Algorithm `ZS-simple` in [25] nor `Modified_ZS-simple` from the previous section is suitable for this variant of the problem because if, for example, i_1 and i_2 are siblings in F with $i_1 < i_2$ and $i_1 \in K(F)$ then the value of $\delta(F\|_{m(i_1):i_2}, G)$ is not calculated by either of the two algorithms, whereas $F\|_{m(i_1):i_2} = F[i_1 \cdot\cdot i_2]$ might in fact be a most similar closed subforest of F to G. Therefore, we have to develop another approach.

In Sect. 5.1, we first extend the results of [25] by deriving a new recurrence for computing certain values of δ that are not covered by Lemma 8. More precisely, we derive a recurrence for $\delta(F\|_{l:x}, G\|_{m(j):y})$ for any $l \in L(F)$, $j \in K(G)$, $x \in \{l, \dots, |F|\}$, and $y \in \{m(j), \dots, j\}$. In Sect. 5.2, we subsequently employ the new recurrence to obtain our algorithm named Algorithm `Most_similar_csf` for finding a most similar closed subforest of F to G based on locating a subforest of F of the form $F\|_{m(i_1):i_2}$ which minimizes $\delta(F\|_{m(i_1):i_2}, G)$ among all siblings i_1, i_2 in F. Finally, Sect. 5.3 analyzes the time and space complexity of our algorithm.

5.1 New Recurrences for δ

It is easy to show that:

Lemma 10 $\delta(\emptyset, \emptyset) = 0$; $\delta(F, \emptyset) = \sum_{i\in V(F)} \gamma(f(i), -)$; $\delta(\emptyset, G) = \sum_{j\in V(G)} \gamma(-, g(j))$.

Proof (Lemma 3 in [25]) The first case is trivial as there is no cost for the empty edit mapping between two empty forests. To prove the second case, observe that for any edit mapping M between F and $\emptyset$, the left-unlinked set contains all the nodes in F and the right-unlinked set is empty, so the cost of M is $\sum_{i \in V(F)} \gamma(f(i), -)$. The third case is symmetric to the second case. □

For the general case, we have:[5]

Lemma 11 *For any $l \in L(F)$, $j \in K(G)$, $x \in \{l, \ldots, |F|\}$, $y \in \{m(j), \ldots, j\}$, the expression $\delta(F\|_{l:x}, G\|_{m(j):y})$ is equal to the minimum of the following three values*:

$$\begin{cases} \delta(F\|_{l:x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \delta(F\|_{l:x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \begin{cases} \delta(\emptyset, G\|_{m(j):m(y)-1}) \\ \quad + \delta(F\|_{l:x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y)), & \textit{if } l \preceq x; \\ \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) \\ \quad + \delta(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y)), & \textit{if } l \not\preceq x. \end{cases} \end{cases}$$

Proof Let M be an optimal edit mapping between $F\|_{l:x}$ and $G\|_{m(j):y}$. Consider the nodes x and y. There are three main cases:

(1) $x \notin M_F$: Then x belongs to the left-unlinked set U_F of M, i.e., x is deleted in the optimal solution given by M, so $\delta(M) = \delta(F\|_{l:x-1}, G\|_{m(j):y}) + \gamma(f(x), -)$.
(2) $y \notin M_G$: Then y belongs to the right-unlinked set U_G of M, i.e., y is deleted in the optimal solution given by M, so $\delta(M) = \delta(F\|_{l:x}, G\|_{m(j):y-1}) + \gamma(-, g(y))$.
(3) $x \in M_F$ and $y \in M_G$: Then x must be linked with y in M. In this case, there are two subcases:
   (i) $l \preceq x$: This means that x is a common ancestor of all nodes in the set $l : x-1$. Furthermore, by the postordering of the nodes, y is always a common ancestor of all nodes in the set $m(y) : y-1$. Since x is linked with y, all nodes in $\{m(j), \ldots, m(y)-1\}$ must be deleted from $G\|_{m(j):y}$, so $\delta(M) = \delta(\emptyset, G\|_{m(j):m(y)-1}) + \delta(F\|_{l:x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$. See Fig. 5 for an illustration.
   (ii) $l \not\preceq x$: Since x is linked with y, the nodes $\{l, \ldots, m(x)-1\}$ are mapped to the nodes $\{m(j), \ldots, m(y)-1\}$, and the nodes $\{m(x), \ldots, x-1\}$ are similarly mapped to the nodes $\{m(y), \ldots, y-1\}$. We obtain $\delta(M) = \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) + \delta(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$. See Fig. 6.

[5]Remark: Since Algorithm `ZS-main` and Algorithm `ZS-simple` in [25] only compute values of δ of the form $\delta(F\|_{m(i):x}, G\|_{m(j):y})$, where $x \in \{m(i), \ldots, i\}$, $y \in \{m(j), \ldots, j\}$, the recurrences derived in [25] did not need to distinguish between the cases $l \preceq x$ and $l \not\preceq x$.

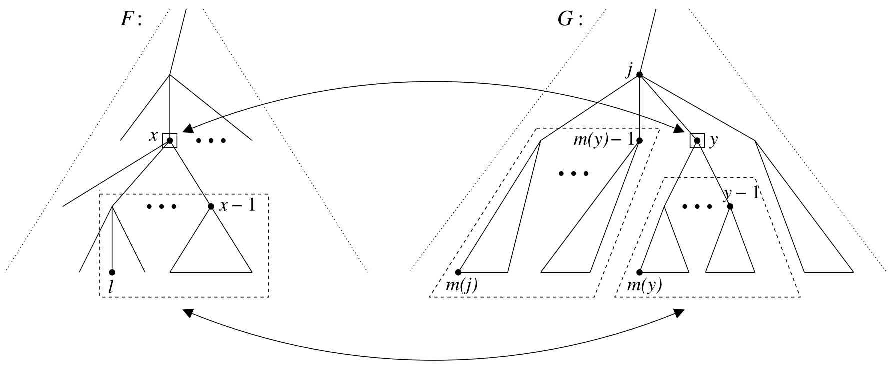

Fig. 5 Illustrating the proof of Lemma 11, subcase (i) of case (3). Here, $l \preceq x$ in F. Nodes x and y are linked, forcing nodes $\{l, \ldots, x-1\}$ in $F\|_{l:x}$ to be mapped to nodes $\{m(y), \ldots, y-1\}$ in $G\|_{m(j):y}$

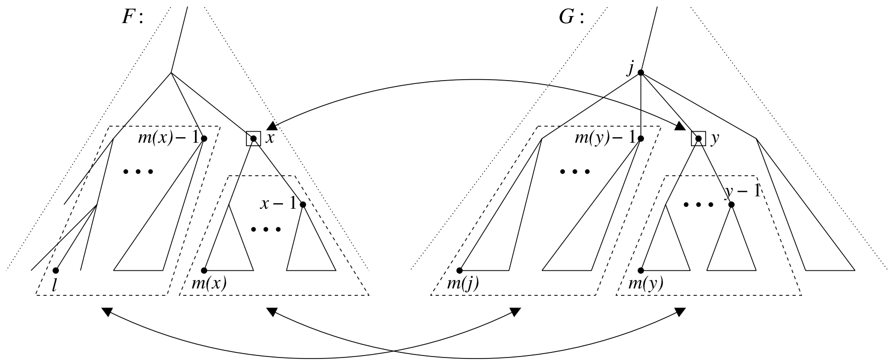

Fig. 6 Illustrating the proof of Lemma 11, subcase (ii) of case (3). In this subcase, $l \npreceq x$ in F. Nodes x and y are linked, so nodes $\{l, \ldots, m(x)-1\}$ in $F\|_{l:x}$ must be mapped to nodes $\{m(j), \ldots, m(y)-1\}$ in $G\|_{m(j):y}$, and nodes $\{m(x), \ldots, x-1\}$ in $F\|_{l:x}$ must be mapped to nodes $\{m(y), \ldots, y-1\}$ in $G\|_{m(j):y}$

Thus, $\delta(M)$ equals the minimum of the values obtained from case (1), case (2), and the appropriate subcase of case (3). $\square$

To simplify the implementation in Sect. 5.3, rewrite the recurrence in Lemma 11 as follows so that $\delta(F\|_{l:x}, G\|_{m(j):y})$ can be computed without accessing the values of $\delta(F\|_{l:x-1}, G\|_{m(y):y-1})$ and $\delta(F\|_{m(x):x-1}, G\|_{m(y):y-1})$. By eliminating the dependency on values of the form $\delta(\ldots, G\|_{\underline{\mathbf{m(y)}}:y-1})$ and $\delta(F\|_{\underline{\mathbf{m(x)}}:x-1}, \ldots)$, we also save a lot of space by not having to store those values throughout the algorithm's entire execution.

Lemma 12 *For any $l \in L(F)$, $j \in K(G)$, $x \in \{l, \dots, |F|\}$, $y \in \{m(j), \dots, j\}$, the expression $\delta(F\|_{l:x}, G\|_{m(j):y})$ is equal to the minimum of the following three values*:

$$\begin{cases} \delta(F\|_{l:x-1}, G\|_{m(j):y}) + \gamma(f(x), -); \\ \delta(F\|_{l:x}, G\|_{m(j):y-1}) + \gamma(-, g(y)); \\ \begin{cases} \delta(F\|_{l:x-1}, G\|_{m(j):y-1}) + \gamma(f(x), g(y)), & \textit{if } l \preceq x \textit{ and } m(j) = m(y); \\ \delta(\emptyset, G\|_{m(j):m(y)-1}) + \delta(F\|_{l:x}, G[y]), & \textit{if } l \preceq x \textit{ and } m(j) \neq m(y); \\ \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) + \delta(F[x], G[y]), & \textit{if } l \not\preceq x. \end{cases} \end{cases}$$

Proof We prove this lemma by showing that the new recurrence is equivalent to the one given in Lemma 11. Note that only the terms in the innermost bracket differ from Lemma 11. There are three possible cases:

(a) $l \preceq x$ and $m(j) = m(y)$: In this case, $\{m(j), \dots, m(y) - 1\} = \emptyset$ and hence $\delta(\emptyset, G\|_{m(j):m(y)-1}) = 0$ by Lemma 10. Also, the equality $m(j) = m(y)$ gives $\delta(F\|_{l:x-1}, G\|_{m(y):y-1}) = \delta(F\|_{l:x-1}, G\|_{m(j):y-1})$.

(b) $l \preceq x$ and $m(j) \neq m(y)$:
By the definition of δ,

$$\delta(F\|_{l:x}, G\|_{m(j):y}) \leq \delta(\emptyset, G\|_{m(j):m(y)-1}) + \delta(F\|_{l:x}, G[y]). \tag{1}$$

Thus, inserting the right-hand side of (1) into the minimum-expression in Lemma 11 does not affect the expression's value, so for $l \preceq x$ we can write $\delta(F\|_{l:x}, G\|_{m(j):y}) = \min\{\dots, \delta(\emptyset, G\|_{m(j):m(y)-1}) + \delta(F\|_{l:x}, G[y])\}$, where $\dots$ denotes the three terms from Lemma 11 with the third term corresponding to the condition $l \preceq x$. Now, by case (a) above, $\delta(F\|_{l:x}, G[y]) = \delta(F\|_{l:x}, G\|_{m(y):y}) \leq \delta(F\|_{l:x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$, so the third term in the min-expression is redundant and can be deleted.

(c) $l \not\preceq x$:
Here, the definition of δ implies that

$$\delta(F\|_{l:x}, G\|_{m(j):y}) \leq \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) + \delta(F[x], G[y]).$$

Therefore, when $l \not\preceq x$, the minimum-expression in Lemma 11 can be written as $\delta(F\|_{l:x}, G\|_{m(j):y}) = \min\{\dots, \delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1}) + \delta(F[x], G[y])\}$, where $\dots$ denotes the three terms from Lemma 11 with the third term corresponding to the condition $l \not\preceq x$. Again, by applying case (a) above, we have $\delta(F[x], G[y]) = \delta(F\|_{m(x):x}, G\|_{m(y):y}) \leq \delta(F\|_{m(x):x-1}, G\|_{m(y):y-1}) + \gamma(f(x), g(y))$, so the third term in the new min-expression is not needed. □

5.2 Description of Algorithm `Most_similar_csf`

We are now ready to present the main algorithm of this section, named Algorithm `Most_similar_csf`. It calculates the minimum forest edit distance among all closed subforests of F to the forest G by dynamic programming.

Algorithm `Most_similar_csf` is listed in Fig. 7. Its overall structure resembles that of Algorithm `ZS-simple` [25], but it exhibits some crucial differences

```
Main loop:
Input: A target forest $F$ and a pattern forest $G$.
Output: The cost of an optimal edit mapping between $G$ and a most similar closed subforest of $F$ to $G$.
1: Preprocessing: Compute the arrays LF and KG and the values of $m(i)$ for all $i \in V(F) \cup V(G)$.
2: $result := 0$.
3: $\delta(\emptyset, \emptyset) := 0$.
4: for $l_1 := |L(F)|, \ldots, 1$ do
5:     for $j_1 := 1, \ldots, |K(G)|$ do
6:         $l :=$ LF[$l_1$]; $j :=$ KG[$j_1$]; Call Compute_delta($l, j$).
7:     end for
8: end for
9: return $result$.
```

```
Procedure Compute_delta($l, j$):
/* Given nodes $l$ and $j$, computes $\delta(F\|_{l:x}, G\|_{m(j):y})$ for all $x \in \{l, \ldots, |F|\}$ and $y \in V(G[j])$. */
1: for $x := l, \ldots, |F|$ do $\delta(F\|_{l:x}, \emptyset) := \delta(F\|_{l:x-1}, \emptyset) + \gamma(f(x), -)$ end for
2: for $y := m(j), \ldots, j$ do $\delta(\emptyset, G\|_{m(j):y}) := \delta(\emptyset, G\|_{m(j):y-1}) + \gamma(-, g(y))$ end for
3: for $x := l, \ldots, |F|$ do
4:     for $y := m(j), \ldots, j$ do
5:         Calculate $\delta(F\|_{l:x}, G\|_{m(j):y})$ according to Lemma 12.
6:         if $j = p(G)$, $y = |G|$, $x$ has a sibling $s$ with $m(s) = l$, and $result > \delta(F\|_{l:x}, G\|_{m(j):y})$
7:         then $result := \delta(F\|_{l:x}, G\|_{m(j):y})$.
8:     end for
9: end for
```

Fig. 7 Algorithm Most_similar_csf for finding a most similar closed subforest of F to G. Note that line 4 of the main loop iterates on the leaves of F in right-to-left order

due to Lemma 12. The main loop treats all pairs of indices $l \in L(F)$ and $j \in K(G)$ in right-to-left and bottom-up order, respectively. For each such pair of indices (l, j), the algorithm calls a procedure named Compute_delta to obtain $\delta(F\|_{l:x}, G\|_{m(j):y})$ for all $x \in \{l, \ldots, |F|\}$ and $y \in \{m(j), \ldots, j\}$ based on Lemmas 10 and 12. Finally, it returns the value of $\min\{\delta(F\|_{m(i_1):i_2}, G) \mid i_1, i_2 \text{ are siblings in } F\}$.

Significantly, the main loop only considers the *leaves* of F, and moreover, the order in which they are handled is *right-to-left*. This is because such an ordering allows an implementation having low time complexity and low space complexity based on exploiting our new recurrence from Lemma 12, as described in detail in Sect. 5.3 below (cf. Lemma 13 and its proof).

A global variable named *result* keeps track of the minimum value so far among all δ of the form $\delta(F\|_{m(i_1):i_2}, G)$, where i_1 and i_2 are siblings in F, during successive calls to Compute_delta. The current value of *result* is updated in Steps 6 and 7 of Compute_delta whenever $j = p(G)$, $y = |G|$, and x has a sibling s satisfying $m(s) = l$. Here, s does not necessarily have to be a proper sibling of x, i.e., $s = x$ is allowed.

The following theorem proves the algorithm's correctness.

Theorem 3 *Given two forests F and G, Algorithm* Most_similar_csf *correctly computes the forest edit distance between G and a most similar closed subforest of F to G over all closed subforests of F.*

Proof By Lemmas 10 and 12, Algorithm Most_similar_csf calculates δ correctly for all subforests that are considered. Suppose that $F[a \cdot\cdot b]$ is a most similar closed subforest of F to G. We need to prove that the algorithm will at some point consider $F[a \cdot\cdot b]$ together with G. For this purpose, observe that $m(a) \in L(F)$ and $b \in \{m(a), \dots, |F|\}$, and $G = G\|_{m(p(G)):|G|}$ where $p(G) \in K(G)$, so when $l = m(a)$, $x = b$, $j = p(G)$, and $y = |G|$, the algorithm will compute $\delta(F\|_{l:x}, G\|_{m(j):y}) = \delta(F\|_{m(a):b}, G\|_{m(p(G)):|G|}) = \delta(F[a \cdot\cdot b], G)$ and store it in *result*. □

5.3 Computational Complexity of Algorithm `Most_similar_csf`

The structure of the expression in Lemma 12 as well as the structure of the algorithm's loops were designed to allow an implementation running in $O(|F| \cdot |G| \cdot |L(F)| \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space. In this subsection, we explain how such an implementation can be achieved.

We start by showing how to evaluate the expression in Lemma 12 efficiently.

The first point to note is that whenever Algorithm `Most_similar_csf` uses the recurrence in Lemma 12, it needs to check the descendant relationship between ℓ and x in order to know which subcase of the third case applies. This is where Lemma 2 comes in handy. After having precomputed $m(i)$ for all $i \in V(F)$ in the preprocessing step (see Sect. 2.5), it is possible to immediately test in $O(1)$ time whether or not $l \preceq x$ for any given $l \in L(F)$ and $x \in V(F)$ simply by checking if the condition $m(x) \leq l \leq x$ is true.

Secondly, we can use three tables M_1, M_2, and M_3 to store the computed optimal solutions to certain subproblems:

- M_1 temporarily stores $\delta(F\|_{l:x}, G\|_{m(j):y})$ for all $x \in \{l-1, \dots, |F|\}$ and $y \in \{m(j)-1, \dots, j\}$ as they are computed by `Compute_delta` for a given $l \in L(F)$ and a given $j \in K(G)$ inside the main loop. Each call to `Compute_delta` reuses this space.
- M_2 temporarily stores $\delta(F\|_{l:x}, G[y])$ for all $x \in \{l, \dots, |F|\}$ satisfying $l \preceq x$ and all $y \in V(G)$ as they are computed for the current $l \in L(F)$, i.e., for each fixed l in the outermost loop of the main program, M_2 keeps these δ-values during $|K(G)|$ calls to `Compute_delta`, and reuses the space when the next $l \in L(F)$ is considered.
- M_3 stores all computed values of the form $\delta(F[x], G[y])$, where $x \in V(F)$, $y \in V(G)$, during the entire execution of the algorithm.

The next lemma proves that M_1, M_2, and M_3 are sufficient for evaluating the expression in Lemma 12 in $O(1)$ time. As in Sect. 2.3, for any $j \in V(G)$, $A(j)$ denotes the nearest key node ancestor of j.

Lemma 13 *Immediately before Step* 5 *in* `Compute_delta` *is performed, every value of* δ *referred to by the expression in Lemma* 12 *will already have been computed and therefore stored in* M_1, M_2, *or* M_3.

Proof First, note that $\delta(F\|_{l:x-1}, G\|_{m(j):y})$, $\delta(F\|_{l:x}, G\|_{m(j):y-1})$, $\delta(F\|_{l:x-1}, G\|_{m(j):y-1})$, $\delta(\emptyset, G\|_{m(j):m(y)-1})$, and $\delta(F\|_{l:m(x)-1}, G\|_{m(j):m(y)-1})$ are always available in M_1 since Steps 3 and 4 in `Compute_delta` consider x and y in increasing order.

If $l \preceq x$ and $m(j) \neq m(y)$, the value of $\delta(F\|_{l:x}, G[y])$ is also needed. But this value has already been computed and stored in M_2 because $y \in \{m(j), \dots, j\}$ and $m(j) \neq m(y)$ means $m(j) < m(y)$ and thus $m(j) < m(A(y))$ by Lemma 3. This in turn implies $A(y) < j$, so the algorithm must have called `Compute_delta`$(l, A(y))$ and hence have computed $\delta(F\|_{l:x}, G\|_{m(A(y)):y}) = \delta(F\|_{l:x}, G[y])$ previously.

On the other hand, if $l \not\preceq x$ then the required value $\delta(F[x], G[y])$ is available in M_3 because $l \not\preceq x$ and $l < x$ imply $l < m(x)$ and because the outermost loop of the algorithm handles the leaves of F in *decreasing* order, so the algorithm has already called `Compute_delta`$(m(x), y)$ and obtained $\delta(F\|_{m(x):x}, G\|_{m(y):y}) = \delta(F[x], G[y])$. □

Next, we demonstrate how to do the check in Step 6 of `Compute_delta`(l, j) in $O(1)$ time.

Lemma 14 *For any fixed $l \in L(F)$, after $O(|F|)$ time preprocessing and using $O(|F|)$ extra space, one can test in $O(1)$ time if any given node in F has a sibling s with $m(s) = l$.*

Proof Define $Q_l = \{i \mid i \in V(F) \text{ and } m(i) = l\}$. By the definition of $m(i)$, Q_l induces a path in F such that every node in F can have at most one child on this path. Accordingly, for each $i \in V(F)$, define $C_l(i)$ as the unique child of i that belongs to Q_l if such a child exists, and 0 otherwise. $C_l(x)$ for all $x \in V(F)$ can easily be computed in $O(|F|)$ time: initially, set all $C_l(x)$ to 0, and then find the non-zero values by traversing F upwards from l until a node i with $m(i) \neq l$ is found. Then, to check in $O(1)$ time if a given node x in F has a sibling s such that $m(s) = l$, just check if $C_l(p(x)) \neq 0$. □

Finally, we have:

Theorem 4 *Algorithm* `Most_similar_csf` *can be implemented to run in $O(|F| \cdot |G| \cdot |L(F)| \cdot \min\{|L(G)|, dp(G)\})$ time and $O(|F| \cdot |G|)$ space.*

Proof The preprocessing in Step 1 of the main loop takes $O(|F| + |G|)$ time. We add a step between Steps 4 and 5 in the outermost **for**-loop which performs the preprocessing of Lemma 14 for the current $l \in L(F)$; summing over all $l \in L(F)$, this takes a total of $O(|F| \cdot |L(F)|)$ time. Now, according to Lemma 13 and Lemma 14, Steps 5–7 of `Compute_delta` take $O(1)$ time which means the algorithm's total running time is $O(|F| + |G|) + O(|F| \cdot |L(F)|) + O(\sum_{l \in L(F)} \sum_{j \in K(G)} |F\|_{l:|F|}| \cdot |G[j]|)$. By Lemma 7 in [25], $\sum_{j \in K(G)} |G[j]| \leq |G| \cdot \min\{|L(G)|, dp(G)\}$, so the total running time can be rewritten as $O(|F| \cdot |G| \cdot |L(F)| \cdot \min\{|L(G)|, dp(G)\})$.

The $O(|F|)$ additional space required by Lemma 14 may be reused for each $l \in L(F)$. Moreover, the tables M_1, M_2, and M_3 take $O(|F| \cdot |G|)$, $O(dp(F) \cdot |G|)$, and $O(|F| \cdot |G|)$ space, respectively, so the total space complexity is $O(|F| \cdot |G|)$. The theorem follows. □

6 Applications

The most similar subforest problem generalizes several other well-studied problems. For example, in *the forest inclusion problem*, the objective is to determine whether a given forest G can be obtained from another given forest F by only deleting nodes from F, and if so, finding the smallest subforest of F in which G is included (this problem and a constrained variant have been studied in, e.g., [14, 24]). However, in case G is *not* included in F, one might still need to find a subforest F' of F such that G is very similar to F', or to measure how far from being included in F the pattern forest G is. This is precisely "the most similar subforest problem". As another example, consider *the approximate string matching problem* [23]: Given two strings S and T, find a substring of S which is as similar as possible to T, using string edit distance. This problem has numerous applications to Stringology, Bioinformatics, Signal processing, Speech recognition, Text retrieval, e-mail spam filtering, Image compression, etc. (see, e.g., [6, 9, 18, 19, 23] and the many references therein). Since any string S can be represented by a tree where all non-leaf nodes have exactly one child and every connected subgraph corresponds to a substring of S, the approximate string matching problem is just a special case of the most similar subforest problem.

Next, we describe some potential applications for algorithms that compute most similar subforests.

- *Subforests in general*: When investigating post-transcriptional gene regulation events or evolutionary relationships between RNA molecules, it is useful to look for substructures of RNA molecules' secondary structures called *motifs* that represent important functional or structural regions [10]. A number of methods of representing the secondary structure of an RNA molecule without pseudoknots by a forest have been proposed in the literature [10, 13, 20]. In such forest representations, motifs correspond to subforests [8, 10, 12], and moreover, the functional similarity of two RNA secondary structures is related to the similarity of their two forest representations [10, 13, 17, 20]. Hence, our algorithms may help in the automatic identification of motifs in RNA secondary structures as follows: to classify a newly discovered RNA secondary structure F, our algorithms can be applied to find several different types of subforests (candidate functional regions) of F which are similar to the functional regions of various known families, and then one can filter out those families which are too unlike F by using some other method. This could be a useful step in RNA structure comparison and classification.
- *Sibling substructures*: The ability to detect changes in electronic documents and hierarchically structured data such as XML files is critical for information management and data archiving applications. Often, the entire history of modifications made to a data file is unavailable but snapshots of previous versions of the file can be obtained and then analyzed and compared to the current version [3]. Suppose that a hierarchically structured data file is represented by a tree and we need to check if a given pattern forest G extracted from an old version of the file occurs somewhere in the current version. Then, algorithms for finding a most similar sibling substructure could be used to search for G when not only the node labels of G itself may have changed but also many extra nodes may have been inserted into the part of the file where G originally occurred.

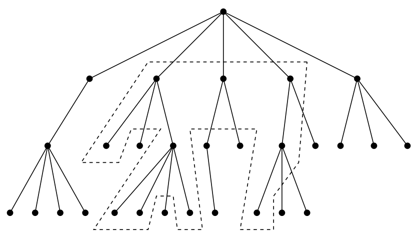
Fig. 8 A gapped subforest of a given forest F

– *Closed subforests*: By comparing various subtrees or subforests of an ordered labeled tree representing a computer program, one can locate fragments of the source code which are identical or nearly identical to each other. Detecting and replacing such "clones" by, e.g., subroutines or macros can improve the structure of (and thus decrease the maintenance costs of) software [1]. Our algorithms may be helpful here since a section of the code that is to be replaced by a subroutine or macro would typically consist of a contiguous code fragment and also start and end on the same nesting level, and thus correspond to a closed subforest.

7 Concluding Remarks

It is straightforward to generalize the algorithms presented here to find a subforest F' of F and a subforest G' of G that are the most similar for any combination of the types of subforests considered above. For example, if F' should be a closed subforest and G' should be a simple substructure then we can modify Algorithm `Most_similar_csf` to allow nodes in G to be cut as in Sect. 3.

We wonder if it is possible to reduce the running time of `Most_similar_csf` from $O(|F| \cdot |G| \cdot |L(F)| \cdot \min\{|L(G)|, dp(G)\})$ to $O(|F| \cdot |G| \cdot \min\{|L(F)|, dp(F)\} \cdot \min\{|L(G)|, dp(G)\})$ while keeping the $O(|F| \cdot |G|)$ space requirement. Such a result would match the complexities of the other three algorithms `ZS-main`, `ZS-simple`, and `Modified_ZS-simple`.

Another important open question is: Is it possible to extend the algorithms in this article to other types of subforests? For example, one might consider *gapped subforests* (introduced in [12]), where a gapped subforest of F is obtained by removing from any closed subforest F' of F a set C of closed subforests such that no two closed subforests in C have the same parent in F' (see Fig. 8). One may also consider subforests of F formed by successively applying the delete operation on the rightmost root of F any number of times and then on the leftmost root any number of times;[6] this type of subforest was called an (i, j)*-deleted subforest of* F in [2], and it includes closed subforests as a special case.

Furthermore, it could be useful in practical applications to have a candidate set of nearly optimal solutions which may then be further evaluated by some additional criteria. Therefore, it would be interesting to generalize the algorithms to return *all*

[6]Klein's algorithm in its simplified form [2, 15] may be used here. See Sect. 1.1 for a short discussion.

subforests of F which are within forest edit distance k of G, where k is an input parameter. Indeed, much work on the analogous approximate string matching problem mentioned in Sect. 6 has focused on variants where this kind of threshold parameter is provided [9, 18, 23].

An alternative measure of the similarity between two forests is the *forest alignment distance* (see [13] or [16] for a formal definition). Although the edit distance and alignment distance are equivalent for *strings*, they are not equivalent for trees and forests [13, 16]. The algorithm of Jiang et al. [13] for computing the forest alignment distance runs in $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G))^2)$ time, and its running time was improved for similar inputs in [11]. The algorithm of Jiang et al. computes an optimal *global* alignment between F and G, meaning that all nodes of F and G contribute to the cost of the final solution. Later, Höchsmann et al. [10] gave an algorithm for computing an optimal *local* alignment between F and G which finds a closed subforest F' of F and a closed subforest G' of G having the minimum forest alignment distance. A more efficient algorithm for this problem, running in $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G))^2)$ time and $O(|F| \cdot |G| \cdot (\deg(F) + \deg(G)))$ space, along with some extensions to other types of subforests were given in [12]. Höchsmann et al. [10] also considered the problem of finding a closed subforest F' of F which minimizes the alignment distance to G (i.e., the analogue of our "most similar closed subforest problem" but using alignment distance instead of edit distance), which they called *the small-in-large closed subforest similarity problem*, and showed how to solve it in $O(|F| \cdot |G| \cdot \deg(F) \cdot \deg(G) \cdot (\deg(F) + \deg(G)))$ time and $O(|F| \cdot |G| \cdot \deg(F) \cdot \deg(G))$ space. It remains to further reduce the time and space complexities of their algorithm.

Acknowledgements We thank the anonymous referees whose suggestions helped us improve the presentation of our results.

References

1. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1998), pp. 368–377 (1998)
2. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. **337**(1–3), 217–239 (2005)
3. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1996), pp. 493–504 (1996)
4. Chen, W.: New algorithm for ordered tree-to-tree correction problem. J. Algorithms **40**(2), 135–158 (2001)
5. Cobéna, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE 2002), pp. 41–52 (2002)
6. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, London (1994)
7. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. ACM Trans. Algorithms **6**(1) (2009). Article 2
8. Griffiths-Jones, S.: The microRNA registry. Nucleic Acids Res. **32**, D109–D111 (2004)
9. Grossi, R., Luccio, F.: Simple and efficient string matching with k mismatches. Inf. Process. Lett. **33**(3), 113–120 (1989)
10. Höchsmann, M., Töller, T., Giegerich, R., Kurtz, S.: Local similarity in RNA secondary structures. In: Proceedings of the IEEE Computational Systems Bioinformatics Conference (CSB 2003), pp. 159–168 (2003)

11. Jansson, J., Lingas, A.: A fast algorithm for optimal alignment between similar ordered trees. Fundam. Inf. **56**(1–2), 105–120 (2003)
12. Jansson, J., Ngo, T.H., Sung, W.-K.: Local gapped subforest alignment and its application in finding RNA structural motifs. J. Comput. Biol. **13**(3), 702–718 (2006)
13. Jiang, T., Wang, L., Zhang, K.: Alignment of trees—an alternative to tree edit. Theor. Comput. Sci. **143**, 137–148 (1995)
14. Kilpeläinen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM J. Comput. **24**(2), 340–356 (1995)
15. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Proceedings of the 6th European Symposium on Algorithms (ESA 1998), pp. 91–102 (1998),
16. Kuboyama, T., Shin, K., Miyahara, T., Yasuda, H.: A theoretical analysis of alignment and edit problems for trees. In: Proceedings of the 9th Italian Conference on Theoretical Computer Science (ICTCS 2005). Lecture Notes in Computer Science, vol. 3701, pp. 323–337. Springer, Berlin, Heidelberg (2005)
17. Ma, B., Wang, L., Zhang, K.: Computing similarity between RNA structures. Theor. Comput. Sci. **276**, 111–132 (2002)
18. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
19. Sankoff, D., Kruskal, J.B. (eds.): Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, Reading (1983)
20. Shapiro, B.A., Zhang, K.: Comparing multiple RNA secondary structures using tree comparisons. Comput. Appl. Biosci. **6**(4), 309–318 (1990)
21. Tai, K.-C.: The tree-to-tree correction problem. J. ACM **26**(3), 422–433 (1979)
22. Touzet, H.: Comparing similar ordered trees in linear-time. J. Discrete Algorithms **5**(4), 696–705 (2007)
23. Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. Theor. Comput. Sci. **92**(1), 191–211 (1992)
24. Valiente, G.: Constrained tree inclusion. J. Discrete Algorithms **3**(2–4), 431–447 (2005)
25. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6), 1245–1262 (1989)