Theoretical Computer Science 410 (2009) 4303–4314

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Fast algorithms for computing tree LCS

Shay Mozes [a], Dekel Tsur [b,*], Oren Weimann [c], Michal Ziv-Ukelson [b]

[a] *Brown University, Providence, RI 02912-1910, USA*
[b] *Ben-Gurion University of the Negev, Be'er Sheva, Israel*
[c] *Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

ARTICLE INFO

Keywords:
Tree LCS
Tree edit distance
Ordered trees
Largest common subforest
Sparse dynamic programming

ABSTRACT

The LCS of two rooted, ordered, and labeled trees F and G is the largest forest that can be obtained from both trees by deleting nodes. We present algorithms for computing tree LCS which exploit the *sparsity* inherent to the tree LCS problem. Assuming G is smaller than F, our first algorithm runs in time $O(r \cdot \text{height}(F) \cdot \text{height}(G) \cdot \lg\lg|G|)$, where r is the number of pairs $(v \in F, w \in G)$ such that v and w have the same label. Our second algorithm runs in time $O(Lr\lg r \cdot \lg\lg|G|)$, where L is the size of the LCS of F and G. For this algorithm we present a novel three-dimensional alignment graph. Our third algorithm is intended for the constrained variant of the problem in which only nodes with zero or one children can be deleted. For this case we obtain an $O(rh\lg\lg|G|)$ time algorithm, where $h = \text{height}(F) + \text{height}(G)$.

1. Introduction

The *longest common subsequence* (LCS) of two strings is the longest subsequence of symbols that appears in both strings. The *edit distance* of two strings is the minimal number of character deletions insertions and replacements required to transform one string into the other. Computing the LCS or the edit distance can be done using similar dynamic programming algorithms in $O(mn)$ time and space, where m and n $(m \leq n)$ are the lengths of the strings [14,27]. The only known speedups to the edit distance algorithm are by polylogarithmic factors [6,10,21]. For the LCS problem however, it is possible to obtain time complexities better than $\widetilde{O}(mn)$ in favorable cases, e.g. [3,9,15–17,23]. This is achieved by exploiting the sparsity inherent to the LCS problem and measuring the complexity by parameters other than the lengths of the input strings. In this paper, we apply this idea to computing the LCS of rooted, ordered, and labeled trees.

The problem of computing string LCS translates to finding a longest chain of matches in the alignment graph of the two strings. Many string LCS algorithms that construct such chains by exploiting sparsity have their natural predecessors in either Hirschberg [15] or Hunt and Szymanski [17]. Given two strings S and T, let L denote the size of their LCS and let r denote the number of matches in the alignment graph of S and T. Hirschberg's algorithm achieves an $O(nL + n\lg|\Sigma|)$ time complexity by computing chains in succession. The Hunt–Szymanski algorithm achieves an $O(r\lg m)$ time complexity by extending partial chains. The latter can be improved to $O(r\lg\lg m)$ by using the successor data-structure of van Emde Boas [26]. Apostolico and Guerra [3] gave an $O(mL \cdot \min(\lg|\Sigma|, \lg m, \lg\frac{2n}{m}))$ time algorithm, and another algorithm with running time $O(m\lg n + d\lg\frac{nm}{d})$ which can also be implemented to take $O(d\lg\lg\min(d, \frac{nm}{d}))$ time [12]. Here, $d \leq r$ is the number of dominant matches (as defined by Hirschberg [15]). Note that in the worst case both d and r are $\Theta(nm)$, while the parameter L is always bounded by m. When there are $k \geq 2$ input strings, the sparse LCS problem extends to the problem

* Corresponding author. Tel.: +972 8 6479831; fax: +972 8 6477650.
E-mail address: dekelts@cs.bgu.ac.il (D. Tsur).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.07.011

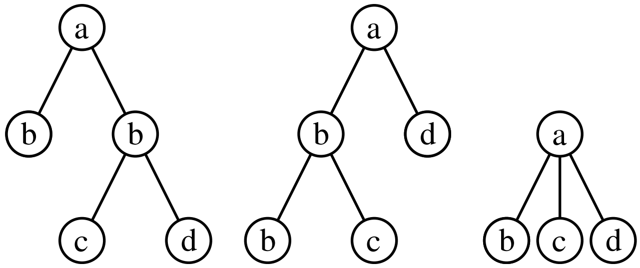


Fig. 1. Two rooted trees F and G and their largest common subforest.

of chaining from fragments in multiple dimensions [1,22]. Here, the match point arithmetic is extended with range search techniques, yielding a running time of $O(r(\lg n)^{k-2} \lg\lg n)$.

The problem of computing the LCS of two trees was considered by Lozano et al. [20] and Amir et al. [2]. The problem is defined as follows.

Definition 1 (*Tree LCS*). The LCS of two rooted, ordered, labeled trees, is the size of the largest forest that can be obtained from both trees by deleting nodes. Deleting a node v means removing v and all edges incident to v. The children of v become children of the parent of v (if it exists) instead of v (see Fig. 1).

We also consider the following constrained variant of the problem.

Definition 2 (*Homeomorphic Tree LCS*). The Homeomorphic LCS (HLCS) of two rooted, ordered, labeled trees is the size of the largest tree that can be obtained from both trees by deleting nodes, such that in the series of node deletions, a deleted node must have 0 or 1 children at the time the deletion is applied.

Tree LCS is a popular metric for measuring the similarity of two trees and arises in XML comparisons, computer vision, compiler optimization, natural language processing, and computational biology [5,7,19,28,24]. To date, computing the LCS of two trees is done by using *tree edit distance* algorithms. Tai [24] gave the first such algorithm with a time complexity of $O(nm \cdot \text{leaves}(F)^2 \cdot \text{leaves}(G)^2)$, where n and m are the sizes of the input trees F and G (with $m \leq n$) and leaves(F) denotes the number of leaves in F. Zhang and Shasha [28] improved this result to $O(nm \cdot \text{cdepth}(F) \cdot \text{cdepth}(G))$, where cdepth$(F)$ is the minimum between height(F) (the height of F) and leaves(F). In the worst case, their algorithm runs in $O(n^2m^2) = O(n^4)$ time. Klein [18] improved this result to a worst-case $O(m^2 n \lg n) = O(n^3 \lg n)$ time algorithm and Demaine et al. [11] further improved to $O(nm^2(1+\lg\frac{n}{m})) = O(n^3)$. Chen [8] gave an $O(nm + n \cdot \text{leaves}(G)^2 + \text{leaves}(F) \cdot M(\text{leaves}(G)))$ time algorithm, where $M(k)$ is the time complexity for computing the distance product of two $k \times k$ matrices. For homeomorphic edit distance (where deletions are restricted to nodes with zero or one child), Amir et al. [2] gave an $O(mn)$ time algorithm.

Our results. We modify Zhang and Shasha's algorithms and Klein's algorithm similarly to the modifications of Hunt-Szymanski and Hirschberg to the classical $O(mn)$-time algorithm for string LCS. We present two algorithms for computing the LCS of two rooted, ordered, and labeled trees F and G of sizes n and m. Our first algorithm runs in time $O(r \cdot \text{height}(F) \cdot \text{height}(G) \cdot \lg\lg m)$ where r is the number of pairs $(v \in F, w \in G)$ such that v and w have the same label. Our second algorithm runs in time $O(Lr \lg r \cdot \lg\lg m)$, where $L = |\text{LCS}(F, G)|$. This algorithm is more complicated and requires a novel three-dimensional alignment graph. In both these algorithms the $\lg\lg m$ factor can be replaced by $\lg\lg(\min(m, r))$ by noticing that if $r < m$ then there are at least $m - r$ nodes in G that do not match any node in F so we can delete them from G and solve the problem on the new G whose size is now at most r. Finally we consider LCS for the case when only homeomorphic mappings are allowed between the compared trees (i.e. deletions are restricted to nodes with zero or one child). For this case we obtain an $O(rh \lg\lg m)$ time algorithm, where $h = \text{height}(F) + \text{height}(G)$. A comparison between previous results and our results is given in Table 1.

Roadmap. The rest of the paper is organized as follows. Preliminaries and definitions are given in Section 2. In Section 3 we present our sparse variant of the Zhang–Shasha algorithm and in Sections 4 and 5 we give such variants for Klein's algorithm. Finally, in Section 6 we describe our algorithm for the homeomorphic tree LCS.

2. Preliminaries

For a forest F, the node set of F is written simply as F, as when we speak of a node $v \in F$. We denote F_v as the subtree of F that contains the node $v \in F$ and all its descendants. A forest obtained from F by deleting nodes is called a *subforest* of F. For a pair of trees F, G, two nodes $v \in F$, $w \in G$ with the same label are called a *match pair*. For the tree LCS problem we assume without loss of generality that the roots of the two input trees form a match pair (if this property does not hold for the two input trees, we can add new roots to the trees and solve the tree LCS problem on the new trees).

Table 1
Comparison between previous results and our results. Note that $r \le nm$ and $Lr \le nm^2$.

Previous results	Our results
Tree LCS	
$O(nm \cdot \text{cdepth}(F) \cdot \text{cdepth}(G))$ [28]	$O(r \cdot \text{height}(F) \cdot \text{height}(G) \cdot \lg\lg m)$
$O(nm^2(1 + \lg \frac{n}{m}))$ [11]	$O(Lr \lg r \cdot \lg\lg m)$
Homeomorphic Tree LCS	
$O(nm)$ [2]	$O(rh \lg\lg m)$

The *Euler string* of a tree F is the string obtained when performing a left-to-right DFS traversal of F and writing down the label of each node twice: when the DFS traversal first enters the node and when it last leaves the node. We define $e_F(i)$ to be the index such that both the ith and $e_F(i)$th characters of the Euler string of F were generated from the same node of F. Note that $e_F(e_F(i)) = i$.

For $i \le j$, we denote by $F[i..j]$ the forest induced by all nodes $v \in F$ whose Euler string indices *both* lie between i and j. A *left-to-right postorder traversal* of a tree F whose root v has children $v_1, v_2, \ldots, v_k$ (ordered from left to right) is a traversal which recursively visits $F_{v_1}, F_{v_2}, \ldots, F_{v_k}$, then finally visits v. The postorder traversal of a forest F is a traversal composed of postorder traversals of the trees of F, visited from left to right.

The tree LCS problem can be formulated in terms of matchings. Let F and G be two forests. We say that a set $M \subseteq V(F) \times V(G)$ is an *LCS matching* between F and G if

1. M is a matching, namely every $v \in F$ appears in at most one pair of M and every $v \in G$ appears in at most one pair.
2. For every $(v, v') \in M$, $\text{label}(v) = \text{label}(v')$.
3. For every $(v, v'), (w, w') \in M$, v is an ancestor of w if and only if v' is an ancestor of w'.
4. For every $(v, v'), (w, w') \in M$, v appears before w in the postorder traversal of F if and only if v' appears before w' in the postorder traversal of G.

An LCS matching M between F and G corresponds to a common subforest of F and G of size $|M|$, and vice versa.

For two forests F and G, let $\text{LCS}_R(F, G)$ (resp., $\text{LCS}_L(F, G)$) denote the size of the largest forest that can be obtained from F and G by node deletions without deleting the root of the rightmost (resp., leftmost) tree in F or G. If the roots of the rightmost trees in F and G are not a match pair then we define $\text{LCS}_R(F, G) = 0$. Clearly, $\text{LCS}_R(F, G) \le \text{LCS}(F, G)$ and $\text{LCS}_L(F, G) \le \text{LCS}(F, G)$.

Lemma 1. *If F and G are trees whose roots have equal labels then* $\text{LCS}_R(F, G) = \text{LCS}_L(F, G) = \text{LCS}(F, G)$.

Proof. Let r and r' be the roots of F and G, respectively. We need to show that there is an LCS matching between F and G of size $\text{LCS}(F, G)$ in which both r and r' are matched. Let M be an LCS matching between F and G of size $\text{LCS}(F, G)$. If r and r' are matched in M we are done. Moreover, we cannot have that both r and r' are not matched in M since in this case $M' = M \cup \{(r, r')\}$ is an LCS matching between F and G of size $\text{LCS}(F, G) + 1$, a contradiction.

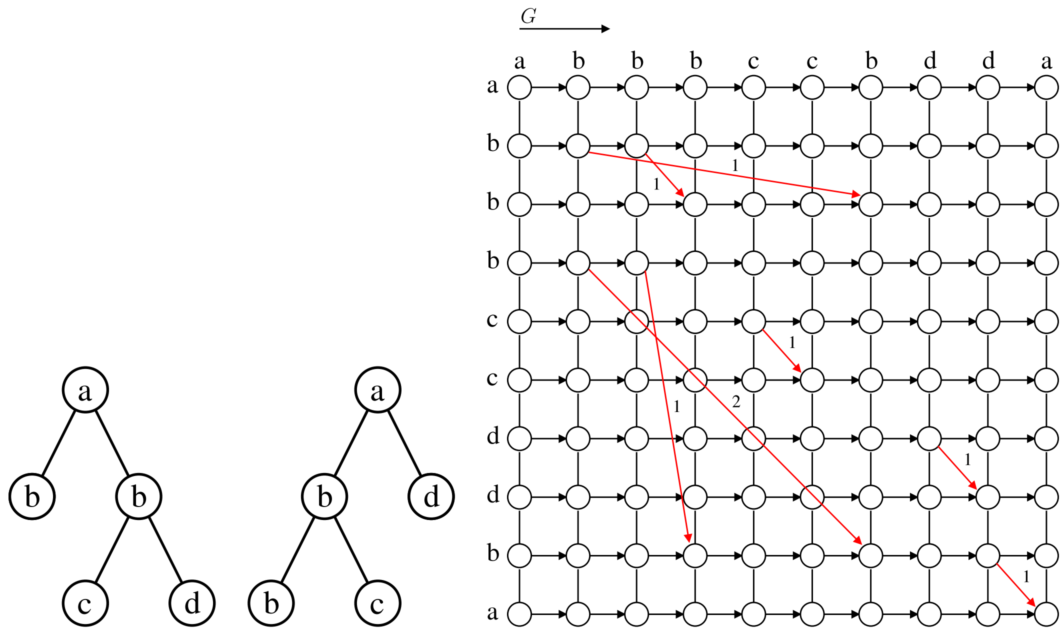
Now, assume w.l.o.g. that r is not matched in M and r' is matched. Let v be the vertex in F that is matched to r' in M. Then, $M' = M \cup \{(r, r')\} \setminus \{(v, r')\}$ is an LCS matching between F and G with size $\text{LCS}(F, G)$. ∎

A *path decomposition* of a tree F is a set of disjoint paths in F such that (1) each path ends in a leaf, and (2) each node appears in exactly one path. The *main path* of F with respect to a decomposition $\mathcal{P}$ is the path in $\mathcal{P}$ that contains the root of F. A *heavy path decomposition* of a tree F was introduced by Harel and Tarjan [13] and is built as follows. We classify each node of F as either *heavy* or *light*: For each node v we pick the child of v with maximum number of descendants and classify it as *heavy* (ties are resolved arbitrarily). The remaining nodes are classified as *light*. The *main* path p of the heavy path decomposition starts at the root (which is light), and at each step moves from the current node v to its heavy child. We next remove the nodes of p from F, and recursively compute a heavy path decomposition for each of the remaining trees. An important property of this decomposition is that the number of light ancestors of a node $v \in F$ is at most $\lg n + 1$.

A *successor data-structure* is a data-structure that stores a set of elements S with a key for each element and supports the following operations: (1) $\text{insert}(S, x)$: inserts x into S (2) $\text{delete}(S, x)$: removes x from S (3) $\text{pred}(S, k)$: returns the element $x \in S$ with maximal key such that $\text{key}(x) \le k$ (4) $\text{succ}(S, k)$: returns the element $x \in S$ with minimal key such that $\text{key}(x) \ge k$. Van Emde Boas presented a data-structure [26] that supports each of these operations in $O(\lg\lg u)$ time, where the set of legal keys is $\{1, 2, \ldots, u\}$.

3. An $O(r \cdot \text{height}(F) \cdot \text{height}(G) \cdot \lg\lg m)$ algorithm

In this section we present an $O(r \cdot \text{height}(F) \cdot \text{height}(G) \cdot \lg\lg m)$ time algorithm for computing the LCS of two trees F and G of sizes n and m and heights $\text{height}(F)$ and $\text{height}(G)$ respectively. The relation between this algorithm and Zhang and Shasha's $O(nm \cdot \text{height}(F) \cdot \text{height}(G))$ time algorithm [28] is similar to the relation between Hunt and Szymanski's $O(r \lg\lg m)$ time algorithm [17] and Wagner and Fischer's $O(mn)$ time algorithm [27] in the string LCS world.



(a) F (b) G (c) Alignment graph

Fig. 2. An alignment graph of two trees F and G. The horizontal and vertical edges are of weight 0. Every diagonal edge corresponds to a match pair, and the weight of the edge is the LCS of the corresponding subtrees of F and G.

We describe an algorithm based on that of Zhang and Shasha using an alignment graph. This approach was also used in [25,4]. The *alignment graph* $B_{F,G}$ of F and G is an edge-weighted directed graph defined as follows. The vertices of $B_{F,G}$ are (i, j) for $1 \le i \le 2n$ and $1 \le j \le 2m$. Intuitively, vertex (i, j) corresponds to $\mathrm{LCS}(F[1..i], G[1..j])$, and edges in the alignment graph correspond to edit operations. The graph has the following edges:

1. Edges $(i - 1, j) \to (i, j)$ and $(i, j - 1) \to (i, j)$ with weight 0 for every i and j. These edges either connect vertices which represent the same pair of forests, or represent deletion of the rightmost root of just one of the forests. Both cases do not change the LCS, hence the zero weight we assign to these edges.
2. An edge for every match pair $v \in F$, $w \in G$, except for the roots of F and G. Let i and $e_F(i)$ be the two characters of the Euler string of F that correspond to v, where $e_F(i) < i$, and let $e_G(j) < j$ be the two characters of the Euler string of G that correspond to w. We add an edge $(e_F(i), e_G(j)) \to (i, j)$ with weight $\mathrm{LCS}(F_v, G_w)$ to $B_{F,G}$. This edge corresponds to matching the rightmost trees of $F[1..i]$ and $G[1..j]$ and its weight is obtained by recursively applying the algorithm on the trees F_v and G_w. Note that we cannot add an edge of this type for the match pair of the roots of F and G because we cannot compute the weight of such edge by recursion.
3. An edge $(2n - 1, 2m - 1) \to (2n, 2m)$ with weight 1, which corresponds to the match between the roots of F and G.

See Fig. 2 for an example. For an edge $e = (i, j) \to (i', j')$, let $\mathrm{tail}(e) = (i, j)$ and $\mathrm{head}(e) = (i', j')$. The ith coordinate of a vector x is denoted by x_i. For example, for e above, $\mathrm{head}(e)_2 = j'$.

Lemma 2. *The maximum weight of a path in $B_{F,G}$ from vertex* $(1, 1)$ *to vertex* (i, j) *is equal to* $\mathrm{LCS}(F[1..i], G[1..j])$.

Proof. We prove the lemma using induction on $i + j$. The base of the induction (when $i = j = 0$) is trivially true. Consider some i and j. Let v and w be the vertices that generate locations i and j in the Euler strings of F and G, respectively.

Let p be a path from $(1, 1)$ to (i, j) of maximum weight. We first show that there is an LCS matching M between $F[1..i]$ and $G[1..j]$ of size at least $\mathrm{weight}(p)$. Let $e = (i', j') \to (i, j)$ be the last edge on p. Denote by p' the prefix of p up to but not including e. From the construction of the graph we have that $i' + j' < i + j$, so by the induction hypothesis, $\mathrm{weight}(p') \le \mathrm{LCS}(F[1..i'], G[1..j'])$. Therefore, there is an LCS mapping M' between $F[1..i']$ and $G[1..j']$ of size $\mathrm{weight}(p')$. There are three cases, depending on the type of e.

1. If e is an edge of the first type above, then $\mathrm{weight}(e) = 0$, and $M = M'$ is the desired matching (note that $F[1..i']$ and $G[1..j']$ are subforests of $F[1..i]$ and $G[1..j]$, respectively, so M' is also an LCS matching between $F[1..i]$ and $G[1..j]$).
2. If e is an edge of the second type above then $i' = e_F(i)$ and $j' = e_G(j)$. Let M'' be an LCS matching between F_v and G_w of size $\mathrm{LCS}(F_v, G_w)$. By construction, $\mathrm{weight}(e) = \mathrm{LCS}(F_v, G_w)$. The forest $F[1..i]$ is the disjoint union of the forests $F[1..i']$ and F_v (as $i' = e_F(i)$), and F_v is the rightmost tree in $F[1..i]$. Similarly, $G[1..j]$ is the disjoint union of the forests $G[1..j']$ and G_w, and G_w is the rightmost tree in $G[1..j]$. Therefore, $M = M' \cup M''$ is an LCS mapping between $F[1..i]$ and $G[1..j]$ of size $\mathrm{weight}(p') + \mathrm{weight}(e) = \mathrm{weight}(p)$.

3. If e is of the third type above then v and w are the roots of F and G, respectively. Hence, $M = M' \cup \{(v, w)\}$ is an LCS mapping between $F[1..i]$ and $G[1..j]$ of size $\text{weight}(p') + 1 = \text{weight}(p)$.

We next prove the opposite direction. Let M be an LCS mapping between $F[1..i]$ and $G[1..j]$ of maximum size. We will show that there is a path p from $(1, 1)$ to (i, j) with weight at least $|M|$. If v is not matched in M then M is an LCS matching between $F[1..i-1]$ and $G[1..j]$. By, induction, there is a path p' from $(1, 1)$ to $(i-1, j)$ of weight at least $|M|$. Since there is an edge $(i-1, j) \to (i, j)$ in $B_{F,G}$, we obtain that there is a path from $(1, 1)$ to (i, j) of weight at least $|M|$. The same argument holds if w is not matched in M. Suppose, therefore, that both v and w are matched in M. We have that $e_F(i) < i$ (otherwise v is not a vertex of $F[1..i]$ so it cannot be matched in M) and $e_G(j) < j$. Moreover, v and w are the last vertices in the postorders of $F[1..i]$ and $G[1..j]$, respectively, so v must be matched to w. If $(i, j) \neq (2n, 2m)$, then $M'' = M \cap (V(F_v) \times V(G_w))$ is an LCS matching between F_v and G_w, and $M' = M \setminus M''$ is an LCS matching between $F[1..i] - F_v = F[1..e_F(i)]$ and $G[1..j] - G_w = G[1..e_G(j)]$. By induction, there is a path p' from $(1, 1)$ to $(e_F(i), e_G(j))$ of weight at least $|M'|$. Therefore, there is a path from $(1, 1)$ to (i, j) with weight at least $|M'| + \text{LCS}(F_v, G_w) \geq |M'| + |M''| = |M|$. Finally, if $(i, j) = (2n, 2m)$ then $M' = M \setminus \{(v, w)\}$ is an LCS matching between $F[1..i-1]$ and $G[1..j-1]$. By induction there is a path p' from $(1, 1)$ to $(i-1, j-1)$ of weight at least $|M'|$, so there is a path from $(1, 1)$ to (i, j) of weight at least $|M'| + 1 = |M|$. ∎

Zhang and Shasha's algorithm computes the maximum weight of a path from $(1, 1)$ to (i, j), for every vertex (i, j) of $B_{F,G}$. By Lemma 2, this gives $\text{LCS}(F, G)$ at the vertex $(2n, 2m)$. If there are only few match pairs, we can do better. Denote the set of edges in $B_{F,G}$ with nonzero weights by $E_{F,G}$. Clearly, $|E_{F,G}| = r$. We will exploit the sparsity of the edges $E_{F,G}$ by ignoring the edges with weight 0 and the vertices that are not the endpoint of an edge in $E_{F,G}$. We define the *score* of $e \in E_{F,G}$ as the maximum weight of a path in $B_{F,G}$ from $(1, 1)$ to $\text{head}(e)$ that passes through e.

Lemma 3. $\text{score}(e) = \text{LCS}_R(F[1..\text{head}(e)_1], G[1..\text{head}(e)_2])$ *for every edge* $e \in E_{F,G}$.

Proof. Fix $e \in E_{F,G}$, and let (v, w) be the corresponding match pair. If v, w are the roots of F, G, respectively, then $F[1..\text{head}(e)_1] = F$ and $G[1..\text{head}(e)_2] = G$. Furthermore, by Lemmas 1 and 2, $\text{score}(e) \leq \text{LCS}(F, G) = \text{LCS}_R(F, G)$. Otherwise, following the proof of Lemma 2, we have that for every maximum weight path p from $(1, 1)$ to $\text{head}(e)$ which passes through e, there is an LCS matching $M = M' \cup M''$ between $F[1..\text{head}(e)_1]$ and $G[1..\text{head}(e)_2]$ whose size is equal to $\text{weight}(p)$. Furthermore, M'' is an LCS matching between F_v and G_w of size $\text{LCS}(F_v, G_w)$. By Lemma 1, we may assume that v is matched to w in M''. It follows that $\text{score}(e) = |M| \leq \text{LCS}_R(F[1..\text{head}(e)_1], G[1..\text{head}(e)_2])$.

In the opposite direction, let M be a matching between $F[1..\text{head}(e)_1]$ and $G[1..\text{head}(e)_2]$ of size $\text{LCS}_R(F[1..\text{head}(e)_1],)$ $G[1..\text{head}(e)_2]$ such that $(v, w) \in M$. Following the proof of Lemma 2 we define a path p from $(1, 1)$ to $\text{head}(e)$ with weight at least $|M|$. Since $(v, w) \in M$, it follows that p passes through e. Therefore, $\text{score}(e) \geq \text{LCS}_R(F[1..\text{head}(e)_1], G[1..]$ $\text{head}(e)_2)$. ∎

By Lemmas 1 and 3 we have that $\text{LCS}(F, G) = \text{score}((2n-1, 2m-1) \to (2n, 2m))$. We now describe a procedure, called ComputeScores, that computes $\text{LCS}(F, G)$ in $O(|E_{F,G}| \cdot \lg\lg m)$ time, assuming we have already computed $\text{LCS}(F_v, G_w)$ for every match pair $v \in F$, $w \in G$ except for the roots of F and G. This procedure computes $\text{score}(e)$ for every $e \in E_{F,G}$. It uses a successor data-structure S that stores edges from $E_{F,G}$, where the key of an edge e is $\text{head}(e)_2$. The pseudocode for procedure ComputeScores is as follows (we assume that $\text{score}(\text{NULL}) = 0$).

Procedure ComputeScores

```
1: for i = 1, ..., 2n do
2:   for every e ∈ E_{F,G} with tail(e)_1 = i do
3:     score(e) ← weight(e) + score(pred(S, tail(e)_2))
4:   for every e ∈ E_{F,G} with head(e)_1 = i do
5:     j ← head(e)_2
6:     if score(e) > score(pred(S, j)) then
7:       insert(S, e)
8:       while succ(S, j + 1) ≠ NULL and score(succ(S, j + 1)) ≤ score(e) do
9:         delete(S, succ(S, j + 1))
```

Example 1. Consider the alignment graph of Fig. 2. Initially S is empty. When $i = 2$ procedure ComputeScores sets the scores of the edges $e_1 = (2, 2) \to (3, 7)$ and $e_2 = (2, 3) \to (3, 4)$ to 1. When $i = 3$, the procedure processes the edges e_1 and e_2 in an arbitrary order. Suppose, for example, that e_1 is processed first. When e_1 is processed, it is added to S. When e_2 is processed, it is added to S and e_1 is removed from S (since $j = 4$, $\text{succ}(S, j + 1) = e_1$ and $\text{score}(e_1) = \text{score}(e_2)$). When $i = 4$, the procedure sets the scores of the edges $(4, 2) \to (9, 7)$ and $(4, 3) \to (9, 4)$ to 2 and 1, respectively. When $i = 5$, the procedure sets the score of the edge $e_3 = (5, 5) \to (6, 6)$ to 2 (since $\text{pred}(S, \text{tail}(e_3)_2) = e_2$). When $i = 6$, the edge e_3 is added to S. The next changes in S occur when $i = 8$ and $i = 10$: When $i = 8$ the edge $(7, 8) \to (8, 9)$ is added to S, and when $i = 10$ the edge $(9, 9) \to (10, 10)$ is added to S.

Let $e_1, e_2, \ldots$ be the edges of $E_{F,G}$ according to the order in which they are processed in line 4. An edge e is *t-relevant* if e is one of the edges $e_1, \ldots, e_t$. We say that a path p is t-relevant if all its nonzero weight edges are t-relevant. Denote by $w(t, i, j)$ the maximum weight of a t-relevant path from $(1, 1)$ to (i, j). The correctness of procedure ComputeScores follows immediately from the following lemma.

Lemma 4. *For every t, the score of e_t is computed correctly by* ComputeScores. *Moreover, for every t, just after e_t is processed in lines 4–9,* $\text{score}(\text{pred}(S, j)) = w(t, \text{head}(e)_1, j)$ *for all j.*

Proof. We prove the lemma by induction on t. Assume that the lemma holds for $1, \ldots, t-1$. Let $e_t = (i_1, j_1) \to (i_2, j_2)$. We first prove that the score of e_t is computed correctly. Let t' be the maximum index such that $\text{head}(e_{t'})_1 \le i_1$. By the induction hypothesis, just after $e_{t'}$ is processed, $\text{score}(\text{pred}(S, j_1)) = w(t', \text{head}(e_{t'})_1, j_1)$. Since there is no t'-relevant edge e with $\text{head}(e_{t'})_1 < \text{head}(e)_1 \le i_1$, it follows that $w(t', \text{head}(e_{t'})_1, j_1) = w(t', i_1, j_1)$. All the paths from $(1, 1)$ to (i_1, j_1) are t'-relevant, so $w(t', i_1, j_1)$ is equal to the maximum weight of a path from $(1, 1)$ to (i_1, j_1). Hence, at the time e_t is processed in line 3, $\text{score}(e_t) = \text{weight}(e_t) + \text{score}(\text{pred}(S, j_1))$.

We next prove that after processing e_t in lines 4–9, $\text{score}(\text{pred}(S, j)) = w(t, \text{head}(e)_1, j)$ for all j. Let $i' = \text{head}(e_{t-1})_1$. By induction, just before e_t is processed $\text{score}(\text{pred}(S, j)) = w(t-1, i', j) = w(t-1, i_2, j)$ for all j, where the second equality is true since there is no $(t-1)$-relevant edge e with $i' < \text{head}(e)_1 \le i_2$. A maximum weight t-relevant path from $(1, 1)$ to (i_2, j) can either pass through e_t or not. In the former case, e_t is the last nonzero weight edge on the path, so the weight of the path is $\text{score}(e_t)$. In the latter case, the path is $(t-1)$-relevant, so the weight of the path is $w(t-1, i_2, j)$. Therefore,

$$w(t, i_2, j) = \begin{cases} w(t-1, i_2, j) & \text{if } j < j_2 \\ \max(w(t-1, i_2, j), \text{score}(e_t)) & \text{otherwise} \end{cases}.$$

The if clause in lines 6–9 updates S with exactly this quantity. It follows that just after processing e_t, $\text{score}(\text{pred}(S, j)) = w(t, i_2, j)$ for all j. ■

To analyze the running time of procedure ComputeScores, let us count the number of times each operation on S is called. Each edge of $E_{F,G}$ is inserted or deleted at most once. The number of successor operations is the same as the number of deletions, and the number of predecessor operations is the same as the number of edges. Hence, the total number of operations on S is $O(|E_{F,G}|)$. Using the successor data-structure of van Emde Boas [26] we can support each operation on S in $O(\lg\lg m)$ time yielding a total running time of $O(|E_{F,G}| \cdot \lg\lg m)$. By running procedure ComputeScores recursively on every match pair we get that the total time complexity is bounded by

$$O\left(\sum_{\text{match pair } (v,w)} |E_{F_v,G_w}| \cdot \lg\lg m\right) = O\left(\lg\lg m \cdot \sum_{\text{match pair } (v,w)} \text{depth}(v) \cdot \text{depth}(w)\right)$$
$$= O\left(\lg\lg m \cdot r \cdot \text{height}(F) \cdot \text{height}(G)\right).$$

4. An $O(mr \lg r \cdot \lg\lg m)$ algorithm

We begin this section by giving an alternative description of Klein's algorithm using an alignment graph. However, as opposed to the alignment graph of [25,4] our graph is three dimensional.

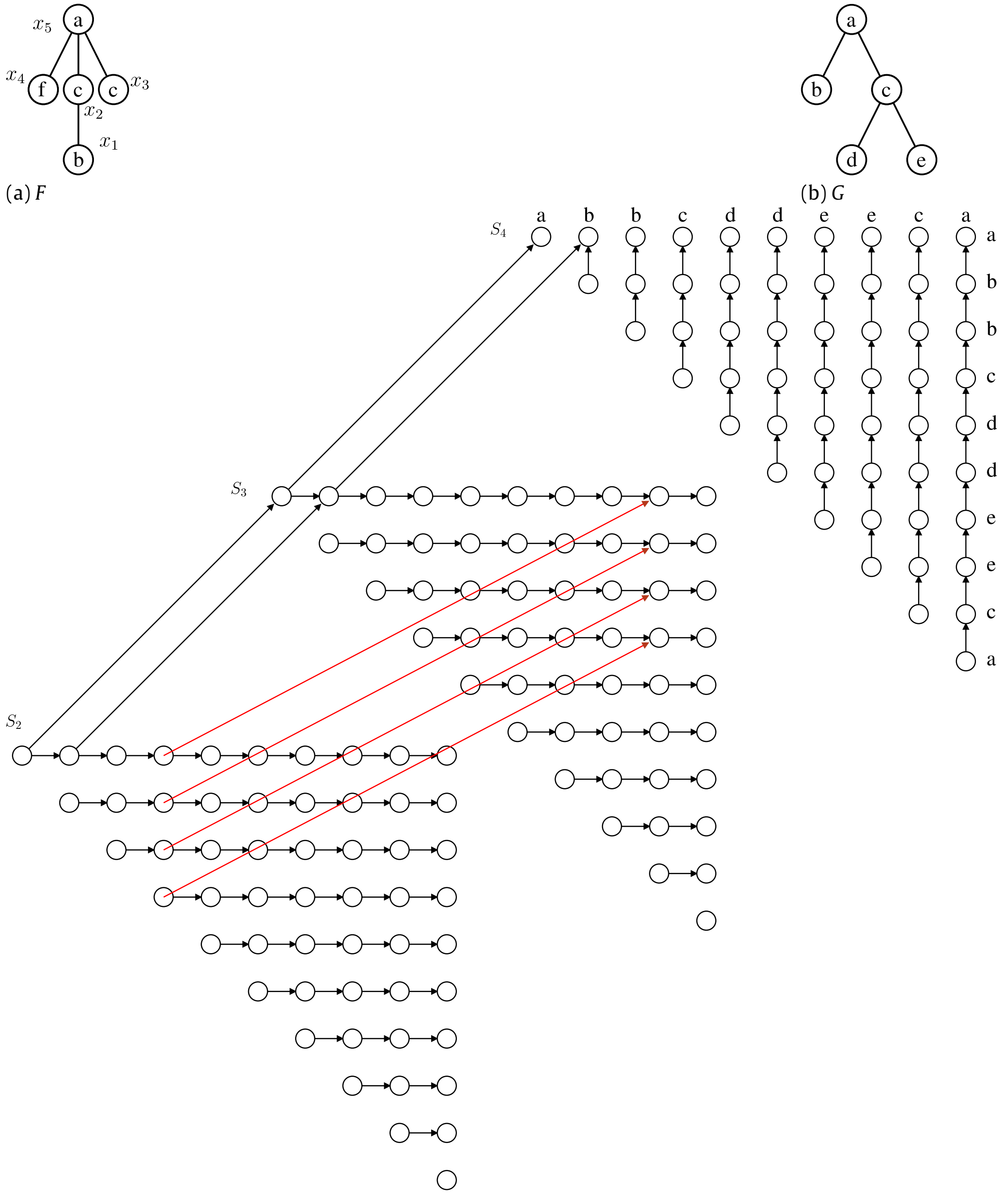
Given a tree F and a path decomposition $\mathcal{P}$ of F we define a sequence of subforests of F as follows. $F(n) = F$, and $F(i)$ for $i < n$ is the forest obtained from $F(i+1)$ by deleting one node: If the root of leftmost tree in $F(i+1)$ is not on the main path of $\mathcal{P}$ then this root is deleted, and otherwise the root of the rightmost tree in F is deleted. Let x_i be the node which is deleted from $F(i)$ when creating $F(i-1)$. Let y_i be the node of G that generates the ith character of the Euler string of G. Let I_{right} be the set of all indices i such that $F(i-1)$ is created from $F(i)$ by deleting the rightmost root of $F(i)$, and $I_{\text{left}} = \{1, \ldots, n\} \setminus I_{\text{right}}$.

The alignment graph $B_{F,G}$ of trees F and G is defined as follows (see Fig. 3). The vertices of $B_{F,G}$ are (i, j, k) for $0 \le i \le n$, $1 \le j \le 2m$, and $j \le k \le 2m$. Intuitively, vertex (i, j, k) corresponds to $\text{LCS}(F(i), G[j..k])$. For a vertex (i, j, k) with $i \in I_{\text{right}}$ the following edges enter the vertex.

1. If $i \ge 1$, an edge $(i-1, j, k) \to (i, j, k)$ with weight 0. This edge corresponds to deletion of the rightmost root of $F(i)$. This does not increase the LCS hence the zero weight.
2. If $j \le k-1$, an edge $(i, j, k-1) \to (i, j, k)$ with weight 0. This edge either connects vertices which represent the same pair of forests, or represent deletion of the rightmost root in $G[j..k]$. Both cases do not change the LCS, hence the zero weight.
3. If x_i, y_k is a match pair, $j \le e_G(k) < k$, and x_i is not on the main path of F, an edge $(i - |F_{x_i}|, j, e_G(k)) \to (i, j, k)$ with weight $\text{LCS}(F_{x_i}, G_{y_k})$. This edge correspond to matching the rightmost tree in $F(i)$ to the rightmost tree of $G[j..k]$.
4. If x_i, y_k is a match pair, $j \le e_G(k) < k$, and x_i is on the main path of F, an edge $(i-1, e_G(k), k-1) \to (i, j, k)$ with weight 1. This edge corresponds to matching x_i (the root of $F(i) = F_{x_i}$) to y_k (the rightmost root of $G[j..k]$). If we match these nodes then only descendants of y_k can be matched to the nodes of $F(i-1)$ (since $F(i)$ is a tree). To ensure this, we set the second coordinate of the tail of the edge to $e_G(k)$ (instead of j as in the previous case), since nodes with indices $j' < e_G(k)$ are not descendants of y_k.

Similarly, for $i \in I_{\text{left}}$ the edges that enter (i, j, k) are

1. If $i \ge 1$, an edge $(i-1, j, k) \to (i, j, k)$ with weight 0.
2. If $j \le k-1$, an edge $(i, j+1, k) \to (i, j, k)$ with weight 0.
3. If x_i, y_j is a match pair, $j < e_G(j) \le k$, and x_i is not on the main path of F, an edge $(i - |F_{x_i}|, e_G(j), k) \to (i, j, k)$ with weight $\text{LCS}(F_{x_i}, G_{y_j})$.
4. If x_i, y_j is a match pair, $j < e_G(j) \le k$, and x_i is on the main path of F, an edge $(i-1, j+1, e_G(j)) \to (i, j, k)$ with weight 1.



(a) F (b) G

(c) Alignment graph

Fig. 3. Part of the alignment graph of two trees F and G. Let $S_i = \{(i, j, k) : 1 \le j \le k \le 2m\}$. Figure (c) shows the vertices in $S_2 \cup S_3 \cup S_4$. To a vertex v in S_4 enters an edge from the vertex in S_4 below v if such vertex exists (as $4 \in I_{\text{left}}$), and an edge from the vertex in S_3 that corresponds to v (only two edges of this type are shown in the figure). Similarly, to a vertex v in S_3 enters an edge from the vertex in S_3 to the left of v if such vertex exists (as $3 \in I_{\text{right}}$), and an edge from the vertex in S_2 that corresponds to v. Additionally, due to the match pair x_3, y_9, to the vertices $(3, 1, 9)$, $(3, 2, 9)$, $(3, 3, 9)$, and $(3, 4, 9)$ enter edges from the vertices $(2, 1, 9)$, $(2, 2, 9)$, $(2, 3, 9)$, and $(2, 4, 9)$, respectively.

The set of all edges in $B_{F,G}$ with nonzero weights is denoted by $E_{F,G}$. In order to build $B_{F,G}$ one needs to know the values of $\mathrm{LCS}(F', G')$ for some pairs of subforests F', G' of F, G. These values are obtained by making recursive calls to Klein's algorithm on the appropriate subforests of F and G.

Lemma 5. *The maximum weight of a path in $B_{F,G}$ from some vertex $(0, l, l)$ to vertex (i, j, k) is equal to* $\mathrm{LCS}(F(i), G[j..k])$.

Proof. We prove the lemma by induction on $i + (k - j)$. The base on the induction ($i - j + k = 0$) is trivially true. Consider some i, j, and k, and suppose that $i \in I_{\text{right}}$ (the proof for $i \in I_{\text{left}}$ is similar).

The proof of the lemma is similar to the proof of Lemma 2. We first show that for a maximum weight path p from some vertex $(0, l, l)$ to (i, j, k), there is an LCS matching between $F(i)$ and $G[j..k]$ of size at least weight(p). This is done by considering the prefix of p up to but not including e, where $e = (i', j', k') \to (i, j, k)$ is the last edge on p. As before, we can use the inductive hypothesis on p' (since we have by the construction of the graph that $i' - j' + k' < i - j + k$) to obtain an LCS mapping M' between $F(i')$ and $G[j'..k']$ of weight weight(p'). We then extend M' into the desired matching M according to the type of the edge e. The arguments are similar to those used in the proof of Lemma 2. Note that in the case when e is an edge of the fourth type, all the vertices in F that are matched in M' are proper descendants of x_i (as $F(i)$ is a tree and $i' = i - 1$), and all the vertices in G that are matched in M' are proper descendants of y_k (as $G[j'..k'] = G_{y_k} - y_k$). Therefore, $M = M' \cup \{(x_i, y_k)\}$ is the desired LCS mapping for that case.

We next prove the opposite direction. We show that for an LCS mapping M between $F(i)$ and $G[j..k]$ of maximum size, there is a path p from some vertex $(0, l, l)$ to (i, j, k) with weight at least $|M|$. We consider several cases according to whether x_i and y_k are matched in M. If both x_i and y_k are matched in M then we consider two cases according to whether x_i is on the main path of F. In each case we choose $M' \subseteq M$ such that there is a path of weight at least $|M'|$ from some vertex $(0, l, l)$ to some vertex (i', j', k'), and from the construction of the graph there is an edge $(i', j', k') \to (i, j, k)$ of weight at least $|M| - |M'|$. ■

Klein's algorithm computes the maximum weight path that ends at each vertex in $B_{F,G}$ using dynamic programming, and returns the maximum weight of a path from some vertex $(0, l, l)$ to $(n, 1, 2m)$, which is equal to LCS(F, G). The path decomposition $\mathcal{P}$ is selected in order to minimize the total size of the alignment graph $B_{F,G}$ and the alignment graphs created by the recursive calls of the algorithm. Using heavy path decomposition [13], the time complexity of Klein's algorithm is $O(n \lg n \cdot m^2)$.

Now, we present an algorithm for computing the LCS based on the sparsity of $E_{F,G}$. Recall that the score of an edge $e \in E_{F,G}$ is the maximum weight of a path in $B_{F,G}$ from some vertex $(0, l, l)$ to head(e) that passes through e.

Lemma 6. *Let e be an edge in $E_{F,G}$ and denote* head$(e) = (i, j, k)$. *If $i \in I_{\text{right}}$ then* score$(e) = \text{LCS}_R(F(i), G[j..k])$, *and otherwise* score$(e) = \text{LCS}_L(F(i), G[j..k])$.

We omit the proof of Lemma 6 as it is similar to the proof of Lemma 3. Knowing the scores of the edges gives us LCS(F, G) as LCS$(F, G) = \text{score}((n - 1, 1, 2m - 1) \to (n, 1, 2m))$. In fact, additional LCS values can be obtained from the scores:

Lemma 7. *For every match pair $x \in F, y \in G$ such that x is on the main path of F there is an edge $e \in E_{F,G}$ such that* LCS$(F_x, G_y) = \text{score}(e)$.

Proof. Let i be the index such that $x = x_i$, and let $e_G(k) < k$ be the indices of the two characters in the Euler string of G that correspond to y. Suppose that $i \in I_{\text{right}}$. Then, $e = (i - 1, e_G(k), k - 1) \to (i, e_G(k), k)$ is an edge in $E_{F,G}$. By Lemma 6, score$(e) = \text{LCS}_R(F(i), G[e_G(k)..k])$. Both $F(i) = F_x$ and $G[e_G(k)..k] = G_y$ are trees, so from Lemma 1 we have that score$(e) = \text{LCS}(F_x, G_y)$. The case of $i \in I_{\text{left}}$ is similar. ■

A high-level description of the algorithm for computing the LCS of F and G is:

Algorithm ComputeLCS

1: Build a path decomposition $\mathcal{P}$ of F.
2: **for** every node x in F in postorder **do**
3:    **if** x is the first node on some path $P \in \mathcal{P}$ **then**
4:       Build the set $E_{F_x,G}$.
5:       Compute the scores of the edges in $E_{F_x,G}$.
6: Output score$((n - 1, 1, 2m - 1) \to (n, 1, 2m))$.

We will explain how to construct the path decomposition $\mathcal{P}$ in step 1 later. For now note just that $\mathcal{P}$ is used when building each of the sets $E_{F_x,G}$ in step 4. To build $E_{F_x,G}$, we need a path decomposition of F_x. We use the path decomposition that is induced from the path decomposition $\mathcal{P}$. In order to build $E_{F_x,G}$ one needs to know the values of LCS$(F_{x'}, G_y)$ for pairs of nodes x' and y, where x' is a node of F_x that is not on the main path of F_x. By Lemma 7, the value of LCS$(F_{x'}, G_y)$ is equal to the score of an edge from $E_{F_{x''},G}$ where x'' is the first vertex on the path $p \in \mathcal{P}$ that contains x' (x'' can equal x'). Since the nodes of F are processed in postorder, the scores of the edges in $E_{F_{x''},G}$ are known when building $E_{F_x,G}$.

It remains to show how to compute the scores of the edges in $E_{F,G}$. The algorithm for computing the scores of the edges uses $4m$ successor data-structures $S_1^{\text{left}}, \ldots, S_{2m}^{\text{left}}$ and $S_1^{\text{right}}, \ldots, S_{2m}^{\text{right}}$. Each of these structures stores a subset of $E_{F,G}$. The key of an edge e in some structure S_i^{right} is head$(e)_3$, and the key of an edge e in some structure S_i^{left} is head$(e)_2$.

Procedure ComputeScoresKlein

for $i = 1, \ldots, n$ **do**
   for every $e \in E_{F,G}$ with tail$(e)_1 = i$ **do**

3: $j \leftarrow \text{tail}(e)_2, k \leftarrow \text{tail}(e)_3$
4: $\text{score}(e) \leftarrow \text{weight}(e) + \max(\text{score}(\text{pred}(S_j^{\text{right}}, k)), \text{score}(\text{succ}(S_k^{\text{left}}, j)))$
5: **for** every $e \in E_{F,G}$ with $\text{head}(e)_1 = i$ **do**
6: $j \leftarrow \text{head}(e)_2, k \leftarrow \text{head}(e)_3$
7: **if** $i \in I_{\text{right}}$ and $\text{score}(e) > \text{score}(\text{pred}(S_j^{\text{right}}, k))$ **then**
8: $\text{insert}(S_j^{\text{right}}, e)$
9: **while** $\text{succ}(S_j^{\text{right}}, k+1) \neq \text{NULL}$ and $\text{score}(\text{succ}(S_j^{\text{right}}, k+1)) \leq \text{score}(e)$ **do**
10: $\text{delete}(S_j^{\text{right}}, \text{succ}(S_j^{\text{right}}, k+1))$
11: **if** $i \in I_{\text{left}}$ and $\text{score}(e) > \text{score}(\text{succ}(S_k^{\text{left}}, j))$ **then**
12: $\text{insert}(S_k^{\text{left}}, e)$
13: **while** $\text{pred}(S_k^{\text{left}}, j-1) \neq \text{NULL}$ and $\text{score}(\text{pred}(S_k^{\text{left}}, j-1)) \leq \text{score}(e)$ **do**
14: $\text{delete}(S_k^{\text{left}}, \text{pred}(S_k^{\text{left}}, j-1))$

We call an edge e with $\text{head}(e)_1 \in I_{\text{right}}$ a *right edge*. Let $e_1, e_2, \ldots$ be the edges of $E_{F,G}$ according to the order in which they are processed in line 5.

Lemma 8. *The scores of all nonzero weight edges are computed correctly by procedure* ComputeScoresKlein. *Moreover, for every t, just after e_t is processed in lines 5–14, for all j and k,* $\text{score}(\text{pred}(S_j^{\text{right}}, k))$ *(resp.,* $\text{score}(\text{succ}(S_k^{\text{left}}, j))$*) is equal to the maximum weight of a t-relevant path from some vertex $(0, l, l)$ to $(\text{head}(e)_1, j, k)$ whose last nonzero weight edge is a right (resp., left) edge.*

We omit the proof since it is very similar to the proof of Lemma 4.

Just as in the previous section, using the successor data-structure of van Emde Boas [26] we have that computing the scores of the edges in $E_{F,G}$ takes $O(|E_{F,G}| \lg\lg m)$ time. The time for computing the LCS between F and G is therefore $O(\sum_{x \in L_{\mathcal{P}}} |E_{F_x,G}| \lg\lg m)$, where $L_{\mathcal{P}}$ is the set of the first nodes of the paths in $\mathcal{P}$. In order to minimize $\sum_{x \in L_{\mathcal{P}}} |E_{F_x,G}|$, we build $\mathcal{P}$ similar to a *heavy path decomposition* but where *heavy* is determined by the number of matches and not by size. This is done as follows. We begin building the main path. We start at the root of F and then we repeatedly extend the path by moving to a child w of the current node that maximizes the number of matches between F_w and G (ties are broken arbitrarily). After obtaining the main path, we remove its nodes from F and then recursively build a path decomposition of each of the remaining trees. The decomposition $\mathcal{P}$ that is obtained has the property that for each node $x \in F$, the number of nodes in $L_{\mathcal{P}}$ that are ancestors of x is at most $\lg r + 1$.

Lemma 9. $\sum_{x \in L_{\mathcal{P}}} |E_{F_x,G}| \leq 2mr(\lg r + 1)$.

Proof. Every edge in $E_{F,G}$ corresponds to a match pair $x \in F, y \in G$. A fixed match pair $x \in F, y \in G$ generates edges in the sets $E_{F_{x'},G}$ for every node $x' \in L_{\mathcal{P}}$ that is an ancestor of x. In each set $E_{F_{x'},G}$ the match pair x, y generates at most $2m$ edges. Therefore $\sum_{x \in L_{\mathcal{P}}} |E_{F_x,G}| \leq \sum_{\text{match pairs}} 2m(\lg r + 1) \leq 2mr(\lg r + 1)$. ∎

We have therefore shown an algorithm that computes the LCS of two trees in $O(mr \lg r \cdot \lg\lg m)$ time.

5. An $O(Lr \lg r \cdot \lg\lg m)$ algorithm

In this section we improve the algorithm of the previous section. Notice that in the alignment graph of the previous section each match pair generates up to $O(m)$ edges (while in the alignment graph of Section 3, each match pair generates exactly one edge). Therefore, the time of processing a match pair is $O(m \lg\lg m)$. We will show how to process each group of edges of a match pair in $O(L \lg\lg m)$ time by exploiting additional sparsity properties of the problem.

Formally, we partition the edges of $E_{F,G}$ into groups, where each group is the edges that correspond to some match pair: For $i \in I_{\text{right}}$ let $E_{F,G,i,a} = \{e \in E_{F,G} | \text{head}(e)_1 = i, \text{head}(e)_3 = a\}$, and for $i \in I_{\text{left}}$ let $E_{F,G,i,a} = \{e \in E_{F,G} | \text{head}(e)_1 = i, \text{head}(e)_2 = a\}$. The total number of groups $E_{F',G,i,a}$ for all the alignment graphs $B_{F',G}$ that are built by the algorithm is at most $r(\lg r + 1)$.

Consider some group $E_{F,G,i,k}$ for $i \in I_{\text{right}}$. Let $s = e_G(k)$. We have that $E_{F,G,i,k} = \{e_1, \ldots, e_s\}$ where $\text{head}(e_j) = (i, j, k)$. Denote $l_1 = \text{score}(e_s) = \text{weight}(e_s)$ and $l_2 = \text{score}(e_1)$. By Lemma 6, $\text{score}(e_1) \geq \text{score}(e_2) \geq \cdots \geq \text{score}(e_s)$. Moreover, for all j, $\text{score}(e_j) \in \{0, \ldots, L\}$ and $\text{score}(e_j) - \text{score}(e_{j+1}) \in \{0, 1\}$. Therefore, there are indices $j_{l_1} = s, j_{l_1+1}, \ldots, j_{l_2}$ such that $\text{score}(e_{j_l}) = l$ and $\text{score}(e_{j_{l+1}}) = l - 1$ (if $l \neq l_1$) for all l. These indices are called the *compact representation* of the scores of $E_{F,G,i,k}$.

To improve the algorithm of the previous section, instead of processing individual edges, we will process groups. For each group, we will compute the compact representation of its scores. The time to process each group will be $O(L \lg\lg m)$ so the total time complexity will be $O(Lr \lg r \cdot \lg\lg m)$.

We define two-dimensional arrays $A_1, \ldots, A_n$, where $A_i[j, k]$ is the maximum weight of path from some vertex $(0, l, l)$ to (i, j, k). By Lemma 5, every array A_i has the following properties.

1. Each row of A_i is monotonically increasing.
2. Each column of A_i is monotonically decreasing.

3. The difference between two adjacent cells in A_i is either 0 or 1.
4. Each cell of A_i is an integer from $\{0, \ldots, L\}$.

The properties above are the same as the properties of the dynamic programming table for string LCS. Following the approach of [15], we define the *l-contour* of A_i (for $1 \le l \le L$) to be the set of all pairs (j, k) such that $A_i[j, k] = l$, $A_i[j + 1, k] < l$ (or $j = 2m$), and $A_i[j, k - 1] < l$ (or $k = 1$). By properties (1) and (2) of A_i we have that for two pairs (j, k) and (j', k') in the l-contour of A_i we have either $j < j'$ and $k < k'$, or $j > j'$ and $k > k'$.

The algorithm processes each i from 1 to n. Again, iteration i consists of two stages: (1) updating the l-contours according to the groups $E_{F,G,i,a}$ for all a (2) computing the compact representation of the scores for each group $E_{F,G,i',a}$ such that the edges $e \in E_{F,G,i',a}$ satisfy $\text{tail}(e)_1 = i$. We next explain each of the two stages in detail.

Computing the l-contours of A_i for all l is done by updating the l-contours of A_{i-1} that were computed in the previous iteration. The l-contour of A_i for the current value of i is kept using two successor data-structures S_l^1 and S_l^2. The key of a pair (j, k) in S_l^1 is j, while the key of (j, k) in S_l^2 is k.

Suppose that $i \in I_{\text{right}}$ (handling $i \in I_{\text{left}}$ is similar). In order to compute the l-contours of A_i, we process the groups $E_{F,G,i,k}$ for all k. Consider some fixed $E_{F,G,i,k}$, and let $j_{l_1}, j_{l_1+1}, \ldots, j_{l_2}$ be the compact representation of the scores of $E_{F,G,i,k}$ (which was computed in a prior iteration of the algorithm). Updating the l-contours according to the scores of the edges in $E_{F,G,i,k}$ is done by:

Procedure UPDATECONTOURS

```
for l = l_1, ..., l_2 do
  if pred(S_l^2, k) = NULL or pred(S_l^2, k)_1 < j_l then
    p ← (j_l, k)
    insert(S_l^1, p)
    insert(S_l^2, p)
    while succ(S_l^2, k + 1) ≠ NULL and succ(S_l^2, k + 1)_1 ≤ j_l do
      p ← succ(S_l^2, k + 1)
      delete(S_l^1, p)
      delete(S_l^2, p)
```

It remains to describe stage (2), which computes the compact representation of the scores of some group $E_{F,G,i',k'}$ such that the edges $e \in E_{F,G,i',k'}$ satisfy $\text{tail}(e)_1 = i$. Suppose that $i' \in I_{\text{right}}$ and denote $E_{F,G,i',k'} = \{e_1, \ldots, e_s\}$ where $e_j = (i, j, k) \to (i', j, k')$. All the edges in $E_{F,G,i',k'}$ have the same weight w. Suppose that $x_{i'}$ is not on the main path of F. Clearly, $\text{score}(e_j) = w + A_i[j, k]$. Therefore the compact representation of the scores of $E_{F,G,i',k'}$ can be computed using $S_1^2, \ldots, S_L^2$:

```
j_w ← s
for l = 1, ..., L do
  if pred(S_l^2, k) ≠ NULL then j_{l+w} ← pred(S_l^2, k)_1
```

If $x_{i'}$ is on the main path of F then $\text{score}(e_1) = \cdots = \text{score}(e_s) = 1 + A_i[s, k]$, and computing the compact representation of the scores is done similarly. The computation of the compact representation of the scores of a group $E_{F,G,i',j'}$ with $i \in I_{\text{left}}$ is done similarly using the structures $S_1^1, \ldots, S_L^1$.

We have established the following theorem:

Theorem 10. *The tree LCS problem can be solved in time $O(Lr \lg r \cdot \lg\lg m)$.*

6. An $O(rh \lg\lg m)$ algorithm for homeomorphic tree LCS

In this section we address the homeomorphic tree LCS problem. For this problem we obtain an $O(rh \lg\lg m)$ time algorithm, where $h = \text{height}(F) + \text{height}(G)$. We start by describing the $O(nm)$ nonsparse algorithm of Amir et al. [2]. Here, the computation of $\text{HLCS}(F, G)$ is done recursively, in a postorder traversal of F and G. For every pair of nodes $v \in F$ and $w \in G$ we compute $\text{score}(v, w)$ which is equal to $\text{HLCS}(F_v, G_w)$. The computation of $\text{score}(v, w)$ is based on the previously computed scores for all children of v and w as follows. Let $c(u)$ denote the number of children of a node u and let $u_1, \ldots, u_{c(u)}$ denote the ordered sequence of the children of u. Then

$$\text{score}(v, w) = \max\left\{\max_{i \le c(v)}\{\text{score}(v_i, w)\}, \max_{i \le c(w)}\{\text{score}(v, w_i)\}, \alpha(v, w) + 1\right\},$$

where $\alpha(v, w)$ is defined as follows. If (v, w) is not a match pair then $\alpha(v, w) = -1$. Otherwise, $\alpha(v, w)$ is the maximum weight of a non-crossing bipartite matching between the vertices $v_1, \ldots, v_{c(v)}$ and the vertices $w_1, \ldots, w_{c(w)}$, where the weight of matching v_i with w_j is $\text{score}(v_i, w_j)$. Computing $\alpha(v, w)$ takes $O(c(v) \cdot c(w))$ time using dynamic programming on a $c(v) \times c(w)$ table.

In order to obtain a sparse version of this algorithm, there are two goals to be met. First, rather than computing $\text{score}(v, w)$ for all nm node pairs, we will only compute the scores for match pairs. Second, we need to avoid the $O(c(v) \cdot c(w))$ time complexity of the dynamic programming algorithm for computing $\alpha(v, w)$ and replace it with sparse dynamic programming. For every match pair (v, w) we have

$$\text{score}(v, w) = \max\left\{\max_{v'}\{\text{score}(v', w)\}, \max_{w'}\{\text{score}(v, w')\}, \alpha(v, w) + 1\right\},$$

where $\max_{v'}$ is maximum over all proper descendants v' of v that have the same label as v, and $\max_{w'}$ is defined similarly. Computing the two maxima above for all match pairs is done as follows. First, we initialize $\text{score}(v, w) = 0$ for all match pairs (v, w). After computing $\text{score}(v, w)$ for some match pair (v, w), we perform $\text{score}(\hat{v}, w) \leftarrow \max\{\text{score}(\hat{v}, w), \text{score}(v, w)\}$ and $\text{score}(v, \hat{w}) \leftarrow \max\{\text{score}(v, \hat{w}), \text{score}(v, w)\}$, where $\hat{v}$ and $\hat{w}$ are the parents of v and w, respectively. Thus, for some match pair (v, w), after processing all match pairs (v', w') where v' is a descendant of v and w' is a descendant of w, we have that $\text{score}(v, w)$ is equal to $\max\left\{\max_{v'}\{\text{score}(v', w)\}, \max_{w'}\{\text{score}(v, w')\}\right\}$, so it remains to compute $\alpha(v, w)$.

To compute $\alpha(v, w)$, define $P_{v,w}$ to be the set of all pairs (v_i, w_j) such that $\text{score}(v_i, w_j) > 0$. Applying a sparse dynamic programming approach to the computation of $\alpha(v, w)$ exploits the fact that $P_{v,w}$ can be much smaller than $c(v) \cdot c(w)$. However, note that just querying all pairs of children of v and w to check which ones have a positive score would already consume $O(c(v) \cdot c(w))$ time. But, given the set $P_{v,w}$, the cost of computing $\alpha(v, w)$ is $O(|P_{v,w}| \lg\lg m)$ instead of $O(c(v) \cdot c(w))$. Thus, in the rest of this section we show how to efficiently construct the sets $P_{v,w}$.

Our approach is based on the observation that, even before the scores are computed, a key subset of the match pairs of F_v and G_w can be identified that have the potential to eventually participate in $P_{v,w}$. For every $i \le c(v)$ and $j \le c(w)$, let $\hat{S}_{v,w,i,j}$ be the set of all match pairs (x, y) such that x is a descendant of v_i and y is a descendant of w_j, and let $S_{v,w,i,j}$ be the set of all match pairs $(x, y) \in \hat{S}_{v,w,i,j}$ for which there is no match pair $(x', y') \neq (x, y)$ in $S_{v,w,i,j}$ such that x' is an ancestor of x and y' is an ancestor of y.

The following lemma shows that $P_{v,w}$ can be built from the sets $S_{v,w,i,j}$.

Lemma 11. *Let (v, w) be a match pair. Let v_i be a child of v and w_j be a child of w such that (v_i, w_j) is not a match pair. Then, $\text{score}(v_i, w_j)$ is equal to the maximum score of a pair in $S_{v,w,i,j}$, or to 0 if $S_{v,w,i,j} = \emptyset$.*

Proof. Since (v_i, w_j) is not a match pair, we have that $\text{score}(v_i, w_j)$ is equal to the maximum score of a pair in $\hat{S}_{v,w,i,j}$, or to 0 if $\hat{S}_{v,w,i,j} = \emptyset$. To finish the proof, we will show that for every match pair $(\hat{v}, \hat{w}) \in \hat{S}_{v,w,i,j}$ there is a match pair $(v', w') \in S_{v,w,i,j}$ with $\text{score}(v', w') \ge \text{score}(\hat{v}, \hat{w})$. Let $(\hat{v}, \hat{w})$ be a match pair in $\hat{S}_{v,w,i,j}$. If $(\hat{v}, \hat{w}) \in S_{v,w,i,j}$ then we are done. Otherwise, by the definition of $S_{v,w,i,j}$, there is a match pair $(v', w') \in S_{v,w,i,j}$ such that v' is an ancestor of $\hat{v}$ and w' is an ancestor of $\hat{w}$. We have that $\text{score}(v', w') = \text{HLCS}(F_{v'}, G_{w'}) \ge \text{HLCS}(F_{\hat{v}}, G_{\hat{w}}) = \text{score}(\hat{v}, \hat{w})$. ■

While it is possible to build the sets $S_{v,w,i,j}$, it is simpler to build sets $S'_{v,w,i,j}$ such that $S_{v,w,i,j} \subseteq S'_{v,w,i,j} \subseteq \hat{S}_{v,w,i,j}$. From the proof of Lemma 11 we have that $\text{score}(v_i, w_j)$ is also equal to the maximum score of a pair in $S'_{v,w,i,j}$, or to 0 if $S'_{v,w,i,j} = \emptyset$. We build the sets $S'_{v,w,i,j}$ as follows. For each match pair (x, y) of F, G we build a list L_x of all proper ancestors v of x such that v is the lowest proper ancestor of x with label equal to $\text{label}(v)$ (the list L_x is generated by traversing the path from x to the root while maintaining a Boolean array that stores which characters were already encountered). We also build a list L_y of all proper ancestors w of y such that w is the lowest proper ancestor of y with label equal to $\text{label}(w)$. For every $v \in L_x$ and every proper ancestor w of y with $\text{label}(w) = \text{label}(v)$, we add the pair (x, y) to $S'_{v,w,i,j}$ where v_i is the child of v which is on the path from v to x, and w_j is the child of w which is on the path from w to y. Similarly, for every $w \in L_y$ and every proper ancestor v of x with $\text{label}(v) = \text{label}(w)$, we add the pair (x, y) to $S'_{v,w,i,j}$.

Lemma 12. *$S'_{v,w,i,j} \supseteq S_{v,w,i,j}$ for all match pairs (v, w) and all i and j.*

Proof. Suppose conversely that there is a match pair (v, w) and indices i and j such that $S'_{v,w,i,j} \not\supseteq S_{v,w,i,j}$. Let (x, y) be a pair in $S_{v,w,i,j}$ which is not in $S'_{v,w,i,j}$. From the fact that $(x, y) \notin S'_{v,w,i,j}$ we have that there is a vertex v' such that v' is a proper ancestor of x, v' is a proper descendant of v, and $\text{label}(v') = \text{label}(v)$. Also, there is a vertex w' such that w' is a proper ancestor of y, w' is a proper descendant of w, and $\text{label}(w') = \text{label}(w)$. We obtain that (v', w') is a match pair, which contradicts the assumption that $(x, y) \in S_{v,w,i,j}$. ■

Theorem 13. *The homeomorphic tree LCS problem can be solved in $O(rh \lg\lg m)$ time, where $h = \text{height}(F) + \text{height}(G)$.*

Proof. The algorithm consists of a preprocessing stage, during which the sets $S'_{v,w,i,j}$ are constructed for every match pair (v, w) and every i and j, and a main stage, in which the scores of match pairs are computed.

In the preprocessing stage, handling a match pair (x, y) takes $O(h)$ time. Therefore, the preprocessing stage is done in $O(rh)$ time. Moreover, $\sum_{\text{match pair }(v,w)} \sum_i \sum_j |S'_{v,w,i,j}| = O(rh)$.

During the main stage, $\text{score}(v, w)$ is computed for every match pair (v, w). For a single match pair (v, w), this takes $O(|P_{v,w}| \lg\lg m)$ time. Since $|P_{v,w}| \le \sum_i \sum_j |S'_{v,w,i,j}|$, we conclude that the total work over all match pairs is $O(rh \lg\lg m)$. ■

References

[1] M.I. Abouelhoda, E. Ohlebusch, Chaining algorithms for multiple genome comparison, Journal of Discrete Algorithms 3 (2–4) (2005) 321–341.
[2] A. Amir, T. Hartman, O. Kapah, R. Shalom, D. Tsur, Generalized LCS, Theoretical Computer Science 409 (3) (2008) 438–449.
[3] A. Apostolico, C. Guerra, The longest common subsequence problem revisited, Algorithmica 2 (1987) 315–336.
[4] R. Backofen, S. Chen, D. Hermelin, G.M. Landau, M.A. Roytberg, O. Weimann, K. Zhang, Locality and gaps in RNA comparison, Journal of Computational Biology 14 (8) (2007) 1074–1087.
[5] P. Bille, A survey on tree edit distance and related problems, Theoretical Computer Science 337 (2005) 217–239.
[6] P. Bille, Pattern matching in trees and strings, Ph.D. Thesis, ITU University of Copenhagen, 2007.
[7] S. Chawathe, Comparing hierarchical data in external memory, in: Proc. 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, U.K., 1999, pp. 90–101.
[8] W. Chen, New algorithm for ordered tree-to-tree correction problem, Journal of Algorithms 40 (2001) 135–158.
[9] F.Y.L. Chin, C.K. Poon, A fast algorithm for computing longest common subsequences of small alphabet size, Journal of Information Processing 13 (4) (1990) 463–469.
[10] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A subquadratic sequence alignment algorithm for unrestricted scoring matrices, SIAM Journal on Computing 32 (2003) 1654–1673.
[11] E.D. Demaine, S. Mozes, B. Rossman, O. Weimann, An optimal decomposition algorithm for tree edit distance, in: Proc. 34th International Colloquium on Automata, Languages and Programming, ICALP, 2007, pp. 146–157.
[12] D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano, Sparse dynamic programming i: linear cost functions, Journal of the ACM 39 (3) (1992) 519–545.
[13] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal of Computing 13 (2) (1984) 338–355.
[14] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Communications of the ACM 18 (6) (1975) 341–343.
[15] D.S. Hirschberg, Algorithms for the longest common subsequence problem, Journal of the ACM 24 (4) (1977) 664–675.
[16] W.J. Hsu, M.W. Du, New algorithms for the LCS problem, Journal of Computer and System Sciences 29 (2) (1984) 133–152.
[17] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (5) (1977) 350–353.
[18] P.N. Klein, Computing the edit-distance between unrooted ordered trees, in: Proc. 6th annual European Symposium on Algorithms, ESA, 1998, pp. 91–102.
[19] P.N. Klein, S. Tirthapura, D. Sharvit, B.B. Kimia, A tree-edit-distance algorithm for comparing simple, closed shapes, in: Proc. 11th ACM-SIAM Symposium on Discrete Algorithms, SODA, 2000, pp. 696–704.
[20] A. Lozano, G. Valiente, On the maximum common embedded subtree problem for ordered trees, in: C.S. Iliopoulos, T. Lecroq (Eds.), String Algorithmics, King's College Publications, 2004, pp. 155–170.
[21] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, Journal of Computer and System Sciences 20 (1) (1980) 18–31.
[22] G. Myers, W. Miller, Chaining multiple-alignment fragments in sub-quadratic time, in: Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 1995, pp. 38–47.
[23] C. Rick, Simple and fast linear space computation of longest common subsequences, Information Processing Letters 75 (6) (2000) 275–281.
[24] K. Tai, The tree-to-tree correction problem, Journal of the ACM 26 (3) (1979) 422–433.
[25] H. Touzet, A linear tree edit distance algorithm for similar ordered trees, in: Proc. 16th Annual Symposium on Combinatorial Pattern Matching, CPM, 2005, pp. 334–345.
[26] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Information Processing Letters 6 (3) (1977) 80–82.
[27] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, Journal of the ACM 21 (1) (1974) 168–173.
[28] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM Journal of Computing 18 (6) (1989) 1245–1262.