Some Lower and Upper Bounds for Tree Edit Distance

Shay Mozes
Department of Computer Science
Brown University
Providence, RI.

March 23, 2008

Abstract

In this report I describe my results on the Tree Edit Distance problem [13, 27]. The edit distance between two ordered rooted trees with vertex labels is the minimum cost of transforming one tree into the other by a sequence of elementary operations consisting of deleting and relabeling existing nodes, as well as inserting new nodes. Tree Edit Distance has applications in many fields such as computer vision, computational biology and compiler optimization. I describe an algorithm that computes the edit distance between two trees of sizes n and m, where $m < n$, and runs in $O(nm^2(1+\log \frac{n}{m})) = O(n^3)$ time and $O(nm)$ space. The previously best known algorithm for this problem, which is due to Philip Klein [22], runs in $O(m^2 n \log n) = O(n^3 \log n)$ time and $O(mn)$ space. Next, a matching lower bound is proved for the family of decomposition strategy algorithms, which includes the previous fastest algorithms for this problem. The best previously known lower bound for this family was $\Omega(n^2 \log^2 n)$. Finally, I describe recent results on the Longest Common Subtree problem. This is an interesting special case of Tree Edit Distance in which only insertions and deletions are considered (i.e., the cost of all relabeling operations is infinite, and the cost of any insertion or deletion is 1). I describe a few algorithms for this problem, the fastest of which runs in $O(Lr \log r \log \log m)$, where L is the size of the LCS ($L \leq m$) and r is the number of pairs of vertices with matching labels, one from each tree ($r \leq nm$). These algorithms combine techniques from sparse string LCS (Longest Common Subsequence), with Tree Edit Distance algorithms.

The tree edit distance paper [13] is a joint work with Erik Demaine, Benjamin Rossman and Oren Weimann. The longest common subtree paper [27] is a joint work with Dekel Tsur, Oren Weimann and Michal Ziv-Ukelson.

Contents

Introduction

1.1 Overview

The problem of comparing trees occurs in diverse areas such as structured text databases like XML, computer vision, compiler optimization, natural language processing, and computational biology [6, 9, 23, 30, 33].

One example for an application is the analysis of RNA molecules in computational biology. *Ribonucleic acid* (RNA) is a polymer consisting of a sequence of nucleotides (Adenine, Cytosine, Guanine, and Uracil) connected linearly via a backbone. In addition, complementary nucleotides (AU, GC, and GU) can form hydrogen bonds, leading to a structural formation called the *secondary structure* of the RNA. Because of the nested nature of these hydrogen bonds, the secondary structure of RNA can be naturally represented by an ordered rooted tree [16, 38] as depicted in Fig. 1.1. Recently, comparing RNA sequences has gained increasing interest thanks to numerous discoveries of biological functions associated with RNA. A major fraction of RNA's function is determined by its secondary structure [26]. Therefore, computing the similarity between the secondary structure of two RNA molecules can help determine the functional similarities of these molecules.

The *tree edit distance* metric is a common similarity measure for rooted ordered trees. It was introduced by Tai in the late 1970's [33] as a generalization of the well-known string edit distance problem [37]. Let F and G be two rooted trees with a left-to-right order among siblings and where each vertex is assigned a label from an alphabet Σ. The *edit distance* between F and G is the minimum cost of transforming F into G by a sequence of

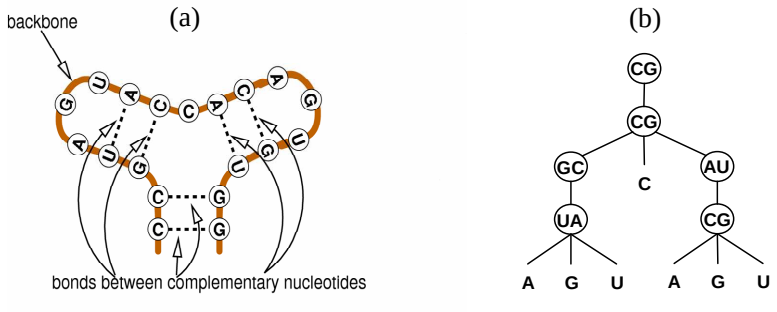

Figure 1.1: Two different ways of viewing an RNA sequence. In (a), a schematic 2-dimensional description of an RNA folding. In (b), the RNA as a rooted ordered tree.

elementary operations consisting of deleting and relabeling existing nodes, as well as inserting new nodes (allowing at most one operation to be performed on each node). These operations are illustrated in Fig. 1.2. Formally, given a node v in F with parent v', *relabel* changes the label of v, *delete* removes a non-root node v and sets the children of v as the children of v' (the children are inserted in the place of v as a subsequence in the left-to-right order of the children of v'), and *insert* (the complement of delete) connects a new node v as a child of some v' in F making v the parent of a consecutive subsequence of the children of v'. The cost of the elementary operations is given by two functions, c_{del} and c_{match}, where $c_{\text{del}}(\tau)$ is the cost of deleting or inserting a vertex with label τ, and $c_{\text{match}}(\tau_1, \tau_2)$ is the cost of changing the label of a vertex from τ_1 to τ_2. Since a deletion in F is equivalent to an insertion in G and vice versa, we can focus on finding the minimum cost of a sequence of just deletions and relabelings in both trees that transform F and G into isomorphic trees.

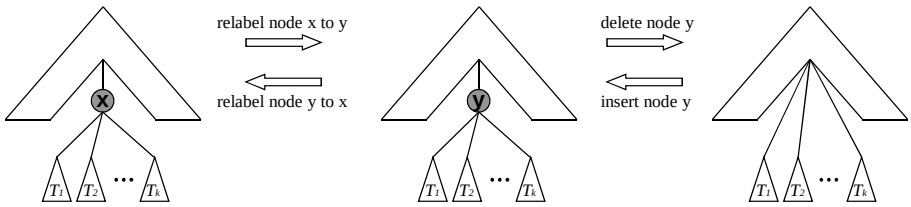

Figure 1.2: The three editing operations on a tree with vertex labels.

The *Longest Common Substring* (string LCS) of two strings, a special case of string edit distance, is the longest subsequence of symbols that appears in both strings. It corresponds to the edit distance with infinite cost for changing labels and unit costs for insertions and deletions. The generalization of this problem to rooted, ordered, labeled trees is called the *Largest Common Subtree* problem, and was considered by Lozano and Valiente [24] and Amir et al. [1]. In this problem we wish to find the minimal number of insertion and deletions operations (as defined for tree edit distance above), that transform one of the trees into the other. Again, since a deletion in one tree is equivalent to an insertion in the other and vice versa, the problem is equivalent to finding a largest forest that can be obtained from each of the two trees by only deleting nodes. The size of such a largest forest (size of the LCS) is a commonly used measure for the similarity between pairs of trees.

1.2 Related work and previous results.

To state running times, we need some basic notation. Let n and m denote the sizes $|F|$ and $|G|$ of the two input trees, ordered so that $n \geq m$. Let n_{leaves} and m_{leaves} denote the corresponding number of leaves in each tree, and let n_{height} and m_{height} denote the corresponding height of each tree, which can be as large as n and m respectively.

1.2.1 Tree edit distance

Tai [33] presented the first algorithm for computing tree edit distance, which requires $O(n^2_{\text{leaves}} m^2_{\text{leaves}} nm)$ time and space, and thus has a worst-case running time of $O(n^3 m^3) = O(n^6)$. Zhang and

Shasha [30] improved this result to an $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$ time algorithm using $O(nm)$ space. In the worst case, their algorithm runs in $O(n^2m^2) = O(n^4)$ time. Klein [22] improved this result to a worst-case $O(m^2n \log n) = O(n^3 \log n)$ time algorithm using $O(nm)$ space. These last two algorithms are based on closely related dynamic programs, and both present different ways of computing only a subset of a larger dynamic program table; these entries are referred to as *relevant subproblems*. Dulucq and Touzet [14] introduced the notion of a *decomposition strategy* (see Section 2.1.4) as a general framework for algorithms that use this type of dynamic program, and proved a lower bound of $\Omega(nm \log n \log m)$ time for any such strategy.

Many other solutions have been developed; see [2, 6, 35] for surveys. The most recent development is by Chen [10], who presented a different approach that uses results on fast matrix multiplication. Chen's algorithm runs in $O(nm + nm^2_{\text{leaves}} + n_{\text{leaves}}m^d_{\text{leaves}})$ time and $O(n + (m + n^2_{\text{leaves}}) \min\{n_{\text{leaves}}, n_{\text{height}}\})$ space. Here d is the exponent for computing the *min-plus* product of two square matrices. According to Chen[1], $d = 2.5$. Hence, the worst case running time of his algorithm is $O(nm^{2.5}) = O(n^{3.5})$. Therefore, Klein's is the fastest in terms of worst-case time complexity among all these algorithms. Previous improvements to Klein's $O(n^3 \log n)$ time bound were achieved only by constraining the edit operations or the scoring scheme [9, 29, 31, 39].

1.2.2 Largest common subtree

To date, computing the LCS of two trees is done using *tree edit distance* algorithms [1]. This is in contrast with the situation for string LCS. The edit distance problem on strings (and hence also string LCS) can be solved in $O(st)$ time and space, where s and t ($s \leq t$) are the lengths of the strings [18, 37]. The only known speedups to the string edit distance algorithm are by a logarithmic factor [12, 25, 7]. For the string LCS problem however, it is possible to obtain time complexities better than $\widetilde{O}(mn)$ in favorable cases [19, 21, 3, 20, 11, 28]. 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. Many string LCS algorithms originate in either Hirschberg [19] or Hunt and Szymanski [21]. Given two strings of lengths s, t ($s \leq t$) over an alphabet Σ, let L denote the size of their LCS (obviously, $L \leq s$), and let r denote the number of matching pairs of characters, one from each string ($r \leq st$). Hirschberg's algorithm achieves an $O(tL + t \lg |\Sigma|)$ time complexity by computing chains in succession. The Hunt-Szymanski algorithm achieves an $O(r \lg s)$ time complexity by extending partial chains, and can be improved to $O(r \lg \lg s)$ by using the successor data-structure of van Emde Boas [36]. No algorithms that exploit sparsity in the tree LCS problem have been previously suggested.

[1] Chen claims that an algorithm due to Fredman [15] achieves $d = 2.5$. Indeed, Fredman proves that $n^{2.5}$ addition and comparison operations suffice to determine the min-plus product of two n by n matrices. However, he does not provide an algorithm that actually computes this product in $O(n^{2.5})$ time. To the best of our knowledge, the best upper bound for this problem is $O(n^3/\log^2 n)$, due to Chan [8]

1.3 Summary of results

1. For the tree edit distance problem We present a new algorithm that falls into the same *decomposition strategy* framework of [14, 22, 30]. In the worst case, our algorithm requires $O(nm^2(1+\log \frac{n}{m})) = O(n^3)$ time and $O(nm)$ space. The corresponding sequence of edit operations can easily be obtained within the same time and space bounds. We therefore improve upon all known algorithms in the worst case time complexity. Our approach is based on Klein's, but whereas the recursion scheme in Klein's algorithm is determined by just one of the two input trees, in our algorithm the recursion depends alternately on both trees.

2. We prove a worst-case lower bound of $\Omega(nm^2(1 + \log \frac{n}{m}))$ time for all decomposition strategy algorithms for the tree edit distance problem. This bound improves the previous best lower bound of $\Omega(nm \log n \log m)$ time [14], and establishes the optimality of our algorithm among all decomposition strategy algorithms.

3. For the longest common subtree problem, we show how to modify Zhang and Shasha's and Klein's algorithms using ideas from Hunt-Szymanski and Hirschberg string LCS algorithms. 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 $O(Lr \cdot \lg r \cdot \lg \lg m))$, where $L = |\text{LCS}(F, G)|$. To derive this algorithm we present a novel three dimensional alignment graph.

Lower and Upper Bounds for Tree Edit Distance

2.1 A unified presentation of previous algorithms

2.1.1 Definitions

Both the existing algorithms and our new algorithm compute the edit distance of finite ordered Σ-labeled forests, henceforth *forests*. These are forests that have a left-to-right order among siblings and each vertex is assigned a label from a given finite alphabet Σ such that two different vertices can have the same label or different labels. The unique empty forest/tree is denoted by $\emptyset$. The vertex set of a forest F is written simply as F, as when we speak of a vertex $v \in F$. For a forest F and $v \in F$, $\sigma(v)$ denotes the label of v, F_v denotes the subtree of F rooted at v, and $F - v$ denotes the forest F after deleting v. The special case of $F - root(F)$ where F is a tree and $root(F)$ is its root is denoted F°. The leftmost and rightmost trees of a forest F are denoted by L_F and R_F and their roots by ℓ_F and r_F. We denote by $F - L_F$ the forest F after deleting the entire leftmost tree L_F; similarly $F - R_F$. A left-to-right postorder traversal of F is the postorder traversal of all its trees $L_F, \ldots, R_F$ from left to right. For a tree T, the postorder traversal is defined recursively as the postorder traversal of the forest T° followed by a visit of $root(T)$ (as opposed to a preorder traversal that first visits $root(T)$ and then T°). A forest obtained from F by a sequence of any number of deletions of the leftmost and rightmost roots is called a *subforest* of F.

Given forests F and G and vertices $v \in F$ and $w \in G$, we write $c_{\text{del}}(v)$ instead of $c_{\text{del}}(\sigma(v))$ for the cost of deleting or inserting $\sigma(v)$, and we write $c_{\text{match}}(v, w)$ instead of $c_{\text{match}}(\sigma(v), \sigma(w))$ for the cost of relabeling $\sigma(v)$ to $\sigma(w)$. $\delta(F, G)$ denotes the edit distance between the forests F and G.

Because insertion and deletion costs are the same (for a node of a given label), insertion in one forest is tantamount to deletion in the other forest. Therefore, the only edit operations we need to consider are relabelings and deletions of nodes in both forests. In the next two sections, we briefly present the algorithms of Shasha and Zhang, and of Klein. This presentation, inspired by the tree similarity survey of Bille [6], is somewhat different from the original presentations and is essential for understanding our algorithm.

2.1.2 Zhang and Shasha's algorithm

Given two forests F and G of sizes n and m respectively, the following lemma is easy to verify. Intuitively, the lemma says that in any sequence of edit operations the two rightmost roots in F and G must either be matched with each other or else one of them is deleted.

Lemma 2.1 ([30]). *$\delta(F, G)$ can be computed as follows:*

- $\delta(\emptyset, \emptyset) = 0$
- $\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{\text{del}}(r_F)$
- $\delta(\emptyset, G) = \delta(\emptyset, G - r_G) + c_{\text{del}}(r_G)$
- $\delta(F, G) = \min \begin{cases} \delta(F - r_F, G) + c_{\text{del}}(r_F), \\ \delta(F, G - r_G) + c_{\text{del}}(r_G), \\ \delta(R_F^\circ, R_G^\circ) + \delta(F - R_F, G - R_G) + c_{\text{match}}(r_F, r_G) \end{cases}$

Lemma 2.1 yields an $O(m^2n^2)$ dynamic programming algorithm. If we index the vertices of the forests F and G according to their left-to-right postorder traversal position, then entries in the dynamic program table correspond to pairs (F', G') of subforests F' of F and G' of G where F' contains vertices $\{i_1, i_1 + 1, \ldots, j_1\}$ and G' contains vertices $\{i_2, i_2 + 1, \ldots, j_2\}$ for some $1 \le i_1 \le j_1 \le n$ and $1 \le i_2 \le j_2 \le m$.

However, we next show that only $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$ different *relevant subproblems* are encountered by the recursion computing $\delta(F, G)$. We calculate the number of *relevant subforests* of F and G independently, where a forest F' (respectively G') is a relevant subforest of F (respectively G) if it occurs in the computation of $\delta(F, G)$. Clearly, multiplying the number of relevant subforests of F and of G is an upper bound on the total number of relevant subproblems.

We now count the number of relevant subforests of F; the count for G is similar. First, notice that for every node $v \in F$, F_v° is a relevant subproblem. This is because the recursion allows us to delete the rightmost root of F repeatedly until v becomes the rightmost root; we then match v (i.e., relabel it) and get the desired relevant subforest. A more general claim is stated and proved later on in Lemma 2.3. We define

$$\text{keyroots}(F) = \{\text{the root of } F\} \cup \{v \in F \mid v \text{ has a left sibling}\}.$$

It is easy to see that every relevant subforest of F is a prefix (with respect to the postorder indices) of F_v° for some node $v \in \text{keyroots}(F)$. If we define v's collapse depth $\text{cdepth}(v)$ to be the number of keyroot ancestors of v, and $\text{cdepth}(F)$ to be the maximum $\text{cdepth}(v)$ over all nodes $v \in F$, we get that the total number of relevant subforest of F is at most

$$\sum_{v \in \text{keyroots}(F)} |F_v| = \sum_{v \in F} \text{cdepth}(v) \le \sum_{v \in F} \text{cdepth}(F) = |F| \text{cdepth}(F).$$

This means that given two trees, F and G, of sizes n and m we can compute $\delta(F, G)$ in $O(\text{cdepth}(F) \cdot \text{cdepth}(G) \cdot nm) = O(n_{\text{height}} \cdot m_{\text{height}} \cdot nm)$ time. Zhang and Shasha also proved that for any tree T of size n, $\text{cdepth}(T) \le \min\{n_{\text{height}}, n_{\text{leaves}}\}$, hence the result. In the worst case, this algorithm runs in $O(m^2n^2) = O(n^4)$ time.

2.1.3 Klein's algorithm

Klein's algorithm is based on a recursion similar to Lemma 2.1. Again, we consider forests F and G of sizes $|F| = n \geq |G| = m$. Now, however, instead of recursing always on the rightmost roots of F and G, we recurse on the leftmost roots if $|L_F| \leq |R_F|$ and on the rightmost roots otherwise. In other words, the "direction" of the recursion is determined by the (initially) larger of the two forests. We assume the number of relevant subforests of G is $O(m^2)$; we have already established that this is an upper bound.

We next show that Klein's algorithm yields only $O(n \log n)$ relevant subforests of F. The analysis is based on a technique called *heavy path decomposition* [17, 32]. We mark the root of F as *light*. For each internal node $v \in F$, we pick one of v's children with maximal number of descendants and mark it as *heavy*, and we mark all the other children of v as *light*. We define $\mathrm{ldepth}(v)$ to be the number of light nodes that are proper ancestors of v in F, and $\mathrm{light}(F)$ as the set of all light nodes in F. It is easy to see that for any forest F and vertex $v \in F$, $\mathrm{ldepth}(v) \leq \log |F| + O(1)$. Note that every relevant subforest of F is obtained by some $i \leq |F_v|$ consecutive deletions from F_v for some light node v. Therefore, the total number of relevant subforests of F is at most

$$\sum_{v \in \mathrm{light}(F)} |F_v| \leq \sum_{v \in F} 1 + \mathrm{ldepth}(v) \leq \sum_{v \in F} (\log |F| + O(1)) = O(|F| \log |F|).$$

Thus, we get an $O(m^2 n \log n) = O(n^3 \log n)$ algorithm for computing $\delta(F, G)$.

2.1.4 The decomposition strategy framework

Both Klein's and Zhang and Shasha's algorithms are based on Lemma 2.1. The difference between them lies in the choice of when to recurse on the rightmost roots and when on the leftmost roots. The family of *decomposition strategy* algorithms based on this lemma was formalized by Dulucq and Touzet in [14].

Definition 2.2. *Let F and G be two forests. A strategy is a mapping from pairs (F', G') of subforests of F and G to $\{\mathsf{left}, \mathsf{right}\}$. A decomposition algorithm is an algorithm based on Lemma 2.1 with the directions chosen according to a specific strategy.*

Each strategy is associated with a specific set of recursive calls (or a dynamic programming algorithm). The strategy of Shasha and Zhang's algorithm is $S(F', G') = \mathsf{right}$ for all F', G'. The strategy of Klein's algorithm is $S(F', G') = \mathsf{left}$ if $|L_{F'}| \leq |R_{F'}|$, and $S(F', G') = \mathsf{right}$ otherwise. Notice that Zhang and Shasha's strategy does not depend on the input trees, while Klein's strategy depends only on the larger input tree. Dulucq and Touzet proved a lower bound of $\Omega(mn \log m \log n)$ time for any decomposition strategy algorithm.

The following lemma states that every decomposition algorithm computes the edit distance between every two root-deleted subtrees of F and G.

Lemma 2.3. *Given a decomposition algorithm with strategy S, the pair (F_v°, G_w°) is a relevant subproblem for all $v \in F$ and $w \in G$ regardless of the strategy S.*

Proof. First note that a node $v' \in F_v$ (respectively, $w' \in G_w$) is never deleted or matched before v (respectively, w) is deleted or matched. Consider the following specific sequence of recursive calls:

- Delete from F until v is either the leftmost or the rightmost root.
- Next, delete from G until w is either the leftmost or the rightmost root.

Let (F', G') denote the resulting subproblem. There are four cases to consider.

1. *v and w are the rightmost (leftmost) roots of F' and G', and $S(F', G') =$* right (left).

   Match v and w to get the desired subproblem.

2. *v and w are the rightmost (leftmost) roots of F' and G', and $S(F', G') =$* left (right).

   Note that at least one of F', G' is not a tree (since otherwise this is case (1)). Delete from one which is not a tree. After a finite number of such deletions we have reduced to case (1), either because S changes direction, or because both forests become trees whose roots are v, w.

3. *v is the rightmost root of F', w is the leftmost root of G'.*

   If $S(F', G') = \mathsf{left}$, delete from F'; otherwise delete from G'. After a finite number of such deletions this reduces to one of the previous cases when one of the forests becomes a tree.

4. *v is the leftmost root of F', w is the rightmost root of G'.*

   This case is symmetric to (3).

□

2.2 An $O(m^2 n(1 + \log \frac{n}{m}))$ algorithm

In this section we present our algorithm for computing $\delta(F, G)$ given two trees F and G of sizes $|F| = n \geq |G| = m$. The algorithm recursively uses a decomposition strategy in a divide-and-conquer manner to achieve $O(nm^2(1+\log \frac{n}{m})) = O(n^3)$ running time in the worst case. For clarity we describe the algorithm recursively and analyze its time complexity. In Section 2.4 we prove that the space complexity of a bottom-up non recursive implementation of the algorithm is $O(mn) = O(n^2)$.

2.2.1 Description of the algorithm

Before presenting our algorithm, let us try to develop some intuition. We begin with the observation that Klein's strategy always determines the direction of the recursion according to the F-subforest, even in subproblems where the F-subforest is smaller than the G-subforest. However, it is not straightforward to change this since even if at some stage we decide to choose the direction according to the other forest, we must still make sure that all subproblems previously encountered are entirely solved. At first glance this seems like a real obstacle since apparently we only add new subproblems to those that are already computed. Our key observation is that there are certain subproblems for which it is worthwhile to choose the

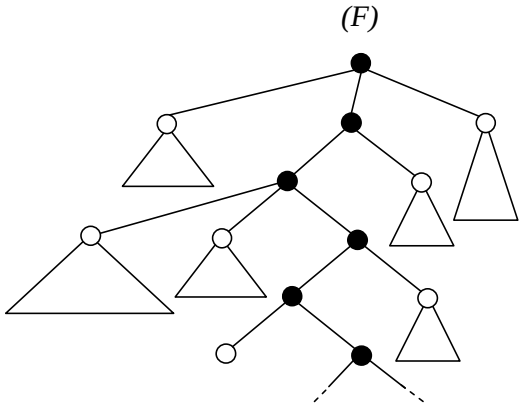

Figure 2.1: A tree F with n nodes. The black nodes belong to the heavy path. The white nodes are in TopLight(F), and the size of each subtree rooted at a white node is at most $\frac{n}{2}$. Note that the root of the tree belongs to the heavy path even though it is light.

direction according to the *currently* larger forest, while for other subproblems we had better keep choosing the direction according to the *originally* larger forest.

The *heavy path* of a tree F is the unique path starting from the root (which is light) along heavy nodes. Consider two trees, F and G, and assume we are given the distances $\delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$. By lemma 2.3, these are relevant subproblems for any decomposition strategy algorithm. How would we go about computing $\delta(F, G)$ in this case? Using Shasha and Zhang's strategy would require $O(|F||G|)$ time, while using Klein's strategy would take $O(|F||G|^2)$ time. Let us focus on Klein's strategy since Shasha and Zhang's strategy is independent of the trees. Note that even if we were not given the distance $\delta(F_u^\circ, G_w^\circ)$ for a node u on the heavy path of F, we would still be able to solve the problem in $O(|F||G|^2)$ time. To see why, note that in order to compute the relevant subproblem $\delta(F_u, G_w)$, we must compute all the subproblems required for solving $\delta(F_u^\circ, G_w^\circ)$ even if $\delta(F_u^\circ, G_w^\circ)$ is given.

We define the set TopLight(F) to be the set of roots of the forest obtained by removing the heavy path of F. Note that TopLight(F) is the set of light nodes with ldepth 1 in F (see the definition of ldepth in Section 2.1.3). This definition is illustrated in Fig. 2.1. It follows from Lemma 2.3 that if we compute $\delta(F_v, G)$ for all $v \in$ TopLight(F), we would also compute all the subproblems $\delta(F_{v'}^\circ, G_w^\circ)$ for any $w \in G$ and v' not on the heavy path of F. Note that Klein's strategy solves $\delta(F_v, G)$ by determining the direction according to F_v even if $|F_v| < |G|$. We observe that we can do better if in such cases we determine the direction according to G. It is important to understand that making the decisions according to the larger forest when solving $\delta(F_v^\circ, G_w^\circ)$ for any $v \in F$ and $w \in G$ (i.e., regardless of whether v is on the heavy path or not) would actually increase the running time. The identification of the set TopLight(F) is crucial for obtaining the improvement.

Given these definitions, the recursive formulation of our algorithm is simply:

(1) *If* $|F| < |G|$*, compute* $\delta(G, F)$ *instead.*

(2) *Recursively compute* $\delta(F_v, G)$ *for all* $v \in \text{TopLight}(F)$*.*

(3) *Compute* $\delta(F, G)$ *using the following decomposition strategy:* $S(F', G') = \mathsf{left}$ *if* F' *is a tree, or if* $\ell_{F'}$ *is not the heavy child of its parent. Otherwise,* $S(F', G') = \mathsf{right}$*. However, do not recurse into subproblems that were previously computed in step (2).*

The algorithm's first step makes sure that F is the larger forest, and the second step makes sure that $\delta(F^\circ_{v'}, G^\circ_w)$ is computed and stored for all v' not in the heavy path of F and for all $w \in G$. Note that the strategy in the third step is equivalent to Klein's strategy for binary trees. For higher valence trees, this variant first makes all left deletions and then all right deletions, while Klein's strategy might change direction many times. They are equivalent in the important sense that both delete the heavy child last. The algorithm is evidentally a decomposition strategy algorithm, since for all subproblems, it either deletes or matches the leftmost or rightmost roots. The correctness of the algorithm follows from the correctness of decomposition strategy algorithms in general.

2.2.2 Time complexity analysis

We show that our algorithm has a worst-case running time of $O(m^2 n(1+\log \frac{n}{m})) = O(n^3)$. We proceed by counting the number of subproblems computed in each step of the algorithm. We call a subproblem trivial if at least one of the forests in this subproblem is empty. Obviously, the number of distinct trivial subproblems is $O(n^2)$. Let $R(F, G)$ denote the number of non-trivial relevant subproblems encountered by the algorithm in the course of computing $\delta(F, G)$. From now on we only count non-trivial subproblems, unless explicitly indicated otherwise.

We observe that any tree F has the following two properties:

$(*)$ $\sum_{v \in \text{TopLight}(F)} |F_v| \leq |F|$. Because F_v and $F_{v'}$ are disjoint for all $v, v' \in \text{TopLight}(F)$.

$(**)$ $|F_v| < \frac{|F|}{2}$ for every $v \in \text{TopLight}(F)$. Otherwise v would be a heavy node.

In step (2) we compute $\delta(F_v, G)$ for all $v \in \text{TopLight}(F)$. Hence, the number of subproblems encountered in this step is $\sum_{v \in \text{TopLight}(F)} R(F_v, G)$. For step (3), we bound the number of relevant subproblems by multiplying the number of relevant subforests in F and in G. For G, we count all possible $O(|G|^2)$ subforests obtained by left and right deletions. Note that for any node v' not in the heavy path of F, the subproblem obtained by matching v' with any node w in G was already computed in step (2). This is because any such v' is contained in F_v for some $v \in \text{TopLight}(F)$, so $\delta(F^\circ_{v'}, G^\circ_w)$ is computed in the course of computing $\delta(F_v, G)$ (by Lemma 2.3). Furthermore, note that in step (3), a node v on the heavy path of F cannot be matched or deleted until the remaining subforest of F is precisely the tree F_v. At this point, both matching v or deleting v result in the same new relevant subforest F°_v. This means

that we do not have to consider matchings of nodes when counting the number of relevant subproblems in step (3). It suffices to consider only the $|F|$ subforests obtained by deletions according to our strategy. Thus, the total number of new subproblems encountered in step (3) is bounded by $|G|^2|F|$.

We have established that if $|F| \geq |G|$ then

$$R(F,G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$$

and if $|F| < |G|$ then

$$R(F,G) \leq |F|^2|G| + \sum_{w \in \text{TopLight}(G)} R(F, G_w)$$

We first show, by a crude estimate, that this leads to an $O(n^3)$ running time. Later, we analyze the dependency on m and n accurately.

Lemma 2.4. $R(F,G) \leq 4(|F||G|)^{3/2}$.

Proof. We proceed by induction on $|F| + |G|$. In the base case, $|F| + |G| = 0$, so both forests are empty and $R(F,G) = 0$. For the inductive step there are two symmetric cases. If $|F| \geq |G|$ then $R(F,G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$. Hence, by the induction hypothesis,

$$\begin{aligned}
R(F,G) &\leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} 4(|F_v||G|)^{3/2} \\
&= |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v|^{3/2} \\
&\leq |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v| \quad \max_{v \in \text{TopLight}(F)} \sqrt{|F_v|} \\
&\leq |G|^2|F| + 4|G|^{3/2}|F|\sqrt{\frac{|F|}{2}} = |G|^2|F| + \sqrt{8}(|F||G|)^{3/2} \leq 4(|F||G|)^{3/2}
\end{aligned}$$

Here we have used facts $(*)$ and $(**)$ and the fact that $|F| \geq |G|$. The case where $|F| < |G|$ is symmetric. □

This crude estimate gives a worst-case running time of $O(n^3)$. We now analyze the dependence on m and n more accurately. Along the recursion defining the algorithm, we view step (2) as only making recursive calls, but not producing any relevant subproblems. Rather, every new relevant subproblem is created in step (3) for a unique recursive call of the algorithm. So when we count relevant subproblems, we sum the number of new relevant subproblems encountered in step (3) over all recursive calls to the algorithm. We define sets $A, B \subseteq F$ as follows:

$$\begin{aligned}
A &= \{a \in \text{light}(F) : |F_a| \geq m\} \\
B &= \{b \in F - A : b \in \text{TopLight}(F_a) \text{ for some } a \in A\}
\end{aligned}$$

Note that the root of F belongs to A. Intuitively, the nodes in both A and B are exactly those for which recursive calls are made with the entire G tree. The nodes in B are the last ones, along the recursion, for which such recursive calls are made. We count separately:

(i) the relevant subproblems created in just step (3) of recursive calls $\delta(F_a, G)$ for all $a \in A$, and

(ii) the relevant subproblems encountered in the entire computation of $\delta(F_b, G)$ for all $b \in B$ (i.e., $\sum_{b \in B} R(F_b, G)$).

Together, this counts all relevant subproblems for the original $\delta(F, G)$. To see this, consider the original call $\delta(F, G)$. Certainly, the root of F is in A. So all subproblems generated in step (3) of $\delta(F, G)$ are counted in (i). Now consider the recursive calls made in step (2) of $\delta(F, G)$. These are precisely $\delta(F_v, G)$ for $v \in \text{TopLight}(F)$. For each $v \in \text{TopLight}(F)$, notice that v is either in A or in B; it is in A if $|F_v| \geq m$, and in B otherwise. If v is in B, then all subproblems arising in the entire computation of $\delta(F_v, G)$ are counted in (ii). On the other hand, if v is in A, then we are in analogous situation with respect to $\delta(F_v, G)$ as we were in when we considered $\delta(F, G)$ (i.e., we count separately the subproblems created in step (3) of $\delta(F_v, G)$ and the subproblems coming from $\delta(F_u, G)$ for $u \in \text{TopLight}(F_v)$).

Earlier in this section, we saw that the number of subproblems created in step (3) of $\delta(F, G)$ is $|G|^2|F|$. In fact, for any $a \in A$, by the same argument, the number of subproblems created in step (3) of $\delta(F_a, G)$ is $|G|^2|F_a|$. Therefore, the total number of relevant subproblems of type (i) is $|G|^2 \sum_{a \in A} |F_a|$. For $v \in F$, define $\text{depth}_A(v)$ to be the number of proper ancestors of v that lie in the set A. We claim that $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$ for all $v \in F$. To see this, consider any sequence $a_0, \ldots, a_k$ in A where a_i is a descendent of a_{i-1} for all $i \in [1, k]$. Note that $|F_{a_i}| \leq \frac{1}{2}|F_{a_{i-1}}|$ for all $i \in [1, k]$ since the a_is are light nodes. Also note that $F_{a_0} \leq n$ and that $|F_{a_k}| \geq m$ by the definition of A. It follows that $k \leq \log \frac{n}{m}$, i.e., A contains no sequence of descendants of length $> 1 + \log \frac{n}{m}$. So clearly every $v \in F$ has $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$.

We now have the number of relevant subproblems of type (i) as

$$|G|^2 \sum_{a \in A} |F_a| \leq m^2 \sum_{v \in F} 1 + \text{depth}_A(v) \leq m^2 \sum_{v \in F} (2 + \log \frac{n}{m}) = m^2 n (2 + \log \frac{n}{m}).$$

The relevant subproblems of type (ii) are counted by $\sum_{b \in B} R(F_b, G)$. Using Lemma 2.4, we have

$$\begin{aligned} \sum_{b \in B} R(F_b, G) &\leq 4|G|^{3/2} \sum_{b \in B} |F_b|^{3/2} \\ &\leq 4|G|^{3/2} \sum_{b \in B} |F_b| \max_{b \in B} \sqrt{|F_b|} \\ &\leq 4|G|^{3/2}|F|\sqrt{m} = 4m^2 n. \end{aligned}$$

Here we have used the facts that $|F_b| < m$ and $\sum_{b \in B} |F_b| \leq |F|$ (since the trees F_b are disjoint for different $b \in B$). Therefore, the total number of relevant subproblems for $\delta(F, G)$ is at most $m^2 n(2 + \log \frac{n}{m}) + 4m^2 n = O(m^2 n(1 + \log \frac{n}{m}))$. This implies:

Theorem 2.5. *The running time of the algorithm is* $O(m^2 n(1 + \log \frac{n}{m}))$. □

2.3 A tight lower bound for decomposition algorithms

In this section we present a lower bound on the worst case running time of decomposition strategy algorithms. We first give a simple proof of an $\Omega(m^2 n)$ lower bound. In the case

where $m = \Theta(n)$, this gives a lower bound of $\Omega(n^3)$ which shows that our algorithm is worst-case optimal among all decomposition algorithms. To prove that our algorithm is worst-case optimal for any $m \leq n$, we analyze a more complicated scenario that gives a lower bound of $\Omega(m^2 n(1 + \log \frac{n}{m}))$, matching the running time of our algorithm, and improving the previous best lower bound of $\Omega(nm \log n \log m)$ time [14].

In analyzing strategies we will use the notion of a *computational path*, which corresponds to a specific sequence of recursion calls. Recall that for all subforest-pairs (F', G'), the strategy S determines a direction: either right or left. The recursion can either delete from F' or from G' or match. A computational path is the sequence of operations taken according to the strategy in a specific sequence of recursive calls. For convenience, we sometimes describe a computational path by the sequence of subproblems it induces, and sometimes by the actual sequence of operations: either "delete from the F-subforest", "delete from the G-subforest", or "match".

We now turn to the $\Omega(m^2 n)$ lower bound on the number of relevant subproblems for any strategy.

Lemma 2.6. *For any decomposition algorithm, there exists a pair of trees (F, G) with sizes n, m respectively, such that the number of relevant subproblems is $\Omega(m^2 n)$.*

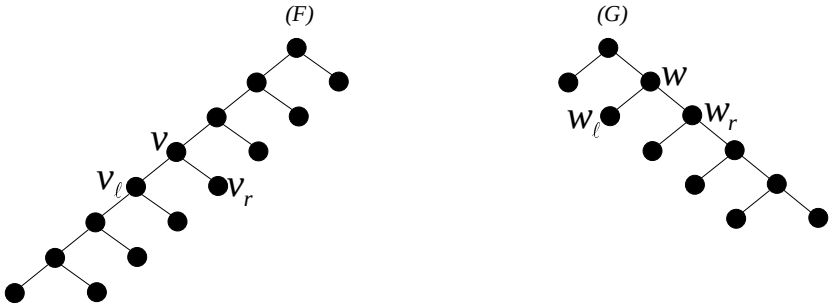

Figure 2.2: The two trees used to prove an $\Omega(m^2 n)$ lower bound (Lemma 2.6).

Proof. Let S be the strategy of the decomposition algorithm, and consider the trees F and G depicted in Fig. 2.3. According to lemma 2.3, every pair (F_v°, G_w°) where $v \in F$ and $w \in G$ is a relevant subproblem for S. Focus on such a subproblem where v and w are internal nodes of F and G. Denote v's right child by v_r and w's left child by w_ℓ. Note that F_v° is a forest whose rightmost root is the node v_r. Similarly, G_w° is a forest whose leftmost root is w_ℓ. Starting from (F_v°, G_w°), consider the computational path $c_{v,w}$ that deletes from F whenever the strategy says left and deletes from G otherwise. In both cases, neither v_r nor w_ℓ is deleted unless one of them is the only node left in the forest. Therefore, the length of this computational path is at least $\min\{|F_v|, |G_w|\} - 1$. Recall that for each subproblem (F', G') along $c_{v,w}$, the rightmost root of F' is v_r and the leftmost root of G' is w_ℓ. It follows that for every two distinct pairs $(v_1, w_1) \neq (v_2, w_2)$ of internal nodes in F and G, the relevant subproblems occurring along the computational paths c_{v_1,w_1} and c_{v_2,w_2} are disjoint. Since there are $\frac{n}{2}$ and $\frac{m}{2}$ internal nodes in F and G respectively, the total number of subproblems

along the $c_{v,w}$ computational paths is given by:

$$\sum_{(v,w) \text{ internal nodes}} \min\{|F_v|, |G_w|\} - 1 = \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{\frac{m}{2}} \min\{2i, 2j\} = \Omega(m^2 n)$$

□

The $\Omega(m^2 n)$ lower bound established by Lemma 2.6 is tight if $m = \Theta(n)$, since in this case our algorithm achieves an $O(n^3)$ running time. To establish a tight bound when m is not $\Theta(n)$, we use the following technique for counting relevant subproblems. We associate a subproblem consisting of subforests (F', G') with the unique pair of vertices (v, w) such that F_v, G_w are the smallest trees containing F', G' respectively. For example, for nodes v and w, each with at least two children, the subproblem (F_v°, G_w°) is associated with the pair (v, w). Note that all subproblems encountered in a computational path starting from (F_v°, G_w°) until the point where either forest becomes a tree are also associated with (v, w).

Lemma 2.7. *For every decomposition algorithm, there exists a pair of trees (F, G) with sizes $n \geq m$ such that the number of relevant subproblems is $\Omega(m^2 n \log \frac{n}{m})$.*

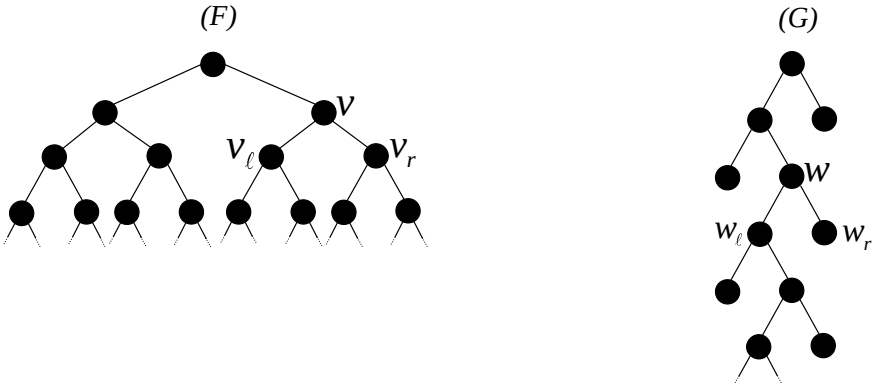

Figure 2.3: The two trees used to prove $\Omega(m^2 n \log \frac{n}{m})$ lower bound (Lemma 2.7).

Proof. Consider the trees illustrated in Fig. 2.3. The n-sized tree F is a complete balanced binary tree, and G is a "zigzag" tree of size m. Let w be an internal node of G with a single leaf w_r as its right subtree and w_ℓ as a left child. Denote $m' = |G_w|$. Let v be a node in F such that F_v is a tree of size $n' + 1$ where $n' \geq 4m \geq 4m'$. Denote v's left and right children v_ℓ and v_r respectively. Note that $|F_{v_\ell}| = |F_{v_r}| = \frac{n'}{2}$

Let S be the strategy of the decomposition algorithm. We aim to show that the total number of relevant subproblems associated with (v, w) or with (v, w_ℓ) is at least $\frac{n'}{4}(m' - 2)$. Starting from the subproblem (F_v°, G_w°), which is relevant by lemma 2.3, let c be the computational path that always deletes from F (no matter whether S says left or right). We consider two complementary cases.

CASE 1: $\frac{n'}{4}$ *left deletions occur in the computational path c, and at the time of the* $\frac{n'}{4}$*th left deletion, there were fewer than* $\frac{n'}{4}$ *right deletions.*

We define a set of new computational paths $\{c_j\}_{1 \le j \le \frac{n'}{4}}$ where c_j deletes from F up through the jth left deletion, and thereafter deletes from F whenever S says right and from G whenever S says left. At the time the jth left deletion occurs, at least $\frac{n'}{4} \ge m' - 2$ nodes remain in F_{v_r} and all $m' - 2$ nodes are present in G_{w_ℓ}. So on the next $m' - 2$ steps along c_j, neither of the subtrees F_{v_r} and G_{w_ℓ} is totally deleted. Thus, we get $m' - 2$ distinct relevant subproblems associated with (v, w). Notice that in each of these subproblems, the subtree F_{v_ℓ} is missing exactly j nodes. So we see that, for different values of $j \in [1, \frac{n'}{4}]$, we get disjoint sets of $m' - 2$ relevant subproblems. Summing over all j, we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w).

CASE 2: $\frac{n'}{4}$ *right deletions occur in the computational path c, and at the time of the* $\frac{n'}{4}$*th right deletion, there were fewer than* $\frac{n'}{4}$ *left deletions.*

We define a different set of computational paths $\{\gamma_j\}_{1 \le j \le \frac{n'}{4}}$ where γ_j deletes from F up through the jth right deletion, and thereafter deletes from F whenever S says left and from G whenever S says right (i.e., γ_j is c_j with the roles of left and right exchanged). Similarly as in case 1, for each $j \in [1, \frac{n'}{4}]$ we get $m' - 2$ distinct relevant subproblems in which F_{v_r} is missing exactly j nodes. All together, this gives $\frac{n'}{4}(m' - 2)$ distinct subproblems. Note that since we never make left deletions from G, the left child of w_ℓ is present in all of these subproblems. Hence, each subproblem is associated with either (v, w) or (v, w_ℓ).

In either case, we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w) or (v, w_ℓ). To get a lower bound on the number of problems we sum over all pairs (v, w) with G_w being a tree whose right subtree is a single node, and $|F_v| \ge 4m$. There are $\frac{m}{4}$ choices for w corresponding to tree sizes $4j$ for $j \in [1, \frac{m}{4}]$. For v, we consider all nodes of F whose distance from a leaf is at least $\log(4m)$. For each such pair we count the subproblems associated with (v, w) and (v, w_ℓ). So the total number of relevant subproblems counted in this way is

$$\sum_{v,w} \frac{|F_v|}{4}(|G_w| - 2) = \frac{1}{4}\sum_{v} |F_v| \sum_{j=1}^{\frac{m}{4}} (4j - 2)$$

$$= \frac{1}{4}\sum_{i=\log 4m}^{\log n} \frac{n}{2^i} \cdot 2^i \sum_{j=1}^{\frac{m}{4}} (4j - 2) = \Omega(m^2 n \log \frac{n}{m})$$

□

Theorem 2.8. *For every decomposition algorithm and* $n \ge m$*, there exist trees* F *and* G *of sizes* $\Theta(n)$ *and* $\Theta(m)$ *such that the number of relevant subproblems is* $\Omega(m^2 n(1 + \log \frac{n}{m}))$.

Proof. If $m = \Theta(n)$ then this bound is $\Omega(m^2 n)$ as shown in Lemma 2.6. Otherwise, this bound is $\Omega(m^2 n \log \frac{n}{m})$ which was shown in Lemma 2.7. □

2.4 Implementing the algorithm in $O(mn)$ space

The recursion presented in Section 2.2 for computing $\delta(F, G)$ translates into an $O(m^2 n(1 + \log \frac{n}{m}))$ time and space algorithm. In this section we reduce the space complexity of this algorithm to $O(mn)$. We achieve this by ordering the relevant subproblems in such a way that we need to record the edit distance of only $O(mn)$ relevant subproblems at any point in time. For simplicity, we assume the input trees F and G are binary. At the end of this section, we show how to remove this assumption.

The algorithm TED fills a global n by m table Δ with values $\Delta_{vw} = \delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$.

TED(F, G)

1: If $|F| < |G|$ do TED(G, F).

2: For every $v \in$ TopLight(F) do TED(F_v, G).

3: Fill Δ_{vw} for all $v \in$ HeavyPath(F) and $w \in G$.

Step 3 runs in $O(|F||G|^2)$ time and assumes Δ_{vw} has already been computed in step 2 for all $v \in F -$ HeavyPath(F) and $w \in G$ (see Section 2.2). In the remainder of this section we prove that it can be done in $O(|F||G|)$ space.

In step 3 we go through the nodes $v_1, \ldots, v_t$ on the heavy path of F starting with the leaf v_1 and ending with the root v_t where $t = |\text{HeavyPath}(F)|$. Throughout the computation we maintain a table T of size $|G|^2$. When we start handling v_p $(1 \leq p \leq t)$, the table T holds the edit distance between $F_{v_{p-1}}$ and all possible subforests of G. We use these values to calculate the edit distance between F_{v_p} and all possible subforests of G and store the newly computed values back into T. We refer to the process of updating the entire T table (for a specific v_p) as a period. Before the first period, in which F_{v_1} is a leaf, we set T to hold the edit distance between $\emptyset$ and G' for all subforests G' of G (this is just the cost of deleting G').

Note that since we assume F is binary, during each period the direction of our strategy does not change. Let left(v) and right(v) denote the left and right children of a node v. If $v_{p-1} =$ right(v_p), then our strategy is left throughout the period of v_p. Otherwise it is right. We now explain what goes into computing a period. This process, which we refer to as ComputePeriod(v_p), both uses and updates tables T and Δ. At the heart of this procedure is a dynamic program. Throughout this description we assume that our strategy is left. The right analogue is obvious. We now describe two simple subroutines that are called by ComputePeriod(v_p).

If $F_{v_{p-1}}$ can be obtained from F_{v_p} by a series of left deletions, the *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence $F_{v_{p-1}} = F_0, F_1 \ldots, F_k = F_{v_p}$ such that $F_{k'-1} = F_{k'} - \ell_{F_{k'}}$ for all $1 \leq k' \leq k = |F_{v_p}| - |F_{v_{p-1}}|$. This concept is illustrated in Fig. 2.4. The subroutine IntermediateLeftSubforestEnum$(F_{v_{p-1}}, F_{v_p})$ associates every $F_{k'}$ with $\ell_{F_{k'}}$ and lists them in the order of the intermediate left subforest enumerations with respect to $F_{v_{p-1}}$ and F_{v_p}. This is the order in which we access the nodes and subforests during the execution of

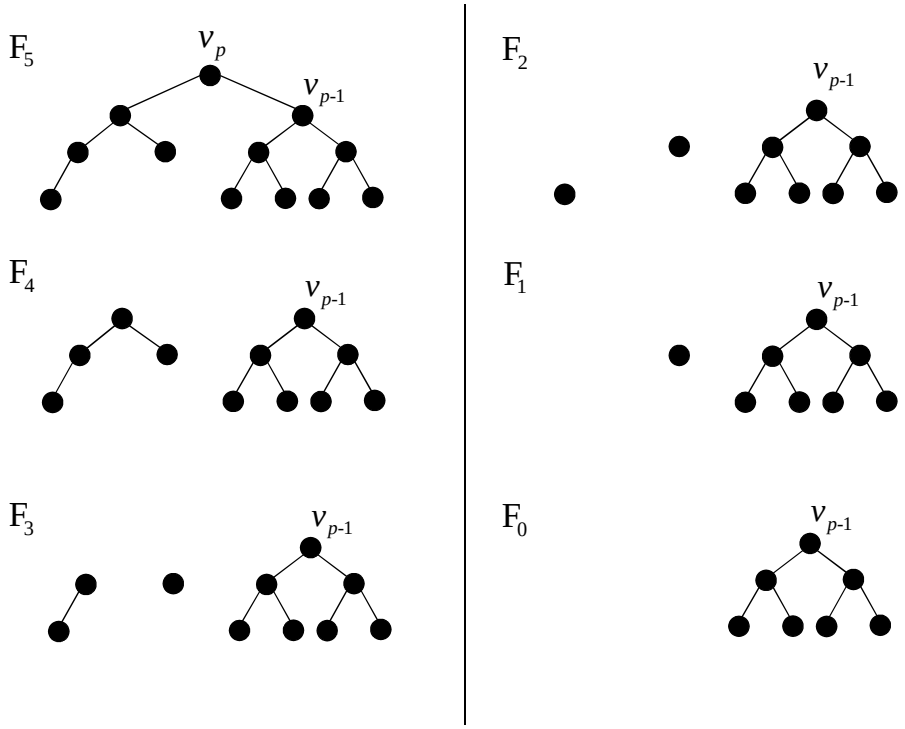

Figure 2.4: The *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence of forests $F_{v_{p-1}} = F_0, F_1, \ldots, F_5 = F_{v_p}$.

ComputePeriod(v_p), so each access will be done in constant time. The intermediate left and right subforest enumerations required for all periods (i.e., for all of the v_ps along the heavy path) can be prepared once in $O(|F|)$ time and space by performing $|F|$ deletions on F according to our strategy and listing the deleted vertices in reverse order.

Let $w_0, w_1, \ldots, w_{|G|-1}$ be the right-to-left preorder traversal of a tree G. We define $G_{i,0}$ as the forest obtained from G by making i right deletions. Notice that the rightmost tree in $G_{i,0}$ is G_{w_i} (the subtree of G rooted at w_i). We further define $G_{i,j}$ as the forest obtained from G by first making i right deletions and then making j left deletions. Let $j(i)$ be the number of left deletions required to turn $G_{i,0}$ into the tree G_{w_i}. We can easily compute $j(0), \ldots, j(|G|-1)$ in $O(|G|)$ time and space by noticing that $j(i) = |G| - i - |G_{w_i}|$. Note that distinct nonempty subforests of G are represented by distinct $G_{i,j}$s for $0 \leq i \leq |G|-1$ and $0 \leq j \leq j(i)$. For convenience, we sometimes refer to $G^\circ_{w_i}$ as $G_{i,j(i)+1}$ and sometimes as the equivalent $G_{i+1,j(i)}$. The two subforest are the same since the forest $G_{i,j(i)}$ is the tree G_{w_i}, so making another left deletion, namely $G_{i,j(i)+1}$ is the same as first making an extra right deletion, namely $G_{i+1,j(i)}$. The *left subforest enumeration* of all nonempty subforests of G is defined as

$$G_{|G|-1,j(|G|-1)}, \ldots, G_{|G|-1,0} \ , \ \cdots \ , \ G_{2,j(2)}, \ldots, G_{2,0} \ , \ G_{1,j(1)}, \ldots, G_{1,0} \ , \ G_{0,0}$$

The subroutine LeftSubforestEnum(G) associates every $G_{i,j}$ with the left deleted vertex $\ell_{G_{i,j}}$ and lists them in the order of the left subforest enumeration with respect to G, so

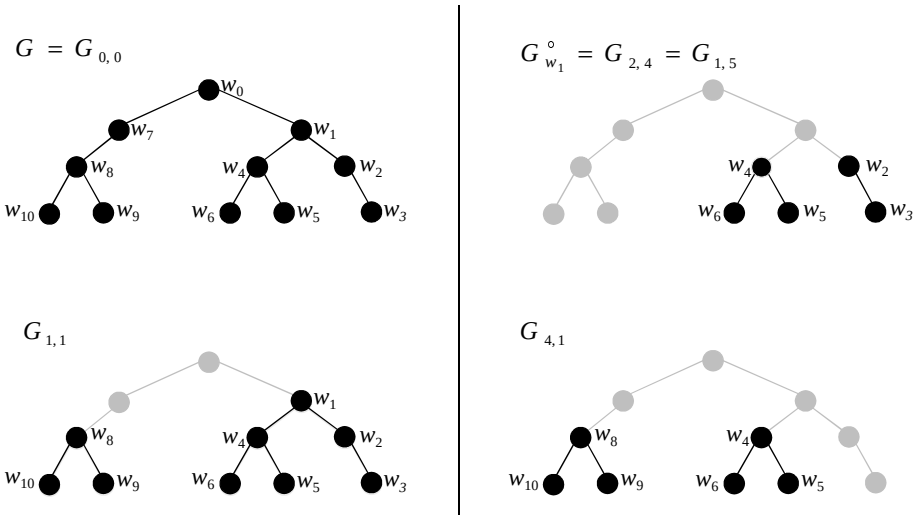

Figure 2.5: The indexing of various subforests (shown in solid black) of G (shown in gray). The right-to-left preorder traversal of G is $w_0, w_1, \ldots, w_{|G|-1}$. The subforest $G_{i,j}$ is the forest obtained from G by first making i right deletions and then making j left deletions. All nonempty subforests of G are captured by all $0 \leq i \leq |G|-1$ and $0 \leq j \leq j(i) = |G|-i-|G_{w_i}|$. The index of G itself is $G_{0,0}$. In the special case of $G^{\circ}_{w_1} = G_{2,4}$ we sometimes use the equivalent index $G_{1,5}$.

that we will be able to access $\ell_{G_{i,j}}$ in this order in constant time per access. This procedure takes $O(|G|)$ time and space for each i by performing first i right deletions and then j left deletions, and listing the left deleted vertices in reverse order. The entire subroutine therefore requires $O(|G|^2)$ time and space. The above definitions are illustrated in Fig. 2.5. There are obvious "right" analogues of everything we have just defined.

The pseudocode for ComputePeriod(v_p) is given below. As we already mentioned, at the beginning of the period for v_p, the table T stores the distance between $F_{v_{p-1}}$ and all subforests of G and our goal is to update T with the distance between F_{v_p} and any subforest of G. For each value of i in decreasing order (the loop in Line 3), we compute a temporary table S of the distances between the forests $F_{k'}$ in the intermediate left subforest enumeration with respect to $F_{v_{p-1}}$ and F_{v_p} and the subforest $G_{i,j}$ for $0 \leq j \leq j(i)$ in the left subforest enumeration of G. Clearly, there are $O(|F||G|)$ such subproblems. The computation is done for increasing values of k' and decreasing values of j according to the basic relation in Line 4. Once the entire table S is computed, we update T, in Line 5, with the distances between $F_k = F_{v_p}$ and $G_{i,j}$ for all $0 \leq j \leq j(i)$. Note that along this computation we encounter the subproblem which consists of the root-deleted-trees $F^{\circ}_{v_p} = F_{k-1}$ and $G^{\circ}_{w_{i-1}} = G_{i,j(i-1)}$. In line 7, we store the value for this subproblem in $\Delta_{v_p,w_{i-1}}$. Thus, going over all possible values for i, the procedures updates the entire table T and all the appropriate entries in Δ, and completes a single period.

ComputePeriod(v_p)

Overwrites T with values $\delta(F_{v_p}, G')$ for all subforests G' of G, and fills in Δ with values $\delta(F^\circ_{v_p}, G^\circ_w)$ for every $w \in G$.

Assumes T stores $\delta(F_{v_{p-1}}, G')$ for all subforests G' of G, and $v_{p-1} = \text{right}(v_p)$ (if $v_{p-1} = \text{left}(v_p)$ then reverse roles of "left" and "right" below).

1: $F_0, \ldots, F_k \leftarrow \text{IntermediateLeftSubforestEnum}(F_{v_{p-1}}, F_{v_p})$

2: $G_{|G|-1,j(|G|-1)}, \ldots, G_{0,0} \leftarrow \text{LeftSubforestEnum}(G)$

3: for $i = |G| - 1, \ldots, 0$ do

4: compute table $S \leftarrow \Big(\delta(F_{k'}, G_{i,j})\Big)_{\substack{k'=1,\ldots k \\ j=j(i),\ldots,0}}$ via the dynamic program:

$$\delta(F_{k'}, G_{i,j}) = \min \begin{cases} c_{\text{del}}(\ell_{F_{k'}}) + \delta(F_{k'-1}, G_{i,j}), \\ c_{\text{del}}(\ell_{G_{i,j}}) + \delta(F_{k'}, G_{i,j+1}), \\ c_{\text{match}}(\ell_{F_{k'}}, \ell_{G_{i,j}}) + \delta(L^\circ_{F_{k'}}, L^\circ_{G_{i,j}}) \\ \qquad\qquad + \delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}}) \end{cases}$$

5: $T \leftarrow \delta(F_{v_p}, G_{i,j})$ for all $0 \le j \le j(i)$, via S

6: $Q \leftarrow \delta(F_{k'}, G_{i,j(i-1)})$ for all $1 \le k' \le k$, via S

7: $\Delta \leftarrow \delta(F^\circ_{v_p}, G_{i,j(i-1)})$ via S

8: end do

The correctness of ComputePeriod(v_p) follows from Lemma 2.1. However, we still need to show that all the required values are available when needed in the execution of Line 4. Let us go over the different subproblems encountered during this computation and show that each of them is available when required along the computation.

$\delta(F_{k'-1}, G_{i,j})$:

- when $k' = 1$, F_0 is $F_{v_{p-1}}$, so it is already stored in T from the previous period.
- for $k' > 1$, $\delta(F_{k'-1}, G_{i,j})$ was already computed and stored in S, since we go over values of k' in increasing order.

$\delta(F_{k'}, G_{i,j+1})$:

- when $j = j(i)$ and $i + j(i) = |G| - 1$, then $G_{i,j(i)+1} = \emptyset$ so $\delta(F_{k'}, \emptyset)$ is the cost of deleting $F_{k'}$, which may be computed in advance for all subforests within the same time and space bounds.
- when $j = j(i)$ and $i + j(i) < |G| - 1$, recall that $\delta(F_{k'}, G_{i,j(i)+1})$ is equivalent to $\delta(F_{k'}, G_{i+1,j(i)})$ so this problem was already computed, since we loop over the values of

i in decreasing order. Furthermore, this problem was stored in the array Q when line 6 was executed for the previous value of i.

- when $j < j(i)$, $\delta(F_{k'}, G_{i,j+1})$ was already computed and stored in S, since we go over values of j in decreasing order.

$\delta(L^\circ_{F_{k'}}, L^\circ_{G_{i,j}})$:

- this value was computed previously (in step 2 of TED) as Δ_{vw} for some $v \in F - \text{HeavyPath}(F)$ and $w \in G$.

$\delta(F_{k'}\text{–}L_{F_{k'}}, G_{i,j}\text{–}L_{G_{i,j}})$:

- if $j \neq j(i)$ then $F_{k'} - L_{F_{k'}} = F_{k''}$ where $k'' = k' - |L_{F_{k'}}|$ and $G_{i,j} - L_{G_{i,j}} = G_{i,j'}$ where $j' = j + |L_{G_{i,j}}|$, so $\delta(F_{k''}, G_{i,j'})$ was already computed and stored in S earlier in the loop.

- if $j = j(i)$, then $G_{i,j}$ is a tree, so $G_{i,j} = L_{G_{i,j}}$. Hence, $\delta(F_{k'}\text{–}L_{F_{k'}}, G_{i,j}\text{–}L_{G_{i,j}})$ is simply the cost of deleting $F_{k''}$.

The space required by this algorithm is evidently $O(|F||G|)$ since the size of S is at most $|F||G|$, the size of T is at most $|G|^2$, the size of Q is at most $|F|$, and the size of Δ is $|F||G|$. The time complexity does not change, since we still handle each relevant subproblem exactly once, in constant time per relevant subproblem.

Note that in the last time ComputePeriod() is called, the table T stores (among other things) the edit distance between the two input trees. In fact, our algorithm computes the edit distance between any subtree of F and any subtree of G. We could store these values without changing the space complexity.

This concludes the description of our $O(mn)$ space algorithm. All that remains to show is why we may assume the input trees are binary. If they are not binary, we construct in $O(m+n)$ time binary trees F' and G' where $|F'| \leq 2n$, $|G'| \leq 2m$, and $\delta(F,G) = \delta(F',G')$ using the following procedure: Pick a node $v \in F$ with $k > 2$ children which are, in left to right order, $\text{left}(v) = v_1, v_2, \ldots, v_k = \text{right}(v)$. We leave $\text{left}(v)$ as it is, and set $\text{right}(v)$ to be a new node with a special label ε whose children are $v_2, v_3, \ldots, v_k$. To ensure this does not change the edit distance, we set the cost of deleting ε to zero, and the cost of relabeling ε to ∞. The same procedure is applied to G as well. We note that another way to remove the binary trees assumption is to modify ComputePeriod() to work directly with non-binary trees at the cost of slightly complicating it. This can be done by splitting it into two parts, where one handles left deletions and the other right deletions.

New Upper Bounds for The Largest Common Subtree Problem

3.1 Preliminaries

In order to discuss the Largest Common Subtree problem we need some definitions in addition to those of section 2.1

For a pair of trees F, G, two nodes $v \in F, w \in G$ with the same label are called a *match pair*. 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).

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 $i \leq j$, 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.

The tree LCS problem can be formulated in terms of matchings[1]. 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. Therefore, the LCS problem is equivalent to finding a maximal size matching.

[1]Matchings can also be used to formulate the tree edit distance problem of Chapter 2, as well as other variants of the problem.

For two forests F and G, let $\mathrm{LCS}_R(F,G)$ (resp., $\mathrm{LCS}_L(F,G)$) 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 $\mathrm{LCS}_R(F,G) = 0$. Clearly, $\mathrm{LCS}_R(F,G) \leq \mathrm{LCS}(F,G)$ and $\mathrm{LCS}_L(F,G) \leq \mathrm{LCS}(F,G)$.

Lemma 3.1. *If F and G are trees whose roots have equal labels then* $\mathrm{LCS}_R(F,G) = \mathrm{LCS}_L(F,G) = \mathrm{LCS}(F,G)$.

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 $\mathrm{LCS}(F,G)$ in which both r and r' are matched. Let M be an LCS matching between F and G of size $\mathrm{LCS}(F,G)$. 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 $\mathrm{LCS}(F,G)+1$, 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 $\mathrm{LCS}(F,G)$. □

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. The *heavy path decomposition* presented in section refklein is an example for a path decomposition of a tree. The *main* path P of the heavy path decomposition is exactly the heavy path of section 2.2.1

A *successor data-structure* is a data-structures that stores a set of elements S with a key for each element and supports the following operations: (1) insert(S,x): inserts x into S (2) delete(S,x): removes x from S (3) pred(S,k): returns the element $x \in S$ with maximal key such that key$(x) \leq k$ (4) succ(S,k): returns the element $x \in S$ with minimal key such that key$(x) \geq k$. Van Emde Boas presented a data structure [36] that supports each of these operations in $O(\lg\lg u)$ time, where the set of legal keys is $\{1,2,\ldots,u\}$.

3.2 An $O(r \cdot \mathrm{height}(F) \cdot \mathrm{height}(G) \cdot \lg\lg m)$ algorithm

In this section we present an $O(r \cdot \mathrm{height}(F) \cdot \mathrm{height}(G) \cdot \lg\lg m)$ time algorithm for computing the LCS of two trees F and G of sizes n and m and heights height(F) and height(G) respectively. The relation between this algorithm and Zhang and Shasha's $O(nm \cdot \mathrm{height}(F) \cdot \mathrm{height}(G))$ time algorithm [30] is similar to the relation between Hunt and Szymanski's $O(r \lg\lg m)$ time algorithm [21] and Wagner and Fischer's $O(mn)$ time algorithm [37] in the string LCS world.

We describe an algorithm based on that of Zhang and Shasha using an alignment graph. This approach was also used in [34, 4, 5]. 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 $1 \leq i \leq 2n$ and $1 \leq j \leq 2m$. 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:

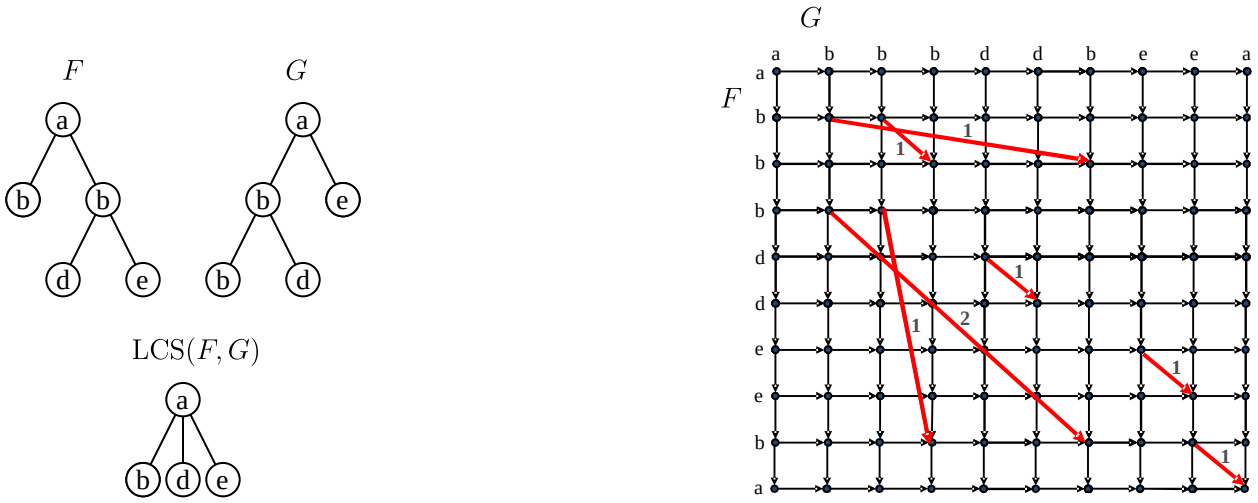

Figure 3.1: Example of an alignment graph for two trees F and G.

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.

Figure 3.1 shows an example of an alignment graph. 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 3.2. *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 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,

weight$(p') \leq \text{LCS}(F[1..i'], G[1..j'])$. Therefore, there is an LCS mapping M' between $F[1..i']$ and $G[1..j']$ of size weight(p'). There are three cases, depending on the type of e.

1. If e is an edge of the first type above, then weight$(e) = 0$, and $M = M'$ is the corresponding 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 $\text{LCS}(F_v, G_w)$. By construction, weight$(e) = \text{LCS}(F_v, G_w)$. 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 weight(p') + weight(e) = 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 weight$(p') + 1$ = 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 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 3.2, 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 head(e) that passes through e.

Lemma 3.3. 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. Following the proof of Lemma 3.2, we have that for every path p from $(1, 1)$ that ends at $\text{head}(e)$ and 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 3.1, we may assume that v is matched to w in M''. It follows that $\text{score}(e) \leq \text{LCS}_R(F[1..\text{head}(e)_1], G[1..\text{head}(e)_2])$.

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 3.2 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 3.1 and 3.3 we have that $\text{LCS}(F, G) = \text{score}((2n - 1, 2m - 1) \to (2n, 2m))$. We now describe a procedure 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 procedure handles the rows of the alignment graph in increasing order. For row i, it first handles all edges e with $\text{head}(e)_1 = i$. An important invariant is that when handling row i, for all j, $\text{pred}(S, j)$ stores the last edge from $E_{F,G}$ in a maximal weight path from $(1, 1)$ to (i, j) among all the paths whose nonzero weight edges were already considered by the algorithm. To maintain the invariant, when considering an edge e whose head is (i, j), if $\text{score}(e) > \text{score}(\text{pred}(S, j))$, then e is a better way to reach (i, j) than $\text{pred}(S, j)$. Hence, we insert e into S. In this case we also check if $\text{score}(\text{succ}(S, j + 1)) \leq \text{score}(e)$. If so, e is also better than $\text{succ}(S, j + 1)$ so we delete $\text{succ}(S, j + 1)$ from S. After handling all edges whose head is in row i, $\text{score}(\text{pred}(S, j))$ is exactly $\text{LCS}(F[1..i], G[1..j],)$. Therefore, we can now update the scores of all the edges e with $\text{tail}(e) = (i, j)$ with $\text{weight}(e) + \text{score}(\text{pred}(S, j))$. The pseudocode for the procedure is as follows (we assume that $\text{score}(\text{NULL}) = 0$).

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

To analyze the running time, 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 [36] 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 the above procedure recursively on every match pair we get that the total time complexity is bounded

by

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

3.3 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 [34, 4, 5] 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 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 $I_{\mathsf{left}} = \{1, \ldots, n\} \setminus I_{\mathsf{right}}$.

The alignment graph $B_{F,G}$ of trees F and G is defined as follows. 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 $i \in I_{\mathsf{right}}$ 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 $i \in I_{\mathsf{left}}$ 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 $j \leq k-1$, an edge $(i, j+1, k) \to (i, j, k)$ with weight 0.

3. If x_i, y_j is a match pair, $j < e_G(j) \leq k$, 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 $\mathrm{LCS}(F_{x_i}, G_{y_j})$.

4. If x_i, y_j is a match pair, $j < e_G(j) \leq k$, 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.

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 3.4. *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 $i \in I_{\mathsf{right}}$ (the proof for $i \in I_{\mathsf{left}}$ is similar).

The proof of the lemma is similar to the proof of Lemma 3.2. We first show that for a path p from some vertex $(0, l, l)$ to (i, j, k) of maximum weight, 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 3.2. 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 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 that ends at $(n, 1, 2m)$, which is equal to $\mathrm{LCS}(F, G)$. 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 [17], 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 $B_{F,G}$ that ends at head(e) and passes through e.

Lemma 3.5. *Let e be an edge in $E_{F,G}$ and denote* $\text{head}(e) = (i, j, k)$. *If $i \in I_{\mathsf{right}}$ then* $\text{score}(e) = \text{LCS}_R(F(i), G[j..k])$, *and otherwise* $\text{score}(e) = \text{LCS}_L(F(i), G[j..k])$.

We omit the proof of Lemma 3.5 as it is similar to the proof of Lemma 3.3.

Knowing the scores of the edges gives us $\text{LCS}(F, G)$ as $\text{LCS}(F, G) = \text{score}((n-1, 1, 2m-1) \rightarrow (n, 1, 2m))$. In fact, additional LCS values can be obtained from the scores:

Lemma 3.6. *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* $\text{LCS}(F_x, G_y) = \text{score}(e)$.

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 $i \in I_{\mathsf{right}}$. Then, $e = (i-1, e_G(k), k-1) \rightarrow (i, e_G(k), k)$ is an edge in $E_{F,G}$. By Lemma 3.5, $\text{score}(e) = \text{LCS}_R(F(i), G[e_G(k)..k])$. Both $F(i) = F_x$ and $G[e_G(k)..k] = G_y$ are trees, so from Lemma 3.1 we have that $\text{score}(e) = \text{LCS}(F_x, G_y)$. The case of $i \in I_{\mathsf{left}}$ is similar. □

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

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 $\text{score}((n-1, 1, 2m-1) \rightarrow (n, 1, 2m))$.

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. In order to build $E_{F_x,G}$ one needs to know the values of $\text{LCS}(F_{x'}, G_y)$ 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 w.r.t. $\mathcal{P}$. By Lemma 3.6, the value of $\text{LCS}(F_{x'}, G_y)$ 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 $E_{F_x,G}$.

The scores of the edges have the following monotonicity property.

Lemma 3.7. *Let e be an edge in $E_{F,G}$ and denote* $\text{head}(e) = (i, j, k)$.

1. *If $i \in I_{\mathsf{right}}$ then for every $j' \leq j$ there is an edge $e' \in E_{F,G}$ such that* $\text{head}(e') = (i, j', k)$ *and* $\text{score}(e') \geq \text{score}(e)$.

2. *If $i \in I_{\mathsf{left}}$ then for every $k' \geq k$ there is an edge $e' \in E_{F,G}$ such that* $\text{head}(e') = (i, j, k')$ *and* $\text{score}(e') \geq \text{score}(e)$.

Proof. The existence of e' with $\text{head}(e') = (i, j', k)$ follows from the construction of $E_{F,G}$. Suppose that $i \in I_{\mathsf{right}}$ (the case $i \in I_{\mathsf{left}}$ is similar). From Lemma 3.5 we know that $\text{score}(e) = \text{LCS}_R(F(i), G[j..k])$ and $\text{score}(e') = \text{LCS}_R(F(i), G[j'..k])$. Since $G[j..k]$ is a subgraph of $G[j'..k]$ it follows that $\text{LCS}_R(F(i), G[j'..k]) \geq \text{LCS}_R(F(i), G[j..k])$. □

It remains to show how to compute the scores of the edges in $E_{F,G}$. The computation of the scores is based on the following lemma.

Lemma 3.8. *For an edge* $e \in E_{F,G}$, $\text{score}(e) = \text{weight}(e) + \max(\{\text{score}(e') | e' \in E_1 \cup E_2\} \cup \{0\})$, *where*

$$E_1 = \left\{ e' \in E_{F,G} \,\middle|\, \begin{array}{l} \text{head}(e')_1 \in I_{\mathsf{right}}, \text{head}(e')_1 \leq \text{tail}(e)_1, \text{head}(e')_2 = \text{tail}(e)_2, \\ \text{head}(e')_3 \leq \text{tail}(e)_3 \end{array} \right\},$$

$$E_2 = \left\{ e' \in E_{F,G} \,\middle|\, \begin{array}{l} \text{head}(e')_1 \in I_{\mathsf{left}}, \text{head}(e')_1 \leq \text{tail}(e)_1, \text{head}(e')_2 \geq \text{tail}(e)_2, \\ \text{head}(e')_3 = \text{tail}(e)_3 \end{array} \right\}.$$

Proof. Fix an edge $e \in E_{F,G}$. Let e' be some edge from $E_1 \cup E_2$. In $B_{F,G}$ there is a path from $\text{head}(e')$ to $\text{tail}(e)$. It follows that there is a path of weight $\text{weight}(e) + \text{score}(e')$ that ends at $\text{head}(e)$ and passes through e. Therefore, $\text{score}(e) \geq \text{weight}(e) + \max(\{\text{score}(e') | e' \in E_1 \cup E_2\} \cup \{0\})$.

To prove the other direction, consider some path P of maximum weight that ends at $\text{head}(e)$ and passes through e. If P does not pass through other edges in $E_{F,G}$ then we are done as $\text{score}(e) = \text{weight}(e) \leq \text{weight}(e) + \max(\{\text{score}(e') | e' \in E_1 \cup E_2\} \cup \{0\})$. Otherwise, let $e_2 = (i_2, j_2, k_2) \to (i, j, k)$ be the last edge P passes through not including e. Since there is a path from (i, j, k) to $\text{tail}(e)$, we have that $i \leq \text{tail}(e)_1$, $j \geq \text{tail}(e)_2$, and $k \leq \text{tail}(e)_3$.

If $i \in I_{\mathsf{right}}$ then by Lemma 3.7, there is an edge $e_3 \in E_{F,G}$ with $\text{head}(e_3) = (i, \text{tail}(e)_2, k)$ and $\text{score}(e_3) \geq \text{score}(e_2)$. Since $i \leq \text{tail}(e)_1$ and $k \leq \text{tail}(e)_3$, the edge e_3 is in E_1. If $i_2 \in I_{\mathsf{left}}$ then again by Lemma 3.7 we have that there is an edge $e_3 \in E_2$ such that $\text{score}(e_3) \geq \text{score}(e_2)$. In both cases, $\text{score}(e) = \text{weight}(e) + \text{score}(e_2) \leq \text{weight}(e) + \text{score}(e_3)$, so $\text{score}(e) \leq \text{weight}(e) + \max(\{\text{score}(e') | e' \in E_1 \cup E_2\} \cup \{0\})$. □

Define the boundary of the alignment graph $B_{F,G}$ as the set of points $(0, \ell, \ell)$ for some ℓ. We call an edge e with $\text{head}(e)_1 \in I_{\mathsf{right}}$ a *right edge*. The algorithm for computing the scores of the edges in $E_{F,G}$ uses $4m$ successor data-structures $S_1^{\mathsf{left}}, \ldots, S_{2m}^{\mathsf{left}}$ and $S_1^{\mathsf{right}}, \ldots, S_{2m}^{\mathsf{right}}$. Each of these structures stores a subset of $E_{F,G}$. The key of an edge e in some structure S_i^{right} is $\text{head}(e)_3$, and the key of an edge e in some structure S_i^{left} is $\text{head}(e)_2$. The algorithm handles the edges in $E_{F,G}$ by increasing order of the first coordinate i. The important invariant is that when handling index i, for all j, k, $\text{pred}(S_j^{\mathsf{right}}, k)$ stores the last edge from $E_{F,G}$ in a maximal weight path that starts anywhere on the boundary of $B_{F,G}$ and ends at (i, j, k), among all the paths whose nonzero weight edges were already considered by the algorithm and whose last nonzero weight edge is a right edge. An analogue invariant holds for the S_k^{left}'s, namely that when handling row i, for all j, k, $\text{succ}(S_k^{\mathsf{left}}, j)$ stores the last edge from $E_{F,G}$ in a maximal weight path that starts anywhere on the boundary of $B_{F,G}$, and ends at (i, j, k) among all the paths whose nonzero weight edges were already considered by the algorithm and whose last nonzero weight edge is a left edge . Assume that in the current iteration, $i \in I_{\mathsf{right}}$. We first handle all edges e with $\text{head}(e)_1 = i$. Since $i \in I_{\mathsf{right}}$, all of these edges are right edges. When considering an edge e whose head is (i, j, k), the invariant for S_k^{left} trivially holds for any k since e is a right edge, so it does not affect S_k^{left} which only stores left edges. To maintain the invariant for S_j^{right}, if $\text{score}(e) > \text{score}(\text{pred}(S_j^{\mathsf{right}}, k))$, then e is a better way to reach (i, j, k) than $\text{pred}(S_j^{\mathsf{right}}, k)$. Hence, we insert e into S_j^{right}. In this case we also check if $\text{score}(\text{succ}(S_j^{\mathsf{right}}, k+1)) \leq \text{score}(e)$. If so, e is also better than $\text{succ}(S_j^{\mathsf{right}}, k+1)$ so we delete $\text{succ}(S_j^{\mathsf{right}}, k+1)$ from S_j^{right}. After

handling all edges whose head is i, by the invariant, $\mathrm{LCS}(F(i), G[i..j])$ is exactly the maximum between $\mathrm{score}(\mathrm{pred}(S_j^{\mathsf{right}}, k))$ (the maximal path that reaches (i,j,k) and ends with a right edge) and $\mathrm{score}(\mathrm{succ}(S_k^{\mathsf{left}}, j))$ (the maximal path that reaches (i,j,k) and ends with a left edge). Therefore, we can now update the scores of all the edges e with $\mathrm{tail}(e) = (i,j,k)$ by $\mathrm{weight}(e) + \max(\mathrm{score}(\mathrm{pred}((S_j^{\mathsf{right}}, k))), \mathrm{score}(\mathrm{succ}(S_k^{\mathsf{left}}, j)))$. The pseudocode for computing the scores is given below (recall that $\mathrm{score}(\mathrm{NULL}) = 0$).

for $i = 1, \ldots, n$ **do**
  for every $e \in E_{F,G}$ with $\mathrm{head}(e)_1 = i$ **do**
    $j \leftarrow \mathrm{head}(e)_2$, $k \leftarrow \mathrm{head}(e)_3$
    if $i \in I_{\mathsf{right}}$ and $\mathrm{score}(e) > \mathrm{score}(\mathrm{pred}(S_j^{\mathsf{right}}, k))$ **then**
      $\mathrm{insert}(S_j^{\mathsf{right}}, e)$
      while $\mathrm{succ}(S_j^{\mathsf{right}}, k+1) \neq \mathrm{NULL}$ and $\mathrm{score}(\mathrm{succ}(S_j^{\mathsf{right}}, k+1)) \leq \mathrm{score}(e)$ **do**
        $\mathrm{delete}(S_j^{\mathsf{right}}, \mathrm{succ}(S_j^{\mathsf{right}}, k+1))$
    if $i \in I_{\mathsf{left}}$ and $\mathrm{score}(e) > \mathrm{score}(\mathrm{succ}(S_k^{\mathsf{left}}, j))$ **then**
      $\mathrm{insert}(S_k^{\mathsf{left}}, e)$
      while $\mathrm{pred}(S_k^{\mathsf{left}}, j-1) \neq \mathrm{NULL}$ and $\mathrm{score}(\mathrm{pred}(S_k^{\mathsf{left}}, j-1)) \leq \mathrm{score}(e)$ **do**
        $\mathrm{delete}(S_k^{\mathsf{left}}, \mathrm{pred}(S_k^{\mathsf{left}}, j-1))$
  for every $e \in E_{F,G}$ with $\mathrm{tail}(e)_1 = i$ **do**
    $j \leftarrow \mathrm{tail}(e)_2$, $k \leftarrow \mathrm{tail}(e)_3$
    $\mathrm{score}(e) \leftarrow \mathrm{weight}(e) + \max(\mathrm{score}(\mathrm{pred}(S_j^{\mathsf{right}}, k)), \mathrm{score}(\mathrm{succ}(S_k^{\mathsf{left}}, j)))$

Just as in the previous section, using the successor data-structure of van Emde Boas [36] 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 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 3.9. $\sum_{x \in L_{\mathcal{P}}} |E_{F_x,G}| \leq 2mr(\lg r + 1)$.

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.

3.4 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 3.2, 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 $i \in I_{\mathsf{right}}$ let $E_{F,G,i,a} = \{e \in E_{F,G} | \text{head}(e)_1 = i, \text{head}(e)_3 = a\}$, and for $i \in I_{\mathsf{left}}$ 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 $i \in I_{\mathsf{right}}$. 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 $l_1 = \text{score}(e_s)$ and $l_2 = \text{score}(e_1)$. By Lemma 3.7, $\text{score}(e_1) \geq \text{score}(e_2) \geq \cdots \geq \text{score}(e_s)$. By Lemma 3.5, $\text{score}(e_j) \in \{0, \ldots, L\}$ and $\text{score}(e_j) - \text{score}(e_{j+1}) \in \{0, 1\}$ for all j. Therefore, there are indices $j_{l_1}, j_{l_1+1}, \ldots, j_{l_2}$ 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 $O(Lr \lg r \cdot \lg\lg m)$.

Following Lemma 3.8, we define for $i \leq n$ a two dimensional array A_i^{right} by

$$A_i^{\mathsf{right}}[j,k] = \max \left\{ \text{score}(e) \,\middle|\, \begin{array}{l} e \in E_{F,G}, \text{head}(e)_1 \in I_{\mathsf{right}}, \text{head}(e)_1 \leq i, \\ \text{head}(e)_2 = j, \text{head}(e')_3 \leq k \end{array} \right\}.$$

Intuitively, $A_i^{\mathsf{right}}[j,k]$ is the score of a maximal weight path that starts anywhere on the boundary of $B_{F,G}$ and ends at (i,j,k), among all the paths whose last nonzero weight edge is a right edge. The array A_i^{right} has the following properties.

1. Each row of A_i^{right} is monotonically increasing (by definition).
2. Each column of A_i^{right} is monotonically decreasing (by Lemma 3.7).
3. The difference between two adjacent cells in A_i^{right} is either 0 or 1 (by Lemma 3.5).
4. Each cell of A_i^{right} is an integer from $\{0, \ldots, L\}$ (by Lemma 3.5).

The properties above are same as the properties of the dynamic programming table for string LCS. Following the approach of [19], we define the *l-contour* of A_i (for $1 \leq l \leq L$) to be the set of all pairs (j,k) such that $A_i^{\mathsf{right}}[j,k] = l$, $A_i^{\mathsf{right}}[j+1,k] < l$ (or $j = 2m$), and $A_i^{\mathsf{right}}[j,k-1] < l$ (or $k = 1$). By properties (1) and (2) of A_i^{right} we have that for two pairs (j,k) and (j',k') in the l-contour of A_i^{right} we have either $j < j'$ and $k < k'$, or $j > j'$ and $k > k'$.

Similarly, define a two dimensional array A_i^{left} by

$$A_i^{\mathsf{left}}[j,k] = \max\left\{ \text{score}(e) \,\middle|\, \begin{array}{l} e \in E_{F,G}, \text{head}(e)_1 \in I_{\mathsf{left}}, \text{head}(e)_1 \leq i, \\ \text{head}(e)_2 \geq j, \text{head}(e')_3 = k \end{array} \right\}.$$

The array A_i^{left} also satisfies properties 1–4 above.

The algorithm for computing the compact representations of the scores processes each i from 1 to n. For each i, the algorithm computes the l-contours of A_i^{right} and A_i^{left} for all l by updating the l-contours of A_{i-1}^{right} and A_{i-1}^{left} that were computed in the previous iteration. The l-contour of A_i^{right} for the current value of i is kept using two successor data-structure $S_{l,1}^{\mathsf{right}}$ and $S_{l,2}^{\mathsf{right}}$. The key of a pair (j,k) in $S_{l,1}^{\mathsf{right}}$ is j, while the key of (j,k) in $S_{l,2}^{\mathsf{right}}$ is k. The l-contour of A_i^{left} is kept in similar structures $S_{l,1}^{\mathsf{left}}$ and $S_{l,2}^{\mathsf{left}}$. As in the previous algorithm, 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 $\text{tail}(e)_1 = i$.

Suppose that $i \in I_{\mathsf{right}}$ (handling $i \in I_{\mathsf{left}}$ is similar). Then, the contours of A_i^{left} are identical to the contours of A_{i-1}^{left}. In order to compute the l-contours of A_i^{right}, 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 previous iteration of the algorithm). Updating the l-contours according to the scores of the edges in $E_{F,G,i,k}$ is straightforward:

1: **for** $l = l_1, \ldots, l_2$ **do**
2: **if** $\text{pred}(S_{l,2}^{\mathsf{right}}, k) = \text{NULL}$ or $\text{pred}(S_{l,2}^{\mathsf{right}}, k)_1 < j_l$ **then**
3: $\text{insert}(S_{l,1}^{\mathsf{right}}, (j_l, k))$
4: $\text{insert}(S_{l,2}^{\mathsf{right}}, (j_l, k))$
5: **while** $\text{succ}(S_{l,2}^{\mathsf{right}}, k+1) \neq \text{NULL}$ and $\text{succ}(S_{l,2}^{\mathsf{right}}, k+1)_1 \leq j_l$ **do**
6: $p \leftarrow \text{succ}(S_{l,2}^{\mathsf{right}}, k+1)$
7: $\text{delete}(S_{l,1}^{\mathsf{right}}, p)$
8: $\text{delete}(S_{l,2}^{\mathsf{right}}, p)$

We now describe how to compute 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 $i' \in I_{\mathsf{right}}$ and denote $E_{F,G,i',k'} = \{e_1, \ldots, e_s\}$ where $\text{head}(e_j) = (i', j, k')$. Let $k = \text{tail}(e_1)_3$. 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. By Lemma 3.8, $\text{score}(e_j) = w + \max(A_i^{\mathsf{right}}[j,k], A_i^{\mathsf{left}}[j,k])$. Therefore the compact representation of the scores of $E_{F,G,i',k'}$ can be computed using $S_{1,2}^{\mathsf{right}}, \ldots, S_{L,2}^{\mathsf{right}}$ and $S_{1,2}^{\mathsf{left}}, \ldots, S_{L,2}^{\mathsf{left}}$:

1: $j_w \leftarrow s$
2: **for** $l = 1, \ldots, L$ **do**
3: $a \leftarrow 0$
4: **if** $\text{pred}(S_{l,2}^{\mathsf{right}}, k) \neq \text{NULL}$ **then** $a \leftarrow \text{pred}(S_{l,2}^{\mathsf{right}}, k)_1$
5: **if** $\text{pred}(S_{l,2}^{\mathsf{left}}, k) \neq \text{NULL}$ **then** $a \leftarrow \max(a, \text{pred}(S_{l,2}^{\mathsf{left}}, k)_1)$
6: **if** $a \neq 0$ **then** $j_{l+w} \leftarrow a$

If $x_{i'}$ is on the main path of F then $\text{score}(e_1) = \cdots = \text{score}(e_s) = 1 + \max(A_i^{\mathsf{right}}[s,k], A_i^{\mathsf{left}}[s,k])$, and computing the compact representation of the scores is done

similarly. The computation of the compact representation of the scores of a group $E_{F,G,i',k'}$ with $i \in I_{\mathsf{left}}$ is done similarly using the structures $S^{\mathsf{right}}_{1,1}, \ldots, S^{\mathsf{right}}_{L,1}$ and $S^{\mathsf{left}}_{1,1}, \ldots, S^{\mathsf{left}}_{L,1}$.

We obtain the following theorem.

Theorem 3.10. *The tree LCS problem can be solved in time* $O(Lr \lg r \cdot \lg \lg m)$.

Conclusions

We presented a new $O(n^3)$-time and $O(n^2)$-space algorithm for computing the tree edit distance between two rooted ordered trees. This algorithm is both symmetric in its two inputs as well as adaptively dependent on them. These features make it faster than all previous algorithms in the worst case. Furthermore, we proved that our algorithm is optimal within the broad class of decomposition strategy algorithms, by improving the previous lower bound for this class. As a consequence, any future improvements in terms of worst-case time complexity would have to rely on a new approach.

Alternatively, as we showed in Chapter 3, we can use our understanding of the structure of the problem to find more efficient algorithms for special cases, such as the largest common subtree. For this problem, we gave in section 3.3 an $O(Lr \lg r \lg \lg m)$ algorithm, which indeed violates our lower bounds for the general tree edit distance problem in the (common) case of sparse matchings. It is worth noting that in the worst (i.e., dense) case, this algorithm is asymptotically slower than our optimal algorithm for the general case by a factor of $\lg r \cdot \lg \lg m$. The source for the $\lg \lg m$ factor is the use of the successor data structure. The $\lg r$ factor originates in the choice of path decomposition, which is similar to the choice made by Klein's algorithm. While in the algorithm of Chapter 2 we managed to eliminate this factor by adapting the recursion according to the larger of the two trees in specific subproblems, in the case of the LCS algorithm a similar solution does not seem to work. The difference between the two cases is that for heavy path decomposition, the dependency is clearly on just one of the input trees. In contrast, when choosing a decomposition according to the number of match pairs, as in the algorithm of Section 3.3, the dependency is on both input trees. It is possible to change the roles of the two input trees as one of them becomes smaller than the other in this algorithm as well. However, our choice of decomposition only guarantees that the number of match pairs decreases sufficiently with each recursive call of the algorithm. We have no guarantee that the actual *size* of either tree decreases substantially. It is, in fact, possible to construct examples for which it does not. It therefore seems that changing the roles of the two trees along the execution of the algorithm does not change the worst case running time, even when considering sparse LCS instances. It would be interesting to find a way to overcome this difficulty, or to prove a lower bound which applies to the largest common subtree problem.

There are many other useful tree similarity measures which are variants of the tree edit distance and the largest common subtree problems. While some of these variants (e.g., unordered trees) are known to be NP-complete, it is plausible that our techniques may be applicable to other variants.

Bibliography

[1] A. Amir, T. Hartman, O. Kapah, B. R. Shalom, and D. Tsur. Generalized LCS. In *Proc. 14th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 50–61, 2007.

[2] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms*. Oxford University Press, Oxford, UK, 1997.

[3] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[4] R. Backofen, D. Hermelin, G. M. Landau, and O. Weimann. Normalized similarity of RNA sequences. In *Proc. 12th symposium on String Processing and Information Retrieval (SPIRE)*, pages 360–369, 2005.

[5] R. Backofen, D. Hermelin, G. M. Landau, and O. Weimann. Local alignment of RNA sequences with arbitrary scoring schemes. In *Proc. 17th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 246–257, 2006.

[6] P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337:217–239, 2005.

[7] P. Bille. *Pattern Matching in Trees and Strings*. PhD thesis, ITU University of Copenhagen, 2007.

[8] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 590–598, New York, NY, USA, 2007. ACM.

[9] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., 1999.

[10] W. Chen. New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40:135–158, 2001.

[11] F. Y. L. Chin and C. K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size. *J. of Information Processing*, 13(4):463–469, 1990.

[12] M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. on Computing*, 32:1654–1673, 2003.

[13] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 146–157, 2007.

[14] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *Proceedings of the 14th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 83–95, 2003.

[15] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Jornal of Computing*, 5:49–60, 1976.

[16] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[17] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[18] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Com. ACM*, 18(6):341–343, 1975.

[19] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *J. of the ACM*, 24(4):664–675, 1977.

[20] W. J. Hsu and M. W. Du. New algorithms for the LCS problem. *J. of Computer and System Sciences*, 29(2):133–152, 1984.

[21] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[22] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA)*, pages 91–102, 1998.

[23] P. N. Klein, S. Tirthapura, D. Sharvit, and B. B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 696–704, 2000.

[24] A. Lozano and G. Valiente. On the maximum common embedded subtree problem for ordered trees. In C. S. Iliopoulos and T. Lecroq, editors, *String Algorithmics*, pages 155–170. King's College Publications, 2004.

[25] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. of Computer and System Sciences*, 20(1):18–31, 1980.

[26] P.B Moore. Structural motifs in RNA. *Annual review of biochemistry*, 68:287–300, 1999.

[27] S. Mozes, D. Tsur, O. Weimann, and M. Ziv-Ukelson. Fast algorithms for computing tree lcs. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2008.

[28] C. Rick. Simple and fast linear space computation of longest common subsequences. *Information Processing Letters*, 75(6):275–281, 2000.

[29] S.M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

[30] D. Shasha and K. Zhang. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[31] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, 1990.

[32] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[33] K. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26(3):422–433, 1979.

[34] H. Touzet. A linear tree edit distance algorithm for similar ordered trees. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 334–345, 2005.

[35] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.

[36] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[37] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[38] M.S. Waterman. *Introduction to computational biology: maps, sequences and genomes,* chapters 13,14. Chapman and Hall, 1995.

[39] K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.