Theoretical Computer Science 409 (2008) 438–449

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Generalized LCS

Amihood Amir [a,b,*], Tzvika Hartman [a,c], Oren Kapah [a], B. Riva Shalom [a], Dekel Tsur [d]

[a] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[b] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, United States
[c] The Caesaria Rothschild Foundation, Institute for Interdisciplinary Applications of Computer Science, Haifa University, Mount Carmel, Haifa 31905, Israel
[d] Department of Computer Science, Ben-Gurion University, Be'er Sheva 84105, Israel

ARTICLE INFO

Article history:
Received 11 September 2007
Received in revised form 16 May 2008
Accepted 25 August 2008
Communicated by M. Crochemore

Keywords:
Longest common subsequence
Matrices
Trees
Non crossing matching

ABSTRACT

The Longest Common Subsequence (LCS) is a well studied problem, having a wide range of implementations. Its motivation is in comparing strings. It has long been of interest to devise a similar measure for comparing higher dimensional objects, and more complex structures. In this paper we study the Longest Common Substructure of two matrices and show that this problem is $\mathcal{NP}$-hard. We also study the Longest Common Subforest problem for multiple trees including a constrained version, as well. We show $\mathcal{NP}$-hardness for $k > 2$ unordered trees in the constrained LCS. We also give polynomial time algorithms for ordered trees and prove a lower bound for any decomposition strategy for k trees.

1. Introduction

The *Longest Common Subsequence* problem, whose first famous dynamic programming solution appeared in 1974 [24], is one of the classical problems in Computer Science. The widely known string version is defined below.

Definition 1. The *String Longest Common Subsequence (*LCS*) Problem*:
Input: Strings A and B over alphabet Σ.
Output: The maximum length of a subsequence that is common to both strings.

For example, for $A = abcddabef$ and $B = efbadeaab$, LCS(A, B) is 4, where a possible such subsequence is $adab$.

The LCS problem, has been well studied. For a survey, see [4]. The main motivation for the problem is comparison of different strings. An immediate example from computational biology is finding the commonality of two DNA molecules. Most previous work deals with the one dimensional (string) version of the problem. However, there has been increasing motivation to consider generalizations of the LCS to higher dimensions (e.g. matrices) and different data structures (e.g. trees). For example, the secondary and tertiary structure of proteins and RNA play an important role in their functionality [7], thus it is an interesting challenge to devise an inherently multi-dimensional method of comparing multidimensional objects, as the LCS compares strings.

The first task we tackle in this paper is to give a natural definition for generalizing the LCS. All generalizations until now are, essentially, linearizations [3]. Edit distance, a closely related problem, has also been generalized, and again the errors are within a single dimension [15,2]. To our knowledge, our definition is the first inherently multi-dimensional generalization in the literature. It elegantly and naturally generalizes the string definition. Unfortunately, it turns out that the LCS problem

* Corresponding author at: Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel. Tel.: +972 3 531 8770; fax: +972 3 5353325.
E-mail addresses: amir@cs.biu.ac.il (A. Amir), hartmat@cs.biu.ac.il (T. Hartman), kapaho@cs.biu.ac.il (O. Kapah), gonenr1@cs.biu.ac.il (B.R. Shalom), dekelts@cs.bgu.ac.il (D. Tsur).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.08.037

	Matrix	Unordered LCSforest	Unordered Con-LCSforest	Ordered LCSforest	Ordered Con-LCSforest
2	$\mathcal{NP}$-hard	$\mathcal{NP}$-hard	$\mathcal{P}$ [28]	$\mathcal{P}$ [8, 16, 22]	$\mathcal{P}$ [29]
$k > 2$	$\mathcal{NP}$-hard	$\mathcal{NP}$-hard	$\mathcal{NP}$-hard	$O(kn^{2k-1})$	$O(kn^{2k})$

Fig. 1. Tractability results.

between two matrices is $\mathcal{NP}$-hard. LCS applied to trees has been previously defined for two trees, via the tree edit distance. We consider the problem for multiple trees, in the case of ordered and unordered trees. We also define a constrained version of the trees LCS (Con-LCS). In addition to creating a generalized framework for the LCS problem we study the tractability of these problems. The tractability results of all the above problems are summarized in the table of Fig. 1, where LCSforest refers to Largest Common Subforest and Con-LCSforest stands for Constrained LCSforest. Previously known results appear in the table with the appropriate citation.

This paper is organized as follows: In Section 2 we pinpoint the essence of the string LCS problem, enabling us to generalize it to more complex structures and higher dimensions. The definition of two dimensional LCS and edit distance as well as the $\mathcal{NP}$-hardness results are shown in Section 3. The generalization to trees is defined in Section 4. In Section 5 we prove $\mathcal{NP}$-hardness for two unordered trees and for $k > 2$ trees of the constrained LCS. Section 6 deals with ordered trees, in which we present an algorithm for computing the LCS of k ordered trees that is based on the algorithm of Demaine et al. [8] for tree edit distance. The time complexity of the algorithm is $O(kn^{2k-1})$, where n is the number of vertices in the largest input tree, which is polynomial for a constant number of trees. We give a lower bound for any decomposition algorithm for k trees. We also present an $O(kn^k)$-time algorithm for the constrained problem that is based on the algorithm of Zhang [25] for constrained tree edit distance and a K non-crossing matching algorithm. Section 7 concludes the paper.

2. Preliminaries

The known solutions for string LCS use dynamic programming algorithms, where the value LCS($A[1, i], B[1, j]$) is computed at each step until LCS($A[1, n], B[1, n]$) is reached. The following observations are trivial, yet important to understand the main characteristic of the problems, which will assist in defining generalized LCS problems consistently with the original LCS 'spirit'.

Observation 1. *A string is a collection of objects (characters) with total precedence order between them, for every two distinct objects, one precedes another.*

Observation 2. *The* LCS *problem is ordered on a line. An* LCS *solution, matching $A[i]$ to $B[j]$ can match $A[i']$, where $i' > i$, only to a $B[j']$ where $j' > j$, and vice versa.*

Lemma 1. *The above characteristics of the* LCS *problem allows its optimal solution to consider at every step increasing prefixes of the input strings.*

Proof. The dynamic programming solution has a single possible direction of enlarging the substrings to which it computes their LCS, since all characters are ordered in precedence order. Therefore, computing LCS($A[1, i], B[1, j]$) depends merely on the LCS of prefixes of A and B shorter in one or zero symbols. ∎

Observation 3. *The* LCS *of strings A, B is the reverse of the* LCS *of A^r, B^r, where S^r is the reversed string of S.*

The above observations suggest a more combinatorial definition of the LCS problem, one that naturally generalizes to higher dimensions. Below is a combinatorial definition of the string LCS problem that supports all the above observations.

Definition 2. The *String Longest Common Subsequence* (LCS) *Problem*:
Input: Strings A and B.
Output: The maximum domain size of a one-to-one function $f : \{1, \ldots, |A|\} \to \{1, \ldots, |B|\}$ such that $A[i] = B[f(i)]$ for every i in the domain of f, and for $i, j \in \mathrm{Dom}(f)$, $i < j$ iff $f(i) < f(j)$.

The advantage of this definition is that it abstracts the LCS into an order preserving matching. A similar order preserving matching that supports the above observations is the natural generalization. However, when dealing with partially ordered structures, the dynamic programming method of computing the necessary calculations on prefixes of increasing size is meaningless, since a prefix can not be defined. A more general approach is used in our dynamic programming solutions of the tree LCS problems.

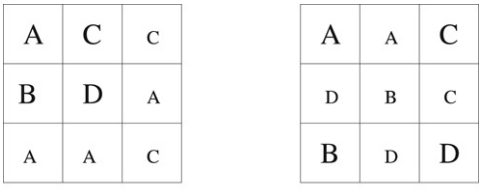

Fig. 2. An example of the 2D-LCS of two matrices.

3. Two dimensional LCS

It has been a longstanding open challenge to define an inherently multi-dimensional model for comparing multidimensional objects, as the LCS compares strings.

In the natural extension of the Longest Common Subsequence problem to a two dimensional problem, the input should be two matrices of symbols in which we seek identical symbols, preserving their order in the matrices. This will not necessarily result in a sub-matrix, but rather a structure containing the symbols common to both matrices which preserves the order relation between them in both matrices. For this reason we name the problem *Two Dimensional Longest Common Substructure* (2D-LCS). As far as we know, no inherently two dimensional version of LCS has previously been defined. Therefore, it provides a novel tool for rating the similarity between a pair of two dimensional objects.

We define the problem in a manner where Observations 2 and 3 are applicable in the two dimensional problem.

Intuitively, the LCS of two matrices A and B is the largest identical substructure that can be obtained from A and B by deleting entries, where identical means that the orientation in the plane is preserved. The formal definition is given below.

Definition 3. The *Two Dimensional Longest Common Substructure* (2D-LCS):
Input: An $n \times n$ matrix A and an $m \times m$ matrix B.
Output: The maximum domain size of a one-to-one function $f : \{1, \ldots, n\}^2 \to \{1, \ldots, m\}^2$ such that $A[i, j] = B[f(i, j)]$ for every $(i, j) \in \mathrm{Dom}(f)$, and for every $(i, j), (i', j') \in \mathrm{Dom}(f)$, the following hold:

1. $i < i'$ iff $f(i) < f(i')$.
2. $i = i'$ iff $f(i) = f(i')$.
3. $j < j'$ iff $f(j) < f(j')$.
4. $j = j'$ iff $f(j) = f(j')$.

An example for the definition above can be seen in Fig. 2. The 2D-LCS of the two matrices in the figure is 4 and can be obtained by the boldface letters.

The definition of 2D-LCS can be easily extended to higher dimensions, in which the matching symbols, required to be identical, preserve the order relation in space, meaning that for a d-dimensional structure other $2d$ constraints should be added, two for every axis.

3.1. 2D-LCS is $\mathcal{NP}$-hard

Theorem 1. *The* 2D-LCS *problem is* $\mathcal{NP}$*-hard.*

Proof. We prove the hardness of the problem by a reduction from the Clique problem.

Definition 4. The *Clique problem*:
Input: A graph G, and a constant K.
Output: Does there exist a complete subgraph of size $\geq K$, in G.

Lemma 2. *Clique* $\leq_m^p$ 2D-LCS.

Proof. Given a graph $G = (V, E)$ with vertices $v_1, \ldots, v_n$ and an integer K, we construct two matrices. Matrix A, of size $K \times K$, contains 1 in all entries except those on the main diagonal, where 2 is placed. Matrix B is the adjacency matrix of G with a slight change. B is of size $n \times n$, where $B[i, j] \in \{0, 1, 2\}$ is defined as:

$$A[i,j] = \begin{cases} 2 & i = j \\ 1 & \text{otherwise} \end{cases} \qquad B[i,j] = \begin{cases} 1 & (v_i, v_j) \in E \\ 2 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

Obviously the construction is done in polynomial time in the size of G. For an example of the construction see Fig. 3.

We now show that G contains a clique of size K iff there is a two dimensional common substructure between A and B with size K^2.

($\Rightarrow$) Suppose that G contains a clique of size K. Let $v_{c_1}, \ldots, v_{c_K}$ be the vertices participating in the clique, listed in increasing order of indices. Hence, $B[c_i, c_j] = 1$ for all $1 \leq i, j \leq K$, $i \neq j$. We get that all K^2 entries of A can be matched

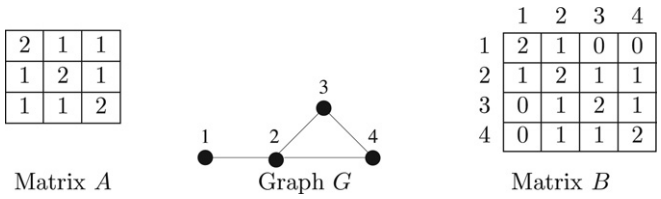

Fig. 3. Matrices A, B constructed for graph G and $K = 3$.

to entries of B in the following way: Define $f(i, j) = (c_i, c_j)$ for all i and j. We have that $A[i, j] = B[f(i, j)]$ for all i and j as $A[i, j] = B[c_i, c_j] = 1$ if $i \neq j$, and $A[i, j] = B[c_i, c_j] = 2$ if $i = j$.

We now show that every pair of symbols from A and their corresponding pair from B preserve the order relation in the plane. Let $A[i, j]$ and $A[i', j']$ be two symbols matched to $B[c_i, c_j]$ and $B[c_{i'}, c_{j'}]$, respectively. Clearly, $i < i'$ iff $c_i < c_{i'}$ and $i = i'$ iff $c_i = c_{i'}$. The same holds for j and j'.

($\Leftarrow$) Suppose there is a common substructure between A and B of size K^2. This means that all entries of A are matched to entries in B. Since the matching must preserve the order in the plane, the matched entries from B must appear in K distinct rows and K distinct columns. More precisely, the entries from the first row of A are matched to entries in one row, say c_1, in B. Moreover, $A[1, 1] = 2$ must be matched to $B[c_1, c_1] = 2$, which is the only entry in the row c_1 that contains the symbol 2. The entries $A[1, j]$ for $1 < j \leq K$ will be matched to entries of B in row c_1 at $K - 1$ columns, say $c_2, \ldots, c_K$, respectively. We have $c_1 < c_2 \cdots < c_K$ due to requirement 4 in the 2D-LCS restrictions.

Similarly, the entries in the hth row of A must be matched to some K entries in a certain row c'_h of B. The entry $A[h, h] = 2$ is bound to be matched to $B[c'_h, c'_h] = 2$. The other $K-1$ entries of row h in A should be matched to ones in row c'_h of B. However, the selection of those entries to be matched in B is not freely done. Due to requirement 4 in the 2D-LCS definition, equality in columns must also be preserved between the matched pairs. Therefore, $A[h, j]$ should be matched to a cell in column c_j of B, because $A[1, j]$ was matched to $B[c_1, c_j]$. We get that $c'_h = c_h$ and $A[h, j]$ is matched to $B[c_h, c_j]$ for all j. All in all we have that B contains K rows $c_1, \ldots, c_K$ and their corresponding columns, such that $B[c_i, c_j] = 1$ for all $i, j \in \{c_1, \ldots, c_K\}$, $i \neq j$. Hence, we have that $(v_{c_i}, v_{c_j}) \in E$ for all $i, j \in \{c_1, \ldots, c_K\}$, $i \neq j$, implying G contains a clique of size K (the vertices of the clique are $v_{c_1}, \ldots, v_{c_K}$). ∎

It can be easily observed, that the corresponding decision problem, is also in $\mathcal{NP}$, therefore the problem is $\mathcal{NP}$-complete. ∎

It is a well known fact that the LCS problem can be considered as a special case of the edit distance, transforming one string to another by operations of substitution, deletion and insertion. Suppose the substitution operation is assigned a high cost, such that it will never be profitable to use it, the edit distance problem is then equivalent to finding the LCS of the strings. It can be easily seen that this notion can be applied to generalized LCS and Edit Distance problems, in the case they are consistently defined. Therefore, the edit distance of two matrices is the minimal number of edit operations applied to the matrices, concluding in identical sub matrices, where identity implies consistency of orientation in the plane. We formally define the Two Dimensional Edit Distance Problem.

Definition 5. The Two Dimensional Edit Distance (2D-ED) Problem:
Input: Matrices A and B.
Output: The minimum number of editing operations required to transform A into B under the restriction that if in the resulting substructure the distinct, undeleted entries $A[i, j]$ and $A[i', j']$ correspond to $B[i_2, j_2]$ and $B[i'_2, j'_2]$, then the following holds:

1. $i < i'$ iff $i_2 < i'_2$.
2. $i = i'$ iff $i_2 = i'_2$.
3. $j < j'$ iff $j_2 < j'_2$.
4. $j = j'$ iff $j_2 = j'_2$.

It can be seen, that the 2D-LCS problem is a special case of the 2D-ED problem, using the same arguments as in the linear version of the problem, appearing above. Consequentially, having proved the hardness of the 2D-LCS implies the hardness of the 2D-ED as well.

4. Largest common subforest (LCSforest)

The problem of comparing trees occurs in many areas such as computer vision, compiler optimization, natural language processing and computational biology. In the latter field, for example, it is possible to represent the RNA secondary structure as a rooted ordered tree [18].

A comparison between two given trees T_1, T_2, can be defined in diverse ways. The Tree Isomorphism Problem [22] seeks the largest isomorphic subtree of T_1, T_2. It can be solved in linear time in the size of the trees. Homeomorphism between trees can be obtained by removing entire subtrees as well as repeatedly removing a degree two vertex and connecting its neighbors [16]. When T_1, T_2 are leaf labeled, the homeomorphic subtree search is converted to the maximum agreement

subtree problem [20] polynomially solved in $O(n^{4.5})$. A possible comparison between objects is, of course, the edit distance. In a recent paper [8], Demaine et al. improve Klein [14] and Shasha and Zhang [19] and give an $O(n^3)$ time algorithm for tree edit distance, for two rooted *ordered* trees, where n is the number of vertices in the larger tree.

We define the LCSforest problem in a way consistent with the String Longest Common Subsequence. All parts of the main structure are labeled and each of them can be deleted to give a common substructure. As a consequence, the Largest Common Subforest Problem should require the matching of inner vertices as well as the leaves, and should enable pruning vertices of the trees. Where pruning a vertex v implies all children of v become the children of the parent of v. Intuitively, LCSforest of T_1 and T_2 can be viewed as the largest forest that is included in both T_1 and T_2. Our general definition easily adapts to trees.

Definition 6. The *Largest Common Subforest Problem* (LCS*forest)*:
Input: Rooted vertex-labeled trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.
Output: The maximum domain size of a one-to-one function $f : V_1 \to V_2$ that satisfies the following requirements:

1. The label of a vertex v is equal to the label of $f(v)$, for every $v \in \mathrm{Dom}(f)$.
2. For every $v, w \in \mathrm{Dom}(f)$, v is an ancestor of w iff $f(v)$ is an ancestor of $f(w)$.

Observe that in the case T_1, T_2 are degenerated trees, in the form of chains, the definition is reduced to the traditional strings LCS (Definition 2). This definition applies to unordered trees. For ordered trees, where order among siblings is significant, a third requirement should be added to the LCSforest definition:

3. For $v, w \in \mathrm{Dom}(f)$, v is to the right of w iff $f(v)$ is to the right of $f(w)$.

A vertex v is to the right of vertex w if v is a right sibling of w, or that an ancestor of v is a right sibling of an ancestor of w.

It is important to note that the LCS*forest* is not necessarily a tree, but could be a forest, implied by the name of the problem. The above definition can be viewed as an extension of the Tree Inclusion problem [6,13,17,23] where one is given a pattern tree P and a text tree T, both with labels on the vertices, and the goal is to decide whether T includes P. A tree is included in another tree if it can be obtained from the larger one by deleting vertices and, in case of unordered trees, by also permuting siblings. The tree inclusion problem on unordered trees is $\mathcal{NP}$-hard [13]. For ordered trees it is polynomially solved in $O(|P| \cdot |T| / \log |T|)$ time [6].

The LCSforest definition differs from the subtree isomorphism problem, where a complete subtree is sought, not allowing pruning. Though there is a previous definition of LCS between trees more related to isomorphism [21], we define it differently, in a consistent way to the general definition.

Regarding known homeomorphism questions between trees, in [20], the maximum agreement homeomorphism subtree of a number of trees was sought. In those trees, the only labels were on leaves, there was no repetition of labels in a tree, and all labels appeared in all the trees. In [16], all vertices were labeled and labels could occur multiple times in a tree, but the problem was of finding whether a homeomorphic image of a given pattern tree P occurs in the text tree T. In addition, in the homeomorphism problem, entire subtrees or degree 2 vertices are removed, while in our LCSforest any vertex can be removed.

Due to the tight relation between the edit distance problem and the LCS problem, and since the edit distance between two trees is a well studied problem [5], in the following sections we will use some edit distance algorithms for trees to solve equivalent LCSforest questions.

4.1. Constrained-LCSforest

An interesting version of the edit distance problem posed by Zhang [25,26] is the constrained edit distance, in which a natural constraint is added to the known tree edit distance, namely that disjoint subtrees must be mapped to disjoint subtrees. The constrained edit distance is motivated from classification tree comparison. We similarly define the constrained LCSforest:

Definition 7. The *Constrained Largest Common Subforest Problem (Con-*LCS*forest)*:
Input: Rooted vertex-labeled trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.
Output: The maximum domain size of a one-to-one function $f : V_1 \to V_2$ such that the following hold:

1. The label of a vertex v is equal to the label of $f(v)$, for every $v \in \mathrm{Dom}(f)$.
2. For every $v, w \in \mathrm{Dom}(f)$, v is an ancestor of w iff $f(v)$ is an ancestor of $f(w)$.
3. For every $v_1, v_2, v_3 \in \mathrm{Dom}(f)$, $\mathrm{lca}(v_1, v_2) = \mathrm{lca}(v_1, v_3)$ iff $\mathrm{lca}(f(v_1), f(v_2)) = \mathrm{lca}(f(v_1), f(v_3))$, where $\mathrm{lca}(u, v)$ denotes the lowest common ancestor of u and v.

Fig. 4 demonstrates the constrained LCSforest between two trees. Note, that for the non-constrained version of the problem we get that the largest common subforest of T_1 and T_2 is equal to T_2.

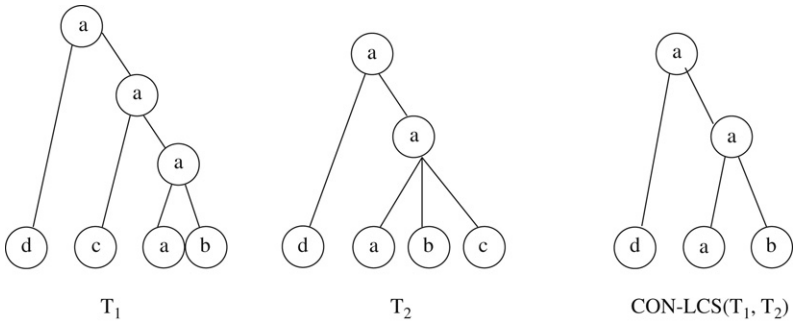

Fig. 4. An example of constrained LCSforest.

4.2. LCSforest of $k \geq 3$ Trees (LCSkforest)

Having defined the LCSforest problem for two trees, it is only natural to consider the LCSforest problem of three and more trees. The definition is a generalization of the definition of the problem between two trees. We formally define here the most general problem, though for ordered trees or for constrained LCSforest the modifications are obvious:

Definition 8. The *Largest Common k Subforest Problem (*LCS*k forest)*:
Input: Rooted vertex-labeled trees $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$.
Output: The maximum size of the intersection of the domains of $k-1$ one-to-one functions $f_i : V_1 \to V_{i+1}$, $1 \leq i \leq k-1$ such that the following requirements hold for every i:

1. The label of a vertex v is equal to the label of $f_i(v)$, for every $v \in \mathrm{Dom}(f_i)$.
2. For every $v, w \in \mathrm{Dom}(f_i)$, v is an ancestor of w iff $f_i(v)$ is an ancestor of $f_i(w)$.

To our knowledge, this problem has not been considered hitherto even for ordered trees, where order among siblings is fixed. In the following subsections we prove $\mathcal{NP}$-hardness even for the Constrained LCSforest of k unordered trees.

In the following sections we separately deal with LCSforest of ordered trees and LCSforest between unordered trees. For each we will explore the tractability of the problem for two trees, for k trees $k \geq 3$ and for a constrained LCSforest.

5. Unordered trees

Our LCSforest definition can be viewed as an extension of the Tree Inclusion problem [23,13,17] where we are given a pattern tree P and a text tree T both with labels on the vertices and the goal is to decide whether T includes P. The Tree Inclusion problem on unordered trees is $\mathcal{NP}$-hard [13]. We thus obtain that the LCSforest problem is $\mathcal{NP}$-hard.

5.1. Constrained LCSforest

The constrained LCSforest between two unordered trees can be computed in polynomial time. The problem can be solved using Zhang's algorithm for the equivalent problem for edit distance [26]. In this subsection we briefly describe the concept of the algorithm as it will prove useful later on. Zhang's algorithm uses dynamic programming and computes the LCSforest for every pair of trees $T_1[t_1]$ and $T_2[t_2]$ (for vertices $t_1 \in T_1$ and $t_2 \in T_2$), where $T[v]$ is the subtree of T induced by v and all its descendants. Consider two subtrees $T_1[t_1]$ and $T_2[t_2]$. Suppose that t_1 has n_1 children in T_1 denoted by $t_1^1, \ldots, t_1^{n_1}$, and t_2 has n_2 children denoted by $t_2^1, \ldots, t_2^{n_2}$. Zhang showed that when looking for the common subtree of two trees, their roots t_1 and t_2 can be either matched to each other or not. If we choose to match them (recall that their labels must be equal), they contribute 1 to the score of Con-LCSforest of the trees. Moreover, every subtree rooted by a child of t_1 can now be compared to every subtree rooted by a child of t_2 resulting in a Con-LCSforest value. Finally, we consider the possible matching between children of t_1 and t_2 and select the maximum matching with regard to the Con-LCSforest values.

If, however, the roots of the trees are not matched with each other, we compute all possible Con-LCSforest values of $T_1[t_1]$ with all the subtrees rooted by the n_2 children of t_2 and vice versa. From all computed values, we select the maximal and save it in the Con-LCSforest$[t_1, t_2]$ entry of the table. We therefore get the following lemma, which is very similar to [26]:

Lemma 3. *For every internal vertex $t_1 \in T_1$ and $t_2 \in T_2$,*

$$\text{Con-LCSforest}[t_1, t_2] = \max \left\{ \begin{array}{l} \max\limits_{1 \leq i \leq n_2} \left\{ \text{Con-LCSforest}[t_1, t_2^i] \right\}, \\ \max\limits_{1 \leq i \leq n_1} \left\{ \text{Con-LCSforest}[t_1^i, t_2] \right\}, \\ \begin{cases} 1 + \text{Match}(\{t_1^1, \ldots, t_1^{n_1}\}, \{t_2^1, \ldots, t_2^{n_2}\}) & \textit{if } \text{equal}(t_1, t_2) \\ 0 & \textit{otherwise} \end{cases} \end{array} \right\}.$$

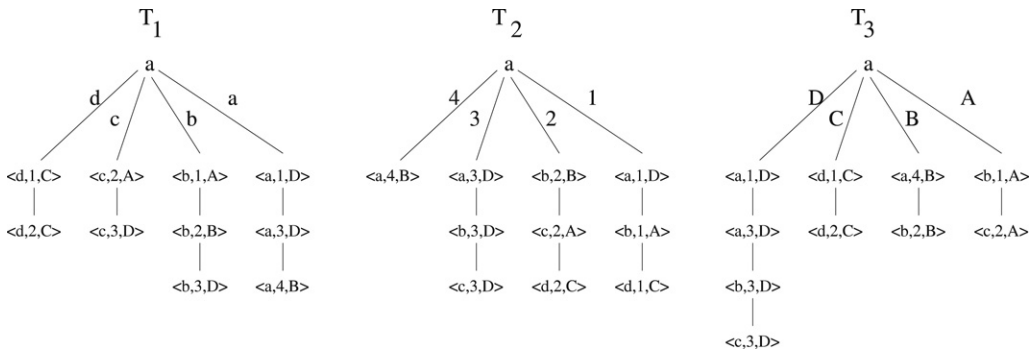

Fig. 5. The construction of T_1, T_2, T_3 according to M.

In the formula above, equal(t_1, t_2) *is a predicate whose value is true if* t_1 *and* t_2 *have equal labels, and* Match$(\{t_1^1, \ldots, t_1^{n_1}\}, \{t_2^1, \ldots, t_2^{n_2}\})$ *stands for the maximum weight of a matching between the children of* t_1 *and the children of* t_2*, where the weight of matching* t_1^i *with* t_2^j *is* Con-LCSforest$[t_1^i, t_2^j]$.

Theorem 2 (*[26]*). *The Constrained* LCS*forest problem for two rooted unordered trees can be solved in* $O(n^{2.5} \log n)$ *time, where* n *is the number of vertices in the larger tree.*

Proof. The recursion appearing at Lemma 3 can be computed using the algorithm of Gabow and Tarjan [10] for maximum weighted matching. The running time of the Gabow–Tarjan algorithm for a bipartite graph $G = (V, E)$ is $O(|E|\sqrt{|V|}\log(|V|C))$, where C is the maximum weight on an edge. When computing Con-LCSforest$[t_1, t_2]$, we solve a matching problem on a graph with $c(t_1) + c(t_2)$ vertices and $c(t_1)c(t_2)$ edges, where $c(v)$ is the number of children of v. The weight of each edge is at most n. Therefore, the Gabow–Tarjan algorithm takes $O(c(t_1)c(t_2)\sqrt{c(t_1) + c(t_2)}\log((c(t_1) + c(t_2))n))$ time.

Observation 4. *For a tree* T *with* n *vertices,* $\sum_{v \in T} c(v) = n - 1$.

We therefore get that the time complexity of the algorithm is

$$O\left(\sum_{t_1 \in T_1}\sum_{t_2 \in T_2} c(t_1)c(t_2)\sqrt{c(t_1) + c(t_2)}\log((c(t_1) + c(t_2))n)\right) = O(n^{2.5}\log n). \quad \blacksquare$$

5.2. Constrained LCSforest for k unordered trees (Con-LCSkforest)

The Constrained LCSforest bears more resemblance to trees homeomorphism than the general LCSforest and the former solution is also related to that of the latter. Amir and Keselman [1] had proved the maximum agreement subtree for three trees to be $\mathcal{NP}$-hard. We show that Con-LCSkforest is also $\mathcal{NP}$-hard. Observe, that the corresponding decision version of the Con-LCSkforest problem is also in $\mathcal{NP}$, thus, the problem is $\mathcal{NP}$-Complete.

Theorem 3. *The Con-*LCS*kforest problem is* $\mathcal{NP}$*-hard.*

Proof. We reduce the 3-Dimensional Matching problem [11] to Con-LCSkforest.

Definition 9. The 3-*Dimensional Matching problem*: (3DM)
Input: Three sets X, Y, W of q elements each, and a set $M \subseteq X \times Y \times W$.
Output: Is there a set $M' \subseteq M$ of size q where no two elements in M' agree in any coordinate. Such a set is called a *matching.*

Lemma 4. 3DM $\leq_m^p$ *Con-*LCS3*tree.*

Proof. Given a set M, we construct three trees T_1, T_2, and T_3 as follows. Each tree T_i has a root r_i labeled by a new symbol a.

Without loss of generality, we assume an order relation on the triples of M. Partition the set of elements M to q sets, according to the X value. (If there are no q sets then there is no chance for a q matching.) In each of these sets define that e is an ancestor of e' if $e < e'$. Each set thus becomes a chain. The root r_1 points to the roots of all q chains. Consequently, r_1 has q children.

We do a similar partition of the M elements by the Y value, and attach the chains to r_2. Finally, the chains created by the partition of M by the W values are attached to r_3.

For example, consider the following sets: $X = \{a, b, c, d\}$, $Y = \{1, 2, 3, 4\}$, $Z = \{A, B, C, D\}$. Suppose $M = \{[a, 4, B], [b, 3, D], [c, 2, A], [d, 1, C], [d, 2, C], [a, 1, D], [a, 3, D], [b, 1, A], [b, 2, B], [c, 3, D]\}$. The constructed trees are depicted in Fig. 5.

We claim that there is a matching M' of size q in M iff there is a Common Subforest of T_1, T_2, T_3 of size $q + 1$. The claim immediately follows from the following observation.

Observation 5. *It can not be the case that two elements of M that appear in the same chain in one of the trees* $T_i,\ i \in \{1, 2, 3\}$ *are both in the common subforest.*

The reason for Observation 5 is that if elements e, e' both appear in the CS, then the ancestor relation between them is the same in all three trees, i.e. in each of the three trees they appear in a common chain. However, by the construction, that means that e and e' have the same first, second and third coordinates, meaning they are equal. But that contradicts the fact that M is a set and not a multiset.

Note, however, that for two elements to be in the same chain means that they have a common value on one of the coordinates. Thus, no two elements from the same chain can appear in a matching.

We conclude that elements can occur in a CS iff they can occur in a matching. Add this to the fact that the root occurs in every Common Subforest and the lemma is proved. ∎

6. Ordered trees

The edit distance problem for ordered trees relates to our ordered case definition as the string edit distance relates to the string LCS. From the tree edit distance algorithm of Demaine et al. [8] we obtain an $O(n^3)$-time algorithm for the LCSforest problem. We now show how to extend the algorithm of Demaine et al. to k trees.

6.1. LCSkforest for ordered trees

We first give some definitions. Let L_F denote the leftmost tree in a forest F, and let R_F denote the rightmost tree. Let l_F and r_F be the roots of L_F and R_F, respectively. The vertices l_F and r_F are called the leftmost root and rightmost root of F. For a vertex v in a forest F, $F(v)$ is the subforest of F induced by all the proper descendants of v.

A *heavy path decomposition* of a forest F was introduced by Harel and Tarjan [12] and is built as follows. We classify each vertex of F as either *heavy* or *light*: For each vertex v we pick the child of v with the maximum number of descendants and classify it as *heavy* (ties are resolved arbitrarily). Also, pick the largest tree in F and classify its root as heavy. The remaining vertices are classified as *light*. The *main path* of the heavy path decomposition starts at the heavy root of F, and at each step moves from the current vertex to its heavy child. We next remove the vertices of the main path from F, and recursively compute a heavy path decomposition for each of the remaining trees. For a path p in a forest F, Top_p is the set of all vertices in F that are not in p but their parent is in p.

Let $F_1, \ldots, F_k$ be a non-empty forest. We can compute $\mathrm{LCSkforest}(F_1, \ldots, F_k)$ using the following recurrence.

Lemma 5. *For every nonempty forest* $F_1, \ldots, F_k$,

$$\mathrm{LCSkforest}(F_1, \ldots, F_k) = \min \left\{ \begin{array}{l} \max\limits_{1 \le l \le k} \left\{ \mathrm{LCSkforest}(F_1, \ldots, F_l - r_{F_l}, \ldots, F_k) \right\}, \\ \begin{cases} 1 + \mathrm{LCSkforest}(F_1 - R_{F_1}, \ldots, F_k - R_{F_k}) & \\ \quad + \mathrm{LCSkforest}(R_{F_1} - r_{F_1}, \ldots, R_{F_k} - r_{F_k}) & \text{if } \mathrm{equal}(r_{F_1}, \ldots, r_{F_k}) \\ 0 & \text{otherwise} \end{cases} \end{array} \right\},$$

where $\mathrm{equal}(r_{F_1}, \ldots, r_{F_k})$ *is true if the labels of* $r_{F_1}, \ldots, r_{F_k}$ *are equal.*

We can also compute $\mathrm{LCSkforest}(F_1, \ldots, F_k)$ using the following recurrence:

Lemma 6. *For every nonempty forest* $F_1, \ldots, F_k$,

$$\mathrm{LCSkforest}(F_1, \ldots, F_k) = \min \left\{ \begin{array}{l} \max\limits_{1 \le l \le k} \left\{ \mathrm{LCSkforest}(F_1, \ldots, F_l - l_{F_l}, \ldots, F_k) \right\}, \\ \begin{cases} 1 + \mathrm{LCSkforest}(F_1 - L_{F_1}, \ldots, F_k - L_{F_k}) & \\ \quad + \mathrm{LCSkforest}(L_{F_1} - l_{F_1}, \ldots, L_{F_k} - l_{F_k}) & \text{if } \mathrm{equal}(r_{F_1}, \ldots, r_{F_k}) \\ 0 & \text{otherwise} \end{cases} \end{array} \right\}.$$

A *decomposition algorithm* is an algorithm that computes the LCS of input trees $T_1, \ldots, T_k$ by recursively applying either Lemma 5 or 6 at each step. Given a decomposition algorithm A and input $T_1, \ldots, T_k$, a k-tuple $(F_1, \ldots, F_k)$ is called a *relevant subproblem* if algorithm A computes the value of $\mathrm{LCSkforest}(F_1, \ldots, F_k)$ during the computation of $\mathrm{LCSkforest}(T_1, \ldots, T_k)$. For a decomposition algorithm A and a relevant subproblem t, $A(t)$ denotes the rule used by the algorithm when computing the LCS value of t, namely, $A(t) =$ right if the algorithm uses Lemma 5, and $A(t) =$ left if the algorithm uses Lemma 6. For a relevant subproblem t, $t[i]$ denotes the ith forest of t.

The following lemma generalizes a result from [8].

Lemma 7. *Let A be a decomposition algorithm A and* $T_1, \ldots, T_k$ *be rooted trees. For every k vertices* $v_1 \in T_1, \ldots, v_k \in T_k$ *that have equal labels, the k-tuple* $(T_1(v_1), \ldots, T_k(v_k))$ *is a relevant subproblem.*

Proof. Start with $t = (T_1, \ldots, T_k)$. Repeatedly delete the leftmost or rightmost root in $t[1]$ according to the rule used by A on the current tuple t, until v_1 becomes either the leftmost or rightmost root of $t[1]$. Next, delete vertices from $t[2]$ until v_2 becomes either the leftmost or rightmost root of $t[2]$, and do the same for $t[3], \ldots, t[k]$. At each step, the current tuple t is a relevant subproblem.

Now, let F_1 be the set of all the rooted forests in t which are trees. Let F_2 (respectively, F_3) be the set of all forests $t[i]$ which are not trees and v_i is the leftmost (respectively, rightmost) root of $t[i]$. While both F_2 and F_3 are not empty, delete vertices from the forests in F_2 or from the forests in F_3: If algorithm A uses Lemma 5 on t then delete the rightmost root from some forest in F_2 and move this forest to F_1 if it becomes a tree after the deletion. Otherwise, delete the leftmost root from some forest in F_3, and move this forest to F_1 if it becomes a tree after the deletion.

The process above ends when either F_2 or F_3 becomes empty. Suppose w.l.o.g. that F_3 became empty. Now, while F_2 is not empty and $A(t) =$ right, delete the rightmost root from some forest in F_2 and move it to F_1 if necessary.

After finishing the process above, if $A(t) =$ left then the tuple $(L_{t[1]} - l_{t[1]}, \ldots, L_{t[k]} - l_{t[k]}) = (T_1(v_1), \ldots, T_k(v_k))$ is a relevant subproblem. Otherwise, we have that the tuple $(R_{t[1]} - r_{t[1]}, \ldots, R_{t[k]} - r_{t[k]}) = (T_1(v_1), \ldots, T_k(v_k))$ is a relevant subproblem (note that $t[1], \ldots, t[k]$ are trees). ∎

We now present the algorithm. We assume that the vertices of the input trees $T_1, \ldots, T_k$ are classified as heavy and light as described above. We define a procedure Compute for computing the LCS of a relevant subproblem. The algorithm computes the LCS of $T_1, \ldots, T_k$ by making a call to Compute$(T_1, \ldots, T_k)$. Procedure Compute$(F_1, \ldots, F_k)$ is as follows.

1. Let F_i be the largest forest from $F_1, \ldots, F_k$, and let p be the main path of F_i.
2. Call Compute$(F_1, \ldots, F_i(v), \ldots, F_k)$ for every $k \in \text{Top}_p$.
3. Start with $t = (F_1, \ldots, F_k)$ and recursively compute LCSkforest(t) for the current relevant subproblem t using Lemmas 5 and 6 as follows: If the rightmost root of $t[i]$ is not in p then use Lemma 6. Otherwise, use Lemma 5.

Note that in step 3 we do not use Lemma 5 or 6 if the value of LCSkforest(t) was already computed in step 2.

Theorem 4. *The algorithm above computes the* LCS *of k trees in time* $O(kn^{2k-1})$.

Proof. Let $r(F_1, \ldots, F_k)$ denote the number of relevant subproblems for the input $F_1, \ldots, F_k$. We show that $r(F_1, \ldots, T_k) \leq 4(|F_1| \cdot |F_2| \cdots |F_k|)^{2-1/k}$. We prove this by induction on $|F_1| + \cdots + |F_k|$. The base of the induction is trivial. Suppose w.l.o.g. that F_1 is the largest forest, and let p be the main path of F_1. From Lemma 7 we have that (see [8] for more details)

$$\begin{aligned}
r(F_1, \ldots, F_k) &\leq |F_1||F_2|^2 \cdots |F_k|^2 + \sum_{v \in \text{Top}_p} r(F_1(v), F_2, \ldots, F_k) \\
&\leq |F_1||F_2|^2 \cdots |F_k|^2 + \sum_{v \in \text{Top}_p} 4(|F_1(v)| \cdot |F_2| \cdots |F_k|)^{2-1/k} \\
&\leq |F_1||F_2|^2 \cdots |F_k|^2 + 4(|F_2| \cdots |F_k|)^{2-1/k} \max_{v \in \text{Top}_p} |F_1(v)|^{1-1/k} \sum_{v \in \text{Top}_p} |F_1(v)| \\
&\leq (|F_1| \cdots |F_k|)^{2-1/k} + \frac{4}{2^{1-1/k}} (|F_1| \cdots |F_k|)^{2-1/k} \\
&\leq 4(|F_1| \cdot |F_2| \cdots |F_k|)^{2-1/k}
\end{aligned}$$

(we use here the inequalities $\max_{v \in \text{Top}_p} |F_1(v)| \leq |F_1|/2$ and $\sum_{v \in \text{Top}_p} |F_1(v)| \leq |F_1|$). ∎

Demaine et al. [8] showed that every decomposition algorithm for tree edit distance requires $\Omega(n^3)$ time. We generalize their result for the LCSkforest problem. The following theorem shows that the algorithm above is asymptotically optimal among all decomposition algorithms when k is constant.

Theorem 5. *For every decomposition algorithm A and every n and* $k \geq 2$ *there are trees* $T_1, \ldots, T_k$ *of size n each such that the number of relevant subproblems for* $T_1, \ldots, T_k$ *is* $\Omega(n^{2k-1}/(12k)^{k-1})$.

Proof. For simplicity we assume that n is divisible by 3. Define a tree T with vertices $v_1, \ldots, v_{n/3}, l_1, \ldots, l_{n/3}, r_1, \ldots, r_{n/3}$. The children of the vertex v_i for $i < n/3$ are l_i, v_{i+1}, and r_i (from left to right), and the children of $v_{n/3}$ are $l_{n/3}$ and $r_{n/3}$. The trees $T_1, \ldots, T_k$ are k copies of the tree T. We now build sets $S_0, \ldots, S_{k-1}$, where each set S_i contains relevant subproblems. Every k-tuple t in a set S_r is assigned a set of *inactive indices* $I(t) \subseteq \{1, \ldots, k\}$, a *left index* $l(t) \in \{1, \ldots, k+1\} \setminus I(t)$, and a *right index* $r(t) \in \{1, \ldots, k+1\} \setminus (I(t) \cup \{l(t)\})$.

The set S_0 consists of tuples $(T_1(v_{i_1}), \ldots, T_k(v_{i_k}))$ for every index $i_1, \ldots, i_k \leq n/6$. The inactive indices set of every tuple in S_0 is empty, the left index is 1, and the right index is 2. From Lemma 7 we have that the tuples in S_0 are relevant subproblems.

We now describe how to build the set S_r from S_{r-1} for some $r > 1$. For every tuple $t \in S_{r-1}$ define a sequence of relevant subproblems $t_1 = t, t_2, t_3, \ldots, t_s$ as follows. Let $j = l(t)$ and $j' = r(t)$. Suppose we have built $t_1, \ldots, t_{i-1}$. Now, if $A(t_{i-1}) =$ left then t_i is obtained from t_{i-1} by deleting the leftmost root of $t_{i-1}[j]$, and otherwise t_i is obtained from t_i by deleting the rightmost root of $t_{i-1}[j']$. We stop this process when either $|t_s[j]| = |t[j]| - \left\lfloor \frac{n}{2(k-1)} \right\rfloor$ or $|t_s[j']| =$

$|t[j']| - \left\lfloor \frac{n}{2(k-1)} \right\rfloor$. If the former case occur, then define indices $i_1, \ldots, i_{n/2(k-1)}$, where i_l is the minimum index such that $|t_{i_l}[j]| = |t[j]| - l$. We add the tuples $t_{i_1}, \ldots, t_{i_{n/2(k-1)}}$ to S_r. For each of these tuples, the inactive set is $I(t) \cup \{j\}$, the left index is $\min(\{1, \ldots, k+1\} \setminus (I(t) \cup \{j, j'\}))$, and the right index is j'. In the latter case we define indices $i_1, \ldots, i_{n/2(k-1)}$ in an analogous way: We define indices $i_1, \ldots, i_{n/2(k-1)}$, where i_l is the minimum index such that $|t_{i_l}[j'] = |t[j']| - l$. We then add the tuples $t_{i_1}, \ldots, t_{i_{n/2(k-1)}}$ to S_r. The inactive set of each tuple is $I(t) \cup \{j'\}$, the left index is j, and the right index is $\min(\{1, \ldots, k+1\} \setminus (I(t) \cup \{j, j'\}))$. In both cases we say that the tuples $t_{i_1}, \ldots, t_{i_{n/2(k-1)}}$ were *generated* from t.

We now claim that for every r, all the tuples in S_r are distinct. To prove the claim, let t and t' be two tuples in S_r. Let $t_0, t_1, \ldots, t_r = t$ be tuples such that $t_i \in S_i$ and t_i was generated from t_{i-1} for all i. Similarly, let $t'_0, t'_1, \ldots, t'_r = t'$ be tuples such that $t'_i \in S_i$ and t'_i was generated from t'_{i-1} for all i.

We consider two cases. In the first case suppose that $t_0 = t'_0$. Let i be the maximum index such that $t_i = t'_i$. By the construction, the set $I(t_{i+1}) \setminus I(t_i)$ contains exactly one element, and let j be that element. We have that $|t[j]| = |t_{i+1}[j]| \neq |t'_{i+1}[j] = |t'[j]|$, and therefore t and t' are distinct.

In the second $t_0 \neq t'_0$. Let $i_1, \ldots, i_k$ be the indices such that $t_0 = (T_1(v_{i_1}), \ldots, T_k(v_{i_k}))$ and let $i'_1, \ldots, i'_k$ be the indices such that $t'_0 = (T_1(v_{i'_1}), \ldots, T_k(v_{i'_k}))$. Let j be an index such that $i_j \neq i'_j$. W.l.o.g. suppose that $i_j < i'_j$. We have that $|t_0[j]| \geq \frac{n}{2} + 2$. Therefore, $|t[j]| \geq |t_0[j]| - \left\lfloor \frac{n}{2(k-1)} \right\rfloor r \geq 2$. Since $t[j]$ was obtained from $t_0[j]$ by either deleting the leftmost root $|t_0[j]| - |t[j]|$ times, or by deleting the rightmost root $|t_0[j]| - |t[j]|$ times, it follows that either l_{i_j} or r_{i_j} is present in $t[j]$. However, these two vertices are not vertices in $t'[j]$, so $t[j] \neq t'[j]$. Therefore t and t' are distinct.

Clearly, $|S_0| = \left(\frac{n}{6}\right)^k$ and $|S_r| = |S_{r-1}| \cdot \left\lfloor \frac{n}{2(k-1)} \right\rfloor$. Thus, $|S_{k-1}| = \left(\frac{n}{6}\right)^k \left\lfloor \frac{n}{2(k-1)} \right\rfloor^{k-1}$. ∎

6.2. Constrained LCSforest for ordered trees (Con-LCSforest)

Applying the Constrained Edit Distance to ordered trees was done by Zhang [25] in a similar way to the unordered case. All rooted subtrees were considered as relevant subproblems. Comparing the LCSforest of two subtrees $T_1[t_1]$ with $T_2[t_2]$, t_1 and t_2 can be matched and then their children should be matched, in a way that the order among the children would be adhered to. To this aim Zhang suggested reducing the problem to string edit distance. In addition, t_1 can be matched to every child of t_2 and vice versa.

For constrained LCSforest for ordered trees, we can use the algorithm of Zhang which has time complexity $O(|T_1||T_2|)$.

Theorem 6 ([25]). *The Constrained-LCSforest for Ordered Trees can be solved in $O(n^2)$ time.*

6.3. Constrained LCSkforest for ordered trees (Con-LCSkforest)

In contrast to general unordered trees, where we proved the Con-LCSkforest to be $\mathcal{NP}$-hard, the problem for k ordered trees is polynomial for a constant number of trees.

We give a solution for Con-LCSforest for k ordered trees whose running time is $O(kn^k)$. For the case of $k = 2$ we get an $O(n^2)$ solution for the LCSforest problem, just as was attained by Zhang [25] for constrained edit distance between two ordered trees. Here again the relevant subproblems are all rooted subtrees of the k trees, therefore the dynamic programming table we fill is of size $O(n^k)$.

Solving the Constrained LCSforest problem for multiple trees can be done similarly to the case of two trees, where we have two cases: First, if all roots of the current trees match, we need to match between their children. On the other hand if we choose to discard a root we need to match one of its children with the other roots. Note, that not only we consider a matching of k dimensions, but in order to preserve the order between siblings, it must be a non-crossing matching.

Definition 10. The *Maximum k Dimensional Weighted Non-crossing Matching Problem* (kMWNM):
Input: A complete weighted k-partite graph $G = (V_1, \ldots, V_k, E)$, with a linear order on each set V_i.
Output: The maximum weight of a k-matching of G that satisfies the following property: If $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_k)$ are tuples in the matching then for every i, $x_1 < y_1$ iff $x_i < y_i$.

In the next subsection we present an algorithm for the kMWNM problem whose running time is $O(kn^k)$. For the second case of the problem, we need to consider $0 < k' < k$ out of k trees, whose roots are matched with themselves. The LCSkforest can be obtained by comparing a subtree from each of the $k - k'$ 'unselected' trees with the k' complete trees. As each of the $k - k'$ trees is represented by a single subtree, we have to consider the Cartesian product of these subtrees.

Observe that, since we keep a dynamic programming table, it suffices to calculate the Con-LCSkforest for $k - 1$ current trees and a single child representative of the kth tree, considering all possible representatives. This is true since the case of two trees T_f, T_g, not having their root in the Con-LCSkforest, is handled when considering all roots but that of T_f as well as when considering all roots but that of T_g.

To solve Con-LCSkforest, we keep a dynamic table of size n^k, where entry Con-LCSkforest $[t_1, \ldots, t_k]$ stores the constrained LCSkforest value of $T_1[t_1], \ldots, T_k[t_k]$. We get following lemma:

Lemma 8. *For $i = 1, \ldots, k$, let t_i be a vertex in T_i with children $t_i^1, \ldots, t_i^{n_i}$. Then,*

$$\text{Con-LCSkforest}[t_1, \ldots, t_k] = \max \left\{ \begin{array}{ll} \max\limits_{1\le j\le k} \left\{ \max\limits_{1\le i\le n_j} \left\{\text{Con-LCSkforest}[t_1, \ldots, t_j^i, \ldots, t_k]\right\}\right\}, & \\ \begin{cases} 1 + \text{kMWNM}(\{t_1^1, \ldots, t_1^{n_1}\}, \ldots, \{t_2^k, \ldots, t_k^{n_k}\}) & \text{if equal}(t_1, \ldots, t_k) \\ 0 & \text{otherwise} \end{cases} & \end{array} \right\}.$$

Theorem 7. *The Con-*LCSk*forest for ordered trees is solvable in $O(kn^k)$ time.*

Proof. The algorithm finding the maximum weight non-crossing matching of G runs in $O(kn_1n_2\cdots n_k)$ time, due to Theorem 8. Summing up all computations of Lemma 8 together, the Con-LCSkforest problem can be solved in time of $O(\sum_{t_1\in T_1}\cdots\sum_{t_k\in T_k} k\cdot c(t_1)c(t_2)\cdots c(t_k))$ time. By Observation 4 we get that the time complexity is $O(kn^k)$. ∎

6.4. The maximum weighted non-crossing k-matching

The problem of Weighted Non-crossing Matching for bipartite graphs was solved by Farach et al. [9] in $O(n^2)$ using a dynamic programming algorithm. The k Dimensional Maximum Weighted Non-crossing Matching Problem (kMWNM) defined in Section 6.3 extends the bipartite non-crossing matching, to match k lists instead of two. Though the multiple alignment problem is closely related to the kMWNM problem, there are some differences between the problems. In contrast to the kMWNM problem, in the alignment problem the cost of a matching between symbols is detached from the position of the actual occurrences of those symbols, moreover, the original location of a symbol in the sequence is changed due to space insertions. The gap penalty in the alignment problem and the cost of deleted/unmatched symbols, does not exist in the kMWNM problem. We therefore present a dynamic programming algorithm for the kMWNM problem.

An instance of the kMWNM problem is $G_{n_1,\ldots,n_k} = (V_1, \ldots, V_k, E)$ where $V_1 = \{v_1^1, \ldots, v_1^{n_1}\}, \ldots, V_k = \{v_k^1, \ldots, v_k^{n_k}\}$. Let n be the largest size of a vertices list. The graph is complete and every hyper-edge $(v_1^{i_1}, \ldots, v_k^{i_k})$ is associated with a weight $w(i_1, \ldots, i_k)$.

The relevant subproblems are the kMWNM of subgraphs of G induced by the vertices $V_1^{i_1}, \ldots, V_k^{i_k}$, where $V_j^l = \{v_j^1, v_j^2, \ldots, v_j^l\}$. Hence, the table we fill contains $\prod_{j=1}^k n_j = O(n^k)$ entries. The entry kMWNM$[i_1, \ldots, i_k]$ stores the maximum weight of a non-crossing matching in the subgraph of G induced by $V_1^{i_1}, \ldots, V_k^{i_k}$, where $V_j^l = \{v_j^1, v_j^2, \ldots, v_j^l\}$.

Computing kMWNM$[i_1, \ldots, i_k]$, can be done by maximizing all k entries of the form kMWNM$[i_1, \ldots, i_j - 1, \ldots, i_k]$, $1 \le j \le k$ and $w(i_1, \ldots, i_j, \ldots, i_k)+$ kMWNM$[i_1 - 1, \ldots, i_j - 1, \ldots, i_k - 1]$. This is true since when computing kMWNM$[i_1, \ldots, i_k]$, each of the i_js may appear or not in the last edge of the maximal matching that can be achieved so far. If they all appear, we get the value $w(i_1, \ldots, i_j, \ldots, i_k)+$ kMWNM$[i_1 - 1, \ldots, i_j - 1, \ldots, i_k - 1]$. For the cases that a single i_j is not included in that last edge, kMWNM$[i_1, \ldots, i_j - 1, i_k]$ contains the best matching value. In the case some i_js are not included in the last matching, for example i_f, i_h the value of the best matching should consider $w(i_1, \ldots, i_f - 1, \ldots, i_h - 1, \ldots, i_k)+$ kMWNM$[i_1 - 1, \ldots, i_f - 2, \ldots, i_h - 2, \ldots, i_k - 1]$ but, this value appears in both kMWNM$[i_1, \ldots, i_f - 1, \ldots, i_k]$ and kMWNM$[i_1, \ldots, i_h - 1, \ldots, i_k]$ already taken into account for a single vertex not included in the last edge. We have that:

Lemma 9.

$$\text{kMWNM}[i_1, \ldots, i_k] = \max \left\{ \begin{array}{l} w((v_1^{i_1}, \ldots, v_k^{i_k})) + \text{kMWNM}[i_1 - 1, \ldots, i_k - 1] \\ \max\limits_{1\le j\le k} \left\{\text{kMWNM}[i_1, \ldots, i_j - 1, \ldots, i_k]\right\} \end{array} \right\}.$$

Theorem 8. *The* kMWNM *algorithm solves the Maximum Weighted Non-crossing Matching problem in $O(kn^k)$ time.*

Proof. The algorithm calculates $O(n_1n_2\ldots\cdot n_k)$ values. Each computation involves a single additive operation and maximizing k previously computed values. Consequentially the algorithm runs in $O(kn^k)$ time. ∎

7. Conclusions and open problems

The main contribution of the paper is generalizing the concept of the traditional Longest Common Subsequence. In this paper we introduced two problems derived from the traditional Longest Common Subsequence. We have proved the problem applied to matrices or to $k \ge 3$ unordered labeled trees in constrained version, is $\mathcal{NP}$-hard, whereas the problem for k ordered trees can be solved in polynomial time for constant k. We also give a lower bound for a decomposition algorithm for LCSforest applied to k trees. Following these new definitions, LCS questions regarding other non trivial structures may be considered and their tractability explored. In addition, it will be interesting to develop approximation algorithms for the $\mathcal{NP}$-hard problems or prove their hardness of approximation.

Acknowledgments

The authors wish to thank the anonymous referees for their helpful comments. The first author was partly supported by ISF grant 35/05.

References

[1] A. Amir, D. Keselman, Maximum agreement subtree in a set of evolutionary trees — metrics and efficient algorithms, SIAM J. Comput. 26 (6) (1997) 1656–1669.
[2] A. Amir, G.M. Landau, Fast parallel and serial multidimensional approximate array matching, Theoret. Comput. Sci. 81 (1) (1991) 97–115.
[3] R. Baeza-Yates, Similarity in two-dimensional strings, in: Proc. COOCON, 1998, pp. 319–328.
[4] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proc. 7th Symposium on String Processing and Information Retrieval, SPIRE, 2000, pp. 39–48.
[5] P. Bille, A Survey on Tree Edit Distance and Related Problems, Theoret. Comput. Sci. 337 (1–3) (2005) 217–239.
[6] P. Bille, I.L. Gørtz, The tree inclusion problem: In optimal space and faster, in: Proc. 32nd International Colloquium on Automata, Languages and Programming, 2005, pp. 66–77.
[7] C. Branden, J. Tooze, Introduction to Protein Structure, Garland Publishing, New York, NY, 1999.
[8] E. Demaine, S. Mozes, B. Rossman, O. Weimann, An optimal decomposition algorithm for tree edit distance, ACM Trans. Algorithms (2008).
[9] M. Farach, T.M. Przytycka, M. Thorup, The maximum agreement subtree problem for binary trees, in: Proc. 3rd Annual European Symposium on Algorithms, ESA, 1995, pp. 381–393.
[10] H.N. Gabow, R.E. Tarjan, Faster scaling algorithms for network problems, SIAM J. Comput. 18 (5) (1989) 1013–1036.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., New York, 1979.
[12] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[13] P. Kilpelinen, H. Mannila, Ordered and unordered tree inclusion, SIAM J. Comput. 24 (2) (1995) 340–356.
[14] P.N. Klein, Computing the edit distance between unrooted ordered trees, in: Proc. 6th European Symposium on Algorithms, ESA, 1998, pp. 91–102.
[15] K. Krithivasan, R. Sitalakshmi, Efficient two-dimensional pattern matching in the presence of errors, Inform. Sci. 43 (3) (1987) 169–184.
[16] R.Y. Pinter, O. Rokhlenko, D. Tsur, M. Ziv-Ukelson, Approximate labeled subtree homeomorphism, in: Proc. 15th Symposium on Combinatorial Pattern Matching, CPM, 2004, pp. 59–73.
[17] T. Richter, A new algorithm for the ordered tree inclusion problem, in: Proc. 8th Symposium on Combinatorial Pattern Matching, CPM, 1997, pp. 150–166.
[18] B.A. Shapiro, K.Z. Zhang, Comparing multiple RNA secondary structures using tree comparisons, Comput. Appl. Biosci. 6 (4) (1990) 309–318.
[19] D. Shasha, K. Zhang, Simple fast algorithms for the editing distance between trees and related problems, SIAM J. Comput. 18 (6) (1989) 1245–1262.
[20] M. Steel, T. Warnow, Kaikoura tree theorems: The maximum agreement subtree problem, Inform. Process. Lett. 48 (1993) 77–82.
[21] Y. Takahashi, Y. Satoh, H. Suzuki, S. Sasaki, Recognition of largest common structural fragment among a variety of chemical structures, Anal. Sci. 3 (1987) 23–28.
[22] G. Valiente, Simple and efficient tree comparison, Technical Report LSI-01-1-R, Technical University of Catalonia, Department of Software, 2001.
[23] G. Valiente, Constrained tree inclusion, J. Discrete Algorithms 3 (2–4) (2005) 431–447.
[24] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1974) 168–173.
[25] K. Zhang, Algorithm for the constrained editing problem between ordered labeled trees and related problems, Pattern Recognit. 28 (1995) 463–478.
[26] K. Zhang, A constrained edit distance between unordered labeled trees, Algorithmica 15 (3) (1996) 205–222.