Tree edit distance with gaps

Hélène TOUZET
LIFL - UPRESA 8022
Bât. M3, USTL, 59 655 Villeneuve d'Ascq Cedex, France
touzet@lifl.fr

Key words : algorithms, computational complexity, tree edit distance, computational biology.

1 Introduction

The purpose of this paper is to study the definition of edit distances with convex gap weights for trees. In the special case of strings, this problem has yield to the definition of classical solutions: Galil and Giancarlo produced in [2] an algorithm in $O(n^2 \log(n))$, for example. For trees, standart edit distance algorithms – [7] or more recently [4] with a $O(n^3 \log(n))$ solution – are concerned with *linear* gap weights induced by pointwise edit operations: inserting or removing one single node (or one single edge) at each step. These algorithms may be adapted to deal with affine gap weights, with open gap penalties and extension gap penalties. However, as far as we know, there is no tentative to extend thoses results to tree edit distances with arbitrary gap weights.

The major motivation for this work comes from computational biology, with comparison of RNA molecules. RNA secondary structures without tertiary interactions, such as pseudoknots or base triples, may be canonically encoded by trees. See [6] for details. So comparing RNA structures amounts to computing edit distances between trees. It is a well-admitted fact that the insertion, or deletion, of a set of contiguous nucleotides can be assumed to result from a single mutationnal event. So it makes no sense to assign linear weight functions, as existing methods use to do. Convex gap weight functions are much more sensitive in this context.

In the paper, we first prove that there exists no polynomial algorithm for the problem with convex gap weights, unless $P = NP$. In the second part, we consider one restriction of the definition of gaps to complete subtrees, and we get a quadratic algorithm for the associated tree edit distance.

2 Preliminary Definitions : Trees, Forests, Distances

We work with ordered trees of arbitrary arity. Given a set of labels $\mathcal{L}$, we write $l(T_1, \ldots, T_n)$ for the tree composed by a root labelled by l and n subtrees $T_1, \ldots, T_n$. A forest is a finite sequence of trees.
We introduce some notations relative to trees.

$\|T\|$	:	size of the tree T,
$\texttt{Ht}(T)$	:	height of the tree T,
$T(i)$	:	i is a node of T and $T(i)$ denotes the tree composed by the node i and all the descendants of i in T,
$\texttt{Depth}(i)$	:	depth of the node i in the reference tree. The depth of the root equals 0,
$\texttt{Arity}(i)$	:	arity of the node i. The arity of a leaf equals 0,
$\texttt{Arity}(T)$	:	$\max\{\texttt{Arity}(i) \mid i \in T\}$,

In the sequel of the paper, we shall need the following technical relationship between the arities of the nodes of a tree and its size.

Lemma 1. *For every tree T, $\sum_{x \in T} \texttt{Arity}(x) = \|T\| - 1$.*

Proof. By structural induction on the construction of T. □

In a sequence, a *gap* is defined as any contiguous piece of letters that is inserted or deleted. In others words, a gap is a subsequence. We respect this philosophy and adapt it to trees.

Definition 1 (Subtree). *Let T be a tree. A subtree t of T is a set of nodes of T satisfying the two conditions below.*

(i) all the nodes of t have a lower common ancestor, called the root of t,

(ii) for each node x of t, with the exception of the root, the parent of x belongs to t.

It is clear from the definition that a subtree is a tree. It is worth noting that our definition differs from the usual definition of subtrees. Here a leaf in the subtree may be an internal node in the original tree.

Definition 2 (Gapped edit distance). *We introduce three elementary edit operations :*

- substitution: *replace a label by another label;*
- deletion: *remove a subtree t from a tree. The children of the terminal nodes of t are attached to the parent of the root of t;*
- insertion: *insert a subtree t into a tree. This is the complementary operation of deletion.*

An edit script *between two trees A and B is any sequence of edit operations transforming A into B. If one associate a cost to each edit operation, the* edit distance *between A and B, denoted $d(A, B)$, is the cost of the edit script of minimal cost from A to B. We assume that the cost function fulfils the usual requirements, so that d is a mathematical distance: symmetry, triangle inequality and $d(i, i) = 0$. In particular, inserting a subtree t has the same cost as deleting it. In the sequel of this paper, we write* $\texttt{Gap}(t)$ *for this cost.*

3 Complexity of tree edit distance with gaps

It is clear that there exists no general polynomial algorithm for distance with arbitrary gap weights, since the number of distinct subtrees in a tree may be exponential. We establish in this section that one cannot expect to improve easily this complexity even if we assume that the gap function is a convex function. For that, we consider a particular instance of the problem and we show that it is NP-hard.

For us, a convex gap weight function satisfies $\texttt{Gap}(t_1(t_2)) \leq \texttt{Gap}(t_1) + \texttt{Gap}(t_2)$, where t_1 and t_2 are subtrees, and $t_1(t_2)$ is a subtree such that t_2 is attached to one of the terminal nodes of t_1. This definition is a natural extension of the one used in [2] for gaps in strings.

3.1 The DIST problem

Labels and trees. We compare trees built up on the infinite set of labels $\mathcal{L} = \{\bullet\} \cup I\!N$.

Edit costs. For the substitution costs, let

$$\begin{array}{rcl} \texttt{Sub}(i, \bullet) = \texttt{Sub}(\bullet, i) & = & 1, \quad \forall i \in I\!N \\ \texttt{Sub}(i, j) & = & 1.5, \quad \forall i \in I\!N, \forall j \in I\!N \end{array}$$

For the cost of insertions and deletions in gaps, we need the following definition: Let T be a tree. We say that T fulfils the property $(\star)$, if *for every label i appearing in T, if $i \in I\!N$, then T contains exactly one node labelled by i.* Note that the label $\bullet$ is not concerned with the property $(\star)$. Define now the Gap function as follows.

$$\begin{array}{rcll} \texttt{Gap}(t) & = & |t| + 1, & \text{if } t \text{ satisfies the property } (\star), \\ & = & |t| + 1.5, & \text{otherwise.} \end{array}$$

It is easy to verify that Gap is a convex function. Moreover, Gap is computable in polynomial time. Our distance problem, called *DIST*, may now be phrased as

- INSTANCE: Two trees A and B, a natural number k.
- QUESTION: $d(A, B) \leq k$, where the underlying cost functions used in d are Sub and Gap?

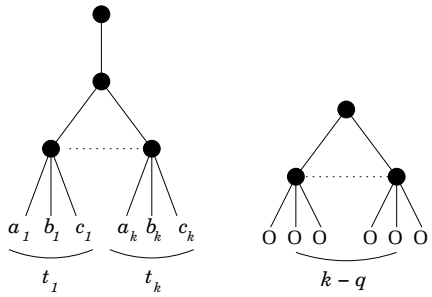

Figure 1: Trees A and B

3.2 DIST is NP-hard

The proof of NP-hardness of *DIST* is performed by reducing the *exact cover by 3-sets* problem ($X3C$) to DIST. $X3C$ is known to be NP-complete (see [3]).

- INSTANCE: Set $X = \{x_1, \ldots, x_{3q}\}$ and a collection $C = \{C_1, \ldots, C_k\}$ of 3-element subsets of X ($q \leq k$).
- QUESTION: Does C contains an exact cover for X, i.e. a subcollection C' of C such that every element of X occurs in exactly one member of C' ?

Given an instance X and C of the *X3C* problem, we define two trees A and B. First define for each j, $1 \leq j \leq k$ the tree t_j as

$$\text{if } C_j = \{x_{a_j}, x_{b_j}, x_{c_j}\}, \text{ then } t_j = \bullet(a_j, b_j, c_j).$$

The tree A is $\bullet(\bullet(t_1, \ldots, t_k))$. Its size is $4k + 2$. The other tree B is defined as $\bullet(\bullet(0, 0, 0)^{(k-q)})$. Its size is $4(k - q) + 1$. See Figure 1.

Fact 1. *$d(A, B) \leq 3k + q + 2$ if, and only if, the associated $X3C$ problem has a solution.*

Proof. If the $X3C$ problem has a solution, there exists a subset $J = \{j_1, \ldots, j_q\} \subseteq \{1, \ldots, k\}$ of cardinality q such that $\cup_{j \in J} C_j = X$. The following edit script transforms A into B, and its cost equals $3k + q + 2$.

1. Delete the subtree $\bullet(t_{j_1}, \ldots, t_{j_q})$. Since for all $j, j' \in J$, $C_j \cap C_{j'} = \emptyset$, this gap enjoys the property ($\star$), and so the cost is $|\bullet(t_{j_1}, \ldots, t_{j_q})| + 1 = 4q + 2$;
2. Substitute each remaining leaf into 0. The cost is $3(k - q)$.

Conversely, assume that $d(A, B) \leq 3k + q + 2k$. By construction, the trees A and B have exactly $k - q + 1$ identical labels in common. Moreover, any edit script for A and B should involve at least one gap, since the size of A and B are different. Looking at the `Sub` and `Cost` functions, it implies that the distance $d(A, B)$ has a lower bound $|A| - (k - q + 1) + 1 = 3k + q + 2$. To achieve this bound, it is mandatory that the script includes one single gap, and that this gap respects the property ($\star$). Moreover the script contains no insertion and each node should experiment at most one edit operation: it cannot be substituted and then deleted. This ensures that the gap contains q subtrees $t_{j_1}, \ldots, t_{j_q}$ amongst $t_1, \ldots, t_k$, such that all leaves of these substrees are distinct. The collection $C_{j_1}, \ldots, C_{j_q}$ is a cover of X. □

Corollary 1. *The* DIST *problem is NP-hard.*

4 An easy algorithm for restricted gaps

We introduce a variation of gapped edit distances, by imposing restrictions on the nature of gaps.

4.1 Gaps as complete subtrees

Definition 3 (Complete subtree). *Let T be a tree. A* complete subtree *t of T is a subtree such that for each node x of t, all descendants of x belong to t.*

We believe that the notion of distance with complete subtrees may be fruitful when applied to RNA secondary structure. In this case, the deletion, or insertion, of a complete subtree corresponds to the deletion of a substructure in the RNA molecule. More precisely, assume that each internal node of the tree encodes a stem, and each leaf encodes a stretch of unpaired bases. The distance based on complete subtrees reflects

- insertion or deletion of base pairs in a stem (substitution),
- insertion or deletion of unpaired bases in a loop (insertion or deletion),
- insertion or deletion of substructures (insertion or deletion).

4.2 Algorithm

In the special case of *linear* gap weight function, distances with complete subtrees are equivalent to edit scripts including no deletion, neither insertion of internal nodes. Chawathe proposed a quadratic algorithm in [1] for that problem. Its method is based on the construction of an edit graph and then the distance is obtained by searching the shortest path in this graph. We present here an alternative approach which deals with arbitrary gap weights.

An edit script for two trees gives raise to a mapping which is a graphical representation of the transformation.

Definition 4 (Mapping). *Let A and B be two trees, and let e be an edit script transforming A into B. A mapping $\mapsto$ is a function of $A \to B$ such that*

1. *the domain of $\mapsto$ is the set of the nodes of A that are not deleted,*
2. *the image of $\mapsto$ is the set of the nodes of B that are not inserted,*
3. *$i \mapsto j$ if and only i is substituted into j, or i is matched with j.*

The cost of the mapping $\mapsto$ is the cost of the underlying edit script, and is denoted $\texttt{Cost}(A \mapsto B)$.

We consider mappings for edit scripts allowing gap operations on complete subtrees only.

Definition 5. *A mapping is a* correct mapping *if the associated edit script is restricted to gaps on complete subtrees.*

Lemma 2. *Let A and B be two trees. A mapping $\mapsto$ from A to B is a correct mapping if, and only if*

1. *for all nodes $i \in A$ and $j \in B$, if $i \mapsto j$, then $\texttt{Depth}(i) = \texttt{Depth}(j)$,*
2. *for all nodes $i \in A$ and $j \in B$, if i is the child of i', j the child of j' and $i \mapsto j$, then $i' \mapsto j'$.*

Proof. By induction on the size of A and B. □

Lemma 3. *Let $f = a_1, \ldots, a_n$ and $f' = b_1, \ldots, b_m$ be two forests. For all $i \in [1, n], j \in [1, m]$, we write $\mapsto_i^j$ for the mapping induced by $d(a_i, b_j)$. Consider the* string *edit distance between f and f' associated to the following costs :*

- *the substitution cost of a_i into b_j is $d(a_i, b_j)$,*
- *the deletion cost of a_i is $d(a_i, \varepsilon)$,*
- *the insertion cost of b_j is $d(\varepsilon, b_j)$.*

and define the mapping $\mapsto$ from f to f' as : $\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \forall x \in a_i$, $\forall y \in b_j$, $x \mapsto y \Leftrightarrow x \mapsto_i^j y$ and a_i is substituted into b_j for the string edit distance. Then $f \mapsto f'$ is an optimal correct mapping.

Proof. The claim that $\mapsto$ is a correct mapping follows from Lemma 2. We now show that it is an optimal correct mapping. Let $\mapsto'$ be a correct mapping from f to f'. We establish by induction on the sizes of f and f' that $\texttt{Cost}(f \mapsto f') \leq \texttt{Cost}(f \mapsto' f')$. For each node i, we write $\texttt{Parent}(i)$ for the parent of the node i in the reference tree. From Lemma 2, it follows that whenever $x \mapsto' y$, then $\texttt{Parent}(x) \mapsto' \texttt{Parent}(y)$. It implies that $a_i \mapsto' b_j$ if and only if $x \mapsto' y$ for some $x \in a_i$ and $y \in b_j$. Let $i_1, \ldots, i_k, j_1, \ldots, j_k$ such that

$$f \mapsto' f' = a_{i_1} \mapsto' b_{i_1} \cup \ldots \cup a_{i_k} \mapsto' b_{i_k}.$$

We have

$$\texttt{Cost}(f \mapsto' f') = \texttt{Cost}(a_{i_1} \mapsto' b_{i_1}) + \cdots + \texttt{Cost}(a_{i_k} \mapsto' b_{i_k}).$$

The induction hypothesis ensures that

$$\texttt{Cost}(f \mapsto' f') \geq \texttt{Cost}(a_{i_1} \mapsto b_{i_1}) + \cdots + \texttt{Cost}(a_{i_k} \mapsto b_{i_k}).$$

By the definition of distance, $\texttt{Cost}(f \mapsto f') \leq \texttt{Cost}(a_{i_1} \mapsto b_{i_1}) + \cdots + \texttt{Cost}(a_{i_k} \mapsto b_{i_k})$. This leads to the desired result. □

As a consequence, we get the following recursive definition for the edit distance d.

Fact 2 (Distance for complete subtrees). *Let $l, l' \in \mathcal{L}$, and let f, f' be two forests.*

$$(1) \qquad d(l(f), \varepsilon) = \texttt{Gap}(l(f))$$

$$(2) \quad d(l(f), l'(f')) = \min \begin{cases} d(l, l') + \texttt{Distance}(f, f') \\ \texttt{Gap}(l(f)) + \texttt{Gap}(l'(f')) \end{cases}$$

`Distance` is the usual *string* edit distance, where strings are forests, that is sequences of trees.

4.3 Implementation and complexity

The algorithm uses a dynamic programming solution. The first possibility is to build up a two-dimensional table of size $(|A| + 1) \times (|B| + 1)$ to store the values $d(i, j)$. Nodes are visited in postorder traversal: the subtrees from left to right (in postorder) first, and then the root. However this approach is expensive from a space allocation point of view, since the algorithm does need to compute $d(i, j)$ if, and only if, i and j are nodes of the same depth. Instead of the two-dimensional table, we define the *product tree* structure, which is a tree labelled by the values $d(i, j)$.

Definition 6 (Product tree). *Let A and B be two trees. The* product tree *of A and B, denoted $A \times B$, is the tree labelled by real numbers defined recursively as*

1. *if $A = l$ and $B = l'$, then $A \times B = d(l, l')$,*
2. *if $A = l$ and $B = l'(b_1, \ldots, b_m)$, then $A \times B = d(A, B)(d(\varepsilon, b_1), \ldots, d(\varepsilon, b_m))$,*
3. *if $A = l(a_1, \ldots, a_n)$ and $B = l'$, then $A \times B = d(A, B)(d(a_1, \varepsilon), \ldots, d(a_n, \varepsilon))$,*
4. *if $A = l(a_1, \ldots, a_n)$ and $B = l'(b_1, \ldots, b_m)$, then*
   $A \times B = d(A, B)(a_1 \times b_1, \ldots, a_1 \times b_m, \ldots, a_n \times b_1, \ldots, a_n \times b_m)$.

It is clear that the construction of the product tree solves the distance problem. The height of the product tree equals $\max\{\texttt{Ht}(A), \texttt{Ht}(B)\}$, its arity is bounded by $\texttt{Arity}(A) \times \texttt{Arity}(B)$. As for the size, an immediate upper bound is $|A| \times |B|$. More precisely, each pair of nodes $i \in A$ and $j \in B$ with the same depth gives raise to a node of arity $\texttt{Arity}(i) \times \texttt{Arity}(j)$ if $\texttt{Arity}(i) \times \texttt{Arity}(j) \neq 0$, or to a node of arity $\texttt{Arity}(i) + \texttt{Arity}(j)$ if $\texttt{Arity}(i) \times \texttt{Arity}(j) = 0$. In practice, the size of the product tree is much smaller than $|A| \times |B|$.

For the computation of the labels of $A \times B$, Fact 2 ensures that the label of a node x of $A \times B$ depends only of the labels of its children. So labels may be determined following the postorder traversal of $A \times B$. Moreover, since the complexity for `Distance` is linear in the product of the data, the time required for x is in $\texttt{Arity}(x)$. Hence for the whole computation of the product tree, the time is in

$$\sum_{x \in A \times B} \texttt{Arity}(x).$$

According to Lemma 1, it implies that the time complexity is linear with the size of the product tree.

The construction of the whole product tree allows us to keep a track of the computation, and then to derive the underlying edit script with tracing back. However if one is interested only by the value of the distance, and not by the edit script (for clustering, for example), then the space memory can be improved. In this context, it is possible to get rid of the product tree and to calculate the labels of the tree "on the fly" with a pushdown stack S. We get the following pseudo-code.

```
S     := empty_stack;
(I,J) := (First_node_of(A), First_node_of(B));
while (I,J) in AxB
   for k=1 to arity(I)
      for l=1 to arity(J)
          Pop(S, d[k,l]);
      end for;
   end for;
   Compute d(I,J) with Distance and d; {Fact 2 - (2)}
   Push(S, d(I,J));
   (I,J) := Successor(I,J);
end while;
```

`First_node_of` denotes the first node of the tree in the postorder notation (so it always refers to a leaf). The `Successor` function associates to each couple of nodes (i, j) the next couple wrt the postorder of $A \times B$. Its specification is as follows.

```
if (J>=|B|) or (J.depth<=J+1.depth) then
        (I,J):=(I.leftmost,J+1);
elsif (I<|A|) and (I.depth>I+1.depth) then
        (I,J):=(I+1,J+1);
else
        (I,J):=(I+1,J+1.leftmost);
end if;
```

The `leftmost` attribute denotes the leftmost descendant of a node. The stack S enjoys the following invariant property: When the current node of $A \times B$ is x, then the $\texttt{Arity}(x)$ elements at the top of the stack are the children of x. We use the table d to store these values.

Lemma 4. *The size of the stack* S *is bounded by* $\min\{\ \texttt{\#leaf}(A \times B),\ \texttt{Arity}(A \times B) \times \texttt{Ht}(A \times B)\ \}$, *where* `#leaf` *denotes the number of leaves.*

Proof. By construction of the stack S, it is direct to verify that for all nodes x and y of S such that x is before y in the postorder traversal

1. $\texttt{Depth}(x) \leq \texttt{Depth}(y)$,
2. the lower common ancestor of x and y is the parent of x.

We write $card(\texttt{S})$ for the size of S. From 2., it follows that $card(\texttt{S}) \leq \texttt{\#leaf}(A \times B)$. From 1. and 2., it follows that there at most $\texttt{Arity}(A \times B)$ nodes of $A \times b$ with the same depth. So $card(\texttt{S}) \leq \texttt{Arity}(A \times B) \times \texttt{Ht}(A \times B)$. □

The construction of the stack S can be improved by enumerating nodes of $A \times B$ in *weighted* postorder, instead of the usual postorder. It means visiting subtrees by size decreasing order, and then visiting the root. Then when calculating `Distance`, one should carefully reorganize the node in the initial order. This trick leads to a lower space complexity.

Lemma 5. *Let T be a tree that is not reduced to a leaf, and let $k = \max\{2, \texttt{Arity}(T)\}$. The size of the stack* S *for the weighted postorder is bounded by $k \times \log_k(|T|)$.*

Proof. First, we need the following additional definition. A tree T is *left-balanced* if for all nodes x and y of T with the same parent, if x is before y, then $|T(y)| \leq |T(x)|$. Obviously, any ordered tree may be transformed into a left-balanced tree, by interverting nodes. Using the weighted postorder traversal instead of the usual postorder traversal amounts to build up the stack S for the associated left-balanced

tree. So it is enougth to analyze the size of a stack constructed for a left-balanced tree. In this case, weighted postorder and usual postorder are the same.

The proof of the Lemma is by induction on the size of T. If $|T| = 2$, then the result is immediate. If T is an arbitrary tree, let x be the lower common ancestor of the set of nodes belonging to $\mathtt{S}$ and let $x_1, \ldots, x_l$ be the children of x. By construction of $\mathtt{S}$, there exists a natural number l', such that $1 \leq l' \leq l \leq k$ and

$$\mathtt{S} \subseteq \{x_1, \ldots, x_{l'-1}\} \cup T(x_{l'}).$$

The set $\mathtt{S} \cap T(x_{l'})$ is governed by the same rules as a stack built on $T(x_{l'})$. So by induction hypothesis, the size of $\mathtt{S}$ is bounded by

$$card(\mathtt{S}) \leq l' - 1 + k \times \log_k(|T(x_{l'})|).$$

Since T is a left-balanced tree, we have $|T(x_{l'})| \leq |T(x)|/l'$, that implies

$$card(\mathtt{S}) \leq l' - 1 + k \times \log_k(|T(x)|/l') \leq k \times \log_k(|T(x)|) \leq k \times \log_k(|T|).$$

This concludes the proof. □

Applying this result to $A \times B$ yields a stack of size $\mathtt{Arity}(A \times B) \times \log_{\mathtt{Arity}(A \times B)}(|A \times B|)$, which is smaller than $\mathtt{Arity}(A) \times \mathtt{Arity}(B) \times (\log(|A|) + \log(|B|))$.

4.4 Miscellaneous Variations

As a conclusion, we would like to mention that the formulation of Fact 2 and the derivated algorithms are flexible. It is possible to replace the advocation of the function `Distance` by any definition of distance on strings.

Similar trees. For comparing similar trees, use a linear distance with k errors.

Circular forests. Circular forests are ordered sets of trees with no begin and no end. To deal with this kind of strutures, use a cyclic distance as defined in [5] for the last application of `Distance`.

Unordered trees. It is known that the edit distance for unordered tree is NP-complete, even for linear gap weights [8]. In this case, forests are no longer sequences of trees, but multi-sets of trees. We can adapt the previous algorithms by replacing the advocation of `Distance` by an edit distance for multi-sets. Comparing multi-sets is a polynomial problem : it is a particular instance of the *Maximum Weighted Matching* problem on bi-partite graphs. So we get a polynomial algorithm for unordered trees with restricted gaps.

Acknowledgments. We would like to thank the reviewers for their constructive remarks. We are also grateful to Sophie Tison and Jean-Marc Talbot for fruitful discussions.

References

[1] S. Chawathe, "Comparing hierarchical data in external memory" *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases* (1999), Edinburgh, Scotland, p. 90-101.

[2] Z. Galil and R. Giancarlo, "Speeding up dynamic programming with applications to molecular biology", *Theoretical Computer Science* 64 (1989), p. 107-118.

[3] M. Garey, D. Johnson, "Computers and Intractability", *Ed. Freeman*

[4] P. Klein, "Computing the edit-distance between unrooted ordered trees" *Proceeding of 6th European Symposium on Algorithms* (1998), p. 91-102.

[5] M. Maes, "On a cyclic string-to-string correction problem" *Information Processing Letters*, 35 (1990), p. 73-78.

[6] B. Shapiro and K. Zhang, "Comparing multiple RNA secondary structures using tree comparisons", *Comput. Appl. Biosciences*, Vol.4, 3 (1988), p. 387-393.

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

[8] K. Zhang, R. Statman and D. Shasha, "On the editing distance between unordered labeled trees" *Information Processing Letters*, 42 (1992), p. 133-139.