Tree Signatures and Unordered XML Pattern Matching

Pavel Zezula[1], Federica Mandreoli[2], and Riccardo Martoglia[2]

[1] Masaryk University, Brno, Czech Republic
zezula@fi.muni.cz
[2] University of Modena and Reggio Emilia, Modena, Italy
{mandreoli.federica, martoglia.riccardo}@unimo.it

Abstract. We propose an efficient approach for finding relevant XML data twigs defined by unordered query tree specifications. We use the tree signatures as the index structure and find qualifying patterns through integration of structurally consistent query path qualifications. An efficient algorithm is proposed and its implementation tested on real-life data collections.

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conform to the *labelled-tree* data model. The idea behind evaluating tree pattern queries, sometimes called the *twig queries*, is to find all existing ways of embedding the pattern in the data. Since XML data collections can be very large, efficient evaluation techniques for tree pattern matching are needed.

From the formal point of view, XML data objects can be seen as ordered labelled trees. Following this model, previous approaches considered also the query trees ordered, so the problem can be characterized as the *ordered tree pattern matching*. Though there are certainly situations where the ordered tree pattern matching perfectly reflects the information needs of users, there are many other that would prefer to consider query trees as unordered. For example, when searching for a twig of the element `person` with the subelements `first name` and `last name` (possibly with specific values), ordered matching would not consider the case where the order of the `first name` and the `last name` is reversed. However, this could exactly be the person we are searching for. The way to solve this problem is to consider the query twig as an unordered tree in which each node has a label and where only the *ancestor-descendant* relationships are important – the *preceding-following* relationships are unimportant.

In general, the process of unordered tree matching is difficult and time consuming. For example, the *edit distance* on unordered trees was found in [ZSS92] NP hard. To improve efficiency, an approximate searching for nearest neighbors, called ATreeGrep, was proposed in [SWS+02]. However, the problem of unordered twig pattern matching in XML data collections has not been studied, to the best of our knowledge.

In this paper we propose an efficient evaluation of the unordered tree matching. We use the tree signature approach [ZAD03], which has originally been proposed for the ordered tree matching. In principle, we decompose the query tree into a collection of root to leaf paths and search for their embedding in the data trees. Then we join the

structurally consistent path qualifications to find unordered query tree inclusions in the data.

The rest of the paper is organized as follows. In Section 2, we summarize the concepts of tree signatures and define their properties that are relevant towards our objectives. In Section 3 we analyze the problem of unordered tree matching, and in Section 4 we propose an efficient algorithm to compute the query answer set. Performance evaluations are presented in Section 5. Final conclusions are in Section 6.

2 Tree Signatures

The idea of *tree signatures* proposed in [ZAD03] is to maintain a small but sufficient representation of the tree structures able to decide the ordered tree inclusion problem for the XML data processing. As a coding schema, the *preorder* and *postorder* ranks [Die82] are used. In this way, tree structures are linearized, and extended string processing algorithms are applied to identify the tree inclusion.

An ordered tree T is a rooted tree in which the children of each node are ordered. If a node $v \in T$ has k children then the children are uniquely identified, left to right, as $i_1, i_2, \ldots, i_k$. A labelled tree T associates a label (name) $t_v \in \Sigma$ (the domain of tree node labels) with each node $v \in T$. If the path from the root to v has length n, we say that the node v is on the level n, i.e. $level(v) = n$. Finally, $size(v)$ denotes the number of nodes rooted at v – the size of any leaf node is zero. In this section, we consider ordered labelled trees.

The preorder and postorder sequences are ordered lists of all nodes of a given tree T. In a preorder sequence, a tree node v is traversed and assigned its (increasing) preorder rank, $pre(v)$, before its children are recursively traversed from left to right. In the postorder sequence, a tree node v is traversed and assigned its (increasing) postorder rank, $post(v)$, after its children are recursively traversed from left to right. For illustration, see the preorder and postorder sequences of our sample tree in Fig. 1 – the node's position in the sequence is its preorder/postorder rank, respectively.

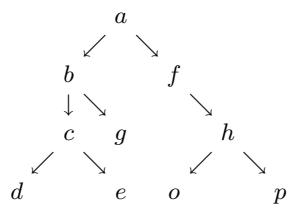


Fig. 1. Preorder and postorder sequences of a tree

Given a node $v \in T$ with $pre(v)$ and $post(v)$ ranks, the following properties are important towards our objectives:

- all nodes x with $pre(x) < pre(v)$ are the *ancestors* or *preceding* nodes of v;
- all nodes x with $pre(x) > pre(v)$ are the *descendants* or *following* nodes of v;
- all nodes x with $post(x) < post(v)$ are the *descendants* or *preceding* nodes of v;
- all nodes x with $post(x) > post(v)$ are the *ancestors* or *following* nodes of v;
- for any $v \in T$, we have $pre(v) - post(v) + size(v) = level(v)$;
- if $pre(v) = 1$, v is the root, if $pre(v) = n$, v is a leaf. For all the other neighboring nodes v_i and v_{i+1} in the preorder sequence, if $post(v_{i+1}) > post(v_i)$, v_i is a leaf.

As proposed in [Gr02], such properties can be summarized in a two dimensional diagram. See Fig. 2 for illustration, where the *ancestor* (A), *descendant* (D), *preceding*

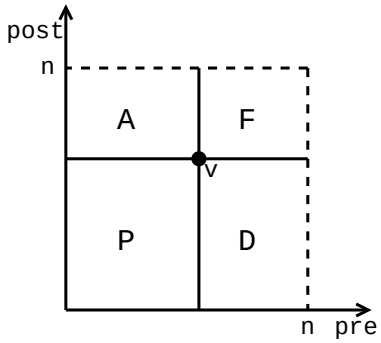


Fig. 2. Properties of the preorder and postorder ranks.

(P), and *following* (F) nodes of v are strictly located in the proper regions. Notice that in the preorder sequence all descendant nodes (if they exist) form a continuous sequence, which is constrained on the left by the reference node v and on the right by the first following node of v (or by the end of the sequence). The parent node of the reference is the ancestor with the highest preorder rank, i.e. the closest ancestor of the reference.

2.1 The Signature

The tree signature is a list of entries for all nodes in acceding preorder. In addition to the node name, each entry also contains the node's position in the postorder sequence.

Definition 1. *Let T be an ordered labelled tree. The signature of T is a sequence, $sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots t_n, post(t_n)\rangle$, of $n = |T|$ entries, where t_i is a name of the node with $pre(t_i) = i$. The $post(t_i)$ value is the postorder value of the node named t_i and the preorder value i.*

Observe that the index in the signature sequence is the node's preorder, so the value serves actually two purposes. In the following, we use the term preorder if we mean the

rank of the node. When we consider the position of the node's entry in the signature sequence, we use the term index. For example, $\langle a, 10; b, 5; c, 3; d, 1; e, 2; g, 4; f, 9; h, 8; o, 6; p, 7\rangle$ is the signature of the tree from Fig. 1. The first signature element a is the tree root. Leaf nodes in signatures are all nodes with postorder smaller than the postorder of the following node in the signature sequence, that is nodes d, e, g, o – the last node, in our example it is the node p, is always a leaf. We can also easily determine the level of leaf nodes, because the $size(t_i) = 0$ for all leaves t_i, thus $level(t_i) = i - post(t_i)$.

Extended Signatures By extending entries of tree signatures with two preorder numbers representing pointers to the *first following*, ff, and the *first ancestor*, fa, nodes, the *extended signatures* are defined in [ZAD03]. The generic entry of the i-th extended signature is $\langle t_i, post(t_i), ff_i, fa_i\rangle$. Such version of the tree signatures makes possible to compute levels for any node as $level(t_i) = ff_i - post(t_i) - 1$, because the cardinality of the descendant node set can be computed as: $size(t_i) = ff_i - i - 1$. For the tree from Fig. 1, the extended signature is: $sig(T) = \langle a, 10, 11, 0; b, 5, 7, 1; c, 3, 6, 2; d, 1, 5, 3;$ $e, 2, 6, 3; g, 4, 7, 2; f, 9, 11, 1; h, 8, 11, 7; o, 6, 10, 8; p, 7, 11, 8\rangle$.

Sub-Signatures A sub-signature $sub\_sig_S(T)$ is a specialized (restricted) view of T through signatures, which retains the original hierarchical relationships of elements in T. Specifically, $sub\_sig_S(T) = \langle t_{s_1}, post(t_{s_1}); t_{s_2}, post(t_{s_2}); \ldots t_{s_n}, post(t_{s_n})\rangle$ is a sub-sequence of $sig(T)$, defined by the ordered set $S = \{s_1, s_2, \ldots s_k\}$ of indexes (preorder values) in $sig(T)$, such that $1 \leq s_1 < s_2 < \ldots < s_k \leq n$. Naturally, the set operations of the union and the intersection can be applied on sub-signatures provided the sub-signatures are derived from the same signatures and the results are kept sorted. For example consider two sub-signatures of the signature representing the tree from Fig. 1, defined by ordered sets $S_1 = \{2, 3, 4\}$ and $S_2 = \{2, 3, 5, 6\}$. The union of S_1 and S_2 is the set $\{2, 3, 4, 5, 6\}$, that is the sub-signature representing the subtree rooted at the node b of our sample tree.

2.2 Ordered tree inclusion evaluation

Let D and Q be ordered labelled trees. The tree Q is included in D, if D contains all elements (nodes) of Q and when the *sibling* and *ancestor* relationships of the nodes in D are the same as in Q. Using the concept of signatures, we can formally define the ordered tree inclusion problem as follows. Suppose the data tree D and the query tree Q specified by signatures

$$sig(D) = \langle d_1, post(d_1); d_2, post(d_2); \ldots d_m, post(d_m)\rangle,$$

$$sig(Q) = \langle q_1, post(q_1); q_2, post(q_2); \ldots q_n, post(q_n)\rangle.$$

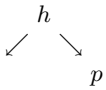
Let $sub\_sig_S(D)$ be the *sub-signature* (i.e. a subsequence) of $sig(D)$ induced by a name sequence-inclusion of $sig(Q)$ in $sig(D)$ – a specific query signature can determine zero or more data sub-signatures. Regarding the node names, any $sub\_sig_S(D) \equiv sig(Q)$, because $q_i = d_{s_i}$ for all i, but the corresponding entries can have different postorder values. The following lemma defines the necessary constrains for qualification.

Lemma 1. *The query tree Q is included in the data tree D if the following two conditions are satisfied: (1) on the level of node names, $sig(Q)$ is sequence-included in $sig(D)$ determining $sub\_sig_S(D)$ through the ordered set of indexes $S = \{s_1, s_2, \ldots s_n\}$, (2) for all pairs of entries i and j in $sig(Q)$ and $sub\_sig_S(D)$, $i, j = 1, 2, \ldots |Q|-1$ and $i+j \leq |Q|$, $post(q_{i+j}) > post(q_i)$ implies $post(d_{s_{i+j}}) > post(d_{s_i})$ and $post(q_{i+j}) < post(q_i)$ implies $post(d_{s_{i+j}}) < post(d_{s_i})$.*

Proof. Because the index i increases according to the preorder sequence, node $i + j$ must be either the descendent or the following node of i. If $post(q_{i+j}) < post(q_i)$, the node $i + j$ in the query is a descendent of the node i, thus also $post(d_{s_{i+j}}) < post(d_{s_i})$ is required. By analogy, if $post(q_{i+j}) > post(q_i)$, the node $i + j$ in the query is a following node of i, thus also $post(d_{s_{i+j}}) > post(d_{s_{i)}}$ must hold.

Observe that Lemma 1 defines a *weak inclusion* of the query tree in the data tree, meaning that the parent-child relationships of the query are implicitly reflected in the data tree as only the ancestor-descendant. However, due to the properties of preorder and postorder ranks, such constraints can easily be strengthened, if required.

For example, consider the data tree D in Fig. 1 and the query tree Q in Fig. 3. Such



$$sig(Q) = \langle h, 3; o, 1; p, 2\rangle$$

Fig. 3. Sample query tree Q

query qualifies in D: $sig(Q) = \langle h, 3; o, 1; p, 2\rangle$ determines $sub\_sig_S(T) = \langle h, 8; o, 6; p, 7\rangle$ through the ordered set $S = \{8, 9, 10\}$ because (1) $q_1 = d_8$, $q_2 = d_9$, and $q_3 = d_{10}$, (2) the postorder of node h is higher than the postorder of nodes o and p, and the postorder of node o is smaller than the postorder of node p (both in the $sig(Q)$ and $sub\_sig_S(T)$). If we change in our query tree Q the lable h for f, we get $sig(Q) = \langle f, 3; o, 1; p, 2\rangle$. Such modified query tree is also included in D, because Lemma 1 does not insist on the strict parent-child relationships, and implicitly considers all such relationships as ancestor-descendant. However, the query tree with the root g, resulting in $sig(Q) = \langle g, 3; o, 1; p, 2\rangle$, does not qualify, even though it is also sequence-included (on the level of names) as the sub-signature $sub\_sig_S(D) = \langle g, 4; o, 6; p, 7\rangle | S = \{6, 9, 10\}$. The reason is that the query requires the postorder to go down from g to o (from 3 to 1) , while in the sub-signature it actually goes up (from 4 to 6). That means that o is not a descendant node of g, as required by the query, which can be verified in Fig. 1.

Multiple nodes with common names may result in multiple tree inclusions. As demonstrated in [ZAD03], the tree signatures can easily deal with such situations just by simply distinguishing between node names and their unique occurrences.

3 Unordered Tree Pattern Matching

In this section, we propose an approach to the unordered tree pattern matching by using the tree signatures. The following definition specifies the notion of the unordered tree inclusion.

Definition 2 (Unordered Tree Inclusion). *Given a query twig pattern Q and an XML tree D, an unordered tree inclusion of Q in D is identified by a total mapping from nodes in Q to some nodes in D, such that only the ancestor-descendent structural relationships between nodes in Q are satisfied by the corresponding nodes in D.*

The unordered tree inclusion evaluation essentially searches for a node mapping keeping the ancestor-descendent relationships of the query nodes in the target data nodes. Potentially, tree signatures are suitable for such a task, because they rely on a numbering scheme allowing a unique identification of nodes in the tree, as well as the ancestor-descendent relationships between them. However, signatures assume (data and query) trees always ordered, so the serialization of trees based on the preorder and postorder ranks does not only capture the ancestor-descendent but also the sibling relationships. For this reason, the unordered tree inclusion can not be evaluated by directly checking inclusion properties of the query in the data tree signature. More formally, using the concept of signature, the unordered query tree Q is included in the data tree D if at least one qualifying index set exists.

Lemma 2. *Suppose the data tree D and the query tree Q to be specified by signatures*

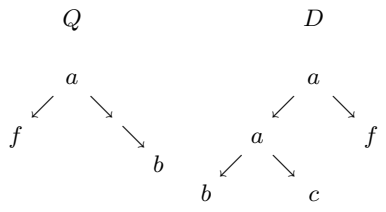
$$sig(D) = \langle d_1, post(d_1); d_2, post(d_2); \ldots d_m, post(d_m)\rangle$$

$$sig(Q) = \langle q_1, post(q_1); q_2, post(q_2); \ldots q_n, post(q_n)\rangle$$

The unordered query tree Q is included in the data tree D if the following two conditions are satisfied: (1) on the level of node names, an ordered set of indexes $S = \{s_1, s_2, \ldots s_n\}$ exists, $1 \leq s_i \leq n$ for $i = 1, \ldots, n$, such that $d_{s_i} = q_i$, (2) for all pairs of entries i and j, $i, j = 1, 2, \ldots |Q| - 1$ and $i + j \leq |Q|$, if $post(q_{i+j}) < post(q_i)$ then $post(d_{s_{i+j}}) < post(d_{s_i})$.

Observe that the index set S is ordered, but unlike the ordered inclusion given by Lemma 1, values of indexes s_i are not necessarily increasing with growing i. In other words, an unordered tree inclusion does not necessarily imply the node-name sequence inclusion of the query signature in the data signature. Should the signature $sig(Q)$ of the query not be sequence-included on the level of node names in the signature $sig(D)$ of the data, S would not determine the qualifying sub-signature $sub\_sig_S(D)$. Anyway, any S satisfying the properties specified in Lemma 2 can always undergo a sorting process in order to determine the corresponding sub-signature of $sig(D)$ qualifying the unordered tree inclusion of Q in D.

Example 1. Consider the query Q and the data tree D of Fig. 4 where the double arrow represents an ancestor-descendant edge. The only sub-signature qualifying the unordered tree inclusion of Q in D is defined by the index set $\{1, 5, 3\}$ and the corresponding sub-signature is $sub\_sig_{\{1,3,5\}}(D) = \langle a, 5; b, 1; f, 4\rangle$.



$sig(Q) = \langle a, 3, 4, 0; f, 1, 3, 1; b, 2, 4, 1\rangle \quad sig(D) = \langle a, 5; a, 3; b, 1; c, 2; f, 4\rangle$

Fig. 4. Sample of query evaluation

The solution we propose basically employs tree signature to represent data trees. Then we transform the query tree into multiple (partial) queries and evaluate the ordered inclusion of such multiple queries to obtain the answer of the unordered tree inclusion. Suppose the data tree D specified by signature $sig(D)$ and a query tree Q, two alternatives for unordered tree inclusion exist:

- Consider all and only such permutations Q_i of the query Q satisfying its ancestor-descendent relationships and compute the answers to the ordered inclusion of the corresponding signatures $sig(Q_i)$ in the data signature $sig(D)$. The union of the partial answers is the result to the unordered inclusion of Q in D. Indeed, the signature of any Q_i maintains all the ancestor-descendent relationships of Q and a specific form of the sibling relationships – the ordered tree inclusion evaluation checks both types of relationships.
- Decompose the query tree Q into a set of root-to-leaf paths P_i and evaluate the inclusion of the corresponding signatures $sig(P_i)$ in the data signature $sig(D)$. Any path P_i represents all (and only) the ancestor-descendent relationships between the involved nodes. Thus, an ordered inclusion of $sig(P_i)$ in $sig(D)$ states that a mapping, keeping the ancestor-descendent relationships, exists from the nodes in P_i to some nodes in D. If there are structurally consistent answers to the ordered inclusion of all the paths P_i in D, the unordered tree inclusion of Q in D is found.

In the following we concentrate on the second approach, which in principle consists of the following three steps:

1. decomposition of the query Q into a set of paths P_i;
2. evaluation of the inclusion of the corresponding signatures $sig(P_i)$ in the data signature $sig(D)$;
3. identification of the set of answers to the unordered inclusion of Q in D.

3.1 Query decomposition

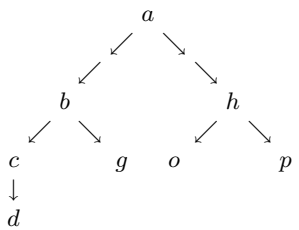
The query decomposition process transforms a query twig into a set of root-to-leaf paths so that the ordered tree inclusion can be safely applied. For efficiency reasons, we sort the paths on the basis of their selectivity, so that in the next phase, the more selective paths are evaluated before the less selective ones. Figure 5 shows an algorithm based

Input: query Q represented by the extended signature
$sig(Q) = \langle t_1, post(t_1), ff_1, fa_1; \ldots; t_n, post(t_n), ff_n, fa_n \rangle$
Output: the ordered set $rew(Q)$ of paths of $sig(Q)$ defined by the index sets P_j
Algorithm:

```
(1)  for j from 1 to n do
(2)     if (ff_j = (j + 1))
(3)       i = j;
(4)       P_j = {i};
(5)       while(fa_i <> 0)
(6)           P_j = P_j ∪ {fa_i};
(7)           i = fa_i;
(8)       push(rew(Q), P_j);
(9)  sort(rew(Q));
```

Fig. 5. The query decomposition algorithm

on the above assumption for the detection of all the root-to-leaf paths of a query Q represented by the extended signature $sig(Q) = \langle t_1, post(t_1), ff_1, fa_1; \ldots; t_n, post(t_n), ff_n, fa_n \rangle$. It first identifies all the root-to-leaf paths and then sorts them on the basis of a predefined policy. The outcome is an ordered set $rew(Q)$ of the sub-signatures $sub\_sig_{P_j}(Q)$ defined by the index sets P_j, for each leaf j. It sequentially scans the signature $sig(Q)$ (line 1) and, whenever it finds a leaf (line 2) j, it follows the path connecting j with the tree root (lines 3-7). The nodes found in the path constitute the set of indexes P_j. Finally, in line 9, it sorts the sets of indexes on the basis of their selectivity. As statistics about the selectivity of paths are lacking, we suppose that as the longer the path is, the more selective it is. Recall that the length of any path corresponds to the level of the leaf node of the path. In this case, as shown in Sec. 2, the level of any leaf j can be easily computed from the extended signature $sig(Q)$ and paths can be sorted according to the leaf node level in descending order. The outcome is an ordered set of root-to-leaf paths covering the overall query Q and arranged according to the selectivity.



$sig(Q) = \langle a, 8, 9, 0; b, 4, 6, 1; c, 2, 5, 2; d, 1, 5, 3; g, 3, 6, 2; h, 7, 9, 1; o, 5, 8, 6; p, 6, 9, 6 \rangle$

Fig. 6. Sample query tree

Example 2. Let us consider the query tree of figure 6. The algorithm sequentially scans the signature $sig(Q)$ up to $j = 4$ since $ff_4 = 5$ (no descendant nodes), so the 4-th node is a leaf defining $P_4 = \{1, 2, 3, 4\}$. Then the algorithm iterates by starting from $j = 5$. Assuming that the paths are ordered on the basis of their lengths, the final outcome is $rew(Q) = \{P_4, P_5, P_7, P_8\}$ such that $P_5 = \{1, 2, 5\}$, $P_7 = \{1, 6, 7\}$, and $P_8 = \{1, 6, 8\}$. Notice that, as $rew(Q)$ is an ordered set, paths will be evaluated in the same order as they appear in $rew(Q)$.

3.2 Path inclusion evaluation

After the query has been decomposed into a sequence of paths, it has to be evaluated against the data signature. The answers to the unordered tree inclusion of Q in D are computed by joining the solutions to the individual paths in $rew(Q)$. As far as the evaluation of each individual path $P \in rew(Q)$ with respect to a data tree D is concerned, it can be performed in an ordered fashion. Indeed, in case of paths, the ordered evaluation coincides with the unordered one. As each $P \in rew(Q)$ identifies a path of Q, we know that each node is the descendant of the nodes appearing before the node in P. Following the numbering scheme of the sub-signature $sub\_sig_P(Q) = \langle t_{p_1}, post(t_{p_1}), ff_{p_1}, fa_{p_1}; \ldots, t_{p_h}, post(t_{p_h}), ff_{p_h}, fa_{p_h} \rangle$ defined by $P = \{p_1, \ldots, p_h\}$, the post-order values of subsequent entries i and $i + j$ ($i, j = 1, 2, \ldots h - 1$ and $i + j \leq h$) always satisfy the inequality $post(q_{p_i}) < post(q_{p_{i+j}})$. The lemma below easily follows from the above observation and from the fact that inequalities are transitive.

Lemma 3. *A path $P \in rew(Q)$ is included in the data tree D, in the sense of Definition 2, if the following two conditions are satisfied: (1) on the level of node names, $sub\_sig_P(Q)$ is sequence-included in $sig(D)$ determining $sub\_sig_S(D)$ through the ordered set of indexes $S = \{s_1, \ldots, s_h\}$, (2) for each $i \in [1, n - 1]$: $post(d_{s_i}) < post(d_{s_{i+1}})$.*

For each path query $P \in rew(Q)$, we are thus able to compute the answer set $ans_P(D) = \{S \mid sub\_sig_S(D)$ qualifies the inclusion of P in $D\}$. Such evaluation is simpler than the ordered tree inclusion evaluation of Lemma 1, because we do not have to check the relationships between post-order values in the query signature, which are strictly of the ancestor-descendant type. Moreover, since the relationship, expressed by the inequality $<$, is transitive, we can simply check inequalities between post-order values of adjacent entries, and limit the verification to $h - 1$ checks, provided the length of the path P is h. Notice that, as in the ordered tree inclusion evaluation case, all hierarchical relationships in the query tree are implicitly considered as the ancestor-descendent, rather than the parent-child relationships. In case the parent-child relationships are strictly required, an additional simple control through the first ancestor pointer is necessary.

3.3 Identification of the answer set

The answer set $ans_Q(D)$ of the unordered inclusion of Q in D can be determined by joining compatible answer sets $ans_P(D)$, for all $P \in rew(Q)$. The main problem is

to establish how to join the answers for the paths in $rew(Q)$ to get the answers of the unordered inclusion of Q in D. Not all pairs of answers of two distinct sets are necessarily "joinable". The fact is that any pair of paths P_i and P_j share a common sub-path (at least the root) and differs in the other nodes (at least the leaves). Such *commonalities* and *differences* must meet a correspondence in any pair of index sets $S_i \in ans_{P_i}(D)$ and $S_j \in ans_{P_j}(D)$, respectively, in order that they are joinable. In this case, we state that $S_i \in ans_{P_i}(D)$ and $S_j \in ans_{P_j}(D)$ are structurally consistent.

Example 3 (cont. Ex. 1). Consider again the query Q and the data tree D of Fig. 4. Notice that the index set $\{1,5,3\}$ satisfies both the conditions of Lemma 2 whereas the index set $\{2,5,3\}$ only matches at the level of node names but it is not a qualifying one. The rewriting of Q gives rise to the following paths $rew(Q) = \{P_2, P_3\}$, where $P_2 = \{1,2\}$ and $P_3 = \{1,3\}$, and the outcome of their evaluation is $ans_{P_2} = \{\{1,5\}\}$ and $ans_{P_3} = \{\{1,3\},\{2,3\}\}$. The common sub-path between P_2 and P_3 is $P_2 \cap P_3 = \{1\}$. The index 1 occurs in the first position both in P_2 and P_3. From the cartesian product of $ans_{P_2}(D)$ and $ans_{P_3}(D)$ it follows that the index sets $\{1,5\} \in ans_{P_2}(D)$ and $\{1,3\} \in ans_{P_3}(D)$ are structurally consistent as they share the same value in the first position and have different values in the second position, whereas $\{1,5\} \in ans_{P_2}(D)$ and $\{2,3\} \in ans_{P_3}(D)$ are not structurally compatible and thus are not joinable.

The following definition states the meaning of structural consistency for two generic subtrees T_i and T_j of Q – paths P_i and P_j are particular instances of T_i and T_j.

Definition 3 (Structural consistency). *Let Q be a query twig, D a data tree, $T_i = \{t_i^1, \ldots, t_i^n\}$ and $T_j = \{t_j^1, \ldots, t_j^m\}$ two ordered sets of indexes determining $sub\_sig_{T_i}(Q)$ and $sub\_sig_{T_j}(Q)$, respectively, $ans_{T_i}(D)$ and $ans_{T_j}(D)$ the answers of the unordered inclusion of T_i and T_j in D, respectively.*

$S_i = \{s_i^1, \ldots, s_i^n\} \in ans_{T_i}(D)$ and $S_j = \{s_j^1, \ldots, s_j^m\} \in ans_{T_j}(D)$ are structurally consistent if:

- *for each pair of common indexes $t_i^h = t_j^k$, $s_i^h = s_j^k$;*
- *for each pair of different indexes $t_i^h \neq t_j^k$, $s_i^h \neq s_j^k$.*

Definition 4 (Join of answers). *Given two structurally consistent answers $S_i \in ans_{T_i}(D)$ and $S_j \in ans_{T_j}(D)$, where $T_i = \{t_i^1, \ldots, t_i^n\}$, $T_j = \{t_j^1, \ldots, t_j^m\}$, $S_i = \{s_i^1, \ldots, s_i^n\}$ and $S_j = \{s_j^1, \ldots, s_j^m\}$, the join of S_i and S_j, $S_i \bowtie S_j$, is defined on the ordered set $T_i \cup T_j = \{t^1, \ldots, t^k\}$ as the index set $\{s^1, \ldots, s^k\}$ where:*

- *for each $h = 1, \ldots, n$, $l \in \{1, \ldots, k\}$ exists such that $t_i^h = t^l$ and $s_i^h = s^l$;*
- *for each $h = 1, \ldots, m$, $l \in \{1, \ldots, k\}$ exists such that $t_j^h = t^l$ and $s_j^h = s^l$.*

Any answer to the unordered inclusion of Q in D is the outcome of a sequence of joins of structurally consistent answers, one for each $P \in rew(Q)$, identifying distinct paths in $sig(D)$. The answer set $ans_Q(D)$ can thus be computed by sequentially joining the sets of answers of the evaluation of the path queries. We denote such operation as structural join.

$ans_{P_2}(D)$: $\mathbf{P_2}$ | **1** | **2** | / | 1 | 5 |

$ans_{P_3}(D)$: $\mathbf{P_3}$ | **1** | **3** | / | 1 | 3 | / | 2 | 3 |

$sj(ans_{P_2}, ans_{P_3})$: $\mathbf{P_2 \cup P_3}$ | **1** | **2** | **3** | / | 1 | 5 | 3 |

Fig. 7. Structural join of Ex.1

Definition 5 (Structural join). *Let Q be a query twig, D a data tree, T_i and T_j two ordered sets of indexes determining $sub\_sig_{T_i}(Q)$ and $sub\_sig_{T_j}(Q)$, respectively, $ans_{T_i}(D)$ and $ans_{T_j}(D)$ the answers of the unordered inclusions of T_i and T_j in D, respectively.*

The structural join $sj(ans_{T_i}(D), ans_{T_j}(D))$ between the two sets $ans_{T_i}(D)$ and $ans_{T_j}(D)$ is the set $ans_T(D)$ where:

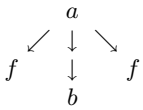
- *$T = \{t^1, \ldots, t^k\}$ is the ordered set obtained by the union $T_i \cup T_j$ of the ordered sets T_i and T_j;*
- *$ans_T(D)$ contains the join $S_i \bowtie S_j$ of each pair of structurally consistent answers $(S_i \in ans_{T_i}(D), S_j \in ans_{T_j}(D))$.*

The structural join $sj(ans_{T_i}(D), ans_{T_j}(D))$ thus returns an answer set defined on the union of the two sub-queries T_i and T_j and contains the join of the structurally consistent answers in $ans_{T_i}(D)$ and $ans_{T_j}(D)$. Starting from the set of answers $\{ans_{P_{x_1}}(D), \ldots, ans_{P_{x_k}}(D)\}$ to as many paths in $rew(Q)$, we can get the answer set $ans_Q(D)$ identifying the unordered inclusion of Q in D by incrementally merging the answer sets by means of the structural join. More precisely, notice that the structural join operator is associative and symmetric and thus we can compute $ans_Q(D)$ as:

$$ans_Q(D) = sj(ans_{P_{x_1}}(D), \ldots, ans_{P_{x_k}}(D)) \tag{1}$$

where $rew(Q) = \{P_{x_1}, \ldots, P_{x_k}\}$.

Example 4. The answer set $ans_Q(D)$ of Example 1 is the outcome of the structural join $sj(ans_{P_2}(D), ans_{P_3}(D)) = ans_{P_2 \cup P_3}(D)$ where $P_2 \cup P_3 = \{1, 2\} \cup \{1, 3\}$ is the ordered set $\{1, 2, 3\}$. The answers to the individual paths and the final answers are shown in Fig. 7 (the first line of each table shows the query). It joins the only pair of structurally consistent answers: $\{1, 5\} \in ans_{P_2}(D)$ and $\{1, 3\} \in ans_{P_3}(D)$.



$sig(Q) = \langle a, 4, 5, 0; f, 1, 3, 1; b, 2, 4, 1; f, 3, 5, 1\rangle$

Fig. 8. Sample of query

$ans_{P_2}(D)$: $\mathbf{P_2}$ | **1** | **2** |
|---|---|
| 1 | 5 |

$ans_{P_3}(D)$: $\mathbf{P_3}$ | **1** | **3** |
|---|---|
| 1 | 3 |
| 2 | 3 |

$ans_{P_4}(D)$: $\mathbf{P_4}$ | **1** | **4** |
|---|---|
| 1 | 5 |

$sj(ans_{P_2}(D), ans_{P_3}(D))$: $\mathbf{P_2} \cup \mathbf{P_3}$ | **1** | **2** | **3** |
|---|---|---|
| 1 | 5 | 3 |

Fig. 9. Structural join of Ex.5

Example 5. Given the query Q of Fig. 8, we show in this example the evaluation of the unordered tree inclusion of Q in the data tree D of Fig. 4. It can be easily verified that there is no qualifying sub-signature since at most two of the three paths find a correspondence in the data tree.

The rewriting phase produces the set $rew(Q) = \{P_2, P_3, P_4\}$ where $P_2 = \{1, 2\}$, $P_3 = \{1, 3\}$, and $P_4 = \{1, 4\}$. The final result $ans_Q(D)$ is the outcome of the structural join:

$$sj(ans_{P_2}(D), ans_{P_3}(D), ans_{P_4}(D)) = sj(sj(ans_{P_2}(D), ans_{P_3}(D)), ans_{P_4}(D)) = \emptyset$$

The answer sets of the separate paths and of $sj(ans_{P_2}(D), ans_{P_3}(D))$ are shown in Fig. 9. The final result is empty since the only pair of joinable answers $\{1, 5, 3\} \in sj(ans_{P_2}(D), ans_{P_3}(D))$ and $\{1, 5\} \in ans_{P_4}(D)$ are not structurally consistent: the two different query nodes $2 \in P_2 \cup P_3$ and $4 \in P_4$ correspond the same data node 5. It means that there are not as many data tree paths as query tree paths.

Theorem 1. *Given a query twig Q and a data tree D, the answer set $ans_Q(D)$ as defined by Eq. 1 contains all and only the index sets S qualifying the unordered inclusion of Q in D according to Lemma 2.*

4 Efficient computation of the answer set

Till now, we have studied how tree signatures can be employed to support unordered tree pattern matching. However, XML data trees can have many nodes and the tree signatures, linearly proportional to the number of nodes, can be very large, so the performance aspects of such operation becomes a matter of concern. In the previous section, we have specified two distinct phases for unordered tree pattern matching: the computation of the answer set for each root-to-leaf path of the query and the structural join of such sets. The main drawback of this approach is that many intermediate results may not be part of any final answer. In the following, we show how these two phases can be merged into one phase in order to avoid unnecessary computations. The basic idea is to evaluate at each step the most selective path among the available ones and to directly combine the partial results computed with structurally consistent answers of the paths.

The full algorithm is depicted in Fig. 10. It makes use of the `pop` operation which extracts the next element from the ordered set of paths $rew(Q)$. The algorithm essentially computes the answer set by incrementally joining the partial answers collected up to that moment with the answer set of the first path P in $rew(Q)$. As paths are sorted on

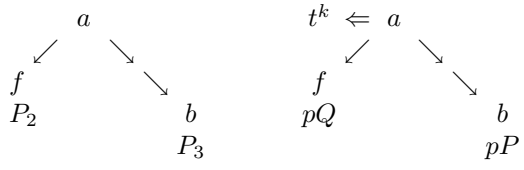
Input: the paths of the rewriting phase $rew(Q)$
Output: $ans_Q(D)$
Algorithm:

(1) P = pop($rew(Q)$);
(2) $pQ = P$;
(3) evaluate $ans_{pQ}(D)$;
(4) while(($rew(Q)$ not empty) AND ($ans_{pQ}(D)$ not empty))
(5) P = pop($rew(Q)$);
(6) $pP = P \setminus (P \cap pQ)$;
(7) t^k is the parent of pP, k is the position in pQ;
(8) $PAns = \emptyset$;
(9) for each answer S in $ans_{pQ}(D)$
(10) evaluate $ans_{pP}(sub\_sig_{\{s_k+1,\ldots,ff_{s_k}-1\}}(D))$;
(11) if($ans_{pP}(sub\_sig_{\{s_k+1,\ldots,ff_{s_k}-1\}}(D))$ not empty)
(12) add $sj(\{S\}, ans_{pP}(sub\_sig_{\{s_k+1,\ldots,ff_{s_k}-1\}}(D)))$ to $PAns$;
(13) pQ = $pQ \cup P$;
(14) $ans_{pQ}(D)$=$PAns$;}

Fig. 10. The unordered tree pattern evaluation algorithm

the basis of their selectivity, P is the most selective path among those which have not been evaluated yet. In particular, from step 1 to step 3, the algorithm initializes the partial query pQ evaluated up to moment to the most selective path P and stores in the partial answer set $ans_{pQ}(D)$ the evaluation of the inclusion of pQ in D. From step 4 to step 12, it iterates the process by joining the partial answer set $ans_{pQ}(D)$ with the answer set $ans_P(D)$ of the next path P of $rew(Q)$. Notice that, at each step, it does not properly compute first the answer set $ans_P(D)$ and the structural join $sj(ans_{pQ}(D), ans_P(D))$ as shown in Eq. 1, but it rather applies a sort of nested loop algorithm in order to perform the two phases in one shot. As each pair of index sets must be structurally consistent in order to be joinable, we compute only such answers in $ans_P(Q)$, which are structurally consistent with some of the answers in $ans_{pQ}(D)$. As a matter of fact, only such answers may be part of the answers to Q. In order to do it, the algorithm tries to extend each answer in $ans_{pQ}(D)$ to answers to $pQ \cup P$ by only evaluating such sub-path of P which has not been evaluated in pQ. In particular, step 6 stores in the sub-path pP such part of the path P to be evaluated not in common sub-path between P and the query pQ evaluated up to that moment: $P \cap pQ$. Step 7 identifies t^k as the parent of the partial path pP where k is its position in pQ. For each index set $S \in ans_{pQ}(D)$, each index set in $ans_P(Q)$, which is structurally consistent with S, must share the same values in the positions corresponding to the common sub-path $P \cap pQ$. Thus, in order to compute the answers in $ans_P(D)$ that are structurally consistent with S and, then, join with S, the algorithm extends S to answers to $P \cup pQ$ by only evaluating in the "right" sub-tree of the data tree the inclusion of the partial path pP of P which has not been evaluated yet (step 10). The right sub-tree consists of the descendants of the data entry s_k corresponding to the parent t_k of the partial path pP and in the tree signature corresponds

to the sequence of nodes having pre-order values from $s_k + 1$ up to $ff_{s_k} - 1$. Then it joins S with such answer set by only checking that different query entries correspond to different data entries (step 12). Notice that, in step 10, by shrinking the index interval to a limited portion of the data signature, we are able to reduce the computing time for the sequence inclusion evaluation.



(a) (b)

The algorithm ends when we have evaluated all the paths in $rew(Q)$ or when the partial answer set collected up to that moment $ans_{pQ}(D)$ becomes empty. The latter case occurs when we evaluate a path P having no answer which is structurally consistent with those in $ans_{pQ}(D)$: $sj(ans_{pQ}(D), ans_P(D)) = \emptyset$. In this case, for each answer S in $ans_{pQ}(D)$ two alternative exists. Either the evaluation of the partial path pP fails (line 11), which means that none of the answers in $ans_P(D)$ share the same values of S in the positions corresponding to the common sub-path $P \cap pQ$, or the structural join between S and the answers to pP fails (line 12), which means that some of the answers in $ans_P(D)$ share the same values of S in positions corresponding to different indexes in P and pQ.

In summary, the proposed solution performs a small number of additional operations on the paths of the query twig Q, but dramatically reduces the number of operations on the data trees by avoiding the computation of useless path answers. In this way, we remarkably reduce computing efforts. Indeed, while query twigs are usually very small and have a limited number of paths, XML trees can have many nodes and tree signatures can be very large.

Example 6. Let us apply the algorithm described in Fig. 10 to Example 1. Since the three paths are of the same length, we start from P_2 whose answer set is $ans_{P_2}(D) = \{\{1, 5\}\}$. Then, the algorithm extracts the next path, e.g. P_3, and computes $P_2 \cap P_3 = \{1\}$, $t_k = 1$, and $pP = \{3\}$. We then consider the only index set $S = \{1, 5\}$ in $ans_{P_2}(D)$ and evaluate $ans_{pP}(D)$ on the sub-tree rooted by $s_k = 1$ that is in the signature $sub\_sig_{\{2,3,4\}}(D) = \langle a, 3; b, 1; c, 2\rangle$. The outcome is thus $ans_{pP}(D) = \{\{3\}\}$ and $ans_Q(D) = \{\{1, 5, 3\}\}$. Being P_2 and P_3 of the same length, we can also start from $ans_{P_3}(D) = \{\{1, 3\}, \{2, 3\}\}$. In this case $pP = \{2\}$. We then consider the first index set $\{1, 3\}$ and evaluate $ans_{pP}(D)$ on the sub-tree rooted by $s_k = 1$. The answer is $ans_{pP}(sub\_sig_{\{2,3,4\}}(D) = \{\{5\}\}$. Thus $ans_Q(D) = \{\{1, 5, 3\}\}$. The next index set $\{2, 3\}$ requires to evaluate $ans_{pP}(D)$ on the sub-tree rooted by $s_k = 2$ that is the sub-tree in D having pre-order values from 3 up to 4 and the answer set is empty.

5 Performance evaluation

In this section we evaluate the performance of our unordered tree inclusion technique. We measured the time needed to process different query twigs using the paths decomposition approach, we deeply described in this paper, and compared it with the results obtained by the permutation approach.

All algorithms are implemented in Java JDK 1.4.2 and the experiments are executed on a Pentium 4 2.5Ghz Windows XP Professional workstation, equipped with 512MB RAM and a RAID0 cluster of 2 80GB EIDE disks with NT file system (NTFS).

Since synthetic data sets are not significant enough to show the performance of real-life XML query scenarios, we performed our experiments on a real data set, specifically the complete DBLP Computer Science Bibliography archive of April 2003. Table 1 shows more details about this XML archive. Notice that the file consists of over 3.8 Millions of elements, where over 3.4 Millions of them have an associated value. The size of the XML file is 156MB.

Middle-level		**Leaf-level**	
Element name	**Occs**	**Element name**	**Occs**
inproceedings	241244	author	823369
article	129468	title	376737
proceedings	3820	year	376709
incollection	1079	url	375192
book	1010	pages	361989
phdthesis	72	booktitle	245795
mastersthesis	5	ee	143068
		crossref	141681
		editor	8032
		publisher	5093
		isbn	4450
		school	77
Summary			
Total number of elements			3814975
Total number of values			3438237
Maximum tree height			3

Table 1. DBLP Test-Collection Statistics

We tested the performance of our approach for queries derived from six query twig templates (see Fig. 11). Such templates present different *element name selectivity*, i.e. the number of elements having a given element name, different *branching factors*, i.e. the maximum number of sibling elements, and different *tree heights*. We refer to the templates as "*xSb-h*', where S stands for element name selectivity and can be H(igh) or L(ow), b is the branching factor, and h the tree height. To understand the element name selectivity, refer to Table 1, showing the number of occurrences of each name in

the DBLP data set. In particular, we used `inproceedings` for the low selectivity and `book` and `phdthesis` for the high selectivity.

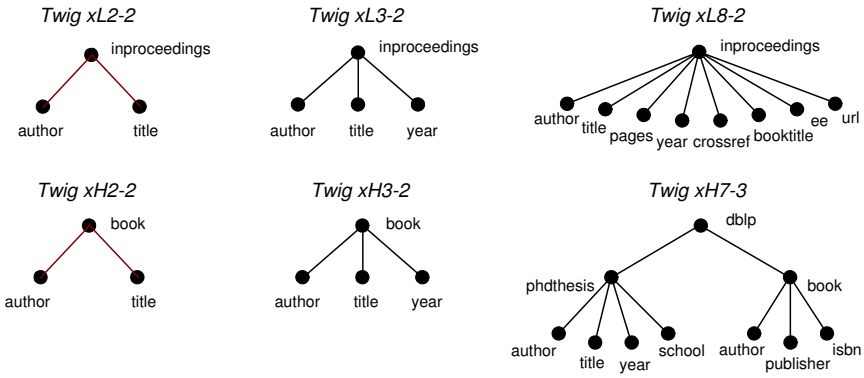


Fig. 11. The query twigs templates used in the performance tests

From each template, we derived both a no-value query (indicated with "*NSb-h*"), identical to the template, and a value query ("*VSb-h*"), obtained by adding a particular author name value to the author elements. Value accesses are supported by a content index. We have chosen the highly content-selective predicates, because we believe that value queries are especially significant for highly selective fields, such as the author name. On the other hand, the performance of queries with low selectivity fields should be very close to their no-value counterparts. With such query structure, we measure the response time for twelve queries, half of which have an associated value.

Query twig	**Elements**	**Values**	**Solutions retrieved**	**Decomposition (sec)**	**Permutation**		
					N	**mean (sec)**	**total (sec)**
NH2-2	3	0	1343	*0.016*	2	0.014	*0.028*
NH3-2	4	0	1343	*0.016*	6	0.015	*0.105*
NH7-3	10	0	90720	*1.1*	288	0.9	*259.2*
NL2-2	3	0	559209	*2.2*	2	2.28	*4.56*
NL3-2	4	0	559209	*4.2*	6	2.49	*14.94*
NL8-2	9	0	149700	*7.7*	40320	4.8	*193536*
VH2-2	3	1	1	*0.015*	2	0.014	*0.028*
VH3-2	4	1	1	*0.016*	6	0.016	*0.096*
VH7-3	10	2	1	*0.031*	288	0.03	*8.64*
VL2-2	3	1	39	*0.65*	2	0.832	*1.664*
VL3-2	4	1	36	*0.69*	6	1.1	*6.6*
VL8-2	9	1	29	*0.718*	40320	2.3	*92736*

Table 2. Performance comparison between the two unordered tree inclusion alternatives

Table 2 summarizes the results of the unordered tree inclusion performance tests for both the approaches we considered. For each query twig, the total number of elements and values, the number of solutions (inclusions) found in the data set, and the processing time, expressed in seconds, are reported. For the permutation approach, the number of needed permutations and the mean per-permutation processing time are also presented. It is evident that the decomposition approach is superior and scores a lower time in every respect. In particular, with low branching factors (i.e. 2), such approach is twice as faster for both selectivity settings. With high branching factors (i.e. 3, 8) the speed increment becomes larger and larger – the number of permutations required in the alternative approach grows factorially: for queries *NL8-2* and *VL8-2* the decomposition method is more than 25000 times faster. The decomposition approach is particularly fast with the high selectivity queries. Even for greater heights (i.e. in *VH7-3*), the processing time stays in order of milliseconds.

Notice that, for the decomposition method, we measured the time needed to solve each of the permutated versions of a query twig and reported only the lower one. As we expected, we found that starting with the most highly selective paths always increases the query evaluation efficiency. In particular, the time spent is nearly proportional to the number of occurrences of such a path in the data. Evaluating query *NL2-2* starting with the `title` path produces a response time of 2.2 seconds, while starting with the less selective `author` path , the time would nearly double (3.9 sec.). This holds for all the query twigs as well – for *NL8-2*, the time ranges from 7.7 sec (`crossref` path) up to 15.7 sec (`author` path). Of course, for the value queries the best time is obtained by starting the evaluation from the value-enabled paths.

Finally, notice that the permutation approach also requires an initial “startup” phase where all the different permutation twigs are generated; the time used to generate such permutations is not taken into account.

6 Conclusions

In this paper, we have studied the problem of efficient evaluation of unordered query trees in XML tree structured data collections. As the underlying concept, we have used the tree signatures, which have proved to be a useful structure for an efficient tree navigation and ordered tree matching, see [ZAD03]. We have identified two evaluation strategies, where the first strategy is based on multiple evaluation of all query tree structure permutations and the second on decomposing a query tree into a collection of all root-to leaf paths.

We have deeply studied the decomposition approach and established rules for decomposition as well as strategies for integration of partial, structurally consistent, results through structural joins. Based on the developed theoretical grounds, an efficient implementation algorithm is proposed.

The permutation and decomposition approaches to the unordered tree matching have been tested on the DBLP data set for various types of queries. The experiments demonstrate a clear superiority of the decomposition approach, which is especially advantages for the large query trees, and for trees with highly selective predicates.

Experiments also confirmed the expected fact that the order of evaluating the paths in the decomposition approach can have significant effects on the overall performance. Though the proposed strategy of starting with the longest path seems to work quite well, we plan to work on this aspect more deeply in the near future.

References

[Die82] P.F. Dietz. Maintaining Order in a Linked List, In *Proceedings of STOC, 14th Annual ACM Symposium on Theory of Computing*, May 1982, San Francisco, CA, 1982, pp. 122-127.

[Gr02] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002, pp. 109-120.

[SWS+02] D. Shasha, J.T.L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate Searching in Unordered Trees. Proceedings of the *14th International Conference on Scientific and Statistical Database Management*, July 24-26, 2002, Edinburgh, Scotland, UK, pp. 89-98.

[ZAD03] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree Signatures for XML Querying and Navigation. In *Proceedings of the XML Database Symposium, XSym 2003*, Berlin, September 2003, LNCS xxxx, Springer, pp. xxx-yyy.

[ZSS92] K. Zhang, R. Statman, and D. Shasha. On the edit distance between unordered labeled trees. *Information Processing Letters*, 42:133-139, 1992.