A New Algorithm for Evaluating Ordered Tree Pattern Queries

Yangjun Chen
Dept. Applied computer Science
University of Winnipeg, Winnipeg, Manitoba, Canada, R3b 2E9

Abstract - An XML tree pattern query, represented as a labeled tree, is essentially a complex selection predicate on both structure and content of an XML. Tree pattern matching has been identified as a core operation in querying XML data. We distinguish between two kinds of tree pattern matchings: ordered and unordered tree matching. By the unordered tree matching, only ancestor-descendant and parent-child relationships are considered. By the ordered tree matching, the order of siblings is also taken into account. While different fast algorithms for unordered tree matching are available, no efficient algorithm for ordered tree matching for complexity is polynomial. In addition, the algorithm can be adapted to an indexing environment with XB-trees being used.

Keywords: XML data stream, tree pattern queries, ordered tree matching

1 Introduction

In XML [43, 44], data is represented as a tree; associated with each node of the tree is an element name from a finite alphabet Σ. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

Accordingly, in most of the XML query languages (e.g. XPath [43], XQuery [44], XML-QL [15], and Quilt [6, 7]), queries are typically expressed by tree patterns (for example, path expressions expressed in XPath, path expressions in the for and let clauses in XQuery.) In such tree patterns, nodes are labeled with symbols from Σ ∪ {*} (* is a wildcard, matching any node name) and string values, and edges are parent-child or ancestor-descendant relationships. As an example, consider the query tree shown in Fig. 1(a), which asks for any node of name b (node 3) that is a child of some node of name a (node 1). In addition, the node of name b (node 3) is the parent of some nodes of name c and e (node 6 and 7, respectively), and the node of name e itself is an ancestor of some node of name d (node 8). The node of name b (node 2) should also be the ancestor of a node of name f (node 5). The query corresponds to the following XPath expression:

\[ a[b[c \text{ and } .//f]]/b[c \text{ and } e//d] \].

Fig. 1(a), there are two kinds of edges: child edges (/edges for short) for parent-child relationships, and descendant edges (/edges for short) for ancestor-descendant relationships. A /edge from node v to node u is denoted by v \( \Rightarrow u \) in the text, and represented by a single arc; u is called a /child of v. A //edge is denoted by v \( \Rightarrow u \) in the text, and represented by a double arc; u is called a //child of v.

![Fig. 1. A query tree and a tree matching a path](image)

In any DAG (directed acyclic graph), a node u is said to be a descendant of a node v if there exists a path (sequence of edges) from v to u. In the case of a tree pattern, this path could consist of any sequence of /edges and/or //edges. We also use label(v) to represent the symbol (∈ Σ \cup {*}) or the string associated with v. Based on these concepts, the tree embedding can be defined as follows.

**Definition 1** An embedding of a tree pattern Q into an XML document T is a mapping f: Q \( \rightarrow T \), from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) Preserve node label: For each \( u \in Q \), label(u) = label(f(u)) (or say, u matches f(u)).

(ii) Preserve parent-child/ancestor-descendant relationship: If \( u \Rightarrow v \) in Q, then f(v) is a child of f(u) in T; if \( u \Rightarrow v \) in Q, then f(v) is a descendant of f(u) in T.

If there exists a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q.

Almost all the existing strategies for evaluating twig join patterns are designed according to this definition [4, 8, 10, 11, 12, 14, 24, 26, 28, 29, 30, 31, 36, 37, 46]. This definition allows a path to match a tree as illustrated in Fig. 1(b).

It is because by Definition 1 the left-to-right relationships between siblings are not taken into account. We call such a problem an unordered tree pattern matching.

We may consider another problem, called an ordered tree pattern matching, defined below.

**Definition 2** An embedding of a tree pattern Q into an XML document T is a mapping f: Q \( \rightarrow T \), from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) same as (i) in Definition 1.
sentences. Therefore, by the evaluation of a tree pattern query

In general, a node in a XML database, we can always store a document as a
tree of nodes. This kind of tree mappings is useful in practice. For
computational complexities. The paper concludes in 3.3, we

The remainder of the paper is organized as follows. In
Section 2, we review the concept of XML data streams. In
Section 3, we discuss our algorithm and analyze its
computational complexities. The paper concludes in Section
4.

2 XML data stream

In a XML database, we can always store a document as a
data stream by using an interesting tree encoding [46], which
can be used to identify different relationships between the
nodes of a tree.

Let T be a document tree. We associate each node v in T
with a quadruple (d, l, r, ln), denoted as (v(l)), where d =
DocID, l = LeftPos, r = RightPos, and ln = LevelNum, defined
to be the nesting depth of the element in the document. (See
Fig. 2 for illustration.) By using such a data structure, the
structural relationships between the nodes in an XML
database can be simply determined [46]:

(i) ancestor-descendant: a node v1 associated with (d1, l1, r1,
ln1) is an ancestor of another node v2 with (d2, l2, r2, ln2)
iff d1 = d2, l1 < l2, and r1 > r2.

(ii) parent-child: a node v1 associated with (d1, l1, r1, ln1) is
the parent of another node v2 with (d2, l2, r2, ln2) iff d1 =
d2, l1 < l2, r1 > r2, and ln2 = ln1 + 1.

(iii) left-to-right order: a node v1 associated with (d1, l1, r1,
ln1) is to the left of another node v2 with (d2, l2, r2, ln2)
iff d1 = d2, r1 < l2.

In Fig. 4, v2 is an ancestor of v6 and we have v5.LeftPos =
the same way, we can verify all the other relationships of
the nodes in the tree. In addition, for each leaf node v, we set
v.LeftPos = v.RightPos for simplicity, which still work
without downgrading the ability of this mechanism. In the rest
of the paper, if for two quadruples α1 = (d1, l1, r1, ln1) and α2 =
(d2, l2, r2, ln2), we have d1 = d2, l1 ≤ l2, and r1 ≥ r2, we say
that α2 is subsumed by α1. For convenience, a quadruple is
considered to be subsumed by itself (i.e., a node is considered
to be an ancestor of itself). In this way, we can store an XML
document as a stream of quadruples sorted by LeftPos or
RightPos values.

Fig. 2. Illustration for tree encoding

If no confusion is caused, we will used v and α(v)
interchangeably. As with DeweyIDs [21], we can also leave
gaps in the numbering space between consecutive labels to
support dynamical changes of documents.

3 Algorithm

In this section, we discuss our strategy for the ordered
tree pattern matching. First, we discuss the main algorithm in
3.1. Then, in 3.2, we show the correctness of our algorithm
and analyze its computational complexities. In 3.3, we
describe how to adapt it to an indexing environment, as well
as how the wildcard and output node can be handled.

3.1 Algorithm description

Our algorithm works bottom-up. Therefore, we need to
sort XML streams by (DocID, RightPos) values. Each time a
query Q is submitted to the system, we will associate each
query node q with a data stream L(q) such that for each v ∈
L(q) label(v) = label(q), as illustrated in Fig. 3, in which each
query node is attached with a list of matching nodes of the
document tree shown in Fig. 2.

Fig. 3. Illustration for L(q)’s

In the figure, for simplicity, we use the node names in a
data stream, instead of the nodes’ quadruples. In addition,
DocIDs are not displayed. We remark that the data streams
associated with different nodes in Q may be the same. So we
use q to represent the set of such query nodes and denote by
L(q) the data stream shared by them. Without loss of
generality, assume that the query nodes in q are sorted by
their RightPos values. We will also use L(Q) = {L(q1), ..., L(ql)} to represent all the data streams with respect to Q,
where each qi (i = 1, ..., l) is a set of sorted query nodes that
share a common data stream.

In order to facilitate the checking of tree embedding,
some more data structures are established for query nodes.
1. First, we will number the nodes of \( Q \) in postorder (see the boldfaced numbers in Fig. 4(a) for illustration). So the nodes in \( Q \) will be referenced by their postorder numbers.

2. For each node \( q \) of \( Q \), a link from it to the left-most leaf node in \( Q[q] \), denoted by \( \delta(q) \), is established. (See Fig. 4(b)). For a leaf node \( q' \), \( \delta(q') = q' \). Additionally, we set a virtual node for \( Q \), numbered 0, which is considered to be to the left of any node in \( Q \).

3. Let \( q' \) be a leaf node in \( Q \). We denote by \( \delta^{-1}(q') \) a set of nodes \( x \) such that for each \( q \in x \) \( \delta(q) = q' \).

4. Each time we create a node \( v \) in \( T' \), we associate it with an array \( A_v \) of length \( |Q| \), indexed from 0 to \( |Q| - 1 \). In \( A_v \), each entry is a query node or \( \phi \), defined below:

\[
A_v[q] = \begin{cases} 
\text{max} [x \mid x \in \delta^{-1}(q') \land \text{if there is a least leaf } q' \text{ larger than } q \text{ such that } T[v] \text{ embeds } Q[x], \delta^{-1}(q') \text{ contains at least one node } x \text{ with } Q[x] \text{ being embedded in } T[v]; & x \in \delta^{-1}(q') \\
\phi, & \text{otherwise}.
\end{cases}
\]

Here, \( Q[x] \) represents a subtree of \( Q \) rooted at \( x \).

See Fig. 5 for illustration.

As an example, consider node \( v_q \) in \( T \) shown in Fig. 2. After it is checked against node 1 (\( q_3 \)) of \( Q \) in Fig. 4, we will set \( A_{v_q}[0] \) to 1 since node 1 (\( q_3 \)) of \( Q \) is the closest node to the right of node 0 (the virtual node of \( Q \) such that \( T[v_q] \) embeds \( Q[q_3] \)). (See Fig. 6(a)). At a later time point, we find that \( T[v_q] \) also embeds \( Q[q_5] \). We will change \( [0] \) to 3 (see Fig. 6(b)). It is because node 1 (\( q_3 \)) is a descendant of node 3 (\( q_5 \)) on the left-most path in \( Q[q_5] \). In the subsequent computation, we will find that \( T[v_q] \) can embed \( Q[q_4] \). In order to record this fact, we will further modified as shown in Fig. 6(c) since node 4 (\( q_4 \)) is the closest node right of node 1 (\( q_3 \)), 2 (\( q_4 \)), and 3 (\( q_4 \)) such that \( T[v_q] \) embeds \( Q[q_4] \).

Based on \( A_v \)'s, the ordered tree embedding can be checked as follows:

- Let \( q \) in \( Q \) and \( v \) in \( T' \) be the nodes encountered.
- Let \( v_1, \ldots, v_k \) be the child nodes of \( v \). Let \( q_1, \ldots, q_l \) be the child nodes of \( q \). We first check \( A_{v_1} \) starting from \( A_{v_k} \), where \( l = \delta(q) - 1 \). We begin the searching from \( \delta(q) - 1 \) because it is the closest node to the left of the first child of \( q \). Let \( A_{v_k}[l] = q' \). If \( q' \) is not an ancestor of \( q_k \), we will check \( A_{v_k} \) in the next step. This process continues until one of the following conditions is satisfied:
  (i) All \( A_v \)'s have been checked, or
  (ii) There exists \( v_j \) such that \( [l] \) is an ancestor of \( q_j \).

If all \( A_v \)'s are checked (case (i)), it shows that \( Q[q_i] \) cannot be embedded in any subtree rooted at a child node of \( v \). So \( T[v] \) cannot embed \( Q[v] \).

If it is case (ii), we know that \( T[v_j] \) embeds \( Q[q_i] \). If \( q_i \) is a left/child, or both \( q_i \) and \( v_j \) are left/children, we will continue to check \( q_i \). (Otherwise, we will continue to check \( A_{v_j} \).)

In terms of the above discussion, we give our algorithm for evaluating ordered tree pattern queries. It mainly consists of two processes: (1) scanning the data streams associated with the query nodes in such an order that each time the quadruple with the least RightPos is accessed; (2) checking tree embedding.

In the first process, for each encountered quadruple \( v \) from a \( L(q) \), a node is created, which will be associated with two links, denoted respectively left-sibling\( (v) \) and parent\( (v) \), to reconstruct \( T \) (or a subtree of \( T \), which contains only those nodes matching a query node) as follows:

1. Identify a data stream \( L(q) \) with the first element being of the minimal RightPos value. Choose the first element \( v \) of \( L(q) \). Remove \( v \) from \( L(q) \).
2. Generate a node for \( v \).
3. If \( v \) is not the first node, we do the following:
   - Let \( v' \) be the node chosen just before \( v \). If \( v' \) is not a child (descendant) of \( v \), create a link from \( v \) to \( v' \), called a left-sibling link and denoted as left-sibling\( (v) = v' \).
   - If \( v' \) is a child (descendant) of \( v \), we will first create a link from \( v' \) to \( v \), called a parent link and denoted as parent\( (v') \).
Input: all data streams be integrated. In the second process, we check, for each v from a L(q), whether T[v] embeds Q[q] for each q in q. For this purpose, in addition to δ(q), A_v, another two data structures are used: S_v - a list of query nodes q such that T[v] embeds Q[q]. τ(v) - a postorder number for a query node associated with node v in T. Such that τ(v) is the subtree currently found to embed Q[τ(v)]. The initial value for each τ(v) is 0.

In the following algorithm, we only show the second process for ease explanation. But the first process can easily be integrated.

Algorithm tree-embedding(L(Q))
Input: all data streams L(Q).
Output: S_v’s, which show the tree embedding.
begin
1. repeat until each L(q) in L(Q) become empty
2. (identify q such that the first element v of L(q) is of the minimal RightPos value; remove v from L(q));
3. generate node v; A_v ← φ; S_v ← φ;
4. let v_1, ..., v_l be the children of v.
5. for each q ∈ q do { (*nodes in q are sorted.*)
6.  let q_1, ..., q_l be the children of q;
7.  if l = 0 then j ← 0
8.  else { p ← δ(q) - 1;
9.      i ← 1; j ← 1; p′ ← A_v [p];
10.     while i ≤ k and p′ ≠ φ and p′ < q do
11.        if (p′ is an ancestor of q_j and (q, q_j) is a /-edge, or both (q, q_j) and (v, v_i) are /-edges))
12.           then { p′ ← A_v [p′]; i ← i + 1; j ← j + 1;}
13.           else { p′ ← A_v [p′]; i ← i + 1;}
14.    }
15.  }
16.  if l = j then
17.    { S_v ← S_v ∪ {q};
18.     if q is to the right of τ(v)
19.      then { a ← τ(v);
20.           for b = a to q - 1 do
21.             { if b is to the left of q then A_v [b] ← q;}
22.          } else { replace with q all those entries pointing to a descendant of q on the left-most path in Q[q] in A_v; }
23.     }
24. }
25. for i = 1 to k do { A_v ← merge(A_v, A_v’);}
26. remove A_v, ..., A_v;
end

In the above algorithm, the nodes in T are created one by one. But for each node v generated for an element from a L(q), A_v is created and each entry is initialized to φ. Then, for each q ∈ q, we will check whether T[v] embeds Q[q]. This is done by executing lines 7 - 15, in which two index variables: i and j are used to scan the children of v and q, respectively. The searching begins from [p], where p = δ(q) - 1 (see line 8). In each iteration of the while-loop (see lines 10 - 14), we check v_j against q_j by examining whether the following two conditions are satisfied:

i) A_v [p] is an ancestor of q_j, and
ii) (q, q_j) is a /-edge, or both (q, q_j) and (v, v_i) are /-edges.

If both the conditions hold, T[v] embeds Q[q]. We will continue to check T[v_j] against Q[q_j]. Special attention should be paid to the statement: p’ ← A_v [p] (line 12), by which we get a query node q_j that is the closest to the right of q_j, such that T[v_j] embeds Q[q_j]. We also notice that if T[v_i] cannot embed Q[q_j], we will check v_i against q_i by doing p’ ← A_v [p] (see line 13).

This process continues until one of the following conditions is met: (1) i > k, (2) p = φ, or (3) p ≥ q. (1) or (2) implies an unsuccessful checking. If (3) holds, we must have l = j (see line 16), showing that each Q[q_i] (1 ≤ j ≤ l) is embedded by a T[v_i] (1 ≤ i ≤ k) in the left-to-right order.

Lines 18 - 23 are used to set the entries in A_v. Finally, we need to merge each A_v into A_v (line 25) since the embedding of a subtree in T[v_i] implies the embedding of that subtree in T[v].

Handling φ as a negative integer (e.g., -1) that represents a descendant of any node, we define merge(A_v, A_v’) as below:

merge(A_v, A_v’) = \{ \begin{align*}
\text{max}(A_v [j], A_v’ [j]), & \text{ if } A_v [j] \text{ and } A_v’ [j] \text{ are on the same path;} \\
\text{max}(A_v [j], A_v’ [j]), & \text{ otherwise.}
\end{align*} \}

Obviously, if A_v [j] and A_v’ [j] are on the same path, merge(A_v, A_v’) A_v [j] should be set to be max(A_v [j], A_v’ [j]). However, if A_v [j] and A_v’ [j] are on different paths, merge(A_v, A_v’) A_v [j] is set to be min(A_v [j], A_v’ [j]). It is because in A_v, each entry A_v [j] is the closest node j’ to the right of j such that T[v] contains Q[j’].

In line 26, we remove A_v,..., A_v since they will not be used any more.

Example 2 As an example, consider Q shown in Fig. 3 and T in Fig. 3 once again. The nodes in Q are postorder numbered, i.e., q_1 = 5, q_2 = 3, q_3 = 1, q_4 = 2, and q_5 = 4. When we apply the above algorithm to them, each node v (except v_j) in T will be associated with an array A_v as shown in Fig. 6.

In Step 1, v_3 is checked against q'' = {2}. Node 2 is a leaf node. So, we have T[v_1] embedding Q[2] and A_v will be established as shown in Fig. 6(a). We notice that A_v [0] = A_v [1] = 2. It is because node 2 is the closest node to the right of node 0 (virtual node) and node 1 (q_3) such that T[v_1] embeds Q[2].
In Step 2, \( v_3 \) is checked against \( q' = \{1, 3, 4\} \). Node 1 is a leaf and so \( T[v_3] \) embeds \( Q[1] \), which sets \( A_{v_3} \) as shown in Fig. 6(b). \( T[v_3] \) is not able to contain \( Q[3] \), but \( Q[4] \). Thus, \( A_{v_3} \) is changed to \([1, 4, 4, 4, \phi]\).

In Step 3, \( v_6 \) is checked against \( q'' = \{2\} \). Node 2 is a leaf and \( T[v_6] \) embeds \( Q[2] \). \( A_{v_6} \) is the same as . See Fig. 6(c).

In Step 4, \( v_4 \) is checked against \( q' = \{1, 3, 4\} \). Since \( T[v_4] \) embeds \( Q[1] \), \( A_{v_4} \) is first set to \([1, \phi, \phi, \phi, \phi]\) (see Fig. 6(d)). When \( v_4 \) is checked against node 3, their children will be examined. The children of \( v_4 \) are \( v_5 \) and \( v_6 \); and the children of node 3 are nodes 1 and 2. First, \( A_{v_4}[0] \) is checked. It is 1, showing that \( T[v_5] \) embeds \( Q[1] \). Next, \( A_{v_4}[1] \) is checked, it is equal to 2, showing that \( T[v_6] \) embeds \( Q[2] \). Therefore, \( T[v_4] \) is able to embed \( Q[3] \) and \( A_{v_4} \) is changed to \([3, \phi, \phi, \phi, \phi]\).

(Note that node 1 is a child of node 3 and also on the left-most path in \( Q[3] \).) By checking \( v_4 \) against node 4, \( A_{v_4} \) becomes \([3, 4, 4, 4, \phi]\). By \( \text{merge}(A_{v_4}, A_{v_5}) \), \( A_{v_6} \) is not changed. But by \( \text{merge}(A_{v_4}, A_{v_6}) \), \( A_{v_6} \) is further changed to \([3, 2, 4, 4, \phi]\).

\[A_{v_4} = [2, 2, \phi, \phi, \phi]\]
\[A_{v_5} = [1, \phi, \phi, \phi, \phi] \Rightarrow [1, 4, 4, 4, \phi]\]  
\[A_{v_6} = [2, 2, \phi, \phi, \phi] \Rightarrow [3, 4, 4, 4, \phi] \Rightarrow [3, 2, 4, 4, \phi]\]  
\[A_{v_4} = [1, \phi, \phi, \phi, \phi] \Rightarrow [3, 4, 4, 4, \phi] \Rightarrow [3, 2, 4, 4, \phi]\]  
\[A_{v_5} = [1, \phi, \phi, \phi, \phi] \Rightarrow [3, 4, 4, 4, \phi] \Rightarrow [3, 2, 4, 4, \phi]\]  
\[A_{v_6} = [1, 4, 4, 4, \phi] \Rightarrow [5, 2, 4, 4, \phi]\]  
\[A_{v_4} = [5, \phi, \phi, \phi, \phi] \Rightarrow [5, 2, 4, 4, \phi]\]  
\[A_{v_5} = [5, \phi, \phi, \phi, \phi] \Rightarrow [5, 2, 4, 4, \phi]\]

Fig. 6. A sample trace

The same analysis applies to Step 5 and 6, by which and are constructed as shown in Fig. 6(e) and (f), respectively.

In Step 7, \( v_1 \) is checked against \( q = \{5\} \). We will first check their children. The children of \( v_1 \) are \( v_2 \) and \( v_6 \); and the children of node 5 are nodes 3 and 4. Since \( A_{v_1}[0] = 3 \), showing that \( T[v_2] \) contains \( Q[3] \). However, since the edge (5, 3) (i.e., \( (q_1, q_3) \)) in \( Q \) is a /-edge, we have to check whether \( v_2 \) in \( T \) is a /-child of \( v_1 \). It is the case. So we will continue to check \( A_{v_1}[3] \). It is equal to 4, demonstrating that \( T[v_3] \) embeds \( Q[4] \). Thus, \( A_{v_1} \) is set to \([5, \phi, \phi, \phi, \phi]\). See Fig. 6(g). By \( \text{merge}(A_{v_1}, A_{v_2}) \), \( A_{v_2} \) is changed to \([5, 2, 4, 4, \phi]\). By \( \text{merge}(A_{v_1}, A_{v_2}) \), \( A_{v_2} \) remains unchanged.

### 3.2 Correctness and time complexities

In this subsection, we prove the correctness of the algorithm and analyze its computational complexities.

**Proposition 2** Algorithm \( \text{tree-matching} \) computes the values in \( A_{v} \), correctly.

**Proof.** We prove the proposition by induction on the heights of nodes in \( T \). We use \( h(v) \) to represent the height of node \( v \).

**Basic step.** It is clear that any node \( v \) with \( h(v) = 0 \) is a leaf node. Then, each entry in \( A_{v} \) corresponds to a leaf node \( q \) in \( Q \) with \( \text{label}(v) = \text{label}(q) \). Since all those leaf nodes in \( Q \) are checked in the order of increasing RightPos values, the entries in \( A_{v} \) must be correctly established.

**Induction step.** Assume that for any node \( v \) with \( h(v) \leq l \), the proposition holds. We will check any node \( v \) with \( h(v) = l + 1 \).

Let \( v_1, ..., v_k \) be the children of \( v \). Then, for each \( v_i (i = 1, ..., k) \), we have \( h(v_i) \leq l \). In terms of the induction hypothesis, each \( A_{v_i} \) is correctly constructed. Let \( q_1, ..., q_k \) be the children of \( q \). In the main while-loop, we will access a sequence:

\[A_{v_1}[p_1], ..., A_{v_k}[p_k]\]

with \( p_1 \leq p_2 \leq ... \leq p_k \). If there exists a subsequence: \( p_1, ..., p_i \) satisfying the following conditions:

i) \( p_i \) is an ancestor of \( q_i \), and

ii) \( (q_i, q_j) \) is a /-edge, or both \( (q_i, q_j) \) and \( (v, v') \) are /-edges, \( T[v] \) embeds \( Q[q] \). If \( q \) is to the right of \( \tau(v) \), then in \( A_{v_i} \), all the entries to the right of \( \tau(v) \) but to the left of \( q \) will be set to be \( q \). If \( q \) is an ancestor, all those entries pointing to a descendant of \( q \) and appearing on the left-most path in \( Q[q] \) are replaced with \( q \). The merging operation is obviously correct since the embedding of a subtree in \( T[v] \) implies that \( T[v] \) also contains that subtree. This completes the proof.

Now we analyze the time complexity of the algorithm. The whole cost can be divided into four parts.

The first part consists of checking \( v \) of \( T \) against \( q \) of \( Q \). Since in each \( A_{v} \), where \( v \) is a child of \( v \), only one entry is checked, this part of cost is bounded by \( O(\sum_{v \in T} d_v) = O(|T|) \), where \( d_v \) represents the outdegree of \( v \).

The second part is the cost for filling in the case that \( q \) is to the right of \( \tau(v) \). For each \( v \in T \), the cost is bounded by \( O(|Q|) \). So this part of cost is in the order of \( O(|T|\cdot |Q|) \).

The third part is the cost for filling in the case that \( q \) is an ancestor of \( \tau(v) \). This part of checking can be slightly improved as follows. In \( A_{v} \), each entry is set to be a pointer to a place storing a postorder number, instead of the number itself, as illustrated in Fig. 7.

In Fig. 7, \( A_{v} \) is stored as an array of pointers. Especially, the postorder numbers in \( A_{v} \) can be organized as a tree in a way similar to the reconstruction of \( T \) from data streams. Therefore, to modify all the entries pointing to a descendant of \( q \) on the left-most path in \( Q[q] \), we need only to search for the place containing the corresponding postorder number. This can be done by traversing along a left-link chain as discussed in 3.1. For a node \( v \in T \) checked against \( q \), the cost of this process is bounded by \( O(q) \).

The forth part of cost is for the merging operation. It can simply be estimated by

\[O(= O(|T|\cdot |Q|)).\]
In terms of the above analysis, we have the following proposition.

**Proposition 3** The time complexity of Algorithm `tree-embedding()` is bounded by \(O(|T|\cdot|Q| + O(|D|\cdot|Q|))\).

The space overhead of Algorithm `tree-embedding()` is in the order of \(O(leaf_f\cdot|Q|)\), where \(leaf_f\) is the number of the leaf nodes of \(T\). It is because after a \(v\) is checked all the arrays associate with its children are removed. So at any time point during the execution, at most \(leaf_f\) nodes in \(T\) are associated with a array (see line 26 in Algorithm `tree-embedding()`).

### 3.3 About index, *, and output nodes

In the previous subsections, the main algorithm has been described in detail. However, three issues yet remain to be addressed. That is, the indexing, wildcards (*) as well as the output node in \(Q\) should be handled carefully.

**- Index**

The index mechanism used in our implementation is a modified XB-tree [4]. As with TwigStack [4], an XB-tree is established over a data stream sorted by LeftPos values. However, we can use the following algorithm to make a transformation of data streams, in which a global stack \(ST\) is maintained to control the process. In \(ST\), each entry is a pair \((q, v)\) with \(q \in Q\) and \(v \in T\) (\(v\) is represented by its quadruple.)

**Algorithm stream-transformation(B(q)_s)**

input: all data streams \(B(q)_s\)'s, each sorted by LeftPos.

output: new data streams \(L(q)_s\)'s, each sorted by RightPos.

```
begin
1. repeat until each \(B(q)_i\) becomes empty
2. \{(identify \(q_i\) such that the first element \(v\) of \(B(q)_i\) is of the minimal LeftPos value; remove \(v\) from \(B(q)_i\);
3. while \(ST\) is not empty and \(ST.top\) is not \(v\)'s ancestor do
4. \{ \(x \leftarrow ST.pop()\); Let \(x = (q, u)\);
5. put \(u\) at the end of \(L(q)_i\); \}
6. \(ST.push(q, v)\);
7. \}
end
```

In the above algorithm, \(ST\) is used to keep all the nodes on a path until we meet a node \(v\) that is not a descendant of \(ST.top\).

Then, we pop up all those nodes that are not \(v\)'s ancestor; put them at the end of the corresponding \(L(q)_i\)'s (see lines 3 - 5); and push \(v\) into \(ST\) (see line 6.) The output of the algorithm is a set of data streams \(L(q)_i\)'s with each being sorted by RightPos values. However, we remark that the popped nodes are in postorder. So we can directly handle the nodes in this order without explicitly generating \(L(q)_i\)'s.

In Fig. 8(b), we demonstrate a XB-tree built on a \(B(q)\) shown in Fig. 8(a).

Each entry in a page (a node) \(P\) of an XB-tree consists of a bounding segment [LeftPos, RightPos] and a pointer to its child page, which contains entries with bounding segments completely included in [LeftPos, RightPos]. The bounding segments may partially overlap, but their LeftPos positions are in increasing order. Besides, each page has two extra data fields: \(P.parent\) and \(P.parentIndex\). \(P.parent\) is a pointer to the parent of \(P\), and \(P.parentIndex\) is a number \(i\) to indicate that the \(i\)th pointer in \(P.parent\) points to \(P\). For instance, in the XB-tree shown in Fig. 8(b), \(P_i.parentIndex = 2\) since the second pointer in \(P_1\) (the parent of \(P_3\)) points to \(P_5\).

![Fig. 8. A quadruple sequence and the XB-tree over it](image)

In our implementation, some modifications have been made. First, for a set of nodes \(q = \{q_1, ..., q_l\}\), we establish only one XB-tree, where \(q_1, ..., q_l\) have the same label. But for each \(q_j \in q\ (j = 1, ..., l)\), we maintain a pair \((P, i)\), denoted , to indicate that the \(i\)th entry in the page \(P\) (in the XB-tree) is currently accessed for \(q_j\). Thus, each \((j = 1, ..., l)\) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over \(B(q_i)\). We use \(advance()\) and \(drilldown()\) to navigate the corresponding XB-tree. Concretely, \(advance()\) advances \(i\). If \(i\) is the last entry in \(P\), \(P\) is replaced with \((P.parent, P.parentIndex)\). By \(drilldown()\), we replace \((P, i)\) with \((P', 0)\) if \(P\) is not a leaf page, where \(P'\) is the child page pointed to by the pointer of the \(i\)th entry in \(P\).

The second modification consists in a different navigation strategy of XB-trees. By Twigstack [4], each time to determine a \(q\) in \(Q\), for which an entry from \(B(q)\) is taken, the following three conditions are satisfied:

i) For \(q\), there exists an entry \(v_q\) in \(B(q)\) such that it has a descendant in each of the streams \(B(q)_i\) (where \(q_i\) is a child of \(q\).)

ii) Each recursively satisfies (i).

iii) \(LeftPos(v_q)\) is minimum.

But for the ordered tree matching, (i) is changed:

- For \(q\), there exists an entry \(v_q\) in \(B(q)\) such that it has a descendant in each of the streams \(B(q)_i\). If \(q\) has a right sibling \(q'\), then there exists an entry \(v_q'\) in \(B(q')\), which is to the right of \(v_q\).

In this way, not only the ancestor-descendant relationship, but also the left-to-right order is utilized to skip over entries in an XB-tree, which substantially reduces the number of disk access.

**- Wildcards**

Using XB-trees, * is handled in the same way as non-wildcard nodes. In fact, for each \(q\) in \(Q\), no matter whether it is a wildcard or not, we will be looking for only one element in the corresponding XB-tree each time. More importantly, using the above \(drilldown\) and \(advance\) operators [4], any entry in an XB-tree (corresponding to a query node) is accessed only once.
- Output node

As for the output node of $Q$, we should notice that the set $S_q$ generated for each node $v$ in $T'$ does not serve as the answer to $Q$ although for each $q \in S_q$ we have $Q|q$. For this reason, we need to slightly modify the algorithm to create two extra data structures $L_o$ and $L_e$ as below.

Each time we insert a $q$ into an $S_q$ (see line 17 in Algorithm $tree-embedding$), we will also add $v$ to $L_o$ if $q$ is the root $r$ of $Q$, or to $L_e$ if $q$ is the output node $o$.

Clearly, in these two data structures, all nodes are increasingly sorted by the RightPos values. Thus, using them, we can create another subtree $T''$ of $T$ (in a way similar to the generation of a matching subtree; see Algorithm $matching-tree-construction$). It contains only those nodes $v$ such that $T'[v]$ embeds $Q[r]$ with $label(v) = label(r)$ or embeds $Q[o]$ with $label(v) = label(o)$. We call a node $v$ an $r$-node if $T'[v]$ contains $Q[r]$ with $label(v) = label(r)$, or an $o$-node if $T'[v]$ embeds $Q[o]$ with $label(v) = label(o)$. Search $T''$. Any node $v$, which is an $o$-node and also a child of some $r$-node, should be an answer if $o$ is a descendant of $r$ or a $/-$child of $r$. If $o$ is a $/-$child of $r$, an $o$-node is an answer only if it is a $/-$child of some $r$-node.

4 Conclusion

In this paper, a new algorithms $tree-embedding$ for processing ordered tree pattern queries is discussed. For the ordered tree pattern queries, not only the parent-child and ancestor-descendant relationships but also the order of siblings are taken into account. The time complexity of the algorithm is bounded by $O(|T'|-|Q|)$ and its space overhead is by $O(leaf_T-|Q|)$, where $Q$ stands for a tree pattern, $T'$ for a subtree of a document tree $T$ containing the nodes that match at least one query node, and $leaf_T$ represents the number of the leaf nodes of $T'$. Our experiments demonstrate that our method is both effective and efficient for the evaluation of ordered tree pattern queries.

5 References


