Tree Inclusion Checking Revisited

Yangjun Chen[1] and Yibin Chen[2]
Dept. Applied Computer Science, University of Winnipeg, Poratge Ave., Winnipeg, Canada
[1]*y.chen@uwinnipeg.ca,* [2]*chenyibin@gmail.com*

Keywords: Tree inclusion, ordered labelled trees, tree matching.

Abstract: In this paper, we discuss an efficient algorithm for the ordered tree inclusion problem, by which it is checked whether a pattern tree (forest) P can be embedded in a target tree (forest) T. The time complexity of this algorithm is bounded by $O(|T|\cdot\log D_P)$, where D_P is the depth of P; and its space overhead is bounded by $O(|T| + |P|)$. This computational complexity is better than any existing algorithm for this problem.

1 INTRODUCTION

Let T be a rooted tree. We say that T is *ordered* and *labeled* if each node is assigned a symbol from an alphabet Σ and a left-to-right order among siblings in T is specified. Let v be a node different of root in T with parent node u. Denote by *delete*(T, v) the tree obtained from T by removing the node v. The children of v become children of u as illustrated in Fig. 1.

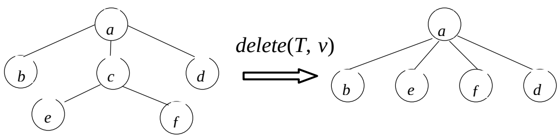

Figure 1. Illustration for node deletion

Given two ordered labeled trees P and T, called the pattern and the target, respectively. We may ask: Can we obtain pattern P by deleting some nodes from target T? That is, is there a sequence $v_1, ..., v_k$ of nodes such that for

$T_0 = T$ and
$T_{i+1} = delete(T_i, v_{i+1})$ for $i = 0, ..., k - 1$,

we have $T_k = P$? If this is the case, we say, P is included in T (H, Mannila and K.-J. Räiha, 1990). Such a problem is called the *tree inclusion problem*.

This problem has been recognized as an important query primitive for XML data and received considerable attention (H. Mannila and K.-J. Räiha, 1990), where a structured document database is considered as a collection of parse trees that represent the structure of the stored texts and the tree inclusion is used as a means of retrieving information from them.

Ordered labeled trees also appear in the natural language processing. As an example, consider querying grammatical structures as illustrated in Fig. 2, which is the parse tree of a natural language sentence (H. Mannila and K.-J. Räiha, 1990).

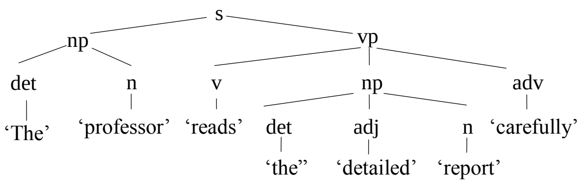

Figure 2.The parse tree of a sentence

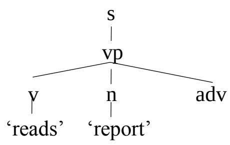

Figure 3. An included tree of the parse tree

One might want to locate, say, those sentences that include a verb phrase containing the verb "reads" and after it a noun "report" followed by any adverb. This is exactly the sentences whose parse tree can be obtained by deleting some nodes from the tree shown in Fig. 2. (See Fig. 3 for illustration.)

A third application of the ordered tree inclusion is the video content-based retrieval. According to (Y. Rui *et al.*, 1999), a video can be successfully decomposed into a hierarchical tree structure, in which each node represents a scene, a group, a shot, a frame, a feature, and so on. Especially, such a tree is an ordered one since the temporal order is very important for video. Some other areas, in which the ordered tree inclusion finds its applications, are the scene analysis, the computational biology (such as RNA structure matching), and the data mining, such as tree mining (M. Zaki, 2002), just to name a few.

In this paper, we discuss an efficient algorithm for this problem.

2 BASIC DEFINITION

We concentrate on labeled trees that are ordered, i.e., the order between siblings is significant. Technically, it is convenient to consider a slight generalization of trees, namely forests. A forest is a finite ordered sequence of disjoint finite trees. A tree T consists of a specially designated node *root*(T) called the root of the tree, and a forest $<T_1, ..., T_k>$, where $k \geq 0$. The trees $T_1, ...,T_k$ are the subtrees of the root of T or the immediate subtrees of tree T, and k is the outdegree of the root of T. A tree with the root t and the subtrees $T_1, ..., T_k$ is denoted by $<t; T_1, ..., T_k>$. The roots of the trees $T_1, ..., T_k$ are the children of t and siblings of each other. Also, we call $T_1, ..., T_k$ the sibling trees of each other. In addition, $T_1, ..., T_{i-1}$ are called the left sibling trees of T_i, and T_{i-1} the immediate left sibling tree of T_i. The root is an ancestor of all the nodes in its subtrees, and the nodes in the subtrees are descendants of the root. The set of descendants of a node v is denoted by *desc*(v). A leaf is a node with an empty set of descendants.

Sometimes we treat a tree T as the forest $<T>$. We may also denote the set of nodes in a forest F by $V(F)$. For example, if we speak of functions from a forest G to a forest F, we mean functions mapping the nodes in $V(G)$ onto the nodes in $V(F)$. The size of a forest F, denoted by $|F|$, is the number of the nodes in F. The restriction of a forest F to a node v with its descendants *desc*(v) is called a subtree of F rooted at v, denoted by $F[v]$.

Let $F = <T_1, ..., T_k>$ be a forest. The preorder of a forest F is the order of the nodes visited during a preorder traversal. A preorder traversal of a forest $<T_1, ..., T_k>$ is as follows. Traverse the trees $T_1, ..., T_k$ in ascending order of the indices in preorder. To traverse a tree in preorder, first visit the root and then traverse the forest of its subtrees in preorder. The postorder is defined similarly, except that in a postorder traversal the root is visited after traversing the forest of its subtrees in postorder. We denote the preorder and postorder numbers of a node v by *pre*(v) and *post*(v), respectively.

Using preorder and postorder numbers, the ancestorship can be easily checked. If there is path from node u to node v, we say, u is an ancestor of v and v is a descendant of u. In this paper, by *ancestor* (*descendant*), we mean a proper ancestor (descendant), i.e., $u \neq v$.

Lemma 1 Let v and u be nodes in a forest F. Then, v is an ancestor of u if and only if $pre(v) < pre(u)$ and $post(u) < post(v)$.

Proof. See Exercise 2.3.2-20 in (D. Knuth, 1969; page 347). □

Similarly, we check the left-to-right ordering as follows.

Lemma 2 Let v and u be nodes in a forest F. v is said to be to the left of u if they are not related by the ancestor-descendant relationship and u follows v when we traverse F in preorder. Then, v is to the left of u if and only if $pre(v) < pre(u)$ and $post(v) < post(u)$.

Proof. The proof is trivial. □

In the following, we use $\prec$ to represent the left-to-right ordering. Also, $v \preceq v'$ iff $v \prec v'$ or $v = v'$. Furthermore, we extend this ordering with two special nodes $\bot \prec v \prec \top$ for any v in F. The *left relatives*, lr(v), of a node $v \in V(F)$ is the set of nodes that are to the left of v and similarly the *right relatives*, rr(v), are the set of nodes that are to the right of v.

The following definition is due to (P. Kilpeläinen and H. Mannila, 1995).

Definition 1 Let F and G be labeled ordered forests. We define an ordered embedding (φ, G, F) as an injective function φ: $V(G) \rightarrow V(F)$ such that for all nodes $v, u \in V(G)$,

i) label(v) = label($\varphi(v)$); (label preservation condition)

ii) v is an ancestor of u iff $\varphi(v)$ is an ancestor of $\varphi(u)$, i.e., iff $pre(\varphi(v)) < pre(\varphi(u))$ and $post(\varphi(u)) < post(\varphi(v))$; (ancestor condition)

iii) v is to the left of u iff $\varphi(v)$ is to the left of $\varphi(u)$, i.e., iff $pre(\varphi(v)) < pre(\varphi(u))$ and $post(\varphi(v)) < post(\varphi(u))$. (Sibling condition) □

If there exists such an injective function from $V(G)$ to $V(F)$, we say, F includes G, F contains G, F covers G, or say, G can be embedded in F.

Fig. 4 shows an example of an ordered inclusion.

Let P and T be two labeled ordered trees. An embedding φ of P in T is said to be *root-preserving* if $\varphi(root(P)) = root(T)$. If there is a root-preserving embedding of P in T, we say that the root of T is an occurrence of P.)

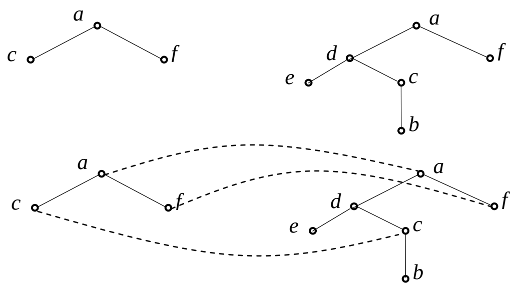

Figure 4: (a) The tree on the left can be included in the tree on the right by deleting the nodes labeled: *d*, *e*, and *b*; (b) the embedding corresponding to (a).

3 ALGORITHM

Now we begin to describe our algorithm. First, we discuss the main idea of the algorithm in 3.1. Then, the formal description of the algorithm is given in 3.2.

3.1 Main Idea

Let $T = <t; T_1, ..., T_k>$ $(k \geq 0)$ be a tree and $G = <P_1, ..., P_q>$ $(q \geq 0)$ be a forest. We handle G as a tree $P = <p_v; P_1, ..., P_q>$, where p_v represents a virtual node, matching any node in T. Note that even though G contains only one single tree it is considered to be a forest. So a virtual root is added. Therefore, each node in G, except the virtual node, has a parent.

Consider a node v in $G = <P_1, ..., P_q>$ with children $v_1, ..., v_j$. We use a pair $<i, v>$ $(i \leq j)$ to represent an ordered forest containing the first i subtrees of v: $<G[v_1], ..., G[v_i]>$. If v is p_v, or a node on the left-most path in P_1, $<i, v>$ is called a *left corner* of G. Especially, $<i, p_v>$ is a left corner, representing the first i subtrees in G: $P_1, ..., P_i$. In addition, we use $\rho(G)$ to represent the left-most leaf node of G. Then, $<i, \rho(G)>$ (with any $i \geq 0$) or $<0, v>$ (with any v in G) stands for an empty left corner. We also use L_G to represent the set of all the left corner in G, including the empty left corner. We also use $\delta(v)$ to represent a link from a node v to the left-most leaf node in $G[v]$, as illustrated in Fig. 5.

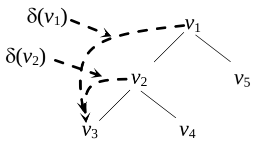

Figure 5. A pattern tree

Let v' be a leaf node in G. $\delta(v')$ is defined to be a link to v' itself. So in Fig. 5, we have $\delta(v_1) = \delta(v_2) = \delta(v_3) = v_3$. Denote by $\delta^{-1}(v')$ a set of nodes $\boldsymbol{x}$ such that for each $v \in \boldsymbol{x}$ $\delta(v) = v'$. Then, in Fig. 5, We have $\delta^{-1}(v_3) = \{v_1, v_2, v_3\}$, $\delta^{-1}(v_4) = \{v_4\}$, and $\delta^{-1}(v_5) = \{v_5\}$.

Let p_1 be the root of P_1. We have $\rho(G) = \delta(p_1)$.

The outdegree of v in a tree is denoted by $d(v)$ while the height of v is denoted by $h(v)$, defined to be the number of edges on the longest downward path from v to a leaf. The height of a leaf node is set to be 0.

As with (Y. Chen and Y.B. Chen, 2006), we arrange two functions to check the tree inclusion. However, in [4], each function returns an integer j, indicating that the first j subtrees in G can be embedded in a target tree or a target forest while in the new algorithm each function returns a left corner in G which can be embedded in the target. Let $\boldsymbol{T}$ and $\boldsymbol{G}$ represent the set of all trees and the set of all forests, respectively. Then, we use $\boldsymbol{L_G}$ to represent all the left corners in all forests in $\boldsymbol{G}$. That is:

$$\boldsymbol{L_G} = \cup_{G \in \boldsymbol{G}} L_G$$

The first function is defined as

$$A: \boldsymbol{T} \times \boldsymbol{G} \rightarrow \boldsymbol{L_G}$$

such that for $T \in \boldsymbol{T}$ and $G \in \boldsymbol{G}$ $A(T, G) = <i, v> \in L_G$ with the following properties:

- If $i > 0$ and $v \neq \rho(G)$, it shows that
  - the first i subtrees of $v \in \delta^{-1}(\rho(G)) \cup \{p_v\}$ can be embedded in T;
  - for any $i' > i$, $<i', v>$ cannot be embedded in T;
  - for any v's ancestor $u \in \delta^{-1}(\rho(G)) \cup \{p_v\}$, there exists no $j > 0$ such that $<j, u>$ is able to be embedded in T.
- If $i = 0$ or $v = \rho(G)$, it indicates that no left corner of G can be embedded in T.

In this sense, we say, $<i, v>$ is the *highest* and *widest* left corner which can be embedded in T.

We notice that if $v = p_v$ and $i > 0$, it shows that $P_1, ..., P_i$ can be included in T.

Similarly, we define the second function as

B: $\boldsymbol{G} \times \boldsymbol{G} \rightarrow L_{\mathbf{G}}$

such that for $G' \in \boldsymbol{G}$ and $G \in \boldsymbol{G}$ $B(G', G) = <i, v> \in L_G$ is the *highest* and *widest* left corner (in G) which can be embedded in G'. Again, if $i = 0$ or $v = \rho(G)$, it shows that no non-empty left corner of G can be embedded in G'.

If the target is a tree and the pattern is a forest, we call A-function. If both the target and pattern are forests, we call B-function. However, during the computation, they will be called from each other.

In the following, we first describe the working process of A-function in great detail. Then, B-function is specified.

- A-function

In $A(T, G)$, we need to handle two cases.

Case 1: $G = <P_1>$; or
$G = <P_1, ..., P_q>$ $(q > 1)$, but $|T| \leq |P_1| + |P_2|$.

In this case, what we can do is to find whether P_1 or a highest and widest left corner $<i, v>$ in P_1 can be embedded in $T = <t; T_1, ..., T_k>$. For this purpose, the following checkings should be conducted:

i) If t is a leaf node, we will check whether label(t) = label($\delta(p_1)$), where p_1 is the root of P_1. If it is the case, return <1, parent of $\delta(p_1)$>. Otherwise, return <0, $\delta(p_1)$>.

(Fig. 6 illustrates this case. Since T contains only a single node, the only left-corner in G, which can possibly be embedded in T is $\delta(p_1)$, represented as <1, parent of $\delta(p_1)$>.)

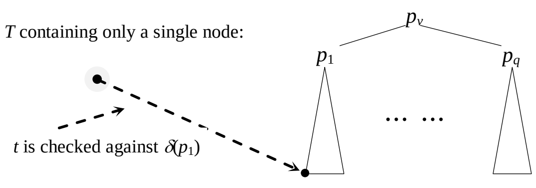

Figure 6.Illustration for the execution of *A*( )

ii) If $|T| > 1$, but $|T| < |P_1|$ and/or $h(t) < h(p_1)$, we will make a recursive call $A(T, <P_{11}, ..., P_{1j}>)$, where $<P_{11}, ..., P_{1j}>$ is a forest of the subtrees of p_1. The return value of $A(T, <P_{11}, ..., P_{1j}>)$ is used as the return value of $A(T, G)$.

(Since $|T| < |P_1|$ and/or $h(t) < h(p_1)$, T is not able to embed the whole P_1. So we will check T against $<P_{11}, ..., P_{1j}>$ to find the highest and widest left-corner within $<P_{11}, ..., P_{1j}>$, which can be embedded in T. See Fig. 7 for illustration.)

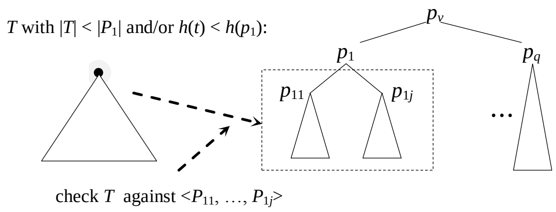

Figure 7.Illustration for a recursive call within *A*( )

iii) If $|T| \geq |P_1|$ and $h(t) \geq h(p_1)$ (but $|T| \leq |P_1| + |P_2|$), we further distinguish between two subcases:

- label(t) = label(p_1). In this case, we will call $B(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>)$.
- label(t) $\neq$ label(p_1). In this case, we will call $B(<T_1, ..., T_k>, <P_1>)$.

In both cases, assume that the return value of B( ) is $<i, v>$. We need to do an extra checking:

- If label(t) = label(v) and $i = d(v)$, the return value of $A(T, G)$ is set to be <1, v's parent>.
- Otherwise, the return value of $A(T, G)$ is the same as $<i, v>$.

Case 2: $G = <P_1, ..., P_q>$ $(q > 1)$, and $|T| > |P_1| + |P_2|$.

In this case, we will call $B(<T_1, ..., T_k>, G)$. Assume that the return value of $B(<T_1, ..., T_k>, G)$ is $<i, v>$. The following checkings will be continually conducted.

iv) If $v \neq p_1$'s parent, check whether label(t) = label(v) and $i = d(v)$. If it is not the case, the return value of $A(T, G)$ is the same as $<i, v>$. Otherwise, the return value of $A(T, G)$ will be set to <1, v's parent>.

v) If $v = p_1$'s parent, the return value of $A(T, G)$ is the same as $<i, v>$. □

- B-function

$B(G', G)$ is designed to handle the case that both G' and G are forests made up of a set of subtrees rooted at nodes that are consecutive siblings in T and P, respectively. Let $G' = <T_1, ..., T_k>$ and $G = <P_1, ..., P_q>$. Denote by t_l the toot of T_l $(l = 1, ..., k)$. Denote by p_j the root of P_j $(j = 1, ...,q)$.In $B(G', G)$, we will make a series of calls $A(T_l, <P_{j_l}, ..., P_q>)$, where l =

1, ..., $x \le k$, $j_1 = 1$, and $j_1 \le j_2 \le ... \le j_x \le q$, controlled as follows.

1. Two index variables l, j are used to scan T_1, ..., T_k and P_1, ..., P_q, respectively. (Initially, l is set to 1, and j is set to 0.) They also indicate that <P_1, ..., P_j> has been successfully embedded in <T_1, ..., T_l>.
2. Let <i_l, v_l> be the return value of $A(T_l$, <P_{j+1}, ..., P_q>). If $v_l = p_{j+1}$'s parent, set j to be $j + i_l$. Otherwise, j is not changed. Set l to be $l + 1$. Go to (2).
3. The loop terminates when all T_l's or all P_j's are examined.

   (Fig. 8 helps for illustration of this iteration process.)

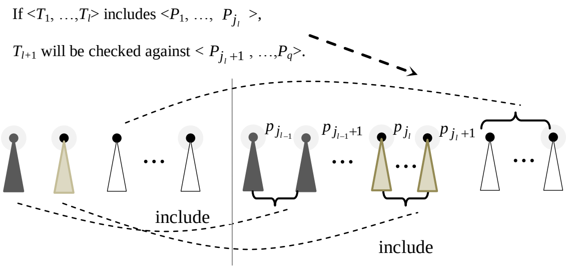

Figure 8.Illustration for an execution of *B*( )

4. If $j > 0$ when the loop terminates, $B(G', G)$ returns <j, p_1's parent>, indicating that G' contains P_1, ..., P_j. Otherwise, $j = 0$, indicating that even P_1 alone cannot be embedded in any T_l ($l \in \{1, ..., k\}$). However, in this case, we need to continue searching for a highest and widest left corner <i, v> in P_1, which can be embedded in G'. This can be done as follows.

   i) Let <i_1, v_1>, ..., <i_k, v_k> be the return values of $A(T_1$, <P_1, ..., P_q>), ..., $A(T_k$, <P_1, ..., P_q>), respectively. Since $j = 0$, each $v_l \in \delta^{-1}(\rho(G))$ ($l = 1, ..., k$).

   ii)If each $i_l = 0$, return <0, $\rho(G)$>. Otherwise, there must be some v_l's with $i_l > 0$. We call such a node a *non-zero point*. Find the first non-zero point v_f with children w_1, ...,w_s such that v_f is not a descendant of any other non-zero point. Then, we will check <T_{f+1}, ..., T_k> against <$P[w_{i_f+1}]$, ..., $P[w_s]$>. Let y be a number such that <$P[w_{i_f+1}]$, ..., $P[w_{i_f+y}]$> can be embedded in <T_{f+1}, ..., T_k>. The return value of $B(T', G)$ should be set to <$i_f + y$, v_f>. □

In the above process, (1), (2) and (3) together are referred to as a *main checking* while (4) alone as a *supplement checking*.

We notice that in the main checking much useless work is conducted since in the case $j = 0$ only one non-zero point <i_f, v_f> is utilized in the subsequent supplement checking while all the other left corners are not used at all. Thus, the effort for looking for such return values brings the void.

For this reason, we slightly change the definitions of both A-function and B-function to let them take a third input parameter which is a node $u \in V(G)$, used to transfer an important message: once we have detected that only a left corner *lower* than u can be produced by the corresponding computation, it should stop immediately (since such a return value will not be used.) We say, a left corner <i, v> is lower than u if $v = u$ or v is a descendant of u. u is then called a *controlling point*. In $A(T, G, u)$, this checking can be made at the very beginning by checking whether p_1's parent is an ancestor of u. If it is not, the left corner to be returned must be lower than u and the computation of the corresponding A-function should not be carried out. In $B(G', G, u)$, u is mainly used to avoid any useless supplement checking (to be discussed in 3.2).

Let $V(\boldsymbol{G}) = \cup_{G \in \boldsymbol{G}} V(G)$.

Our functions are redefined as follows:

$$A: \boldsymbol{T} \times \boldsymbol{G} \times V(\boldsymbol{G}) \to L_{\boldsymbol{G}}$$

such that for $T \in \boldsymbol{T}$, $G \in \boldsymbol{G}$ and $u \in V(G)$ $A(T, G, u)$ = <i, v> $\in L_G$ is the *highest* and *widest* left corner (in G) embeddable in T if it not lower than u. Otherwise, $A(T, G, u)$ returns an empty left corner.

$$B: \boldsymbol{G} \times \boldsymbol{G} \times V(\boldsymbol{G}) \to L_{\boldsymbol{G}}$$

such that for $G' \in \boldsymbol{G}$, $G \in \boldsymbol{G}$ and $u \in V(G)$ $B(G', G, u)$ = <i, v> $\in L_G$ is the *highest* and *widest* left corner (in G) embeddable in G' if it not lower than u. Otherwise, $B(T, G, u)$ returns an empty left corner.

Initially, u is set to be $\rho(G)$ for both functions.

Elaboration on the controlling points leads to an almost linear time algorithm.

3.2 Algorithm Description

In this subsection, we give the formal description of our algorithm.

- *A*-function

```
function A(T, G, u) (*Initially, u = ρ(G).*)
input: T = <t; T_1, ..., T_k>, G = <P_1, ..., P_q>.
output: <i, v> specified above.
begin
1. if p_1's parent is not an ancestor of u then return <0, ρ(G)>;
2. if (q = 1 or |T| ≤ |P_1| + |P_2|)
3. then
   {let P_1 = <p_1; P_11, ..., P_1j>;                    (*Case 1*)
4.   if t is a leaf then
     {  let δ(p_1) = v;                                  (*Case 1 - (i)*)
5.      if label(t) = label(v)  then return <1, v's parent>
6.                              else return <0, v>;
7.   }
8.   if (|T| < |P_1| ∨ h(t) < h(p_1)) then return A(T, <P_11, ..., P_1j>, u);
                                                         (*Case 1 - (ii)*)
9.    if label(t) = label(p_1)                           (*Case 1- (iii)*)
10.   then {if p_1 is a leaf then {v := p_1's parent; i := 1;}
11.           else {  if p_1 = u
                      then <i, v>:= B(<T_1, ..., T_k>, <P_11, ..., P_1j>, p_11)
12.                   else <i, v>:= B(<T_1, ..., T_k>, <P_11, ..., P_1j>, u);
13.                   if label(t) = label(v) and i = d(v)
14.                   then {v := v's parent; i := 1; }
15.                 }
16.        }
17.   else <i, v> := B(<T_1, ..., T_k>, <P_1>, u);
                               (*If label(t) ≠ label(p_1), call B( ).*)
18.   return <i, v>;
19. }
20. else{ if label(t) = label(u)                         (*Case 2*)
21.       then <i, v> := B(<T_1, ..., T_k>, G, u's first child);
22.       else <i, v> := B(<T_1, ..., T_k>, G, u)
23.       if v ≠ p_1's parent                            (*Case 2 - (iv)*)
24.       then {  if (label(t) = label(v)) ∧ i = d(v)
25.               then return <1, v's parent>;
26.            }
27.       return <i, v>;                                 (*Case 2 - (v)*)
28. }
end
```

The above algorithm can be viewed as composed of three parts: line 1, lines 2 - 19, and lines 20 - 28. In line 1, we only check whether p_1's parent is an ancestor of u. If not, return <0, ρ(G)>. Otherwise, we go to the second part, in which we first check whether $q = 1$ or $|T| \leq |P_1| + |P_2|$ (see line 2). If it is the case, we have *Case* 1 and then lines 3 - 19 are executed. In this process, all the three subcases (i), (ii), and (iii) are checked. If $q > 1$ and $|T| > |P_1| + |P_2|$, we go to the third part. That is, lines 20 - 28 will be carried out, in which we handle *Case* 2. This is done by calling $B(<T_1, ..., T_k>, G, u)$. Depending on its return value, subcase (iv) or (v) will be conducted.

Special attention should be paid to line 8, 11, 12, 17, 21, and 22 to see how a controlling point is propagated by a recursive call. For this, we also distinguish among five cases, i.e., *Case* 1 – (i), *Case* 1 – (ii), *Case* 1 – (iii), *Case* 2 – (iv), and *Case* 2 – (v).

In *Case* 1 – (i), no recursive call is conducted and thus the cut u is not transferred.

In *Case* 1 – (ii), we will call $A(T, <P_{11}, ..., P_{1j}>, u)$, by which the controlling point u is directly transferred to the recursive call since its return value will be used as the return value of $A(T, G, u)$. (See line 8.)

In *Case* 1 – (iii), we will call the B-function to check $<T_1, ..., T_k>$ against $<P_{11}, ..., P_{1j}>$ or against $<P_1>$, depending on whether label(t) = label(p_1) or label(t) ≠ label(p_1). Concerning the controlling point transfer, we need to consider three cases:

- label(t) = label(p_1) and $p_1 = u$. In this case, we will call $B(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, p_{11})$ with the controlling point being set to be p_{11}. It is because in this case the main checking of the B-function execution may reveal that $<T_1, ..., T_k>$ is able to embed the whole $<P_{11}, ..., P_{1j}>$. In the case, the return value of $A(T, G, u)$ will be set to <1, p_1's parent>, higher than u. So it is a useful computation; and downgrading the controlling point from $u = p_1$ to p_{11} will let it go through. On the other hand, p_{11} will effectively prohibit any possible supplement checking in this B-function execution since such a checking can only bring out a left corner lower than p_{11} and will not be used. (See line 11.)
- label(t) = label(p_1) and $p_1 \leadsto u$. In this case, we will call $B(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, u)$, by which u is directly transferred since we must have $p_{11} \underset{-}{\leadsto} u$ and no useful computation can be eliminated by the controlling point u. (See line 12.)
- label(t) ≠ label(p_1). In this case, we will call $B(<T_1, ..., T_k>, <P_1>, u)$, by which u is directly transferred for the same reason as *Case* 1 – (ii). (See line 17.)

In *Case* 2, we will call $B(<T_1, ..., T_k>, G, x)$, where x is u or u's first child, depending on whether label(t) ≠ label(p_1) or label(t) = label(p_1).

- If label(t) = label(p_1), the controlling point for this recursive call should be set to u's first child. It is because $<T_1, ..., T_k>$ may not be able to cover P_1, but all the subtrees each rooted as a child of u. In this case, the whole T embeds $G[u]$ and the return value of $A(T, G, u)$ should be set to <1, u's parent>, higher than u. So, setting the controlling point to u's first child will keep this computation not skipped over. (See line 21.)
- If label(t) ≠ label(p_1), the controlling point u will be directly transferred (i.e., $x = u$) since in this case, only the left corner (returned by

$B(<T_1, ..., T_k>, G, u)$) higher than u will be used. (See line 22.)

Accordingly, *Case* 2 – (iv) is handled in lines 24 - 25, while *Case* 2 – (v) in line 27.

- *B*-function

In $B(G', G, u)$, the treatment of controlling points is more complicated than in Algorithm *A*():

1. Let $G' = <T_1, ..., T_k>$ and $G = <P_1, ..., P_q>$. At the very beginning, we need to check whether $u = p_1$, where p_1 is the root of P_1. If it is the case, only the main checking needs to be conducted. (The supplement checking can only deliver a left corner lower than p_1 and therefore should not be carried out.)

2. In the main checking, a series of calls of *A*-functions will be carried out. During this process, the controlling point for each *A*-function call needs to be dynamically changed as described below.

   - Let $<i_l, v_l>$ be the return value of $A(T_l, <p_{j_l}, ..., P_q>, u_l)$ for $l = 1, ..., x \le k$, where $j_1 = 1$, $j_1 \le j_2 \le ... \le j_x \le q$, and $u_1 = u$. In addition, for $2 \le l \le x$, u_l is determined as follows:

   - Let s be an integer such that any of $T_1, ..., T_s$ is not able to embed P_1, but T_{s+1} embeds $<P_1, ..., P_j>$ for some $j > 0$. Then, for $2 \le l \le s$, we have

$$u_l = \begin{cases} v_{l\text{-}1}, \text{ if } v_{l\text{-}1} \text{ is an ancestor of } u_{l\text{-}1} \text{ and } i_{l\text{-}1} > 0; \\ u_{l\text{-}1}, \text{ if } v_{l\text{-}1} \text{ is not an ancestor of } u_{l\text{-}1} \text{ or } i_{l\text{-}1} = 0; \end{cases} \quad (3.1)$$

and for $s + 1 \le l \le k$, we have

$$u_l = p_{j_l}. \quad (3.2)$$

The formula (3.1) shows how the controlling points are changed before we find the first subtree in *T* which is able to embed some subtrees in *G*. After such a subtree is found, the controlling points are determined in terms of the formula (3.2). It is because for each subsequent *A*-function call to check a T_l against $<p_{j_l}, ...,P_q>$, a returned left corner lower than p_{j_l} will not be used in the continuing computation.

If $s < k$, it shows that $<T_1, ..., T_k>$ includes $<P_1, ..., P_m>$ for some m ($1 \le m \le q$), and the supplement checking will not be conducted. If $s = k$, $<T_1, ..., T_k>$ does not include any subtree in *G*, but some T_l's each may include a non-empty left corner in P_1. Assume that we can find a subtree T_f such that it embeds a left corner $<i_f, v_f>$ in P_1 with the following properties: i) $i_f > 0$, ii) v_f is not a descendant of any other non-zero point, and iii) v_f is also an ancestor of u. Then, a supplement checking will be performed as described in 3.1. Otherwise, no supplement checking is needed.

In terms of the above discussion, we design two subfunctions of the *B*-function: *B-without-s*(G', G), in which no supplement checking will be carried out; and *B-with-s*(G', G, u), in which a supplement checking may be invoked. Then, during the execution of $B(G', G, u)$, if $u = p_1$, call *B-without-s*(G', G); otherwise, call *B-without-s*(G', G, u).

```
function B(G', G, u) (*Initially, u = ρ(G).*)
input: G' = <T1, ..., Tk>, G = <P1, ..., Pq>
output: <i, v> specified above.
begin
1.  if u = p1 then return B-without-s(G', G)
2.  else return B-with-s(G', G, u);
end
function B-without-s(G', G)
begin
1.  l := 1; j := 0;
2.  while ( j < q and l ≤ k) do
3.  {   <il, vl> := A(Tl, <Pj+1, ..., Pq>, pj+1);
4.      if vl = p1's parent and il > 0) then j := j + il;
5.  }
6.  return <j, p1's parent>
end
function B-with-s(G', G, u)
begin
1.  l := 1; j := 0; v := u; f := 0;
2.  while (j < q and l ≤ k) do                (*main checking*)
3.  {   <il, vl> := A(Tl, <Pj+1, ..., Pq>, v)
4.      if (vl = p1's parent and il > 0) then {j := j + il; v := pj;}
5.      else if (vl is an ancestor of v and il > 0)
            then {v := vl; f := l;}
6.      l:= l + 1;
7.  }
8.  if j > 0 then return <j, p1's parent>;
9.  if f = 0 then return <0, δ(p1)>;
10. let w1, ..., ws be the children of vf;      (*supplement
checking*)
11. l := f + 1; j := if;
12. while (j < s and l ≤ k) do
13. {   <il, vl> := A(Tl, <G[wj+1], ..., G[ws]>, wj+1);
14.     if (vl = vf and il > 0) then j := j + il;
15.     l:= l + 1;
16. }
17. return <j, vf>;
end
```

In $B(G', G, u)$, we first check whether $u = p_1$. If it is the case, we will call *B-without-s*(G', G) (see line 1), in which, by a series of *A*-function calls, we try to find a largest j such that $<T_1, ..., T_k>$ is able to embed $<P_1, ..., P_j>$, but not able to embed $<P_1, ..., P_j, P_{j+1}>$. In this process, for each *A*-function call of the form $A(T_l, <P_{j_l}, ..., P_q>, u_l)$, u_l is set to be p_{j_l}, the root of P_{j_l} (see line 3 in *B-without-s*( )). No supplement checking will be conducted since the left corner produced by a supplement checking must be lower than p_{j_l}.

If $u \neq p_1$, we will call *B-with-s*(G', G, u) (see line 2 in B( )), in which we have two *while*-loops: one from line 2 to 7 and the other from line 12 to 16. In the first *while*-loop, we do the main checking to find the largest j such that <T_1, ..., T_k> embeds <P_1, ..., P_j>. In this process, by each A-function call, the corresponding controlling point will be set according to the formulas (3.1) and (3.2). In the second *while*-loop, the *supplement checking* will be carried out. However, this is done only when the following two conditions are satisfied:

(1) $j = 0$, and
(2) There exists at least a non-zero point <i_f, v_f> such that v_f is not lower than u.

In *B-with-s*(G', G, u), we use a variable f to record the first of such non-zero points, which is not a descendant of any other non-zero point. Initially, f is set 0. Therefore, if (2) is not satisfied, we must have $f = 0$ after the main checking is completed. So only when $j = 0$ and $f > 0$, the supplement checking will be conducted (See lines 8 and 9.)

We also notice that in the supplement checking, for each A-function call of the form $A(T_l$, <$G[w_{j+1}]$, ..., $G[w_s]$>, $w_{j+1})$, the controlling point is set to be w_{j+1} to prohibit a further supplement checking since this can only return a useless left corner lower than w_{j+1}.

4 CONCLUSIONS

In this paper, a new algorithm is proposed to solve the ordered tree inclusion problem. Up to now, the best algorithm for this problem needs quadratic time. However, ours requires only $O(|T|\cdot\log D_P)$ time and $O(|T| + |P|)$ space, where T and P are a target and a pattern tree (forest), respectively; and D_P is the depth of P. The critical concept of our algorithm is the *left corner*, which enables us to develop a deep insight into the tree inclusion problem and extend it to a more general one to return a left corner as a result. In practice, the general problem seems to be more useful than the original one since if P cannot be embedded in T, we may want to know whether any part of P can be embedded in T.

REFERENCES

L. Alonso and R. Schott. On the tree inclusion problem. In *Proceedings of Mathematical Foundations of Computer Science*, pages 211-221, 1993.

W. Chen. More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, 26:370-385, 1998.

Y. Chen and Y.B. Chen, A New Tree Inclusion Algorithm, *Information Processing Letters* 98(2006) 253-262, Elsevier Science B.V.

Y. Chen, A New Algorithm for Twig Pattern Matching, in: *Proc. of Int. Conf. on Enterprise Information Systems (ICEIS'2007)*, IEEE, Funchal-madeira, Portugal, June 2007, pp. 44-51.

Y. Chen and Y.B. Chen, Subtree Reconstruction, Query Node Intervals and Tree Pattern Query Evaluation, *Journal of Information Science and Engineering* 28, 263-293 (2012).

Y. Chen and Y.B. Chen, A Linear-Space Top-down Algorithm for Tree Inclusion Problem, in: *Proc. 2nd Int. Conf. on Computer Science and Service System (CSSS2012)*, April 2012, *Nanjing, China,* April 2012, IEEE, pp. 2127-2131.

Y. Chen and Y.B. Chen, On the Tree Inclusion Problem, *2013 Int. Conf. on Computer, Networks and Communication Engineering (ICCNCE 2013)*, Beijing, China, May 23-24, 2013.

Y. Chen and L. Zou, Unordered tree matching and ordered tree matching: the evaluation of tree pattern queries, Int. *J. Information Technology, Communications and Convergence*, Vol. 1, No. 3, 2011, pp. 254-279.

Y. Chen and Y.B. Chen, A New Tree Inclusion Algorithm, *Information Processing Letters* 98(2006) 253-262, Elsevier Science B.V.

H.L Cheng and B.F Wang, On Chen and Chen's new tree inclusion algorithm, *Information Processing Letters*, 2007, Vol. 103, 14-18, Elsevier Science B.V.

P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput*, 24:340-356, 1995.

D.E. Knuth, *The Art of Computer Programming, Vol. 1 (1st edition)*, Addison-Wesley, Reading, MA, 1969.

H. Mannila and K.-J. Räiha, On Query Languages for the p-string data model, in "Information Modelling and Knowledge Bases" (H. Kangassalo, S. Ohsuga, and H. Jaakola, Eds.), pp. 469-482, IOS Press, Amsterdam, 1990.

T. Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings* of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in *Lecture Notes of Computer Science (LNCS), volume 126*4, pages 150-166. Springer, 1997.

Y. Rui, T.S. Huang, and S. Mehrotra, Constructing table-of-content for videos, *ACM Multimedia Systems Journal, Special Issue Multimedia Systems on Video Libraries*, 7(5):359-368, Sept 1999.

M. Zaki, Efficiently mining frequent trees in a forest. In *Proc. of KDD*, 2002.