On the Tree Inclusion Problem

Yangjun Chen\textsuperscript{1}, Yibin Chen\textsuperscript{2}
Dept. Applied Computer Science, University of Winnipeg, Canada
\textsuperscript{1}y.chen@uwinnipeg.ca, \textsuperscript{2}chenyibin@gmail.com

Abstract - The ordered tree inclusion is an interesting problem, by which we will check whether a pattern tree \( P \) can be included in a target tree \( T \), in which the order of siblings is significant. In this paper, we propose an efficient algorithm for this problem. Its time complexity is almost linear in the size of \( T \) and \( P \). Up to now the best algorithm for this problem needs quadratic time.

Keywords: tree inclusion; ordered labeled trees; tree matching.

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 \( \Sigma \) and a left-to-right order among siblings in \( T \) is specified.

A tree \( T \) consisting of a specially designated node \( \text{root}(T) = t \) (called the root of the tree) and a forest \( <T_1, \ldots, T_k> \) (where \( k \geq 0 \)) is denoted as \( <t; T_1, \ldots, T_k> \). We also call \( T_j (1 \leq j \leq k) \) a direct subtree of \( t \).

The preorder of a forest \( F = < T_1, \ldots, T_k > \) is the order of the nodes visited during a preorder traversal. A preorder traversal of a forest \( < T_1, \ldots, T_k > \) is as follows. Traverse the trees \( T_1, \ldots, T_k \) 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 \( \text{pre}(v) \) and \( \text{post}(v) \), respectively.

Let \( u, v \) be two nodes in \( T \). 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., \( u \neq v \). Using the preorder and postorder numbers, the ancestorship can be easily checked:

\( u \) is an ancestor of \( v \) if and only if \( \text{pre}(u) < \text{pre}(v) \) and \( \text{post}(u) < \text{post}(v) \). (See Exercise 2.3.2-20 in [5], page 347.)

Similarly, \( u \) is said to be to the left of \( v \) if they are not related by the ancestor-descendant relationship and \( v \) follows \( u \) when we traverse \( T \) in preorder. Then, \( u \) is to the left of \( v \) if and only if \( \text{pre}(u) < \text{pre}(v) \) and \( \text{post}(u) < \text{post}(v) \).

In the following, we use \( < \) to represent the left-to-right ordering. Also, \( v < v' \) if \( v \) is to the left of \( v' \) or \( v = v' \). We will also use \( V(T) \) and \( E(T) \) to represent the set of nodes and the set of edges in \( F \), respectively.

The following definition is due to [4].

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

i) \( \text{label}(v) = \text{label}(\varphi(v)) \); (label preservation condition)

ii) \( v \) is an ancestor of \( u \) if \( \varphi(v) \) is an ancestor of \( \varphi(u) \), i.e., \( \text{pre}(v) < \text{pre}(u) \) and \( \text{post}(u) < \text{post}(v) \) if \( \varphi(v) < \text{pre}(\varphi(u)) \) and \( \text{post}(\varphi(u)) \) (ancestor condition)

iii) \( v \) is to the left of \( u \) if \( \varphi(v) \) is to the left of \( \varphi(u) \), i.e., \( \text{pre}(v) < \text{pre}(u) \) and \( \text{post}(v) < \text{post}(u) \) if \( \varphi(v) < \text{pre}(\varphi(u)) \) and \( \text{post}(\varphi(u)) < \text{post}(\varphi(u)) \). (Sibling condition)

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

**Fig. 1** shows an example of an ordered inclusion.

![Figure 1](image)

**Figure 1:** (a) The tree \( P \) on the left can be included in the tree \( T \) on the right; (b) an embedding of \( P \) in \( T \).

Throughout the rest of the paper, we refer to the labeled ordered trees simply as trees.

The ordered tree inclusion problem was initially introduced by Knuth [5], where only a sufficient condition for this problem is given. Its first polynomial time algorithm was presented by Kilpeläinen and Mannila [4] with \( O(|T|\cdot|P|) \) time and space being used. Most of the later improvements are refinements of this algorithm.

Recently, a break-through is achieved by Bille and Gørtz [1]. They got a space-economical algorithm with its space overhead bounded by \( O(|T| + |P|) \), but with its time complexity bounded by

\[
\min \begin{cases} 
O(|T|\cdot|\text{leaves}(P)|) \\
O(|\text{leaves}(T)|\cdot|\text{leaves}(P)|\cdot|\log|\text{leaves}(P)||) + \\
O(|T|\cdot|P|/(\log|T|)) 
\end{cases}
\]

where \( D_T \) (resp. \( D_P \)) is the depth of \( T \) (resp. \( P \)), and \( \text{leaves}(T) \) (resp. \( \text{leaves}(P) \)) stands for the set of the leaves of \( T \) (resp. \( P \)).

In [3], a top-down algorithm was first proposed. Its space requirement is also bounded by \( O(|T| + |P|) \). However, its time complexity is not polynomial, as shown in [6].

In this paper, we present a new algorithm for this problem. Its space overhead remains \( O(|T| + |P|) \), but its time complexity is...
If both the target and the pattern are forests, we call a function \( A(<T_1, ..., T_k>, <P_1, ..., P_q>) \). If the target is a tree and the pattern is a forest, we call another function \( B(T, <P_1, ..., P_q>) \). But during the computation, they will be called from each other.

Let \( <i, v> \) be a left corner returned by \( A(<T_1, ..., T_k>, <P_1, ..., P_q>) \) (or by \( B(T, <P_1, ..., P_q>) \)). Then, the following properties are satisfied:

- If \( i > 0 \) and \( v \neq \rho(G) \), it shows that
  - the first \( i \) subtrees of \( v \) in \( \delta^{-1}(\rho(G)) \) can be embedded in \( <T_1, ..., T_k> \) (or in \( T \));
  - for any \( i' > i \) and \( <i', v> \) cannot be embedded in \( <T_1, ..., T_k> \) (or in \( T \)); and
  - for any \( v \)'s ancestor \( u \) in \( \delta^{-1}(\rho(G)) \), there exists no \( j > 0 \) such that \( <j, u> \) is able to be embedded in \( <T_1, ..., T_k> \) (or in \( T \)).

- If \( i = 0 \) or \( v = \rho(G) \), it indicates that no left corner of \( G \) can be embedded in \( <T_1, ..., T_k> \) (or in \( T \)).

In this sense, we say, \(<i, v>\) is the highest and widest left corner which can be embedded in \( <T_1, ..., T_k> \) (or in \( T \)).

We notice that if \( v = p_i \) and \( i > 0 \), it shows that \( P_1, ..., P_P \) can be included in \( <T_1, ..., T_k> \) (or in \( T \)).

Finally, we say, a left corner \(<i, v>\) is higher than a node \( u \) (or another left corner \(<j, u>\) if \( v \) is an ancestor of \( u \)).

### 2.2 \( A \)-function

First, \( A \)’s have a look at a naïve way to evaluate \( A(F, G) \), where \( F = <T_1, ..., T_k> \) and \( G = <P_1, ..., P_q> \).

Two index variables \( j, l \) are used to scan \( T_1, ..., T_k \) and \( P_1, ..., P_q \), respectively. Initially, \( j \) is set to 1, and \( l \) is set to 0. (They also indicate that \( <P_1, ..., P_q> \) has been successfully embedded in \( <T_1, ..., T_k> \)). In each step, we call \( B(T_j, <P_{j+1}, ..., P_q>) \).

Let \( <i, v> \) be the return value of \( B(T_j, <P_{i+1}, ..., P_q>) \). If \( v = \text{parent}(p_i) \), set \( j \) to \( j+1 \). Otherwise, let \( l \) be unchanged. Set \( j \) to \( j+1 \). Go to (2).

3. The loop terminates when all \( T_j \)'s or all \( P_j \)'s are examined.

If \( i > 0 \) when the loop terminates, \( B(T, G) \) returns \( <l, \text{parent}(p_i)> \), indicating that \( F \) contains \( P_1, ..., P_l \).

Otherwise, \( l = 0 \), indicating that even \( P_1 \) alone cannot be embedded in any \( T_j \) (\( j \in \{1, ..., k\} \)). However, in this case, we need to continue to look for a highest and widest left corner \(<i, v>\) in \( G \), which can be embedded in \( G \). This is done as described below.

4. Let \( <i_1, v_1>, ..., <i_k, v_k> \) be the return values of \( B(T_i, <P_1, ..., P_q>) \), respectively. Since \( i = 0 \), each \( v_j \) in \( \delta^{-1}(v') \) (\( j = 1, ..., k \)), where \( v' \) is the left-most leaf in \( P_1 \).

5. If each \( i_j = 0 \), return \( <0, \rho(G)> \). Otherwise, there must be some \( v_j \)'s such that \( i_j > 0 \). We call such a node a non-zero point. Find the first non-zero point \( v_j \) with children \( w_j, ..., w_j \) such that \( v_j \) is not a descendant of any other non-zero point. Then, we will check \( <T_{i+k}, ..., T_k> \) against \( G[w_j, ..., w_j] \).

Let \( x \) (\( 0 \leq x \leq y - i \)) be a number such that \( G[w_j, ..., w_j] \) can be embedded.
in \( <T_{p_1}, ..., T_t > \). Thereturn value of \( A(F, G) \) should be set to
\( <i_1^* \times x, y^* > \).

In the above process, (1) - (3) are referred to as a main checking while (4) - (5) as a supplement checking.

We notice that in the supplement checking, only the first non-zero is utilized for forming the final result while all the other calls of the form \( B(T_{q}, <P_1, ..., P_{q'}>) \) (done in the main checking) are not used at all. Their efforts for looking for the corresponding return values bring the void, and therefore should be avoided.

For this purpose, we introduce the concept of cuts to integrate a kind of control into the above working process.

**Definition 2** A cut for a call of the form \( A(F, G) \) is a node \( u \neq p_j \) \( \in \delta^1(\rho(G)) \), indicating that if the suplement checking in \( A(F, G) \) can only bring out a left corner \( <i, v> \) not higher than \( u \), the corresponding computation makes no contribution to the final result.

The following example helps for illustration.

**Example 1** Consider target forest \( F \) and pattern forest \( G \) shown in Figure 3, in which each node in \( F \) is identified with \( t_i \) such as \( t_1, t_2, t_11 \), and so on; and each node in \( G \) is identified with \( p_j \). Besides, each subtree rooted at \( t_i \) (resp. \( p_j \)) is represented by \( T_i \) (resp. \( P_j \)).

\[
\begin{align*}
F &= <T_1, T_2>: \\
G &= <P_1, P_2>: \\
\end{align*}
\]

![Figure 3. A target and a pattern forest](image)

Initially, for \( A(<T_1, T_2>, <P_1, P_2>) \), we will set its cut \( u_0 \) to be \( \rho(G) = p_{111} \), imposing in fact no control on its supplement checking at all. When executing \( B(T_i, <P_1, P_2>) \), its cut \( u_i \) is set to be the same as \( u_0 \) (i.e., \( u_1 = u_0 \)).

It can be seen that \( T_1 \) is able to include only \( G[p_{11}] \). So the return value of this call should be \( <1, p_1> \). Then, for \( B(T_2, <P_1, P_2>) \), the cut \( u_2 \) should be set \( p_{1} \), indicating that the supplement checking within \( B(T_2, <P_1, P_2>) \) should be cut off if it can only produce a left corner not higher than \( p_1 \) since the result will not be used.

As will be seen later, during the execution of \( B(T_2, <P_1, P_2>) \), \( A(<T_2, T_2>, <P_1, P_2>) \) will be called and the cut \( u_{21} \) for this call is the same as \( p_{112} \). Then, after the main checking of \( A(<T_2, T_2>, <P_1, P_2>) \), its supplement checking will be discarded since in its main checking the return values of \( B(T_1, <P_1, P_2>) \) and \( B(T_2, <P_1, P_2>) \) are \( <1, p_1> \) and \( <0, \rho(G)> \), respectively; and the supplement checking will not create a left corner higher \( p_1 \).

With the cuts being considered, the \( A \)-function should be changed to take three inputs: \( F=<T_1, ..., T_t>, G=<P_1, ..., P_{q'}>, u \in \delta^1(p_j) \). (Initially, \( u \) is set to \( \rho(G) \).) In the main checking of \( A(F, G, u) \), the cut for each \( B \)-function call will be dynamically changed as described below.

i) At the very beginning, we will check whether \( u \) is higher than \( p_1 \), where \( p_1 \) is the root of \( P_1 \). If it is the case, we simply return \( <0, \rho(G)> \) since the computation will not make any contribution to the final result. Otherwise, we will do the following.

ii) For the first \( B \)-function call \( B(T_1, <P_1, ..., P_{q'}>, u_1) \) (where \( l_1 = 1 \), \( u_1 = u \). Let \( <i_1, v_1> \) be its return value. We will call \( B(T_2, ..., P_{q'}, u_2) \) in a next step. If \( v_1 = \text{parent}(p_1), l_2 = l_1 + 1 \) and \( u_2 \) is set to be \( p_{12} \). If \( v_1 \neq \text{parent}(p_1), l_2 = l_1 + 1 \) and \( u_2 \) is set to \( v_1 \).

iii) In general, let \( <i_j, v_j> \) be the return value of \( B(T_j, <P_{j}, ..., P_{q'}, u_j>) \) for \( j = 1, ..., x \leq k, j_1 = 1, j_1 \leq j_2 \leq ... \leq j_s \leq q \). Let \( s \) be an integer such that \( l_1 = ... = l_s = 1 \), but \( u_{j+1} > 1 \). Then, for \( 2 \leq j \leq s \), we have

\[
u_j = \begin{cases} v_{j+1}, & \text{if} \ u_{j+1} \text{is higher than} u_{j+1}, \text{and} \ i_{j+1} \geq 0; \\
u_{j+1}, & \text{if} \ u_{j+1} \text{is not higher than} u_{j+1} \text{or} \ i_{j+1} = 0; \
\end{cases}
\]

and for \( s + 1 \leq j \leq x \), we have

\[
u_j = p_{ij}.
\]

The formula (2.1) shows how the cuts are changed before we find the first \( T \)-which is able to embed some subtrees in \( G \). After \( T \), the cuts are determined in terms of the formula (2.2). Setting \( u_j \) to be \( p_{ij} \) will effectively prohibit the supplement checking in the execution of \( B(T_j, <P_{ij}, ..., P_{q'}>, u_j) \), which will definitely return a left corner not higher than \( p_{ij} \) and therefore is useless.

After the main checking, the following checks will be conducted to determine whether a supplement checking will be carried out.

- If \( l = q \), we will record the embedding.
- If \( j < k \), we will continue to find a next embedding by making a recursive call \( A(<T_{j+1}, ..., T_t>, <P_1, ..., P_{q'}>, p_k>) \).
- If there is at least an embedding, return \( <q, p_2> \).
- If \( 0 < l < q \), return \( <l, p_2> \).
- If \( <l, p_2> = 0 \) and \( f = 0 \), return \( <0, \delta(p_1)> \).
- Otherwise, a supplement checking will be conducted.

Let \( <i_j, v_j> \) be the return value of some \( B(T_j, <P_{j}, ..., P_{q'}>, u_j) \) such that \( v_j \) is not a descendant of any other non-zero point. We will make a recursive call \( A(<T_{j+1}, ..., T_t>, <G[w_{j+1}], ..., G[w_q]> , w_{j+1} = 1) \) for doing a supplement checking, where \( w_1, ..., w_q \) are the children of \( v_j \). We notice that the cut for this recursive call is set to be \( w_{j+1} \) to cut off a possible supplement checking in this execution.

In terms of the above discussion, we give the following algorithm.

**function** \( A(F, G, u) \) (Initially, \( u = \rho(G) \))

**input:** \( F = <T_1, ..., T_t>, G = <P_1, ..., P_{q'}>, u - a \) cut.
output: \(<i, v>\) specified above.

\begin{enumerate}
  \item if \(p_1\) is a descendant of \(u\) then return \(<0, \delta(p_1)>\);
  \item \(j:=1; l:=0; v:=u; f:=0; i:=0; \)
  \item while \((i < q \text{ and } j \leq k)\) do (*main checking*)
    \begin{enumerate}
      \item \(<j, v>:=B(T_j, <P_{j1}, ..., P_{j}>), v>\)
      \item if \((v_j = p_j)\) then \(l:=i+j; v:=p_{ji};\)
      \item else if \((v_j \text{ is an ancestor of } v \text{ and } j>0)\)
        \begin{enumerate}
          \item \(v:=v_j; i:=j; f:=f_j;\)
          \item \(j:=j+1;\)
        \end{enumerate}
      \item if \(q = \emptyset\) then record the embedding;
    \end{enumerate}
  \item if \(<k \text{ then } \{<i, v>', v'>: A(<T_{ji}, ..., T_{j}>, G, p_i>);\}
  \item if there is at least an embedding then return \(<q, p>\);
  \item if \(i > 0\) then return \(<i, p>\);
  \item if \(i = 0\) then return \(<0, \delta(p_1)>\);
  \item let \(w_1, ..., w_k\) be the children of \(v\) of \(v_i\);
  \item \(j:=f+1;\) (*supplement checking*)
  \begin{enumerate}
    \item \(<i, v>:=A(<T_{j1}, ..., T_{j}>, G[w_1], ..., G[w_k], w_1>);\)
    \item if \((v' = v \text{ and } i'>0)\) then return \(<i+i', v>\);
  \end{enumerate}
\end{enumerate}

\subsection{B-function}

In \(B(T, G, u)\), we need to distinguish between two cases.

\textbf{Case 1:} \(G = <P_1, ..., P_q> (q > 1), \) but \(|\mathcal{T}| \leq |P_1| + |P_2|\).

In this case, what we can do is to find whether \(P_1\) or a left corner in \(P_1\) can be embedded in \(T = <t, T_1, ..., T_q>.\) For this purpose, the following checks should be conducted:

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

ii) If \(|\mathcal{T}| > 1, \) but \(|\mathcal{T}| < |P_1| \text{ or } h(t) < h(p_1)\), we will make a recursive call \(B(T, <P_{11}, ..., P_{1i}>), u),\) where \(<P_{11}, ..., P_{1i}>\) is a forest of the subtrees of \(P_1.\) The return value of \(B(T, <P_{11}, ..., P_{1i}>), u)\) is used as the return value of \(B(T, G, u).\) It is because in this case, \(T\) is not able to include the whole \(P_1.\) So what we can do is to check \(T\) against \(<P_{11}, ..., P_{1i}>\).

iii) If \(|\mathcal{T}| \geq |P_1| \text{ and } h(t) \geq h(p_1) (\text{ but } |\mathcal{T}| \leq |P_1| + |P_2|),\) we further distinguish between two sub-cases:

- \(\text{label}(t) = \text{label}(p_1).\) In this case, we will call \(A(<T_{11}, ..., T_{1i}>, <P_{11}, ..., P_{1i}>), u)\) or \(A(<T_{11}, ..., T_{1i}>, <P_{11}, ..., P_{1i}>, u),\) depending on whether \(u = p_1.\) If \(u = p_1,\) the cut for this call is set to be \(p_{11}\) because if the left corner returned by this call is not higher than \(p_{11},\) it will not be used.

- \(\text{label}(t) \neq \text{label}(p_1).\) In this case, we will call \(A(<T_{11}, ..., T_{1i}>, <P_{11}, ..., P_{1i}>, <P_{11}, ..., P_{1i}>)\).

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

- If \(\text{label}(t) = \text{label}(v)\) and \(i = d(v),\) the return value of \(B(T, G, u)\) is set to be \(<1, v>\).
- Otherwise, the return value of \(B(T, G, u)\) is the same as \(<i, v>\).

\textbf{Case 2:} \(G = <P_1, ..., P_q> (q > 1), \) and \(|\mathcal{T}| > |P_1| + |P_2|).\)

In this case, we will call \(A(<T_{11}, ..., T_{1i}>, G, u)\). Assume that the return value of \(A(<T_{11}, ..., T_{1i}>, G, u)\) is \(<i, v>\). The following checkings will be continually conducted.

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

v) If \(v = p_1's\ parent,\) the return value of \(B(T, G, u)\) is the same as \(<1, v>\).

In terms of the above discussion, we give the following formal description of the algorithm, in which \(B(\ )\) is used to handle Case 1 and \(B'(\ )\) for case 2.

\textbf{function} \(B(T, G, u)\) (*Initially, \(u = \rho(G)^\).*

\begin{enumerate}
  \item let \(T = <t, T_1, ..., T_q>, G = <P_1, ..., P_q>, u\) - cut.
  \item output: \(<i, v>\) specified above.
\end{enumerate}

\textbf{begin}

1. if \(p_1\) is a descendant of \(u\) then return \(<0, \delta(p_1)>\);

2. if \((q = 1 \text{ or } |\mathcal{T}| \leq |P_1| + |P_2|)\) then return \(B'(T, P, u)\);

3. else return \(B'(T, G, u)\); (*Case 1*)

\textbf{function} \(B'(T, P, u)\) (*Case 1*)

\begin{enumerate}
  \item let \(T = <t, T_1, ..., T_q>, p = <P_1, ..., P_q>\);
  \item if \(u\) is a leaf then (*Case 1 - (i)*)
    \begin{enumerate}
      \item \(\text{let}(p) = v;\)
      \item if \(\text{label}(t) = \text{label}(v)\) then return \(<1, v>\)’s parent;
      \item else return \(<0, v>\);
    \end{enumerate}
  \item if \(|\mathcal{T}| < |P_1| \text{ or } h(t) < h(p)\) then return \(B(T, <P_1, ..., P_q>, u);\)
    (*Case 1 - (ii)*)
  \item if \(\text{label}(t) = \text{label}(p)\) (*Case 1 - (iii)*)
    \begin{enumerate}
      \item \(\text{if}\) \(p\) is a leaf then \(v := p\)’s parent; \(i := 1;\)
      \item else \(\text{if}\) \(u = p\)
        \begin{enumerate}
          \item \(\text{then}\) \(v := A(<T_{1}, ..., T_q>, <P_{1}, ..., P_q>, \text{parent of } p_1);\)
          \item \(\text{else}\) \(v := A(<T_{1}, ..., T_q>, <P_{1}, ..., P_q>, u);\)
          \item \(\text{if}\) \(\text{label}(t) = \text{label}(v)\) and \(i = d(v)\)
            \begin{enumerate}
              \item \(\text{then}\) \(v := v\)’s parent; \(i := 1;\)
            \end{enumerate}
        \end{enumerate}
    \end{enumerate}
  \item \(\text{end}\)
\end{enumerate}

\textbf{end}\n
\textbf{function} \(B''(T, G, u)\) (*Case 2*)

\begin{enumerate}
  \item let \(T = <t, T_1, ..., T_q>;\)
  \item \(<i, v> := A(<T_{1}, ..., T_q>, G, u);\)
  \item if \(v \neq p_1\)’s parent (*Case 2 - (iv)*)
    \begin{enumerate}
      \item \(\text{if}\) \(\text{label}(t) = \text{label}(v)\) and \(i = d(v)\)
        \begin{enumerate}
          \item \(\text{then}\) return \(<1, v>\)’s parent;
        \end{enumerate}
    \end{enumerate}
  \item \(\text{end}\)
\end{enumerate}

\textbf{end}\n
\textbf{end}\n
\textbf{end}\n
\textbf{end}
7.   }
8. return<i, v>;  (*Case 2 - (v)*)
end

This algorithm is for Case 2. Again, the cut for the B-function is directly propagated to the subfunction calls of A( ) (see line 2.)

3. Conclusion

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|-logD_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.

4. References