Introducing Cuts into Top-down Search: A New Way to Check Tree Inclusion

Yangjun Chen
Department of Applied Computer Science,
University of Winnipeg, Canada.

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, where the order of siblings in both P and T is significant. In this paper, we propose an efficient algorithm for this problem. Its time complexity is bounded by $O(|T|\cdot\log h_P)$ with $O(|T| + |P|)$ space being used, where h_P represents the height of P. Up to now the best algorithm for this problem needs $\Theta(|T|\cdot|leaves(P)|)$ time [1], where leaves(P) stands for the set of the leaves of P.

Keywords: tree matching, tree inclusion, top-down tree search.

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.

Technically, it is convenient to consider a slight generalization of trees, namely forests, which are defined to be a set of disjoint trees. A tree T consisting of a specially designated node $root(T) = t$ (called the root of the tree) and a forest $<T_1, ..., T_k>$ (where $k \geq 0$) is denoted as $<t; T_1, ..., T_k>$. We also call T_j $(1 \leq j \leq k)$ a direct subtree of t, and denote the set of nodes and edges by $V(T)$ and $E(V)$, respectively. The *size* of T is denoted by $|T|$.

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., $u \neq v$. We will use $u \Rightarrow v$ to represent that u is a proper ancestor of v.

The ancestorship in a tree can be checked very efficiently by using a kind of tree encoding, which labels each node v in a tree with an interval $I_v = [a_v, b_v]$, where b_v denotes the rank of v in a *post-order* traversal of the tree. Here the ranks are assumed to begin with 1, and all the children of a node are assumed to be ordered and fixed during the traversal. Furthermore, a_v denotes the lowest rank for any node u in $T[v]$ (the subtree rooted at v, including v). Thus, for any node u in $T[v]$, we have $I_u \subseteq I_v$ since the post-order traversal visits a node after all its children have been visited.

Let $I = [a, b]$ be an interval. We will refer to a and b as $I[1]$ and $I[2]$, respectively.

Lemma 1 For any two intervals I and I' generated for two nodes in a tree T, one of four relations holds: $I \subset I'$, $I' \subset I$, $I[2] < I'[1]$, or $I'[2] < I[1]$. □

Based on Lemma 1, the left-to-right ordering of nodes can also formally be defined. A node 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 $I_u[2] < I_v[1]$.

In the following, we use $\prec$ to represent the left-to-right ordering. Also, $v \preceq v'$ iff $v \prec v'$ or $v = v'$.

The following definition is due to [1].

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) $v \Rightarrow u$ iff $\varphi(v) \Rightarrow \varphi(u)$, i.e., $I_u \subset I_v$ iff $I_{\varphi(u)} \subset I_{\varphi(v)}$; (ancestor condition)

iii) $v \prec u$ iff $\varphi(v) \prec \varphi(u)$, i.e., $I_v[2] < I_u[1]$ iff $I_{\varphi(v)}[2] < I_{\varphi(u)}[1]$. (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. 1 shows an example of an ordered tree inclusion.

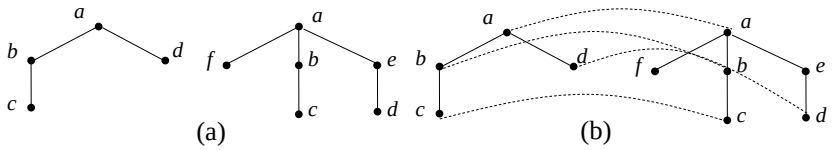

Figure 1: (a) The tree on the left can be included in the tree on the right; (b) an embedding represented by the dashed lines.

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.

Fig. 1(b) also shows an example of a root preserving embedding. According to [1], restricting to root-preserving embedding does not lose generality. In fact, the method to be discussed works top-down and always tries to find root-preserving subtree embeddings.

2 Main Idea

In this section, we discuss the main idea of our algorithm and show why this idea will lead to an optimal computational complexity.

The main idea of our algorithm consists in a mechanism called *cut checking* introduced into a top-down tree search to get rid of useless computations.

Let $T = \langle t; T_1, ..., T_k \rangle$ ($k \geq 0$) be a tree and $G = \langle P_1, ..., P_q \rangle$ ($q \geq 0$) be a forest. We handle G as a tree $P = \langle v_G; P_1, ..., P_q \rangle$, where v_G 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 $G = \langle P_1, ..., P_q \rangle$ with children $v_1, ..., v_k$. We use a pair $\langle [i, j], v \rangle$, called an *interval* rooted at v, to represent an ordered forest $\langle G[v_i], ..., G[v_j] \rangle$ made up of a series of subtrees rooted at $v_i, ..., v_j$, respectively. Especially, $\langle [1, i], v \rangle$ (or simply denoted as $\langle i, v \rangle$) represents an ordered forest containing the first i subtrees of v: $\langle G[v_1], ..., G[v_i] \rangle$. If v is v_G, or a node on the left-most path in P_1, $\langle i, v \rangle$ is called a *left-corner* of G [6]. Obviously, $\langle i, v_G \rangle$ is a left-corner, representing the first i subtrees in G: $P_1, ..., P_i$. So, $\langle q, v_G \rangle$ stands for the whole G. In addition, we will use $\overline{\langle i, v \rangle}$ to represent the forest $\langle G[v_{i+1}], ..., G[v_k] \rangle$, referred to as the complement of $\langle i, v \rangle$. When it is clear from a context, we may use $\langle G[v_i], ..., G[v_j] \rangle$ and $\langle [i, j], v \rangle$ interchangeably without causing any confusion. Let u be a node on the left-most path in P_1. Let $\langle i, v \rangle$ be a left-corner of $G = \langle P_1, ..., P_q \rangle$. If $v = u$, we say that $\langle i, v \rangle$ and u are *level-equal*, denoted as $\langle i, v \rangle \cong u$. If v is an ancestor of u, we say, $\langle i, v \rangle$ is higher than u, denoted as $\langle i, v \rangle \rightsquigarrow u$. Then, $\langle i, v \rangle \underline{\rightsquigarrow} u$ represents that $\langle i, v \rangle$ is higher than or level-equal to u.

In particular, we will use $A(T, G) = \langle i, v \rangle$ to represent a checking of G against T, returning a left-corner $\langle i, v \rangle$ in G with the following properties:

- If $i > 0$ and v is not the left-most leaf node, it shows that
  - the first i subtrees of v can be embedded in T;
  - for any $i' > i$, $\langle i', v \rangle$ cannot be embedded in T; and
  - for any v's ancestor u on the left-most path in G, there exists no $j > 0$ such that $\langle j, u \rangle$ is able to be embedded in T.
- If $i = 0$ or v is the left-most leaf node of G (denoted as $\rho(G)$), it indicates that no left-corner of G can be embedded in T.

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

Now we consider a tree T and a forest G shown in Fig. 2, in which each node in T 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 (p_j) is represented by T_i (resp. P_j).

In order to check whether T includes $G = \langle P_1, P_2 \rangle$, we can first check whether T_1 includes G. That is, we will perform a recursive call as follows:

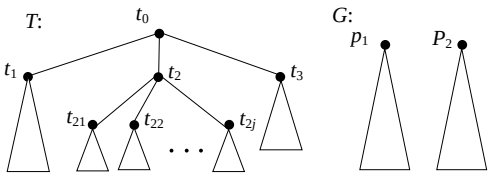

Figure 2. A tree and a forest

$A(T, <P_1, P_2>) \rightarrow A(T_1, <P_1, P_2>)$.

Assume that $A(T_1, <P_1, P_2>)$ returns $<i, v>$. We may have one of three cases:

Case 1: $<i, v> = <2, v_G>$.

Case2: $<i, v> = <1, v_G>$.

Case 3: $v \neq v_G$, but a node on the left-most path in P_1. That is, T_1 contains only a left-corner not higher than p_1.

In Case 1, T_1 contains G. In Case 2, T_1 contains only P_1, and we will call $A(T_2, <P_2>)$ in a next step. In Case 3, we will continually check whether T_2 alone is able to include G (by calling $A(T_2, <P_1, P_2>)$) . This time, however, we will use v (from $<i, v>$) to control the working process to cut off part of computation once we find that it cannot lead to a left-corner higher than v. It is because such a computation will not make any contribution to the final result due to the following operations to be conducted.

Assume that $A(T_2, <P_1, P_2>)$ returns $<i', v'>$ with $v = v'$ or $v \Rightarrow v'$. Then, in a next step, we will check T_3 against $<P_1, P_2>$ by calling $A(T_3, <P_1, P_2>)$. If its return left-corner is higher than v, then we will use this left-corner as the return value of $A(T, <P_1, P_2>)$. Then, $<i', v'>$ will not be used. If its return left-corner is not higher than v, we will make a *supplement checking* of $<T_2, T_3>$ against $\overline{<i, v>}$ to see whether $<T_2, T_3>$ is able to embed some subtrees in $\overline{<i, v>}$. Assume that $<T_2, T_3>$ embeds the first j subtrees in $\overline{<i, v>}$. Then, the return value of $A(T, <P_1, P_2>)$ should be $<i + j, v>$. In this case, $<i', v'>$ will not be used, either, according to the following analysis:

If $v \Rightarrow v'$, or $v = v'$ but $i' \leq i$, $<i', v'>$ is obviously useless for the final result. However, even if $v = v'$ with $i' > i$, it is still useless since in this case there is definitely an integer $j \geq i' - i$ such that $<T_2, T_3>$ embeds the first j subtrees in $\overline{<i, v>}$, and the supplement computation will find this embedding.

The above discussion shows that if $A(T_2, <P_1, P_2>)$ cannot return a left-corner higher than v, the corresponding work is futile and should be avoided. However, avoiding the whole work seems not possible. Yet we can really effectively block a significant part of the useless computation by using the partial results obtained in the previous steps.

We refer to a node which is used to eliminate useless work as a *cut*. With respect to cuts, two issues have to be addressed: (i) how a cut is transferred between two consecutive recursive calls of the *A*-function; and (ii) how a cut is checked during an execution of the *A*-function, which will be specified in the next section where the whole algorithm will be discussed.

3 Algorithm

In this section, we present our algorithm to check a tree T (= <t; T_1, …, T_k>) against a forest G (= <P_1, ..., P_q>), by which for the purpose of optimality, a cut v is utilized. So, T, G, and v should be its input. For simplicity, it is denoted as $A(T, G, v)$ and considered to be a variant of the A-function discussed in Section 2.

Initially, v is set to be $\rho(G)$, and therefore no cutting at the very beginning is in fact imposed. In addition, the algorithm works in a multiple recursive way in the sense that different kinds of recursive calls will be carried out in terms of different characteristics of inputs. First, as mentioned in the previous section, a simple-checking of cuts will be conducted to see whether p_1's parent $\Rightarrow v$. If it is not the case, the algorithm will output <0, ρ(G)>. Otherwise, the checking will be conducted, by which two general cases need to be recognized:

In Case 1, we have G = <P_1>; or G = <P_1, ..., P_q> with $q > 1$, but $|T| \leq |P_1| + |P_2|$. In this case, what we can do is to check T against P_1 since it is not possible for T to embed more than one subtree in G.

In Case 2, we have G = <P_1, ..., P_q> with $q > 1$, and $|T| > |P_1| + |P_2|$. In this case, we will check <T_1, ..., T_k> against the whole G since in this case we may have a sequence of subtrees T_{l_1} , …, T_{l_m} with each being able to embed some subtrees in G. For this reason, we define two subfunctions: α-function and β-function, used to handle Case 1 and Case 2, respectively.

$\alpha(T, P_1, v)$ returns P_1, or a highest and wildest left-corner in P_1, which can be embedded in T, higher than v. Otherwise, it returns <0, ρ(G)>. Similarly, $\beta(T, G, v)$ returns a highest and wildest left-corner in G, embeddable in <T_1, ..., T_k> and higher than v. Otherwise, it returns <0, ρ(G)>.

Here, our intention is quite straightforward: in Case 1 we will call $\alpha(T, P_1, v)$ and in Case 2 we will call $\beta(<T_1, ..., T_k>, G, v)$. However, in Case 2, the return value <j, u> of $\beta(<T_1, ..., T_k>, G, v)$ needs to be further checked as follows:

- If $u \neq p_1$'s parent, check whether label(t) = label(u) and $j = d(u)$. If it is not the case, the return value of $A(T, G, v)$ is the same as <j, u>. Otherwise, the return value of $A(T, G, v)$ will be set to <1, u's parent>.
- If $u = p_1$'s parent, the return value of $A(T, G, v)$ is the same as <j, u>, showing that T embeds <P_1, ..., P_j>.

By using the α-function and the β-function, the algorithm for $A(T, G, v)$ can be described as in Fig. 3.

In the following, both the α-function and β-function will be discussed in great detail.

- α-function

In order to implement the α-function, we need to associate each node v in G with a link to the left-most leaf node in $G[v]$, denoted as $\delta(v)$, as illustrated in Fig. 4(a).

FUNCTION $A(T, G, v)$

input: $T = <t; T_1, ..., T_k>$, $G = <P_1, ..., P_q>$.
output: a left corner.
begin
1 **if** p_1's parent is not an ancestor of v **then** return <0, $\rho(G)$>;
2 **if** (q = 1 or $|T[t]| \le |G[p_1]| + |G[p_2]|$)
3 **then** return $\alpha(T, P_1, v)$
4 **else** $<j, u> := \beta(<T_1, ..., T_k>, G, v)$;
5 **if** $v \neq p_1$'s parent
6 **then if** (label(t) = label(u) $\wedge$ $j = d(u)$ **then** return <1, u's
7 return <j, u>; parent>;

Figure 3. *A*-function

Let v' be a leaf node in G. $\delta(v')$ is defined to be a link to v' itself. So in Fig. 4(a), we have $\delta(v_1) = \delta(v_2) = \delta(v_3) = \delta(v_4) = v_4$, $\delta(v_5) = \delta(v_6) = v_6$, $\delta(v_7) = v_7$, and $\delta(v_8) = v_8$. 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. 4(a), we have $\delta^{-1}(v_4) = \{v_1, v_2, v_3, v_4\}$, $\delta^{-1}(v_6) = \{v_5, v_6\}$, $\delta^{-1}(v_7) = \{v_7\}$, and $\delta^{-1}(v_8) = \{v_8\}$.

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

Let $T = <t; T_1, \ldots, T_k>$, $G = <P_1, ..., P_q>$, and v be a node on the left-most path in P_1. In $\alpha(T, P_1, v)$, altogether seven different cases as listed in Fig. 4(b) should be checked.

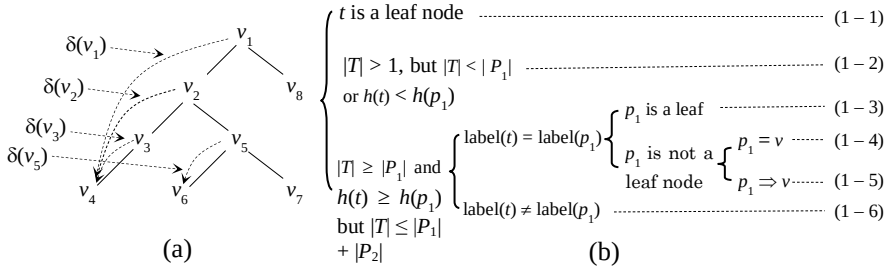

Figure 4. $\delta(v)$ and different cases to be checked in α-function

Obviously, in Case (1-1), where t is a leaf node, we will check whether label(t) = label($\delta(p_1)$) since $\delta(p_1)$ is the only left-corner which can possibly be covered by t. If it is the case, return <1, parent of $\delta(p_1)$>. Otherwise, return <0, $\delta(p_1)$>.

In Case (1-2), where $|T| > 1$, but $|T| < |P_1|$ or $h(t) < h(p_1)$, we will make a recursive call $A(T, <P_{11}, ..., P_{1j}>, v)$, where $<P_{11}, ..., P_{1j}>$ is a forest containing all the direct subtrees of p_1. The return value of $A(T, <P_{11}, ..., P_{1j}>, v)$ is used as the return value of $\alpha(T, P_1, v)$. It is because in this case, T is not able to embed the whole P_1. So we will try to find whether T is able to embed a left-corner within $<P_{11}, ..., P_{1j}>$.

In Case (1-3), where $|T_1| \ge |P_1|$, $h(t) \ge h(p_1)$ (but $|T| \le |P_1| + |P_2|$)), p_1 is a leaf node and label(t) = label(p_1), we will simply return <1, p_1's parent>. If p_1 is not a leaf node, we have Case (1-4) or (1-5), depending on whether $p_1 = v$ or $p_1 \Rightarrow v$. If $p_1 = v$ (Case

1-4), we will call $\beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, p_{11})$. Here, we have a vertical cut propagation and the cut is changed from $v = p_1$ to p_{11}. If $p_1 \Rightarrow v$ (Case 1-5), we will call $\beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, v)$ with the cut not updated. In both Cases 1-4 and 1-5, let $<j, u>$ be the return value of the corresponding β-function call. We will further check whether $j = d_{out}(u)$ and label(t) = label(u). If it is the case, the return value of $\alpha(T, P_1, v)$ will be set to <1, u's parent>. Otherwise, it should be the same as $<j, u>$.

In Case (1-6), where $|T| \geq |P_1|$, $h(t) \geq h(p_1)$, and label(t) ≠ label(p_1), we will call $\beta(<T_1, ..., T_k>, <P_1>, v)$ and the return value of this call will be used as the return value of $\alpha(T, P_1, v)$.

According to the above discussion, we give the formal algorithm for the α-function in Fig. 5.

FUNCTION $\alpha(T, P_1, v)$

input: $T = <t; T_1, ..., T_k>$, $P_1 = <p_1 ; P_{11}, ..., P_{1j}>$.
output: a left corner.
begin
1 **if** (1-1) **then** **if** label(t) = label($\delta(p_1)$)
2. **then** return <1, $\delta(p_1)$'s parent>
3. **else** return <0, $\delta(p_1)$>;
4. **if** (1-2) **then** return $A(T, <P_{11}, ..., P_{1j}>, v)$;
5. **if** (1-3) **then** return <1, p_1's parent>;
6. **if** (1-4) or (1-5) **then**
7. **if** (1-4) **then** $<j, u> := \beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, p_{11})$
8. **else** $<j, u> := \beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, v)$;
9. **if** $j = d_{out}(u)$ and label(t) = label(u)
10. **then** return <1, u's parent>
11. **else** return $<j, u>$;
12. **if** (1-6) **then** return $\beta(<T_1, ..., T_k>, <P_1>, v)$;
end

Figure 5. α-function

- β-function

In comparison with the α-function, the β-function is more interesting. It is designed to handle the general

Case 2. Let $F = <T_1, ..., T_k>$, $G = <P_1, ..., P_q>$, and $v \in \delta^{-1}(\rho(G))$. Denote by t_l the root of T_l (l = 1, ...,k). Denote by p_j the root of P_j (j = 1, ...,q). In $\beta(F, G, v)$, we will make a series of calls $A(T_l, <P_{j_l}, ..., P_q>, v_l)$, where l = 1, ..., $x \leq k$, $j_1 = 1$, $j_1 \leq j_2 \leq ... \leq j_x \leq q$, and $v_1 = v$, 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 $<i_l, u_l>$ be the return value of $A(T_l, <P_{j+1}, ..., P_q>, v_l)$. If $u_l = p_{j+1}$'s parent, set j to $j + i_l$ and v_{l+1} to p_j. Otherwise, j is not changed. Set l to $l + 1$ and v_{l+1} to $higher\{u_l, v_l\}$. Go to (2).

3. The loop terminates when all T_l's or all P_j's are examined.
4. If $j > 0$ when the loop terminates, $\beta(F, G, v)$ returns <j, p_1's parent>, indicating that F 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 looking for a highest and widest left-corner <i, u> in P_1, which can be embedded in F. This can be done as follows.
   i) Let <i_1, u_1>, ...,<i_k, u_k> be the return values of $A(T_1, <P_1, ..., P_q>, v_1)$, ..., $A(T_k, <P_1, ..., P_q>, v_k)$, respectively. Since $j = 0$, each u_l, $v_l \in \delta^{-1}(\rho(G))$ ($l = 1, ..., k$).
   ii) If each $i_l = 0$, the return value of $\beta(F, G, v)$ should be <0, $\rho(G)$>. Otherwise, there must be some u_l's (higher than v) with $i_l > 0$. We call such a node a *non-zero point*. Find the first non-zero point u_f with children w_1, ...,w_s such that u_f is not a descendant of any other non-zero point. Then, we will check <T_{f+1}, ..., T_k> against <$P[w_{i_f+1}]$, ..., $P[w_s]$>. This can be done by a recursive call $\beta(<T_{f+1}, ..., T_k>, <P[w_{i_f+1}], ..., P[w_{i_f+y}]>, w_{i_f+1})$. 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 $\beta(F, G, v)$ should be set to <$i_f + y$, u_f>. □

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

Also, special attention should be paid to the condition under which a supplement computation is conducted:

- $j = 0$, and
- there exists at least non-zero point, which is higher than v.

We refer to this condition as the *supplement checking condition* (*SCC*-condition for short). We also notice that in a supplement computation no further supplement computation will be carried out due to the way the cut for this is set, by which the cut is set to be the root of the first subtree of the forest to be checked. This will effectively block any supplement computation with a supplement computation.

In terms of the above discussion, we give the formal algorithm for the β-function in Fig. 6, which is in fact an extension of the *bottom-up* process given in [2, 3], but with the cuts integrated into the process to control the supplement computation.

In the above algorithm, we have two *while*-loops: one from line 2 to 7 and the other from line 12 to 15. In the first *while*-loop, we do the main computation to find a largest j such that <T_1, ..., T_k> embeds <P_1, ..., P_j>. In this process, by the first A-function call we have a vertical cut propagation while by the subsequent A-function calls the cuts are horizontally propagated.

In the second *while*-loop, the *supplement computation* will be conducted. However, this is done only when the *SCC*-condition is satisfied.

```
FUNCTION β(F, G, v)
input: F = <T_1, ..., T_k>, G = <P_1, ..., P_q>.
output: a left corner.
begin
1  l := 1; j := 0; u := v; f := 0;
2  while (j < q and l ≤ k) do          (*main checking*)
3     <i_l, u_l> := A(T_l, <P_{j+1}, ..., P_q>, u)
4     if (u_l = p_1's parent and i_l > 0) then {j := j + i_l; u := p_j;}
5     else if (u_l is an ancestor of u and i_l > 0)
6          then {u := u_l; f := l;}
7     l := l + 1;
8  if j > 0 then return <j, p_1's parent>;
9  if f = 0 then return <0, δ(p_1)>;
10 let w_1, ..., w_s be the children of u_f;  (*supplement checking*)
11 l := f + 1; j := i_f;
12 while (j < s and l ≤ k) do
13    <i_l, v_l> := A(T_l, <G[w_{j+1}], ..., G[w_s]>, w_{j+1});
14    if (v_l = v_f and i_l > 0) then j := j + i_l;
15    l := l + 1;
16 return <j, u_f>;
end
```

Figure 6. β-function

Finally, we point out that corresponding to a α-function call we may have more than one α-function calls (see line 4), by which a node t is checked against more than one node along a left-most path in G to find a node p in a subtree rooted at a certain node in G (see line 4 in α( )) such that $|T[t]| \geq |G[p]|$ and $h(t) \geq h(p)$. This process can be trivially improved by storing each left-most path in a sorted array and using a binary search.

4 Conclusion

In this paper, a new algorithm is proposed to solve the ordered tree inclusion problem. It requires only $O(|T| \cdot \log h_P)$ time and $O(|T| + |P|)$ space, where T and P are a target and a pattern tree (forest), respectively; and h_P is the height of P.

References

[1] P. Bille and I.L. Gørtz, The Tree Inclusion Problem: In Linear Space and Faster, *ACM Transaction on Algorithms*, Vol. 7, No. 3, Article 38, July 2011, pp. 38:1-38:47.

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

[3] Y. Chen and Y.B. Chen, A Time and Space Efficient Algorithm for Tree Inclusion Problem, in: *Proc. International Conference on Future Communication, Information and Computer Science (FCICS 2014)*, Beijing, China, May 22-23, 2014.