Introducing Cuts into a Top-down Process for Checking Tree Inclusion

Yangjun Chen and Yibin Chen

Abstract—By the ordered tree inclusion 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 \( T \) matters. This problem has many applications in practice, such as retrieval of documents, data mining, and RNA structure matching. In this paper, we propose an efficient algorithm for this problem. Its time complexity is bounded by \( O(|T| \cdot \min(h_T, |\text{leaves}(P)|)) \), with \( O(|T| + |P|) \) space being used, where \( h_T \) represents the height of \( P \) (resp. \( T \)) and \( |\text{leaves}(P)| \) stands for the set of the leaves of \( P \). Up to now the best algorithm for this problem needs \( \Theta(|T| \cdot \text{leaves}(P)) \) time and \( O(|P| + |T|) \) space. Extensive experiments have been done, which show that the new algorithm can perform much better than the existing ones in practice.

Index Terms—Tree matching, tree inclusion, ordered trees, cuts, cut propagation

1 INTRODUCTION

The ordered tree inclusion is important in applications such as document retrieval, data mining, and RNA structure matching, 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 \( T \) counts.

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. Let \( v \) be a node different of the root in \( T \) with parent node \( u \). Denote by \( \text{delete}(T, v) \) the tree obtained by removing the node \( v \) from \( T \), by which the children of \( v \) become part of the children of \( u \) as illustrated in Fig. 1.

![Figure 1. Illustration of node deletion](image1.png)

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

\[
T_0 = T \\
T_{i+1} = \text{delete}(T_i, v_{i+1}) \text{ for } i = 0, ..., k - 1,
\]

we have \( T_k = P \)? If this is the case, \( T \) is said to include \( P \). Such a problem is called the tree inclusion problem [11].

This problem has been recognized as an important query primitive for XML data [14], where a structured document database is considered as a collection of parse trees that represent the structure of the stored texts and tree inclusion is used as a means of retrieving information from them.

As an example, consider the tree shown in Fig. 2, representing an XML document for the book *Arts of Programming* authored by D. Knuth. One might want to find this book in an XML database by forming a pattern tree as shown in Fig. 3 as a query, which can be obtained by deleting some nodes from the tree shown in Fig. 2. Thus, a tree inclusion checking needs to be conducted to evaluate this query.

![Figure 2. A XML document (target) tree](image2.png)

![Figure 3. A pattern tree](image3.png)

Another application of this problem is to query the grammatical structures of English sentences, which can also be represented as an ordered tree since a sentence can always be divided into several ordered components such as noun phrases, verb phrases, and adverbs; and a noun phrase itself normally contains an article and a noun while...
a verb phrase may contain a verb, a noun phrase, an adverb, and so on. To check whether a concrete sentence is grammatically correct, we will represent it as a pattern tree and make a tree inclusion checking against some target grammatical tree structures.

A third application of the ordered tree inclusion is the video content-based retrieval. According to [16], 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 computational biology, such as RNA structure matching [13], and data mining, such as tree mining discussed in [17], just to name a few.

Up to now, the best algorithm for this problem requires $O(|T| + |P|)$ space and $O(|T| - |leaves(P)|)$ time [2], where $leaves(P)$ stands for the set of the leaves of $P$.

In this paper, we propose an efficient algorithm for this problem. Its time and space complexities are bounded by $O(|T| - \text{min} [h_T, |leaves(P)|])$, and $O(|T| + |P|)$, respectively, where $h_T$ (resp. $h_P$) is the height of $T$ (resp. $P$), defined to be the number of edges on the longest downward path from the root to a leaf node.

The rest of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we give some basic definitions and describe what is a tree inclusion. In Section 4, we discuss the main idea of our method. Section 5 and 6 are devoted to the algorithm description. First, in Section 5, a basic algorithm is presented, and then how it can be improved is discussed in Section 6. In Section 7, we analyze the computational complexities. In Section 8, we report the test results. Finally, a short conclusion is set forth in Section 9.

2 Related Work

The ordered tree inclusion was initially introduced by Knuth [12], where only a sufficient condition for this problem is given. Its first polynomial time algorithm was proposed by Kilpeläinen and Mannila [11] with $O(|T| - |P|)$ time and space being used. This computational complexity has been slightly improved by several researchers, but none of them is able to break through the quadratic time bottleneck.

In [15], Richter gave an algorithm using $O(|a(P)| \cdot |T| + m(P, T) \cdot h_T)$ time, where $a(P)$ is the alphabet of the labels of $P$, $m(P, T)$ is the number of matches, defined as all the pairs $(v, w) \in P \times T$ such that label($v$) = label($w$), and $h_T$ (resp. $h_P$) is the height of $T$ (resp. $P$). Hence, if the number of matches is small, the time complexity of this algorithm is better than $O(|T| - |P|)$. The space complexity of the algorithm is $O(|a(P)| \cdot |T| + m(P, T))$. In [3], Chen proposed a more sophisticated algorithm which requires $O(|T| - |leaves(P)|)$ time and $O(|leaves(P)| \cdot \text{min} [h_T, |leaves(T)|] + |T| + |P|)$ space, where $leaves(T)$ (resp. $leaves(P)$) stands for the set of the leaves of $T$ (resp. $P$). The method discussed in [1] is an efficient average case algorithm. Its average time complexity is $O(|T| + C(P, T) \cdot |P|)$, where $C(P, T)$ represents the number of $T$'s nodes that have been examined during the inclusion search. However, its worst time complexity is still $O(|T| + |P|)$. Recently, Bille and Gertz presented a space-economical algorithm [2]. Its space overhead is $O(|T| + |P|)$, but with its time complexity bounded by

\[
\min \left\{ \begin{array}{l}
O(|T| - |leaves(P)|) \\
O(|leaves(T)| - |leaves(P)| \cdot \log \log |T| + |T|) \\
O(|T| - |P| \log |T| + |T| \log |T|)
\end{array} \right.
\]

In [4], a first top-down algorithm was proposed. Its space requirement is bounded by $O(|T| + |P|)$. However, its time complexity is not polynomial, as shown in [9]. This algorithm is improved by [6, 8]. The algorithm discussed in [6] needs $O(|T| - |P|)$ time while the algorithm in [8] requires $O(|T| - d \cdot h_T)$, where $d$ is the largest outdegree of a node in $P$. However, in both [6] and [8], no time analysis is delivered. The algorithm given in [7] fails to produce correct answers in some cases.

In this paper, we revisit this issue and present a new top-down algorithm to remove any redundancy of [4] by introducing cuts into a top-down working process to get rid of any useless computation. Its space overhead is bounded by $O(|T| + |P|)$, and its time complexity is reduced to $O(|T| - \text{min} [h_T, |leaves(P)|])$.

The tree inclusion problem on unordered trees is NP-complete (see [11]) and not discussed in this paper.

3 Basic Definition

In this section, we mainly define the notations that will be used throughout the paper. Let $T$ be a labeled tree that is ordered, i.e., the order between siblings is significant. We denote the set of nodes and edges by $V(T)$ and $E(T)$, respectively. By the size of $T$ we mean the number of nodes in $T$, denoted as $|T|$.

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 $<T_1, ..., T_k>$ is denoted as $t; T_1, ..., T_k$, where $k \geq 0$ and the root of each $T_j (1 \leq j \leq k)$ is a child of $t$. We also call $T_j (1 \leq j \leq k)$ a direct subtree of $t$.

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 $u \sim v$ 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 [10], 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. In Fig. 4, we illustrate such a tree encoding, assuming that the children are ordered from left to right. It is easy to see that by interval containment we can
check whether two nodes are on a same path. For example, $v_2 \sim v_{10b}$ since $I_{v_2} = [1, 5]$, $I_{v_{10b}} = [3, 3]$, and $[3, 3] \subset [1, 5]$; but $v_9$ is not reachable from $v_2$ since $I_{v_9} = [10, 13]$, $I_{v_2} = [1, 2]$, and $[1, 2] \not\subset [10, 13]$.

Let $I = [l, r]$ be an interval. We will refer to $l$ and $r$ in $I$ as $II$ and $Ir$, respectively. The following lemma is from [10].

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

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 $I_u, r < I_v, l$.

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

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

ii) $v \sim u$ if $\phi(v) \sim \phi(u)$, i.e., $I_v \subset I_u$ iff $I_{\phi(v)} \subset I_{\phi(u)}$; (ancestor condition)

iii) $u < v$ if $\phi(u) < \phi(v)$, i.e., $I_v \subset I_u$ iff $I_{\phi(v)} \subset I_{\phi(u)}$. (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. 5 shows an example of an ordered tree inclusion.

**Figure 4. Illustration for tree encoding**

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

The following definition is due to Kilpeläinen and Mannila [11].

Definition 2 Consider an interval of the form $<i, j>$, representing an ordered forest containing the first $i$ subtrees of $v$: $G[v_i], ..., G[v_j]$. For simplicity, it is also denoted as $<i, >$. If $v$ is $v_i$, or a node on the left-most path in $P$, $<i, >$ is called a left-corner of $G$ [5]. □

The motivation to introduce such a concept is that our algorithms are designed to find left-corners. Especially, $<i, v_i>$ is a left-corner, representing the first $i$ subtrees in $G$: $Pv_i, ..., P_i$. So, $<q, v_i>$ stands for the whole $G$.

In addition, we will use $<i, v>$ to represent the forest $<G[v_i], ..., G[v_j]>$, referred to as the complement of $<i, v>$.

When it is clear from the context, we may use $<G[v_i], ..., G[v_j]> and <i, j>$, interchangeably without causing any confusion.

Let $u$ be a node on the left-most path in $P_i$. Let $<i, v>$ be a left-corner of $G$. If $v = u$, we say that $<i, v>$ and $u$ are level-equivalent, denoted as $<i, v> \equiv u$. If $v$ is an ancestor of $u$, we say, $<i, v>$ is higher than $u$, denoted as $<i, v> \sim u$. Then, $<i, v> \sim u$ represents that $<i, v>$ is higher than or level-equivalent to $u$.

In particular, we will use $A(T, G) = <i, v>$ to represent a checking of $G$ against $T$, returning a highest and widest left-corner $<i, v>$ in $G$ with the following properties:

Fig. (b) also shows an example of a root preserving embedding. According to Kilpeläinen and Mannila [11], 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.

Throughout the rest of the paper, the outdegree of $v$ (the number of $v$'s children) in a tree is denoted by $d(v)$ while the height of $v$ is denoted by $h(v)$. The height of a leaf node is set to be 0. In addition, we refer to the labeled ordered trees simply as trees.

In the Appendix I, we show all the notations and symbols used in the paper for reference.

**4 Main Idea - Cuts**

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 computation.

Let $T = <t; T_{1}, ..., T_{k} > (k \geq 0)$ be a tree and $G = <P_{u}, ..., P_{r} > (q \geq 0)$ be a forest (as illustrated in Fig. 6).

**Figure 6. A tree and a forest**

We handle $G$ as a tree $P = <v_{0}, P_{v_{1}}, ..., P_{v_{k}}>$, where $v_{0}$ represents a virtual node, matching any node in $T$. Note that even when $G$ contains only one single tree it is still 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 $G$ with children $v_{1}, ..., v_{k}$. We use a pair $<[i, j], v>$, called an interval rooted at $v$, to represent an ordered forest $<G[v_i], ..., G[v_j]> made up of a series of subtrees rooted at $v_{1}, ..., v_{k}$ respectively.

Definition 2 Consider an interval of the form $[i, j]$, $v>$, representing an ordered forest containing the first $i$ subtrees of $v$: $G[v_i], ..., G[v_j]$. For simplicity, it is also denoted as $<i, >$. If $v$ is $v_i$, or a node on the left-most path in $P_i$, $<i, >$ is called a left-corner of $G$ [5]. □

The motivation to introduce such a concept is that our algorithms are designed to find left-corners. Especially, $<i, v_i>$ is a left-corner, representing the first $i$ subtrees in $G$:

$Pv_i, ..., P_i$. So, $<q, v_i>$ stands for the whole $G$.

In addition, we will use $<i, v>$ to represent the forest $<G[v_i], ..., G[v_j]>$, referred to as the complement of $<i, v>$.

When it is clear from the context, we may use $<G[v_i], ..., G[v_j]> and <i, j>$, interchangeably without causing any confusion.

Let $u$ be a node on the left-most path in $P_i$. Let $<i, v>$ be a left-corner of $G$. If $v = u$, we say that $<i, v>$ and $u$ are level-equivalent, denoted as $<i, v> \equiv u$. If $v$ is an ancestor of $u$, we say, $<i, v>$ is higher than $u$, denoted as $<i, v> \sim u$. Then, $<i, v> \sim u$ represents that $<i, v>$ is higher than or level-equivalent to $u$.

In particular, we will use $A(T, G) = <i, v>$ to represent a checking of $G$ against $T$, returning a highest and widest left-corner $<i, v>$ 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 \( i' \) larger than \( i \), \( < i', v > \) 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 \( < j, u > \) 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 \( T \).

Now we consider a tree \( T \) and a forest \( G \) shown in Fig. 6, in which each node in \( T \) is identified with \( t_{i..j} \) such that \( t_{i..j} \) is the \( k \)th child of \( t_{i..j-1} \). For example, \( t_{i0} \) (simplified as \( t_i \)) is the first child of \( t_{i0} \), \( t_{i12} \) is the second child of \( t_i \), and so on; and each node in \( G \) is identified with \( p_{i..j} \). Besides, each subtree rooted at \( t_{i..j} \) is represented by \( T_{i..j} \) (resp. \( P_{i..j} \)).

In Fig. 6, in order to check whether \( G \) includes \( T \) (called \( G < P \), \( P \supset G \)), we can first check whether \( T \) alone includes \( G \). That is, it will perform a recursive call as follows:

\[
A(T, <P, P>) \rightarrow A(T_i, <P_i, P_i>).
\]

Assume that \( A(T, <P_i, P_i>) \) returns \(<i, v> \). We may have one of three cases:

Case 1: \( <i, v> = <2, w> \). In this case, \( T_i \) contains \( G \).

Case 2: \( <i, v> = <1, w> \). In this case, \( T_i \) contains only \( P_i \), and we will call \( A(T, <P>) \) in a next step.

Case 3: \( v \neq w \), but a node on the left-most path in \( P_i \). That is, \( T_i \) contains only a left-corner not higher than \( p_i \). This case is complicated and needs to be handled carefully, as described below.

In Case 3, we continue to check whether \( T \) alone is able to include \( G \) (by calling \( A(T, <P_i, P_i>) \)). This time, however, we will use \( v \) (in calling \( A(T, <P, P>) \), the return value of \( A(T, <P_i, P_i>) \)) to control the working process to cut off part of the computation once we find that a left-corner higher than \( v \) cannot be produced. It is because such a computation will not make any contribution to the final result due to the following observation.

Assume that \( A(T, <P_i, P_i>) \) returns \(<i', v'> \) with \( v = v' \) or \( v \sim v' \). Then, in a next step, we will check \( T_i \) against \( P_i \) (by calling \( A(T, <P_i, P_i>) \)).

If its return left-corner is higher than \( v \), then we will use this left-corner as the return value \(<i'', v''> \) of \( A(T, <P_i, P_i>) \). Then, \(<i', v'> \) is not used, as illustrated in Fig. 7(a).

In order to check whether \( T \) includes \( G \), we will first check \( T_i \) against \( G = <P_i, P_i> \). Obviously, \( T_i \) is not able to embed \( G \). However, it can embed \( P_i \) and therefore the return value of this checking should be \(<1, p_i> \). In a next step, we will check \( T_i \) against \( G \), and try to see if \( T \) alone is able to embed \( G \). But this time, \( p_i \) will be utilized to control the process. More specifically, it will effectively block the checking of \( T_i \) against \( P_i \) since checking can only possibly return a left-corner not higher than \( p_i \).

We refer to a node like \( p_i \) in Example 1 as a cut.

**Definition 3** A cut for a call \( A(T, <P_i, ..., P_i>) (q ≥ 1) \) is a node \( v \) on the left-most in \( P_i \), indicating that only a left-corner higher than \( v \) will be returned by \( A(T, G, v) \) if it is embeddable in \( T \). Otherwise, \( A(T, <P_i, ..., P_i>) \) returns \( <0, \rho(G)> \).

In the following, we will first, for ease of understanding, give an algorithm for checking tree inclusion without cuts in Section 5. Then, in Section 6, a complete algorithm with
cuts (more specifically, with cut propagation and cut checking) will be presented.

5 Basic Algorithm
In this section, we present our basic algorithm $A(T, G)$ to check a tree $T = (u; T_1, ..., T_k)$ against a forest $G = (P_1, ..., P_p)$. 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. In general, two cases need to be recognized:

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

In Case 2, we have $G = \{P_1, ..., P_p\}$ 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 $T_i, ..., T_m$, with each being able to embed some subtrees in $G$.

It seems that Case 1 is a base case while Case 2 is a general one and needs to be reduced to Case 1 for handling. Due to the hierarchical structure of trees, however, when handling Case 1, we may meet Case 2 again. That is, these two cases can be interleaved in some way. For this reason, we define two subfunctions: $\alpha$-function and $\beta$-function, used to handle Case 1 and Case 2, respectively:

$$\alpha(T, P_i) = \{ i, v_i \}$$

where $i, v_i$ is a highest and widest left-corner in $P_i$, which can be embedded in $T$.

$$\beta(T, i, v_i) = \{ j, u_i \}$$

where $j, u_i$ is a highest and widest left-corner in $G_i$, which can be embedded in $T_i, ..., T_k$.

Here, our intention is quite straightforward:

In Case 1 we will call $\alpha(T, P_i)$ and in Case 2 we will call $\beta(T_i, ..., T_k, G_i)$. However, during the working process, they may call each other recursively.

Additionally, in Case 2, the return value $j, u_i$ of $\beta(T_i, ..., T_k, G_i)$ needs to be further checked as follows:

- If $\text{label}(t) = \text{label}(u)$ and $j = d(u)$, the return value of $A(T, G)$ should be set to $<i, u_i$, showing that $T$ includes $G[u]$. Otherwise, the return value of $A(T, G)$ is the same as $<j, u_i$. (For this reason, $d(u)$ is set to be $\infty$ larger than the outdegree of any node in both $T$ and $G$. Thus, in the case that $T$ contains $<P_i, ..., P_p>$, the return value must be $<q, v_q>$, not $<i, u_i$'s parent>.)

- If $\text{label}(t) \neq \text{label}(u)$ or $j \neq d(u)$, the return value of $A(T, G)$ is the same as $<j, u_i$, showing that $T$ embeds $<P_i, ..., P_p>$.

By using the $\alpha$-function and the $\beta$-function, the algorithm for $A(T, G)$ can be described as below.

FUNCTION 1. $A(T, G)$

input: $T = (u; T_1, ..., T_k), G = \{P_1, ..., P_p\}$

output: a left corner.

begin
1. if $|q| = 1$ or $|T_i| \leq |G[p_1]| + |G[p_2]|$ (*Case 1*)
2. then return $\alpha(T, P_i)$
3. else $j, u_i := \beta(T_i, ..., T_k, G); (*Case 2*)$

4. if $\text{label}(t) = \text{label}(u)$ and $j = d(u)$
5. then return $<i, u_i$'s parent$>$
6. return $<j, u_i;$

end

In the following, both the $\alpha$-function and $\beta$-function will be discussed in great detail.

- $\alpha$-function

In order to implement the $\alpha$-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. 9.

Let $v'$ be a leaf node in $G$. $\delta(v')$ is defined to be a link to $v'$ itself. So in Fig. 9, we have $\delta(v_1) = \delta(v_2) = \delta(v_3) = \delta(v_4) = v_6, \delta(v_5) = v_5, \delta(v_7) = v_7, \delta(v_8) = v_8$. Denote by $\alpha(v)$ a set of nodes $x$ such that for each $v \in x \delta(v) = v'$. Thus, in Fig. 9, we have $\Delta(v_1) = \{v_1, v_2, v_3, v_4\}, \Delta(v_2) = \{v_5, v_6\}, \Delta(v_3) = \{v_7\},$ and $\Delta(v_8) = \{v_8\}$. Let $p_r$ be the root of $P_r$. We also have $\rho(G) = \delta(p_r)$.

![Figure 9](image-url)  
**Figure 9. A pattern tree**

Let $T = (u; T_1, ..., T_k), G = \{P_1, ..., P_p\}$. In $A(T, P_i)$, altogether five different cases as listed in Fig. 10 should be checked.

<table>
<thead>
<tr>
<th>$i$ is a leaf node</th>
<th>1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$\text{label}(t) \neq \text{label}(p_i)$</td>
<td>1-3</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$\text{label}(t) = \text{label}(p_i)$</td>
<td>1-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p_i$ is a leaf</th>
<th>1-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$ is not a leaf</td>
<td>1-5</td>
</tr>
</tbody>
</table>

![Figure 10](image-url)  
**Figure 10. Different cases to be checked in $\alpha$-function**

Obviously, in Case (1-1), where $i$ is a leaf node, we will check whether $\text{label}(t) = \text{label}(\rho(p_i))$ since $\rho(p_i)$ is the only left-corner which can possibly be covered by $t$. If it is the case, return $<1, \rho(p_i)$'s parent$. Otherwise, return $<\rho, \rho(p_i)$).

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

If $|T| \geq |P_1| + |P_2|$, and $h(t) \geq h(p_i)$ (but $\text{label}(t) = \text{label}(p_i)$), we further distinguish among three cases: Case (1-3), (1-4) and (1-5).

In Case (1-3), we have $\text{label}(t) \neq \text{label}(p_i)$, and we will call $\beta(T_1, ..., T_k)\{P_1\}$ to see whether $\beta(T_1, ..., T_k)$ is able to embed $P_i$.

In Case (1-4), we have $\text{label}(t) = \text{label}(p_i)$ and $p_i$ is a leaf node. In this case, we return $<1, p_i$'s parent$>$. 


In Case (1-5), we have label(i) = label(p1), but p1 is not a leaf node. In this case, we need to call β(T1, ..., Ti, <P1w1, ..., P1i>). Assume that the return value of β(1) is less than v. We need to do an extra checking:

- If label(i) = 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 <1, v>

According to the above discussion, we give the following formal algorithm for the α-function.

FUNCTION 2. α(T, P1)

input: T = <T1, ..., Ti, ..., Tj>, P1 = <P1w1, ..., P1i, ..., P1j>
output: a left corner.
begin
1. if (1-1) then return <1, δ(p1)>
2. then return <1, δ(p1)>
3. else return <0, δ(p1)>;
4. if (1-2) then return A(T, <P1w1, ..., P1i>)
5. if (1-3) then return β(<T1, ..., Ti, <P1w1, ..., P1i>)
6. if (1-4) then return <1, p1's parent>
7. if (1-5) then return <1, u: β(<T1, ..., Ti, <P1w1, ..., P1i>)
8. if j = d(u) & label(i) = label(u) return <1, u's parent>
9. else return <j, u>
end

- 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 = <T1, ..., Ti, ..., Tj> and G = <P1w1, ..., P1i>. Denote by h: the root of Ti (l = 1, ..., k). Denote by p1 the root of P1 (j = 1, ..., q). In β(F, G), we will make a series of calls A(Ti, <P1w1, ..., P1i>) where l = 1, ..., x ≤ k, j = 1, j ≤ j ≤ ... ≤ j ≤ q, controlled as follows.

1. Two index variables l, j are used to scan T1, ..., Ti and P1w1, ..., P1i, respectively. (Initially, l is set to 1, and j is set to 0.) They also indicate that <P1w1, ..., P1i> has been successfully embedded in <T1, ..., Ti, Tj, ...>, Ti.
2. Let <i, u> be the return value of A(Ti, <P1w1, ..., P1i>). If u = P1w1's parent, set j = j + 1. Otherwise, j is not changed. Set l to l + 1. Go to (2) (i.e., repeat this step.)
3. The loop terminates when all Tl's or all Pl's are examined. (Fig. 11 helps for illustration of this iteration process.)
4. If j > 0 when the loop terminates, β(F, G) returns <j, p1's parent>, indicating that F contains P1w1, ..., P1i, Otherwise, j = 0, indicating that even P1 alone cannot be embedded in any Ti (l ∈ {1, ..., k}). However, in this case, we need to continue looking for a highest and widest left-corner <i, u> in P1w1 which can be embedded in F. This can be done as follows.

i) Let <i, v1>, ..., <i, vn1> be the return values of A(Ti, <P1w1, ..., P1i>) = A(Ti, P1w1, ..., P1i), respectively. Since j = 0, each v ∈ Ac(p1) (l = 1, ..., k).
ii) If each i = 0, the return value of β(F, G) should be <0, p1(G)>. Otherwise, there must be some v's with i = 0.

We can call such a node a non-zero point. Find the first non-zero point v with children w1, ..., wn such that v is not a descendant of any other non-zero point. Then, we will check <T1, ..., Tj> against <P1w1, ..., P1i, ...>. This can be done by a recursive call β(<T1, ..., Tj>, <P1w1, ..., P1i, ...>, P1v). Let y be a number such that <P1w1, ..., P1i, y> can be embedded in <T1, ..., Tj>. The return value of β(F, G) should be set to <i + y, u>. □

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

If <T1, ..., Tj> includes <P1w1, ..., P1i>, Tδ will be checked against <P1w1, ..., P1i, ...>.

Figure 11. Illustration for an execution of β-function

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

- j = 0, and
- there exists at least a non-zero point.

We refer to this condition as the supplement checking condition (SCC-condition for short). In terms of the above discussion, we give the following formal algorithm for the β-function.

FUNCTION 3. β(F, G)

input: F = <T1, ..., Ti, ..., Tj> = <P1w1, ..., P1i, ...>
output: a left corner.
begin
1. l = 1; j = 0; v := p1(G); f := 0;
2. while (j < q and l < k) do (*main checking*)
3. if (v < p1's parent and i > 0) then j := j + 1;
4. else if (v is an ancestor of v and i > 0) then v := v; f := l;
5. l := l + 1;
6. if j > 0 then return <j, p1's parent>;
7. if f = 0 then return <1, δ(p1)>;
8. l := 1 + 1;
9. l := l + 1;
10. let v1, ..., vn be the children of v;
11. (*supplement checking*)
12. if (v1 < s and l < k) do
13. if (v1 < P1w1) then j := j + 1;
14. if (v1 < v and i > 0) then j := j + 1;
15. l := l + 1;
16. return <j, v>;
end

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 <T1, ..., Tj> embeds <P1w1, ..., P1i>. In the second while-loop, the supplement computation will be conducted when the SCC-condition is satisfied.

In order to record the first non-zero point which is not a descendant of any other non-zero point, variable f is used. Initially, f is set to 0. If no non-zero point is found, we must have f = 0 after the main computation is completed. So only when j = 0 and f > 0, the SCC-condition
is satisfied and the supplement computation will be performed (see lines 8 and 9), in which we check $<\ell_{i},...,\ell_{t}>$ against $<P[w_{i+1}],...,P[w_{t}>$, where $w_{i+1},...,w_{t}$ are all those children of the first non-zero point $v_{j}$ such that the subtrees rooted at them are not covered by $<\ell_{i},...,\ell_{t}>$. (Notice that $v_{j}$ is the return value of $A(\ell_{i},<p_{i},...,p_{t}>$ with $i > 0$.) In Appendix II, we will trace an execution of the basic algorithm when applied to the tree $T$ and the forest $G$ shown in Fig. 8.

- Correctness

Concerning the correctness of this algorithm, we first give two lemmas, based on which a strict proof can then be established.

**Lemma 1** If both the $\alpha$-function and $\beta$-function return the correct value, then the $A$-function must return a correct value. That is, the return value of $A$-function must be a highest and widest left corner in $G$ that can be embedded in $T$.

**Proof.** Let $T = <\ell_{1},...,\ell_{t}>$, $G = <p_{1},...,p_{t}>$. In $A(T,G)$, we distinguish between two cases: (i) $q = 1$ or $|T| \leq |P_{1}| + |P_{2}|$, and (ii) $q > 1$ and $|T| > |P_{1}| + |P_{2}|$. In case (i), what we can do is to check $T$ against $P_{1}$ to find the highest and widest left corner which can be embedded in $T$. This is done by calling $a(T,P_{1})$. If $\alpha$-function is correct, then the return value of $A$-function is correct in this case. In case (ii), if $T$ is able to cover $<p_{i},...,p_{t}>$ with $1 \leq q$, then $<\ell_{i},...,\ell_{t}>$ must be able to cover $<p_{i},...,p_{t}>$ since we cannot use $t$ to map any node $G$.

If we map $t$ to a node, say $p$ in $G$, all the nodes in $<\ell_{i},...,\ell_{t}>$ have to be mapped to the nodes in $G[p]$, excluding $p$, to satisfy the ancestor condition in the definition. So we call $\beta(<\ell_{i},...,\ell_{t},G)$ to do this task. Let $<\ell_{i},w>$ be the return value of $\beta(<\ell_{i},...,\ell_{t},G)$. We further need to check whether label($u$) = label($w$). If it is the case, $T$ covers $<1$, $w$'s parent>. Otherwise, $T$ only covers $<\ell_{i},w>$. Thus, in case (ii), if the $\beta$-function is correct, the return value of the $A$-function must be correct. □

In a similar way, we can prove Lemma 2.

**Lemma 2** If the $A$-function returns the correct value, then the return values of both the $\alpha$-function and $\beta$-function must be correct.

**Proof.** The lemma can be proven by analyzing the five cases in the $\alpha$-function, as well as the main checking and the supplement checking in the $\beta$-function. □

Obviously, we cannot claim the correctness of the algorithm based on Lemma 1 and 2 since they are just a kind of circular arguments. But they can be used in the induction step of an induction proof given in the following proposition if the correctness of $A$-function, or $\alpha$-function and $\beta$-function for the basic case can be established.

**Proposition 1** Let $T = <\ell_{1},...,\ell_{t}>$ and $G = <p_{1},...,p_{t}>$. The return value of $A(T,G)$ is the highest and wildest left corner in $G$, which can be embedded in $T$.

**Proof.** We prove the proposition by induction on the sum of the heights of $T$ and $G$, $H = h_{T} + h_{G}$.

Basic step. When $H = 0$, $T$ is a singular $t$, and $G$ is a set of nodes: $p_{1},...,p_{t}$. In this case, the algorithm returns $<0,p_{t}>$ or $<1,v>$, depending on whether $\text{label}(t) = \text{label}(p_{t})$. See lines 4 - 6 in $A()$.

When $H = 1$, we need to consider the following two cases.

(i) $T$ is a tree of height 1: $<\ell_{1},...,\ell_{t}>$, and $G$ is a set of nodes: $<p_{1},...,p_{t}>$.

(ii) $T$ is a singular $t$; but $G$ is a set of trees of height 1.

In case (i), we further distinguish between two cases.

- If $|T| \leq |P_{1}| + |P_{2}|$ (i.e., if $t$ has at most one child $t_{0}$, $a(T,P_{1})$ will be called (see lines 1 - 2 in $A()$). If label($t$) $\neq$ label($p_{1}$), we have Case (1-3) and will call $\beta(<\ell_{1},<p_{1}>)$, which leads to the call $A(T_{1},<p_{1}>)$ (see line 3 in $\beta()$) and then to the call $a(T_{1},P_{1})$. Since $T_{1}$ contains only a single node $h_{1}$, we have Case (1-1) and returns $<1,v>$ or $<0,v>$ depending on whether $\text{label}(h_{1}) = \text{label}(p_{1})$ (see lines 1 - 3 in $a()$). If label($t$) = label($p_{1}$), we have Case (1-4) since $p_{1}$ is a leaf, and return $<1,v>$.

- If $|T| > |P_{1}| + |P_{2}| = 2$, $\beta(<\ell_{1},...,\ell_{t},<p_{1},...,p_{t}>)$ will be invoked (see line 4 in $A()$), which will find a sequence of integers: $k_{1},...,k_{t}$ such that label($k_{t}$) = label($p_{i}$) ($i = 1, ..., x$) (see line 3 in $\beta()$). The return value is $<x,v>$ ($0 \leq x \leq q$).

In case (ii), the return value is $<0,p_{1}>$ or $<1,p_{1}>$, depending on whether $t$ matches the first child of $p_{1}$. See lines 1 - 2 in $A()$, and Case (1-1) in $a()$.

Induction hypothesis. Assume that when $H = h \geq 1$, the proposition holds.

Consider $T = <\ell_{1},...,\ell_{t}>$ and $G = <p_{1},...,p_{t}>$ with $H = h_{T} + h_{G} = h + 1$.

If $q = 1$, or $q > 1$ but $|T| \leq |P_{1}| + |P_{2}|$, $a(T,P_{1})$ will be invoked. If it is case (1-1), or (1-4), the proposition obviously holds.

If it is case (1-2), $A(T_{1},<p_{1},...,p_{t}>)$ will be invoked. Since the sum of the height of $T$ and the height of $<p_{1},...,p_{t}>$, $H$ is equal to $h_{1}$, according to the induction hypothesis, the proposition holds.

If it is case (1-3), $\beta(<\ell_{1},...,\ell_{t},<p_{1},...,p_{t}>)$ will be called, by which a series of calls $A(T_{1},<p_{1},...,p_{t}>)$ will be conducted, where $l = 1, ..., x \leq k_{t}, j_{1} = 1, j_{2} \leq j_{3} \leq ... \leq j_{t} \leq q$. According to the induction hypothesis, each $A(T_{1},<p_{1},...,p_{t}>)$ returns a correct value. Thus, in terms of Lemma 2, the return value of $\beta(<\ell_{1},...,\ell_{t},<p_{1},...,p_{t}>)$ must be correct.

If it is case (1-5), $\beta(<\ell_{1},...,\ell_{t},<p_{1},...,p_{t}>)$ will be invoked, by which a series of calls $A(T_{1},<p_{1},...,p_{t}>)$ will be carried out. Again, the sum of the height of $T$ and the height of $<p_{1},...,p_{t}>$ equals $h - 1$. So, according to the induction hypothesis and Lemma 2, the proposition also holds.

If $q < 1$ and $|T| > |P_{1}| + |P_{2}|$, $\beta(<\ell_{1},...,\ell_{t},<p_{1},...,p_{t}>)$ will be called, by which a series of calls $A(T_{1},<p_{1},...,p_{t}>)$ will be conducted, where $l = 1, ..., x \leq k_{t}, j_{1} = 1, j_{2} \leq j_{3} \leq ... \leq j_{t} \leq q$. In the same way as (1-3) and (1-5), we can demonstrate the correctness of $A(T,G)$ for this case. □

By Proposition 1, the algorithm will always return a correct answer. However, it is not an efficient algorithm since much useless work has to be conducted, as illustrated in Fig. 7.
6 Algorithm with Cuts

In order to use cuts to discard useless computations, two issues have to be addressed: (i) how a cut is checked during an execution of the \(A\)-function, and (ii) how a cut is transferred between two consecutive recursive calls of the \(A\)-function, \(\alpha\)-function, as well as \(\beta\)-function.

To this end, we change \(A(T, G, v)\) to take an extra parameter \(t \in \Delta(p(G))\), indicating that only a left corner higher than \(v\) will be returned by \(A(T, G, v)\) if it is embeddable in \(T\). Otherwise, \(A(T, G, v)\) returns \(<0, \rho(G)\). \(\alpha\)-function and \(\beta\)-function will also be accordingly changed such that within \(A(T, G, v)\), the cut \(v\) can be transferred to both \(\alpha(T, P_i, v)\) and \(\beta(T_i, \ldots, T_j, G, v)\).

We first slightly modify the \(A\)-function as below. Initially, the cut is set to be \(\rho(G)\).

FUNCTION 4. \(A(T, G, v)\) *(Initially, \(v\) is set to be \(\rho(G)\).)*

input: \(T = t, T_i, \ldots, T_j, G = <P_i, \ldots, P_j>\).

output: a left corner.

Begin
1. if \(p_i\)'s parent is not an ancestor of \(v\) then return \(<0, \rho(G)>\).
2. if \((q = 1 \text{ or } |T_i| < |G[p_1]| + |G[p_2]|)) \quad \text{(Case 1)*}
3. then return \(\alpha(T, P_i, v)\).
4. else if \(\text{label}(t) = \text{label}(v)\) \quad \text{(Case 2*)}
5. then \(<j, u> := \beta(T_i, \ldots, T_j, G, v)\).
6. else \(<j, u> := \beta(T_i, \ldots, T_j, G, v)\);
7. if \(v \neq p_j\)’s parent
8. then \(\text{then return } <1, u\text{'s parent}>\).
9. return \(<j, u>\);

end

In the algorithm, \(\alpha(T, P, v)\) and \(\beta(T_i, \ldots, T_j, G, v)\) are defined as follows.

\[
\alpha(T, P, v) = \begin{cases} 
<j, u>, & \text{if } <j, u> \text{ is a highest and widest left-corner in } P, \text{ which can be embedded in } T, \text{ higher than } v; \\
<0, \rho(G)>, & \text{otherwise.}
\end{cases}
\]

(3)

\[
\beta(T_i, \ldots, T_j, G, v) = \begin{cases} 
<j, u>, & \text{if } <j, u> \text{ is a highest and widest left-corner in } G, \text{ which can be embedded in } \\
\alpha(T_i, \ldots, T_j, G, v) \text{ higher than } v; \\
<0, \rho(G)>, & \text{otherwise.}
\end{cases}
\]

(4)

First, we note that in line 1 we check whether \(p_i\)'s parent is an ancestor of \(v\). If it is not the case, return \(<0, \rho(G)\) since no useful results can be produced. Otherwise, we will call \(\alpha\)-function for Case 1, and \(\beta\)-function for Case 2 as in the basic algorithm, but with a cut transferred.

In addition, in Case 2, depending on whether \(\text{label}(t) = \text{label}(v)\), we will call \(\beta(T_i, \ldots, T_j, G, v)\)’s first child or \(\beta(T_i, \ldots, T_j, G, v)\) (see lines 4 – 6). It is because if \(\text{label}(t)\) is label(v), we may have \(T\) covering \(G[v]\) if \(T_i, \ldots, T_j\) is able to embed a forest made up of all direct subtrees of \(v\). In this case, the return value of \(A(T, G, v)\) should be set to \(<1, v\text{'s parent}>\), higher than \(v\). So, the cut needs to be downgraded to \(v\text{'s first child so that this part of computation will not be blocked.}

- Cut Propagation in \(\alpha\)-function

In \(\alpha(T, G, v)\), for different cases, the cut will be propagated by recursive calls in different ways.

As with the basic version of the \(\alpha\)-function, we will distinguish among five cases, i.e., Case (1-1), (1-2), (1-3), (1-4), and (1-5).

In Case (1-1), no recursive call is conducted and thus the cut \(u\) is not transferred.

In Case (1-2), we will call \(A(T, <P_i, \ldots, P_j>, v)\) by which the cut \(v\) is directly transferred to the recursive call since its return value will be used as the return value of \(\alpha(T, G, v)\).

In Case (1-3), we have \(\text{label}(t) \neq \text{label}(p_i)\). In this case, we will simply call \(\beta(<T_i, \ldots, T_j, G, v>)\), by which \(v\) is directly transferred for the same reason as Case (1-2).

In Case (1-4), there is no recursive call and thus no cut transfer.

In Case (1-5), we will call the \(\beta\)-function to check \(<T_i, \ldots, T_j\) against \(<P_i, \ldots, P_j>\). In this case, we have \(\text{label}(t) = \text{label}(p_i)\). Concerning the cut transfer, we need to consider two subcases:

\(i)\) \(p_i = v\). In this case, we will call \(\beta(<T_i, \ldots, T_j, G, v>)\) with the cut being set to \(p_i\). It is because in this case the main checking of the \(\beta\)-function execution may reveal that \(<T_i, \ldots, T_j>\) is able to embed the whole \(<P_i, \ldots, P_j>\). In this case, the return value of \(\alpha(T, G, v)\) will be set to \(<1, p_i\text{'s parent}>\), higher than \(v\). So it is a useful computation; and downgrading the cut from \(v = p_i\) to \(p_i\) will let it go through. On the other hand, \(p_i\) will effectively prohibit any possible further supplement checking in this \(\beta\)-function execution since such a checking can only bring out a left corner lower than \(p_i\) and will not be used.

\(ii)\) \(p_i \not= v\). In this case, we will call \(\beta(<T_i, \ldots, T_j, G, v>)\), by which \(v\) is directly transferred since we must have \(p_i \not= v\) and no useful computation can be eliminated by cut \(v\).

According to the above discussion, we give the following formal algorithm for the \(\alpha\)-function with cuts.

FUNCTION 5. \(\alpha(T, P_i, v)\)

input: \(T = t, T_i, \ldots, T_j, P_i = <P; P_i, \ldots, P_j>\).

output: a left corner.

Begin
1. if (1-1) then \(\text{if label}(t) = \text{label}(\delta(p_i))\)
2. \(\text{then return } <1, \delta(p_i)'s parent>\)
3. \(\text{else return } <0, \delta(p_i)>\)
4. if (1-2) then return \(\alpha(T, <P_i, \ldots, P_j>, v)\);
5. if (1-3) then return \(\beta(<T_i, \ldots, T_j, G, v>)\);
6. if (1-4) then return \(<1, p_i’s parent>\);
7. if (1-5) then \(<j, u> := \beta(<T_i, \ldots, T_j, G, v>)\);
8. if (1-5) then \(<j, u> := \beta(<T_i, \ldots, T_j, G, v>)\);
9. if \(j = d(u)\) and \(\text{label}(t) = \text{label}(\delta(p_i))\) then return \(<1, u’s parent>\)
10. \(\text{else return } <j, u>\);

end

The only difference of the above algorithm from the basic version is that Case (1-5) is divided into (1-5-i) and (1-5-ii) as aforementioned.
- **Cut Propagation in β-function**

The cut propagation conducted in the α-function is considered as a kind of vertical cut of cuts, by which a cut is propagated to a nested recursive call. By the β-function, however, what we have is a kind of horizontal transfer, by which the local result of a recursive call will be used as a cut for a next parallel recursive call.

Specifically, what we need to do is to determine the cut for each recursive call to check a Ti against a forest of the form < Pj, ..., Pk > with |j| ≥ 1 in the main checking of β(< Ti, ..., Ti >, G, v). Without loss of generality, assume that < w, v > is the return value of A(Ti, < Pj, ..., Pk >, u) for |i| = 1, ..., x ≤ k with j = 1, j ≤ k ≤ ... ≤ j ≤ q. Then, we have

- \( u = v \), a value transferred to β(< Ti, ..., Ti >, G, v).
- \( u = v \) is the return value of A(Ti, < Pj, ..., Pk >, u) for |i| = 1, ..., x ≤ k with j = 1, j ≤ k ≤ ... ≤ j ≤ q. Then, we have

\[ \begin{align*}
  \text{for } 2 \leq i \leq s, \ 
  u & = \begin{cases} 
    v_{i-1}, & \text{if } v_{i-1} \text{ is the ancestor of } u_i \text{ and } i_1 > 0; \\
    w_{i-1}, & \text{if } w_{i-1} \text{ is not an ancestor of } u_i \text{ or } i_1 = 0;
  \end{cases} \\
  \text{and for } s + 1 \leq l \leq k, \ 
  u & = p_{j_l};
\end{align*} \]

(5)

The formula (5) shows how the cuts are changed before we meet the first subtree in < Ti, ..., Ti > which is able to embed some subtrees Pj, ..., Pm. After such a subtree is found, the cuts will be determined in terms of (6). It is because for each subsequent A-function call to check a Ti against < Pj, ..., Pk >, a returned left corner lower than pji will not be used in the subsequent computation.

If s < k, it shows that < Ti, ..., Ti > includes < Pj, ..., Pk > for some m (1 ≤ m ≤ q), and the supplement checking will not be conducted. If s = k, < Ti, ..., Ti > does not include any subtree in G, but some Ti/s each may include a non-empty left corner in Pj. If it is the case and the left corner is also higher than cut v, then a supplement checking will be performed as described in Section 5. That is, when the following two conditions are satisfied, a supplement checking will be carried out:

- \( j = 0 \), and
- there exists at least a non-zero point, which is higher than cut v.

They are referred to as the strict supplement condition (strict SCC-condition for short). In comparison with the supplement property given in Section 5, one more condition with respect to cuts has to be met, i.e., the non-zero point must be higher than cut v.

Besides, in a supplement computation no further supplement computation will be conducted due to the way the cut for this is set, by which the cut is set to be the root of the first subtree in the forest to be checked. This will effectively block any further supplement computation within a supplement computation.

In terms of the above discussion, we give the following formal algorithm for the β-function, which is similar to FUNCTION 3, but with the cuts integrated into the process to control the supplement computation.

FUNCTION 6: β(F, G, v)

**input:** F = < Ti, ..., Ti >, G = < Pj, ..., Pk >.
**output:** a left corner.

1. \( l := 1; j := 0; u := v; f := 0; \) (*main checking*)
2. while \((j < q \text{ and } l \leq k)\)
3. \( i \leftarrow <v, u := A(Ti, < Pj, ..., Pk >, u)\)
4. \( \text{if } (w = p_i \text{'s parent and } i > 0) \) then \( j := j + i; u := p_j; \)
5. \( \text{else if } (w \text{ is an ancestor of } u \text{ and } i > 0) \)
6. \( \text{then } [u := u; f := l]; \)
7. \( l := l + 1; \)
8. \( \text{if } j > 0 \) then return \( j, p_i \text{'s parent;} \)
9. \( \text{if } f = 0 \) then return \( 0, \delta(p_i); \)
10. \( \text{let } w_v = w \text{ be the children of } u; \)
11. (*supplement checking*)
12. while \((j < s \text{ and } l \leq k)\)
13. \( i \leftarrow <v, u := A(G, < T_1 >, ..., G, v), v>; \)
14. \( \text{if } (w = v \text{ and } i > 0) \) then \( j := j + i; \)
15. \( l := l + 1; \)
16. return \( j, u; \)

As in the basic version of the β-function, 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 < Ti, ..., Ti > embeds < Pj, ..., Pk >. In this process, by the first A-function call we have cut v, which is the same as the cut propagated to β(< Ti, ..., Ti >, G, v) while the subsequent A-function calls the cuts are horizontally propagated.

In the second **while-loop**, we do the supplement computation, but conducted only when the strict SCC-condition is satisfied.

As in Function 3, variable f is used to record the first non-zero point which is not a descendant of any other non-zero point. Initially, it is set 0. Therefore, if no non-zero point higher than cut v is found, we must have \( f = 0 \) after the main computation. Thus, only when \( j = 0 \) and \( f > 0 \), the strict SCC-condition is satisfied and the supplement computation will be performed. (See lines 8 and 9.)

In Appendix III, we will give a sample trace of the improved algorithm when applied to the tree T and the forest G shown in Fig. 8. In appendix IV, its correctness is formally proven.

## 7 Computational Analysis

In this section, we mainly analyze the computational complexities of the improved algorithm discussed in Section 6. First, we discuss its space requirement in 7.1. Then, in 7.2, its worst time complexity is analyzed.

### 7.1 Space Complexity

The space overhead of our algorithm is mainly composed of two parts. One part is the intervals associated with the nodes in both T and G to check reachability. It is obviously bounded by \( O(|T| + |G|) \). The other part is the space used for storing recursive calls of functions in the system stack.
But it must be proportional to the size of a longest recursive function call chain \( L \). (To know what is that, see Fig. 12(a), which is a chain corresponding to lines 1–8 in the sample trace given in Appendix III.) This chain is produced when applying our algorithm to the target tree and pattern forest shown in Fig. 8. Therefore, to know the size of the second part, we need to estimate \( L \)'s length.

We first note that each recursive call needs only a constant space. It is because a tree \( T \) can be always referred to by its root \( t \) while a forest \( <T_1>, \ldots, T_k> \) (resp. \( <P_1>, \ldots, P_q> \) ) by a pair \( <t, k> \) (resp. \( <p, q> \) ). It is because any forest involved in a recursive call is always made up of a set of subtrees rooted respectively at a set of consecutive child nodes (starting from a specific child to the last child) of a certain node in \( T \) or in \( G \). Thus, \( \alpha(T, P, v) \) can be simply represented by \( \alpha(t, p, v) \) while \( \beta(T_1>, T_2>, P_1>, \ldots, P_q>, v) \) by \( \beta(t, k, p, q, v) \), which indicates that only a constant space is needed to record a recursive call.

Furthermore, each \( A \)-function call is always followed by an \( \alpha \)-function call or a \( \beta \)-function call in a function call chain, as demonstrated in Fig. 12(a). So we can merge each \( A \)-function call into its successor to simplify analysis and view \( L \) as a chain containing only two kinds of function calls, i.e., the calls of the form \( \alpha(t, p, v) \) and \( \beta(t, k, p, q, v) \). Thus, we can simply divide \( L \) into two sequences: \( L_\alpha \) and \( L_\beta \) such that \( L_\alpha \) contains only the \( \alpha \)-function calls while \( L_\beta \) only the \( \beta \)-function calls. \( L_\alpha \) and \( L_\beta \) are called the \( \alpha \)-subchain and \( \beta \)-subchain of \( L \), respectively. For example, the chain shown in Fig. 12(a) can be divided into an \( \alpha \)-function call chain and a \( \beta \)-function call chain, as shown in Fig. 12(b) and (c), respectively.

\[
A(T, G, p_{111})
\]

\[
\beta(<T_1>, T_2>, G, p_{111})
\]

\[
\beta(<T_1>, T_2>, G, p_{111})
\]

\[
\alpha(T_1, P_1, p_{111})
\]

\[
A(T_1, P_1, p_{111})
\]

\[
\alpha(T_1, P_1, p_{111})
\]

\[
\beta(<T_{12}, P_{112}, P_{111}>, P_{111})
\]

\[
\beta(<T_{12}, P_{112}, P_{111}>, P_{111})
\]

\[
A(T_1, P_{111}, P_{111})
\]

\[
\beta(<T_{11}, P_{112}, P_{111}>, P_{111})
\]

\[
\beta(<T_{11}, P_{112}, P_{111}>, P_{111})
\]

\[
A(T_{11}, P_{111}, P_{111})
\]

\[
\alpha(T_1, P_1, p_{111})
\]

\[
\alpha(T_1, P_1, p_{111})
\]

\[
\beta(<T_1, T_2>, G, p_{111})
\]

Through the \( A \)-function call, \( \alpha < \alpha' \), \( \beta \), \( \gamma > \) is invoked. Therefore, \( p' \) is a child of \( p \).

We illustrate this process in Fig. 13(a). This shows that if a node \( t \) in \( T \) is checked against two consecutive nodes \( p \) and \( p' \) in \( G \) along a recursive call chain, \( p \) must be the parent of \( p' \).

Now we consider (ii). If \( p = p' \), it shows that \( G[p] \) is involved in a second call of the \( \alpha \)-function, which happens when Case (1-3) is satisfied, i.e., when \( |T| \leq |P| + z \), where \( z \) is the size of the subtree rooted at \( p \)'s right sibling, but \( |T| > |P| \), \( h(t) \geq h(p) \), and \( |P| \neq |label(p)| \). In this case, we will check the forest containing the subtrees respectively rooted at the children of \( t \) against \( G[p] \) by calling the \( \beta \)-function (see line 5 in Function 5), through which \( \alpha < \alpha' \), \( \beta \), \( \gamma > \) is invoked. See Fig. 13(b) for illustration, which shows that if a node \( p \) in \( G \) is checked against two consecutive nodes \( t \) and \( t' \) along a recursive call, \( t \) must be the parent of \( t' \). This completes the proof. \( \square \)

From Lemma 3, we can see that \( |L_\alpha| \) is bounded by \( O(h_t + h_c) \).

In a similar way, we can prove that that \( |L_\beta| \) is also bounded by \( O(h_t + h_c) \). Therefore, \( |L| = |L_\alpha| + |L_\beta| \) is in the order of \( O(h_t + h_c) \).

Proposition 2 The space used by Algorithm \( A(T, G, \rho(G)) \) is bounded by \( O(|T| + |G|) \).

Proof. See the above analysis. \( \square \)

## 7.2 Time Complexity

Now we analyze the time complexity of the algorithm. This will be done in two steps. First, we show that the time used by the improved algorithm is bounded by \( O(|T| \cdot h_c) \). Then, we further demonstrate that the time requirement is also bounded by \( O(|T| \cdot \min\{h_c, |\text{leaves}(G)|\}) \). This indicates that the time complexity of our algorithm is \( O(|T| \cdot \min\{h_c, |\text{leaves}(G)|\}) \).

We first notice that in a supplement checking no further supplement checking will be conducted. It is because in a supplement checking of the form \( \beta(<P_{13}, \ldots, T_2>, G[w_1], \ldots, G[w_2], u) \) we always have \( u = w_1 \), by which any further supplement checking is effectively blocked.

In order to see that the time complexity is bounded by \( O(|T| \cdot h_c) \), we analyze, in the worst case, how many \( \beta \)-function calls each node \( t \) in \( T \) can be involved in.

Let \( t' \) be a node in \( T \). Let \( t'' \) be a child of \( t' \). Assume that in the computation there exists a \( \beta \)-function call of the form \( \beta(<T_1>, \ldots, T_2>, \ldots, P_{13}, \ldots, P_1>, u)' \), in whose execution \( \beta(<T_1>, \ldots, T_2>, \ldots, u)'' \) is invoked (possibly through an \( A \)-function call invoked during the execution of \( \beta(<T_1>, \ldots, T_2>, \ldots, u)' \); see line 3 in FUNCTION 6.) Then, \( u' \) and \( u'' \) can be in one of three relationships:
1. $u'' \sim u'$. In this case, $t''$ can possibly be involved in a supplement checking, but $t'$ definitely not since the left corner of $\beta(T[t'])$, $<p_t, ..., p_r, u'>$, must be higher than $u'$.  
2. $u'' = u'$. In this case, $t''$ will definitely not be involved in a supplement checking. It is because in the execution of $\beta(<T[p], ..., <p_t, ..., p_r, u'>)$, the node corresponding to the first highest non-zero point (if any) can only be $t''$ or a node to the right of $t''$ (see line 4 in FUNCTION 6.  
   However, $t''$ may be involved in a supplement checking, depending on the results of checking the subtrees rooted at its left siblings against $<p_t, ..., p_r, u'>$ as well as the return value of $\beta(<T[p], ..., <p_t, ..., p_r, u'>)$ itself.
3. $u'' \not\sim u''$ (more exactly, $u''$ is the first child of $u'$). This happens when through an $A$-function call, a $\beta$-function is invoked, by which the cut is downgraded (see line 5 in FUNCTION 4); or an $\alpha$-function is invoked, in which we have Case (1-5-i) satisfied and the cut is also downgraded (see line 7 in FUNCTION 5.) In these cases, both $t'$ and $t''$ may be involved in a supplement checking. Especially, during the supplement computation involving $t'$, $t''$ can possibly be involved in another $\beta$-function call once again.

Obviously, (3) is the worst case, by which the number of $\beta$-function calls $t''$ is involved in is maximized. Now, we observe the parent of $t'$ and assume that in the execution of $\beta(<T[p], ..., <p_t, ..., p_r, u'>)$, $\beta(T[t'])$, $<p_t, ..., p_r, u'>$ is invoked. Repeating the above analysis, we can see that if $u \sim u'$, this happens when through an $A$-function call, a $\beta$-function is also involved in a supplement checking. This shows that if we have $u \sim u' \sim u''$ $t$ can be involved in two $\beta$-function calls, $t'$ in three $\beta$-function calls, and $t''$ in four $\beta$-function calls. In general, the number of $\beta$-function calls, in which a certain node $t$ in $T$ is involved, must be bounded by $h_c + 1$ since any sequence of cuts: $u_1 \sim u_2 \sim \cdots \sim u_t$, in $G$ cannot contain more than $h_c$ nodes, and any recursive call with $t$ involved corresponds to a cut at a different level.

In the terms of the above analysis, we have the following lemma.

**Lemma 4** The time complexity of the algorithm $A(T, G, \rho(G))$ (FUNCTION 4 in Section 6) is bounded by $O(|T| \cdot h_c)$.  
\textbf{Proof.} We need to show that any node $t$ in $T$ can also be involved in at most $O(h_c)$ $A$-function calls. For this purpose, we notice that between any two consecutive $A$-function calls along a function call chain we can have at most an $\alpha$-function and a $\beta$-function. This property can be observed by analyzing the basic algorithm given in Section 5, by which we can clearly see three kinds of $A$-to-$A$ (from an $A$-function call to a next $A$-function call) chains:

$A \rightarrow \alpha \rightarrow A$ (see line 2 in FUNCTION 1, line 4 in FUNCTION 2)

$A \rightarrow \alpha \rightarrow \beta \rightarrow A$ (see line 2 in FUNCTION 1, line 5, 7 in FUNCTION 2, line 3, 13 in FUNCTION 3)

$A \rightarrow \beta \rightarrow A$ (see line 3 in FUNCTION 1, line 3, 13 in FUNCTION 3)

Clearly, for the second and third kinds of $A$-to-$A$ chains, the number of $A$-function calls is bounded by the number of $\beta$-function calls. For the first kind of $A$-to-$A$ chains, the number of $A$-function calls for each $t$ is also bounded by $(h_c + 1)$ since each of such chains happens when $|T[t]| < |G[p]|$ or $h(t) < h(p)$ (Case 1-2). In this case, a function call of the form $A(T, <p_t, ..., p_r, u>)$ will be conducted (see line 4 in FUNCTION 5). Thus, if $t$ is checked for a second time, it must be against a descendant of $p$. So, the lemma holds.

**Lemma 5** The time complexity of the algorithm $A(T, G, \rho(G))$ (FUNCTION 4 in Section 6) is bounded by $O(|T| \cdot |\text{leaves}(G)|)$.

\textbf{Proof.} To show that the time complexity of the algorithm is also bounded by $O(|T| \cdot |\text{leaves}(G)|)$, we observe the worst case, i.e., case 3 (the case $u'' \sim u'$ in the above discussion) again and assume that $t'$ is involved in a supplement checking (referred to as $SC_1$) and $t''$ is involved in another supplement checking (referred to as $SC_2$). Then, by $SC_1$ $T[t']$ will be checked against a forest containing a set of subtrees respectively rooted at some right siblings of $u'$ while by $SC_2$ $T[t'']$ will be against a forest $G'$ containing a set of subtrees respectively rooted at some right siblings of $u''$, as illustrated in Fig. 14. Thus, if $t'$ is checked for a second time during the $SC_1$ involving $t'$, it must be checked against a node which is to the right of $G'$. This shows that the number of nodes in $G$, which are checked against $t'$ is bounded by $O(|\text{leaves}(G)|)$. Therefore, for whole $T$, the number of checkings is bounded by $O(|T| \cdot |\text{leaves}(G)|)$.  

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure14}
\caption{Illustration for supplement checking}
\end{figure}

In Fig. 14, we illustrate that when $t'$ is checked during $SC_1$, $t''$ can also be checked once again, but against some node to the right of a forest $G'$, which is checked during $SC_2$.

In the above discussion, we should remark that in the proof of Lemma 4, we use cuts to explain that any node in $T$ can be involved in at most $O(h_c)$ function calls while in the proof of Lemma 5, we show that any node in $T$ can be checked against at most $O(|\text{leaves}(G)|)$ nodes in $G$.

From Lemma 3 and 4, we immediately get the following proposition.

**Proposition 3** The time complexity of $A(T, G, \rho(G))$ is bounded by $O(|T| \cdot \min\{|h_v|, |\text{leaves}(G)|\})$.

\textbf{Proof.} This can be derived from Lemma 4 and 5.

### 8 Experiments

In our experiments, we have tested altogether four different methods:

- Kilpeläinen’s algorithm [9],
- Chen’s algorithm [3],
- Bille’s algorithm [2], and
- Ours

All the four methods are implemented in C++, compiled by GNU make utility with optimization of level 2. In addition, all of our experiments are performed on a 64-bit Ubuntu operating system, run on a single core of a 2.40GHz Intel Xeon E5-2630 processor with 32GB RAM.

7.1 Data Sets

The data sets used for the tests are TreeBank data set, DBLP data set (both of them can be found in U of Washington XML Repository, http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata), and a synthetic XMark data set (The XML-benchmark project, http://monetdb.cwi.nl/xml). The TreeBank data set is a real data set with a narrow and deeply recursive structure that includes multiple recursive elements. The DBLP data set is another real data set with high similarity in structure. It is in fact a wide and shallow document. The XMark (with scaling factors 1, 3, and 5) is a well-known benchmark data set, by which a document generator xmlgen is provided, used for scalability analysis. The important parameters of these data sets are summarized in Table 1.

Table 1: Data sets for experiment evaluation

<table>
<thead>
<tr>
<th>TreeBank</th>
<th>DBLP</th>
<th>XMARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size (MB)</td>
<td>82</td>
<td>127</td>
</tr>
<tr>
<td>No. of nodes (million)</td>
<td>2.43</td>
<td>3.33</td>
</tr>
<tr>
<td>Max/average height</td>
<td>36/7.9</td>
<td>8/2.9</td>
</tr>
</tbody>
</table>

7.2 Test Results

For each data set, we have tested two groups of pattern trees. For the first group, we generate pattern trees by randomly selecting subtrees of 100 nodes from the target tree. For the second group, each time we randomly select 200 nodes, but with different heights. We record the numbers of label comparisons and elapsed times. For each execution, an average of 100 measurements is taken.

- Tests on TreeBank

In Fig. 15(a) and (b), we show the numbers of label comparisons and the times spent on different execution, respectively. From Fig. 15(a), we can see that our method outperforms all the other three algorithms uniformly, and the Kilpeläinen’s has the worst performance. We can also see that the Bill’s and Chen’s are comparable. For small sized pattern trees, the Bille’s is slightly better than Chen’s. However, as the size of pattern trees increases, the Chen’s works better. It is because by the Bill’s algorithm extra time is used to check and remove useless data generated to record intermediate results to reduce space overhead and this part of time matters for large pattern trees.

In Fig. 16(a) and (b), we demonstrate the result of the second group test. From Fig. 16(a), we can see that the number of label comparisons made by our method linearly depends on the height of pattern trees. But the number of label comparisons made by the Bille’s and Chen’s algorithms decreases as the height increases. The Kilpeläinen’s algorithm is not sensitive to the height of patterns trees. Again, the time spent by the Kilpeläinen’s algorithm is much worse than all the other three algorithms.

In Fig. 17, we show the space overhead of the tested method over the treebank data. From this figure, we can see that our method uses much less space than the other three methods. Among them, the Kilpeläinen’s is the worst while the Bille’s is best and a little bit better than the Chen’s. In fact, the Bille’s and the Chen’s methods work almost in the same way. The main difference is that in the Chen’s method, the siblings of a node in a pattern P are always handled from left to right while in the Bille’s method, the so-called heavy child is always handled first. By a heavy child, we mean a node v such that P[v] has the most leaf nodes. The other difference is that by the Bille’s method only the deep occurrences of P in a target T (i.e., the nodes u at low levels in T such that T[u] contains P) is checked. These arrangements can reduce somehow the size of intermediate results, but cannot bring down the space overhead by an order of magnitude.
quite short (on average their lengths are bounded by 3) and the number of leaf nodes is large and comparable to the whole size of the tree itself.

In Fig. 19(a), we show the space usage of the tested method over the DPLP data. From this figure, we can see that three methods, except ours, have almost the same space overhead. The reason for this is that the DPLP is a very shallow tree as mentioned above and the randomly generated pattern trees are also shallow. So the second difference of the Bille’s method from the Chen’s brings no significant improvement. Again, our method uses much less space than all of them.

The Bill’s is a little better than the Chen’s. Again, our method is uniformly better than all the other algorithms. However, as the height of patterns increases, we can clearly see the increment of the number of label comparisons. It confirms to our theoretical analysis. But for the Bille’s and Chen’s, the number of label comparisons is reduced with higher patterns. It is because for patterns with a fixed number of nodes, the higher they are, the less leaf nodes they may have.

In Fig. 20(b), we demonstrate the space overhead of the tested method over the XMark data. This figure shares the same flavour as Fig. 17, but all the methods use much less space than the treebank data.

9 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 $O(|T| \cdot \min\{|h_T|, |\text{leaves}(P)|\})$ time and $O(|h_T + h_P|)$ space (besides the space for storing $T$ and $P$ themselves), where $T$ and $P$ are a target and a pattern tree (forest), respectively; $h_T$ ($h_P$) is the height of $P$ (resp. $T$) and
leaves(P) is the set containing all the leaf nodes of P. The critical concepts of our algorithm are the left-corner and cuts, 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. In addition, our algorithm is more efficient than any existing method for the problem by using cuts to skip over useless computations.

---

10 REFERENCES


Dr. Yangjun Chen got his PhD in Computer Science from the University of Kaiserslautern, Germany, in 1995. He is now a professor in Dept. Applied Computer Science, University of Winnipeg, Canada. He has about 200 publications in Computer Science and Computer engineering.

Mr. Yibin Chen received his BS and master degree from the Department of Electrical and Computer Engineering, University of Waterloo, and in the Department of Electrical and Computer Engineering, University of Toronto, Canada, respectively. Now he is a software engineer.
## Appendix I Symbols and Notations

In this appendix, we summarize all the symbols and notations used throughout the paper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = &lt;t, T_t, ..., T_i&gt;$</td>
<td>target tree with root $t$ and its direct subtrees $T_{T_t}, ..., T_{T_i}$</td>
</tr>
<tr>
<td>$G = &lt;P_{T_t}, ..., P_{T_i}&gt; (q \geq 0)$</td>
<td>pattern, which is a forest containing subtrees $P_{T_t}, ..., P_{T_i}$</td>
</tr>
<tr>
<td>$V(T)$</td>
<td>set of nodes in $T$</td>
</tr>
<tr>
<td>$E(T)$</td>
<td>set of edges in $T$</td>
</tr>
<tr>
<td>$h_T$</td>
<td>height of $T$</td>
</tr>
<tr>
<td>leaves($P$)</td>
<td>all leaf nodes of $P$</td>
</tr>
<tr>
<td>$d(v)$</td>
<td>outdegree of node $v$</td>
</tr>
<tr>
<td>$G[v]$</td>
<td>subtree rooted at $v$</td>
</tr>
<tr>
<td>$v &lt; v'$</td>
<td>$v$ is to the left of $v'$</td>
</tr>
<tr>
<td>$v \leq v'$</td>
<td>$v &lt; v'$ or $v = v'$</td>
</tr>
<tr>
<td>$v \sim u$</td>
<td>$v$ is a proper ancestor of $u$</td>
</tr>
<tr>
<td>$\varphi: V(G) \rightarrow V(F)$</td>
<td>an injective function mapping nodes in $V(G)$ to nodes in $V(F)$</td>
</tr>
<tr>
<td>$\rho(G)$</td>
<td>left-most leaf node in $G$</td>
</tr>
<tr>
<td>$v_G$</td>
<td>virtual root of $G$</td>
</tr>
<tr>
<td>$&lt;[i, j], v&gt;$</td>
<td>an interval rooted at $v$, to represent an ordered forest $&lt;G[v_i], ..., G[v_j]&gt;$, made up of a series of subtrees rooted at the children of $v$: $v_{T_t}, ..., v_{T_i}$, respectively.</td>
</tr>
<tr>
<td>$&lt;i, v&gt;$</td>
<td>abbreviation of $&lt;[1, i], v&gt;$. If $v$ is $v_G$ or a node on the left-most path of $P_{T_t}$, it is called a left corner of $G$.</td>
</tr>
<tr>
<td>$\leq i, v \equiv u$</td>
<td>level equal, i.e., $v = u$</td>
</tr>
<tr>
<td>$&lt;i, v&gt; \sim u$</td>
<td>$&lt;i, v&gt;$ is higher than $u$, i.e., $v \sim u$</td>
</tr>
<tr>
<td>$&lt;i, v&gt; \gg u$</td>
<td>$&lt;i, v&gt;$ is higher than or level equal to $u$</td>
</tr>
<tr>
<td>$A(T, G)$</td>
<td>a checking of $G$ against $T$, returning a highest and widest left-corner $&lt;i, v&gt;$ in $G$</td>
</tr>
<tr>
<td>$\alpha(T, P)$</td>
<td>a function returning a highest and widest left-corner in $P$, which can be embedded in $T$, where $P$ is a subtree in $G$.</td>
</tr>
<tr>
<td>$\beta(T, ..., T_i, G)$</td>
<td>a function returning a highest and widest left-corner in $P$, which can be embedded in $&lt;T_{T_t}, ..., T_{T_i}&gt;$.</td>
</tr>
<tr>
<td>$\delta(v)$</td>
<td>a link to the left-most leaf node in $G[v]$</td>
</tr>
<tr>
<td>$\Delta(v)$</td>
<td>a set of nodes $x$ such that for each $v \in x$, $\delta(v) = v'$</td>
</tr>
<tr>
<td>$h(t)$</td>
<td>height of $T[t]$</td>
</tr>
</tbody>
</table>

### Main computation
- $A(T, G, v)$: part of $\alpha$-function execution process
- $\beta(T, ..., T_i, G, v)$: part of $\beta$-function execution process

### SCC-condition
- $A(T, G, v)$: variant of $A(T, G)$, where $v$ is used as a cut such that only a highest and widest left-corner in $G$ is returned if it can be embedded in $T$ and higher than $v$. Otherwise, it returns $\langle 0, \rho(G) \rangle$. |
- $\alpha(T, P, v)$: variant of $\alpha(T, P)$, where $v$ is used as a cut such that only a highest and widest left-corner in $P$ is returned if it can be embedded in $T$ and higher than $v$. Otherwise, it returns $\langle 0, \rho(G) \rangle$. |
- $\beta(T, ..., T_i, G, v)$: variant of $\beta(T, ..., T_i, G)$, where $v$ is used as a cut such that only a highest and widest left-corner in $G$ is returned if it can be embedded in $<T_{T_t}, ..., T_{T_i}>$ and higher than $v$. Otherwise, it returns $\langle 0, \rho(G) \rangle$. |

### Strict SCC-condition
- $\alpha(T, P, v)$: variant of $\alpha(T, P)$, where $v$ is used as a cut such that only a highest and widest left-corner in $P$ is returned if it can be embedded in $T$ and higher than $v$. Otherwise, it returns $\langle 0, \rho(G) \rangle$. |
Appendix II Sample Trace of Basic Algorithm

In the above sample trace, since both F and G are forests (general case 2), in the execution of A(F, G) β(F, G) will be invoked, in which we will call A(T1, <p1, p2>, A(T2, <p3, p4>, and A(T3, <p1, p2>) in turn (see lines 3 – 20, lines 21 – 70, and lines 71 – 88 in the sample trace). We call A(T2, <p1, p2>, after A(T1, <p1, p2>) because the return value of A(T1, <p1, p2>) is <1, p1> (see line 20), showing that T1 is not able to embed P1. Since T2 is not able to include P1, T2 contains only <1, p1>, see line 70), either, A(T3, <p1, p2>) will be invoked (see lines 71 – 88), whose return value is <0, p1>,<1>. So, a supplement checking will be carried to check after T3 > contains a single tree. Call A(T3, <p1, p2>, (see lines 89 – 116). It is because A(T3, <p1, p2>) returns <1, p1>, higher than both <1, p1> (return value of A(T3, <p1, p2>)) and <0, p1> (return value of A(T3, <p1, p2>)) and we need to know whether the left corner <1, p1> can be expanded by the supplement checking, in which we will first check T2 against after <p12, p12> (see lines 89 – 112). Since it returns <1, p1> (showing that T2 covers P12), we will call A(T3, <p13>, in a next step (see lines 113 – 116), which also returns <1, p1> (showing that T3 covers P13) Therefore, the whole process of β(F, G) returns <3, p1>, showing that T includes <p11, p12, p13, p14>.

In the above sample trace, if t123 were something other than an ‘a’, then A(T12, <p1, p2>, 2) would return <0, p11> and the supplement checking (after A(T13, <p1, p2>) returns <1, p1>) would check <T12, T13> against <p112> since T11 includes P11.
(1-2) holds in the execution of \alpha \). Call \( A(.) \) (see line 4 in \( \alpha \)). 
\[ A(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
\[ \beta \langle T_2 \alpha, P_{112} \rangle \]
\[ \alpha(T_2, P_{112}) \]
Appendix III Sample Trace of Improved Algorithm

Below we trace the execution of the improved algorithm when applied to the tree T and the forest G shown in Fig. 11. As can be seen, this is a much shorter process (than the sample trace of the basic algorithm when applied to the same target and pattern trees), by which almost the whole computation of A(T_2, G) and A(T_3, G) are discarded by using cuts.

First, we notice that the return value of A(T_1, G, p_{11}) is <1, p_1> (see line 20.) So the cut transferred to A(T_2, G, p_1) is p_1. Then, we will have the following recursive calls (see lines 21, 22, and 23 in the sample trace):

\[ A(T_2, G, p_1) \rightarrow \alpha(T_2, P_{11}, p_1) \rightarrow A(T_2, P_{11}, P_{12}, p_1). \]

Since p_1's parent is p_1 (instead of an ancestor of P_1), A(T_2, P_{11}, P_{12}, p_1) cannot return a useful left corner.

\[
\begin{align*}
1. & \quad A(T, G, p_{11}) \\
2. & \quad \beta(T_2, T_3, p_{11}, G, p_{11}) \\
3. & \quad A(T, G, p_{11}) \\
4. & \quad \alpha(T_1, P_{11}, p_{11}) \\
5. & \quad A(T_1, P_{11}, P_{12}, P_{13}, p_{11}) \\
6. & \quad \beta(T_1, T_2, p_{11}, P_{12}, P_{13}, p_{11}) \\
7. & \quad A(T_1, P_{12}, P_{13}, p_{11}) \\
8. & \quad \alpha(T_1, P_{12}, P_{13}, p_{11}) \\
9. & \quad \text{return} <1, p_{11}> \\
10. & \quad \text{return} <1, p_{11}> \\
11. & \quad A(T_2, P_{12}, P_{13}, p_{11}) \\
12. & \quad \text{return} <1, p_{11}> \\
13. & \quad \text{return} <2, p_1> \\
14. & \quad \text{return} <1, p_1> \\
15. & \quad \text{return} <1, p_1> \\
16. & \quad \text{return} <1, p_1> \\
17. & \quad A(T_2, G, p_1) \\
18. & \quad \alpha(T_2, P_{11}, p_1) \\
19. & \quad A(T_2, P_{11}, P_{12}, P_{13}, p_1) \\
20. & \quad \text{return} <0, p_{11}> \\
21. & \quad \text{return} <0, p_{11}> \\
22. & \quad \text{return} <0, p_{11}> \\
23. & \quad \text{return} <0, p_{11}> \\
24. & \quad \text{return} <0, p_{11}> \\
25. & \quad \text{return} <0, p_{11}> \\
26. & \quad \text{return} <0, p_{11}> \\
27. & \quad \text{return} <0, p_{11}> \\
28. & \quad \text{return} <0, p_{11}> \\
29. & \quad \text{return} <0, p_{11}> \\
30. & \quad \text{return} <0, p_{11}> \\
31. & \quad \text{return} <0, p_{11}> \\
32. & \quad \text{return} <0, p_{11}> \\
33. & \quad \text{return} <0, p_{11}> \\
34. & \quad \text{return} <0, p_{11}> \\
35. & \quad \text{return} <0, p_{11}> \\
36. & \quad \text{return} <0, p_{11}> \\
37. & \quad \text{return} <0, p_{11}> \\
38. & \quad \text{return} <0, p_{11}> \\
39. & \quad \text{return} <0, p_{11}> \\
40. & \quad \text{return} <0, p_{11}> \\
41. & \quad \text{return} <0, p_{11}> \\
42. & \quad \text{return} <0, p_{11}> \\
43. & \quad \text{return} <0, p_{11}>
\end{align*}
\]

So the corresponding computation needn’t be performed and we simply set its return value to be <0, p_{11}> (see line in the modified A-function; also see line 24 in the sample trace.)

In a next step, we will call A(T_3, G, p_1) and the cut transferred to it is still p_1. Accordingly, we have the following recursive calls (see lines 27, 28, and 29 in the sample trace):

\[ A(T_3, G, p_1) \rightarrow \alpha(T_3, P_{11}, p_1) \rightarrow A(T_3, P_{11}, P_{12}, P_{13}, p_1). \]

For the same reason, A(T_3, P_{11}, P_{12}, P_{13}, p_1) will not be carried out, either, but with <0, p_{11}> returned.

From the above explanation, we can see that the modified algorithm will return the same result as the basic version, but require much less running time.
step-by-step trace:  

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.</td>
<td>$A(T_x, &lt;P_i&gt;, p_{i3})$</td>
<td>in the 1st while-loop of line 3 in FUNCTION 4.</td>
</tr>
<tr>
<td>46.</td>
<td>$\alpha(T_x, P_{i3}, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>47.</td>
<td>$\beta(T_x, &lt;P_i&gt;, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>48.</td>
<td>$A(T_{zi}, &lt;P_{i3}&gt;, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>49.</td>
<td>$\alpha(T_{zi}, &lt;P_{i3}&gt;, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>50.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>51.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>52.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>53.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>54.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>55.</td>
<td>return $&lt;0, p_{i3}&gt;$</td>
<td></td>
</tr>
<tr>
<td>56.</td>
<td>return $&lt;1, p_i&gt;$</td>
<td></td>
</tr>
<tr>
<td>57.</td>
<td>$A(T_{zi}, &lt;P_{i3}&gt;, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>58.</td>
<td>$\theta(T_{zi}, P_{i3}, p_{i3})$</td>
<td></td>
</tr>
<tr>
<td>59.</td>
<td>return $&lt;1, p_i&gt;$</td>
<td></td>
</tr>
<tr>
<td>60.</td>
<td>return $&lt;1, p_i&gt;$</td>
<td></td>
</tr>
<tr>
<td>61.</td>
<td>return $&lt;3, p_i&gt;$</td>
<td></td>
</tr>
<tr>
<td>62.</td>
<td>return $&lt;3, p_i&gt;$</td>
<td></td>
</tr>
</tbody>
</table>

**Appendix IV Correctness of the Algorithm with cuts**

In this Appendix, we prove the correctness of the Algorithm $A(T, G, v)$, where $T = \langle t_i; T_i >, G = \langle P_i; ..., P_i >$. Initially, $v$ is set to be $p(G)$ and is trivially correct.

In the subsequent execution, the cut will be changed and transferred from a function call to another. To see that it is always correctly conducted, we need to examine three kinds of $A$-to-$A$ chains defined in the proof of Lemm 4:

- $A \rightarrow \alpha \rightarrow A_
- A \rightarrow \alpha \rightarrow \beta \rightarrow A$, and
- $A \rightarrow \beta \rightarrow A$.

What we want to do is to demonstrate that by each of these chains the cut is both correctly changed and transferred.

First, we notice that by $A \rightarrow \alpha$ the cut transfer is obviously correct. (See line 3 in FUNCTION 4.)

Next, by $A \rightarrow \beta$, we distinguish between two cases:

1. **label($t$) = label($v$)**, where $v$ is the cut by the $A$-function call. (See line 4 in FUNCTION 4.)

   - **ii) label($t$) $\neq$ label($v$)**. (See line 5 in FUNCTION 4.)

   In Case (i), the cut $v$ for the $\beta$-function call is downgraded to $v$'s first child. It is because if $\langle T_i, ..., T_i >$ is able to cover the forests composed of all the subtrees respectively rooted at all the children of $v$, $T$ includes $G[v]$. Downgrading $v$ to $v$'s first child will let the corresponding computation get through.

   In Case (ii), the cut for the $\beta$-function call is still $v$ since any left corner returned by the $\beta$-function call will not be used by the subsequent computation if it is lower than $v$.

   Concerning the correctness of the cut transfer by $\alpha \rightarrow A, \alpha \rightarrow \beta$, and $\beta \rightarrow A$, we need to repeat the discussion on Cut Propagation in $\alpha$-function, as well as Cut Propagation in $\beta$-function, in Section 6. By these discussions, we can see that both the cut change and cut transfer are correctly done in all the cases. Therefore, by each of the three chains the cut is either correctly changed or correctly transferred. So, we have the following proposition.

**Proposition 4** Let $T = \langle t_i; T_i >, G = \langle P_i; ..., P_i >$. The return value of $A(T, G, v)$ is the highest and wildest left-corner in $G$, which can be embedded in $T$ and is higher than $v$, or $<0, p(G)>$. □