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 $O(|T| \cdot min\{h_P, |leaves(P)|\})$, with O(|*T*| + |*P*|) space being used, where h_P (h_T) represents the height of *P* (resp. *T*) and leaves(*P*) stands for the set of the leaves of *P*. Up to now the best algorithm for this problem needs Θ(|*T*|·|*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 Σ 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 *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.

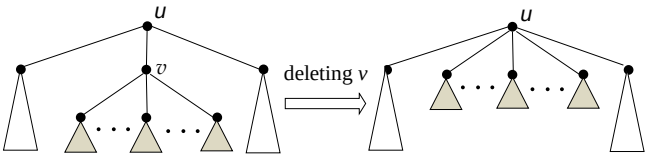


Figure 1. Illustration of node deletion

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

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

we have $T_k = P$? If this is the case, *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.

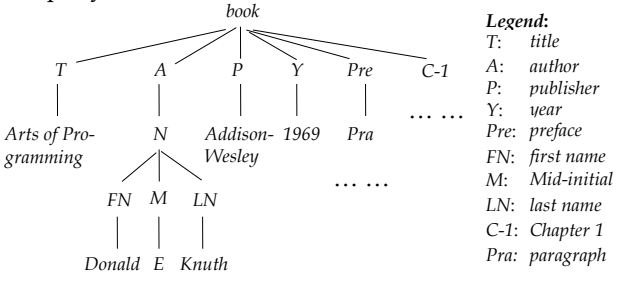


Figure 2. A XML document (target) tree

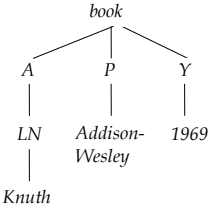


Figure 3. A pattern tree

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

- *Y. Chen is with the Dept. Applied Computer Science, Uni. of Winnipeg, Canada. E-mail: y.chen@uwinnipeg.ca.*
- *Y. B. Chen is with Intel Corp. E-mail: chenyibin@gmail.com.*

The paper is a modification and extension of two conference papers: Y. Chen and Y. B. Chen, Tree Inclusion Checking Revisited, 5th Int.Conf. DATA2013, July 24 – 26, 2015, Lisbon, Portugal, pp. 301 – 308; Chen and Y. B. Chen, On the Tree Inclusion Problem,Int. Conf. ICCNCE2013, May 23-24, Beijing, China, pp. 131 – 135.

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 $\Theta(|T| \cdot |\text{leaves}(P)|)$ 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 $O(|T| \cdot min\{h_P, |\text{leaves}(P)|\})$, and $O(|T| + |P|)$, respectively, where h_P (h_T) is the height of P (resp. T), 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 $O(|T| \cdot |P|)$ time and space being used. This computational complexity has be 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 $O(|T| \cdot |P|)$. 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 $O(|T| \cdot |\text{leaves}(P)|)$ time and $O(|\text{leaves}(P)| \cdot min\{h_T, |\text{leaves}(T)|\} + |T| + |P|)$ 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 $O(|T| \cdot |P|)$. Recently, Bille and Gørtz presented a space-economical algorithm [2]. Its space overhead is $O(|T| + |P|)$, but with its time complexity bounded by

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

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 $O(|T| \cdot |P|)$ time while the algorithm in [8] requires $O(|T| \cdot d \cdot h_p)$ time, 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 $O(|T| \cdot min\{h_P, |\text{leaves}(P)|\})$.

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 $u \rightsquigarrow v$ 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, $v_3 \rightsquigarrow v_{10}$, since I_{v_3} = [1, 5], $I_{v_{10}}$ = [3, 3], and [3, 3] ⊂ [1, 5]; but v_9 is not reachable from v_2 since I_{v_2} = [10, 13], I_{v_9} = [1, 2], and [1, 2] ⊄ [10, 13].

Let I = [l, r] be an interval. We will refer to l and r in I as $I.l$ and $I.r$, 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: $I \subset I'$, $I' \subset I$, $I.r < I'.l$, or $I'.r < I.l$. □

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 $I_u.r < I_v.l$.

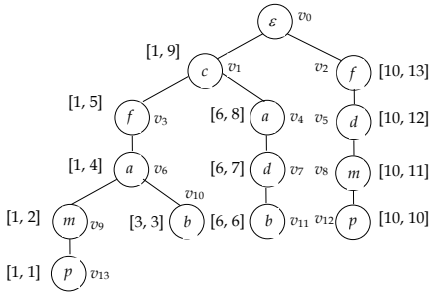


Figure 4. Illustratin for tree encoding

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

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

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) $v \rightsquigarrow u$ iff $\varphi(v) \rightsquigarrow \varphi(u)$, 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., $I_v.r < I_u.l$ iff $I_{\varphi(v)}.r < I_{\varphi(u)}.l$. (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.

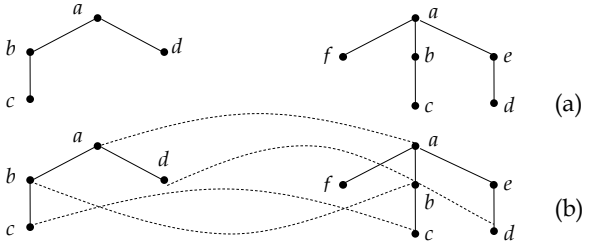


Figure 5: (a) The tree on the left can be included in the tree on the right by deleting the nodes labeled: f and e; (b) the embedding corresponding to (a).

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. 5(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 chldren) 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_1, ..., P_q> ($q \geq 0$) be a forest (as illustrated in Fig. 6).

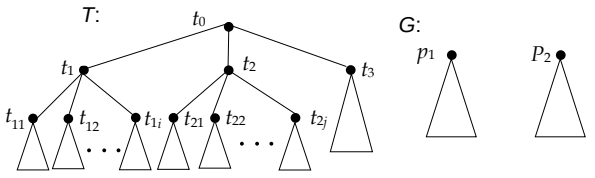


Figure 6. A tree and a forest

We handle G as a tree P = <v_G; P_1, ..., P_q>, where v_G 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_i, ..., v_j, respectively.

Definition 2 Consider an interval of the form <[1, i], v>, representing an ordered forest containing the first i subtrees of v: <$G[v_1]$, ..., $G[v_i]$>. For simplicity, it is also denoted as <i, v>. If v is v_G, or a node on the left-most path in P_1, <i, v> 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_G> is a left-corner, representing the first i subtrees in G: P_1, ..., P_i. So, <q, v_G> stands for the whole G.

In addition, we will use $\overline{<i, v>}$ to represent the forest <$G[v_{i+1}]$, ..., $G[v_k]$>, 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], v> interchangeably without causing any confusion.

Let u be a node on the left-most path in P_1. Let <i, v> be a left-corner of G. If $v = u$, we say that <i, v> and u are *level-equal*, denoted as <i, v> ≅ u. If v is an ancestor of u, we say, <i, v> is higher than u, denoted as <i, v> ⇝ u. Then, <i, v> $\underline{\rightsquigarrow}$ u represents that <i, v> is higher than or level-equal 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 $t_{i...jk}$ is the kth child of $t_{i...j}$. For example, t_{01} (simplified as t_1) is the first child of t_0, t_{12} is the second child of t_1, and so on; and each node in G is identified with $p_{l...k}$. Besides, each subtree rooted at $t_{i...j}$ ($p_{l...k}$) is represented by $T_{i...j}$ (resp. $P_{l...k}$).

In Fig. 6, in order to check whether T includes $G = <P_1, P_2>$, we can first check whether T_1 alone includes G. That is, we will perform a recursive call as follows:

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

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

Case 1: $<i, v> = <2, v_G>$. In this case, T_1 contains G.

Case 2: $<i, v> = <1, v_G>$. In this case, T_1 contains only P_1, and we will call $A(T_2, <P_2>)$ in a next step.

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. This case is complicated and needs to be handled carefully, as described below.

In Case 3, we continue to 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 (in $<i, v>$, the return value of $A(T_1, <P_1, P_2>)$) 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 $A(T_2, <P_1, P_2>)$ returns $<i', v'>$ with $v = v'$ or $v \rightsquigarrow v'$. 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 $<i'', v''>$ of $A(T, <P_1, P_2>)$. Then, $<i', v'>$ is not used, as illustared in Fig. 7(a).

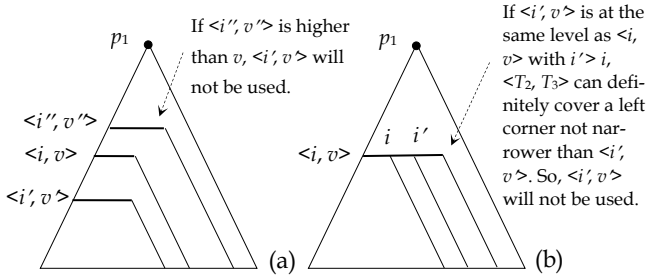


Figure 7. Illustration for cutting computation

If $<i'', v''>$ is not higher than v, we will make a *supplement checking* of $<T_2, T_3>$ against $<i, v>$ (the complement of $<i, v>$) to see whether $<T_2, T_3>$ is able to embed some subtrees in $<i, v>$. Assume that $<T_2, T_3>$ embeds the first j subtrees in $<i, v>$. Then, the return value of $A(T, <P_1, P_2>)$ should be $<i + j, v>$. It is because in this case we have T_1 cover $<i, v> = <G(v_1), \ldots, G(v_i)>$ and $<T_2, T_3>$ cover $<G(v_{i+1}), \ldots, G(v_{i+j})>$ in $<i, v>$. So, $<T_1, T_2, T_3>$ together covers $<G(v_1), \ldots, G(v_i), G(v_{i+1}), \ldots, G(v_{i+j})>$. Again, $<i', v'>$ is not used, but without impacting the correctness according to the following analysis:

> If $v \rightsquigarrow v'$, 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 $<i, v>$, and the supplement computation will find this embedding.

The reasoning here is quite simple. If T_2 alone is able to cover $<i', v'>$ with $v' = v$ and $i' > i$, $<T_2, T_3>$ can definitely cover a left corner not narrower than $<i', v'>$, and then is able to cover $<[i + 1, i'], v'>$ or possibly a wider interval starting from $i + 1$. We use a supplement checking to do this while cut off the computation for $<i', v'>$ to avoid redundancy.

See Fig. 7(b) for illustration.

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 (represented as a left corner) obtained in the previous steps.

The following example helps for illustration on how to cut off futile work.

Example 1 Consider the tree T and the forests G shown in Fig. 8.

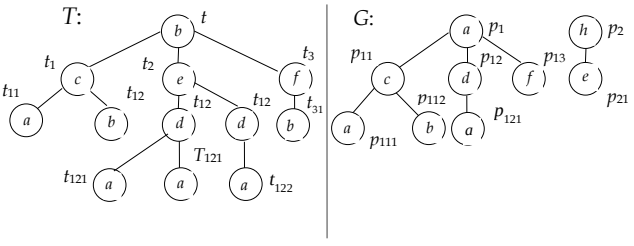


Figure 8. A target treeand a pattern forest

In order to check whether T includes G, we will first check T_1 against $G = <P_1, P_2>$. Obviously, T_1 is not able to embed G. However, it can embed P_{11} and therefore the return value of this checking should be $<1, p_1>$. In a next step, we will check T_2 against G, and try to see if T_2 alone is able to embed G. But this time, p_1 will be utilized to control the process. More specifically, it will effectively block the checking of T_2 against P_{11} since this checking can only possibly return a left-corner not higher than p_1. □

We refer to a node like p_1 in Example 1 as a *cut*.

Definition 3 A cut for a call $A(T, <P_1, \ldots, P_q>)$ $(q \geq 1)$ is a node v on the left-most in P_1, indicating that only a left corner higher than v will be returned by $A(T, G, v)$ if it is embeddable in T. Otherwise, $A(T, <P_1, \ldots, P_q>)$ 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 (= <t; T_1, …, T_k>) against a forest G (= <P_1, ..., P_q>). 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_1>; or G = <P_1, ..., P_q> with $q > 1$, but $|T| \le |P_1| + |P_2|$. 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.

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: α-function and β-function, used to handle *Case* 1 and *Case* 2, respectively:

$$\alpha(T, P_1) = \langle i, v \rangle, \quad (1)$$

where <i, v> is a highest and widest left-corner in P_1, which can be embedded in T.

$$\beta(\langle T_1, ..., T_k \rangle, G) = \langle j, u \rangle, \quad (2)$$

where <j, u> is a highest and widest left-corner in G, which can be embedded in <T_1, ..., T_k>.

Here, our intention is quite straightforward:

In Case 1 we will call $\alpha(T, P_1)$ and in Case 2 we will call $\beta(\langle T_1, ..., T_k \rangle, G)$. However, during the working process, they may call each other recursively.

Additionally, in Case 2, the return value <j, u> of $\beta(\langle T_1, ..., T_k \rangle, G)$ needs to be further checked as follows:

- If label(t) = label(u) and $j = d(u)$, the return value of $A(T, G)$ should be set to <1, u's parent>, showing that T includes $G[u]$. Otherwise, the return value of $A(T, G)$ is the same as <j, u>. (For this reason, $d(v_G)$ is set to be ∞, larger than the outdegree of any node in both T and G. Thus, in the case that T contains <P_1, …, P_q>, the return value must be <q, v_G>, not <1, v_G's parent>.)
- If label(t) ≠ label(u) or $j \ne d(u)$, the return value of $A(T, G)$ 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 $A(T, G)$ can be described as below.

FUNCTION 1. $A(T, G)$

input: T = <t; T_1, ..., T_k >, G = <P_1, ..., P_q>.
output: a left corner.
begin
1. **if** (q = 1 or $|T[t]| \le |G[p_1]| + |G[p_2]|$) (*Case 1*)
2. **then** return $\alpha(T, P_1)$
3. **else** <j, u> := $\beta(\langle T_1, ..., T_k \rangle, G)$; (*Case 2*)
4. **if** (label(t) = label(u) and $j = d(u)$
5. **then** return <1, u's parent>;
6. return <j, u>;

end

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. 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 $\delta(v_1) = \delta(v_2) = \delta(v_3) = \delta(v_4) = v_4$, $\delta(v_5) = \delta(v_6) = v_6$, $\delta(v_7) = v_7$, and $\delta(v_8) = v_8$. Denote by $\Delta(v')$ a set of nodes $\boldsymbol{x}$ such that for each $v \in \boldsymbol{x}$ $\delta(v) = v'$. Thus, in Fig. 9, we have $\Delta(v_4) = \{v_1, v_2, v_3, v_4\}$, $\Delta(v_6) = \{v_5, v_6\}$, $\Delta(v_7) = \{v_7\}$, and $\Delta(v_8) = \{v_8\}$. Let p_1 be the root of P_1. We also have $\rho(G) = \delta(p_1)$.

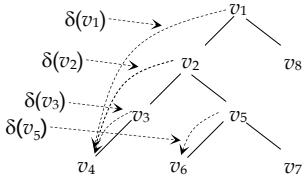


Figure 9. A pattern tree

Let T = <t; T_1, …, T_k>, G = <P_1, ..., P_q>. In $\alpha(T, P_1)$, altogether five different cases as listed in Fig. 10 should be checked.

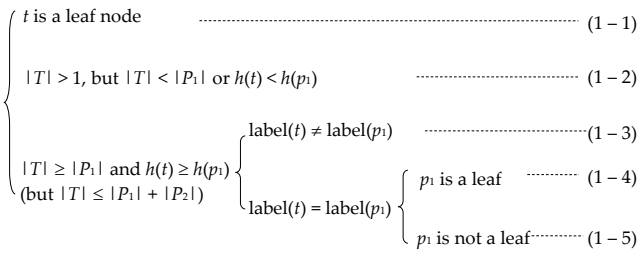


Figure 10. Different cases to be checked in α-function

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 $A(T, \langle P_{11}, ..., P_{1j} \rangle)$, where <$P_{11}$, ..., P_{1j}> is a forest containing all the direct subtrees of p_1. The return value of $A(T, \langle P_{11}, ..., P_{1j} \rangle)$ is used as the return value of $\alpha(T, P_1)$. 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}>.

If $|T| \ge |P_1|$ and $h(t) \ge h(p_1)$ (but $|T| \le |P_1| + |P_2|$), we further distinguish among three cases: Case (1-3), (1-4) and (1-5).

In Case (1-3), we have label(t) ≠ label(p_1), and we will call $\beta(\langle T_1, ..., T_k \rangle, \langle P_1 \rangle)$ to see whether $\beta(\langle T_1, ..., T_k \rangle$ is able to embed P_1.

In Case (1-4), we have label(t) = label(p_1) and p_1 is a leaf node. In this case, we return <1, p_1's parent>.

In Case (1-5), we have label(t) = label(p_1), but p_1 is not a leaf node. In this case, we need to call β(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>). Assume that the return value of β( ) is <i, v>. We need to do an extra checking:

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

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

FUNCTION 2. $\alpha(T, P_1)$

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(δ(p_1))
2. **then** return <1, δ(p_1)'s parent>
3. **else** return <0, δ(p_1)>;
4. **if** (1-2) **then** return $A(T$, <P_{11}, ..., P_{1j}>);
5. **if** (1-3) **then** return β(<T_1, ..., T_k>, <P_1>);
6. **if** (1-4) **then** return <1, p_1's parent>;
7. **if** (1-5) **then** <j, u> := β(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>);
8. **if** $j = d(u) \wedge$ label(t) = label(u) **then** 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 = <T_1, ..., T_k> and G = <P_1, ..., P_q>. 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 β(F, G), we will make a series of calls $A(T_l$, <P_{j_l}, ..., P_q>), where l = 1, ..., $x \le k$, $j_1 = 1$, $j_1 \le j_2 \le ... \le j_x \le q$, controlled as follows.

1. Two index variables l, j are used to scan T_1, ..., T_k and P_1, ..., P_q, respectively. (Initially, l is set to 1, and j is set to 0.) They also indicate that <P_1, ..., P_j> has been successfully embedded in <T_1, ..., T_l>.
2. Let <i_l, u_l> be the return value of $A(T_l$, <P_{j+1}, ..., P_q>). If u_l = p_{j+1}'s parent, set j to $j + i_l$. Otherwise, j is not changed. Set l to l + 1. Go to (2) (i.e., repeat this step.)
3. The loop terminates when all T_l's or all P_j'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, 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_λ ($\lambda \in \{1, ..., k\}$). 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, v_1>, ..., <i_k, v_k> be the return values of $A(T_1$, <P_1, ..., P_q>), ..., $A(T_k$, <P_1, ..., P_q>), respectively. Since j = 0, each $v_l \in \Delta(\rho(G))$ (l = 1, ..., k).
   ii) If each i_l = 0, the return value of β(F, G) should be <0, ρ(G)>. Otherwise, there must be some v_l's with $i_l > 0$. We call such a node a *non-zero point*. Find the first non-zero point v_f with children w_1, ..., w_s such that v_f is not a descendant of any other non-zero point. Then, we will check <T_{f+1}, ..., T_k> against <$P[w_{i_f+1}]$, ..., $P[w_s]$>.

   This can be done by a recursive call β(<T_{f+1}, ..., T_k>, <$P[w_{i_f+1}]$, ..., $P[w_s]$>). Let y be a number such that <$P[w_{i_f+1}]$, ..., $P[w_{i_f+y}]$> can be embedded in <T_{f+1}, ..., T_k>. The return value of β(F, G) should be set to <i_f + y, v_f>. □

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

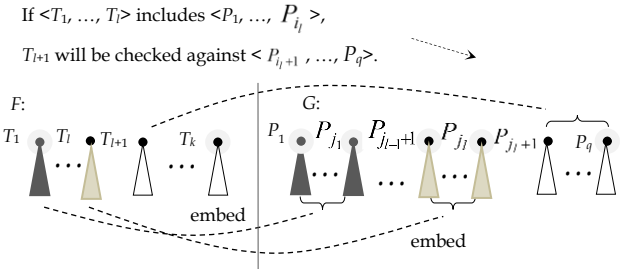


Figure 11. Illustration for an execution of β-function

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 = <T_1, ..., T_k>, G = < P_1, ..., P_q>.
output: a left corner.
begin
1. l := 1; j := 0; v := ρ(G); f := 0;
2. **while** ($j < q$ and $l \le k$) **do** (*main checking*)
3. <i_l, v_l> := $A(T_l$, <P_{j+1}, ..., P_q>)
4. **if** ($v_l = p_1$'s parent and $i_l > 0$) **then** {$j := j + i_l$;}
5. **else if** (v_l is an ancestor of v and $i_l > 0$)
6. **then** {$v := v_l$; $f := l$;}
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 v_f;
    (*supplement checking*)
11. $l := f + 1$; $j := i_f$;
12. **while** ($j < s$ and $l \le k$) **do**
13. <i_l, v_l> := $A(T_l$, <$G[w_{j+1}]$, ..., $G[w_s]$>);
14. **if** ($v_l = v_f$ and $i_l > 0$) **then** $j := j + i_l$;
15. $l := l + 1$;
16. return <j, v_f>;

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 <T_1, ..., T_k> embeds <P_1, ..., P_j>. 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 0. Therefore, 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 <T_{f+1}, ..., T_k> against <$P[w_{i_f+1}]$, ..., $P[w_s]$>, where w_{i_f+1}, …, w_s are all those children of the first non-zero point v_f such that the subtrees rooted at them are not covered by <T_1, ..., T_f>. (Notice that <i_f, v_f> is the return value of $A(T_f$, <P_1, ..., P_q> with $i_f > 0$.) 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 α-function and β-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 = <t; T_1, ..., T_k >, G = <P_1, ..., P_q>. In $A(T, G)$, we distinguish between two cases: (i) $q = 1$ or $|T| \le |P_1| + |P_2|$, 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 $\alpha(T, P_1)$. If α-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_1, ..., P_l> with $1 < l \le q$, then <T_1, ..., T_k> must be able to cover <P_1, ..., P_l> 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 <T_1, ..., T_k> have to be mapped to the nodes in $G[p]$, excluding p, to satisfy the ancestor condition in the definition. So we call β(<T_1, ..., T_k>, G) to do this task. Let <j, u> be the return value of β(<T_1, ..., T_k>, G). We further need to check whether label(u) = label(u). If it is the case, T covers <1, u's parent>. Otherwise, T only covers <j, u>. Thus, in case (ii), if the β-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 α-function and β-function must be correct.

Proof. The lemma can be proven by analyzing the five cases in the α-function, as well as the main checking and the supplement checking in the β-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 α-function and β-function for the basic case can be established.

Proposition 1 Let T = <t; T_1, ..., T_k > and G = <P_1, ..., P_q>. 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_q. In this case, the algorithm returns <0, p_1> or <1, v_G>, depending on whether label(t) = label(p_1). 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: <t; t_1, ...,t_k>, and G is a set of nodes: <p_1, ..., p_q>.

(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 $|T| \le |P_1| + |P_2| = 2$ (i.e., if t has at most only one child t_1), $\alpha(T, P_1)$ will be called (see lines 1 – 2 in A( )). If label(t) ≠ label(p_1), we have Case (1-3) and will call β(<T_1>, <P_1>), which leads to the call $A(T_1$, <P_1>) (see line 3 in β( )) and then to the call $\alpha(T_1, P_1)$. Since T_1 contains only a single node t_1, we have Case (1-1) and returns <1, v_G> or <0, p_1> depending on whether label(t_1) = label(p_1) (see lines 1 – 3 in α( )). If label(t) = label(p_1), we have Case (1-4) since p_1 is a leaf, and return <1, v_G>.
- If $|T| > |P_1| + |P_2| = 2$, β(<t_1, ...,t_k>, <p_1, ..., p_q>) will be invoked (see line 4 in A( )), which will find a sequence of integers: k_1, ..., k_x such that label(t_{k_i}) = label(p_i) (i = 1, ..., x) (see line 3 in β( )). The return value is <x, v_G> ($0 \le x \le q$).

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 α( ).

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

Consider T = <t; T_1, ..., T_k> and G = <P_1, ..., P_q> with $H = h_T + h_G = h + 1$.

If $q = 1$, or $q > 1$ but $|T| \le |P_1| + |P_2|$, $\alpha(T, P_1)$ will be invoked. If it is case (1-1), or (1-4), the proposition obviously holds.

If it is case (1-2), $A(T$, <P_{11}, ..., P_{1j}>) will be invoked. Since the sum of the height of T and the height of <P_{11}, ..., P_{1j}> is equal to h, according to the induction hypothesis, the proposition holds.

If it is case (1-3), β(<T_1, ..., T_k>, <P_1>) will be called, by which a series of calls $A(T_l$, <P_1>) will be conducted, where $l = 1, ..., x \le k$, $j_1 = 1$, $j_1 \le j_2 \le ... \le j_x \le q$. According to the induction hypothesis, each $A(T_l$, <P_1>) returns a correct value. Thus, in terms of Lemma 2, the return value of β(<T_1, ..., T_k>, <P_1>) must be correct.

If it is case (1-5), β(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>) will be invoked, by which a series of calls $A(T_l$, <P_{11}, ..., P_{1j}>) will be carried out. Again, the sum of the height of T_l and the height of <P_{11}, ..., P_{1j}>) equals h - 1. So, according to the induction hypothesis and Lemma 2, the proposition also holds.

If $q > 1$ and $|T| > |P_1| + |P_2|$, β(<T_1, ..., T_k>, G) will be called, by which a series of calls $A(T_l$, < P_{j_1}, ..., P_q>) will be conducted, where $l = 1, ..., x \le k$, $j_1 = 1$, $j_1 \le j_2 \le ... \le j_x \le q$. 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 illustarted 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, α-function, as well as β-function.

To this end, we change $A(T, G)$ to take an extra parameter (i.e., a cut) $v \in \Delta(\rho(G))$, 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 $<0, \rho(G)>$. α-function and β-function will also be accordingly changed such that within $A(T, G, v)$, the cut v can be transferred to both $\alpha(T, P_1, v)$ and $\beta(<T_1, \ldots, T_k>, G, v)$.

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: $T = <t; T_1, ..., T_k>$, $G = <P_1, ..., P_q>$.
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 $|T[t]| \le |G[p_1]| + |G[p_2]|$) (*Case 1*)
3. **then** return $\alpha(T, P_1, v)$
4. **else if** label(t) = label(v) (*Case 2*)
5. **then** $<j, u> := \beta(<T_1, ..., T_k>, G,$ v's first child);
6. **else** $<j, u> := \beta(<T_1, ..., T_k>, G, v)$;
7. **if** $v \neq p_1$'s parent
8. **then if** (label(t) = label(u) $\wedge$ $j = d(u)$
9. **then** return $<1, u$'s parent$>$;
10. return $<j, u>$;
end

In the algorithm, $\alpha(T, P_1, v)$ and $\beta(<T_1, ..., T_k>, G, v)$ are defined as follows.

$$\alpha(T, P, v) = \begin{cases} <j, u>, & \text{if } <j, u> \text{ is a highest and widest left-corner in } P, \text{ which can be embedded in } T, \text{ higher than } v; \\ <0, \rho(G)>, & \text{otherwise.} \end{cases} \quad (3)$$

$$\beta(<T_1, ..., T_k>, G, v) = \begin{cases} <j, u>, & \text{if } <j, u> \text{ is a highest and widest left-corner in } G, \text{ which can be embedded in } \alpha(<T_1, ..., T_k>, \text{higher than } v; \\ <0, \rho(G)>, & \text{otherwise.} \end{cases} \quad (4)$$

First, we note that in line 1 we check whether p_1'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 α-function for Case 1, and β-function for Case 2 as in the basic algorithm, but with a cut transferred.

In addition, in Case 2, depending on whether label(t) = label(v), we will call $\beta(<T_1, ..., T_k>, G, v$'s first child) or $\beta(<T_1, ..., T_k>, G, v)$ (see lines 4 – 6). It is because if label(t) = label(v), we may have T covering $G[v]$ if $<T_1, ..., T_k>$ 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 $<1, v$'s parent$>$, higher than v. So, the cut needs to be downgraded to v's first child so that this part of computation will not be blocked.

- *Cut Propagation in α-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 α-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 $A(T, <P_{11}, ..., P_{1j}>, v)$, 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 label(t) ≠ label(p_1). In this case, we will simply call $\beta(<T_1, ..., T_k>, <P_1>, v)$, 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 β-function to check $<T_1, ..., T_k>$ against $<P_{11}, ..., P_{1j}>$). In this case, we have label(t) = label(p_1). Concerning the cut transfer, we need to consider two subcases:

i) $p_1 = v$. In this case, we will call $\beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, p_{11})$ with the cut being set to be p_{11}. It is because in this case the main checking of the β-function execution may reveal that $<T_1, ..., T_k>$ is able to embed the whole $<P_{11}, ..., P_{1j}>$. In this case, the return value of $\alpha(T, G, v)$ will be set to $<1, p_1$'s parent$>$, higher than v. So it is a useful computation; and downgrading the cut from $v = p_1$ to p_{11} will let it go through. On the other hand, p_{11} will effectively prohibit any possible further supplement checking in this β-function execution since such a checking can only bring out a left corner lower than p_{11} and will not be used.

ii) $p_1 \rightsquigarrow v$. In this case, we will call $\beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, v)$, by which v is directly transferred since we must have $p_{11} \underline{\rightsquigarrow} v$ and no useful computation can be eliminated by cut v.

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

FUNCTION 5. $\alpha(T, P_1, v)$

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 $\beta(<T_1, ..., T_k>, <P_1>, v)$;
6. **if** (1-4) **then** return $<1, p_1$'s parent$>$;
7. **if** (1-5-i) **then** $<j, u> := \beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, p_{11})$;
8. **if** (1-5-ii) **then** $<j, u> := \beta(<T_1, ..., T_k>, <P_{11}, ..., P_{1j}>, v)$;
9. **if** $j = d(u)$ and label(t) = label(u) **then** return $<1, u$'s parent$>$
10. **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* transfer 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 T_l against a forest of the form $< P_{j_l}, ..., P_q >$ with $j_l \geq 1$ in the main checking of $\beta(<T_1, ..., T_k>, G, v)$. Without loss of generality, assume that $<i_l, v_l>$ is the return value of $A(T_l, < P_{j_l}, ..., P_q >, u_l)$ for $l = 1, ..., x \leq k$ with $j_1 = 1$, $j_1 \leq j_2 \leq ... \leq j_x \leq q$. Then, we have

- $u_1 = v$, a value transferred to $\beta(<T_1, ..., T_k>, G, v)$.
- For $2 \leq l \leq x$, u_l is determined as follows:
  Let s be an integer such that any of $T_1, ..., T_s$ is not able to embed P_1, but T_{s+1} embeds $<P_1, ..., P_j>$ for some $j > 0$. Then, for $2 \leq l \leq s$, we have

$$u_l = \begin{cases} v_{l-1}, & \text{if } v_{l-1} \text{ is an ancestor of } u_{l-1} \text{ and } i_{l-1} > 0; \\ u_{l-1}, & \text{if } v_{l-1} \text{ is not an ancestor of } u_{l-1} \text{ or } i_{l-1} = 0; \end{cases} \quad (5)$$

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

$$u_l = p_{j_l}. \quad (6)$$

The formula (5) shows how the cuts are changed before we meet the first subtree in $<T_1, ..., T_k>$ which is able to embed some subtrees $P_1, ..., P_j$ $(j > 0)$. 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 T_l against $< P_{j_l}, ..., P_q >$, a returned left corner lower than p_{j_l} will not be used in the subsequent computation.

If $s < k$, it shows that $<T_1, ..., T_k>$ includes $<P_1, ..., P_m>$ for some m $(1 \leq m \leq q)$, and the supplement checking will not be conducted. If $s = k$, $<T_1, ..., T_k>$ does not include any subtree in G, but some T_l's each may include a non-empty left corner in P_1. 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 supplelement 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 processs to control the supplement computation.

FUNCTION 6. $\beta(F, G, v)$

```
input: F = <T1, ..., Tk>, G = < P1, ..., Pq>.
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.      <il, ul> := A(Tl, <Pj+1, ..., Pq>, u)
4.      if (ul = p1's parent and il > 0) then {j := j + il; u := pj;}
5.      else if (ul is an ancestor of u and il > 0)
6.           then {u := ul; f := l;}
7.      l:= l + 1;
8.  if j > 0 then return <j, p1's parent>;
9.  if f = 0 then return <0, δ(p1)>;
10. let w1, ..., ws be the children of uf;
    (*supplement checking*)
11. l := f + 1; j := if;
12. while (j < s and l ≤ k) do
13.     <il, vl> := A(Tl, <G[wj+1], ..., G[ws]>, wj+1);
14.     if (vl = vf and il > 0) then j := j + il;
15.     l:= l + 1;
16. return <j, uj>;
end
```

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 $<T_1, ..., T_k>$ embeds $<P_1, ..., P_j>$. In this process, by the first *A*-function call we have cut v, which is the same as the cut propagated to $\beta(<T_1, ..., T_k>, G, v)$ while by 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 $<t_1, k>$ (resp. $<p_1, q>$). 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, $\alpha(T, P_1, v)$ can be simply represented by $\alpha(t, p_1, v)$ while $\beta(<T_1, \ldots, T_k>, <P_1, \ldots, P_q>, v)$ by $\beta(t_1, k, p_1, q, v)$, which indicates that only a constant space is needed to record a recursive call.

Furthermore, each A-function call is always followed by an α-function call or a β-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 $\alpha(t, p_1, v)$, and $\beta(t_1, k, p_1, q, v)$. Thus, we can simply divide L into two sequences: L_α and L_β such that L_α contains only the α-function calls while L_β only the β-function calls. L_α and L_β are called the α-subchain and β-subchain of L, respectively. For example, the chain shown in Fig. 12(a) can be divided into an α-function call chain and a β-function call chain, as shown in Fig. 12(b) and (c), respectively.

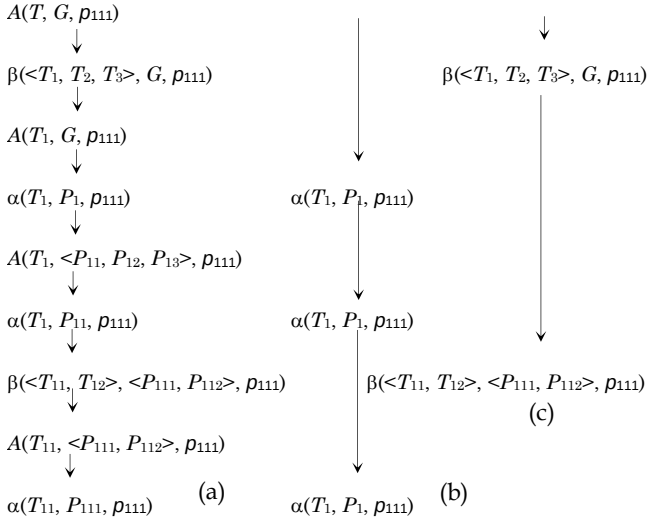


Figure 12. A recursive call chain

For L_α, we have the following lemma.

Lemma 3 Let $x = \alpha<t, p, v>$ and $x' = \alpha<t', p', v'>$ be two consecutive function calls on an L_α. Then, (i) if $t = t'$, p' is a child of p; and (ii) if $p = p'$, t' is a child of t.

Proof. We first prove (i). If $t = t'$, it shows that $T[t]$ is involved in a call of the A-function during the execution of $\alpha<t, p, v>$, but checked against a forest containing the subtrees respectively rooted at the children of p. This occurs when Case (1-2) is satisfied (see line 4 in Function 5). Through the A-function call, $\alpha<t', p', v'>$ 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 α-function, which happens when Case (1-3) is satisfied, i.e., when $|T| \le |P| + z$, where z is the size of the subtree rooted at p's right sibling, but $|T| \ge |P|$, $h(t) \ge h(p)$, and label(t) ≠ 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 β-function (see line 5 in Function 5), through which $\alpha<t', p', v'>$ is invoked. See Fig. 13(b) for illustration, which shows that if a node p in G is checked against two consective nodes t and t' in T along a recursive call, t must be the parent of t'. This completes the proof. □

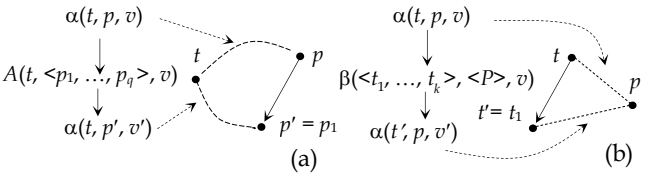


Figure 13. Illustration for Lemma 3

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

In a similar way, we can prove that that $|L_\beta|$ is also bounded by $O(h_T + h_G)$.

Therefore, $|L| = |L_\alpha| + |L_\beta|$ is in the order of $O(h_T + h_G)$.

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

Proof. See the above analysis. □

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 $O(|T| \cdot h_G)$. Then, we further demonstrate that the time requirement is also bounded by $O(|T| \cdot |leaves(G)|)$. This indicates that the time complexity of our algorithm is $O(|T| \cdot min\{h_G, |leaves(G)|\})$.

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 $\beta(<T_{f+1}, \ldots, T_s>, <G[w_{j+1}], \ldots, G[w_s]>, u)$ we always have $u = w_{j+1}$, by which any further supplement checking is effectively blocked.

In order to see that the time complexity is bounded by $O(|T| \cdot h_G)$, we analyze, in the worst case, how many β-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 β-function call of the form $\beta(<T[t'], \ldots>, <P_1, \ldots, P_q>, u')$, in whose execution $\beta(<T[t''], \ldots>, \ldots, u'')$ is invoked (possibly through an A-function call invoked during the execution of $\beta(T[t'], <P_1, \ldots, P_q>, u')$; see line 3 in FUNCTION 6.) Then, u' and u'' can be in one of three relationships:

1. $u'' \rightsquigarrow u'$. In this case, t'' can possibly be involved in a supplement checking, but t' definitely not since the left corner of $\beta(T[t'], <P_1, \ldots, P_q>, u')$ 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 $\beta(<T[t'], \ldots>, <P_1, \ldots, P_q>, u')$, 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 $<P_1, \ldots, P_q>$, as well as the return value of $\beta(<T[t'], \ldots>, <P_1, \ldots, P_q>, u')$ itself.
3. $u' \rightsquigarrow u''$ (more exactly, u'' is the first child of u'.) This happens when through an A-function call, a β-function is invoked, by which the cut is downgraded (see line 5 in FUNCTION 4); or an α-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 β-function call once again.

Obviously, (3) is the worst case, by which the number of β-function calls t'' is involved in is maximized. Now, we observe the parent t of t' and assume that in the execution of $\beta(<T[t], \ldots>, \ldots, u)$, $\beta(T[t'], <P_1, \ldots, P_q>, u')$ is invoked. Repeating the above analysis, we can see that if $u \rightsquigarrow u'$ (case 3), both t and t' can also be involved in a supplement checking. This shows that if we have $u \rightsquigarrow u' \rightsquigarrow u''$ t can be involved in two β-function calls, t' in three β-function calls, and t'' in four β-function calls. In general, the number of β-function calls, in which a certain node t in T is involved, must be bounded by $h_G + 1$ since any sequence of cuts: $u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_z$ in G cannot contain more than h_G nodes, and any recursive call with t involved corresponds to a cut at a different level.

In 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 $O(|T| \cdot h_G)$.
Proof. We need to show that any node t in T can also be involved in at most $O(h_G)$ 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 α-function and a β-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 β-function calls. For the first kind of A-to-A chains, the number of A-function calls for each t is also bounded by $(h_G + 1)$ 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 $A(T, <P_1, \ldots, P_q>, u)$ 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 $O(|T| \cdot |leaves(G)|)$.
Proof. To show that the time complexity of the algorithm is also bounded by $O(|T| \cdot |leaves(G)|)$, we observe the worst case, i.e., case 3 (the case $u' \rightsquigarrow u''$ 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 aginst 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 $O(|leaves(G)|)$. Therefore, for whole T, the number of checkings is bounded by $O(|T| \cdot |leaves(G)|)$. □

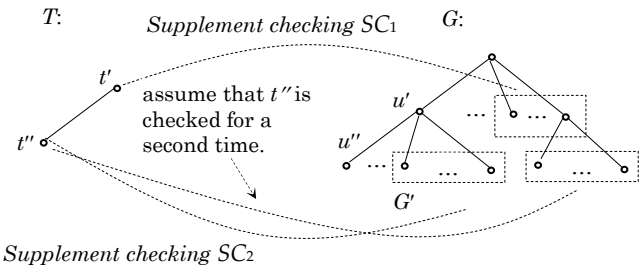


Figure 14. Illustration for supplement checking

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 Lemm 4, we use cuts to explain that any node in T can be involved in at most $O(h_G)$ function calls while in the proof of Lemma 5, we show that any node in T can be checked against at most $O(|leaves(G)|)$ 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 $O(|T| \cdot min\{h_P, |\text{leaves}(G)|\})$.
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

	TreeBank	DBLP	XMARK		
			1	3	5
Data size (MB)	82	127	113	340	568
No. of nodes (million)	2.43	3.33	1.72	5.1	8.33
Max/average height	36/7.9	6/2.9	12/6.2		

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.

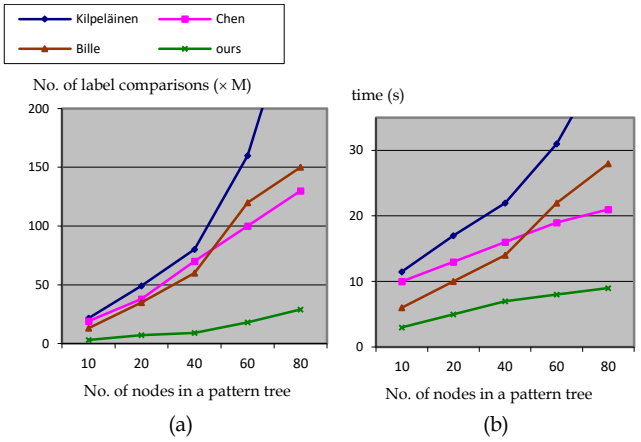


Figure 15. Results on varying sizes of patterns - treeBank

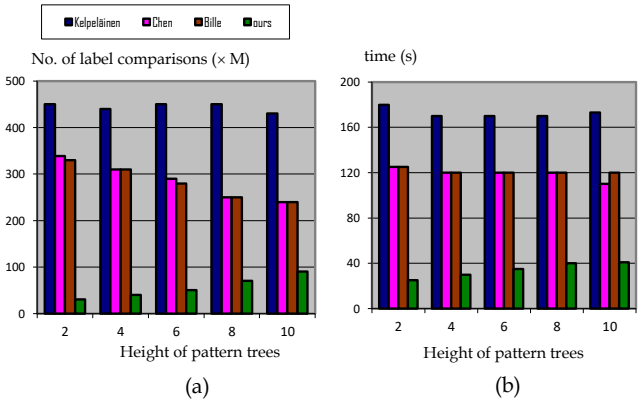


Figure 16. Results on varying heights of patterns - treeBank

- *Tests on DBLP*

In Fig. 18, we show the test results on the DBLP data set, by which only the numbers of label comparisons are demonstrated. Since the elapsed time is always propotional to the number of label comparisons as we can see from Fig. 15 and 16, we show here only the number of label comparisons. Again, our method has the best performance for this test. Especially, for the patterns of different heights, the numbers of label comparisons are not much changed. It is because in all document trees most of the paths are

quite short (on average their lenghts are bounded by 3) and the number of leaf nodes is large and comparable to the whole size of the tree itself.

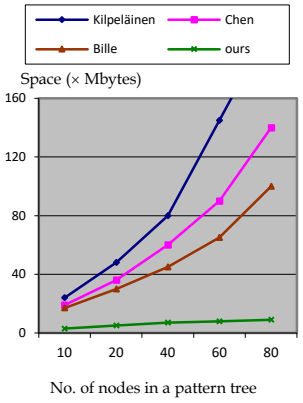


Figure 17. Space overhead on varying patterns - treeBank

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 uniformally 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 comparions 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.

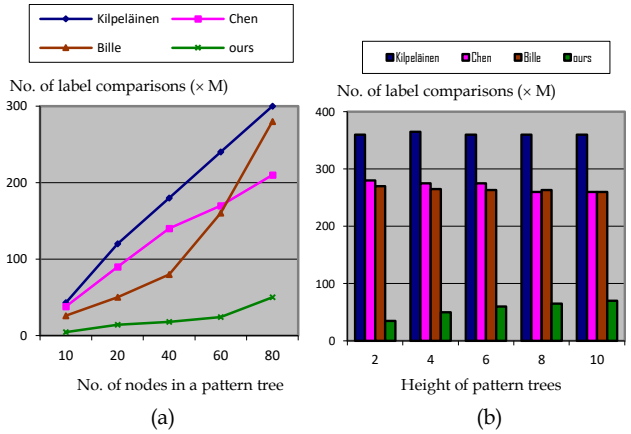


Figure 18. Results on varying sizes and heights of patterns – DBLP

- *Tests on XMARK*

In Fig. 20(a) and (b), we show the number of label comparisons for matching patterns against the XMark data set with the scaling factor = 1. As by the treebank and the DBLP, the Kilpeläinen's has the worst performance, and

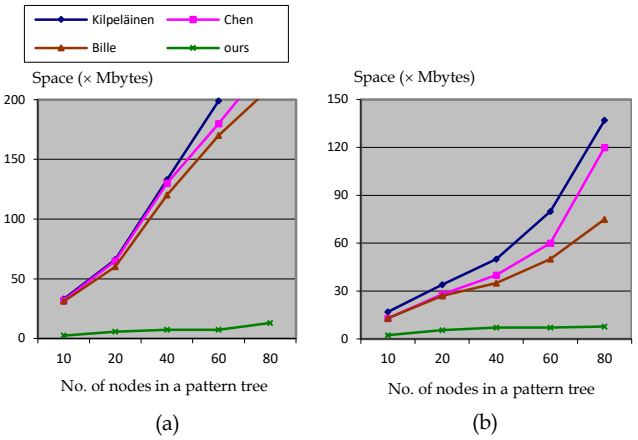


Figure 19. Space overhead on varying patterns - XMark

In Fig. 19(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.

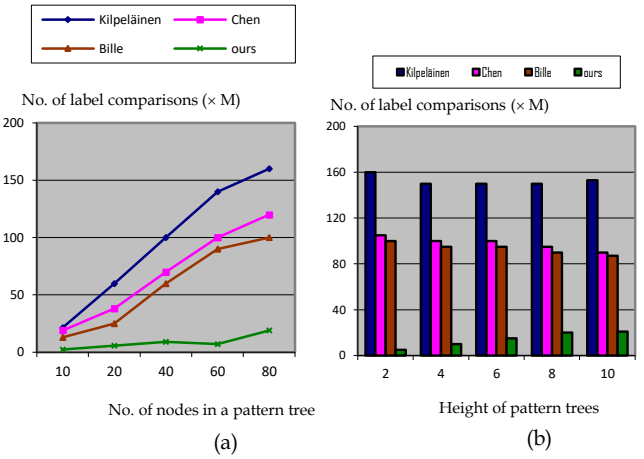


Figure 20. Results on varying sizes and heights of patterns - XMark

- *On Scalability*

Now we test the scalability of our method by varying the sizes of taget tress. For this purpose, we change the scaling factors when generating XMark data. In Fig. 21, we report the test results on the XMark data with the scaling factor = 3 while in Fig. 22, we report the results for the scaling factor = 5. For the scale factor equal to 3, the number of generated nodes is about 5.1 millions. For the scale factor equal to 5, the number of generated nodes is about 8.33 millions. In both of these two tes(a), our method exhibits the best performance.

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 $O(|T| \cdot min\{h_P, |leaves(P)|\})$ time and $O(h_P + h_T)$ space (besides the space for storing T and P themselves), where T and P are a target and a pattern tree (forest), respectively; h_P (h_T) is the hight 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.

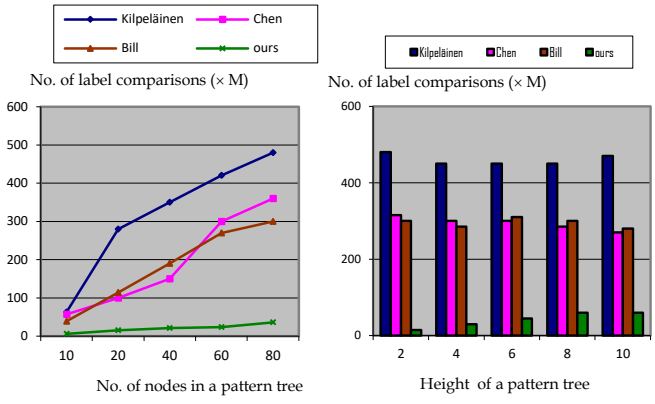


Figure 21. Results on varying sizes and heights of patterns - XMark

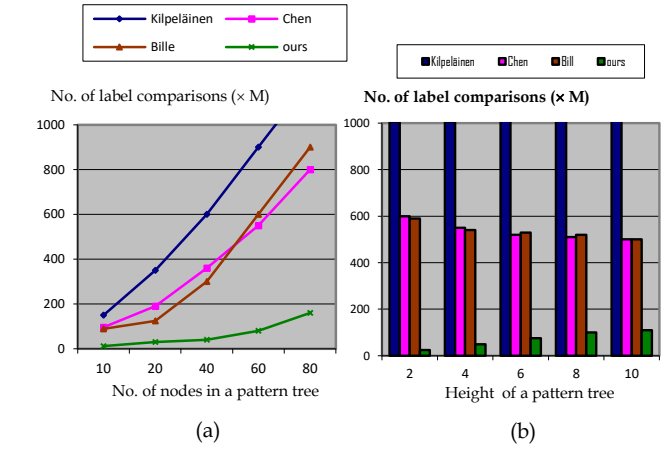


Figure 22. Results on varying sizes and heights of patterns - XMark

10 REFERENCES

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

[2] 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.

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

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

[5] Y. Chen and Y.B. Chen: Decomposing DAGs into spanning trees: A new way to compress transitive closures, in *Proc. 27th Int. Conf. on Data Engineering (ICDE 2011)*, IEEE, April 2011, pp. 1007-1018.

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

[7] Y. Chen and Y.B. Chen, On the Tree Inclusion Problem, in Proc. *ICCNCE2013*, pp. 131 – 135.

[8] 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.

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

[10] P. F. Dietz. Maintaining order in a linked list. In *Proc. STOC*, 1982.

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

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

[13] R.B. Lyngs, M. Zuker& C.N.S. Pedersen, Internal loops in RNA secondary structure prediction, in *Proceedings of the 3rd annual international conference on computational molecular biology (RECOMB)*, 260-267 (1999).

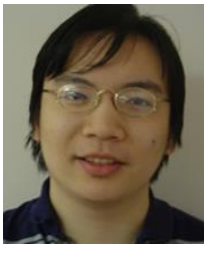
[14] H. Mannila and K.-J.Räiha, On Query Languages for the p-string data model, in "Information Modelling and Knowledge Bases" (H. Kangassalo, S. Ohsuga, and H. Jaakola, Eds.), pp. 469-482, IOS Press, Amsterdam, 1990.

[15] Thorsten Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings* of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in *LectureNotes of Computer Science (LNCS), volume 1264*, pages 150-166. Springer, 1997.

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

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

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 2: Symbols and notations

$T = <t; T_1, ..., T_k>$	target tree with root t and its direct subtrees $T_1, ..., T_k$
$G = <P_1, ..., P_q>$ $(q \geq 0)$	pattern, which is a forest containing subtrees $P_1, ..., P_q$
$V(T)$	set of nodes in T
$E(T)$	set of edges in T
h_T	height of T
leaves(P)	all leaf nodes of P
$d(v)$	outdegree of node v
$G[v]$	subtree rooted at v
$v \prec v'$	v is to the left of v'
$v \preceq v'$	$v \prec v'$ or $v = v'$
$v \rightsquigarrow u$	v is a proper ancestor of u
φ: $V(G) \rightarrow V(F)$	an injective function mapping nodes in $V(G)$ to nodes in $V(F)$
$\rho(G)$	left-most leaf node in G
ν_G	virtual root of G
$<[i, j], v>$	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 the children of v: $v_i, ..., v_j$, respectively.
$<i, v>$	abbreviation of $<[1, i], v>$. If v is ν_G or a node on the left-most path of P_1, it is called a *left corner* of G.
$\overline{<i, v>}$	represent a forest $<G[v_{i+1}], ..., G[v_k]>$, referred to as the *complement* of $<i, v>$
$<i, v> \cong u$	level-equal, i.e., $v = u$
$<i, v> \rightsquigarrow u$	$<i, v>$ is higher than u, i.e., $v \rightsquigarrow u$
$<i, v> \underline{\rightsquigarrow} u$	$<i, v>$ is higher than or level-equal to u
$A(T, G)$	a checking of G against T, returning a *highest* and *widest* left-corner $<i, v>$ in G
$\alpha(T, P)$	A function returning a highest and widest left-corner in P, which can be embedded in T, where P is a subtree in G.
$\beta(<T_1, ..., T_k>, G)$	A function returning a highest and widest left-corner in P, which can be embedded in $<T_1, ..., T_k>$.
$\delta(v)$	a link to the left-most leaf node in $G[v]$
$\Delta(v')$	a set of nodes $\boldsymbol{x}$ such that for each $v \in \boldsymbol{x}$ $\delta(v) = v'$
$h(t)$	height of $T[t]$
main computation	part of β-function execution process
supplement computation	part of β-function execution process
SCC-condition	*supplement checking 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 $<0, \rho(G)>$.
$\alpha(T, P, v)$	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 $<0, \rho(G)>$.
$\beta(<T_1, ..., T_k>, G, v)$	variant of $\beta(<T_1, ..., T_k>, G)$, 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 $<T_1, ..., T_k>$ and higher than v. Otherwise, it returns $<0, \rho(G)>$.
strict SCC-condition	*strict supplement checking condition*

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 $A(F, G)$ $\beta(F, G)$ will be invoked, in which we will call $A(T_1, <P_1, P_2>)$, $A(T_2, <P_1, P_2>)$, and $A(T_3, <P_1, P_2>)$ in turn (see lines 3 – 20, lines 21 – 70, and lines 71 – 88 in the sample trace). We call $A(T_2, <P_1, P_2>)$ after $A(T_1, <P_1, P_2>)$ because the return value of $A(T_1, <P_1, P_2>)$ is $<1, p_1>$ (see line 20), showing that T_1 is not able to embed P_1. Since T_2 is not able to include P_1 (T_2 contains only $<1, p_{11}>$, see line 70), either, $A(T_3, <P_1, P_2>)$ will be invoked (see lines 71 – 88), whose return value is $<0, p_{111}>$. So, a sumplement checking will be carried to check $<T_2, T_3>$ against $<P_{12}, P_{13}>$ (see lines 89 – 116). It is because $A(T_1, <P_1, P_2>)$ returns $<1, p_1>$, higher than both $<1, p_{11}>$ (return value of $A(T_2, <P_1, P_2>)$ and $<0, p_{111}>$ (return value of $A(T_3, <P_1, P_2>)$) and we need to know whether the left corner $<1, p_1>$ can be expanded by the supplement checking, in which we will first check T_2 against $<P_{12}, P_{13}>$ (see lines 89 – 112). Since it returns $<1, p_1>$ (showing that T_2 covers P_{12}), we will call $A(T_3, <P_{13}>)$ in a next step (see lines 113 – 116), which also returns $<1, p_1>$ (showing that T_3 covers P_{13}.) Therefore, the whole process of $\beta(F, G)$ returns $<3, p_1>$, showing that T includes $< P_{11}, P_{12}, P_{13}>$. □

In the above sample trace, if t_{121} were something other than an 'a', then $A(T_{12}, <P_1, P_2>)$ would return $<0, p_{111}>$ and the supplement checking (after $A(T_{13}, <P_1, P_2>)$ returns $<1, p_{11}>$) would check $<T_{12}, T_{13}>$ against $<P_{112}>$ since T_{11} includes P_{111}.

step-by-step trace:	explanation:
1. $A(T, G)$	$A(T, G)$ begins.
2. $\beta(<T_1, T_2, T_3>, G)$	since G is a forest and $\|T\| > \|P_1\| + \|P_2\|$, call $\beta(\ )$ (see line 3 in $A(\ )$.)
3. $A(T_1, G)$	in the 1st **while**-loop (lines 2 – 7) of $\beta(\ )$, call A-functions in turn (line 3 in $\beta(\ )$.)
4 $\alpha(T_1, P_1)$	since $\|T_1\| \leq \|P_1\| + \|P_2\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
5 $A(T_1, <P_{11}, P_{12}, P_{13}>)$	(1-2) holds (in the execution of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
6 $\alpha(T_1, P_{11})$	since $\|T_1\| \leq \|P_{11}\| + \|P_{12}\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
7 $\beta(<T_{11}, T_{12}>, <P_{111}, P_{112}>)$	(1-5) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 7 in $\alpha(\ )$.)
8 $A(T_{11}, <P_{111}, P_{112}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (see line 3 in $\beta(\ )$.)
9 $\alpha(T_{11}, P_{111})$	since $\|T_{11}\| \leq \|P_{111}\| + \|P_{112}\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
10 return $<1, p_{11}>$	(1-4) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$. (see line 6 in $\alpha(\ )$.)
11 return $<1, p_{11}>$	return value of $A(T_{11}, <P_{111}, P_{112}>)$ (see line 2 in $A(\ )$.)
12 $A(T_{12}, <P_{112}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (see line 3 in $\beta(\ )$.)
13 $\alpha(T_{12}, P_{112})$	$<P_{112}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
14 return $<1, p_{11}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{12}) = lable(p_{112}) = b$ (see line 2 in $\alpha(\ )$.)
15 return $<1, p_{11}>$	return value of $A(T_{12}, <P_{112}>)$ (see line 2 in $A(\ )$.)
16 return $<2, p_{11}>$	the result of the 1st **while**-loop in the execution of $\beta(<T_{11}, T_{12}>, <P_{111}, P_{112}>)$.
17 return $<1, p_1>$	return value of $\alpha(T_1, P_{11})$ (Since $lable(p_{11}) = lable(t_1) = c$ and $d(p_{11}) = 2$, return $<1, p_1>$. see line 8 in $\alpha(\ )$.)
18 return $<1, p_1>$	return value of $A(T_1, <P_{11}, P_{12}, P_{13}>)$ (see line 2 in $A(\ )$.)
19 return $<1, p_1>$	return value of $\alpha(T_1, P_1)$ (see line 4 in $\alpha(\ )$.)
20 return $<1, p_1>$	return value of $A(T_1, G)$ (see line 2 in $A(\ )$.)
21 $A(T_2, G)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (see line 3 in $\beta(\ )$.)
22 $\alpha(T_2, P_1)$	since $\|T_2\| \leq \|P_1\| + \|P_2\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
23 $A(T_2, <P_{11}, P_{12}, P_{13}>)$	(1-2) holds (in the execution of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
24 $\beta(<T_{21}, T_{22}>, <P_{11}, P_{12}, P_{13}>)$	$\|T_2\| > \|P_{11}\| + \|P_{12}\|$. Call $\beta(\ )$ (see line 3 in $A(\ )$.)
25 $A(T_{21}, <P_{11}, P_{12}, P_{13}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (line 3 in $\beta(\ )$.)
26 $\alpha(T_{21}, P_{11})$	since $\|T_{21}\| \leq \|P_{11}\| + \|P_{12}\|$, call $\alpha(\ )$ (line 2 in $A(\ )$.)
27 $\beta(<T_{211}, T_{212}>, <P_{11}>)$	(1-3) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (line 5 in $\alpha(\ )$.)
28 $A(T_{211}, <P_{11}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (see line 3 in $\beta(\ )$.)
29 $\alpha(T_{211}, P_{11})$	$<P_{11}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
30 return $<1, p_{11}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{211}) = lable(p_{111}) = a$ (see line 2 in $\alpha(\ )$.)
31 return $<1, p_{11}>$	return value of $A(T_{211}, <P_{11}>)$ (see line 2 in $A(\ )$.)
32 $A(T_{212}, <P_{11}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (see line 3 in $\beta(\ )$.)
33 $\alpha(T_{212}, P_{11})$	$<P_{11}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
34 return $<1, p_{11}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{212}) = lable(p_{111}) = a$ (see line 2 in $\alpha(\ )$.
35 return $<1, p_{11}>$	return value of $A(T_{212}, <P_{11}>)$ (see line 2 in $A(\ )$.)
36 $A(T_{212}, <P_{112}>)$	*supplement computation* (done by a series of A-function calls.)
37 $\alpha(T_{212}, P_{112})$	$<P_{112}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
38 return $<0, p_{112}>$	(1-1) holds, but $lable(t_{212}) \neq label(t_{212})$. Return $<0, p_{112}>$ (line 3 in $\alpha(\ )$).
39 return $<0, p_{112}>$	return value of $A(T_{212}, <P_{112}>)$ (see line 2 in $A(\ )$.)
40 return $<1, p_{11}>$	result of the 2nd **while**-loop of $\beta(\ )$ (lines 12 – 16 in $\beta(\ )$).
41 return $<1, p_{11}>$	return value of $\alpha(T_{21}, P_{11})$ (see line 5 in $\alpha(\ )$.)
42 return $<1, p_{11}>$	return value of $A(T_{21}, <P_{11}, P_{12}, P_{13}>)$ (line 2 in $A(\ )$).
43 $A(T_{22}, <P_{11}, P_{12}, P_{13}>)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn (line 3 in $\beta(\ )$.)

90.	$\alpha(T_{22}, P_{11})$	since $\|T_{22}\| \le \|P_{11}\| + \|P_{12}\|$, call $\alpha(\ )$ (line 2 in $A(\ )$.)
91.	$A(T_{22}, <P_{111}, P_{112}>)$	(1-2) holds (in the execution of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
92.	$\alpha(T_{22}, P_{111})$	since $\|T_1\| \le \|P_{111}\| + \|P_{112}\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
93.	$\beta(<T_{221}>, <P_{111}>)$	(1-5) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 7 in $\alpha(\ )$.)
94.	$A(T_{221}, <P_{111}>)$	in 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (line 3 in $\beta(\ )$.)
95.	$\alpha(T_{221}, P_{111})$	$<P_{111}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
96.	return $<1, p_{11}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{221}) = lable(p_{111}) = a$ (see line 2 in $\alpha(\ )$.)
97.	return $<1, p_{11}>$	return value of $A(T_{221}, <P_{111}>)$ (see line 2 in $A(\ )$.)
98.	return $<1, p_{11}>$	return value of $\beta(<T_{221}>, <P_{111}>)$ (see lines 2 – 7, 8 in $\beta(\ )$.)
99.	return $<1, p_{11}>$	return value of $\alpha(T_{22}, P_{111})$ (see line 7, 9 in $\alpha(\ )$.)
100.	return $<1, p_{11}>$	return value of $A(T_{22}, <P_{111}, P_{112}>)$ (see line 2 in $A(\ )$.)
101.	return $<1, p_{11}>$	return value of $\alpha(T_{22}, P_{11})$ (see line 4 in $\alpha(\ )$.)
102.	return $<1, p_{11}>$	return value of $A(T_{22}, <P_{11}, P_{12}, P_{13}>)$ (see line 3, 7 in $A(\ )$.)
103.	$A(T_{22}, <P_{112}>)$	*supplement computation* (done by a series of *A*-function calls.)
104.	$\alpha(T_{22}, P_{112})$	$<P_{112}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
105.	$\beta(<T_{221}>, <P_{112}>)$	(1-3) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 5 in $\alpha(\ )$.)
106.	$A(T_{221}, <P_{112}>)$	in the1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (line 3 in $\beta(\ )$.)
107.	$\alpha(T_{221}, P_{112})$	$<P_{112}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
108.	return $<0, p_{112}>$	(1-1) holds, but $lable(t_{212}) \neq lable(p_{112})$. Return $<0, p_{112}>$. (line 3 in $\alpha(\ )$.)
109.	return $<0, p_{112}>$	return value of $A(T_{221}, <P_{112}>)$ (see line 2 in $A(\ )$.)
110.	return $<0, p_{112}>$	return value of $\beta(<T_{221}>, <P_{112}>)$ (see lines 2 – 7, 9 in $\beta(\ )$.)
111.	return $<0, p_{112}>$	return value of $\alpha(T_{22}, P_{112})$ (see line 5 in $\alpha(\ )$.)
112.	return $<0, p_{112}>$	return value of $A(T_{22}, <P_{112}>)$ (see line 2 in $A(\ )$.)
113.	return $<1, p_{11}>$	result of the 1st and 2nd **while**-loop of $\beta(\ )$ (see lines 2 – 7 and 12 – 16 in $\beta(\ )$.)
114.	return $<1, p_{11}>$	return value of $A(T_2, <P_{11}, P_{12}, P_{13}>)$ (see lines 3, 6 in $A(\ )$.
115.	return $<1, p_{11}>$	return value of $\alpha(T_2, P_1)$ (see line 4 in $\alpha(\ )$.)
116.	return $<1, p_{11}>$	return value of $A(T_2, G)$ (see line 2 in $A(\ )$.)
117.	$A(T_3, G)$	in the 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
118.	$\alpha(T_3, P_1)$	since $\|T_3\| \le \|P_1\| + \|P_2\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
119.	$A(T_3, <P_{11}, P_{12}, P_{13}>)$	(1-2) holds (in the exec. of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
120.	$\alpha(T_3, P_{11})$	since $\|T_3\| \le \|P_{11}\| + \|P_{12}\|$, call $\alpha(\ )$ (see line 2 in $A(\ )$.)
121.	$A(T_3, <P_{111}, P_{112}>)$	(1-2) holds (in the execution of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
122.	$\alpha(T_3, P_{111})$	since $\|T_3\| \le \|P_{111}\| + \|P_{112}\|$, call $\alpha(\ )$ (line 2 in $A(\ )$.)
123.	$\beta(<T_{31}>, <P_{111}>)$	(1-3) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 5 in $\alpha(\ )$.)
124.	$A(T_{31}, <P_{111}>)$	in 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
125.	$\alpha(T_{31}, P_{111})$	$<P_{111}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
126.	return $<0, p_{111}>$	(1-1) holds, but $lable(t_{31}) \neq lable(p_{111})$. Return $<0, p_{112}>$ (see line 3 in $\alpha(\ )$.)
127.	return $<0, p_{111}>$	return value of $A(T_{31}, <P_{111}>)$ (see line 2 in $A(\ )$.)
128.	return $<0, p_{111}>$	return value of $\beta(<T_{31}>, <P_{111}>)$ (see lines 2 – 7, 9 in $\beta(\ )$.)
129.	return $<0, p_{111}>$	return value of $\alpha(T_3, P_{111})$ (see line 5 in $\alpha(\ )$.)
130.	return $<0, p_{111}>$	return value of $A(T_3, <P_{111}, P_{112}>)$ (see line 2 in $A(\ )$.)
131.	return $<0, p_{111}>$	return value of $\alpha(T_3, P_{11})$ (see line 4 in $\alpha(\ )$.)
132.	return $<0, p_{111}>$	return value of $A(T_3, <P_{11}, P_{12}, P_{13}>)$ (see line 3, 6 in $A(\ )$.)
133.	return $<0, p_{111}>$	return value of $\alpha(T_3, P_1)$ (see line 4 in $\alpha(\ )$.)
134.	return $<0, p_{111}>$	return value of $A(T_3, G)$ (see line 2 in $A(\ )$.)
135.	$A(T_2, <P_{12}, P_{13}>)$	*supplement checking* (done by a series of *A*-function calls.)
136.	$\beta(<T_{21}, T_{22}>, <P_{12}, P_{13}>)$	$\|T_2\| > \|P_{12}\| + \|P_{13}\|$. Call $\beta(\ )$ (line 3 in $A(\ )$.)
137.	$A(T_{21}, <P_{12}, P_{13}>)$	in the 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
138.	$\alpha(T_{21}, P_{12})$	since $\|T_{21}\| \le \|P_{12}\| + \|P_{13}\|$, call $\alpha(\ )$ (line 2 in $A(\ )$.)
139.	$\beta(<T_{211}, T_{212}>, <P_{121}>)$	(1-5) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 7 in $\alpha(\ )$.)
140.	$A(T_{211}, <P_{121}>)$	in the 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
141.	$\alpha(T_{211}, P_{121})$	$<P_{121}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
142.	return $<1, p_{12}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{211}) = lable(p_{121}) = a$ (see line 2 in $\alpha(\ )$.)
143.	return $<1, p_{12}>$	return value of $A(T_{211}, <P_{121}>)$ (see line 2 in $A(\ )$.)
144.	return $<1, p_{12}>$	return value of $\beta(<T_{211}, T_{212}>, <P_{121}>)$ (see lines 2 - 7, 8 in $\beta(\ )$.)
145.	return $<1, p_1>$	return value of $\alpha(T_{21}, P_{12})$ (see line 7, 9 in $\alpha(\ )$.)
146.	return $<1, p_1>$	return value of $A(T_{21}, <P_{12}, P_{13}>)$ (see line 2 in $A(\ )$.)
147.	$A(T_{22}, <P_{13}>)$	in the 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
148.	$\alpha(T_{22}, P_{13})$	$<P_{13}>$ contains a single tree. Call $\alpha(\ )$ (see line 2 in $A(\ )$.)
149.	$\beta(<T_{221}>, <P_{13}>)$	(1-3) holds (in the execution of $\alpha(\ )$.) Call $\beta(\ )$ (see line 5 in $\alpha(\ )$.)
150.	$A(T_{221}, <P_{13}>)$	in the 1st **while**-loop of $\beta(\ )$, call *A*-functions in turn (see line 3 in $\beta(\ )$.)
151.	$\alpha(T_{221}, P_{13})$	$<P_{13}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
152.	return $<0, p_{13}>$	(1-1) holds, but $l(t_{221}) \neq l(p_{13})$. Return $<0, p_{13}>$. (line 3 in $\alpha(\ )$.)
153.	return $<0, p_{13}>$	return value of $A(T_{221}, <P_{13}>)$ (see line 2 in $A(\ )$.)
154.	return $<0, p_{13}>$	return value of $\beta(<T_{221}>, <P_{13}>)$ (see lines 2 – 7, 9 in $\beta(\ )$.)
155.	return $<0, p_{13}>$	return value of $\alpha(T_{22}, P_{13})$. (see line 5 in $\alpha(\ )$.)
156.	return $<0, p_{13}>$	return value of $A(T_{22}, <P_{13}>)$. (see line 2 in $A(\ )$.)
157.	return $<1, p_1>$	return value of $\beta(<T_{21}, T_{22}>, <P_{12}, P_{13}>)$. (see lines 2 – 7, 8 in $\beta(\ )$.)
158.	return $<1, p_1>$	return value of $A(T_2, <P_{12}, P_{13}>)$. (see lines 3, 4 – 5 in $A(\ )$.)
159.	$A(T_3, <P_{13}>)$	*supplement checking*
160.	$\alpha(T_3, P_{13})$	$<P_{13}>$ contains a single tree. Call $\alpha(\ )$ (line 2 in $A(\ )$.)
161.	return $<1, p_1>$	(1-4) holds (in the exec. of $\alpha(\ )$.) Return $<1, p_{11}>$. (see line 6 in $\alpha(\ )$.)
162.	return $<1, p_{11}>$	return value of $A(T_3, <P_{13}>)$ (see line 2 in $A(\ )$.)
163.	return $<3, p_1>$	return value of $\beta(<T_1, T_2, T_3>, G)$. (see line 14, 16 in $\beta(\ )$.)
164.	return $<3, p_1>$	return value of $A(T, G)$. (see line 3, 7 in $A(\ )$.)

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 $A(T_1, G, p_{111})$ 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):

$$A(T_2, G, p_1) \rightarrow \alpha(T_2, P_1, p_1) \rightarrow A(T_2, <P_{11}, P_{12}, P_{13}>, p_1).$$

Since p_{11}'s parent is p_1 (instead of an ancestor of p_1), $A(T_2, <P_{11}, P_{12}, P_{13}>, p_1)$ c annot return a useful left corner. So the corresponding computation needn't be performed and we simply set its return value to be $<0, p_{111}>$ (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):

$$A(T_3, G, p_1) \rightarrow \alpha(T_3, P_1, p_1) \rightarrow A(T_3, <P_{11}, P_{12}, P_{13}>, p_1).$$

For the same reason, $A(T_3, <P_{11}, P_{12}, P_{13}>, p_1)$ will not be carried out, either, but with $<0, p_{111}>$ 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:	explanation:
1. $A(T, G, p_{111})$	$A(T, G)$ begins. Initially, the cut is set to be $\rho(G) = p_{111}$.
2. $\beta(<T_1, T_2, T_3>, G, p_{111})$	since G is a forest and $\|T\| > \|P_1\| + \|P_2\|$, call $\beta(\ )$ and the cut is transferred to $\beta(\ )$ (line 4 in $A(\ )$.)
3. $A(T_1, G, p_{111})$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is transferred to $A(\ )$ (line 3 in $\beta(\ )$.)
4. $\alpha(T_1, P_1, p_{111})$	since $\|T_1\| \leq \|P_1\| + \|P_2\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
5. $A(T_1, <P_{11}, P_{12}, P_{13}>, p_{111})$	(1-2) holds (in the exec. of $\alpha(\ )$.) Call $A(\ )$ and the cut is transferred to $A(\ )$ (line 4 in $\alpha(\ )$.)
$\alpha(T_1, P_{11}, p_{111})$	since $\|T_1\| \leq \|P_{11}\| + \|P_{12}\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
6. $\beta(<T_{11}, T_{12}>, <P_{111}, P_{112}>, p_{111})$	(1-5-ii) holds (in the exec. of $\alpha(\ )$.) Call $\beta(\ )$ and the cut is transferred to $\beta(\ )$ (line 8 in $\alpha(\ )$.)
7. $A(T_{11}, <P_{111}, P_{112}>, p_{111})$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is transferred to $A(\ )$ (line 3 in $\beta(\ )$.)
8. $\alpha(T_{11}, P_{111}, p_{111})$	since $\|T_{11}\| \leq \|P_{111}\| + \|P_{112}\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
9. return $<1, p_{11}>$	(1-4) holds (in the exec. of $\alpha(\ )$.) Return $<1, p_{11}>$. (line 6 in $\alpha(\ )$.)
10. return $<1, p_{11}>$	return value of $A(T_{11}, <P_{111}, P_{112}>, p_{111})$ (see line 3 in $A(\ )$.
11. $A(T_{12}, <P_{112}>, p_{112})$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is changed to p_{112} (line 3 in $\beta(\ )$.)
12. $\alpha(T_1, P_{112}, p_{112})$	$<P_{112}>$ contains a single tree. Call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
13. return $<1, p_{11}>$	(1-1) holds (in the execution of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{12}) = lable(p_{112}) = b$ (see line 2 in $\alpha(\ )$.)
14. return $<1, p_{11}>$	return value of $A(T_{12}, <P_{112}>, p_{112})$ (see line 3 in $A(\ )$.)
15. return $<2, p_{11}>$	the result of the 1st **while**-loop in the execution of $\beta(<T_{11}, T_{12}>, <P_{111}, P_{112}>, p_{111})$.
16. return $<1, p_1>$	return value of $\alpha(T_1, P_{11}, p_{111})$ (Since $lable(p_{11}) = lable(t_1) = c$ and $d(p_{11}) = 2$, return $<1, p_1>$. see line 8 in $\alpha(\ )$.)
17. return $<1, p_1>$	return value of $A(T_1, <P_{11}, P_{12}, P_{13}>, p_{111})$ (see line 3 in $A(\ )$.)
18. return $<1, p_1>$	return value of $\alpha(T_1, P_1, p_{111})$ (see line 4 in $\alpha(\ )$.)
19. return $<1, p_1>$	return value of $A(T_1, G, p_{111})$ (see line 3 in $A(\ )$.)
20. $A(T_2, G, p_1)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is changed to p_1 (line 3 in $\beta(\ )$.)
21. $\alpha(T_2, P_1, p_1)$	since $\|T_2\| \leq \|P_1\| + \|P_2\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
22. $A(T_2, <P_{11}, P_{12}, P_{13}>, p_1)$	(1-2) holds (in the exec. of $\alpha(\ )$.) Call $A(\ )$ (see line 4 in $\alpha(\ )$.)
23. return $<0, \rho(P_{11})>$	Here, the computation is cut off since p_{11}'s parent is not an ancestor of p_1.
24. return $<0, p_{111}>$	return value of $\alpha(T_2, P_1, p_1)$ (see line 4 in $\alpha(\ )$.)
25. return $<0, p_{111}>$	return value of $A(T_2, G, p_1)$ (see line 3 in $A(\ )$.)
26. $A(T_3, G, p_1)$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is still p_1 (line 3 in $\beta(\ )$.)
27. $\alpha(T_3, P_1, p_1)$	since $\|T_3\| \leq \|P_1\| + \|P_2\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
28. $A(T_3, <P_{11}, P_{12}, P_{13}>, p_1)$	(1-2) holds (in the exec. of $\alpha(\ )$.) Call $A(\ )$ (line 4 in $\alpha(\ )$.)
29. return $<0, \rho(P_{11})>$	Here, the computation is cut off again since p_{11}'s parent is not an ancestor of p_1.
30. return $<0, p_{111}>$	the return value of $\alpha(T_3, P_1, p_1)$ (see line 4 in $\alpha(\ )$.)
31. return $<0, p_{111}>$	the return value of see $A(T_3, G, p_1)$ (line 3 in $A(\ )$.)
32. $A(T_2, <P_{12}, P_{13}>, p_{12})$	*supplement checking* (done by a series of A-function calls.) The cut is set to p_{12} since T_1 contains P_{11}.
33. $\beta(<T_{21}, T_{22}>, <P_{12}, P_{13}>, p_{12})$	$\|T_2\| > \|P_{12}\| + \|P_{13}\|$. Call $\beta(\ )$ and the cut is transferred to $\beta(\ )$ (line 4 in $A(\ )$.)
34. $A(T_{21}, <P_{12}, P_{13}>, p_{12})$	in 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is transferred to $A(\ )$ (line 3 in $\beta(\ )$.)
35. $\alpha(T_{21}, P_{12}, p_{12})$	since $\|T_{21}\| \leq \|P_{12}\| + \|P_{13}\|$, call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
36. $\beta(<T_{211}, T_{212}>, <P_{121}>, p_{121})$	(1-5-i) holds (in the exec. of $\alpha(\ )$.) Call $\beta(\ )$ and the cut transferred to $\beta(\ )$ is changed to p_{121} (line 7 in $\alpha(\ )$.)
37. $A(T_{211}, <P_{121}>, p_{121})$	in the 1st **while**-loop of $\beta(\ )$, call A-functions in turn and the cut is transferred to $A(\ )$ (line 3 in $\beta(\ )$.)
38. $\alpha(T_{211}, P_{121}, p_{121})$)	$<P_{121}>$ contains a single tree. Call $\alpha(\ )$ and the cut is transferred to $\alpha(\ )$ (line 3 in $A(\ )$.)
39. return $<1, p_{12}>$	(1-1) holds (in the exec. of $\alpha(\ )$.) Return $<1, p_{11}>$ since $lable(t_{211}) = lable(p_{121}) = a$ (line 2 in $\alpha(\ )$.)
40. return $<1, p_{12}>$	return value of $A(T_{211}, <P_{121}>)$ (see line 2 in $A(\ )$.)
41. return $<1, p_{12}>$	return value of $\beta(<T_{211}, T_{212}>, <P_{121}>)$ (see lines 2 - 7, 8 in $\beta(\ )$.)
42. return $<1, p_1>$	return value of $\alpha(T_{21}, P_{12})$ (see line 7, 9 in $\alpha(\ )$.)
43. return $<1, p_1>$	return value of $A(T_{21}, <P_{12}, P_{13}>)$ (see line 2 in $A(\ )$.)

step-by-step trace:	explanation:
45. $A(T_{22}, <P_{13}>, p_{13})$	in the 1st **while**-loop of β( ), call *A*-functions in turn and the cut is changed to p_{13} (line 3 in β( ).)
46. $\alpha(T_{22}, P_{13}, p_{13})$	$<P_{13}>$ contains a single tree. Call α( ) and the cut is transferred to α( ) (line 3 in *A*( ).)
47. $\beta(<T_{221}>, <P_{13}>, p_{13})$	(1-3) holds (in the exec. of α( ).) Call β( ) and the cut is transferred to β( ) (line 5 in α( ).)
48. $A(T_{221}, <P_{13}>, p_{13})$	in the 1st **while**-loop of β( ), call *A*-functions in turn and the cut is transferred to *A*( ) (line 3 in β( ).)
49. $\alpha(T_{221}, P_{13}>, p_{13})$	$<P_{13}>$ contains a single tree. Call α( ) and the cut is transferred to α( ) (line 3 in *A*( ).)
50. return $<0, p_{13}>$	(1-1) holds, but $l(t_{221}) \neq l(p_{13})$. Return $<0, p_{13}>$. (line 3 in α( ).)
51. return $<0, p_{13}>$	return value of $A(T_{221}, <P_{13}>)$ (see line 3 in *A*( ).)
52. return $<0, p_{13}>$	return value of $\beta(<T_{221}>, <P_{13}>)$ (see lines 2 – 7, 9 in β( ).)
53. return $<0, p_{13}>$	return value of $\alpha(T_{22}, P_{13})$. (see line 5 in α( ).)
54. return $<0, p_{13}>$	return value of $A(T_{22}, <P_{13}>)$. (see line 2 in *A*( ).)
55. return $<1, p_1>$	return value of $\beta(<T_{21}, T_{22}>, <P_{12}, P_{13}>)$. (see lines 2 – 7, 8 in β( ).)
56. return $<1, p_1>$	return value of $A(T_2, <P_{12}, P_{13}>)$. (see lines 3, 4 – 5 in *A*( ).)
57. $A(T_3, <P_{13}>, p_{13})$	*supplement checking* (the cut transferred to *A*( ) is set to be p_{13}.)
58. $\alpha(T_3, P_{13}, p_{13})$	$<P_{13}>$ contains a single tree. Call α( ) and the cut is transferred to α( ) (line 2 in *A*( ).)
59. return $<1, p_1>$	(1-4) holds (in the exec. of α( ).) Return $<1, p_{11}>$. (see line 6 in α( ).)
60. return $<1, p_1>$	return value of $A(T_3, <P_{13}>)$ (see line 2 in *A*( ).)
61. return $<3, p_1>$	return value of $\beta(<T_1, T_2, T_3>, G)$ (see line 14, 16 in β( ).)
62. return $<3, p_1>$	return value of $A(T, G)$. (see line 3, 7 in *A*( ).)

Appendix IV Correctness of the Algorithm with cuts

In this Appendix, we prove the correctness of the Algorithm $A(T, G, v)$, where $T = <t; T_1, ..., T_k>$, $G = <P_1, ..., P_q>$. Initially, v is set to be $\rho(G)$ 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 three 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:

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

ii) label(t) ≠ label(v). (See line 5 in FUNCTION 4.)

In Case (i), the cut v for the β-function call is downgraded to v's first child. It is because if $<T_1, ..., T_k>$ 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 β-function call is still v since any left corner returned by the β-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 α-function*, as well as *Cut Propagation in β-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 $T = <t; T_1, ..., T_k>$ and $G = <P_1, ..., P_q>$. 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 $<0, \rho(G)>$. □