A Linear-Space Top-down Algorithm for Tree Inclusion Problem

Yangjun Chen[1], Yibin Chen[2]

Dept. Applied Computer Science, University of Winnipeg, Canada

[1]y.chen@uwinnipeg.ca, [2]chenyibin@gmail.com

***Abstract*— We consider the following tree-matching problem: Given labeled, ordered trees *P* and *T*, can *P* be obtained from *T* by deleting nodes? Deleting a node *v* entails removing all edges incident to *v* and, if *v* has a parent *u*, replacing the edges from *u* to *v* by edges from *u* to the children of *v*. The best known algorithm for this problem needs O(|*T*|·|leaves(*P*)|) time and O(|leaves(*P*)|·*min*{D_T, |leaves(*T*)|} + |*T*| + |*P*|) space, where leaves(*T*) (resp. leaves(*P*)) stands for the set of the leaves of *T* (resp. *P*), and D_T (resp. D_P) for the height of *T* (resp. *P*). In this paper, we present an efficient algorithm that requires O(|*T*|·|leaves(*P*)|) time and O(|*T*| + |*P*|) space.**

I. INTRODUCTION

Let *T* be a tree and *v* be a node different of root in *T* with parent node *u*. Denote by *delete*(*T*, *v*) the tree obtained from *T* by removing the node *v*. The children of *v* become the children of *u* as illustrated in Fig. 1.

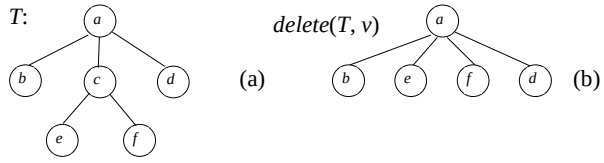

Fig. 1 The effect of removing a node from a tree

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

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

we have $T_k = P$? If this is the case, we say, *P* is included in *T* [9]. Such a problem is called the *tree inclusion problem*. Ordered labeled trees appear in various research fields, including programming language implementation, natural language processing, and molecular biology.

As an example [9], consider querying grammatical structures as shown in Fig. 2(a), which is the parse tree of a natural language sentence.

One might want to locate, say, those sentences that include a verb phrase containing the verb "reads" and after it a noun "book" followed by any adverb. This is exactly the sentences whose parse tree can be obtained by deleting some nodes from the tree shown in Fig. 2(a). See Fig. 2(b) for illustration. The ordered tree inclusion problem was initially introduced by Knuth [10], where only a sufficient condition for this problem is given. The tree inclusion has been suggested as an important primitive for expressing queries on structured document databases [4, 5, 6, 12]. A structured document database is considered as a collection of parse trees that represent the structure of the stored texts and the tree inclusion is used as a means of retrieving information from them. Another application of the ordered tree matching is the video content-based retrieval. According to Rui *et al*. [14], 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. In addition, the ordered tree matching can also be applied in the scene analysis, the computational biology (such as the RNA structure matching [11]), as well as in the data mining (such as the tree mining [15]).

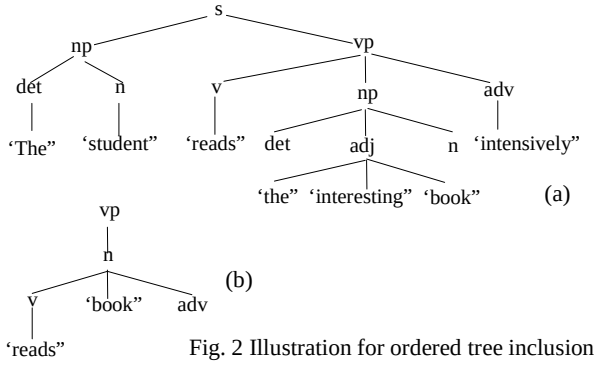

Fig. 2 Illustration for ordered tree inclusion

This problem has been the attention of much research. Kilpeläinen and Mannila [9] presented the first polynomial time algorithm using O(|*T*|·|*P*|) time and space. Most of the later improvements are refinements of this algorithm. In [13], Richter gave an algorithm using O(|α(*P*)|·|*T*| + *m*(*P*, *T*)·D_T) time, where α(*P*) is the alphabet of the labels of *P*, *m*(*P*, *T*) is the size of a set called *matches*, defined as all the pairs (*v*, *w*) ∈ *P* × *T* such that *label*(*v*) = *label*(*w*), and D_T (resp. D_P) is the depth of *T* (resp. *P*). Hence, if the number of matches is small, the time complexity of this algorithm is better than O(*T*|·|*P*|). The space complexity of the algorithm is O(|α(*P*)|·|*T*| + *m*(*P*, *T*)). In [3], a more sophisticated algorithm was presented using O(|*T*|·|leaves(*P*)|) time and O(|leaves(*P*)|·*min*{D_T, |leaves(*T*)|} + |*T*| + |*P*|) space. In [1], an efficient average case algorithm was discussed. Its average time complexity is O(|*T*|

978-1-4673-0719-2/12/$26.00 ©2012 IEEE

+ $C(P, T)\cdot|P|$), 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 $O(|T|\cdot|P|)$. In [2], another bottom-up algorithm is proposed. The time complexity of the algorithm is bounded by

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

But it is claimed that the algorithm needs only $O(|T| + |P|)$ space. A careful analysis reveals that the space complexity of the algorithm is the same as that of [3]. In the algorithm, a data structure $EMB(v)$ for each v in P is used to record deep occurrences of $P[v]$ in T. It is of size $O(|leaves(T)|)$ in the worst case. $EMB(v)$ is generated recursively and works in a way similar to the concept of *shell* discussed in [3]. So the analysis of *shell* applies to $EMB(v)$'s.

In our earlier work [7], a top-down algorithm was proposed with $O(|T| + |P|)$ space requirement. However, its time complexity is not polynomial, as shown in [11].

In this paper, we revisit this issue and present a new top-down algorithm to remove any redundancy of [10]. The time complexity of the new one is bounded by $O(|T|\cdot|leaves(P)|)$. Although the time complexity of our algorithm is comparable to Chen's algorithm [3], it is more efficient than Chen's since in Chen's algorithm each node in T will be checked against, besides some internal nodes, all the leaf nodes in P. But in our algorithm, a node in T may be checked so many times only when some conditions are satisfied.

More importantly, our algorithm needs only linear space $O(|T| + |P|)$.

The tree inclusion problem on unordered trees is *NP*-complete [9] and not discussed in this paper.

II. BASIC DEFINIION

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

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

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

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

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

Proof. See Exercise 2.3.2-20 in [10] (page 347).

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

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

Proof. The proof is trivial. □

In the following, we use the postorder numbers to define an ordering of the nodes of a forest F given by $v \prec v'$ iff $post(v) < post(v')$. Also, $v \preceq v'$ iff $v \prec v'$ or $v = v'$. Furthermore, we extend this ordering with two special nodes $\bot \prec v \prec \top$. The *left relatives*, $lr(v)$, of a node $v \in V(F)$ is the set of nodes that are to the left of v and similarly the *right relatives*, $rr(v)$, are the set of nodes that are to the right of v.

The following definition is due to [9].

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

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

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

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

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

Fig. 3 shows an example of an ordered inclusion.

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

Fig. 3(b) also shows an example of a root preserving embedding. According to [9], restricting to root-preserving

embedding does not lose generality. In fact, what can be found by the top-down algorithm to be discussed is a root-preserving tree embedding.

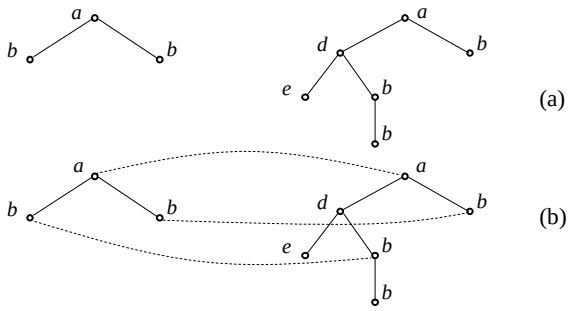

Fig. 3: (a) The tree on the left can be included in the tree on the right by deleting the nodes labelled : *d*, *e*, and *b*; (b) the embedding corresponding to (a).

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

III. Algorithm description

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

Consider a node v in $G = <P_1, ..., P_q>$ with children $v_1, ..., v_j$. We use a pair $<i, v>$ ($i \leq j$) to represent an ordered forest containing the first i subtrees of v: $<G[v_1], ..., G[v_i]>$. If v is p_v, or a node on the left-most path in P_1, $<i, v>$ is called a *left corner* of G. Especially, $<i, p_v>$ is a left corner, representing the first i trees in G: $P_1, ..., P_i$.

In addition, $\delta(v)$ represents a link from a node v to the left-most leaf node in $G[v]$, as illustrated in Fig. 4.

Let v' be a leaf node in G. $\delta(v')$ is defined to be a link to v' itself. So in Fig. 5, we have $\delta(v_1) = \delta(v_2) = \delta(v_3) = v_3$. We also denote by $\delta^{-1}(v')$ a set of nodes $\boldsymbol{x}$ such that for each $v \in \boldsymbol{x}$ $\delta(v) = v'$. Therefore, in Fig. 5, $\delta^{-1}(v_3) = \{v_1, v_2, v_3\}$, $\delta^{-1}(v_4) = \{v_4\}$, and $\delta^{-1}(v_5) = \{v_5\}$. The out-degree of v in a tree is denoted by $d(v)$ while the height of v is denoted by $h(v)$, defined to be the number of edges on the longest downward path from v to a leaf. The height of a leaf node is set to be 0.

As with [7], we arrange two functions: *top-down*(T, G) and *bottom-up*(T', G) to check tree inclusion, where T is a tree, and T' and G are two forests. However, different from [7], each of the two functions returns a left corner $<i, v>$ of G with the following properties:

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

In this sense, we say, $<i, v>$ is the *highest* and *widest* left corner which can be embedded in T (or in T').

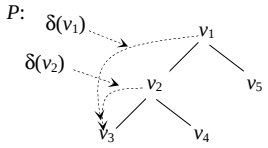

Fig. 4 A pattern tree

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

In [7], both *top-down*(T, G) and *bottom-up*(T', G) return an integer i to indicate that T embeds the first i trees in G. Although our algorithm follows the arrangement of [7], the main idea is quite different. It is not necessary to refer to [7] to understand the following discussion.

If the target is a tree and the pattern is a forest, we call the function *top-down.* If both the target and the pattern are forests, we call the function *bottom-up*. But during the computation, they will be called from each other.

In *top-down*(T, G), we need to handle two cases.

Case 1: $G = <P_1>$; or $G = <P_1, ..., P_q>$ ($q > 1$), but $|T| \leq |P_1| + |P_2|$. In this case, to find the highest and widest left corner $<i, v>$ that can be embedded in $T = <t; T_1, ..., T_k>$, the following checkings will be conducted:

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

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

iii) If $|T| \geq |P_1|$ and $h(t) \geq h(p_1)$, we further distinguish between two subcases:
   - label(t) = label(p_1). In this case, we will call *bottom-up*($<T_1, ..., T_k>$, $<P_{11}, ..., P_{1j}>$).
   - label(t) ≠ label(p_1). In this case, we will call *bottom-up*($<T_1, ..., T_k>$, $<P_1>$).

   In both cases, assume that the return value of *bottom-up*( ) is $<i, v>$. We need to perform a further checking:
   - If label(t) = label(v) and $i = d(v)$, the return value of *top-down*(T, G) is set to be <1, v's parent>.
   - Otherwise, the return value of *top-down*(T, G) is the same as $<i, v>$.

Case 2: $G = <P_1, ..., P_q>$ ($q > 1$), and $|T| > |P_1| + |P_2|$. In this case, we will call *bottom-up*($<T_1, ..., T_k>$, G). Assume that the return value of *bottom-up*($<T_1, ..., T_k>$, G) is $<i, v>$. The following checkings will be continually conducted.

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

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

The following is a formal description of the algorithm. In the process, each node t in T is associated with a data structure, referred to as $\kappa(t)$. Initially, each $\kappa(t)$ is set to ϕ. Each time a call of the form *top-down*($T[t]$, G') returns a left corner $<i, v>$, $\kappa(t)$ will be changed to $<i, v>$, where G' is a forest 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 G. This value is mainly used in *bottom-up*( ) to avoid redundancy. However, for simplicity, in the following algorithm $\kappa(t)$ is not explicitly represented.

function *top-down*(T, G)
input: $T = <t; T_1, ..., T_k>$, $G = <P_1, ..., P_q>$.
output: $<i, v>$ specified above.
begin
1. **if** (q = 1 or $|T| \le |P_1| + |P_2|$)
2. **then**
   {let $P_1 = <p_1; P_{11}, ..., P_{1j}>$; (**Case* 1*)
3. **if** t is a leaf **then** {
   {let $\delta(p_1) = v$; (**Case* 1 - (i)*)
4. **if** label(t) = label(v) **then** return <1, v's parent>
   else return <0, v>; }
5. **if** ($|T| < |P_1|$ or $h(t) < h(p_1)$)
   then return *top-down*(T, $<P_{11}, ..., P_{1j}>$); (*Case 1 - (ii)*)
6. **if** label(t) = label(p_1) (**Case* 1 - (iii)*)
7. **then** { **if** p_1 is a leaf **then** {$v := p_1$'s parent; $i := 1$;}
8. **else** {$<i, v> :=$ *bottom-up*($<T_1, ..., T_k>$, $<P_{11}, ..., P_{1j}>$);
9. **if** label(t) = label(v) and $i = d(v)$
   then {$v := v$'s parent; $i := 1$; }
10. }
11. **else** $<i, v> :=$ *bottom-up*($<T_1, ..., T_k>$, $<P_1>$);
   (*If label(t) ≠ label(p_1), call *bottom-up*( ).*)
12. return $<i, v>$;
13. }
14. **else**
   {$<i, v> :=$ *bottom-up*($<T_1, ..., T_k>$, G); (**Case* 2*)
15. **if** $v \ne p_1$'s parent **then** (**Case* 2 - (iv)*)
16. **if** (label(t) = label(v)) and $i = d(v)$ **then** return <1, v's parent>;
17. return $<i, v>$; (**Case* 2 - (v)*)
18. }
end

In the above algorithm, we first check whether $q = 1$ or $|T| \le |P_1| + |P_2|$ (see line 1). If it is the case we have *Case* 1 and then lines 2 - 13 are executed. In this process, all the three subcases (i), (ii), and (iii) are checked. If $q > 1$ and $|T| > |P_1| + |P_2|$, we have *Case* 2 and lines 14 - 18 will be carried out, in which we first call *bottom-up*($<T_1, ..., T_k>$, G). Depending on its return value, (vi) or (v) is conducted.

bottom-up(T', G) is designed to handle the case that both T' and G are forests made up of a set of subtrees rooted at nodes that are consecutive siblings in T and P, respectively. Let $T' = <T_1, ..., T_k>$ 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 *bottom-up*(T', G), we will make a series of calls *top-down*(T_l, $<P_{j_l}, ..., P_q>$), where l = 1, ..., k, $j_1 = 1$, and $j_1 \le j_2 \le ... \le j_h \le q$ (for some $h \le k$), controlled as follows.

1. Two index variables l, j are used to scan $T_1, ..., T_k$ and $P_1, ..., P_q$, respectively. (Initially, l is set to 1, and j is set to 0.) They also indicate that $<P_1, ..., P_j>$ has been successfully embedded in $<T_1, ..., T_l>$.
2. Let $<i_l, v_l>$ be the return value of *top-down*(T_l, $<P_{j+1}, ..., P_q>$). If $v_l = p_1$'s parent, set j to be $j + i_l$. Otherwise, j is not changed. Set l to be $l + 1$. Go to (2).
3. The loop terminates when all T_l's or all P_j's are examined.

If $j > 0$ when the loop terminates, *bottom-up*(T', G) returns $<j, p_1$'s parent>, indicating that T' contains $P_1, ..., P_j$.

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

i) Let $<i_1, v_1>, ..., <i_k, v_k>$ be the return values of *top-down*(T_1, $<P_1, ..., P_q>$), ..., *top-down*(T_k, $<P_1, ..., P_q>$), respectively. Since $j = 0$, each $v_l \in \delta^{-1}(v')$ (l = 1, ..., k), where v' is the left-most leaf in P_1.
ii) If each $i_l = 0$, return <0, left-most leaf of P_1>. Otherwise, there must be some v_l's such that $i_l > 0$. We call such a node a *non-zero point*. Find the first non-zero point v_f with children $w_1, ..., w_s$ such that v_f is not a descendant of any other non-zero point. Then, we will check $<T_{f+1}, ..., T_k>$ against $<P[w_{i_f+1}], ..., P[w_s]>$. Let x ($0 \le x \le s - i_f$) be a number such that $<P[w_{i_f+1}], ..., P[w_{i_f+x}]>$ can be embedded in $<T_{f+1}, ..., T_k>$. The return value of *bottom-up*(T', G) should be set to $<i_f + x, v_f>$.

In the *bottom-up* process, $\kappa(t)$ can be used to avoid redundant computation. Concretely, each time before we make a call of the form *top-down*(T_l, $<P_j, ..., P_q>$), we will calculate a function κ-*checking*(t_l, p_j) defined below to determine whether this call can be skipped over, where t_l and p_j are the roots of T_l and P_j, respectively.

function κ-*checking*(t, p)
input: t - a node in T; p - a node in G.
output: ϕ or $<i, v>$ specified above.
begin
1. **if** $\kappa(t_l) \ne \phi$ **then** {
2. let $\kappa(t_l) = <i, v>$;
3. **if** $i = 0$ **then** return ϕ;
4. **if** $i > 0$, $\delta(v) = \delta(p)$, and p is equal to v's first child or an ancestor of v's first child
5. **then** return $<i, v>$;
6. **if** $i > 0$, $\delta(v) = \delta(p)$, and p is a descendant of v's first child
7. **then** return <d(p's parent), p's parent>.
8. **else** return ϕ.
end

Only when κ-*checking*(t_l, p_j) returns ϕ, *top-down*(T_l, $<P_j, ..., P_q>$) will be carried out. Otherwise, we use the value of κ-*checking*(t_l, p_j) as the return value of *top-down*(T_l, $<P_j, ..., P_q>$).

In terms of the above discussion, we arrange a new subprocedure to check a T_l against a forest $<P_j, ..., P_q>$, doing the same work as the *top-down* process but with κ-*checking*(t_l, p_j) being used to avoid unnecessary checkings.

function *top-down*-κ(T, $<P_1, ..., P_q>$)
input: T - a tree; $<P_1, ..., P_q>$ - a forest.
output: $<v, i>$ specified above.
begin
1. **if** κ-*checking*(t, p_1) = ϕ

then $<i, v> := top\text{-}down(T, <P_1, ..., P_q>)$
2. **else** $<i, v> = \kappa\text{-}checking(t, p_1)$;
3. return $<i, v>$;

end

In the following algorithm, we use *top-down*-κ( ), instead of *top-down*( ), to check a tree against a forest.

function *bottom-up*(T', G)
input: $T' = <T_1, ..., T_k>$, $G = <P_1, ..., P_q>$
output: $<i, v>$ specified above.
begin
1. $l := 1$; $j := 0$;
2. **while** ($j < q$ and $l \leq k$) **do** (*main checking*)
3. { $<i_l, v_l> := top\text{-}down\text{-}\kappa(T_l, <P_{j+1}, ..., P_q>)$
4. **if** ($v_l = p_1$'s parent and $i_l > 0$) **then** $j := j + i_l$;
5. $l := l + 1$; }
6. **if** $j > 0$ **then** return $<j, p_1$'s parent$>$;
7. **if** for all $<i_l, v_l>$'s $i_l = 0$ **then** return <0, left-most leaf in G>
8. **else** { let v_f be the first non-zero point such that it is not a descendant of any other non-zero point;
9. let $w_1, ..., w_s$ be the children of v_f;
10. $l := f + 1$; $j := i_f$;
11. **while** ($j < s$ and $l \leq k$) **do** (*supplement checking*)
12. { $<i_l, v_l> := top\text{-}down\text{-}\kappa(T_l, <G[w_{j+1}], ..., G[w_s]>)$;
13. **if** ($v_l = v_f$ and $i_l > 0$) **then** $j := j + i_l$;
14. $l := l + 1$; }
15. return $<j, v_f>$;
16. }

end

In *bottom-up*(T', G), we have two *while*-loops: one from line 2 to 5 and another from line 11 to 14. In the first *while*-loop, we check $<T_1, ..., T_k>$ against $<P_1, ..., P_q>$, referred to as the *main bottom-up* checking (or simply the *main checking*). In this checking, each T_l is checked one by one, by repeatedly calling *top-down*-κ(T_l, $<P_{j+1}, ..., P_q>$) (line 3), by which κ-*checking*(t_l, p_{j+1}) is used to remove redundancy (see lines 1 - 2 in *top-down*-κ( )).

In the second *while*-loop, we do a *supplement checking*. This is carried out only when the following two conditions are satisfied (see lines 6 and 7):

(1) $j = 0$, and

(2) There exists at least a non-zero point $<i_l, v_l>$ (return value of *top-down*-κ(T_l, $<P_1, ..., P_q>$) such that $i_l > 0$.

We refer to these two conditions as the *supplement checking condition*.

Let v_f be the first non-zero point such that v_f is not a descendant of any other non-zero point. Let $w_1, ..., w_s$ be the children of v_f. In the supplement checking, we will check $<T_{f+1}, ..., T_k>$ against $<G[w_{i_f+1}], ..., G[w_s]>$ (see lines 10 - 16.)

IV. CONCLUSION

In this paper, a new algorithm is proposed to improve the algorithm discussed in [7]. The main idea behind it is to let any subprocedure call return a pair to indicate a subtree (subforest) embedding while in [7], only a single integer is returned to indicate whether a whole forest (or the first several subtrees of the forest) is embedded by the corresponding target subtree. Together with a simple data structure associated with each node in the target tree to transfer the result obtained in a previous step to the next step computation to avoid any useless effort, high performance is achieved. The time complexity of the new algorithm is bounded by $O(|T| \cdot |leaves(P)|)$ while the space requirement is bounded by $O(|T| + |P|)$, where T and P are a target and a pattern tree, respectively.

REFERENCES

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

[2] P. Bille and I.L. Gørtz, An Ordered Tree Inclusion Algorithm Based on Dynamic Tree Labeling, in *Proc.*32th Intl. Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 3580, 2005, pp. 66-77.

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

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

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

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

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

[8] 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.

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

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

[11] 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).

[12] 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.

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

[14] 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.

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