Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 98 (2006) 253–262
Information Processing Letters
www.elsevier.com/locate/ipl

A new tree inclusion algorithm

Yangjun Chen [*,1], Yibin Chen

Department of Applied Computer Science, University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9

Received 9 September 2005; accepted 28 November 2005

Available online 30 March 2006

Communicated by L. Boasson

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 existing algorithm for this problem needs $O(|T||\text{leaves}(P)|)$ time and $O(|\text{leaves}(P)|\min\{D_T, |\text{leaves}(T)|\})$ space, where leaves(P) (leaves(T)) stands for the number of the leaves of $P(T)$, and D_T for the height of T. In this paper, we present a new algorithm that requires $O(|T|\min\{D_P, |\text{leaves}(P)|\})$ time and no extra space, where D_P represents the height of P.

1. Introduction

Let T be a tree and v be a node 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 becomes children of u as illustrated in Fig. 1.

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 $v_1, \ldots, v_k$ of nodes such that for

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

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

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

As an example, consider querying grammatical structures as shown in Fig. 2, 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

* Corresponding author.
E-mail addresses: ychen2@uwinnipeg.ca (Y. Chen), chenyibin@hotmail.com (Y. Chen).

1 The author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

0020-0190/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2005.11.022

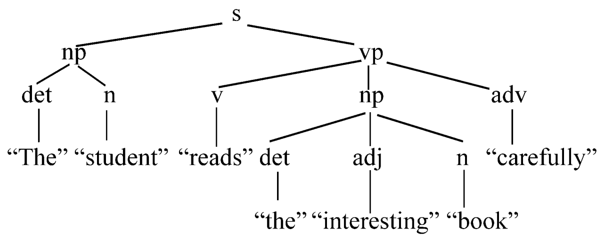

Fig. 2. The parse tree of a sentence.

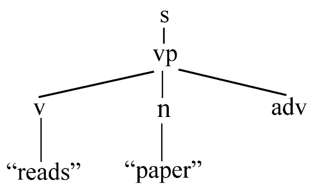

Fig. 3. An included tree of the parse tree.

by deleting some nodes from the tree shown in Fig. 2. (See Fig. 3 for illustration.)

The ordered tree inclusion problem was initially introduced by Knuth [5], 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 [3]. 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. This problem has been the attention of much research. Kilpelainen and Mannila [4] presented the first polynomial time algorithm using $O(|T| \cdot |P|)$ time and space. Most of the later improvements are refinements of this algorithm. In [6], Richter gave an algorithm using $O(|\alpha(P)| \cdot |T| + m(P, T) \cdot D_T)$ time, where $\alpha(P)$ is the alphabet of the labels of P, $m(P, T)$ is the size of a set called *matches*, defined as all the pairs $(v, w) \in P \times T$ such that $\text{label}(v) = \text{label}(w)$, and D_T is the depth of T. 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 $O(|\alpha(P)| \cdot |T| + m(P, T))$. In [2], a more complex algorithm was presented using $O(|T| \cdot |\text{leaves}(P)|)$ time and $O(|\text{leaves}(P)| \cdot \min\{D_T, |\text{leaves}(T)|\})$ space. In [1], an efficient average case algorithm was discussed. Its average time complexity is $O(|T| + C(S, T) \cdot |P|)$, 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 $O(|T| \cdot |P|)$.

All the above algorithms work in a bottom-up way. In this paper, we propose a new algorithm by integrating a top-down process into a bottom-up computation. It needs only $O(|T| \cdot \min\{D_P, |\text{leaves}(P)|\})$ time and no extra space, where D_P represents the height of P.

2. Orderings and embeddings

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 $\langle T_1, \ldots, T_k\rangle$, where $k \geqslant 0$. The trees $T_1, \ldots, T_k$ are the subtrees of the root of T or the immediate subtrees of tree T, and k is the out-degree of the root of T. A tree with the root t and the subtrees $T_1, \ldots, T_k$ is denoted by $\langle t; T_1, \ldots, T_k\rangle$. The roots of the trees $T_1, \ldots, T_k$ are the children of t and siblings of each other. Also, we call $T_1, \ldots, T_k$ the sibling trees of each other. In addition, $T_1, \ldots, T_{i-1}$ are called the left sibling trees of T_i, and T_{i-1} the direct 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 $\langle T\rangle$. 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 F to a forest G, we mean functions mapping the nodes of F onto the nodes of G. 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 is called a subtree of F rooted at v, denoted by $F[v]$.

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

Using preorder and postorder numbers, the ancestorship can be checked as follows.

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 2.3.2-2 in [5]. □

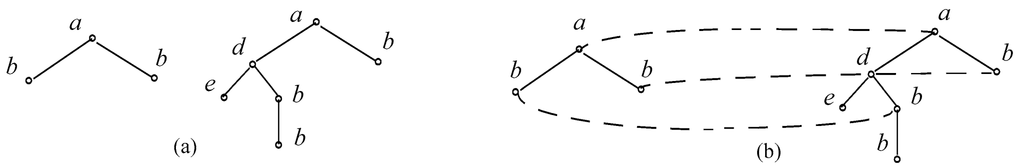

Fig. 4. (a) The tree on the left can be included in the tree on the right by deleting the nodes labeled: d, e and b. (b) The embedding corresponding to (a).

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

Lemma 2. *Let v and u be nodes in a forest F. Then, v appears on the left side of u if and only if $pre(v) < pre(u)$ and $post(v) < post(u)$.*

Proof. The proof is trivial. □

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

(i) $\text{label}(v) = \text{label}(\phi(v))$ (label preservation condition);
(ii) v is an ancestor of u iff $\phi(v)$ is an ancestor of $\phi(u)$, i.e., $pre(v) < pre(u)$ and $post(u) < post(v)$ iff $pre(\phi(v)) < pre(\phi(u))$ and $post(\phi(u)) < post(\phi(v))$ (ancestor condition);
(iii) v is to the left of u iff $\phi(v)$ is to the left of $\phi(u)$, i.e., $pre(v) < pre(u)$ and $post(v) < post(u)$ iff $pre(\phi(v)) < pre(\phi(u))$ and $post(\phi(v)) < post(\phi(u))$ (Sibling condition).

Fig. 4 shows an example of an ordered inclusion.

Now we give two concepts that are useful to explain the main idea of our algorithm. Throughout the rest of the paper, we refer to the labeled ordered trees simply as trees.

Definition 2. Let P and T be trees. A root-preserving embedding of P in T is an embedding ϕ of P in T such that $\phi(root(P)) = root(T)$. If there is a root-preserving embedding of P in T, we say that the root of T is an occurrence of P.

Fig. 4(b) shows an example of a root preserving embedding. According to [4], restricting to root-preserving embedding does not lose generality.

Obviously, to find a root-preserving embedding, we have to work in a top-down fashion.

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*, $\text{lr}(v)$, of a node $v \in V(F)$ is the set of nodes that are to the left of v and similarly the *right relatives*, $\text{rr}(v)$, are the set of nodes that are to the right of v.

The next definition gives a name for the embeddings that are searched for by the bottom-up procedure discussed in [4].

Definition 3. Let $G = \langle P_1, \ldots, P_k\rangle$ and F be a forests, and $\boldsymbol{E}$ be a collection of embeddings of G in F. An embedding $\phi \in \boldsymbol{E}$ is a left embedding of $\boldsymbol{E}$ if for every $\gamma \in \boldsymbol{E}$, we have

$$post\big(\phi\big(root(P_k)\big)\big) \leqslant post\big(\gamma\big(root(P_k)\big)\big).$$

A left embedding of the set of all embeddings of G in F is a left embedding of G in F. Obviously, if G is included in F, there must exist a left embedding of G in F.

3. Algorithm

The algorithm to be presented attempts to find the number of subtrees $j\,(\geqslant 0)$ within an ordered forest $G = \langle P_1, \ldots, P_q\rangle$ $(q \geqslant 1)$, which are embedded in a target tree T. If $j = q$, we say that G is embedded in T. If $j < q$, then only the trees $P_1, \ldots,$ and P_j are embedded in T. Let $p_1, \ldots, p_q$ and t be the roots of $P_1, \ldots, P_q$ and T, respectively. Since a forest does not have a root, we use a virtual node p_v to serve as a substitute for $root(G)$. Thus, $root(G)$ will return p_v if $G = \langle P_1, \ldots, P_q\rangle$ with $q > 1$, and will return p_1 if $q = 1$.

There are three cases that need to be considered when designing an algorithm to check the tree embedding:

Case 1: $root(G) \neq p_v$ (i.e., $G = \langle P_1\rangle$ and $root(G) = p_1$), and $\text{label}(p_1) \neq \text{label}(t)$. If G is embedded in T, then there must exist a subtree T_i of t such that it contains the whole G. The algorithm should return 1 if an embedding can be found and 0 if it cannot. (See Fig. 5 for illustration.)

Case 2: $root(G) \neq p_v$ (i.e., $G = \langle P_1\rangle$ and $root(G) = p_1$), and $\text{label}(p_1) = \text{label}(t)$. Let $\langle P_{11}, \ldots, P_{1l}\rangle$ $(l \geqslant 0)$

be the forest of subtrees of p_1 and $\langle T_1, \ldots, T_k \rangle$ the forest of subtrees of t. If G is embedded in T, there must exist two sequences of integers: $k_1, \ldots, k_g$ and $l_1, \ldots, l_g$ ($g \leqslant l$) such that T_{k_i} includes $\langle P_{1(l_{i-1}+1)}, \ldots, P_{1l_i} \rangle$ ($i = 1, \ldots, g$, $l_0 = 0$, $l_g = l$), where $\langle P_{1(l_{i-1}+1)}, \ldots, P_{1l_i} \rangle$ represents a forest containing subtrees $P_{1(l_{i-1}+1)}, \ldots,$ and P_{1l_i}. Thus, if $l_g = l$, the algorithm should return 1 since we have a root preserving inclusion of G in T. Otherwise, it should return 0. (See Fig. 6 for illustration.)

Case 3: *root*$(G) = p_v$ and there exists an integer j ($0 \leqslant j \leqslant q$) such that $\langle P_1, \ldots, P_j \rangle$ is included in T. If $j = q$, then the whole G is embedded in T. There are two possibilities to be considered when looking for j. The first possibility is similar to Case 2, where there are two sequences of integers: $k_1, \ldots, k_g$ and $l_1, \ldots, l_g$ ($g \leqslant q$) that represent the order, in which the subtrees of *root*(G) are embedded in the subtrees of *root*(T). $j = l_g$. The second possibility is that there exists a root preserving inclusion of P_1 in T, i.e., label(p_1) = label(t) and the subtrees of p_1 are included in the subtrees of t. In this case, $j = 1$. (See Fig. 7 for illustration.)

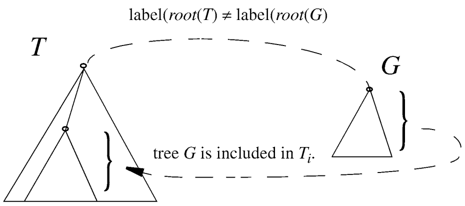

Fig. 5. Illustration for Case 1.

In order to understand how to arrive at a tree inclusion algorithm based on the checking of these three cases, we first consider the trivial case that G contains only one single node p. In such a case, we have only Cases 1 and 2 to be encountered, and the problem can be easily solved using a preorder tree traversal starting at *root*(T). Thus, if label$(p) \neq$ label(t), the subsequent operations are to compare all child nodes of t with p. Once the algorithm finds a node t' such that label(p) = label(t'), it returns the result to the original caller of the algorithm.

Now we need to consider the case of attempting to find an embedding of an order forest G consisting of a list of single nodes $p_1, \ldots, p_q$ into a tree T of height 2 with root t and child nodes $t_1, \ldots, t_k$ (Case 3). Since the nodes are ordered, it is simple to scan from t_1 to t_k to find $k_1, \ldots, k_s$ and $l_1, \ldots, l_s$ ($s \leqslant l$) such that label(t_{k_i}) = label(p_{l_i}). If after this scan we have $j = l_s = 0$, then the remaining possibility is that label(t) = label(p_1). If it is the case, we have $j = 1$.

If the height of T is greater than two, a more complicated bottom-up process is required to handle Case 3. Let t_s be a node in T with subtrees $T_{s1}, \ldots, T_{sk}$. Let G_s be a subset of single nodes $\langle p_f, \ldots, p_q \rangle$ of G, which have not been found in the part left to any ancestor of t_s in T. Denote j_{return} the number of the subtrees that are found in the tree rooted at t_s. Then, in order to find j_{return}, we first need to search T_{s1}, and find the number of the nodes j_f, which are included in T_{s1}. If $j_f > 0$, then G_s is reduced to $\langle p_r, \ldots, p_q \rangle$, where $r = j_f + f$. This process is then repeated for $T_{s2}, \ldots, T_{sk}$ to find $j_{f+1}, j_{f+2}, \ldots, j_{f+k-1}$ until either all sub-

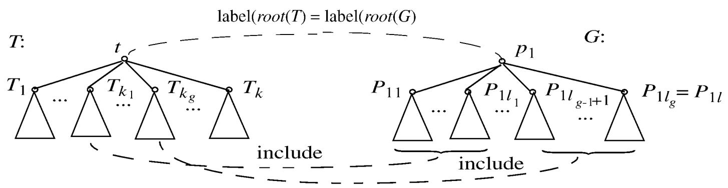

Fig. 6. Illustration for Case 2.

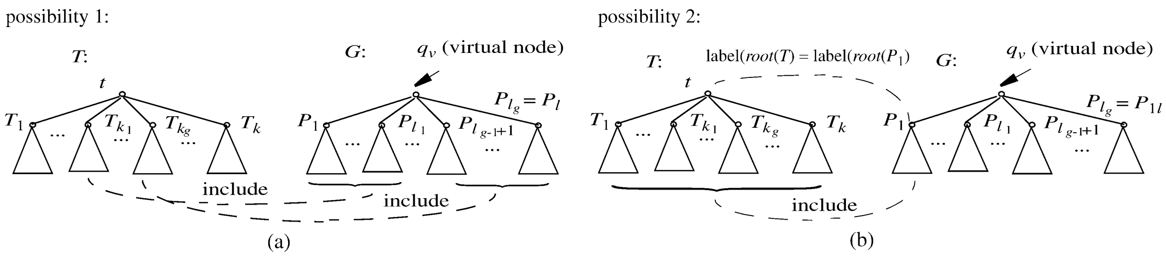

Fig. 7. Illustration for Case 3.

trees of t_s are traversed or G_s becomes empty. If after all the subtrees are searched and $j_{\text{total}} = j_f + j_{f+1} + \cdots + j_{f+k-1} = 0$, an additional check for comparing label(p_f) and label(t_s) is required to ensure that all possible match patterns are considered. Therefore, for $G = \langle p_1, \ldots, p_q\rangle$, which is a forest containing a list of single nodes, the result should be

if $j_{\text{total}} > 0,\ j = f - 1 + j_{\text{return}} = f - 1 + j_{\text{total}}$
else if label(p_f) = label(t_s),
$j = f - 1 + j_{\text{return}} = f - 1 + 1 = f$
else $j = f - 1 + j_{\text{return}} = f - 1$.

In the general Case 3, G is a list of subtrees $\langle P_1, \ldots, P_q\rangle$ with roots $\langle p_1, \ldots, p_q\rangle$, the same analysis as above applies. That is, we will first check $\langle T_1, \ldots, T_k\rangle$ against $\langle P_1, \ldots, P_q\rangle$. If the return value g is larger than 0, we set j equal to g. If $g = 0$, we will check label(p_1) against label(t_s) and $\langle T_1, \ldots, T_k\rangle$ against the subtrees $\langle P_{11}, \ldots, P_{1l}\rangle$ of p_1. It is exactly Case 2. If label(p_1) = label(t_s) and $\langle T_1, \ldots, T_k\rangle$ includes $\langle P_{11}, \ldots, P_{1l}\rangle$, we set $j = 1$, which shows a root preserving inclusion. Otherwise, $j = 0$. Obviously, we have height(P_{1x}) < height(P_1) for $1 \leqslant x \leqslant l$. Since for each label match found, the height of the sub-pattern is effectively reduced by at least 1, we are guaranteed to eventually arrive at a pattern forest consisting of only single nodes (trivial case).

In the general Case 2, G is a non-trivial tree $\langle p; P_1, \ldots, P_q\rangle$ and $root(t) = p$. We need to check $\langle T_1, \ldots, T_k\rangle$ against $\langle P_1, \ldots, P_q\rangle$. To do that, we will check T_i in turn against $\langle P_{l_{i-1}+1}, \ldots, P_q\rangle$ ($i = 1, \ldots, j$, $l_0 = 0$). So the control will be switched over to Case 3. This emergence of Case 3 within Case 2, as well as Case 2 within Case 3 (see the above discussion), hints a recursive solution using two functions that interleavingly call each other.

Finally, we notice that Case 1 always ends up with Case 2 unless the whole tree T is searched and no nodes are found matching p_1.

An observation shows that we can arrange the computation in such a way that Case 2 is always handled as early as possible by the size checking as follows:

when we check T against a forest $\langle P_1, \ldots, P_q\rangle$, if $|P_1| \leqslant |T| < |P_1| + |P_2|$, the control is switched over to Case 2 immediately to check T against P_1.

In fact, Case 2 is handled in a top-down way, which effectively restrict the searching range of the subsequent bottom-up computation.

The following Algorithm 1 consists of two functions: *top-down-process*(T, G) and *bottom-up-process*(T', G'), where T and T' are trees, G' is a forest, and G can be a tree or a forest. In the algorithm, a tree is always handled as a tree with a virtual root.

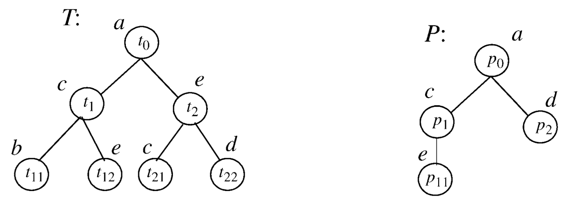

Fig. 8. Two trees.

Intuitively, *top-down-process*(T, G) is designed to handle Cases (i), (ii), and the second possibility in Case (iii) while *bottom-up-process*(T', G') is for the first possibility in Case (iii).

In *top-down-process*(T, G), we will first check whether *root*(G) is virtual (line 1). If it is the case, we will further check whether $|T| < |P_1| + |P_2|$ or p has only one child (line 2). If both do not hold, we will try to find a j such that $\langle P_1, \ldots, P_j\rangle$ is covered by the subtrees of t by invoking *bottom-up-process*(T, G) (line 4). If $j = 0$, we will check whether the subtrees of t covers the subtrees of P_1's root by invoking *bottom-up-process*(T, P_1) if label(t) = label(P_1's root) (see lines 6–7). If $|T| < |P_1| + |P_2|$ or p has only one child, we will directly check whether T includes P_1. This is done by assigning P_1 to G (see line 3) and then going to line 9. In line 10, we compare label(t) and label(*root*(G)). If label(t) = label(*root*(G)), we will check whether all the subtrees of *root*(G) are covered by the subtrees of t by changing *root*(G) to a virtual node and then invoking *bottom-up-process*(T, G) (see lines 10–13). If label(t) $\neq$ label(*root*(G)), we attempt to find an i such that T_i includes the whole G (see lines 15–19).

In *bottom-up-process*(T, G), we try to find two sequences of integers: $k_1, \ldots, k_j$ and $l_1, \ldots, l_j$ ($j \leqslant l$) such that T_{k_i} includes $\langle P_{1(l_{i-1}+1)}, \ldots, P_{1l_i}\rangle$ ($i = 1, \ldots, j$, $l_0 = 0$).

Example 1. Consider two ordered, labeled trees T and P shown in Fig. 8, where each node in T is identified with t_i, such as t_0, t_1, t_{11}, and so on; and each node in P is identified with p_j. In addition, each subtree rooted at $t_i(p_j)$ is represented by $T_i(P_j)$.

In the following step-by-step trace (given in Table 1), we use j_x^{down} to represent the return value of a call *top-down-process*(T_x, G') for some sub-forest G' in P and j_y^{up} the return value of a call *bottom-up-process*(T_y, G'') for some sub-forest G'' in P. In addition, we use p to represent a virtual node.

function *top-down-process*(T, G)
input: $T = \langle t; T_1, \ldots, T_k \rangle$, $G = \langle p; P_1, \ldots, P_q \rangle$ (* p may or may not be a virtual node *)
output: if $root(G)$ is virtual, returns $j \geqslant 0$; else returns 1 if T includes G; otherwise returns 0.
begin
1. **if** $root(G)$ is virtual
2. **then** {**if** ($|T| < |P_1| + |P_2|$ or p has only one child)
3. **then** $G := P_1$;
4. **else** { $j :=$ *bottom-up-process*(T, G);
5. **if** ($j = 0$ and label(t) = label(P_1's root)) (* second possibility in Case 3 *)
6. **then** {change P_1's root to a virtual node; $x :=$ *bottom-up-process*(T, P_1);
7. **if** ($x =$ the number of the children of P_1's root) **then** $j := 1$ **else** $j := 0$;}
8. return j; }}
9. **if** $|T| < |G|$ return 0;
10. **else** {**if** (label(t) = label(p)) (* handling Case 2 *)
11. **then** { $p :=$ virtual node;
12. $j :=$ *bottom-up-process*(T, G);
13. **if** ($j = l$) **then** return 1 **else** 0;}
14. **else** { **if** t is a leaf **then** return 0; (* handling Case 1 *)
15. $i := 1$;
16. **while** ($i \leqslant k$) **do**
17. { **if** *top-down-process*$(T_i, G) > 0$ **then** return 1;
18. $i := i + 1$;}
19. return 0;}}
end

function *bottom-up-process*(T, G)
input: $T = \langle t; T_1, \ldots, T_k \rangle$, $G = \langle p; P_1, \ldots, P_q \rangle$
output: j—an integer
begin
1. $j := 0$; $i := 1$; (* first possibility in Case 3 *)
2. **while** ($j < q$ and $i \leqslant k$) **do**
3. { $x :=$ *top-down-process*(T_i, G);
4. $j := j + x$; $G := \langle p; P_{j+1}, \ldots, P_q \rangle$; $i := i + 1$; }
end

Algorithm 1.

4. Correctness and computational complexities

In this section, we prove the correctness of the algorithm and analyze its computational complexities.

4.1. Correctness

Proposition 1. *Let $T = \langle t; T_1, \ldots, T_k \rangle$ and $G = \langle p; P_1, \ldots, P_q \rangle$. If p is a real node (i.e., not virtual), Algorithm top-down-process(T, G) returns 1 if T includes G; otherwise 0. If p is a virtual node, it returns an integer i, indicating that T includes $\langle P_1, \ldots, P_i \rangle$.*

Proof. We prove the proposition by induction on the sum of the heights of T and G, h. Without loss of generality, assume that height$(T) \geqslant 1$ and height$(G) \geqslant 1$.

Basic step. When $h = 2$, we consider two cases.

(i) Both T and G are singulars: r_1 and r_2.
(ii) T is a singular; but G is a set of nodes.

In Case (i), if r_1 and r_2 have the same label, the algorithm returns 1 (see lines 10–13); otherwise returns 0 (see line 14). In Case (ii), a virtual root p will be constructed for G. Then, lines 2–3 will be executed, leading to lines 9–14. According to the above discussion, the result must be correct.

When $h = 3$, we need to consider the following two cases.

(iii) T is a tree of height 2 and G is a set of nodes.
(iv) T is a singular; but G is a set of trees of height 2.

In Case (iii), a virtual root will be constructed for G. Then, line 4 will be executed to invoke *bottom-up-process*(T, G). Let $T = \langle t; t_1, \ldots, t_k \rangle$ and $G = \langle p; p_1, \ldots, p_q \rangle$, where t_i ($1 \leqslant i \leqslant k$) and p_j ($1 \leqslant j \leqslant q$) are single nodes and p is virtual. In the execution of *bottom-up-process*(T, G), we will have a series of calls of the form *top-down-process*(t_i, G_j), where $G_j = \langle p; p_j, \ldots, p_q \rangle$ ($1 \leqslant j$) and $p_1, \ldots, p_{j-1}$ are assumed to be covered by $t_1, \ldots, t_{i-1}$. Each of such calls

Table 1
Trace of Algorithm 1

Step-by-step trace:	Explanation:
top-down-process(T, P)	*top-down-process*(T, P) begins.
p is a real node	since p is a real node, go to line 9 to check T against P (line 1).
$\lvert T\rvert > \lvert\langle P\rangle\rvert$	compare the size of T and $\langle P\rangle$ (line 9).
$\text{label}(t_0) = \text{label}(p_0)$	check t_0 against p_0 (line 10).
bottom-up-process$(T, \langle p; P_1, P_2\rangle)$	call *bottom-up-process*$(T, \langle p; P_1, P_2\rangle)$ (line 12).
top-down-process$(T_1, \langle p; P_1, P_2\rangle)$	in the bottom-up process, call the top-down process.
$\lvert T_1\rvert = \lvert\langle P_1, P_2\rangle\rvert$	compare the size of T_1 and $\langle P_1, P_2\rangle$ (line 2).
bottom-up-process$(T_1, \langle p; P_1, P_2\rangle)$	in the top-down process, call the bottom-up process (line 4).
top-down-process$(T_{11}, \langle p; P_1, P_2\rangle)$	in the bottom-up process, call the top-down process.
$\lvert T_{11}\rvert < \lvert\langle P_1, P_2\rangle\rvert$	compare the size of T_{11} and $\langle P_1, P_2\rangle$ (line 2).
$\lvert T_{11}\rvert < \lvert\langle P_1\rangle\rvert$	compare the size of T_{11} and $\langle P_1\rangle$ (line 9).
return $j_{11}^{\text{down}} = 0$	since $\lvert T_{11}\rvert < \lvert\langle P_1\rangle\rvert$, *top-down-process*$(T_{11}, \langle p; P_1, P_2\rangle)$ returns 0.
top-down-process$(T_{12}, \langle p; P_1, P_2\rangle)$	in the bottom-up process, call the top-down process.
$\lvert T_{12}\rvert < \lvert\langle P_1, P_2\rangle\rvert$	compare the size of T_{12} and $\langle P_1, P_2\rangle$ (line 2).
$\lvert T_{12}\rvert < \lvert\langle P_1\rangle\rvert$	compare the size of T_{12} and $\langle P_1\rangle$ (line 9).
return $j_{12}^{\text{down}} = 0$	since $\lvert T_{12}\rvert < \lvert\langle P_1\rangle\rvert$, *top-down-process*$(T_{12}, \langle p; P_1, P_2\rangle)$ returns 0.
return $j_1^{\text{up}} = 0$	*bottom-up-process*$(T_1, \langle p; P_1, P_2\rangle)$ returns 0, which shows that the subtrees of T_1's root do not cover any subtree in $\langle P_1, P_2\rangle$.
$\text{label}(t_1) = \text{label}(p_1) = c$	since $\text{label}(t_1) = \text{label}(p_1)$, it is possible for T_1 itself to include P_1 (line 5).
bottom-up-process$(T_1, \langle p; P_1\rangle)$	p_1 is replaced with the virtual node p, call the bottom-up process.
top-down-process$(T_{11}, \langle p; P_{11}\rangle)$	in the bottom-up process, call the top-down process.
p has only one child	since p has only one child, check T_{11} against P_{11} immediately (line 2).
$\lvert T_{11}\rvert = \lvert\langle P_{11}\rangle\rvert$	compare the size of T_{11} and $\langle P_{11}\rangle$ (line 9).
$\text{label}(t_{11}) \neq \text{label}(p_{11})$	check t_{11} against p_{11} (line 10).
return $j_{11}^{\text{down}} = 0$	since $\text{label}(t_{11}) \neq \text{label}(p_{11})$, *top-down-process*$(T_{11}, \langle p; P_{11}\rangle)$ returns 0.
top-down-process$(T_{12}, \langle p; P_{11}\rangle)$	in the bottom-up process, call the top-down process.
p has only one child	since p has only one child, check T_{12} against P_{11} immediately (line 2).
$\lvert T_{12}\rvert = \lvert\langle P_{11}\rangle\rvert$	compare the size of T_{12} and $\langle P_{11}\rangle$ (line 9).
$\text{label}(t_{12}) = \text{label}(p_{11}) = e$	check t_{12} against p_{11} (line 10).
return $j_{12}^{\text{down}} = 1$	since $\text{label}(t_{12}) = \text{label}(p_{11})$, *top-down-process*$(T_{12}, \langle p; P_{11}\rangle)$ returns 1.
return $j_1^{\text{up}} = 1$	*bottom-up-process*$(T_1, \langle p; P_1\rangle)$ returns 1.
return $j_1^{\text{down}} = 1$	since $\text{label}(t_1) = \text{label}(p_1)$ and T_{12} includes $\langle P_{11}\rangle$, *top-down-process*$(T_1, \langle p; P_1, P_2\rangle)$ returns 1 (line 7).
top-down-process$(T_2, \langle p; P_2\rangle)$	in the bottom-up process, call the top-down process.
p has only one child	since p has only one child, check T_2 against P_2 immediately (line 2).
$\lvert T_2\rvert > \lvert\langle P_2\rangle\rvert$	compare the size of T_2 and $\langle P_2\rangle$ (line 9).
$\text{label}(t_2) \neq \text{label}(p_2)$	check t_2 against p_2 (line 10).
top-down-process$(T_{21}, \langle p; P_2\rangle)$	since $\text{label}(t_2) \neq \text{label}(p_2)$, we check *top-down-process*$(T_{21}, \langle p; P_2\rangle)$ and *top-down-process*$(T_{22}, \langle p; P_2\rangle)$ in turn (line 17).
p has only one child	since p has only one child, check T_{21} against P_2 immediately (line 2).
$\lvert T_{21}\rvert = \lvert\langle P_2\rangle\rvert$	compare the size of T_{21} and $\langle P_2\rangle$ (line 9).
$\text{label}(t_{21}) \neq \text{label}(p_2)$	check t_{21} against p_2 (line 10).
return $j_{21}^{\text{down}} = 0$	since $\text{label}(t_{21}) \neq \text{label}(p_2)$, *top-down-process*$(T_{21}, \langle p; P_2\rangle)$ return 0 (line 14).
top-down-process$(T_{22}, \langle p; P_2\rangle)$	call *top-down-process*$(T_{22}, \langle p; P_2\rangle)$ (line 17).
p has only one child	since p has only one child, check T_{22} against P_2 immediately (line 2).
$\lvert T_{22}\rvert = \lvert\langle P_2\rangle\rvert$	compare the size of T_{22} and $\langle P_2\rangle$ (line 9).
$\text{label}(t_{22}) = \text{label}(p_2)$	check t_{22} against p_2 (line 10).
return $j_{22}^{\text{down}} = 1$	since $\text{label}(t_{22}) = \text{label}(p_2)$, *top-down-process*$(T_{22}, \langle p; P_2\rangle)$ returns 1.
return $j_2^{\text{down}} = 1$	*top-down-process*$(T_2, \langle p; P_2\rangle)$ returns 1 (line 17).
return $j^{\text{up}} = 2$	*bottom-up-process*$(T, \langle p; P_1, P_2\rangle)$ return 2.
return $j^{\text{down}} = 1$	*top-down-process*(T, P) returns 1 (line 13).

is exactly Case (ii). Then, the result must be correct (see line 4 in *bottom-up-process*( )). Case (iv) is trivial. In this case, the algorithm returns 0 by executing line 8.

Induction hypothesis. Assume that when $h = l$, the proposition holds.

Consider $T = \langle t; T_1, \ldots, T_k\rangle$ and $G = \langle p; P_1, \ldots, P_q\rangle$ with $\text{height}(T) + \text{height}(G) = l + 1$. First, we assume that p is a real node. Obviously, we have $\text{height}(T_i) + \text{height}(G) \leqslant l$ and $\text{height}(T) + \text{height}(P_j) \leqslant l$. If $\text{label}(t) = \text{label}(p)$, the algorithm partitions the integer sequence: $1, \ldots, q$ into some subsequences: $\{j_0 + 1, \ldots, j_1\}, \{j_1 + 1, \ldots, j_2\}, \ldots, \{j_{m-1} + 1, \ldots, j_m\}$, where $j_0 = 0$ and $j_m \leqslant q$, such that each T_i ($i = 1, \ldots, m$; $m \leqslant k$) includes $\langle P_{j_{i-1}+1}, \ldots, P_{j_i} \rangle$ but not $\langle P_{j_{i-1}+1}, \ldots, P_{j_i}, P_{j_i+1} \rangle$. This is done by invoking *bottom-up-process*(T, G'), where G' is a forest obtained by replacing the root of G with a virtual node p' (see line 12). During the execution of *bottom-up-process*(T, G'), a series of calls of the form *top-down-process*(T_i, G_j) will be performed, where $G_j = \langle p'; P_j, \ldots, P_q \rangle$ $(1 \leqslant j)$ and $P_1, \ldots, P_{j-1}$ are covered by $T_1, \ldots, T_{i-1}$. In terms of the induction hypothesis, the partition is correct. Thus, the algorithm will return 1 if $j_m = q$, indicating that T includes G; otherwise 0 (see line 13). If $\text{label}(t) \neq \text{label}(p)$, algorithm will try to find the first T_i such that it includes the whole G. (See lines 14–19.) In terms of the induction hypothesis, the return value must be correct.

Now we assume that p is a virtual node. In terms of the induction hypothesis, the algorithm will find the correct integer i such that T includes $\langle P_1, \ldots, P_i \rangle$ (see lines 4 and 8). It completes the proof. □

4.2. Computational complexities

For a node v in T, denote $G(v)$ a list of nodes in G: $[u_1, u_2, \ldots, u_s]$ such that each u_i $(1 \leqslant i \leqslant s)$ is checked against v, and for any i and j, if $1 \leqslant i < j \leqslant s$, we have u_i checked before u_j. We will prove that $u_i \neq u_j$ if $i \neq j$ and all u_i's in $G(v)$ are on a same path.

Proposition 2. *Let $G(v) = [u_1, u_2, \ldots, u_s]$. Then, $u_i \neq u_j$ if $i \neq j$ and all u_i's are on a same path.*

Proof. Let $G = \langle p; P_1, \ldots, P_q \rangle$, where p may or may not be a virtual node. Let $v_1, v_2, \ldots, v_{c-1}, v_c = v$ be a path in T such that v is checked against u_1. Assume that u_1 appears in P_i for some i. According to the algorithm, v will be checked for a second time only when the following condition is satisfied:

bottom-up-process$(T[v_{c-1}], \langle p'; P_i[u_1], \ldots \rangle)$ returns 0, where p' is a virtual node; and $\text{label}(v_{c-1}) = \text{label}(u_1)$. (See lines 4–5 in *top-down-process*( ).)

In this case, *bottom-up-process*$(T[v_{c-1}], P_i[u])$ will be invoked (see line 6), where u is a virtual node and $P_i[u]$ is a subtree obtained by replacing u_1 with u. Thus, $v = v_c$ may be checked for a second time. However, v_c cannot be checked against u_1; but a node in $P_i[u_1]$. Therefore, u_2 must be different from u_1 but a descendant of u_1. In the same way, we can show that u_{i+1} is different from u_i but a descendant of u_i for $i = 1, \ldots, s - 1$. □

Proposition 3. *The time complexity of the algorithm is bounded by* $O(|T| \cdot \text{height}(G))$.

Proof. It can be easily derived from Proposition 2. □

Now we show that the time complexity of the algorithm with redundancy removing is also bounded by $O(|T| \cdot |\text{leaves}(P)|)$. To see this, we note that the repeated checking of a node in T is caused by the execution of line 6 in *top-down-process*( ). A necessary condition of this line's execution is that the function call *bottom-up-process*(T, G) in line 4 returns 0. However, to have this function call invoked, G must be a sub-forest; otherwise, the control switches over to line 9. Obviously, each sub-forest corresponds to a node in P, whose outdegree is larger than 1. Therefore, each repeated checking of a node in T corresponds to such a node. Denote A the set containing all those nodes in P, whose outdegree is larger than 1. Then, $|A| \leqslant |\text{leaves}(P)|$.

Proposition 4. *The time complexity of the algorithm with redundancy removing is bounded by* $O(|T| \cdot |\text{leaves}(P)|)$.

Proof. See the above analysis. □

Finally, we notice that during the execution of the algorithm, no data structures are created. Thus, the algorithm needs no extra space.

5. Experiments

We have compared our algorithm with the algorithm proposed by Kilpelainen and Mannila [4], and the algorithm by Chen [2] experimentally. All the algorithms are coded in Java 1.4 and tested on Pentium 4 1.6 GHz machine with 1 GB of RAM.

The target tree is generated from an XML document of Shakespeare's play—*The Tragedy of Antony and Cleopatra*. The tree generated contains 11 500 nodes and is of height 7.

We have tested two groups of pattern trees. For the first group, we generate pattern trees by randomly selecting nodes from the target tree. For the second group, each time we randomly select 2000 nodes, but with different heights. We record the numbers of label compar-

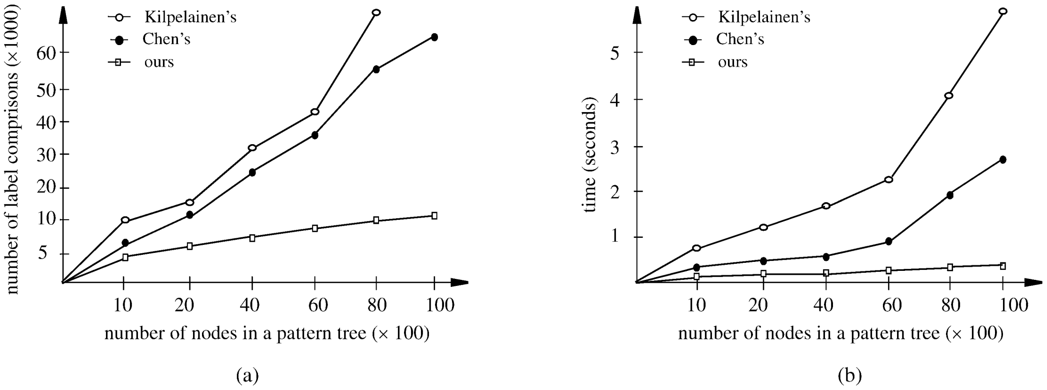

Fig. 9. Test results of the first group.

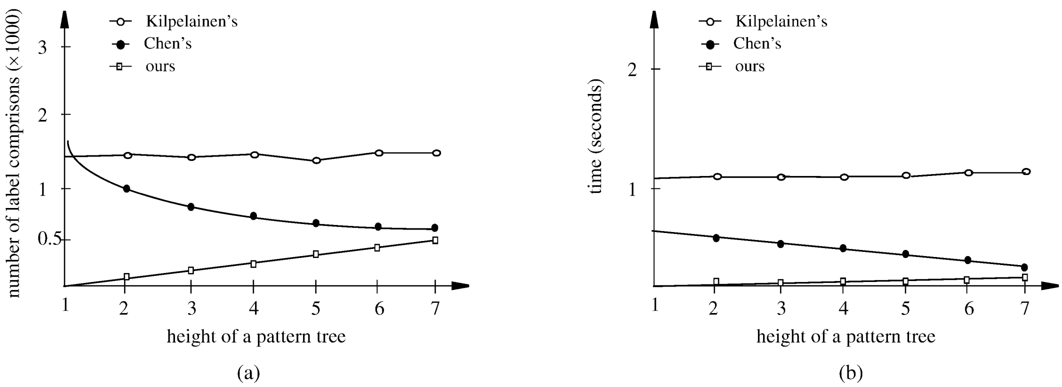

Fig. 10. Test results of the second group.

isons and elapsed times. For each execution, an average of 20 measurements is taken.

In Fig. 9(a) and (b), we show the numbers of label comparisons and the times spent on different executions, respectively.

From Fig. 9(a), we can see that our method outperforms Kilpelainen's and Chen's algorithms uniformly. In addition, we see that the number of label comparisons made by Kilpelainen's is not much higher than Chen's. However, as shown in Fig. 9(b), the time used by Kilpelainen's is much worse than Chen's. It is because by Kilpelainen's algorithm, a huge $(n \times m)$ matrix has to be created and initialized, where n stands for the number of the nodes in the target tree and m for the number of the nodes in the pattern tree. This dominates the execution time.

In Fig. 10(a) and (b), we demonstrate the result of the second group test. From Fig. 10(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 Chen's algorithm decreases as the height increases. Kilpelainen's algorithm is not sensitive to the height of patterns trees. Again, the time spent by Kilpelainen's algorithm is much worse than Chen's and ours.

6. Conclusion

In this paper, a new algorithm for checking the inclusion of a pattern tree P in a target tree T is discussed. The main idea of this is to integrate the top-down searching into a bottom-up computation. The algorithm needs $O(|T| \cdot \min\{D_P, |\text{leaves}(P)|\})$ time and no extra space, where D_P represents the height of P.

References

[1] L. Alonso, R. Schott, On the tree inclusion problem, in: Proceedings of Mathematical Foundations of Computer Science, 1993, pp. 211–221.
[2] W. Chen, More efficient algorithm for ordered tree inclusion, J. Algorithms 26 (1998) 370–385.
[3] H. Mannila, K.-J. Raiha, On Query Languages for the p-string data model, in: H. Kangassalo, S. Ohsuga, H. Jaakola (Eds.), Information Modelling and Knowledge Bases, IOS Press, Amsterdam, 1990, pp. 469–482.

[4] P. Kilpelainen, H. Mannila, Ordered and unordered tree inclusion, SIAM J. Comput. 24 (1995) 340–356.
[5] D.E. Knuth, The Art of Computer Programming, vol. 1, Addison-Wesley, Reading, MA, 1969.
[6] T. Richter, A new algorithm for the ordered tree inclusion problem, in: Proc. 8th Annual Symp. on Combinatorial Pattern Matching (CPM), in: Lecture Notes in Comput. Sci. (LNCS), vol. 1264, Springer, Berlin, 1997, pp. 150–166.

Further reading

[7] H. Andre-Joesson, D. Badal, Using signature files for querying time-series data, in: Proc. 1st European Symp. on Principles of Data Mining and Knowledge Discovery, 1997.
[8] Y. Chen, On the signature trees and balanced signatures, in: 22nd Int. Conf. on Data Engineering, Tokyo, Japan, April 5–8, 2005.
[9] S. Christodoulakis, C. Faloutsos, Design consideration for a message file server, IEEE Trans. Software Engrg. 10 (2) (1984) 201–210.
[10] R. Cole, R. Hariharan, P. Indyk, Tree pattern matching and subset matching in deterministic $O(n \log^3 m)$ time, in: Proceedings of the Tenth Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 1999, pp. 245–254.
[11] W.W. Chang, H.J. Schek, A signature access method for the STARBURST database system, in: Proc. 19th VLDB Conf., 1989, pp. 145–153.
[12] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, A. Pathria, Multimedia document presentation, information extraction and document formation in MINOS—A model and a system, ACM Trans. Office Inform. Systems 4 (4) (1986) 345–386.
[13] C. Faloutsos, Access methods for text, ACM Comput. Surv. 17 (1) (1985) 49–74.
[14] C. Faloutsos, Signature files, in: W.B. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures & Algorithms, Prentice-Hall, New Jersey, 1992, pp. 44–65.
[15] C. Faloutsos, R. Lee, C. Plaisant, B. Shneiderman, Incorporating string search in hypertext system: User interface and signature file design issues, HyperMedia 2 (3) (1990) 183–200.
[16] Y. Ishikawa, H. Kitagawa, N. Ohbo, Evaluation of signature files as set access facilities in OODBs, in: Proc. of ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., May 1993, pp. 247–256.
[17] W. Lee, D.L. Lee, Signature file methods for indexing object-oriented database systems, in: Proc. ICIC'92—2nd Int. Conf. on Data and Knowledge Engineering: Theory and Application, Hongkong, Dec. 1992, pp. 616–622.
[18] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, J. Zobel, Atlas: A nested relational database system for text application, IEEE Trans. Knowledge Data Engrg. 7 (3) (1995) 454–470.
[19] H.S. Yong, S. Lee, H.J. Kim, Applying signatures for forward traversal query processing in object-oriented databases, in: Proc. of 10th Internat. Conf. on Data Engineering, Houston, TX, Feb. 1994, pp. 518–525.