On the top-down evaluation of tree inclusion problem

Yangjun Chen* and Yibin Chen

Department of Applied Computer Science,
University of Winnipeg,
Winnipeg, Manitoba, R3B 2E9, Canada
E-mail: ychen2@uwinnipeg.ca
E-mail: chenyibin@gmail.com
*Corresponding author

Abstract: We consider the following tree-matching problem: Given labelled, 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 a new algorithm that requires O(|T|·|leaves(P)|) time but only O(|T| + |P|) space.

Keywords: ordered labelled trees; tree inclusion; TreeBank.

Reference to this paper should be made as follows: Chen, Y. and Chen, Y. (xxxx) 'On the top-down evaluation of tree inclusion problem', *Int. J. Information Technology, Communications and Convergence*, Vol. X, No. Y, pp.000–000.

Biographical notes: Yangjun Chen received his BS in Information System Engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and PhD in Computer Science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a Post-Doctor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a Senior Engineer at the German National Research Center of Information Technology (GMD) for more than two years. Since 2000, he has been a Professor in the Department of Applied Computer Science at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, document databases, constraint satisfaction problem, graph theory and combinatorics. He has more than 150 publications in these areas.

Yibin Chen received his BS degree from University of Waterloo in 2006 and his Master degree from University of Toronto, in 2008. He is a Software Engineer.

This paper is a revised and expanded version of a paper entitled 'A new top-down algorithm for tree inclusion' presented at Cyberc'2010, HuangShan, China, 10–12 October 2010.

1 Introduction

Ordered labelled trees are trees whose nodes are labelled and in which the left-to-right order among siblings is significant. Given two ordered labelled trees T and P, called the *target* and *pattern*, respectively, the *tree inclusion problem* is to determine whether it is possible to obtain P from T by deleting nodes. Deleting a node v in T means making each child of v a child of the parent of v and then removing v. The children of v are placed in the position of v in the left-to-right order among the siblings of v. Ordered labelled trees appear in various research fields, including programming language implementation, natural language processing, and molecular biology.

As an example, consider querying grammatical structures as shown in Figure 1, which is the parse tree of a natural language sentence.

Figure 1 The parse tree of a sentence

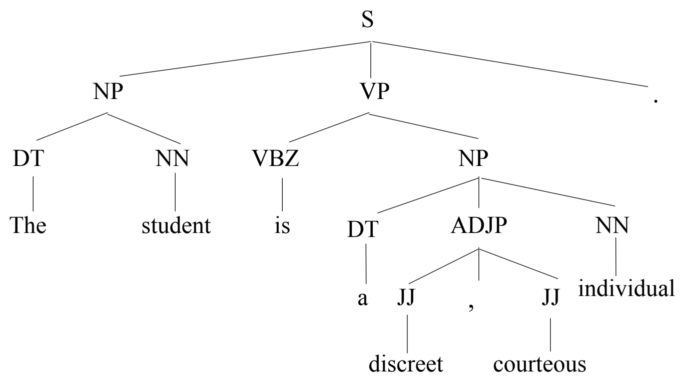
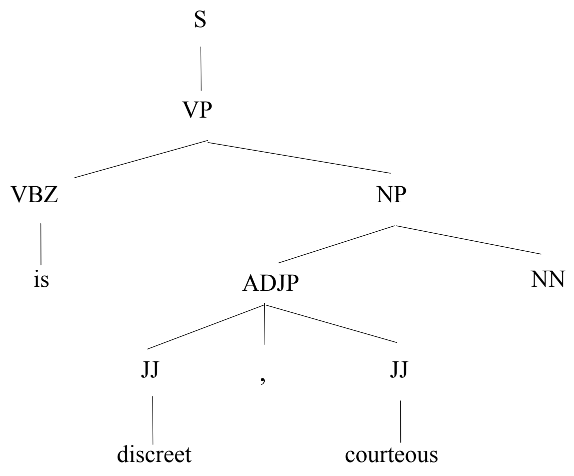


Figure 2 An included tree of the parse tree



One might want to locate, say, those sentences that include a verb phrase containing the verb ‘is’ and after it there are two adjectives ‘discreet’ and ‘courteous’ followed by any noun. They are exactly the sentences whose parse tree can be obtained by deleting some nodes from the tree shown in Figure 1 (see Figure 2 for illustration).

The ordered tree inclusion problem was initially introduced by Knuth (1969). It has been suggested as an important primitive for expressing queries on structured document databases (Mannila and Raiha, 1990). 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 (1995) 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 Richter (1997), 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 $label(v) = label(w)$, 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 $O(T|\cdot|P|)$. The space complexity of the algorithm is $O(|\alpha(P)|\cdot|T| + m(P, T))$. In Chen (1998), a more sophisticated algorithm was presented using $O(|T|\cdot|\text{leaves}(P)|)$ time and $O(|\text{leaves}(P)|\cdot min\{D_T, |\text{leaves}(T)|\} + |T| + |P|)$ space. In Alonso and Schott (1993), an efficient average case algorithm was discussed. 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 $O(|T|\cdot|P|)$. In Bille and Gortz (2005), another bottom-up algorithm is proposed. It is claimed that the algorithm needs only $O(|T| + |P|)$ space. However, a careful analysis reveals that the space complexity of the algorithm is the same as that of Chen (1998). 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 $O(|\text{leaves}(T)|)$ in the worst case. $EMB(v)$ is generated recursively and works in a way similar to the concept of *shell* discussed in Chen (1998). So the analysis of *shell* applies to $EMB(v)$’s.

In our earlier work (Chen and Chen, 2004, 2006a, 2006b), a top-down algorithm was proposed with $O(|T| + |P|)$ space requirement. But its time complexity is not polynomial, as shown in Cheng and Wang (2007).

In this paper, we improve our earlier work and present a new top-down algorithm to remove any redundancy of Chen and Chen (2006a). The time complexity of the new one is bounded by $O(|T|\cdot|\text{leaves}(P)|)$. It is the same as Chen’s algorithm (Chen, 1998). But the space requirement remains $O(|T| + |P|)$.

The tree inclusion problem on unordered trees is *NP*-complete (Kilpelainen and Mannila, 1995) and not discussed in this paper.

2 Basic definitions

We concentrate on labelled trees that are ordered, i.e., the order between siblings is significant. Technically, it is convenient to consider a slight generalisation 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 $<T_1, \ldots, T_k>$, where $k \geq 0$. 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 outdegree of the root of T. A tree with the root t and the subtrees $T_1, \ldots, T_k$ is denoted by $<t; T_1, \ldots, T_k>$. 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 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 $F = <T_1, \ldots, T_k>$ 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 $<T_1, \ldots, T_k>$ 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 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 Knuth (1969, p.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, $v \preceq v'$ if $v \prec v'$ or $v = v'$. Furthermore, we extend this ordering with two special nodes $\perp \prec v \prec \mathrm{T}$. The *left relatives*, lr(v), of a node $v \in V(F)$ are 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 Kilpelainen and Mannila (1995).

Definition 1. Let F and G be labelled 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 $v, u \in V(G)$,

1 $label(v) = label(\varphi(v))$; (label preservation condition)

2 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)

3 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.

Figure 3 shows an example of an ordered inclusion.

Figure 3 Illustration for ordered tree inclusion

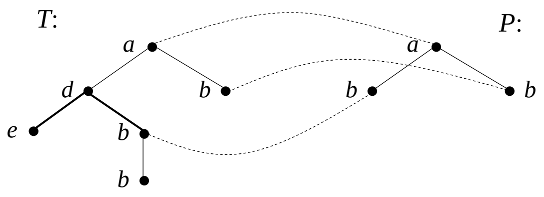
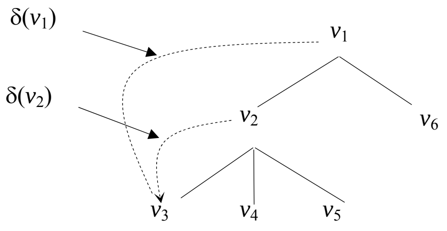


Figure 4 A pattern tree



Let P and T be two labelled 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.

Figure 3 also shows an example of a root preserving embedding. According to Kilpelainen and Mannila (1995), 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.

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

3 Algorithm

Let $G = <P_1, ..., P_l>$ ($l \geq 1$) be a forest. We handle G as a tree $P = <p_v; P_1, \ldots, P_l>$, where p_v represents a virtual node, matching any node in T. Note that even though G contains only one single tree it is considered to be a forest. So a virtual root is added. Therefore, each node in G, except the virtual node, has a parent.

Consider a node v in P with children $v_1, \ldots, v_j$. We use a pair $<i, v>$ ($i \leq j$) to represent an ordered forest containing the first i subtrees of v: $<P[v_1], \ldots, P[v_i]>$. Then, $<j, p_v>$ represents the first j trees in G. In addition, $\delta(v)$ represents a link from v in P to the left-most leaf node in $P[v]$, as illustrated in Figure 4.

Let v' be a leaf node in P. $\delta(v')$ is defined to be v' itself. In addition, we denote by $\delta^{-1}(v')$ a set of nodes $\boldsymbol{x}$ such that for each $v \in x$ $\delta(v) = v'$.

Also, $h(v)$ represents the height of v in a tree, 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.

The following algorithm takes a target $F = <T_1, \ldots, T_k>$ ($k \geq 1$) and a pattern $G = <P_1, \ldots, P_l>$ ($l \geq 1$) as the input; and returns a pair $<i, v>$ as the output, where v is p_v or a node on the left-most path in P_1, showing that F embeds the first i subtrees of v. We denote by t_s the root of T_s ($s = 1, \ldots, k$); and by p_j the root of P_j ($j = 1, \ldots, l$).

In the algorithm, we distinguish between two cases:

Case 1 $k > 1$. That is, F is a forest containing more than one trees.

Case 2 $k = 1$. That is, F is a forest containing only a single tree.

In Case 1, $F = <T_1, \ldots, T_k>$ with $k > 1$. We will make a series of recursive calls to the algorithm itself to check $<T_s>$ against $<P_{j_s}, \ldots, P_l>$, where $s = 1, \ldots, k$, $j_1 = 1$, and $j_1 \leq j_2 \leq \ldots \leq j_x \leq l$ (for some $x \leq k$), controlled as follows.

1 Two index variables s, j are used to scan $T_1, \ldots, T_k$ and $P_1, \ldots, P_l$, respectively. (Initially, s is set to 1, and j is set to 0.) They also indicate that $<P_1, \ldots, P_j>$ has been successfully embedded in $<T_1, \ldots, T_s>$.

2 Let $<i_s, v_s>$ be the return value of checking $<T_s>$ against $<P_{j+1}, \ldots, P_l>$). If $v_s = p_1$'s parent, set j to be $j + i_s$. Otherwise, j is not changed. Set s to be $s + 1$. Go to (2).

3 The loop terminates when all T_s's or all P_j's are examined.

If $j > 0$ when the loop terminates, the algorithm returns $<j, p_1$'s parent$>$.

Otherwise, $j = 0$. In this case, we will continue to search for a pair $<i, v>$ such that F contains the first i subtrees of v, where $v \in \delta^{-1}(v')$ and v' is the left-most leaf node in P_1, as described below.

1 Let $<i_1, v_1>, \ldots, <i_k, v_k>$ be the return values of the recursive calls to check $<T_1>$ against $<P_1, \ldots, P_l>, \ldots, <T_k>$ against $<P_1, \ldots, P_l>$), respectively. Since $j = 0$, each $v_s \in \delta^{-1}(v')$ ($s = 1, \ldots, k$).

2 If each $i_s = 0$, return $<0, \phi,>$, where ϕ is considered to be a descendant of any node in G. Otherwise, there must be some v_s's such that $i_s > 0$. We call such a node a *non-zero point*. Find the first non-zero point v_f (with $i_f > 0$) with children $w_1, \ldots, w_g$ such that v_f is not a descendant of any other non-zero point. Then, make a recursive call to check $<T_{f+1}, \ldots, T_k>$ against $<P[w_{i_f+1}], \ldots, P[w_g]>$). Let $<x, y>$ be its return value. If $y = v_f$, then the return value of the algorithm is set to be $<i_f + x, v_f>$. Otherwise, the return value is $<i_f, v_f>$.

In Case 2, $F = <T>$. We need to handle two sub-cases.

a *Sub-case 1*: $G = <P_1>$; or $G = <P_1, ..., P_l>$ ($l > 1$), but $|T| \leq |P_1| + |P_2|$. In this case, to find a pair $<i, v>$ as described above, we will do the following checkings:

1 Let $T = <t; T_1, \ldots, T_k>$. If t is a leaf node (i.e., T is a single 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, parent of $\delta(p_1)>$.

2. If $|T| < |P_1|$ or $h(t) < h(p_1)$, we will make a recursive call to check $<T>$ against $<P_{11}, \ldots, P_{1j}>$), where $<P_{11}, \ldots, P_{1j}>$ be a forest of the subtrees of p_1. The return value of this recursive call is used as the return value of the checking of $<T>$ against G.
3. If $|T| \geq |P_1|$ and $h(t) \geq h(p_1)$, we further distinguish between two cases:
   - $label(t) = label(p_1)$. In this case, we will make a recursive call to check $<T_1, \ldots, T_k>$ against $<P_{11}, \ldots, P_{1j}>$.
   - $label(t) \neq label(p_1)$. In this case, we will make a recursive call to check $<T_1, \ldots, T_k>$ against $<P_1>$.

In both cases, assume that the return value is $<i, v>$. A further checking needs to be conducted:

- If $label(t) = label(v)$ and i = the outdegree of v, the return value should be <1, v's parent>.
- Otherwise, the return value is the same as $<i, v>$.

b *Sub-case 2*: $G = <P_1, ..., P_l>$ ($l > 1$), and $|T| > |P_1| + |P_2|$. In this case, we will make a recursive call to check $<T_1, \ldots, T_k>$ against G. Assume that the return value is $<i, v>$. The following checkings will be continually conducted.

1. If $v = p_1$'s parent, the return value is the same as $<i, v>$.
2. If $v \neq p_1$'s parent, check whether $label(t) = label(v)$ and i = the outdegree of v. If so, the return value will be changed to <1, v's parent>. Otherwise, the return value remains $<i, v>$.

In terms of the above analysis, we give the formal description of the algorithm as below.

function $td(F, G)$

input: $F = <T_1, \ldots, T_k>$, $G = <P_1, \ldots, P_l>$.

output: $<i, v>$ specified above.

begin

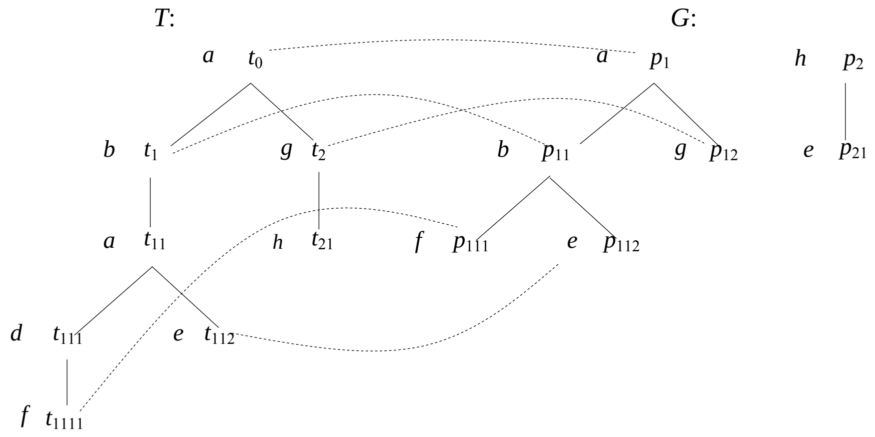
```
1   if $k > 1$ then { (*Case 1*)
2       $s := 1; j := 0$;
3       while ($j < l$ and $s \leq k$) do
4       {       $<i_s, v_s> := td(<T_s>, <P_{j+1}, \ldots, P_l>)$;
5               if ($v_s = p_1$'s parent and $i_s > 0$) then $j := j + i_s$;
6               $l := l + 1$;
7       }
8       if $j > 0$ then return $<j, p_1$'s parent>;
9       if for all $<i_s, v_s>$'s $i_s = 0$ then return $<0, \phi>$
10      else {let $v_f$ be the first non-zero point such that it is not a descendant of
```

```
                   any other non-zero point;
11                 let $w_1, \ldots, w_g$ be the children of $v_f$;
12                 $<i, v> := td(<T_{f+1}, ..., T_k>, <P[w_{i_f+1}], \ldots, P[w_g]>)$;
11         **if** $v = v_f$ **then** return $<i_f + i, v_f>$ **else** return $<i_f, v_f>$; }
12    }
13    **else** { (*$k = 1$ – *Case* 2*)
14         $T := T_1$;
15         **if** ($|T| \leq |P_1| + |P_2|$ or $l = 1$)
16         **then** {let $P_1 = <p_1; P_{11}, \ldots, P_{1j}>$;
17                 **if** $t$ is a leaf **then** {
18                 let $\delta(p_1) = v$;
19                 **if** $label(t) = label(v)$ **then** return <1, $v$'s parent> **else** return <0, $v$'s
                   parent>;
20                 }
21                 **if** ($|T| < |P_1|$ or $h(t) < h(p_1)$)
22                 **then** return $td(<T>, <P_{11}, \ldots, P_{1j}>)$;
23                 **if** $label(t) = label(p_1)$                (*$|T| \geq |P_1|$ and $h(t) \geq h(p_1)$*)
24                 **then** $<i, v> := td(<T_1, \ldots, T_k>, <P_{11}, \ldots, P_{1j}>)$
25                 **else** $<i, v> := td(<T_1, \ldots, T_k>, <P_1>)$;
26                 **if** ($label(t) = label(v)$ and $i$ = $v$'s outdegree )
27                 **then** return <1, $v$'s parent>
28                 **else** return $<i, v>$;
29    }
30         **else** {$<i, v> := td(<T_1, \ldots, T_k>, G)$;
31         **if** $v \neq p_1$'s parent **then**
32         **if** (label($t$) = label($v$)) and $i$ = $v$'s outdegree)
33         **then** return <1, $v$'s parent>;
34         return $<i, v>$;
35         }
36    }
**end**
```

In the above algorithm, Case 1 (when $k > 1$) is handled in lines 1–12 while Case 2 (when $k = 1$) is in lines 13–36.

Example 1. Consider the tree T and the forest G shown in Figure 5. As indicated by the dashed lines, we have an ordered embedding of a subtree of G in T.

Figure 5 A target tree and a pattern forest



In Figure 5, each node in T is identified with t_i, such as t_0, t_1, t_{11}, and so on; and each node in G is identified with p_j. Besides, each subtree rooted at t_i (p_j) is represented by T_i (resp. P_j).

In Figure 6, we trace the computation process when applying the algorithm to <T> and G. In this figure, indentation is used to represent recursive calls. Associated with each recursive call are several conditions, under which the corresponding recursive call is conducted. The conditions are placed before each corresponding recursive call.

The return value of the whole procedure is <1, p_v>, showing that T contains P_1. □

From the sample trace, we can see that a node in T can be checked multiple times, but against different nodes in G. For instance, t_{112} is first checked against p_{111}, and then against p_{112}. t_2 is also checked two times, against p_{111} and p_{12}, respectively.

In order to estimate the number of such checkings, we count how many times t is involved in a recursive call of the form $td(T[t], <G[v_i], \dots, G[v_l]>)$, where $v_i, \dots, v_l$ are the consecutive children of a certain node in G. For simplicity, we denote such a recursive call by $[t, v_i]$.

First, we note that for each $[t, v_i]$, t can be checked at most two times. The first checking is performed in line 23 or in line 32 in $td(F, G)$, and the second checking is in line 26.

Next we pay attention to line 4 and 12 in $td(F, G)$. Assume that $[t, v]$ is the recursive call with t involved for the first time (in line 4). It is possible for t to be involved in a second recursive call $[t, v']$ (see line 12). But v' must be a descendant of v. In addition, v' cannot be a node on the left-most path in $G[v]$. It is because the following conditions must be satisfied to have a second checking (see line 9 and 10):

1 $j = 0$

2 at least there exists a $<i_f, v_f>$ such that $i_f > 0$.

Since $j = 0$, v_f must be a node on the left-most path in $G[v]$. But its $(i_f + 1)th$ child w_{i_f+1} is definitely not on such a path, and v' (if $[t, v']$ is invoked) is a node appearing in the subtree rooted at w_{i_f+1}, or to the right of w_{i_f+1} (see line 12).

Figure 6 A sample trace

Step-by-step trace:	**Explanation**:
$td(<T>, G)$	$td(<T>, G)$ begins.
$\lvert T\rvert > \lvert P_1\rvert + \lvert P_2\rvert$	It is *Case* II. Go to line 13.
$td(<T_1, T_2>, <P_1, P_2>)$	Since $\lvert T\rvert > \lvert P_1\rvert + \lvert P_2\rvert$, go to line 30.
$td(<T_1>, <P_1, P_2>)$	In the **while**-loop, first check $<T_1>$ against $<P_1, P_2>$.
$\lvert T_1\rvert = \lvert P_1\rvert$, $label(t_1) \neq label(p_1)$	Since $\lvert T_1\rvert = \lvert P_1\rvert$, $label(t_1) \neq label(p_1)$, check T_{11}
$td(<T_{11}>, <P_1>)$	against $<P_1>$.
$\lvert T_{11}\rvert < \lvert P_1\rvert$	Since $\lvert T_{11}\rvert < \lvert P_1\rvert$, check $<T_{11}>$ against $< P_{11}, P_{12}>$.
$td(<T_{11}>, <P_{11}, P_{12}>)$	See lines 21 - 22.
$\lvert P_{11}\rvert < \lvert T_{11}\rvert = \lvert P_{11}\rvert + \lvert P_{12}\rvert$, $label(t_{11}) \neq label(p_{11})$	$\lvert T_{11}\rvert = \lvert P_{11}\rvert + \lvert P_{12}\rvert$. lines 16 – 29 will be executed.
$td(<T_{111}, T_{112}> <P_{111}, P_{112}>)$	See line 25.)
$td(<T_{111}>, <P_{111}, P_{112}>)$	It is *Case* I. Lines 3 – 7 will be executed.
$\lvert P_{111}\rvert < \lvert T_{111}\rvert = \lvert P_{111}\rvert + \lvert P_{112}\rvert$, $label(t_{111}) \neq label(p_{111})$	Since $\lvert T_{111}\rvert = \lvert P_{111}\rvert + \lvert P_{112}\rvert$, $label(t_{11}) \neq label(p_{11})$,
$td(<T_{1111}>, <P_{111}>)$	check T_{1111} against $<P_{111}>$ (see line 25).
T_{1111} is a leaf, $label(t_{1111}) = label(p_{111}) = f$	T_{1111} is a leaf. Check it against $\delta(p_{111}) = p_{111}$.
return $<1, p_{11}>$	Return value of $td(<T_{1111}>, < P_{111}, P_{112}>)$. See line 19.
return $<1, p_{11}>$	Return value of $td(<T_{111}>, < P_{111}, P_{112}>)$.
$td(<T_{112}>, <P_{11}, P_{12}>)$	Since T_{111} does not contain any subtree in $< P_{11}, P_{12}>$.
T_{112} is a leaf, $label(t_{112}) \neq label(p_{111})$	T_{112} is a leaf. Check it against $\delta(p_{11}) = p_{111}$.
return $<0, p_{11}>$	Return value of $td(<T_{112}>, < P_{11}, P_{12}>)$ is $<0, p_{11}>$.
$td(<T_{112}>, <P_{112}>)$	It is because $td(<T_{111}>, < P_{11}, P_{12}>)$ returns $<1, p_{11}>$.
T_{112} is a leaf, $label(t_{112}) = label(p_{112})$	T_{112} is a leaf. Check it against $\delta(p_{112}) = p_{112}$.
return $<1, p_{11}>$	Return value of $td(<T_{112}>, <P_{112}>)$.
return $<2, p_{11}>$	Return value of $td(<T_{111}, T_{112}>, \ldots)$. See line 11.
return $<2, p_{11}>$	Return value of $td(<T_{11}>, <P_{11}, P_{12}>)$
return $<2, p_{11}>$	Return value of $td(<T_{11}>, <P_1>)$.
$label(t_1) = label(p_{11})$	Since $td(<T_{11}>, <P_1>)$ returns $<2, p_{11}>$ and $label(t_1)$
return $<1, p_1>$	$= label(p_{11})$, $td(<T_1>, <P_1, P_2>)$ should be $<1, p_1>$.
$td(<T_2>, <P_1, P_2>)$	Since $td(<T_1>, <P_1, P_2>)$ returns $<1, p_1>$, call $td(<T_2>$,
	$<P_1, P_2>)$. See lines 5 – 6.
$\lvert T_2\rvert < \lvert P_1\rvert$	Since $\lvert T_2\rvert < \lvert P_1\rvert$, check $<T_{12}>$ against $< P_{11}, P_{12}>$ by
	calling $td(<T_2>, < P_{11}, P_{12}>)$.
$td(<T_2>, <P_{11}, P_{12}>)$	See lines 21 – 22.
$\lvert T_2\rvert < \lvert P_{11}\rvert$	Since $\lvert T_2\rvert < \lvert P_{11}\rvert$, check $<T_{12}>$ against
	$< P_{111}, P_{112}>$ by $td(<T_2>, < P_{111}, P_{112}>)$.
$td(<T_2>, <P_{111}, P_{112}>)$	See lines 21 – 22.
$\lvert P_{111}\rvert < \lvert T_2\rvert = \lvert P_{111}\rvert + \lvert P_{112}\rvert$, $label(t_2) \neq label(p_{111})$	Since $\lvert T_2\rvert = \lvert P_{111}\rvert + \lvert P_{112}\rvert$, $label(t_2) \neq label(p_{111})$,
	check the subtrees rooted at
$td(<T_{21}>, < P_{111}>)$	t_2's children against $< P_{111}, P_{112}>$ (see line 25).
	t_2 has only one child t_{21}.
T_{21} is a leaf, $label(t_{21}) \neq label(p_{111})$	t_{21} is a leaf. Check it against $\delta(p_{111}) = p_{111}$.
return $<0, p_{11}>$	Since $label(t_{21}) \neq label(p_{111})$, the return value of
	$td(<T_{21}>, < P_{111}>)$ is $<0, p_{11}>$.
return $<0, p_{11}>$	Return value of $td(<T_2>, <P_{111}, P_{112}>)$.
return $<0, p_{11}>$	Return value of $td(<T_2>, <P_{11}, P_{12}>)$.
return $<0, p_{11}>$	Return value of $td(<T_2>, <P_1, P_2>)$.
$td(<T_2>, <P_{12}>)$	$< T_1, T_2>$ does not contain P_1. But T_1 contains
	P_{11}. So call $td(<T_2>, <P_{12}>)$.
$label(t_2) = label(p_{12})$, p_{21} is a leaf.	$label(t_2) = label(p_{12})$ and p_{21} is a leaf.
return $<1, p_1>$	$td(<T_2>, <P_{12}>)$ returns $<1, p_1>$.
return $<2, p_1>$	$td(<T_1, T_2>, <P_1, P_2>)$ returns $<2, p_1>$ because
	$td(<T_1>, <P_1, P_2>)$ returns $<1, p_1>$.
$label(t_0) = label(p_1)$, p_1's outdegree = 2.	In line 26, compare the labels of and t_0 and
	p_1; and check p_1's outdegree.
return $<1, p_v>$	Since $label(t_0) = label(p_1)$ and p_1's outdegree = 2,
	T contains P_1.

C_2. Then,

1 if $t_1 = t_2$, p_2 is a child of p_1

2 if $p_1 = p_2$, t_2 is a child of t_1.

Proof. We first prove (1). If $t_1 = t_2$, it shows that $T[t_1]$ is involved in a second call of the second kind, but checked against a forest containing the subtrees respectively rooted at the children of p_1 (see line 21 in *td*(F, G).) Therefore, p_2 is a child of p_1.

Now we consider (2). If $p_1 = p_2$, it shows that $G[p_1]$ is involved in a second call of the second kind, which happens when the size of $T[t_1]$ is larger that the size of $G[p_1]$ plus the size of the subtree rooted at p_1's direct right sibling. This leads to a first kind of call to check the forest containing the subtrees respectively rooted at the children of t_1 against the subtrees respectively rooted at p_1 and its right siblings [see line 30 in *td*(F, G)]. In the execution of the first kind of call, p_1 will be checked for a second time, but against a child of t_1. □

In terms of Proposition 2, we can see that $|C_2|$ is bounded by $O(D_F + D_G)$. In a similar way, we can show that $|C_1|$ is also bounded by $O(D_F + D_G)$. Therefore, $|C| = |C_1| + |C_2|$ is in the order of $O(D_F + D_G)$.

Proposition 3. The space complexity of the algorithm is bounded by $O(|F| + |G|)$.

Proof. See the above analysis. □

4 Correctness

In this section, we prove the correctness of our algorithm.

Proposition 4. Let $T = <t; T_1, \ldots, T_k>$ and $G = <P_1, \ldots, P_l>$. If Algorithm *td*($<T>$, G) returns $<i, v>$, where $v = p_v$, or $v \in \delta^{-1}(v')$ and v' is the left-most leaf node in P_1, then T embeds the first i subtrees of v, and there is not any ancestor v'' of v such that T embedding $<j, v''>$ with $j > 0$.

Proof. We prove the proposition by induction on the sum of the heights of T and G, $H = h(T) + h(G)$. $h(T)$ is defined to be the height of its root, and $h(G)$ is the height of its highest subtrees.

Basic step. When $H = 0$, T is a singular t, and G is a set of nodes: $p_1, \ldots, p_k$. In this case, the algorithm returns $<0, p_v>$ or $<1, p_v>$, depending on whether $label(t) = label(p_1)$. See lines 17–19 in *td*(F, G).

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

1 T is a tree of height 1: $<t; t_1, \ldots, t_k>$ and G is a set of nodes: $<p_1, \ldots, p_l>$.

2 T is a singular t; but G is a set of trees of height 1 or height 0, but at least one of them is of height 1.

In Case (1), lines 23 - 29 will be executed if $|T| \leq 2$. If $label(t) = label(p_1)$, return $<1, p_v>$. Otherwise, we will call $td(<t_1>, <p_1>)$. If $label(t_1) = label(p_1)$, it returns $<1, p_v>$; Otherwise, it returns $<0, p_v>$. If $|T| > 2$, $td(<t_1, \ldots, t_k>, <p_1, \ldots, p_l>)$ is called (see line 30), which will find a sequence of integers: $k_1, \ldots, k_f$ such that $label(t_{k_i}) = label(p_i)$. The return value is $<f, p_v>$ $(0 \leq f \leq l)$.

In Case (2), the return value is <0, p_1> or <1, p_1>, depending on whether t matches the first child of p_1. See lines 17–19 in *td*(F, G).

Induction hypothesis. Assume that when $H = q$, the proposition holds.

Consider T = <t; T_1, …, T_k> and G = <P_1, …, P_l> with $h(T) + h(G) = q + 1$.

If $l = 1$, or $l > 1$ but $|T| \leq |P_1| + |P_2|$, the following checkings will be performed.

If $|T| < |P_1|$ or $h(t) < h(p_1)$, we will make a recursive call *td*(<T >, <P_{11}, …, P_{1j}>), where <P_{11}, …, P_{1j}> is a forest of the subtrees of p_1. The return value of *td*(<T>, <P_{11}, …, P_{1j}>) is used as the return value of *td*(<T>, G). According to the induction hypothesis, the return value is correct. Otherwise, we further distinguish between two cases:

- *label*(t) = *label*(p_1). In this case, we will call *td*(<T_1, …, T_k>, <P_{11}, …, P_{1j}>).
- *label*(t) ≠ *label*(p_1). In this case, we will call *td*(<T_1, …, T_k>, <P_1>).

In both cases, assume that the return value is <i, v>. A further checking needs to be conducted:

If *label*(t) = *label*(v) and i = the outdegree of v, the return value should be <1, v's parent>.

Otherwise, the return value is the same as <i, v>.

In the execution of *td*(<T_1, …, T_k>, <P_{11}, …, P_{1j}>), a series of recursive calls of the form: *td*(<T_s>, <P_{1a}, …, P_{1b}>) will be performed. According to the induction hypothesis, each of them returns a correct value.

Assume that the return value of each *td*(<T_s>, <P_{1a}, ..., P_{1b}>) is <i_s, v_s>. If <T_1, ..., T_k> is not able to cover any subtree from P_{11}, ..., P_{1j}, then each v_s cannot be the parent of p_{1a}, but a node on the left-most path in P_{1a}. In this case, an extra recursive call of the form: *td*(<T_{f+1}, …, T_k>, <$P[w_{i_f+1}]$, ..., $P[w_g]$>) will be conducted, where v_f (with $i_f > 0$) is the first non-zero point such that it is not a descendant of any other non-zero point, and w_1, …, w_g are all its children. Repeat the above analysis, we can show the correctness of *td*(<T_1, …, T_k>, <P_{11}, …, P_{1j}>).

The above discussion demonstrates that the return value of *td*(<T_1, …, T_k>, <P_{11}, …, P_{1j}>) is correct. In the same way, we can also show the return value of *td*(<T_1, …, T_k>, <P_1>) is correct.

If $l > 1$ and $|T| > |P_1| + |P_2|$. In this case, we will call *td*(<T_1, …, T_k>, G). Assume that the return value is <i, v>. The following checkings will be conducted.

- If $v = p_1$'s parent, the return value is the same as <i, v>.
- If $v \neq p_1$'s parent, check whether *label*(t) = *label*(v)) and i = the out degree of v. If so, the return value will be changed to <1, v's parent>. Otherwise, the return value remains <i, v>.

According the induction hypothesis, in this case, the return value is correct. It completes the proof. □

5 Experiments

We have compared our algorithm with the algorithm by Chen (1998) experimentally. We conducted our experiments on a DELL desktop PC equipped with Pentium III 1.6 GHz processor, 1.00 GB RAM and 20GB hard disk. The code was compiled using Microsoft Visual C++ compiler version 6.0, running standalone.

In our experiments, two data sets are used as the target trees. One is part of *DBLP*, which is a popular computer science bibliography in XML format (http://www.cs.washington.edu/research/xmldatasets). The other is *TreeBank*, which is a database containing a set of linguistic text structures (http://www.cs.washington.edu/research/xmldatasets). The important parameters of these data sets are summarised in Table 1.

Table 1 Data set for experimental evaluation

	Part of DBLP	*TreeBank*
Data size	2 (MB)	82 (MB)
Number of nodes	53k	2,437k
Max/Avg. depth	6/2.9	36/7.9

The pattern trees (P_1 and P_2) used for testing the first data set is shown in Figure 7. This test is to check the impact of the number of leaf nodes in a tree pattern. The test results of P_1 and P_2 are shown in Table 2 and 3, respectively.

Figure 7 Pattern trees checked against DBLP

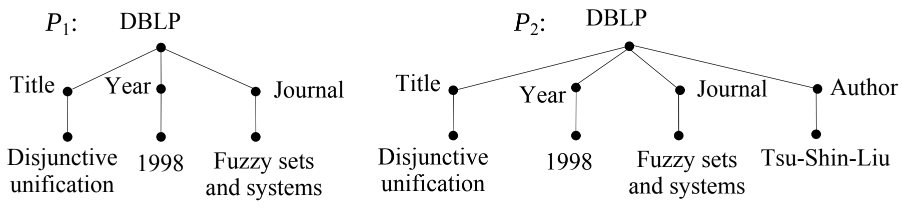


Table 2 Test result of checking P_1 against DBLP

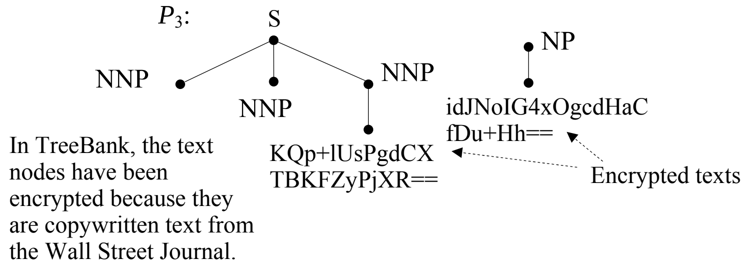
	Time	*Number of node comparison*
Top-down	0.54s	178,287
Bottom-up	0.56s	179,345

Table 3 Test result of checking P_2 against DBLP

	Time	*Number of node comparison*
Top-down	0.647	191,252
Bottom-up	1.73s	263,912

From Table 2 and 3, we can see that although our *top-down* algorithm has the same worst-case time complexity as Chen's *bottom-up* method (Chen, 1998), our algorithm makes much less node comparisons than Chen's. This confirms the theoretical analysis conducted in Section 3.

Figure 8 Pattern trees checked against TreeBank


P_3:
S
NNP
NNP
NNP
NP
KQp+lUsPgdCX TBKFZyPjXR==
idJNoIG4xOgcdHaC fDu+Hh==
Encrypted texts
In TreeBank, the text nodes have been encrypted because they are copywritten text from the Wall Street Journal.

The pattern tree (P_3) for testing the second data set is shown in Figure 8.

It is a forest containing two trees. In this test, a huge target tree is used to observe the difference caused by the space overheads of these two methods.

Table 4 Test result of checking P_3 against TreeBank

	Time	*Number of node comparison*
Top-down	54.62s	7,038,874
Bottom-up	372.32s	49,449,170

From Table 4, we can see that for a large target tree the difference between ours and Chen's is significant. It is not only due to the fewer comparisons conducted by our method, but also to the much less space used by ours. Chen's method needs to create and maintain a set of data structures (called *interval* in Chen, 1998) for each node of the target tree, which requires extra time.

6 Conclusions

In this paper, a new algorithm is proposed to improve the algorithm discussed in Chen and Chen (2006a). The main idea behind it is to let any subprocedure call return a pair to indicate a subtree (subforest) embedding while in Chen and Chen (2006a), 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 kind of *information transferring* to transfer the results 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 $O(|T|\cdot|\text{leaves}(P)|)$ while the space requirement is bounded by $O(|T| + |P|)$, where T and P are a target and a pattern tree, respectively.

References

Alonso, L. and Schott, R. (1993) 'On the tree inclusion problem', *Proceedings of Mathematical Foundations of Computer Science*, pp.211–221.

Bille, P. and Gortz, I.L. (2005) 'An ordered tree inclusion algorithm based on dynamic tree labeling', *Proc. 32nd Intl. Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*, Vol. 3580, pp.66–77.

Chen, W. (1998) 'More efficient algorithm for ordered tree inclusion', *Journal of Algorithms*, Vol. 26, pp.370–385.

Chen, Y. and Chen, Y.B. (2004) 'An efficient top-down algorithm for tree inclusion', *Proc. of 18th Intl. Conf. Symposium on High Performance Computing System and Application*, May, pp.183–187, IEEE, Winnipeg, Canada.

Chen, Y. and Chen, Y. (2006) 'On the top-down tree inclusion', *Proc: Intl. Conf. on Advances in Computer Science and Technology (ACST 2006)*, January, Vol. 25, pp.61–66 Puerto Vallarta, Mexico.

Chen, Y. and Chen, Y.B. (2006) 'A new tree inclusion algorithm', *Information Processing Letters*, Vol. 98, pp.253–262, Elsevier Science B.V.

Cheng, H.L. and Wang, B.F. (2007) 'On Chen and Chen's new tree inclusion algorithm', *Information Processing Letters*, Vol. 103, pp.14–18, Elsevier Science B.V.

Kilpelainen, P. and Mannila, H. (1995) 'Ordered and unordered tree inclusion', *SIAM Journal of Computing*, Vol. 24, pp.340–356.

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

Mannila, H. and Raiha, K-J. (1990) 'On query languages for the p-string data model', in Kangassalo, H., Ohsuga, S. and Jaakola, H. (Eds.): *Information Modelling and Knowledge Bases*, pp.469–482, IOS Press, Amsterdam.

Richter, T. (1997) 'A new algorithm for the ordered tree inclusion problem', *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), Lecture Notes of Computer Science (LNCS)*, Vol. 1264, pp.150–166, Springer.

U of Washington XML Repository (2007) available at http://www.cs.washington.edu/research/xmldatasets.