An Efficient Top-down Algorithm for Tree Inclusion

Yangjun Chen and Yibin Chen
Dept. of Applied Computer Science
University of Winnipeg
Winnipeg, Canada R3B 2E9
Email: ychen2@uwinnipeg.ca

Abstract —**Let *T* and *S* be ordered, labeled trees. We say that *S* is included in *T* if there is a sequence of delete operations performed on *T*, which make *T* isomorphic to *S*. In this paper, we propose a top-down algorithm for this problem, which needs only O(|*T*|·|leaves(*S*)|) time and O(|*T*| + |*S*|) space.**

I. Introduction

Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. Given two ordered labeled trees *T* and *S*, the *tree inclusion problem* is to determine whether it is possible to obtain *S* from *T* by deleting nodes. Deleting a node *v* in tree *T* means making the children of *v* become the children of the parent of *v* and then removing *v*. If *S* can be obtained from *T* by deleting nodes, we say that *T* includes *S*.

The ordered tree inclusion problem was initially introduced by Knuth [7], 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 [5]. 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 [6] presented the first polynomial time algorithm using O(/*T*|·|*S*|) time and space. Most of the later improvements are refinements of this algorithm. In [8], Richter gave an algorithm using O(|α(*S*)|·|*T*| + *m*(*S*, *T*)·D_T) time, where α(*S*) is the alphabet of the labels of *S*, *m*(*S*, *T*) is the size of a set called *matches*, defined as all the pairs (*v*, *w*) ∈ *S* × *T* such that label(*v*) = 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*|·|*S*|). The space complexity of the algorithm is O(|α(*S*)|·|*T*| + *m*(*S*, *T*)). In [2], a more complex algorithm was presented using O(|*T*|·|leaves(*S*)|) time and O(|leaves(*S*)|·*min*{D_T, |leaves(*T*)|}) space. In [1], an efficient average case algorithm was discussed. Its average time complexity is O(|*T*| + *C*(*S*, *T*)·|*S*|), where *C*(*S*, *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*|·|*S*|).

All the above algorithms work in a bottom-up way. In this paper, we propose a top-down algorithm for this problem, which needs only O(|*T*|·|leaves(*S*)|) time. The space complexity is bounded by O(|*T*| + |*S*|. The top-down algorithm does not only have better computational complexities than the bottom-up strategies, but can also be combined with some kinds of heuristics such as *signatures* [4] to speed-up query evaluation [3].

II. Orderings and Embeddings

Let *T* be an ordered, rooted tree with root *v* and children v_1, v_2, ..., v_i. The *postorder* traversal of *T*(*v*) (the tree rooted at *v*) is obtained by visiting T(v_k), $1 \le k \le i$ in order, recursively, and then visiting the *v*. The *postorder number*, post(*v*), of a node *v* ∈ *V*(*T*) is the number of nodes preceding *v* in the postorder traversal of *T*. We define an ordering of the nodes of *T* 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 $\perp \prec v \prec \top$. The *left relatives*, lr(*v*), of a node *v* ∈ *V*(*T*) 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*.

Definition 1. Let *S* and *T* be rooted labeled trees. We defined an ordered embedding (*f*, *S*, *T*) as an injective function *f*: *V*(*S*) → *V*(*T*) such that for all nodes *v*, *u* ∈ *V*(*S*),

i) label(*v*) = label(*f*(*v*)); (label preservation condition)

ii) *v* is an ancestor of *u* iff *f*(*v*) is an ancestor of *f*(*u*); (ancestor condition)

iii) *v* is to the left of *u* iff *f*(*v*) is to the left of *f*(*u*). (Sibling condition) □

Fig. 1 shows an example of an ordered inclusion.

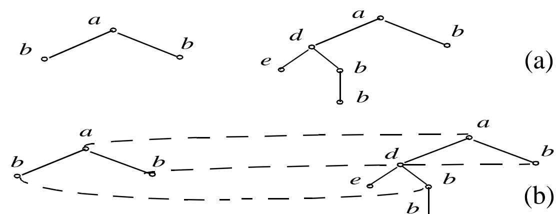

Fig. 1: Illustration of tree inclusion

In Fig. 1(a), we show that the tree on the left can be included in the tree on the right by deleting the nodes labeled: *d*, *e* and *b*. Fig. 1(b) shows a possible embedding.

An embedding is *root preserving* if *f*(root(*S*)) = root(*T*). Fig. 1(b) shows an example of a root preserving embedding.

III. ALGORITHM DESCRIPTION

Our algorithm is based on the following three observations:

(1) Let r_1 and r_2 be the roots of T and S, respectively. If T includes S and label(r_1) = label(r_2), we must have a root preserving embedding.

(2) Let T_1, ..., and T_k be the subtrees of r_1. Let S_1, ..., and S_l be the subtrees of r_2. If T includes S and label(r_1) = label(r_2), There must exist $k_1, ..., k_j$ and $l_1, ..., l_j$ ($j \le l$) such that T_{k_i} includes $< S_{l_{i-1}+1}, ..., S_{l_i} >$ ($i = 1, ..., j$), where $< S_{l_{i-1}+1}, ..., S_{l_i} >$ represents a forest containing subtrees $S_{l_{i-1}+1}$, ..., and S_{l_i}.

(3) If T includes S, but label(r_1) $\neq$ label(r_2), there must exist an i such that T_i contains the whole S.

In terms of the above observation, we devise a computation process as below. First of all, in the case of label(r_1) = label(r_2), we will check whether T_1 includes $<S_1, ..., S_l>$. The process returns an integer i, indicating that T_1 includes $<S_1, ..., S_i>$. If $i > 0$, then we will check whether T_2 includes $<S_{i+1}, ..., S_l>$ in a next step. If $i = 0$, it shows that no subtrees of T_1's root includes any subtrees in $<S_1, ..., S_l>$. In this case, we need to check whether T_1 includes S_1. It is because although no subtrees of T_1's root includes any subtrees in $<S_1, ..., S_l>$, T_1 may include S_1. If T_1 includes S_1, i will be changed to 1; otherwise, it remains 0. However, if the root of T_1 does not match the root of S_1, we know that T_1 cannot include S_1 since in this case we will have to check the subtrees of T_1's root against S_1; and we have already done that with the result $i = 0$. We repeat this process until we find a k_j such that T_{k_j} contains all the remaining subtrees of r_2, or find that such a k_j does not exist.

In the following algorithm *tree-inclusion*(T, S), T is a tree and S is a tree or a forest. If S is a forest, a virtual root for it is constructed, which matches any label. Thus, we will actually check the subtrees of T's root against the subtrees in S, respectively.

Function *tree-inclusion*(T, S)

Input: T, S

Output: 1, if T includes S; otherwise, 0.

begin

1. **if** $|T| < |S|$ **then** {**if** S is a forest: $<S_1, ..., S_l>$
2. **then** $S := <S_1, ..., S_i>$ for some i such that $|<S_1, ..., S_i>| \le |T| < |<S_1, ..., S_{i+1}>|$
3. **else** return 0;}
4. let r_1 and r_2 be the roots of T and S, respectively;
5. (*If S is a forest, construct a virtual root for it, which matches any label.*)
6. let $T_1, ..., T_k$ be the subtrees of r_1;
7. let $S_1, ..., S_l$ be the subtrees of r_2;
8. **if** *label*(r_1) = *label*(r_2)
9. **then** {**if** r_1 is a leaf **then** {**if** r_2 is not a virtual root **then** return 1 **else** return 0;}
10. $temp := <S_1, ..., S_l>$;
11. $S_0 := \phi$;
12. $i := 1; j := 0; x := 0;$
13. **while** ($i \le k \wedge temp \neq \phi$) **do**
14. { $x :=$ *tree-inclusion*(T_i, *temp*);
15. **if** $x > 0$ **then** $temp := temp/<S_{j+1}, ..., S_{j+x}>$;
16. **else** {**if** T_i and S_{j+1} have the same label **then** {$x :=$ *tree-inclusion*(T_i, S_{j+1}); $temp := temp/<S_{j+x}>$;}}
17. $i := i + 1; j := j + x;$}
18. **if** $temp \neq \phi$ **then** {**if** r_2 is a virtual root **then** return j **else** return 0;}
19.
20. **else** {**if** r_2 is a virtual root **then** return l
21. **else** return 1;}}
22. **else** {**for** i = 1 to k **do**
23. {$x :=$ *tree-inclusion*(T_i, S);
24. **if** x = *number-of-forest*(S) return x;
(* *number-of-forest*(S) returns the number of the forests in S.*)
25. return 0;}

end

In Algorithm *tree-inclusion*(T, S), line 1 checks whether $|T| < |S|$. If it is the case, the algorithm returns 0 if S is a tree. If S is a forest, we will check T against the first i subtrees such that $|<S_1, ..., S_i>| \le |T| < |<S_1, ..., S_{i+1}>|$ (see line 2). In addition, when we check T against a forest $<S_1, ..., S_l>$, a virtual root for it is constructed, which matches any label. Thus, we will actually check the subtrees of T's root: $T_1, ..., T_k$ against S_1, ..., and S_l to see whether they include $<S_1, ..., S_l>$ (see line 5). This is performed in a while-loop over T_i's. In each step, a recursive call: *tree-inclusion*(T_i, $< S_{l_i}, ..., S_l>$) ($i = 1, ..., j$ for some j) is carried out, which returns an integer x, indicating that T_i includes $< S_{l_i}, ..., S_{l_i+x-1} >$ (see line 14). If $x = 0$, i.e., the subtrees of T_i's root do not include any subtree in $S_{l_i}, ..., S_l$, we need to check whether T_i include S_{l_i} since when we check T_i against $S_{l_i}, ..., S_l$, what we have done is to check the subtrees of check T_i's root, not T_i itself (see line 16). If S is a tree, the algorithm return 1 if it is included; otherwise, 0 (see line 19 and 21). Finally, we note that if the root of T does not match the root of S, the algorithm tries to find the first T_i that contains the whole S (see lines 22 - 25).

In the following, we apply the algorithm to the trees shown in Fig. 3. and trace the computation step-by step for a better understanding.

Example 1. Consider two ordered, labeled trees T and S shown in Fig. 2, 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 S is identified with s_j. In addition, each subtree rooted at t_i (s_j) is represented by T_i (S_j).

In the following step-by-step trace, i_k is used as an index variable for scanning the subtrees of T_k's root; j_k is used to scan the corresponding subtrees in S; and x_k is used as a temporary variable.

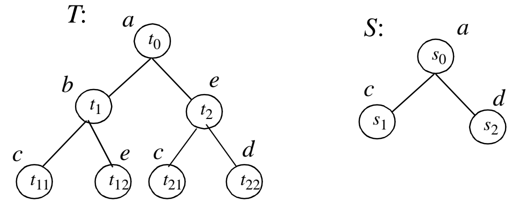

Fig. 2: Two trees

IV. CORRECTNESS

In this section, we show the correctness of Algorithm *tree-inclusion*(T, S).

Proposition 1. If S is a tree, Algorithm *tree-inclusion*(T, S) returns 1 if T includes S; otherwise 0. If S is a forest of the form:

Step-by-step trace:	*Explanation*:
tree-inclusion(T, S)	Call *tree-inclusion*(T, S)
label(t_0) = label(s_0)	Check t_0 against s_0.
$i_0 := 1; j_0 := 0; x_0 := 0$	i_0 is for scanning the subtrees of t_0; j_0 is used to record how many subtrees of s_0 is included; x_0 is a temporary variable.
tree-inclusion(T_1, $<S_1, S_2>$)	recursive call *tree-inclusion*(T_1, $<S_1, S_2>$).
label(t_1) = label(*virtual-root*)	Check t_1 against a virtual root. It always succeeds.
$i_1 := 1; j_1 := 0; x_1 := 0$	i_1 is for scanning the subtrees of t_1; j_1 is used to record how many subtrees of s_0 is included; x_1 is a temporary variable.
tree-inclusion(T_{11}, $<S_1, S_2>$)	recursive call *tree-inclusion*(T_{11}, $<S_1, S_2>$).
$/T_{11}/ < /<S_1, S_2>/$	compare the sizes of T_{11} and $<S_1, S_2>$.
remove S_2 from $<S_1, S_2>$	since $<S_1, S_2>$ is larger than T_{11}, remove S_2 from $<S_1, S_2>$.
label(t_{11}) = label(s_1)	Check t_{11} against s_1.
return 1	it returns 1, indicating that T_{11} includes S_1.
$x_1 = 1; j_1 = 1; i_1 = 2$	i_1 is increased by 1; x_1 is equal to 1 and then j_1 is increased by 1.
tree-inclusion(T_{12}, S_2)	recursive call *tree-inclusion*(T_{12}, S_2).
label(t_{12}) ≠ label(s_2)	Check t_{12} against s_2.
return 0	it returns 0, indicating that T_{12} does not include S_2.
$x_1 = 0; j_1 = 1; i_1 = 3$	i_1 is increased by 1; x_1 is equal to 0 and then j_1 is not increased.
return 1	it returns 1, indicating that T_1 includes S_1 of $<S_1, S_2>$.
$x_0 = 1; j_0 = 1; i_0 = 2$	i_0 is increased by 1; x_0 is equal to 1 and then j_0 is increased by 1.
tree-inclusion(T_2, S_2)	recursive call *tree-inclusion*(T_2, S_2).
label(t_2) ≠ label(s_2)	Check t_2 against s_2. Since they do not match, all the subtrees of t_2 will be checked one by one.
$i_2 := 1; j_2 := 0; x_2 := 0$	i_{12} is for scanning the subtrees of t_2; j_1 is used to record how many subtrees of s_0 is included; x_2 is a temporary variable.
tree-inclusion(T_{21}, S_2)	recursive call *tree-inclusion*(T_{21}, S_2).
label(t_{21}) ≠ label(s_2)	Check t_{21} against s_2.
return 0	it returns 0, indicating that T_{21} does not include S_2.
$x_2 = 0; j_2 = 0; i_2 = 2$	i_2 is increased by 1; x_2 is equal to 0 and then j_2 is not increased.
tree-inclusion(T_{22}, S_2)	recursive call *tree-inclusion*(T_{22}, S_2).
label(t_{22}) = label(s_2)	Check t_{22} against s_2.
return 1	it returns 1, indicating that T_{22} includes S_2.
$x_2 = 1; j_2 = 1; i_2 = 3$	i_2 is increased by 1; x_2 is equal to 1 and then j_2 is increased by 1.
return 1	it returns 1, indicating that T_2 includes S_2 of $<S_1, S_2>$.
$x_0 = 1; j_0 = 2; i_0 = 3$	i_0 is increased by 1; x_0 is equal to 1 and then j_0 is not increased by 1.
return 2	since $j_0 = 2$, *tree-inclusion*(T_2, S_2) returns 2.
return 1	it returns 1, indicating that T includes S. □

$<S_1, ..., S_l>$, Algorithm *tree-inclusion*(T, S) returns an integer i, indicating that T includes $<S_1, ..., S_i>$.

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

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

(i) Both T and S are singulars: r_1 and r_2.

(ii) T is a singular; but S is a set of nodes.

In case (i), if r_1 and r_2 have the same label, the algorithm returns 1 (see line 9); otherwise returns 0 (see line 25). In case (ii), a virtual root will be constructed for S, which matches any label. Then, we will check the subtrees of r_1 against all the nodes in S. Since r_1 does not have any subtrees, we will get 0 as the return value (see line 9). Then, r_1 will be checked against the first node in S and return 1 if they have the same label; otherwise, return 0 (see lines 13 -17).

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

Consider two trees T and S with height(T) + height(S) = $l + 1$. Assume that S is a tree. Let r_1 and r_2 be the roots of T and S, respectively. Let T_1, ..., and T_k be the subtrees of r_1. Let S_1, ..., and S_l be the subtrees of r_2. Then, height(T_i) + height(S) ≤ l and height(T) + height(S_j) ≤ l. If label(r_1) = label(r_2), the algorithm partitions the integer sequence: 1, ..., l into some subsequences: $\{j_0 + 1, ..., j_1\}$, $\{j_1 + 1, ..., j_2\}$, ..., $\{j_{m-1} + 1, ..., j_m\}$, where $j_0 = 0$ and $j_m \le l$, such that each T_i ($i = 1, ..., m$; $m \le k$) includes $< S_{j_{i-1}+1}, ..., S_{j_i} >$ but not $< S_{j_{i-1}+1}, ..., S_{j_i}, S_{j_i+1} >$ (see lines 13 - 17). In terms of the induction hypothesis, the partition is correct. Thus, the algorithm will return 1 if $j_m = l$, indicating that T includes S (see line 21); otherwise 0 (see line 19). If label(r_1) ≠ label(r_2), algorithm will try to find the first T_i such that it includes the whole S. In terms of the induction hypothesis, the return value must be correct.

Assume that S is a forest of the form: $<S_1, ..., S_l>$, a virtual root will be constructed for it. In terms of the induction hypothesis, the algorithm will find the correct integer i such that T includes $<S_1, ..., S_i>$ (see line 18 and 20). It completes the proof. □

V. COMPUTATIONAL COMPLEXITIES

In this section, we discuss the computational complexity of the top-down algorithm discussed in Section 3.

Let T_0 and S_0 be two ordered, labeled trees. Let T_1, ..., and T_k be the subtrees of T_0's root. Let S_1, ..., and S_l be the subtrees of S_0's root.

Denote $C_{i,j}$ the number of label comparisons for checking whether T_i contains S_j. Denote $C_{i, l_i...l_j}$ the number of label comparisons for checking whether T_i contains $< S_{l_i}, ..., S_{l_j} >$. Then, according to the top-down algorithm, we have

$$C_{0,0} \le 1 + p(\sum_{i=1}^{k}(C_{i, l_i...l} + pC_{i, l_i})) + q(\sum_{i=1}^{k} C_{i,0}) \qquad (1)$$

where p represents the probability that a node in T matches a node in S, and $q = 1 - p$. Especially, we notice that

$$\sum_{i=1}^{k}(C_{i, l_i...l} + pC_{i, l_i}) \qquad (2)$$

is the cost of executing lines 13 - 17, in which $C_{i, l_i...l}$ stands for the time spent on checking T_i against $< S_{l_i}, ..., S_l>$ (see line 14) and C_{i, l_i} for the time on checking T_i against S_{l_i} (see line 16). In addition, $C_{0,0}$ stands for the cost of checking T_0 against S_0; and $C_{i,0}$ for the cost of checking a T_i against S_0 (see lines 22 - 24).

Now we consider $C_{i, l_i...l}$. Let T_{i1}, ..., T_{ij_i} be the subtrees of T_i's root. In the worst case, the algorithm checks T_{i1}, ..., and T_{ij_i} in turn against S_{l_i} without success, which leads to an extra computation, *i.e.*, the checking of T_i against S_{l_i}. If there is any T_{ik} $(1 \le k \le j_i)$ that includes S_1, the corresponding computation will not be done and then, pC_{i, l_i} should be removed from (1). In terms of this analysis, we change (1) to the following form:

$$C_{0,0} \le 1 + p(\sum_{i}^{k}\left(\sum_{j=1}^{j_i}(C_{ij, l_i}) + pC_{i, l_i}\right)) + q(\sum_{i=1}^{k} C_{i,0}) \qquad (3)$$

In terms of (3), we have the following proposition.

Proposition 2. Let T_0 and S_0 be two ordered labeled trees. Let T_1, ..., and T_k be the subtrees of T_0's root. Let S_1, ..., and S_l be the subtrees of S_0's root. If $|\text{leaves}(S_j)| \le |\text{leaves}(S_0)|/(1 + p)$ for every j, $C_{0,0} \le |T_0| \cdot |\text{leaves}(S_0)|$.

Proof. We prove the proposition by induction on the sum of the heights of T_0 and S_0, h. Without loss of generality, assume that height(T_0) ≥ 1 and height(S_0) ≥ 1.

Basic step. When $h = 2$, the proposition obviously holds.

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

Consider two trees T_0 and S_0 with height(T_0) + height(S_0) = l + 1. Then, height(T_i) + height(S_0) ≤ l and height(T_0) + height(S_j) ≤ l. In terms of (3) and the induction hypothesis, we have

$$C_{0,0} \le 1 + p(\sum_{i=1}^{k}\sum_{j=1}^{j}|T_{ij}| \cdot |\text{leaves}(S_{j_i})| + p\sum_{i=1}^{k}|T_i| \cdot |\text{leaves}(S_{j_i})|) + q(\sum_{i=1}^{k}|T_i| \cdot |\text{leaves}(S_0)|) \qquad (4)$$

$$\le 1 + \frac{p}{1+p}((1+p)|T_0| - (k + 1 + p))|\text{leaves}(S_0)| + q(|T_0| - 1)| \cdot |\text{leaves}(S_0)|$$

$$\le |T_0| \cdot |\text{leaves}(S_0)|.$$ □

In addition, the algorithm needs no extra space. Thus, the space complexity is on O(|T| + |S|).

VI. FURTHER IMPROVEMENT

In this section, we elaborate the algorithm discussed above. What we want is to replace the recursive call *tree-inclusion*(T_i, S_{j+1}) in line 16 with a simple checking. For this purpose, we rearrange the algorithm so that each recursive call *tree-inclusion*(T_i, $<S_1, ..., S_l>$) returns a pair <*num*, *subnum*> instead of a single number, where *num* is 0 or an integer k to indiacte S_1, ..., S_k are included in T_i, and *subnum* is defined as follows.

Let Z_1, ..., Z_m be the subtrees of S_1's root. If *num* = 0, *subnum* = j >= 0, indicating that Z_1, ..., Z_j are included in T_1, ..., T_i. Otherwise, *subnum* is set to 0 (and will not be used in the subsequent computation).

In addition, the modified algorithm takes three arguments: T_i, $<S_1, ..., S_l>$ and a, where a is an integer indicating that Z_1, ..., Z_a are included in T_1, ..., T_{i-1}.

Function *modified-inclusion*(T, S, a)

Input: T, S

Output: 1, if T includs S; otherwise, 0.

```
begin
1.   if |T| < |S| then {if S is a forest: <S1, ..., Sl>
2.                then if there exists i such that |<S1, ..., Si>| ≤ |T| <
                         |<S1, ..., Si+1>|
3.                   then S := <S1, ..., Si>;
4.                   else {let Z1, ..., Zm be the subtrees of S1's root;
5.                      if there exists j such that
                        |<Za+1, ..., Zj>| ≤ |T| < |<Za+1, ..., Zj+1>|
6.                      then {S := <Za+1, ..., Zj>; tag = 1;}
7.                      else return <0, a>;}
8.   let r1 and r2 be the roots of T and S, respectively;
9.   (*If S is a forest, construct a virtual root for it, which matches any
       label.*)
10.  let T1, ..., Tk be the subtrees of r1;
11.  let S1, ..., Sl be the subtrees of r2;
12.  if label(r1) = label(r2)
13.  then {if r1 is a leaf then {if r2 is not a virtual root
                        then return 1else return <0, a>;}
14.        temp := <S1, ..., Sl>;
15.        S0 := φ;
16.        i := 1; j := 0; x := 0;
17.        while (i ≤ k ∧ temp ≠ φ) do
```

```
18.            {x := modified-inclusion(T_i, temp, a);
19.             if x.num > 0 then temp := temp/<S_{j+1}, ..., S_{j+x.number}>;
20.             else {if T_i and S_{j+1} have the same label
21.                   then {if x.subnum = number of S_{j+1}'s subtrees
                            then temp := temp/<S_{j+1}>; a := 0; x.num := 1;}
22.                         else a := a + x.subnum;}
23.             i := i + 1; j := j + x;}
24.         if tag = 1
25.         then {tag := 0;
26.               if j > 0 then {if r_2 is a virtual root then return <0, a + j>;
27.                              else return <0, a>;}
28.              }
29.         if temp ≠ ϕ then {if r_2 is a virtual root then return <j, 0>
30.                           else return <0, j>;}
31.         else {if r_2 is a virtual root then return <l, 0>
32.               else return <1, 0>;}}
33.  else {for i = 1 to k do
34.         {x := modified-inclusion(T_i, S);
35.          if x.num = number-of-forest(S) return <x.num, 0>;}
36.          (* number-of-forest(S) returns the number of the forests in S.*)
37.          return <0, subnum>;}
end
```

The algorithm works in the same fashion as *tree-inclusion*( ), but with a significant difference: *modified-inclusion*( ) returns a pair <*num*, *subnum*> instead of a single number, which leads to the following changes.

1) In lines 1 - 7, we try to find an i such that $|<S_1, ..., S_i>| \le |T| < |<S_1, ..., S_{i+1}>$. If it is not the case, we will try to find an j such that $|<Z_{a+1}, ..., Z_j>| \le |T| < |<Z_{a+1}, ..., Z_{j+1}>|$, where $Z_{a+1}, ..., Z_{j+1}$ are some subtrees of S_1. At the very beginning, $a = 0$. In addition, we set $tag = 1$, indicating that the current computation is to check whether T contains any subtrees of S_1. The result is recorded in *subnum*.

2) In lines 20 - 23, we handle the case that the return value of *modified-inclusion*(T_i, *temp*, *a*) is of the form <0, *subnum*>. It indicates that the subtrees of T_i's root do not include any of S_1, ..., and S_i. However, if the labels of T_i's root and S_1's root are the same and *subnum* is equal to the number of the children of S_1's root, we have S_1 included in T_i. Obviously, such a check needs only a contant time. But in *tree-inclusion*( ), a recursive call of the function is performed for the same task. Therefore, *modified-inclusion*( ) is much more efficient than *tree-inclusion*( ).

The time complexity of *modified-inclusion*( ) can be analysed as follows.

$$C_{0,0} \le 1 + p\Big(\sum_{i=1}^{k}(C_{i,l_i\ldots l} + pc)\Big) + q\Big(\sum_{i=1}^{k} C_{i,0}\Big) \qquad (5)$$

where c represents a constant.

Then, we have

$$C_{0,0} \le 1 + p\Big(\sum_{i=1}^{k}\sum_{j=1}^{j}|T_{ij}| \cdot |\mathrm{leaves}(S_{j_i})| + pc\Big) + q\Big(\sum_{i=1}^{k}|T_i| \cdot |\mathrm{leaves}(S_0)|\Big) \qquad (6)$$

$$\le 1 + p\Big(|T_0| - 1 + \frac{pc}{|\mathrm{leaves}(S_0)|}\Big)|\mathrm{leaves}(S_0)| + q(|T_0| - 1)|\cdot|\mathrm{leaves}(S_0)|$$

$$\le |T_0|\cdot|\mathrm{leaves}(S_0)|.$$

The space complexity of *modified-inclusion*( ) is still bounded by $O(|T| + |S|)$.

VII. CONCLUSION

In this paper, a top-down algorithm for checking the tree inclusion of S in T is discussed. The time and space complexities of the algorithm are bounded by $O(|T|\cdot|\mathrm{leaves}(S)|)$ and $O(|T| + |S|)$, repectively.

ACKNOWLEDGEMENT

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

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

[3] Y. Chen, "Query Evaluation and Web Recognition in Document Databases," in *Proc. 7th IASED Int. Conf. on Internet and Multimedia Systems and Applications*, Honolulu, Hawaii, USA, Aug. 13-15, 2003.

[4] C. Faloutsos, "Signature Files," in: *Information Retrieval*: *Data Structures & Algorithms*, edited by W.B. Frakes and R. Baeza-Yates, Prentice Hall, New Jersey, 1992, pp. 44-65.

[5] H. Mannila and K.-J. Raiha, "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.

[6] P. Kilpelainen and H. Mannila, "Ordered and unordered tree inclusion," *SIAM Journal of Computin*g, 24:340-356, 1995.

[7] D.E. Knuth, *The Art of Computer Programming, Vol. 1*, Addison-Wesley, Reading, MA, 1969.

[8] T. 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 1264*, Springer Verlag, pp. 150-166, 1997.