 ＩＳＢＮ：978-986-02-7580-3

A Top Down Algorithm for Constrained Tree Inclusion

Keh-Ning Chang, Yu-Hsiang Hsiao, and Chun-Chieh Liao
Department of Computer Science and Information Engineering
National Chi Nan University, Nantou, Taiwan
{klim,s94321904}@ncnu.edu.tw, nick5205819@gmail.com

Abstract

An ordered labeled tree is a tree which nodes are labeled and in which the left-to-right order among siblings is significant. Given two ordered labeled trees P and T, the constrained tree inclusion problem is to determine whether it is possible to obtain P from T by deleting degree-one or degree-two nodes. G. Valiente proposed a bottom up algorithm which solves the problem in $O(|P||T|)$ time and space. In this paper, a top down matching algorithm is presented, which solves the problem in $O(|P||T|)$ time and $O(|P||leaves(T)|)$ space.

1 Introduction

As appeared in almost every branch of algorithm design, tree is an important data structure. Any possible operations of it may lead to interesting applications. Among the operations, editing is the most basic one and it does have been studied thoroughly [5, 2, 7]. But sometimes even the primitive editing is considered too powerful. Generally, an editing operation is one of the three: deletion, insertion, and substitution. If only deletion is allowed, the original tree editing problem is reduced to tree inclusion problem. The reduction can be applied even further that deletion is only possible for nodes with degree one or two. This more restricted problem, which is the focus of this paper, is called *constrained tree inclusion problem.*

Let T be a tree, and x, y are nodes in T with y being the parent of x. Let $delete(T, x)$ denote the tree obtained from T by removing the node x. The children of x become the children of y. An example of deletion operation is given in Figure 1.

The *tree inclusion problem* is defined as follows. Given two trees P and T, the *pattern* and the *text tree* respectively, can we obtain P by deleting some nodes from T? That is, is there a sequence $x_1, \ldots, x_k$ of nodes such that $T_0 = T, T_i = delete(T_{i-1}, x_i)$ for $i = 1, \ldots, k$, and then, we have

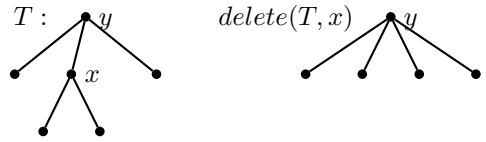

Figure 1: The effect of removing a node from a tree.

$T_k = P$? If this is the case, we say *P is included in T*. In constrained tree inclusion, these deletion operations are allowed on degree-one nodes (leaves) and degree-two nodes (with one child) only. For distinction, we say *P is c-included in T* if P can be obtained from T by deleting degree-one or degree-two nodes.

Although tree inclusion is just a special case of the general tree editing problem, it has its motivation on Internet. Bille and Gørtz [1] give a good explanation on the relationship between tree inclusion and Internet. When surfing the net, one often submits queries to search engines to find the information. The structure of web pages written in HTML is just a tree and the query is just another small tree. In order to determine whether a document satisfies the query, tree inclusion test should be made. This is why the tree inclusion problem deserves its special treatment.

Due to the generality of tree inclusion, the solution to a tree inclusion query is not sensitive to the structure of the query [6]. Many structural forms of the same pattern (that is, many different pattern trees with the same labeling) may be included in the same text tree. For instance, three different structural forms of the same query (pattern tree) are shown in Fig. 2 which are all included in the same text tree.

The need to distinguish the structure aroused the design of constrained tree inclusion. Gabriel [6] proposed algorithms which run in $O(m^{1.5}n)$ time for unordered trees with m and n nodes, and in $O(mn)$ time for ordered trees, both using $O(mn)$ additional space. In this paper, a

 ＩＳＢＮ：978-986-02-7580-3

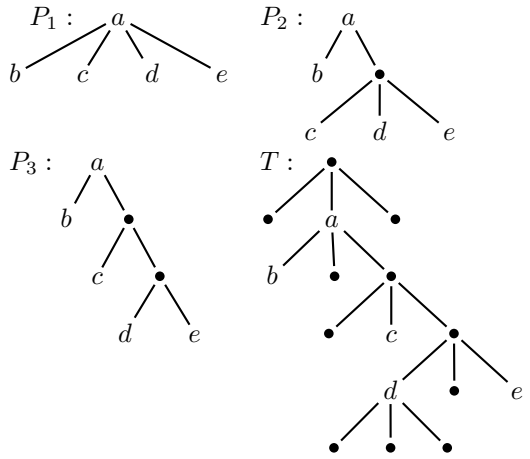

Figure 2: Three forms of the same query are all included at the node labeled a in the text tree.

top down algorithm, for ordered trees, is proposed which solves the problem in $O(mn)$ time but $O(|P||leaves(T)|)$ space is used.

2 Notation

Assume T is an ordered labeled tree with root T_r. If the number of nodes of T is greater than one, T consists of a root T_r and the immediate subtrees $T_1, \ldots, T_k$. For any node x in T, a subtree of T consisting of x and its descendants is denoted by $T(x)$. Let $label(x)$, $post(x)$ and $pre(x)$, respectively, denote the label of node x, the postorder number of x and the preorder number of node x. Let $V(T)$ denote the set of nodes of T and $outdeg(x)$ denotes the number of children of node x. Notation $leaf[x]$ is boolean and it indicates if x is a leaf or an internal node.

The set of left relatives of a node x, denoted by $lr(x)$, are the nodes that precede x both in preorder and postorder. The definition is $lr(x)$ is the set of nodes y such that $pre(y) < pre(x)$ and $post(y) < post(x)$. The right relatives of x are those nodes that follow x both in preorder and postorder. A node on the right (left) side of x is a node in $rr(x)$ ($lr(x)$). For example, in Figure 3, the first number companied with the label of each node is a preorder number, and the second is a postorder number. So $pre(a) = 1, post(a) = 7$. The left relatives of node c is $\{b, g\}$. Node f is on the right side of c.

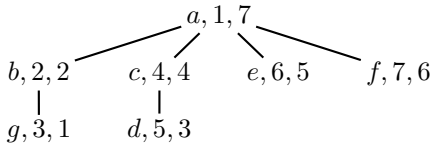

Figure 3: A labeled tree with preorder and postorder numbers.

3 Embedding

This section discusses tree inclusion is equivalent to tree embedding and constrained tree inclusion has its special treatment for tree embedding.

Let P and T be two trees. An embedding of P in T is an injection $f : V(P) \to V(T)$ if the following conditions hold:

1. f preserves labels: $label(x) = label(f(x))$, $\forall x \in V(P)$;
2. f preserves ancestors: x is an ancestor of y iff $f(x)$ is an ancestors of $f(y)$, $\forall x, y \in V(P)$;
3. f preserves the left-to-right order of nodes: $pre(x) < pre(y)$ iff $pre(f(x)) < pre(f(y))$, $\forall x, y \in V(P)$.

Kilpeläinen [4] has shown that P is included in T iff there is an embedding of P in T. So tree inclusion problem can be solved in the way of node mapping [3]. To solve constrained tree inclusion in the same way, one more condition must be involved in tree embedding:

Lemma 3.1. *Let P and T be two trees. Assume there is an embedding of P in T and p is mapped to t for some node p of P and some node t of T. P is c-included in T iff at most one child of p is mapped to one child of t.*

It is easy to see why the condition is necessary and sufficient for constrained tree inclusion. If P is c-included in T and more than one child of p are mapped to one child of t, there must exist a node t with degree at least three to be deleted, contradicting the deletion operation of constrained tree inclusion. If the condition is hold, it is obvious that all deleted nodes are with degree one or two.

For example, P_1 in Fig. 2 is not c-included in T since $T(c), T(d), T(e)$ of P are all embedded in one immediate subtree of $T(a)$ in T. P_3, however, is c-included in T since all nodes of P_3 follow this condition.

The above discussion, thus, suggests a top down approach which is shown in Algorithm 1: First parent node p is compared with nodes of $T(t)$ in

preorder, looking for nodes with the same label. If some node is found, the children of p are processed in similar way except the starting point in $T(t)$ should be chosen such that it follows the all conditions of tree embedding (the ancestor and left-to-right order are preserved; at most one child of p is mapped to one child of t).

The time complexity of Algorithm 1 is, however, exponential. Consider P a b-leaf with n a-ancestors and T a chain of $2n$ a-nodes. For some node p of P and some node t of T, t is compared with p again and again in Algorithm 1; more than $\binom{2n}{n}$ node comparisons are made by the algorithm. Next section will show how to avoid the repetitive computation.

Algorithm 1 TopDown(p, t)

```
 1. if label(p) = label(t) then
 2.     if leaf[p] then
 3.         return success
 4.     i ← 1; j ← 1
 5.     while i ≤ outdeg(p) and j ≤ outdeg(t)
        do
 6.         result ← TopDown(p_i, t_j)
 7.         if result =success then
 8.             i = i + 1
 9.         j = j + 1
10.     if i > outdeg(p) then
11.         return success
12. for j = 1 to outdeg(t) do
13.     result ← TopDown(p, t_j)
14.     if result =success then
15.         return success
16. return fail
```

4 The Improved Algorithm

Although previous top down approach conducts many repetitions, the comparisons involved with P_r, the root of the pattern tree, is an exception. Each node of T is compared with P_r at most once and the comparisons are done in preorder. This simple fact can be easily deduced from the code.

If the compared nodes are saved, the property of P_r can be extended to other nodes of P. Let's see how this work is done.

Let us examine when the nodes of P are compared. Each node of P, except P_r, is compared with some node of T only when its parent's label has been compared successfully (line 1 of Algorithm 1). Then the *while* loop is executed and the comparison is invoked (line 6). If we can avoid the repetitive invocation of TopDown() in *while* loop, repetitive comparisons will not occur. The avoidance is achieved in two steps. First,

Algorithm 2 ImpTopDown(p, t)

```
 1. for i = 1 to outdeg(p) do
 2.     // p_i^l[0] : (t_m, t_n, r_0)
 3.     if pre(t) ≥ pre(t_n) then
 4.         remove p_i^l[0]
 5. if label(p) = label(t) then
 6.     if leaf[p] then
 7.         return (t, success)
 8.     i ← 1; j ← 1
 9.     while i ≤ outdeg(p) and j ≤ outdeg(t) do
10.         if t_j in p_i^l[0] : (t_m, t_n, r_0) then
11.             if (t_j is ancestor of t_n or t_j = t_n) and
                r_0 = success then
12.                 i = i + 1
13.             j = j + 1
14.         else
15.             m ← t_j
16.             for k = j to outdeg(t) do
17.                 n, r ← ImpTopDown(p_i, t_k)
18.                 if r = success then
19.                     break
20.             if t in range p_i^l[0] then
21.                 insert (m, n, r) before p_i^l[1]
22.             else
23.                 insert (m, n, r) before p_i^l[0]
24.             j = k + 1
25.             if r = success then
26.                 i = i + 1
27.     if i > outdeg(p) then
28.         return (t, success)
29. node, result ← (t, fail)
30. for j = 1 to outdeg(t) do
31.     node, result ← ImpTopDown(p, t_j)
32.     if result =success then
33.         break
34. return (node, result)
```

when TopDown(p_i, t_j) (line 6) succeeds, a range of nodes, $(t_m : t_n)$, is saved $(m, n \in N,\ m \le n)$. $(t_m : t_n)$ indicates a range of nodes with preorder numbers from m to n. Those nodes have been compared with p_i. Since each p_i might have a couple of ranges during the course of Algorithm 1, a list of ranges is maintained by p_i. Second, when parent's label is compared successfully and children are ready to be compared again (line 5), the saved list are employed to cut the unnecessary invocation, thus avoiding the repetition.

It seems that each element of a list has to be checked to avoid repetition. But it is sufficient

only to check the first element of a list.

Let p_i^l denote the list of p_i and $p_i^l[k]$ denote the k-th element of p_i^l. The improved algorithm is shown in Algorithm 2. In Algorithm 2, line 2 asserts the value of $p_i^l[0]$ is (t_m, t_n, r_0). Line 10-13 indicates the avoidance of repetitive comparisons. Line 15-23 shows how a range is created and inserted into a list.

4.1 Example

Given two ordered trees P and T in Fig 4, the trace of Algorithm 2 is shown in table 1. The first column of table 1 presents the invocations generated by ImpTopDown(P_r, T_r) and the returned pair by each invocation. The second column shows what happened in each invocation, where we pay more attention on the status of the saved lists and how the unnecessary invocation is cut. Each node of T is identified with t_i in which i is the preorder number, so is each node of P.

You may find out it is possible to check the first element of a list when cutting the unnecessary invocation. Next subsection shows the correctness of this property.

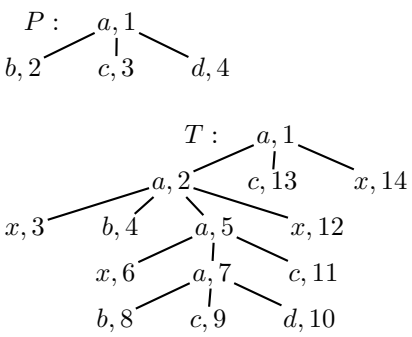

Figure 4: Two trees.

4.2 Checking the Front

For some invocation ImpTopDown(p, t), suppose that p_i is a child of p and $p_i^l = [(t_{m_0}, t_{n_0}, r_0), (t_{m_1}, t_{n_1}, r_1), \ldots, (t_{m_k}, t_{n_k}, r_k)]$ with the order $t_{n_q} \in lr(t_{m_{q+1}})$ for $q = 0, \ldots, k-1$. For some element (t_{m_q}, t_{n_q}, r_q) of p_i^l and some child t_j of t, if $pre(t) \geq pre(t_{n_q})$, then $pre(t_j) > pre(t_{n_q})$, which indicates (t_{m_q}, t_{n_q}, r_q) is of no use to test the repetitive comparison of p_i with t_j. Thus, (t_{m_q}, t_{n_q}, r_q) is removed from p_i^l. For example in table 1, when ImpTopDown(p_1, t_4) is invoked, the element $(t_2, t_4, succ)$ is removed from p_2^l.

In addition to removing the useless elements, the generated range in each invocation has to be inserted at the proper position of p_i^l to preserve the order of p_i^l. Due to the preservation of the order, it is sufficient to check only the first element of p_i^l to avoid the repetitive comparison. Let's see how the order is kept.

When the useless elements are removed, t is either in the range of $p_i^l[0]$ $(pre(t_{m_0}) \leq pre(t) < pre(t_{n_0}))$ or on the left side of t_{m_0}. The latter case implies $t_j \in lr(m_0)$, which indicates t_j has not been compared with p_i. Thus, a range is generated and inserted at the front of p_i^l. For instance, (t_5, t_9) in Fig 5 is inserted at the front of p_3^l.

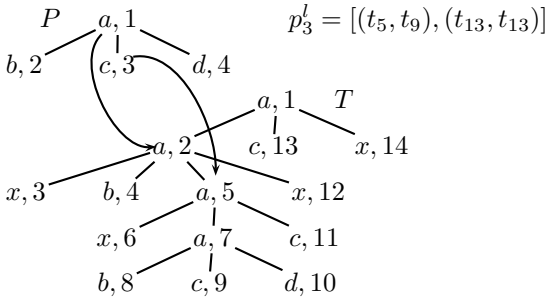

Figure 5: Insertion at the front of p_3^l.

When t is in the range of $p_i^l[0]$, t_j is either in the same range or on the right side of t_{n_0}. The former case indicates t_j has been compared with p_i. An instance of this case is shown in Fig. 6, where the comparison of p_2 with t_3 is avoided. The latter case implies $t_j \in lr(m_1)$ (since $t \in lr(m_1)$). Thus the generated range is inserted before the second element of p_i^l. For example, (t_{11}, t_{11}) in Fig. 7 is inserted at index 1 of p_3^l.

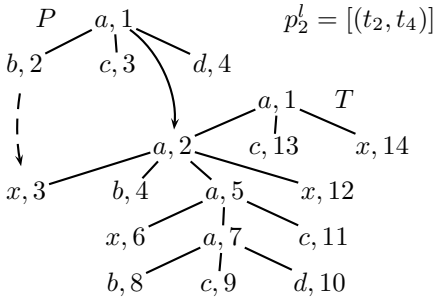

Figure 6: The avoidance of the repetitive comparison indicated by a dashed line.

4.3 Complexity

The improved algorithm has an important property which can be easily deduced from above

Table 1: Trace of Algorithm 2.

Invocation:	Explanation:
Imp $a,1$ $a,1$	p_2^l, p_3^l, p_4^l are all empty.
. Imp $b,2$ $a,2$	compare children (line 17)
. . Imp $b,2$ $x,3$	
. . ret $(t_3, fail)$	
. . Imp $b,2$ $b,4$	
. . ret $(t_4, succ)$	
. ret $(t_4, succ)$	$p_2^l = [(t_2, t_4, succ)]$ (line 23)
. Imp $c,3$ $c,13$	
. ret $(t_{13}, succ)$	$p_3^l = [(t_{13}, t_{13}, succ)]$
. Imp $d,4$ $x,14$	
. ret $(t_{14}, fail)$	$p_4^l = [(t_{14}, t_{14}, succ)]$
. Imp $a,1$ $a,2$	nothing is removed;
. .	$p_2^l = [(t_2, t_4, succ)], p_3^l = [(t_{13}, t_{13}, succ)], p_4^l = [(t_{14}, t_{14}, fail)]$).
. .	the comparison of p_2 with t_3, t_4 are avoided (line 10). (See Fig. 6.)
. .	compare p_3 with t_5. t_5 is not in range $p_3^l[0]$.
. . Imp $c,3$ $a,5$	
. . . Imp $c,3$ $x,6$	
. . . ret $(t_6, fail)$	
. . . Imp $c,3$ $a,7$	
. . . . Imp $c,3$ $b,8$	
. . . . ret $(t_8, fail)$	
. . . . Imp $c,3$ $c,9$	
. . . . ret $(t_9, succ)$	
. . . ret $(t_9, succ)$	
. . ret $(t_9, succ)$	$p_3^l = [(t_5, t_9, succ), (t_{13}, t_{13}, succ)]$ (See Fig. 5.)
. . Imp $d,4$ $x,12$	
. . ret $(t_{12}, fail)$	$p_4^l = [(t_{12}, t_{12}, fail), (t_{14}, t_{14}, fail)]$
. . Imp $a,1$ $x,3$	nothing is removed.
. . ret $(t_3, fail)$	
. . Imp $a,1$ $b,4$	$(t_2, t_4, succ)$ is removed from p_2^l. $p_2^l = [], p_3^l = [(t_5, t_9, succ), (t_{13}, t_{13}, succ)]$,
. .	$p_4^l = [(t_{12}, t_{12}, fail), (t_{14}, t_{14}, fail)]$
. . ret $(t_4, fail)$	
. . Imp $a,1$ $a,5$	nothing is removed.
. . . Imp $b,2$ $x,6$	
. . . ret $(t_6, fail)$	
. . . Imp $b,2$ $a,7$	
. . . . Imp $b,2$ $b,8$	
. . . . ret $(t_8, succ)$	
. . . ret $(t_8, succ)$	$p_2^l = [(t_6, t_8, succ)]$
. . . Imp $c,3$ $c,11$	
. . . ret $(t_{11}, succ)$	$p_3^l = [(t_5, t_9, succ), (t_{11}, t_{11}, succ), (t_{13}, t_{13}, succ)]$(line 21) (See Fig. 7.)
. . . Imp $a,1$ $x,6$	nothing is removed. $p_2^l = [(t_6, t_8, succ)], p_4^l = [(t_{12}, t_{12}, fail), (t_{14}, t_{14}, fail)]$
. . .	$p_3^l = [(t_5, t_9, succ), (t_{11}, t_{11}, succ), (t_{13}, t_{13}, succ)]$
. . . ret $(t_6, fail)$	
. . . Imp $a,1$ $a,7$	nothing is removed. the comparison of p_2 with t_8 is avoided.
. . .	the comparison of p_3 with t_9 is avoided.
. . .	compare p_4 with t_{10}, t_{10} is not in range $p_4^l[0]$
. . . . Imp $d,4$ $d,10$	nothing is removed.
. . . . ret $(t_{10}, fail)$	
. . . ret $(t_7, succ)$	
. . ret $(t_7, succ)$	
. ret $(t_7, succ)$	
ret $(t_7, succ)$	

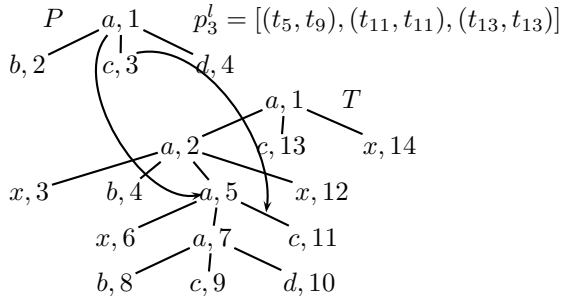

Figure 7: Insertion before the second element of p_3^l.

discussion and is formalized in the following lemma.

Lemma 4.1. *Given two trees, P and T, each node of T is compared with p at most once for all $p \in V(P)$ in the algorithm.*

And this lemma implies the following theorem.

Theorem 4.2. *Given two trees P and T, the proposed algorithm tests whether P is c-included in T in time $O(mn)$, where m and n are the number of the nodes of P and T, respectively.*

Proof. The dominant operations of the algorithm are the invocations of ImpTopDown() , the tests of $p_i^l[0]$ (line 10), and removing $p_i^l[0]$. Each invocation of ImpTopDown() contains exactly one comparison. So, from the previous lemma, it can be deduced that the number of invocation of ImpTopDown() is $O(mn)$.

For a certain node p of P and a certain t of T, when the labels of p and t are compared successfully, line 10 is tested. And the number of the tests is $O(outdeg(t))$. Since p compares each node of T at most once, the total number of the tests, for a certain p, is $O(n)$. Thus the total number of the tests of $p_i^l[0]$, for all nodes of P, is $O(mn)$.

In similar deduction, the number of removing $p_i^l[0]$ is $O(mn)$. Therefore, the time complexity of the algorithm is $O(mn)$. □

For the list of some node p of P, since the elements of the list are not overlapped, the space used by the list is $O(|leaves(T)|)$. So the space complexity of the improved algorithm is $O(|P||leaves(T)|)$.

5 Conclusion

We have presented a top down algorithm for constrained tree inclusion on ordered trees, which improves the previous one [6] in space complexity. The key idea is the use of the saved list to avoid the repetitive computation. Constant-time checking of the list enables us to run the algorithm in $O(|P||T|)$ time.

References

[1] P. Bille and I. Li Gørtz. The tree inclusion problem: In optimal space and faster. *Automata, Languages and Programming*, pages 66–77, 2005.

[2] E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):1–19, 2009.

[3] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24:340, 1995.

[4] Pekka Kilpelinen. Tree matching problems with applications to structured text databases. Technical report, 1992.

[5] K.C. Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):433, 1979.

[6] G. Valiente. Constrained tree inclusion. *Journal of Discrete Algorithms*, 3(2-4):431–447, 2005.

[7] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.