The tree inclusion problem

Pekka Kilpeläinen Heikki Mannila

Department of Computer Science
University of Helsinki
Teollisuuskatu 23
SF-00510 Helsinki, Finland

Abstract

We consider the following problem: Given ordered labeled trees S and T, can S be obtained from T by deleting nodes? Deletion of the root node u of a subtree with children $\langle T_1, \ldots, T_n \rangle$ means replacing the subtree by the trees $T_1, \ldots, T_n$. The problem is motivated by the study of query languages for structured text data bases. The simple solutions to this problem require exponential time. We give an algorithm based on dynamic programming requiring $O(|S||T|)$ time and space.

1 Introduction and motivation

Let T be a tree and u the parent node of a node v in T. Denote by

$$delete(T, v)$$

the tree obtained from T by removing the node v. The children of v become children of u. (See Figure 1.)

We consider the following problem: Given two trees S and T, can we obtain S by deleting some nodes from T? That is, is there a sequence $u_1, \ldots, u_k$ of nodes such that for

$$\begin{aligned} T_0 &= T \\ T_{i+1} &= delete(T_i, u_{i+1}),\ i = 0, \ldots, k-1, \end{aligned}$$

we have $T_k = S$. If this is the case, what are the nodes in T that are not deleted by the sequence? We call this problem the *tree inclusion problem.*

Our motivation comes from the study of query languages for structured text databases. Gonnet and Tompa have defined an elegant data model where instances are parse trees [GoT87]. Tree inclusion seems to capture essential properties of ordering and hierarchical containment involved in Gonnet and Tompa's p-string algebra [MaR90]. Therefore it appears to be a useful concept for expressing the operations of the p-string algebra. Another application is the processing of queries in object-oriented databases [Bee90]: a query describes an included subtree of the type structure.

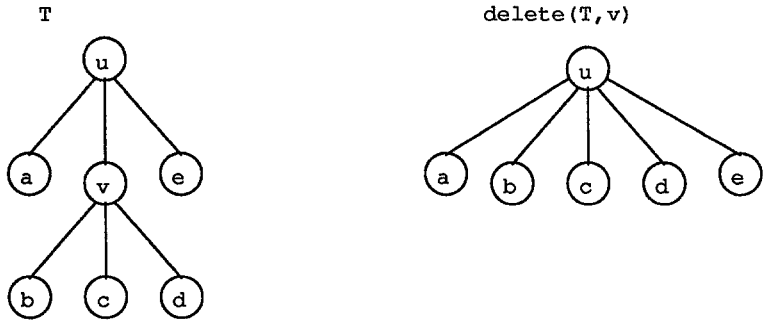

Figure 1: The effect of removing the node v from the tree T

The tree inclusion problem resembles the *tree pattern matching problem* [HoO82, Kos89, CPT90, DGM90], which has several applications. In that problem one wants to find out whether the tree S occurs as a connected component of the tree T, or, using the terminology above, whether one can obtain a tree having S as a subtree by removing complete subtrees from T. The resemblance of the problems does not apparently extend to the algorithms. For the tree pattern matching problem the trivial solution works in time $O(|S||T|)$, where $|S|$ is the number of nodes in the tree S, and $|T|$ in the tree T. (This bound has only recently been broken [Kos89, DGM90].) The reason why tree pattern matching is a fairly simple problem is that there are only $|T|$ possible places where S can be matched in T. Hence the trivial algorithm can simply test all the possibilities.

For the tree inclusion problem, there can generally be exponentially many ways to obtain the included tree. Thus it is not feasible to check all the possibilities. In this paper we give a polynomial time algorithm for solving the tree inclusion problem. The algorithm works in time $O(|S||T|)$. It is based on dynamic programming: to use that approach we need to develop some results about the properties of tree inclusion. In this paper we concentrate on these properties and the polynomial time algorithm, and do not describe the applications in more detail (see [Kil91]).

We start by giving definitions and some properties of tree inclusion in Section 2. In Section 3 we develop the idea of *left embeddings* that is the base for our algorithms. Section 4 presents the basic algorithm. This algorithm may repeat the same computations many times, which leads to an exponential behaviour. In Section 5 we develop the algorithm further. The use of a table for storing the results of subcomputations leads to an algorithm requiring $O(|S||T|)$ running time and storage. Section 6 discusses related work. Section 7 is a short conclusion.

2 Forest inclusion

Notationally it is useful to be slightly more general and to consider ordered and labeled forests instead of trees. 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 $\langle T_1, \ldots, T_n \rangle$, where $n \geq 0$. The trees $T_1, \ldots, T_n$ are called the *children* of the root. The tree rooted by node u is denoted by $tree(u)$. The root is an *ancestor* of all the nodes in its

children, and the nodes in the children are *descendants* of the root. The set of descendants of a node u is denoted by $desc(u)$. A *leaf* is a node without any descendants.

We sometimes treat a tree T as the forest $\langle T \rangle$. We may denote the set of nodes in a forest F by F, too. For example, if we speak of functions from a forest F to a forest G, we mean functions mapping *nodes* of F onto *nodes* of G. The *size* of a forest F, denoted by $|F|$, is the number of nodes in F.

Let $F = \langle T_1, \ldots, T_n \rangle$ be a forest. The *preorder of a forest* F is the order of the nodes visited during the preorder traversal. The *preorder traversal of a forest* $\langle T_1, \ldots, T_n \rangle$ is defined as follows: Traverse the trees $T_1, \ldots, T_n$ in ascending order of the indices in preorder. To traverse a tree in preorder, first visit the root and then traverse the children of the tree in preorder. The *postorder* is defined similarly, except that the root is visited *after* traversing the children in postorder. The preorder and postorder numbers of a node u are denoted by $pre(u)$ and $post(u)$.

The following lemma binds the ancestorship and the preorder and postorder together.

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

Proof. See Exercise 2.3.2-20 in [Knu69]. □

Let Σ be a set of labels and N a set of nodes. A *labeling* for N is a function from N to Σ.

Definition 1 Let F and F' be forests, N and N' the sets of their nodes, and l a labeling for $N \cup N'$. An injective function

$$f : N \to N'$$

is *an embedding of* F *into* F', if for all nodes $u, v \in N$

1. f *preserves labels*, i.e., $l(f(u)) = l(u)$.
2. f *preserves preorder*, i.e., $pre(u) < pre(v) \Leftrightarrow pre(f(u)) < pre(f(v))$.
3. f *preserves ancestors*, i.e., $f(u) \in desc(f(v)) \Leftrightarrow u \in desc(v)$.

If there is an embedding of F into F', we say that F' *includes* F, denoted $F \sqsubseteq F'$. F is an *included forest* of F', and F' is an *including forest* of F. Forests F and F' are *isomorphic*, if there is a bijective embedding of F into F'. □

Using Lemma 1 one can show that we get an equivalent definition for embeddings if we replace condition (2) or condition (3) above by the requirement that for all nodes u and v

4. f preserves postorder, i.e., $post(u) < post(v) \Leftrightarrow post(f(u)) < post(f(v))$.

Example 1 The included forests of the forest $\langle a(b, c) \rangle$ are $\langle\rangle$, $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, $\langle b, c \rangle$, $\langle a(b) \rangle$, $\langle a(c) \rangle$, and $\langle a(b, c) \rangle$. □

Lemma 2 The relation $\sqsubseteq$ is a partial ordering (up to isomorphism), i.e., it is reflexive, transitive, and antisymmetric. □

The next lemma justifies the formulation of the tree inclusion problem given in the introduction.

Lemma 3 A forest G includes a forest F if and only if a forest isomorphic to F can be obtained from G by deleting nodes. (The deletion of a node v replaces $tree(v)$ by the children of v.) □

The proof of Lemma 3 is based on the fact that if u and v are nodes in G, the removal of a third node does not change the relative ordering of u and v.

A forest may have exponentially many included forests, as shown in the next lemma.

Lemma 4 A forest F may have $\binom{|F|}{m}$ non-isomorphic included forests of size m.

Proof. If every node in F has a different label, we get a non-isomorphic included forest of size m for every possible selection of $|F| - m$ nodes to be deleted. The number of such selections is $\binom{|F|}{m}$. □

Example 2 The forest $\langle a(x, b(y(d, e), f), c)\rangle$ has 8 nodes with different labels. Therefore it has 256 non-isomorphic included forests. If the labels are similar, the number of non-isomorphic included forests is smaller. For example, the forest $\langle a(a, a(a(a, a), a), a)\rangle$ has same size and form as the previous one, but it has only 68 non-isomorphic included forests. □

3 Left embeddings

In this section we develop concepts that help us to solve the tree inclusion problem efficiently. Throughout the rest of the paper, let l be a labeling for the nodes of the trees.

We concentrate on searching *root preserving embeddings.*

Definition 2 Let S and T be trees. A *root preserving embedding of S into T* is an embedding f of S into T such that $f(root(S)) = root(T)$. □

The following simple lemma states that the we do not lose generality by concentrating on the root preserving embeddings.

Lemma 5 Given forests F and G, let S and T be trees such that $l(root(S)) = l(root(T))$ and the children of $root(S)$ form the forest F and the children of $root(T)$ form the forest G. Then $F \sqsubseteq G$ if and only if there is a root preserving embedding of S into T. □

There may be exponentially many ways to embed the children of a tree S into the children of a tree T. In order to limit the search among these embeddings, we develop algorithms that search for a root preserving embedding of S into T by trying to embed the subtrees of S as deep and as left as possible in T.

In order to discuss the order of images of sibling nodes in an embedding we define the *right and left relatives* of a node.

Definition 3 Let F be a forest, N the set of its nodes, and u a node in F. The set of *right relatives* of u is defined by

$$rr(u) = \{x \in N \mid pre(u) < pre(x) \ \wedge \ post(u) < post(x)\},$$

i.e., the right relatives of u are those nodes that follow u both in preorder and in postorder. (See Figure 2.)

The set of *left relatives* of a node u, denoted by $lr(u)$, is the set of nodes that precede u both in preorder and in postorder. □

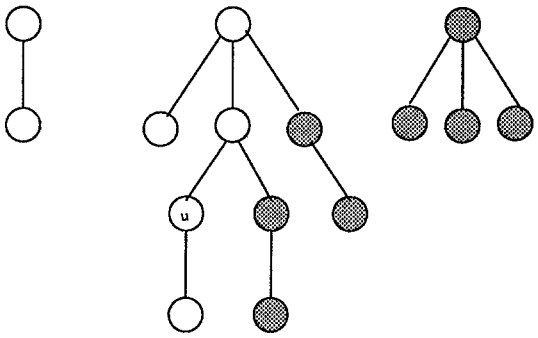

Figure 2: The right relatives of the node u

Lemma 6 Let u, v, x be three nodes in a forest. Then it is not possible that

$$\begin{array}{lllll} pre(u) & < & pre(v) & < & pre(x) \quad \wedge \\ post(x) & < & post(u) & < & post(v) \end{array}$$

Proof. The above conditions imply by Lemma 1 that

> u and v are ancestors of x, and
> neither u nor v is an ancestor of the other,

but this is not possible in a forest. □

The next simple lemma states that the descendants of a right relative are also right relatives.

Lemma 7 Let u and v be nodes, and assume $v \in rr(u)$. Then $desc(v) \subset rr(u)$. □

The next lemma states that the right relatives of a node v are contained in the right relatives of the nodes that precede v in postorder. This fact is the justification for the strategy of embedding the trees as early as possible in postorder, when searching for an embedding of a forest.

Lemma 8 Let u and v be nodes in a forest. If $post(u) \leq post(v)$, then $rr(v) \subseteq rr(u)$. □

Lemmas 7 and 8 follow directly from Definition 3, Lemma 1, and Lemma 6.

Lemma 9 Let $F = \langle T_1, \ldots, T_k \rangle$ and G be forests. There is an embedding of F into G if and only if for every $i = 1, \ldots, k$ there is an embedding f_i of T_i into G, such that $f_{i+1}(root(T_{i+1}))$ is a right relative of $f_i(root(T_i))$, whenever $1 \leq i < k$.

Proof. 'Only if': Let f be an embedding of F into G. It is obvious that restricting f into each T_i gives embeddings f_i satisfying the lemma.

'If': We show by induction on k that the union of the embeddings f_i is an embedding of F into G. Inductive step: Let g be an embedding of $\langle T_1, \ldots, T_{k-1} \rangle$ into G such that $g(root(T_{k-1}))$ is a left relative of $f_k(root(T_k))$ and let $f = g \cup f_k$. It suffices to show that f preserves the relative order of any two nodes $x \in T_1 \cup \ldots \cup T_{k-1}$ and $y \in T_k$. Now $root(T_{k-1})$ is the last node of $\langle T_1, \ldots, T_{k-1} \rangle$ in postorder. Therefore, by Lemma 8 we have $root(T_k) \in rr(x)$ and Lemma 7 states that $y \in rr(x)$ also for $y \neq root(T_k)$. Then $f(y) \in rr(f(x))$ by the same argument applied to the nodes $f(x)$, $f(root(T_{k-1}))$, $f(root(T_k))$, and $f(y)$. □

Definition 4 Let $F = \langle T_1, \ldots, T_n \rangle$ and G be forests, and let $C(F, G)$ be the collection of embeddings of F into G. An embedding $f \in C(F, G)$ is a *left embedding* of F into G if for every $g \in C(F, G)$

$$post(f(root(T_i))) \leq post(g(root(T_i))), \ i = 1, \ldots, n.$$

□

Theorem 1 A forest F is an included forest of a forest G if and only if there is a left embedding of F into G.

Proof. 'If': It is obvious that the condition is sufficient.

'Only if': Assume that $F \sqsubseteq G$. If F is an empty forest, its inclusion into G is the only embedding in $C(F, G)$, and thus a left embedding.

Let $F = \langle T_1, \ldots, T_n \rangle$, $(n \geq 1)$. We prove the statement by induction on n. For the base case, let $F = \langle T \rangle$. Since there is an embedding of F into G,

$$V = \{f(root(T)) \in G \mid f \in C(F, G)\}$$

is a non-empty finite set of nodes. Therefore

$$min\{post(x) \mid x \in V\} = post(f(root(T)))$$

for some $f \in C(F, G)$. Such f is a left embedding.

For the inductive step, let $F = \langle T_1, \ldots, T_n \rangle$, $(n \geq 2)$. Since $\langle T_1, \ldots, T_{n-1} \rangle \sqsubseteq F$, we know by Lemma 2 that $\langle T_1, \ldots, T_{n-1} \rangle \sqsubseteq G$. By the inductive assumption there is a left embedding f of the forest $\langle T_1, \ldots, T_{n-1} \rangle$ into G. Let g be an embedding of F into G. Its restriction into $T_1 \cup \ldots \cup T_{n-1}$ is an embedding of the forest $\langle T_1, \ldots, T_{n-1} \rangle$ into G, and thus

$$post(f(root(T_{n-1}))) \leq post(g(root(T_{n-1}))). \tag{1}$$

By Lemma 9 we know that $g(root(T_n)) \in rr(g(root(T_{n-1})))$, and by Lemma 8 we get from (1) that $g(root(T_n)) \in rr(f(root(T_{n-1})))$. Therefore

$$V = \{h(root(T_n)) \in G \mid h \in C(T_n, G) \ \wedge \ h(root(T_n)) \in rr(f(root(T_{n-1})))\}$$

is a nonempty finite set. Thus

$$min\{post(x) \mid x \in V\} = post(h(root(T_n))) \tag{2}$$

for some $h \in C(T_n, G)$. Now $f \cup h$ is a left embedding of F into G: Let ℓ be an embedding of F into G. Then

$$post(f(root(T_i))) \leq post(\ell(root(T_i))) \tag{3}$$

for all $i = 1, \ldots, n-1$. The node $\ell(root(T_n))$ is a right relative of $\ell(root(T_{n-1}))$, and by (3) and Lemma 8 it is also a right relative of $f(root(T_{n-1}))$; therefore $\ell(root(T_n)) \in V$, and by (2)

$$post(h(root(T_n))) \leq post(\ell(root(T_n))).$$

□

4 The basic algorithm

Now we are ready to give an algorithm for testing whether there is a root preserving embedding of $tree(u)$ into $tree(v)$. Let $T_1, \ldots, T_k$ be the children of the node u. First our algorithm searches for the image $f(root(T_1))$ under a left embedding f of T_1 into the children of the node v, if there is any. The algorithm uses a pointer p for traversing the descendants of the node v. After finding a left embedding f for the forest $\langle T_1, \ldots, T_i \rangle$, the pointer p points at the node $f(root(T_i))$. In order to extend f to a left embedding of $\langle T_1, \ldots, T_{i+1} \rangle$ into the children of v we find the next right relative x of p in postorder, such that there is a root preserving embedding of T_{i+1} into $tree(x)$, and x is a descendant of v. (The construction used is the same as in the proof of Theorem 1.)

The algorithm manipulates nodes as postorder numbers. For example, the *min* of a set of nodes is the first node of the set in postorder. The algorithm refers to auxiliary nodes 0 and ∞. Node 0 precedes every node of $tree(v)$ in preorder and in postorder, and every node of $tree(v)$ precedes node ∞ in both orders. Especially, all nodes of $tree(v)$ are right relatives of node 0.

Algorithm 1 Finding a root preserving left embedding of S into T.

Input: Nodes u and v ($u = root(S), v = root(T)$).

Output: **true** if and only if there is a root preserving embedding of S into T.

Method:

```
1. function emb(u, v);
2.    if l(u) ≠ l(v) then return false ;
3.    elsif u is a leaf then return true ;
4.    else
5.          let u_1, ..., u_k be the roots of the children of u;
6.          p := min(desc(v) ∪ {∞}) − 1;
7.          i := 0;
8.          while i < k and p < v do
9.                p := min({x ∈ rr(p) | emb(u_{i+1}, x)} ∪ {∞});
10.               if p ∈ desc(v) then
11.                     i := i + 1;
12.               fi ;
13.         od ;
14.         if i = k then return true ;
15.         else return false ;
16.         fi ;
17.   fi ;
18. end ;
```

The correctness of the algorithm is based on the loop invariant stating that the forest $\langle tree(u_1), \ldots, tree(u_i) \rangle$ has a left embedding f into the children of v, and $p = f(u_i)$. If node v is a leaf but u has children, $tree(u)$ can not be embedded into $tree(v)$. In this

case p gets value ∞ on line 6 of the algorithm, which prevents the execution of lines 8–14. Otherwise p gets the value $min(desc(v)) - 1$, which is the closest left relative of v. (The descendants of v are as postorder numbers $\{p+1, p+2, \ldots, v-1\}$.) By Lemma 7, $desc(v) \subset rr(p)$. These observations mean that the first execution of line 9 finds the first subtree of v, such that $tree(u_1)$ has a root preserving embedding into it, if there is any.

The primitive operations of moving to the first descendant node or to the next or previous node in postorder can be performed in constant time, after a linear time preprocessing of tree T. Note that the tests $p \in desc(v)$ and $x \in rr(p)$ can be realized as simple comparisons of preorder and postorder numbers. (See Lemma 1 and Definition 3.)

The algorithm above may still need exponential time in the size of the trees. Consider trees $S_n = r(a(a(\ldots a(a(b))\ldots)))$ (a b-leaf with n a-ancestors below root r) and $T_n = r(a(a(\ldots a(\ldots a(a)\ldots), b)\ldots)))$ (a chain of $2n$ a-nodes below root r; the nth a has also a b-leaf as a child). The algorithm tries in the recursive calls on line 9 a total of $\binom{2n}{n}$ embeddings before it finds the right images for the a-nodes and an embedding for the whole tree.

5 A dynamic programming solution

In the worst case the previous algorithm repeats same computations an exponential number of times. To avoid this, we use an $m \times n$ table ($m = |S|$, $n = |T|$) to store results of subcomputations. As before, we test for the existence of a root preserving embedding of $tree(u)$ into $tree(v)$ ($u \in S, v \in T$) by trying to find a left embedding of u's children into v's children. The key idea is to organize the evaluation so that the step that extends the left embedding of $\langle T_1, \ldots, T_{i-1} \rangle$ into an embedding of $\langle T_1, \ldots, T_i \rangle$ can find the image node of $root(T_i)$ in *constant time*.

Let us define a table e having rows $1, \ldots, m$ and columns $0, \ldots, n-1$. As before, we refer to the nodes of S and T by their postorder numbers; the numbers of S are used as row indexes of the table, and the numbers of T are used as column indexes and contents of the table. Denote by $R(S,T)$ the collection of root preserving embeddings of a tree S into a tree T. We compute into table e values ($u \in S, v \in T$)

$$e(u,v) = min(\{x \in rr(v) \mid \exists\, f \in R(tree(u), tree(x))\} \cup \{\infty\}). \tag{4}$$

The initial value for the entries of e is ∞. The result of the computation can be found on row $root(S)$ of the table. There is a root preserving embedding of S into $tree(v)$ for every $v \in T$ that appears on row $root(S)$, and S can be embedded into T only if $e(root(S), 0) < \infty$.

Let $T_1, \ldots, T_k$ be the children of node $u \in S$. Like before, we use a pointer p for finding the images of the $root(T_i)$'s in T under a left embedding. When a root preserving embedding of $tree(u)$ into $tree(v)$ is found, another pointer q is used for writing value v into $e(u,q)$ for those nodes q that satisfy equation (4). (Note that those nodes q are left relatives of node v. Therefore columns $0 \ldots n-1$ suffice for the table, since node n has no right relatives except ∞.)

Algorithm 2 Evaluate table e.

Input: The nodes of S and T as postorder numbers,
$e(u,v) = \infty$ for all $1 \le u \le m$, $0 \le v \le n-1$.

Output: $e(u,v) = min(\{x \in rr(v) \mid \exists\, f \in R(tree(u), tree(x))\} \cup \{\infty\})$.

Method:

```
1. for  u := 1,...,m do
2.     let u_1,...,u_k be the roots of the children of u;
3.     q := 0;
4.     for  v := 1,...,n do
5.         if  l(u) = l(v) then
6.             p := min(desc(v) ∪ {∞}) − 1;
7.             i := 0;
8.             while  i < k and  p < v do
9.                 p := e(u_{i+1}, p);
10.                if  p ∈ desc(v) then  i := i + 1; fi ;
11.            od ;
12.            if  i = k then
13.                while  q ∈ lr(v) do
14.                    e(u, q) := v;
15.                    q := q + 1;
16.                od ;
17.            fi ;
18.        fi ;
19.    od ;
20. od ;
```

As an example, consider how Algorithm 2 finds the embedding of a tree $S = A(C,E)$ into a tree $T = A(B(C), A(B(D), A(B(E))))$. The trees and the result of the computation can be seen in Figure 3. Each column of table e is shown to the right of the corresponding node of tree T. (Empty elements of the array denote ∞.)

First, $u = 1$, the leaf of S labelled by C, and $v = 1$, the similar leaf of T. Since the labels match and u has no children ($i = k = 0$), we have an embedding. The value of $v = 1$ is written into $e(1,0)$ only, since 0 is the only left relative of v. After that, no more matching labels are found for $u = 1$ in nodes $v = 2,\ldots,9$ of T.

Next $u = 2$, the second leaf of S. The first node v of T such that $l(v) = l(u) = E$ is node 5. As above, we have an embedding, and the value of $v = 5$ is written into $e(2,q)$ for the left relatives $q = 0,\ldots,4$ of v. Then again the remaining nodes $v = 6,\ldots,9$ are scanned without encountering any matching labels.

Finally $u = 3 = root(S)$. Node $v = 7$ is the first node of T with $l(v) = l(u) = A$. The node p preceeding $desc(v)$ in postorder is node 4. The first child of u is node number 1. Its image in $rr(p) \supset desc(v)$ is looked up from $e(1,4)$; value ∞ means that there is no embedding of the child into $desc(v)$. Next, a similar failure occurs with $v = 8$ and $p = 2$. Finally $v = 9 = root(T)$, $l(v) = l(u) = A$, which leads to testing the embedding of the children of u by executing $p := 0$, $p := e(1,0) = 1$, and $p := e(2,1) = 5$. The algorithm

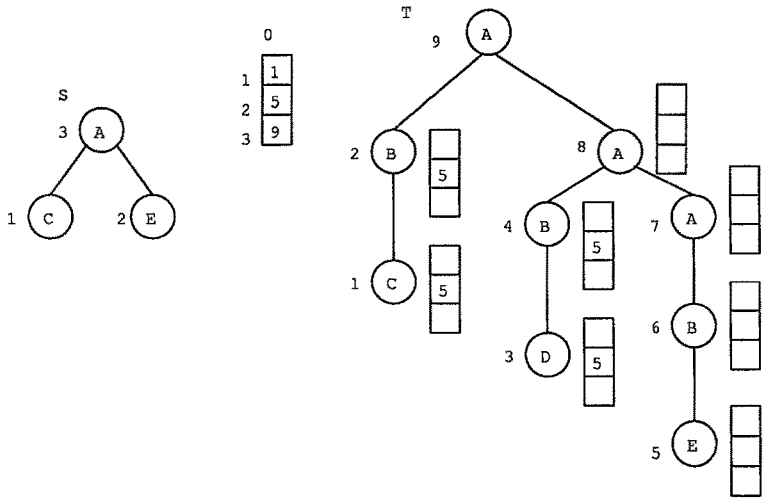

Figure 3: The result of applying the algorithm to trees S and T

has found a root preserving embedding of S into T. The value of $v = 9$ is written as the final result into $e(3, 0)$. Since 0 is the only left relative of 9, the computation ends.

Theorem 2 Algorithm 2 fills table e correctly.

Proof. Keep node u fixed. We outline the proof that all columns of row u get correct values in the **while** -loop on lines 13–16 of the algorithm. First, the precondition that all columns up to q have got the right values is initially true. The following invariant holds on line 14:

$$\exists\, f \in R(tree(u), tree(v)) \quad \wedge \quad v \in rr(q) \wedge \tag{5}$$

$$\forall\, 1 \leq x < v : (\exists\, g \in R(tree(u), tree(x)) \quad \Rightarrow \quad x \notin rr(q)). \tag{6}$$

The invariant tells that the loop writes correct values into $e(u, q)$. Since q only increases in the loop, the truth of the invariant maintains the precondition for the subsequent executions of the loop.

Assume that u is a leaf. Then $e(u, q)$ should be assigned the number of the first node v in $rr(q)$ such that $l(v) = l(u)$. It is clear that (5) holds when we are on line 14. (Note that $q \in lr(v)$ and $v \in rr(q)$ are equivalent.) When line 14 is executed for the first time (6) must be true. By Lemma 8 we can strengthen (6) into

$$\forall\, 1 \leq x < v, \forall\, y \geq q : (\exists\, g \in R(tree(u), tree(x)) \Rightarrow x \notin rr(y)). \tag{7}$$

When we finish the loop, we know that $v \notin rr(q)$, which allows us to deduce from (7)

$$\forall\, 1 \leq x \leq v, \forall\, y \geq q : (\exists\, g \in R(tree(u), tree(x)) \Rightarrow x \notin rr(y)). \tag{8}$$

This postcondition makes (6) true on the subsequent executions of the loop. It also tells that the writing is complete, i.e., value v must not be written into $e(u, y)$ for any column $y \geq q$.

Next, assume that u is a non-leaf node, and the rows of the table e corresponding to the children of u have been correctly computed. Then, as in Algorithm 1 the **while** -loop on lines 8–11 finds a left embedding of the children of u into the children of v, if there is any. The correctness of the **while** -loop on lines 13–16 is verified as in the base case. □

Theorem 3 Algorithm 2 requires $O(mn)$ time and space.

Proof. Space: Table e requires $O(mn)$ space.

Time: During every execution of the outermost loop q may increase in steps of one from 0 to n. Therefore the **while** -loop incrementing q requires $O(n)$ steps per one outermost loop. One execution of the **while** -loop on lines 8–11 requires time $O(1 + k_u)$, where k_u is the number of the children of node u. We get total time

$$O(\sum_{u=1}^{m}(n + \sum_{v=1}^{n}(1 + k_u))) = O(n\sum_{u=1}^{m}(2 + k_u)).$$

The sum $\sum_{u=i}^{m} k_u$ equals the number of edges in tree S, which is $m - 1$. Therefore the total time is $O(n(3m - 1)) = O(mn)$. □

6 Related problems

The tree inclusion problem can be found in Excercise 2.3.2-22 of [Knu69]. Knuth gives a sufficient condition for the existence of an embedding.

The tree inclusion problem can be considered to be a special case of the editing distance problem for trees [Tai79, ZhS89]. In [ZhS89] Zhang and Shasha give an algorithm for computing the edit distance. It can also be used for solving the tree inclusion problem. Their algorithm requires time

$$O(|S| \cdot |T| \cdot min(depth(S), leaves(S)) \cdot min(depth(T), leaves(T))).$$

Thus their solution is slower than ours by a factor of

$$min(depth(S), leaves(S)) \cdot min(depth(T), leaves(T)).$$

Shasha and Zhang have recently presented new sequential and parallel algorithms for the editing distance problem with unit cost edit operations [ShZ90].

The tree inclusion problem is a generalization of the *subsequence problem* for strings ("Can the string x be obtained from the string y by deleting characters?"), and a special case of the *graph minor problem* ("Can the graph H be obtained from a subgraph of G by contracting edges?") [RoS86, Joh87]. The subsequence problem can be solved in linear time by a straightforward scan. The minor problem for graphs and even for unordered trees is NP-complete, when both H and G are given as inputs.[1]

The classification above suggests some further problems. It is possible to compute a maximal common subsequence for two strings in quadratic time and in linear space [Hir75]. The corresponding problem of finding a maximal common included tree of two trees can be solved by Zhang and Shasha's editing distance algorithm [ZhS89]. Our

[1]For every *fixed* planar graph H (and therefore for every forest) there is a polynomial time algorithm for testing whether H is a minor of a given graph G.

algorithm specialized for the tree embedding problem is simpler and more efficient than Zhang and Shasha's algorithm. It is not clear yet if the algorithm can be extended to solve the largest common included tree problem also.

A related problem is the *tree pattern matching problem.* (See e.g. [HoO82, CPT90].) In tree pattern matching one is given a pattern tree, possibly with variables standing for arbitrary subtrees, and a subject tree. The problem is to locate the subtrees of the subject tree that are isomorphic to some tree presented by the pattern. The $O(mn)$ time bound of the naive algorithm has been difficult to improve for the general case. A recent paper of Kosaraju [Kos89] presents an $O(nm^{0.75}polylog(m))$ algorithm. Dubiner, Galil and Magen improve this result in their paper [DGM90] by presenting an $O(n\sqrt{m}\,polylog(m))$ algorithm. Checking whether an *unordered tree* is a subtree of another one can be done in time $O(mn^{3/2})$ using maximal matching [Rey77].

7 Conclusions

We have considered the tree inclusion problem, which arises from database query processing. We have given a dynamic processing solution requiring $O(mn)$ time, where m and n are the sizes of the trees. The algorithm is faster than the previous ones.

There are several open problems. One is improving the running time of the method. Breaking the mn-barrier seems rather hard, however. Another promising area is trying to reduce some matching problems to the tree inclusion problem; this could give upper or lower bounds for the complexity of this problem.

Acknowledgements

We thank Kari-Jouko Räihä for useful comments.

References

[Bee90] Beeri, C., A formal approach to object-oriented databases. Data & Knowledge Engineering 5 (1990), 353–382.

[CPT90] Cai, J., Paige, R., Tarjan, R., More efficient bottom-up tree pattern matching. In: Proc. of the 15th Colloqium on Trees in Algebra and Programming (CAAP'90), p. 72–86.

[DGM90] Dubiner, M., Galil, Z., Magen, E., Faster tree pattern matching. In: Proc. of the Symposium on Foundations of Computer Science (FOCS'90), p. 145–150.

[GoT87] Gonnet, G., Tompa, F., Mind your grammar - a new approach to text databases. In: Proc. of the Conference on Very Large Data Bases (VLDB'87), p. 339–346.

[HoO82] Hoffman, C.M., O'Donnell, M.J., Pattern matching in trees. J. ACM, 29, 1 (January 1982), 68–95.

[Joh87] Johnson, D.S., The NP-completeness column: an ongoing guide. J. Algorithms 8 (1987), 285–303.

[Hir75] Hirschberg, D.S., A linear space algorithm for computing maximal common subsequences. CACM 18, 6 (June 1975), 341–343.

[Kil91] Kilpeläinen, P., Query languages for structured text databases. Ph.D. Thesis, University of Helsinki; in preparation.

[Knu69] Knuth, D.E., The Art of Computer Programming, Vol. 1. Addison-Wesley 1969.

[Kos89] Kosaraju, S.R., Efficient tree pattern matching. In: Proc. of the Symposium on Foundations of Computer Science (FOCS'89), p. 178–183.

[MaR90] Mannila, H., Räihä, K.-J., On query languages for the p-string data model. In: Information Modelling and Knowledge Bases, H. Kangassalo, S. Ohsuga, H. Jaakkola (eds.), IOS Press, 1990, p. 469–482.

[Rey77] Reyner, S.W., An analysis of a good algorithm for the subtree problem. SIAM J. Computing 6, 4 (December 1977), 730–732.

[RoS86] Robertson, N., Seymour, P.D., Graph Minors. II. Algorithmic aspects of tree-width. J. Algorithms 7 (1986), 309–322.

[ShZ90] Shasha, D., Zhang, K., Fast algorithms for the unit cost editing distance between trees. J. Algorithms 11 (1990), 581–621.

[Tai79] Tai, K.-C., The tree-to-tree correction problem. J. ACM 26, 3 (July 1979), 422–433.

[ZhS89] Zhang, K., Shasha, D., Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal of Computing 18, 6 (December 1989), 1245–1262.