A New Algorithm for the Ordered Tree Inclusion Problem*

Thorsten Richter
Department of Computer Science IV, University of Bonn
Roemerstr. 164, 53117 Bonn, Germany
e–mail: richter@cs.uni-bonn.de

May 31, 1997

Abstract

In the problem of *ordered tree inclusion* two ordered labeled trees P and T are given, and the *pattern tree* P matches the *target tree* T at a node x, if there exists a one-to-one map f from the nodes of P to the nodes of T which preserves the labels, the ancestor relation and the left-to-right ordering of the nodes. In [12] Kilpeläinen and Mannila give an algorithm that solves the problem of ordered tree inclusion in time and space $\Theta(|P| \cdot |T|)$. In this paper we present a new algorithm for the ordered tree inclusion problem with time complexity $O(|\Sigma_P| \cdot |T| + \#matches \cdot \text{DEPTH}(T))$, where Σ_P is the alphabet of the labels of the pattern tree and $\#matches$ is the number of pairs $(v, w) \in P \times T$ with $\text{LABEL}(v) = \text{LABEL}(w)$. The space complexity of our algorithm is $O(|\Sigma_P| \cdot |T| + \#matches)$.

1 Introduction and Motivation

The problem of *ordered tree inclusion* [11] can be seen as an extension of the classic problem of *tree pattern matching* [6], [14], [2]. In the latter problem two ordered labeled trees P and T are given, and the *pattern tree* P matches the *target tree* T at a node x, if there exists a one-to-one map f from the nodes of P to the nodes of T , such that

(1) the root of P maps to x,

(2) $\forall v \in P$: v and $f(v)$ have the same labels,

(3) $\forall v \in P$: if v is not a leaf, then the i-th child of v maps to the i-th child of $f(v)$.

In the problem of ordered tree inclusion the third condition is replaced by

(3') $\forall v_1, v_2 \in P$:

(a) v_1 is an ancestor of $v_2 \iff f(v_1)$ is an ancestor of $f(v_2)$,

(b) v_1 is to the left of $v_2 \iff f(v_1)$ is to the left $f(v_2)$.

*This work was supported by the DFG under grant Bl 320/2-1

This is obviously a relaxation. Another formulation of this problem is given by Knuth in [13], Exercise 2.3.2-22.

One motivation for considering the ordered tree inclusion problem and other tree inclusion problems comes from the concept of structured text databases. One can use context-free grammars (see [7], for example) to describe the structure of natural language sentences in terms of their parse trees (see [19] for example). In Figure 1 an example of a parse tree of the sentence *The professor feeds the well-behaved dog with fresh meat* is given. Hence a structured text database can be realized as a collection of parse trees (see [3], [9]). Then one can use tree inclusion as a means of retrieving information from documents stored in such a database [10]. Figure 2 gives an example of a query tree which can be used to retrieve information on what professors do with dogs. In reality the parse tree as well as the query tree could be augmented with some more linguistic information.

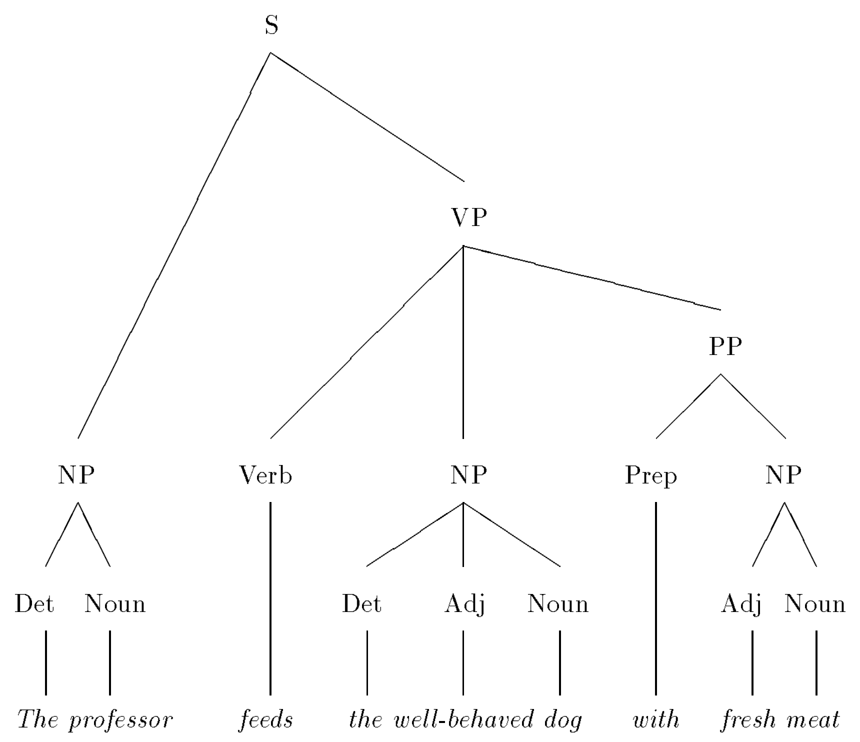

Figure 1: *An example of a parse tree.*

In [12] Kilpeläinen and Mannila give an algorithm that solves the problem of ordered tree inclusion in time and space $\Theta(|P| \cdot |T|)$. In that paper it is also shown that the tree inclusion problem becomes $\mathcal{NP}$-complete when considering unordered trees.

In this paper we present a new algorithm for the ordered tree inclusion problem with time complexity

$$O(|\Sigma_P| \cdot |T| + \#matches \cdot \text{DEPTH}(T)),$$

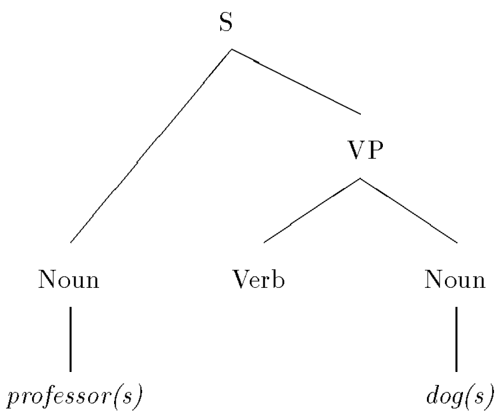

Figure 2: *An example of a query tree.*

where Σ_P is the alphabet of the labels of the pattern tree and *#matches* is the number of pairs $(v, w) \in P \times T$ with $\textsc{label}(v) = \textsc{label}(w)$. This complexity beats the complexity of the algorithm of [12] if the number of matches is relatively small, i. e. $\#matches = o(|P| \cdot |T| \,/\, \textsc{depth}(T))$. Furthermore, the time bound of our algorithm is not a tight one, but an upper bound. The space complexity of our algorithm is $O(|\Sigma_P| \cdot |T| + \#matches)$.

The main idea of our algorithm is to construct an inclusion map f by considering and mapping the nodes of P in ascending preorder, either until the pattern tree is completely mapped, or until it arrives at a point where it is impossible to continue with the construction of the inclusion map. In the latter case the algorithm returns to a node of P that has already been mapped, and maps it to another candidate. To avoid duplicate work, it derives as much information as possible from such a "dead end". Then it considers and maps the remaining nodes of P again in ascending preorder. Hence our algorithm uses some kind of *backtracking.*

This paper is organized as follows. In Section 2 we define and illustrate the problem of ordered tree inclusion. In the following section we relate the problem of ordered tree inclusion to other problems on trees and pattern matching. In Chaper 4 we present our new algorithm to solve the ordered tree inclusion problem. In this section we first introduce the main concepts used by the algorithm and give a survey of the algorithm. Then we describe the parts of the algorithm in detail. Finally we illustrate the interaction of the main parts of the algorithm. In the following section we sketch an implementation of our algorithm. In Section 6 we prove the correctness of the algorithm and analyze its complexity. In Section 7 we sketch how our algorithm can be used to enumerate *all* inclusion maps from the pattern tree to the target tree. Finally in Section 8 we discuss our algorithm and give some suggestions for further work.

2 The Ordered Tree Inclusion Problem

Let $T = (V, E)$ be an ordered labeled tree. Then we use for a node $u \in V$ the following notations:

- LABEL(u) is the label of u. We assume that the labels of the pattern and target tree are chosen from a finite alphabet Σ;
- $T[u]$ denotes the subtree of T with root u;
- RIGHTMOST_LEAF(u) is the rightmost leaf of the subtree $T[u]$;
- PARENT(u) denotes the parent of u;
- LEFT_SIBLING(u) and RIGHT_SIBLING(u) denote the left and the right sibling of u, respectively;
- LEFTMOST_CHILD(u) and RIGHTMOST_CHILD(u) are the leftmost and the rightmost children of u, respectively.

When we use in the following the terms *ancestor* and *successor*, we mean *proper* ancestors and proper successors. Analogously, we mean by *to the left of* and *to the right of* always *properly* left and *properly* right.

Now the problem of ordered tree inclusion can be formally defined as follows.

Definition 1 (Ordered Tree Inclusion) *In the* problem of ordered tree inclusion *there are given two ordered labeled trees P and T with $|P| \leq |T|$. P is the* pattern tree *and T is the* target tree. *Sought is a one-to-one map f from the nodes of P to the nodes of T, such that $\forall v, v_1, v_2 \in P$ the following conditions hold*

- label condition: LABEL$(v) =$ LABEL$(f(v))$*;*
- ancestor condition: v_1 *is ancestor of* $v_2 \iff f(v_1)$ *is ancestor of* $f(v_2)$*;*
- order condition: v_1 *is to the left of* $v_2 \iff f(v_1)$ *is to the left of* $f(v_2)$*.*

We call such a map f an inclusion map *from P to T.* □

Note that there may be *exponentially* many inclusion maps from a pattern tree to a target tree. Thus it is not feasible to look for *all* inclusion maps. Hence we look for only *one* inclusion map in the following.

In the remainder of this paper $P = (V_P, E_P)$, $V_P = \{v_1, \ldots, v_n\}$, is the given pattern tree and $T = (V_T, E_T)$, $V_T = \{w_1, \ldots w_m\}$, is the given target tree. $V_P[i]$ denotes the set of the first i nodes of the pattern tree in preorder. A pair $(v, w) \in V_P \times V_T$ with LABEL$(v) =$ LABEL(w) is called a *match*; then w is called a *candidate* for v.

The NEXT **array.** If $\Sigma_P = \{s_1, \ldots, s_t\}$ is the alphabet of the labels of the nodes of the pattern tree, then we denote, for all $1 \leq i \leq t$ and $1 \leq j \leq m$, by NEXT(s_i, w_j) the first node of T with label s_i and a preorder number greater than j, if there is one. If there is none, the value of NEXT(s_i, w_j) is NIL, i. e. it is undefined.

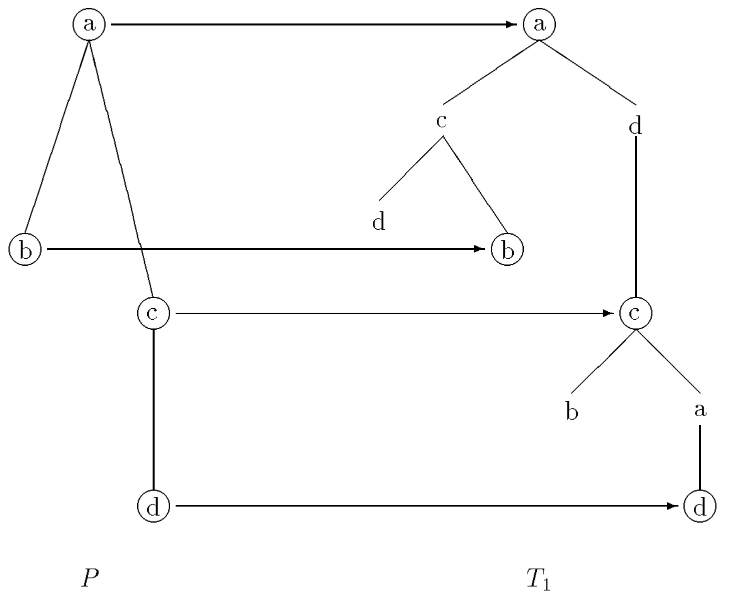

Figure 3: *An inclusion map from the nodes of P to the nodes of T_1 satisfying the conditions of ordered tree inclusion.*

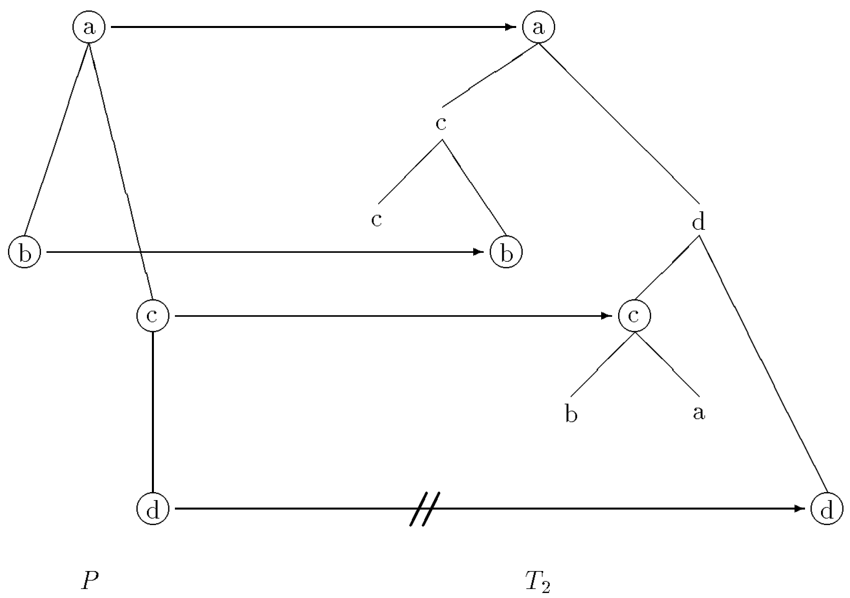

Figure 4: *The leaf of P with label* d *cannot be mapped to the leaf of T_2 with label* d.

Example 1 *In Figure 3 an inclusion map from the nodes of a pattern tree P to the nodes of a target tree T_1 that satisfies the conditions of ordered tree inclusion is shown.*

In Figure 4 the mapping of the leaf of P with label d to the leaf T_2 with label d would violate the ancestor condition of ordered tree inclusion. The node of P with label c is an ancestor of the leaf of P with label d, but the image of the node of P with label c is to the left of the leaf of T_2 with label d. Furthermore, the leaf of P with label d cannot be mapped to the inner node of T_2 with label d, since this would switch the ancestor relation.

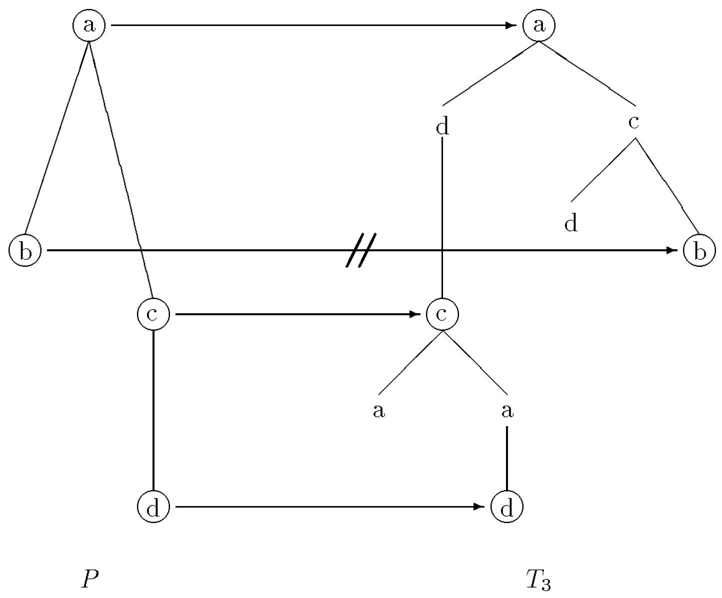

Figure 5: *The leaf of P with label* b *cannot be mapped to the leaf of T_3 with label* b*.*

In Figure 5 the mapping of the leaf of P with label b to the leaf T_3 with label b would violate the order condition of ordered tree inclusion. The leaf of P is to the left of the inner node of P with label c, but the leaf of T_3 is to the left of the image of the inner node of P. □

3 Related Problems

The objective of this section is to relate the problem of ordered tree inclusion to other problems on trees and on pattern matching.

3.1 Other Tree Inclusion Problems

The tree pattern matching and the ordered tree inclusion are only two representatives of the class of tree inclusion problems. We mention two inclusion problems that are "between" the previous ones.

In the *ordered path inclusion problem* the ancestor condition of ordered tree inclusion is replaced by the *child-of* condition: v_1 is child of $v_2 \iff f(v_1)$ is child of $f(v_2)$. The order condition remains. In the *ordered region inclusion problem* the order condition is tightened by the *sibling* condition: v_1 is a sibling of $v_2 \iff f(v_1)$ is a sibling of $f(v_2)$. Both problems can be solved in time $O(|P| \cdot |T|)$ [11].

Obviously we have: a solution of tree pattern matching is also a solution of ordered region inclusion; a solution of ordered region inclusion is also a solution of ordered path inclusion; and a solution of ordered path inclusion is also a solution of ordered tree inclusion. Note that the reverse does not hold. There may be a solution of ordered tree inclusion even if there is no solution of tree pattern matching. In [11] a uniform treatment of tree inclusion problems is given.

3.2 The Tree Editing Problem

The problem of ordered tree inclusion can also be seen as a special case of the *tree editing problem* [18], [20]. In the tree editing problem two ordered labeled trees T_1 and T_2 are given, and sought is a partial one-to-one map M (a *mapping*) from the nodes of T_1 to the nodes of T_2 with minimal cost, such that for all v_1, v_2 in the domain $\textsc{Domain}(M)$ of M the following hold:

(a) v_1 is an ancestor of $v_2 \iff M(v_1)$ is an ancestor of $M(v_2)$,

(b) v_1 is to the left of $v_2 \iff M(v_1)$ is to the left of $M(v_2)$.

The cost $\gamma(M)$ of a mapping M is defined as follows

$$\begin{aligned}\gamma(M) \quad = \quad & \sum_{v \in \mathrm{DOMAIN}(M)} \gamma(v, M(v)) \\ & + \sum_{v \in T_1 \setminus \mathrm{DOMAIN}(M)} \gamma(v, \textsc{nil}) \quad + \sum_{w \in T_2 \setminus \mathrm{RANGE}(M)} \gamma(\textsc{nil}, w)\end{aligned}$$

(by $(v, \textsc{nil})$ and $(\textsc{nil}, w)$ we denote that v maps to no node of T_2 and that no node of T_1 maps to w, respectively).

The ordered tree inclusion problem can be reduced to the tree editing problem as follows. If γ is for all $v \in T_1$ and $w \in T_2$ defined by

$$\gamma(v, \textsc{nil}) = \gamma(\textsc{nil}, w) = 1$$

and

$$\gamma(v, w) = \left\{ \begin{array}{l} 0, \text{ if } v \text{ and } w \text{ have the same label,} \\ 2, \text{ otherwise} \end{array} \right\},$$

a pattern tree P matches a target tree T at a node x in the sense of ordered tree inclusion, if and only if there is a mapping M from P to T, such that

(1) the root of P maps to x,

(2) $\gamma(M) = |T| - |P|$.

With the algorithm of Zhang and Shasha [20] for the tree editing problem we have an algorithm for ordered tree inclusion with time complexity

$$O(|P| \cdot |T| \cdot \min\{\textsc{depth}(P), \textsc{leaves}(P)\} \cdot \min\{\textsc{depth}(T), \textsc{leaves}(P)\}).$$

Note that this complexity is beaten by the specialized algorithm of Kilpeläinen and Mannila and by ours.

3.3 Subsequence and Substructure Problems

Turning to strings, the tree inclusion problem can be seen as a generalization of the *subsequence problem* for strings. In this problem one asks whether a pattern string P is a subsequence of a target string T. Obviously, there is a straightforward algorithm that solves this problem in linear time. We simply map the first symbol of the pattern string to the first occurence of this symbol in the target string. Then we map the second symbol of the pattern string to the first occurence of this symbol in the remainder of target string, and so on. Figure 6 gives an example of this approach.

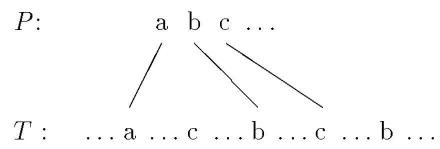

Figure 6: *An example of the subsequence problem.*

Hence a problem that is related to both the ordered tree inclusion problem and the tree editing problem is the problem of determining the *largest common substructure* of two trees. The latter problem is an extension of the well known *longest common subsequence* problem (see [4], [5], [1] and [16], for example) to trees. Here one seeks for the largest common substructure of two ordered labeled trees T_1 and T_2 that can be obtained by deleting nodes from T_1 and T_2 in the sense of the tree editing problem. This problem can be solved by an algorithm for the tree editing problem by a reduction similar to that for the ordered tree inclusion problem. Hence one can also compute the largest common substructure of two ordered labeled trees also in time

$$O(|P| \cdot |T| \cdot \min\{\text{DEPTH}(P), \text{LEAVES}(P)\} \cdot \min\{\text{DEPTH}(T), \text{LEAVES}(P)\}).$$

Up to now, no better specialized algorithm for this problem is known.

3.4 The Minor Containment Problem

The tree inclusion problem can be seen as a special case of the *minor containment problem* for graphs (see [17], [8]). In this problem two graph G and H are given, and the question is whether G contains H as a *minor*, i. e. whether G can be converted to H by a sequence of contractions of two adjacent nodes into a single new node.

This problem is known to be $\mathcal{NP}$-complete even when restricted to trees. This implies that minor containment is $\mathcal{NP}$-complete also for rooted trees. In other words, tree inclusion is $\mathcal{NP}$-complete for unordered tree. In [15] a proof for the $\mathcal{NP}$-completeness of the unordered tree inclusion is given which is independent of that in [12].

4 The New Algorithm for the Ordered Tree Inclusion

The **input** to our new algorithm *OrderedTreeInclusion* for the ordered tree inclusion problem consists of two ordered labeled trees, the pattern tree $P = (V_P, E_P)$ and the target tree $T = (V_T, E_T)$, where $V_P = \{v_1, \ldots, v_n\}$, $V_T = \{w_1, \ldots, w_m\}$, and $n \leq m$. In our description we assume that the subscripts of the nodes correspond to their preorder number.

We further assume without loss of generality that the roots of the pattern and target tree have the same label.

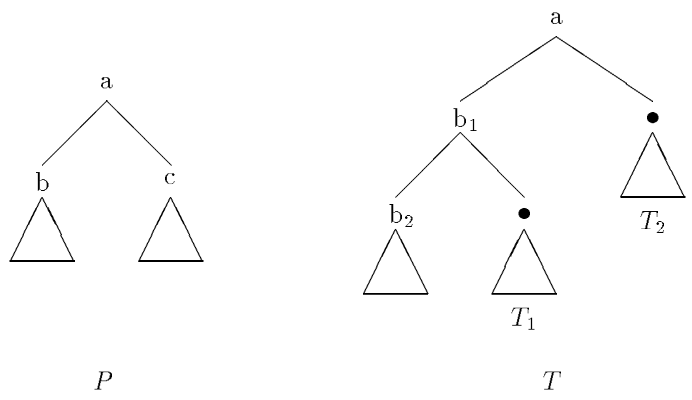

Figure 7: *An example of the ordered tree inclusion problem*

The idea of our algorithm is to transfer the straightforward approach for the subsequence problem (cf. Subsection 3.3) to the tree case. After some preprocessing our algorithm begins to construct an inclusion map f from the pattern tree P to the target tree T iteratively. Thereby it considers and maps the nodes of P in ascending preorder, and a node of the pattern tree is mapped to the in ascending preorder first eligible node of the target tree. This is, if a string is considered as a tree, our algorithm proceeds just as the algorithm for the subsequence problem does. But in contrast to the string case the algorithm can come to a "dead end" in the tree case, as illustrated by the following exapmle.

Example 2 *Consider the pattern tree P and the target tree T given in Figure 7. We start with mapping the root of P to the root T. Then we consider the left child of the root of the pattern tree with label* b*. The first eligible node of the target tree with the same label is the node labeled with* b_1*. Hence, one could map the left child of the root of P to this node of T. But in contrast to the string case this need not be a good choice. Even if the subtree $P[\mathrm{b}]$ can be completely mapped to the subtree $T[\mathrm{b}_1]$, it may happen that the subtree $P[\mathrm{c}]$ of the pattern tree cannot be completely mapped to the subtree T_2 of the target tree. In this case the algorithms eventually comes to a "dead end". Hence, the node* b_1 *is no suitable candidate for the left child of the root of the target tree.*

Note that a similar problem can occur when the algorithm considers the nodes a postorder. Assume that the left child of the root of the pattern tree is mapped to the node b_2 *of the target tree. In this case the subtree $P[\mathrm{c}]$ can be alternatively mapped to the subtree T_1 of the target tree. But now it may happen that the subtree $P[\mathrm{b}]$ can be completely mapped to the subtree $T[\mathrm{b}_1]$ but not to the subtree $T[\mathrm{b}_2]$. Hence the algorithm can come to a dead end in either traversal order.* □

The preceeding example has shown that our algorithm can arrive at dead end – at a point where it is impossible to continue with the construction of the inclusion map. In this case it returns to a node of P that has already been mapped and maps it to another candidate. Afterwards, it considers and maps the remaining nodes of P again in ascending preorder, either until the pattern tree is completely mapped or until it arrives at another dead end. This means that our algorithm uses some kind of *backtracking*.

If the algorithm has been successful in constructing an inclusion map from P to T, its **output** is the constructed inclusion map f. Otherwise, it returns a message that it is not possible to map the pattern tree completely to the target tree.

In the following subsections we first introduce the basic terms und facts for our algorithm; then we give a survey of how our algorithm proceeds. The third subsection describes the preprocessing part and the fourth subsection the main part of the algorithm, where the main procedures of the algorithm are explained in detail. The final subsection of this section is devoted to illustrating the correct interaction of these procedures.

4.1 Basics of the Algorithm

Suppose that we have already mapped the first i, $1 \leq i < |V_P|$, nodes of the pattern tree to nodes of the target tree according to the conditions of ordered tree inclusion.

Definition 2 (Partial inclusion map) *We call a map f from $V_P[i]$ to the nodes of T a* partial inclusion map for $V_P[i]$, *if f satisfies for all nodes of $V_P[i]$ the conditions of ordered tree inclusion.* □

If we want to extend a partial inclusion map for $V_P[i]$ to $V_P[i+1]$, we have to consider only candidates for v_{i+1} which are compatible with this partial inclusion map.

Definition 3 (Feasible candidate) *Let f be a partial inclusion map for $V_P[i]$. We call a candidate w for v_{i+1}* feasible with respect to f, *if the map f' from $V_P[i+1]$ to the nodes of T, defined by*

$$f'(u) = f(u), \ u \in V_P[i]$$

and

$$f'(v_{i+1}) = w,$$

is a partial inclusion map for $V_P[i+1]$.

In the following we often omit the specification with respect to f, *where it is clear which partial inclusion map is meant.* □

Hence to extend a partial inclusion map f for $V_P[i]$ to $V_P[i+1]$ it suffices to consider only feasible candidates for v_{i+1}. The next lemma specifies their position in the target tree (see Figure 8).

Lemma 1 *Let f be a partial inclusion map for $V_P[i]$. A candidate w for v_{i+1} is feasible with respect to f, if and only if*

(a) w is successor of $f(\text{PARENT}(v_{i+1}))$;

(b) w is to the right of $f(\text{LEFT\_SIBLING}(v_{i+1}))$, if v_{i+1} has a left sibling.

PROOF.

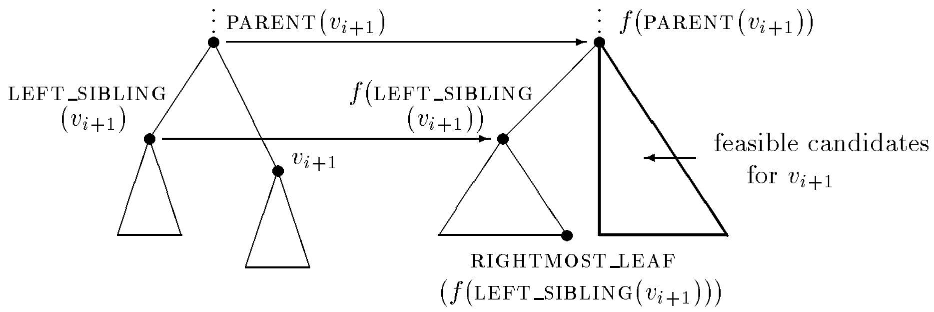

Figure 8: *The position of the feasible candidates for* v_{i+1}.

"$\Rightarrow$" Let w be a feasible candidate for v_{i+1}. Since v_{i+1} is successor of $\text{PARENT}(v_{i+1})$, w has to be successor of $f(\text{PARENT}(v_{i+1}))$ because of the ancestor condition. If v_{i+1} has a left sibling, it has to be to the right of $f(\text{LEFT\_SIBLING}(v_{i+1}))$ because of the order condition.

"$\Leftarrow$" Let w be successor of $f(\text{PARENT}(v_{i+1}))$, and let it be to the right of $f(\text{LEFT\_SIBLING}(v_{i+1}))$, if v_{i+1} has a left sibling.

First we have to show that for every node $u \in V_P[i]$ which is ancestor of v_{i+1}, $f(u)$ is ancestor of w: so let u be ancestor of v_{i+1}. Since w is successor of $f(\text{PARENT}(v_{i+1}))$, it is also successor of $f(u)$.

Then we have to show that for every node $u \in V_P[i]$ which is to the left of v_{i+1}, $f(u)$ is to the left of w. First let v_{i+1} have no left sibling, and let u be to the left of v_{i+1}. Since every node which is to the left of v_{i+1} is also to the left of $\text{PARENT}(v_{i+1})$, and since w is successor of $f(\text{PARENT}(v_{i+1}))$, $f(u)$ is to the left of w. Now let v_{i+1} have a left sibling, let u_1 be successor of $\text{LEFT\_SIBLING}(v_{i+1})$, and let u_2 be to the left of $\text{LEFT\_SIBLING}(v_{i+1})$. Since $f(\text{LEFT\_SIBLING}(v_{i+1}))$ is to the left of w, $f(u_1)$ and $f(u_2)$ are also to the left of w. □

This lemma implies that a feasible candidate for v_{i+1} must have a greater preorder number than $f(\text{PARENT}(v_{i+1}))$ or than $\text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{i+1})))$, if v_{i+1} has a left sibling. Together with the definition of the NEXT array, we have the following corollary.

Corollary 1 *Let f be a partial inclusion map for $V_P[i]$, and let there be a feasible candidate for v_{i+1}. Then we have*

(a) if v_{i+1} has no left sibling,

$$\text{NEXT}(\text{LABEL}(v_{i+1}), f(\text{PARENT}(v_{i+1})))$$

is the feasible candidate for v_{i+1} with the smallest preorder number;

(b) if v_{i+1} has a left sibling,

$$\text{NEXT}(\text{LABEL}(v_{i+1}), \text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{i+1}))))$$

is the feasible candidate for v_{i+1} with the smallest preorder number. □

From this result we can derive a criterion for the existence of a feasible candidate.

Corollary 2 *Let f be a partial inclusion map for $V_P[i]$. Then there is no feasible candidate for v_{i+1}, if and only if*

(a) v_{i+1} has no left sibling and
$$\text{NEXT}(\text{LABEL}(v_{i+1}), f(\text{PARENT}(v_{i+1})))$$
is not successor of $f(\text{PARENT}(v_{i+1}))$;

(b) v_{i+1} has a left sibling and
$$\text{NEXT}(\text{LABEL}(v_{i+1}), \text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{i+1}))))$$
is not successor of $f(\text{PARENT}(v_{i+1}))$. □

Hence to extend a partial inclusion map f for $V_P[i]$ to $V_P[i+1]$ we consider the node

$$w_k = \begin{cases} \text{NEXT}(\text{LABEL}(v_{i+1}), f(\text{PARENT}(v_{i+1}))), & \text{if } v_{i+1} \text{ has no left sibling;} \\ \text{NEXT}(\text{LABEL}(v_{i+1}), \text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{i+1})))), & \text{otherwise.} \end{cases}$$

If w_k is not successor of $f(\text{PARENT}(v_{i+1}))$, we know, by Corollary 2, that there is no feasible candidate for v_{i+1} at all. If w_k is successor of $f(\text{PARENT}(v_{i+1}))$, we can map v_{i+1} to it to get a partial inclusion map for $V_P[i+1]$. By Corollary 1 we know that we have not skipped any eligible candidate for v_{i+1}.

During the following construction of the inclusion map it may happen that there is no feasible candidate for a node v_j, $j > i+1$, if v_{i+1} is mapped to w_k. That is,the partial inclusion map constructed for $V_P[i+1]$ cannot be extended to an inclusion map for the whole pattern tree P: although w_k is a feasible candidate for v_{i+1}, it is not suited for the construction of an inclusion map for P.

Definition 4 (Suitable candidate) *Let f be a partial inclusion map for $V_P[i]$, and let w be a feasible candidate for v_{i+1}. We call w* not suitable with respect to f, *if there is a $j > i+1$ such that there is no partial inclusion map f' for $V_P[j]$ with*

$$f'(u) = f(u), u \in V_P[i],$$

and

$$f'(v_{i+1}) = w.$$

If there is no such j, then w is called suitable with respect to f. □

If it turns out that the candidate w_k to which v_{i+1} has been mapped is not suitable, this candidate is dismissed and another candidate for v_{i+1} is chosen. The following corollary generalizes the Corollaries 1 and 2 to this situation.

Corollary 3 *Let f be a partial inclusion map for $V_P[i]$, and let u be a node of T which is successor of $f(\text{PARENT}(v_{i+1}))$ and to the right of $f(\text{LEFT\_SIBLING}(v_{i+1}))$, if v_{i+1} has a left sibling. That is, if u has the label $\text{LABEL}(v_{i+1})$, then it is a feasible candidate for v_{i+1}. Let all feasible candidates for v_{i+1} whose preorder numbers are less than or equal to that of u be not suitable, as shown in Figure 9. Then we have*

(a) if there are suitable candidates for v_{i+1} whose preorder numbers are greater than those of u, then $\text{NEXT}(\text{LABEL}(v_{i+1}), u)$ is the one with the smallest preorder number;

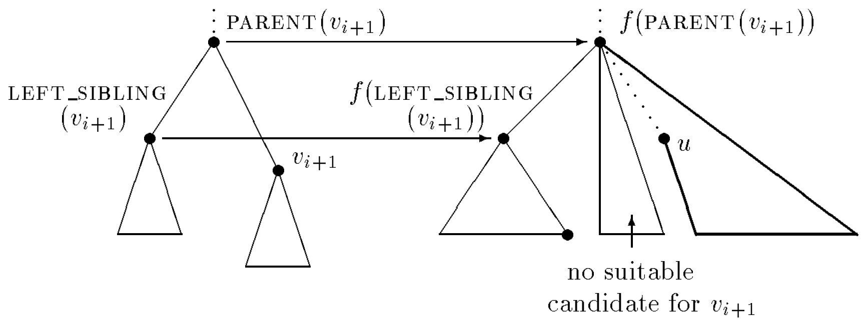

Figure 9: *The situation of Corollary 3.*

(b) *if* NEXT(LABEL$(v_{i+1}), u)$ *is not successor of* f(PARENT(v_{i+1})), *then there are no suitable candidates for* v_{i+1}. □

Hence, after dismissing the candidate w_k chosen first, we consider the candidate $\widetilde{w_k} =$ NEXT(LABEL$(v_{i+1}), u)$ for v_{i+1}. If it is feasible, we map v_{i+1} to it. Otherwise, we know that there is no suitable candidate for v_{i+1}. From the latter we can conclude that the candidate chosen for v_i is not suitable either.

It later turns out that we can conclude from a feasible candidate being not suitable that some following feasible candidates are also not suitable. Consequently, we can skip them. However, for the correctness of our algorithm it is necessary that we skip only feasible candidates which we know for certain are not suitable. If we have no such information, we have to go from one feasible candidate to the next.

Definition 5 (Next eligible candidate) *Let f a partial inclusion map for $V_P[i]$. We call a candidate w for v_{i+1}* the next eligible candidate *for v_{i+1}, if*

- *the preorder number of w is greater than that of f(PARENT(v_{i+1})) or than that of* RIGHTMOST_LEAF(f(LEFT_SIBLING(v_{i+1}))), *if v_{i+1} has a left sibling;*
- *all feasible candidates for v_{i+1} with a smaller preorder number than w are not suitable;*
- *whether w is suitable or not is still open.* □

Note that the next eligible candidate for v_{i+1} is feasible if and only if it is successor of f(PARENT(v_{i+1})). But if it is not feasible, we know by Corollary 3 that there is no feasible candidate for v_{i+1} at all.

Which candidate the next eligible candidate is depends on the "knowledge" of the algorithm, but is unequivocal at any time. Hence we can always choose the next eligible candidate for the node of the pattern tree under consideration.

4.2 Survey of the Algorithm

In every iteration of the algorithm we have the following **start state**: we have already constructed a partial inclusion map for $V_P[next - 1]$ and are now considering the node

v_{next}, for which we have chosen the next eligible candidate w_{next}. We then check whether w_{next} is successor of $f(\text{PARENT}(v_{next}))$, i. e. whether w_{next} is a feasible candidate for v_{next}. If this is the case, we can map v_{next} to w_{next}: we carry out a **forward step**. If not, then there is no suitable candidate for v_{next}, and hence we have come to a dead end: we carry out a **backward step**.

In a **forward step** we map v_{next} to w_{next}, i. e. we extend the partial inclusion map for $V_P[next-1]$ to $V_P[next]$. Then we choose the next eligible candidate for v_{next+1} and proceed with a new iteration of the algorithm. If the candidate chosen for v_{next+1} is feasible, we carry out another forward step in this iteration; otherwise we turn to a backward step.

If we enter a **backward step**, the current candidate w_{next} is not feasible. Hence we know that there is no suitable candidate for v_{next} at all. So we have come to a dead end in trying to map $P[\text{PARENT}(v_{next})]$ to $T[f(\text{PARENT}(v_{next}))]$.

If v_{next} has a left sibling, we know by Corollary 3 that there is no suitable candidate for v_{next} to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$. Hence $f(\text{LEFT\_SIBLING}(v_{next}))$ is not a suitable candidate for $\text{LEFT\_SIBLING}(v_{next})$. Consequently we dismiss this candidate, choose the next eligible candidate for $\text{LEFT\_SIBLING}(v_{next})$, and proceed with a new iteration of the algorithm. If the candidate chosen for $\text{LEFT\_SIBLING}(v_{next})$ is feasible, we turn to a forward step in this iteration. Otherwise there is no feasible candidate for $\text{LEFT\_SIBLING}(v_{next})$, such that we carry out another backward step, and so on.

If v_{next} has no left sibling, then there is no suitable candidate for v_{next} at all among the successors of $f(\text{PARENT}(v_{next}))$. This means that there is no inclusion map from the subtree $P[\text{PARENT}(v_{next})]$ to the subtree $T[f(\text{PARENT}(v_{next}))]$. Hence $f(\text{PARENT}(v_{next}))$ is not a suitable candidate for $\text{PARENT}(v_{next})$ and we dismiss it. Then we choose the next eligible candidate for $\text{PARENT}(v_{next})$ and proceed with a new iteration of the algorithm.

By mapping v_{next} to w_{next} in a forward step, we begin to construct an inclusion map from $P[v_{next}]$ to $T[w_{next}]$. This construction is called a **phase of the algorithm** for the match (v_{next}, w_{next}). At the end of this phase we have either constructed an inclusion map from $P[v_{next}]$ or $T[w_{next}]$ or ascertained that there is none.

The STATE array. To store the results of the phases, we use an array STATE, which is defined for the matches, i. e. for the pairs $(v, w) \in V_P \times V_T$ with $\text{LABEL}(v) = \text{LABEL}(w)$. Its fields can have three values:

$$\text{STATE}(v,w) = \left\{ \begin{array}{ll} \text{NIL}, & \text{if the match } (v,w) \text{ has not been considered yet;} \\ \text{TRUE}, & \text{if } P[v] \text{ can be completely mapped to } T[w]; \\ \text{FALSE}, & \text{if } P[v] \text{ cannot be completely mapped to } T[w] \end{array} \right\}.$$

Initially all fields have the value NIL. At the end of the phase for the match (v_{next}, w_{next}) we can set $\text{STATE}(v_{next}, w_{next})$ either to TRUE or to FALSE. A phase for the match (v_{next}, w_{next}) can include phases for matches (v_k, w_l) where v_k is successor of v_{next} and w_l is successor of w_{next}. But, as it does not overlap with other phases, we can consider this phase as a closed entity.

One consequence of the use of the STATE array is that we have to distinguish within the forward step three cases, which depend on the value of $\text{STATE}(v, w)$. If it has the value NIL, we consider the match (v, w) the first time. But if it has another value, we have already considered it previously. If it then has the value FALSE, we know that $P[v]$ cannot be completely mapped to $T[w]$. Hence w is not a suitable candidate for v, and we can dismiss it immediately by choosing another candidate for v. This avoids duplicate work. If $\text{STATE}(v, w)$ has the value TRUE, we already know that $P[v]$ can be completely mapped to $T[w]$. Hence we do not have to map $P[v]$ again to $T[w]$, but can implicitly make use of the inclusion map constructed before. Again, this avoids duplicate work.

The RANGE array. Within a phase for the match (v_{next}, w_{next}) we may discover that there is no candidate for a child v_k of v_{next} to the right of a node w_l of T whose choice would let us map $P[v_{next}]$ completely to $T[w_{next}]$. Hence there is no suitable candidate for v_k. This information restricts the mapping range of v_k within this phase.

To use this information, we store it in an array RANGE that is defined for the nodes of P. First, the entries of this array all have the value NIL, meaning that there is no restriction on the mapping range of the corresponding nodes (except the general condition for an inclusion map). If we get within a phase of the algorithm the information that there is no suitable candidate for the node v_k to the right of the node w_l, we set the value of $\text{RANGE}(v_k)$ to w_l. At the end of this phase, we reset all entries of the RANGE array for the children of v_{next} that have been set in this phase.

We take the RANGE array into account by checking in the start state of an iteration described above not ony if the current candidate w_{next} is feasible, but also if it is within the range specified by $\text{RANGE}(v_{next})$. Thus we can avoid unnecessary work.

In the description of our algorithm we assume, without loss of generality, that the root w_1 of the target tree T is a candidate for the root v_1 of the pattern tree P. Hence we can start the construction of the inclusion map by mapping v_1 to w_1 and choosing the next eligible candidate for v_2. Then we are in the start state of an iteration described above.

4.3 Preprocessing

In the preprocessing part of the algorithm we compute some information on the pattern and target tree that is used in the main part of the algorithm. In our description we assume that for every node u of P and T, respectively, the values of $\text{PARENT}(u)$, $\text{LEFT\_SIBLING}(u)$, $\text{RIGHT\_SIBLING}(u)$ and $\text{RIGHTMOST\_CHILD}(u)$ are already known. If the trees are given by adjacency lists, for example, these values can obviously be computed in linear time.

We start preprocessing by traversing both trees in preorder, storing the preorder numbers of the nodes of the pattern and target tree in the arrays PRE_P and PRE_T, respectively. The target tree is also traversed in postorder in order to store in the array POST_T the postorder numbers of its nodes. This can be used to determine the relative position of two nodes in the tree.

Fact 1 *Let v and w be two nodes of a tree. Then*

(1) w is successor of v $\iff \text{PRE}(w) > \text{PRE}(v) \wedge \text{POST}(w) < \text{POST}(v)$;

(2) w is to the right of v $\iff \text{PRE}(w) > \text{PRE}(v) \wedge \text{POST}(w) > \text{POST}(v)$.

PROOF.

(1) "$\Rightarrow$" If w is successor of v, its preorder number is greater than that of v, and its postorder number is smaller than that of v.

"$\Leftarrow$" Since $\textsc{pre}(w) > \textsc{pre}(v)$, w is either successor of or to the right of v. Since $\textsc{post}(w) < \textsc{post}(v)$, w is either to the left of or successor of v.

(2) Analogously. □

For each node w of the target tree T we compute the value of $\textsc{rightmost\_leaf}(w)$. Since we have

$$\textsc{rightmost\_leaf}(w) = \left\{ \begin{array}{ll} w, & \text{if } w \text{ is a leaf;} \\ \textsc{rightmost\_leaf}(\textsc{rightmost\_child}(w)), & \text{otherwise} \end{array} \right\},$$

we can compute these values by traversing T in descending preorder.

To implement the $\textsc{state}$ array efficiently, we proceed in the preprocessing as follows.

First we traverse the pattern tree P in preorder, where we store for every node v with label s its preoder number in a list $\textsc{occurences}_P(s)$. Then we traverse the target tree T in preorder to build analogous lists $\textsc{occurences}_T$. Here we maintain for every label $s \in \Sigma_P$ a counter $\textsc{occurence\_number}(s)$, which is initially zero for every $s \in \Sigma_P$. If we visit a node w with label s during the traversal of T, we increase $\textsc{occurence\_number}(s)$ by one, store its new value in $\textsc{label\_number}(w)$, and add it to the list $\textsc{occurences}_T(s)$.

After the traversal of T we can define for every node $v \in P$ an array

$$\textsc{state}(v)()$$

with exactly $\textsc{occurence\_number}(\textsc{label}(v))$ entries, which are initialized with the value $\textsc{nil}$. During the main part of the algorithm we can access the $\textsc{state}$ value of a match (v, w) via

$$\textsc{state}(v, \textsc{label\_number}(w)).$$

Since a traversal of a tree can be done in linear time, the initialization of the $\textsc{state}$ array takes the time $O(\#matches)$.

Furthermore, we compute the values of the array $\textsc{next}$ during preprocessing. Since we have for all $s \in \Sigma_P$

$$\textsc{next}(s, w_m) = \textsc{nil}$$

and for all $1 \leq j < m$

$$\textsc{next}(s, w_j) = \left\{ \begin{array}{ll} w_{j+1}, & \text{if } \textsc{label}(w_{j+1}) = s; \\ \textsc{next}(s, w_{j+1}), & \text{otherwise} \end{array} \right\},$$

this can be done in time $O(|\Sigma_P| \cdot |T|)$ by traversing the nodes of T again in descending preorder.

Altogether, the preprocessing part of the algorithm has a time complexity of $O(|\Sigma_P| \cdot |T| + \#matches)$.

4.4 Main Part of the Algorithm

Since we have assumed that the root of the target tree is a candidate for the root of the pattern tree, i. e. $\textsc{label}(v_1) = \textsc{label}(w_1)$, we can start the construction of the inclusion map f by mapping v_1 to w_1. Next we consider the node v_2 of P. The next eligible candidate for v_2 is the node with the smallest preorder number that has the same label as v_2, but is not the root of T. This node is obviously successor of $f(\text{PARENT}(v_2)) = f(v_1)$. Hence we can set

$$\begin{aligned} v_{next} &:= v_2 \\ w_{next} &:= \text{NEXT}(\text{LABEL}(v_2), w_1) \end{aligned}$$

and proceed with the first iteration of the algorithm.

4.4.1 The Start State of an Iteration of the Algorithm

In the **start state** of an iteration of the algorithm we have already constructed a partial inclusion map for $V_P[next - 1]$, and are now considering the node v_{next}, for which we have chosen the next eligible candidate w_{next}. Then we check whether w_{next} is successor of $f(\text{PARENT}(v_{next}))$, and whether it is within the range specified by $\text{RANGE}(v_{next})$. If it is not successor of $f(\text{PARENT}(v_{next}))$, it is not feasible. Hence by Corollaries 2 and 3, respectively we have that there is no feasible candidate at all. If w_{next} is out of the range specified by $\text{RANGE}(v_{next})$, it is not suitable. In both cases we proceed with a backward step; otherwise we proceed with a forward step.

4.4.2 The Forward Step

When we carry out a **forward step**, w_{next} is successor of $f(\text{PARENT}(v_{next}))$, hence a feasible candidate for v_{next}, and it is within the range specified by $\text{RANGE}(v_{next})$. This means that we can map v_{next} to w_{next}. Then we have to distinguish the following cases depending on the value of $\text{STATE}(v_{next}, w_{next})$.

(1) If $\text{STATE}(v_{next}, w_{next}) = \text{NIL}$, we have not tried to map $P[v_{next}]$ to $T[w_{next}]$ yet. Consequently we map v_{next} to w_{next} now:

$$f(v_{next}) := w_{next}.$$

In choosing the next eligible candidate for v_{next+1}, we have to distinguish the following cases.

(1.a) If v_{next} *has a child,* the next node v_{next+1} is the leftmost child of v_{next}, as shown in Figure 10. Then the next eligible candidate for v_{next+1} must only have a greater preorder number than $f(v_{next})$ (cf. Corollary 1.(a)). Hence, we set

$$\begin{aligned} w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next+1}), f(v_{next})), \\ v_{next} &:= v_{next+1} \end{aligned}$$

in this case.

Remark 1 (Subtree heuristics) *Before beginning to construct an inclusion map from $P[v_{next}]$ to $T[w_{next}]$ we could use some heuristics to check if this is possible at all. To map $P[v_{next}]$ completely to $T[w_{next}]$, $T[w_{next}]$ has to be at least as large and has high as $P[v_{next}]$, and it must have at least as many leaves as $P[v_{next}]$. If one of these conditions is not fulfilled, $P[v_{next}]$ cannot be completely mapped to $T[w_{next}]$. Hence we could set the value of* $\text{STATE}(v_{next}, w_{next})$ *to* FALSE *without actually trying to construct an inclusion map.*

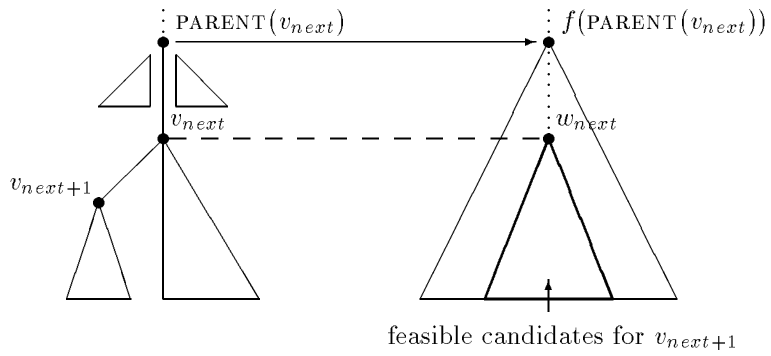

Figure 10: *Forward step, Case (1.a), v_{next} has a child.*

The required size, height and number of leaves of the subtrees could be computed during preprocessing by traversing the trees in descending preorder in time $O(|P| + |T|)$.

Furthermore, we could compile information on the occurrences of labels during preprocessing. Then we could check, for example, if all the labels that occur in $P[v_{next}]$ also occur in $T[w_{next}]$. If not, $P[v_{next}]$ cannot be completely mapped to $T[w_{next}]$. Finally, we could even pay attention to the number of occurrences of the labels in the subtrees of P and T. $\square$

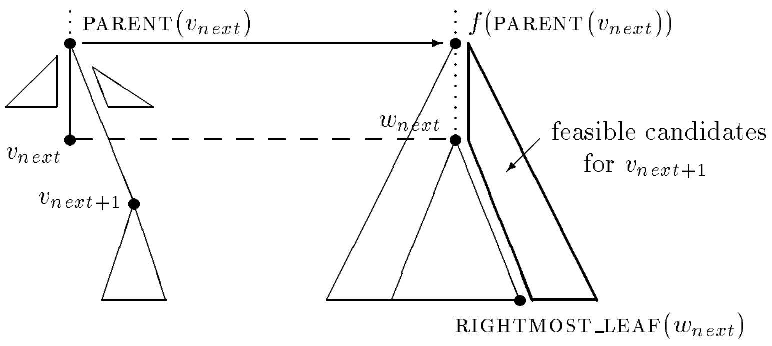

Figure 11: *Forward step, Case (1.b), v_{next} has no child, but a right sibling.*

(1.b) If v_{next} *has no child, but a right sibling*, then v_{next} is a leaf, as shown in Figure 11. Hence we can set

$$\text{STATE}(v_{next}, w_{next}) := \text{TRUE}.$$

The next node v_{next+1} is the right sibling of v_{next}, and the next eligible candidate for v_{next+1} has to be to the right of $f(v_{next})$ (cf. Corollary 1.(b)). Consequently, we set

$$\begin{aligned} w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next+1}), \text{RIGHTMOST\_LEAF}(f(v_{next}))), \\ v_{next} &:= v_{next+1} \end{aligned}$$

in this case.

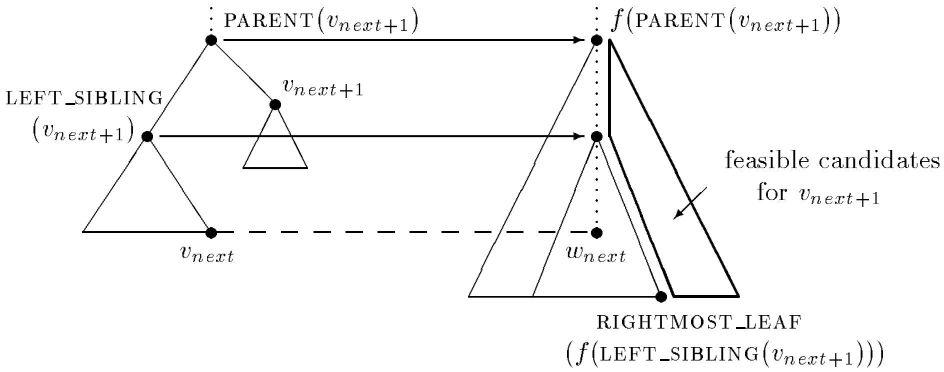

Figure 12: *Forward step, Case (1.c), v_{next} is a rightmost leaf.*

(1.c) If v_{next} *is a rightmost leaf*, we have mapped a subtree of P completely to a subtree of T, as shown in Figure 12. To store this, we go up the path from v_{next} to its first ancestor with a right sibling. Thereby we set for every node v on this path

$$\text{STATE}(v, f(v)) := \text{TRUE},$$

and, for each of its children u the RANGE-value of which has been set, we reset the value of RANGE(u) to NIL.

If v_{next} has no ancestor with a right sibling, v_{next} is the rightmost leaf of the whole pattern tree. This means that we have mapped the pattern tree completely to the target tree. Hence we can finish the algorithm.

If v_{next} is not the rightmost leaf of the pattern tree, the next node v_{next+1} is the right sibling of the first ancestor of v_{next} with a right sibling, and the next eligible candidate for v_{next} has to be to the right of RIGHTMOST_LEAF(f(LEFT_SIBLING(v_{next+1}))) (cf. Corollary 1.(b)). Consequently, we set

$$\begin{aligned} w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next+1}), \text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{next+1})))), \\ v_{next} &:= v_{next+1} \end{aligned}$$

in this case.

(2) If STATE(v_{next}, w_{next}) = FALSE, we already know that $P[v_{next}]$ cannot be completely mapped to $T[w_{next}]$. Hence w_{next} is not a suitable candidate for v_{next}, and we do not map v_{next} to w_{next}, but immediately choose a new candidate for v_{next}. Since successors of w_{next} with label LABEL(v_{next}) are also not suitable candidates for v_{next}, the next eligible candidate for v_{next} has to be to the right of w_{next} (see Figure 13). Hence we set

$$w_{next} := \text{NEXT}(\text{LABEL}(v_{next}), \text{RIGHTMOST\_LEAF}(w_{next}))$$

in this case (cf. Corollary 3.(a)).

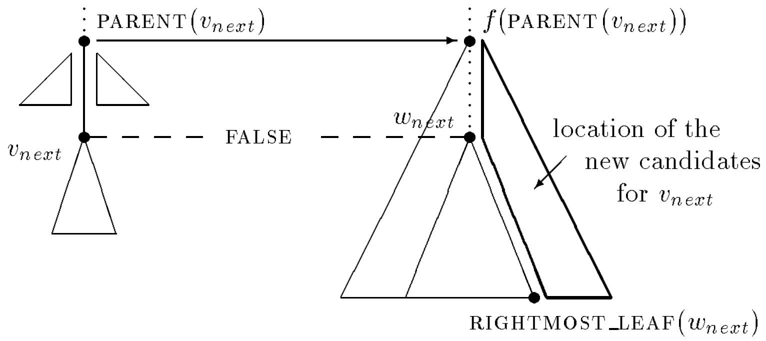

Figure 13: *Forward step, Case (2),* STATE(v_{next}, w_{next}) = FALSE.

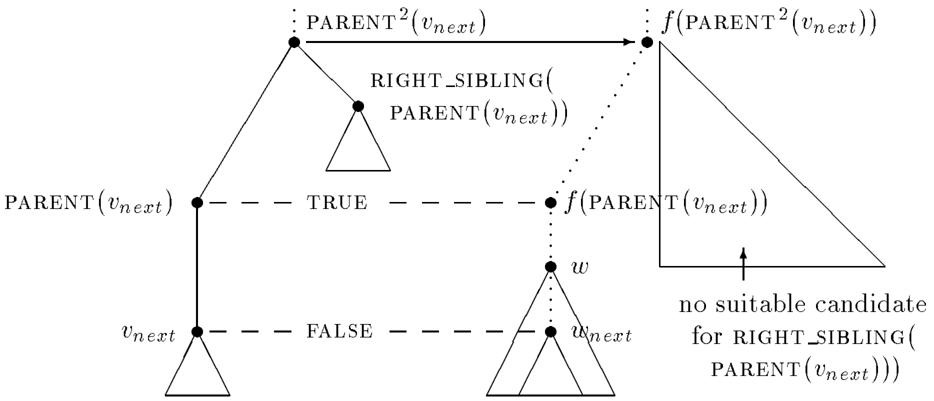

Figure 14: *A match* (v_{next}, w_{next}) *with* STATE(v_{next}, w_{next}) = FALSE *is considered again.*

Remark 2 *Note that this case can actually occur. In the situation shown in Figure 14,* $\text{STATE}(v_{next}, w_{next})$ *has been set to* FALSE. *Nevertheless, it has been possible to map* $P[\text{PARENT}(v_{next})]$ *completely to* $T[f(\text{PARENT}(v_{next}))]$ *afterwards with another mapping of* v_{next}. *Hence* $\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ *has been set to* TRUE. *But now it is impossible to map* $\text{RIGHT\_SIBLING}(\text{PARENT}(v_{next}))$. *Hence the algorithm dismisses the match* $(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ *and seeks for another candidate* w *for* $\text{PARENT}(v_{next})$ *among the successors of* $f(\text{PARENT}(v_{next}))$ *(cf. Case (1) of the backward step). If it finds one, and if this is also ancestor of* w_{next}, *it may happen that the algorithm considers the match* (v_{next}, w_{next}) *again.* □

(3) If $\text{STATE}(v_{next}, w_{next}) = \text{TRUE}$, we have already mapped $P[v_{next}]$ completely to $T[w_{next}]$. Nevertheless, we again map v_{next} to w_{next}:

$$f(v_{next}) := w_{next}.$$

But we do not have to map the entire subtree $T[v_{next}]$ again; instead we can implicitly make use of the inclusion map from $P[v_{next}]$ to $T[w_{next}]$ constructed before. Consequently the constructed inclusion map f from P to T may be incomplete. Hence we have to reconstruct the missing parts after (successfully) finishing the algorithm. Nevertheless, we can immediately go to the next node of P that is not successor of v_{next}. Thereby we have to distinguish two cases depending on its position.

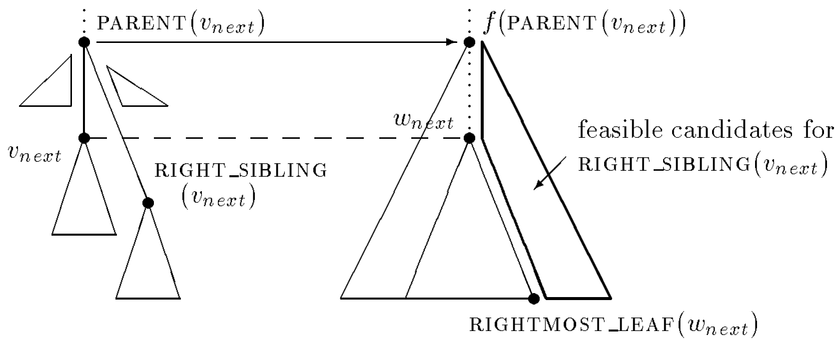

Figure 15: *Forward step, Case (3.a),* v_{next} *has a right sibling.*

(3.a) If v_{next} *has a right sibling*, as shown in Figure 15, we consider this right sibling next. The next eligible candidate for it must be to the right of w_{next} (cf. Corollary 1.(a)). Hence we set

$$\begin{aligned} w_{next} &:= \text{NEXT}(\text{LABEL}(\text{RIGHT\_SIBLING}(v_{next})), \text{RIGHTMOST\_LEAF}(w_{next})), \\ v_{next} &:= \text{RIGHT\_SIBLING}(v_{next}) \end{aligned}$$

in this case.

Remark 3 *Note that this situation can actually occur. In Figure 16* v_{next} *had been mapped first to* w_{old}, *but with this choice it was not possible to map* $\text{RIGHT\_SIBLING}(v_{next})$. *Subsequently another feasible candidate* w_{next} *for* v_{next} *was found in a backward step, so that* v_{next}

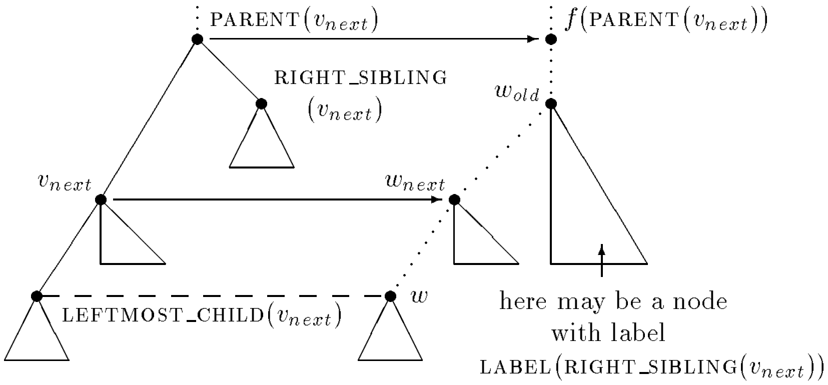

Figure 16: *The inclusion map of a subtree can be taken over completely.*

*was mapped to it in the following forward step. But the subtree $T[w]$, on which the subtree $P[$*LEFTMOST_CHILD*$(v_{next})]$ has been mapped previously, is not only subtree of $T[w_{old}]$, but also subtree of $T[w_{next}]$. Hence the inclusion map from $P[$*LEFTMOST_CHILD*$(v_{next})]$ to $T[w]$ can be taken over completely. However, note that there may be new possibilities to map* RIGHT_SIBLING(v_{next}) *through the change from w_{old} to w_{next}.* □

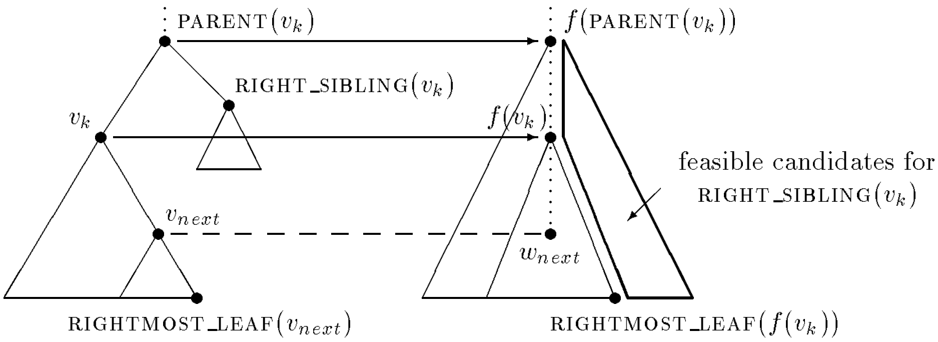

Figure 17: *Forward step, Case (3.b), v_{next} has no right sibling.*

(3.b) In the case when v_{next} *has no right sibling*, as shown in Figure 17, we have mapped a subtree of P completely to a subtree of T. In this case, the values of STATE of the nodes on the path from v_{next} to v_k, its first ancestor with a right sibling, have already been correctly set. Hence we consider the right sibling of v_k (note that this is the node which succeeds RIGHTMOST_LEAF(v_{next}) in preorder) next. The next eligible candidate for it has to be to the right of $f(v_k)$ (cf. Corollary 1.(b)). Hence, we set

$$\begin{aligned} v_{next} &:= v_{\text{PRE}(\text{RIGHTMOST\_LEAF}(v_{next}))+1}, \\ w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next}), \text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{next})))) \end{aligned}$$

in this case.

In every case of the forward step, we go with the new match (v_{next}, w_{next}) to a new iteration of the algorithm. Note that the candidate w_{next} chosen in the forward step is always the next eligible candidate for v_{next}.

4.4.3 The Backward Step

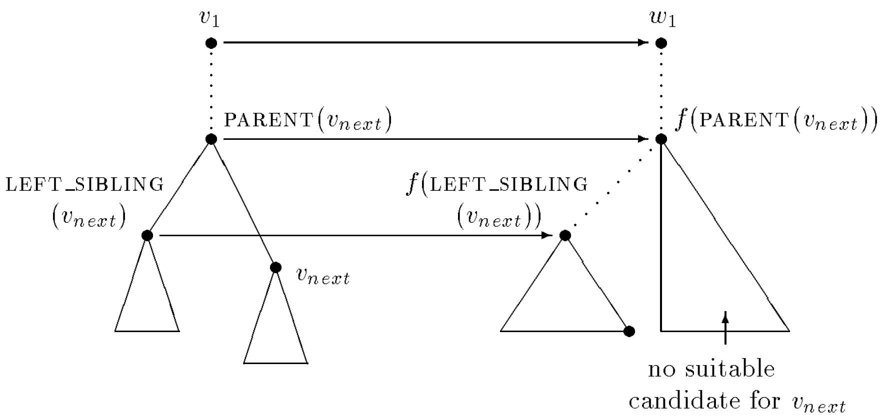

Figure 18: *The start situation of a backward step.*

When we carry out a **backward step**, w_{next} is either not successor of $f(\text{PARENT}(v_{next}))$ or it is out of the range specified by $\text{RANGE}(v_{next})$. In the first case w_{next} is not feasible and in the second case it is not suitable, as shown in Figure 18. In either case we know that there is no suitable candidate for v_{next} , so that $f(\text{PARENT}(v_{next}))$ or $f(\text{LEFT\_SIBLING}(v_{next}))$, if v_{next} has a left sibling, are not suitable either. Depending on whether v_{next} has a left sibling, we have to distinguish two cases in the backward step.

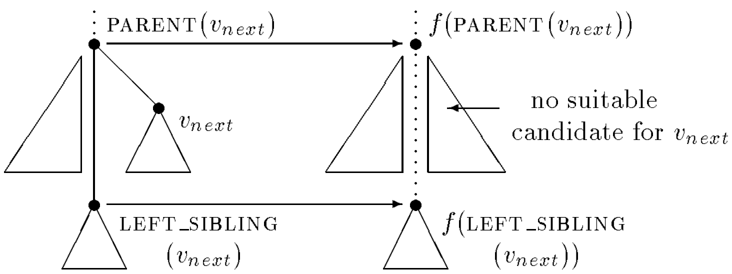

Figure 19: *Backward step, Case (1), v_{next} has a left sibling.*

(1) If v_{next} *has a left sibling*, as shown in Figure 19, there is no candidate for v_{next} to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$, the choice of which enables us to map $P[\text{PARENT}(v_{next})]$

completely to $T[f(\text{PARENT}(v_{next}))]$.

Lemma 2 *Let w_{next} be no suitable candidate for v_{next} and let v_{next} have a left sibling. Then there is no suitable candidate for v_{next} to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$.*
PROOF. We inductively distinguish two cases.

- If we have entered the backward step, because w_{next} is not successor of $f(\text{PARENT}(v_{next}))$, then there is no suitable candidate for v_{next} to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$, because w_{next} is the next eligible candidate for v_{next}.
- If we have entered the backward step, because w_{next} is to the right of $\text{RANGE}(v_{next})$, then there is no suitable candidate for v_{next} to the right of $\text{RANGE}(v_{next})$ due to the induction hypothesis. As w_{next} is the next eligible candidate for v_{next}, there is no suitable candidate for v_{next}, neither between $f(\text{LEFT\_SIBLING}(v_{next}))$ and $\text{RANGE}(v_{next})$ nor among the successors of $\text{RANGE}(v_{next})$. □

From this lemma we can conclude that $P[\text{PARENT}(v_{next})]$ cannot be completely mapped to $T[f(\text{PARENT}(v_{next}))]$, if $\text{LEFT\_SIBLING}(v_{next})$ is mapped to $f(\text{LEFT\_SIBLING}(v_{next}))$ or to a node that is to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$.

Corollary 4 *Let w_{next} be no suitable candidate for v_{next}, and let v_{next} have a left sibling. Then neither $f(\text{LEFT\_SIBLING}(v_{next}))$ nor a node that is to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$ is a suitable candidate for $\text{LEFT\_SIBLING}(v_{next})$.* □

Hence we have restricted the mapping range of v_{next} and $\text{LEFT\_SIBLING}(v_{next})$. For both nodes there is no suitable candidate to the right of $f(\text{LEFT\_SIBLING}(v_{next}))$. Consequently we set

$$\text{RANGE}(v_{next}) := f(\text{LEFT\_SIBLING}(v_{next}))$$

and

$$\text{RANGE}(\text{LEFT\_SIBLING}(v_{next})) := f(\text{LEFT\_SIBLING}(v_{next})).$$

Note that the value of $\text{RANGE}(\text{LEFT\_SIBLING}(v_{next}))$ does not exclude that $\text{LEFT\_SIBLING}(v_{next})$ is mapped to $f(\text{LEFT\_SIBLING}(v_{next}))$ again. But the following lemma shows that the algorithm does not do this again.

Lemma 3 *Let w_{next} be no suitable candidate for v_{next} and let v_{next} have a left sibling. Then the algorithm does not again try to map $P[\text{LEFT\_SIBLING}(v_{next})]$ to $T[f(\text{LEFT\_SIBLING}(v_{next}))]$ in the phase for the match $(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ after it has set $\text{RANGE}(\text{LEFT\_SIBLING}(v_{next}))$ to $f(\text{LEFT\_SIBLING}(v_{next}))$.*
PROOF. After the algorithm has set $\text{RANGE}(\text{LEFT\_SIBLING}(v_{next}))$ to $f(\text{LEFT\_SIBLING}(v_{next}))$, it chooses a new candidate w_k for $\text{LEFT\_SIBLING}(v_{next})$, the preorder number of which is greater than that of $f(\text{LEFT\_SIBLING}(v_{next}))$, i. e. w_k is either successor of $f(\text{LEFT\_SIBLING}(v_{next}))$ or it is to the right of it.

If the new candidate w_k is not successor of $f(\text{LEFT\_SIBLING}(v_{next}))$, the algorithm directly returns to the backward step. If $\text{LEFT\_SIBLING}(v_{next})$ has a left sibling, it sets there $\text{RANGE}(\text{LEFT\_SIBLING}(v_{next}))$ to $f(\text{LEFT\_SIBLING}^2(v_{next}))$. Since $f(\text{LEFT\_SIBLING}(v_{next}))$ is to the left of this new value, the algorithm does not again try to map $P[\text{LEFT\_SIBLING}(v_{next})]$ to $T[f(\text{LEFT\_SIBLING}(v_{next}))]$. If $\text{LEFT\_SIBLING}(v_{next})$ has no left sibling, then the algorithm finishes the phase for the match

$(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ immediately.

If the new candidate w_k is successor of $f(\text{LEFT\_SIBLING}(v_{next}))$, the algorithm begins with the construction of an inclusion map from $P[\text{LEFT\_SIBLING}(v_{next})]$ to $T[w_k]$. It returns afterwards to the node $\text{LEFT\_SIBLING}(v_{next})$ only if this new candidate w_k has been proven to be not suitable. There can be two reasons for this.

The first reason is that $P[\text{LEFT\_SIBLING}(v_{next})]$ cannot be completely mapped to $T[w_k]$. Then the algorithm again chooses a new candidate $\widetilde{w_k}$ whose preorder number is greater than that of w_k and hence greater than that of $f(\text{LEFT\_SIBLING}(v_{next}))$. That is, $\widetilde{w_k}$ is either successor if $f(\text{LEFT\_SIBLING}(v_{next}))$ or it is to the right of it.

The other reason is that it comes again to a dead end in mapping v_{next}. Then it sets the value of $\text{RANGE}(\text{LEFT\_SIBLING}(v_{next}))$ to w_k and chooses a new candidate $\widetilde{w_k}$ for $\text{LEFT\_SIBLING}(v_{next})$. This new candidate has a greater preorder number than w_k and hence than $f(\text{LEFT\_SIBLING}(v_{next}))$.

Now we have by induction that any new candidate chosen for $\text{LEFT\_SIBLING}(v_{next})$ in the remainder of the phase for the match $(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ has a greater preorder number than $f(\text{LEFT\_SIBLING}(v_{next}))$. □

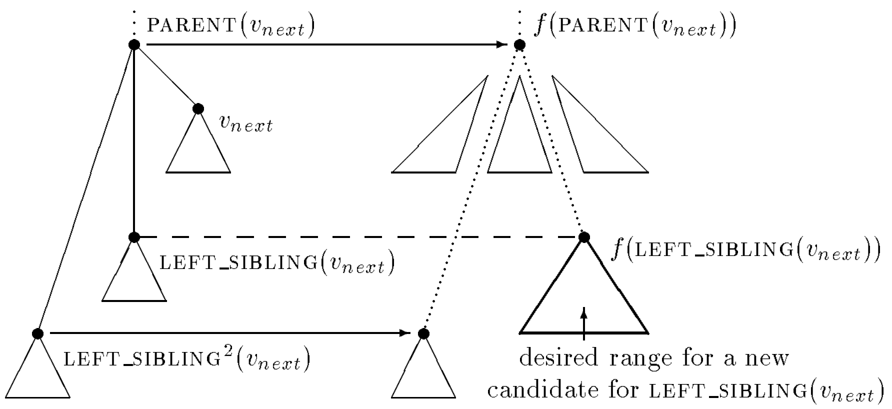

Figure 20: *The desired range for* $\text{LEFT\_SIBLING}(v_{next})$.

However,
to check if $P[\text{PARENT}(v_{next})]$ can be completely mapped to $T[f(\text{PARENT}(v_{next}))]$, i. e. if $f(\text{PARENT}(v_{next}))$ is a suitable candidate for $\text{PARENT}(v_{next})$, we must still consider other candidates for the left sibling of v_{next}. Thereby only those candidates are in question which are successors of its current image $f(\text{LEFT\_SIBLING}(v_{next}))$. Candidates which are to the left of this current image or ancestors of it are either not feasible or not suitable (note that we have always chosen the next elegible candidate). Furthermore, candidates which are to the right of this image are either not feasible or they are excluded by the corollary above (see Figure 20). Hence we set

$$\begin{aligned} v_{next} &:= \text{LEFT\_SIBLING}(v_{next}) \\ w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next}), f(v_{next})) \end{aligned}$$

in this case (cf. Corollary 3.(a)).

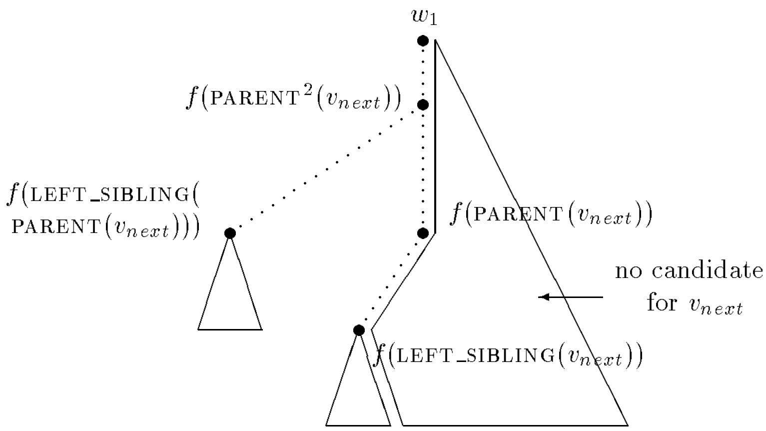

Figure 21: w_{next} is not defined.

Remark 4 (Global range values) *A special case occurs if w_{next} is not defined. Then there is no node with label* LABEL(v_{next}) *at all to the right of* $f(\text{LEFT\_SIBLING}(v_{next}))$*, as shown in Figure 21. This means that it does not make sense to look later for a new candidate for* PARENT(v_{next}) *to the right of* $f(\text{LEFT\_SIBLING}(v_{next}))$ *because there is no opportunity to map v_{next} there. This can be taken into account by maintaining* global *range values in an array* GLOBAL_RANGE *beside the local ones. In the situation described in the beginning one could set*

$$\text{GLOBAL\_RANGE}(\text{PARENT}(v_{next})) := f(\text{LEFT\_SIBLING}(v_{next})).$$

Contrary to the values of RANGE*, the values of* GLOBAL_RANGE *subsist beyond the current phase.*

If the phase for the match $\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ *is finished, the* GLOBAL_RANGE *information could be transferred to the parent of* PARENT(v_{next}) *by setting*

$$\text{GLOBAL\_RANGE}(\text{PARENT}^2(v_{next})) := f(\text{LEFT\_SIBLING}(v_{next})),$$

because if there is no suitable candidate for PARENT(v_{next}) *to the right of* $f(\text{LEFT\_SIBLING}(v_{next}))$*, then there is also no a suitable candidate for* PARENT$^2(v_{next})$ *there.*

If $\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})))$ *has been set to* FALSE*, the* GLOBAL_RANGE *information can then be used by not looking to the right of* $f(\text{PARENT}(v_{next}))$ *for a new candidate for* PARENT(v_{next})*, but by immediately choosing a new candidate for the left sibling of* PARENT(v_{next})*, and so on.* □

(2) If v_{next} *has no left sibling*, there is no suitable candidate for v_{next} in the whole subtree $T[f(\text{PARENT}(v_{next}))]$, as shown in Figure 22. Hence $P[\text{PARENT}(v_{next})]$ cannot be completely mapped to $T[f(\text{PARENT}(v_{next}))]$.

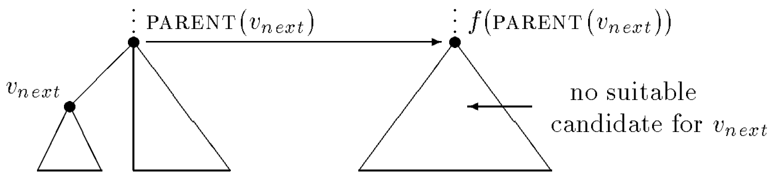

Figure 22: *Backward step, Case (2), v_{next} has no left sibling.*

Lemma 4 *Let w_{next} be no suitable candidate for v_{next}, and let v_{next} have no left sibling. Then $P[\text{PARENT}(v_{next})]$ cannot be completely mapped to $T[f(\text{PARENT}(v_{next}))]$.*
PROOF. We inductively distinguish two cases.

- If we have entered the backward step, because w_{next} is not successor of $f(\text{PARENT}(v_{next}))$, then there is no suitable candidate for v_{next} in the whole subtree $T[f(\text{PARENT}(v_{next}))]$, because w_{next} is the next eligible candidate for v_{next}.

- If we have enterd the backward step, because w_{next} is to the right of RANGE(v_{next}), then Corollary 4 implies that there is no suitable candidate for v_{next} to the right of RANGE(v_{next}), and RANGE(v_{next}) itself is also not a suitable candidate. Since w_{next} is the next eligible candidate for v_{next}, there is a suitable candidate for v_{next} neither to the left of RANGE(v_{next}) nor among the successors of RANGE(v_{next}). □

Therefore we set

$$\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next})) := \text{FALSE}.$$

If PARENT(v_{next}) is the root of P, we have found that the pattern tree P cannot be completely mapped to the target tree T. Hence we can finish the algorithm. Otherwise we reset the RANGE-values of the children of PARENT(v_{next}) which have been set to NIL. Then we go to the next eligible candidate for PARENT(v_{next}). Since $f(\text{PARENT}(v_{next}))$ and successors of it with label LABEL(PARENT(v_{next})) are not suitable candidates for PARENT(v_{next}), the next eligible candidate has to be to the right of $f(\text{PARENT}(v_{next}))$. Hence, we set

$$\begin{aligned} v_{next} &:= \text{PARENT}(v_{next}), \\ w_{next} &:= \text{NEXT}(\text{LABEL}(v_{next}), \text{RIGHTMOST\_LEAF}(f(v_{next}))) \end{aligned}$$

in this case.

When we leave the backward step, we have dismissed a mapping of a node of the pattern tree to a node of the target tree, and chosen a new candidate w_{next} for this node of P – either for PARENT(v_{next}) or for LEFT_SIBLING(v_{next}). Note further that the candidate w_{next} chosen in the backward step is always the next eligible candidate for v_{next}.

4.5 The Interaction of Forward and Backward Steps

In this subsection we illustrate the interaction of forward and backward steps. We start in Case (1) of the backward step: there is no suitable candidate for v_{next}, but it has a left sibling. This means that the node $w_{previous}$, to which $\text{LEFT\_SIBLING}(v_{next})$ is mapped currently, is not suitable either. Hence the algorithm sets

$$\text{RANGE}(v_{next}) := w_{previous}$$

and

$$\text{RANGE}(\text{LEFT\_SIBLING}(v_{next})) := w_{previous}.$$

Then it goes back to the left sibling of v_{next}

$$v_{next} := \text{LEFT\_SIBLING}(v_{next})$$

and chooses a new candidate for it:

$$w_{next} := \text{NEXT}(\text{LABEL}(v_{next}), w_{previous}).$$

Now we have to distinguish two cases depending on the position of w_{next}.

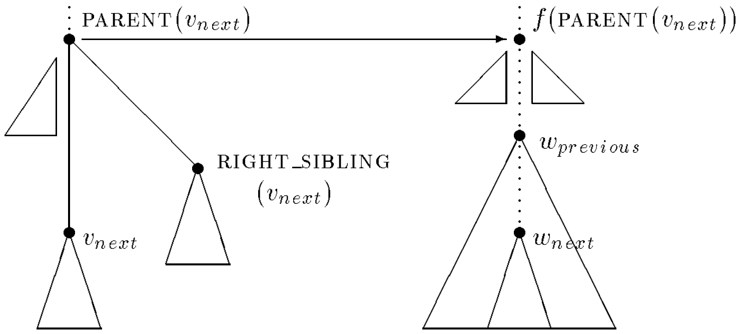

Figure 23: *Interaction, Case (1), w_{next} is successor $w_{previous}$.*

(1) If w_{next} *is successor of* $w_{previous}$, it is also successor of $f(\text{PARENT}(v_{next}))$ and within the range specified by $\text{RANGE}(v_{next})$. Hence it is within the desired mapping range for v_{next}, as shown in Figure 23, and may be a suitable candidate. Consequently a forward step is carried out in this iteration of the algorithm. This means that the algorithm begins with the construction of an inclusion map from $P[v_{next}]$ to $T[w_{next}]$. We distinguish two cases depending on the success of this construction.

(1.a) If $P[v_{next}]$ *can be completely mapped to* $T[w_{next}]$, the algorithm eventually sets

$$\text{STATE}(v_{next}, w_{next}) := \text{TRUE}$$

and proceeds with considering the right sibling of v_{next}. If the algorithm now finds a suitable candidate for $\text{RIGHT\_SIBLING}(v_{next})$, it proceeds with a forward step, and so on.

But if the algorithm does again not find a suitable candidate for $\text{RIGHT\_SIBLING}(v_{next})$, it proceeds with another backward step, where it sets

$$\text{RANGE}(\text{RIGHT\_SIBLING}(v_{next})) := w_{next}$$

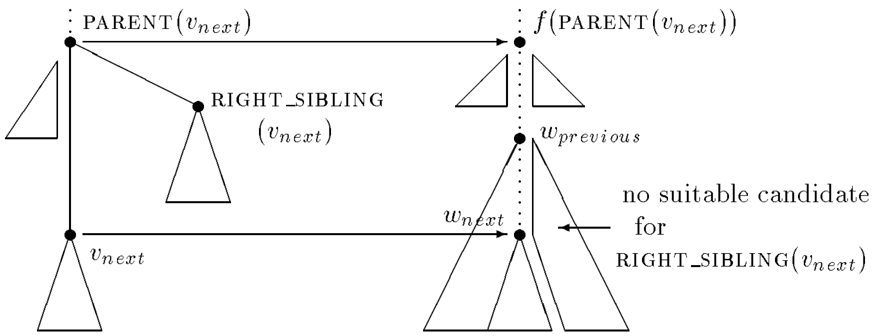

Figure 24: *Backward step, Case (1.a), w_{next} is successor of $w_{previous}$ and $P[v_{next}]$ can be completely mapped to $T[w_{next}]$, but there is no suitable candidate for* RIGHT_SIBLING(v_{next}).

and

$$\text{RANGE}(v_{next}) := w_{next}.$$

Then it chooses the next eligible candidate $\widetilde{w_{next}}$ for v_{next}:

$$\widetilde{w_{next}} := \text{NEXT}(\text{LABEL}(v_{next}), w_{next}).$$

Now the algorithm is in a situation that is similar to the situation we started with: it seeks for a new candidate for v_{next} among the successors of w_{next} (see Figure 24). Note that the new values of RANGE(RIGHT_SIBLING(v_{next})) and RANGE(v_{next}) are refinements of the old ones, hence the mapping range that was excluded previously remains excluded.

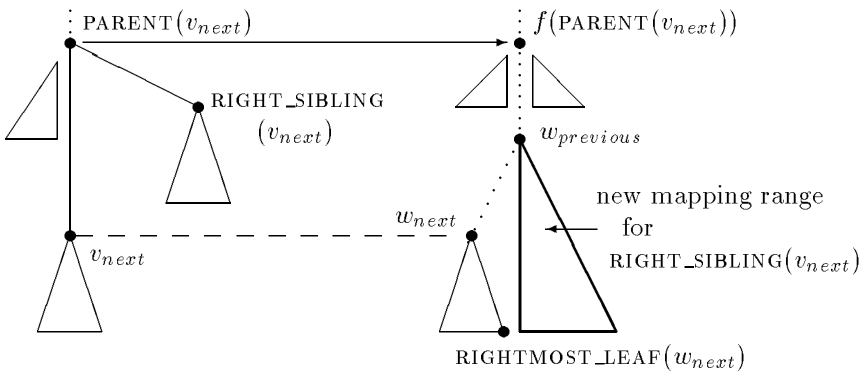

Figure 25: *The situation of the label heuristics*

Remark 5 (Label heuristics) *In the situation just described, we can avoid superfluous work by checking whether there is a node with label* LABEL(RIGHT_SIBLING(v_{next})) *in the* *new mapping range for* RIGHT_SIBLING(v_{next}) *at all, before we begin with the construction of an inclusion map from* $P[v_{next}]$ *to* $T[w_{next}]$ *(see Figure 25). The next eligible candidate for* RIGHT_SIBLING(v_{next}) *would have to be to the right of* RIGHTMOST_LEAF(w_{next}). *Hence we could check whether* NEXT(LABEL(RIGHT_SIBLING(v_{next})), RIGHTMOST_LEAF(w_{next})) *is successor of* $w_{previous}$. *If this is not the case, it would make no sense to map* $P[v_{next}]$ *to* $T[w_{next}]$*, but we could immediately go to the next eligible candidate* $\widetilde{w_{next}}$ *for* v_{next}.

However, this heuristics may result in additional work. For example, from the fact that $P[v_{next}]$ *cannot be completely mapped to* $T[w_{next}]$*, we can conclude that all successors of* w_{next} *with label* LABEL(v_{next}) *are not suitable candidates for* v_{next} *either (cf. Case (1.b)). But if we use the label heuristics, it may happen that the test of the heuristics is negative for* v_{next} *and all of its successors. This means that we have to consider all of these nodes before we find that* $P[v_{next}]$ *cannot be completely mapped to* $T[w_{next}]$.

A weaker form of this heuristics consists of checking whether there is a node with label LABEL(RIGHT_SIBLING(v_{next})) *among the successors of* $w_{previous}$ *at all, before we choose a new candidate for* v_{next} *in Case (1) of the backward step. To do so, we could check whether* NEXT(LABEL(RIGHT_SIBLING(v_{next})), $w_{previous}$) *is successor of* $w_{previous}$. *If this is not the case, then it is useless to look for a new candidate for* v_{next} *among the successors of* $w_{previous}$*, since there is no suitable one. Instead we could immediately go back to the left sibling of* v_{next}. □

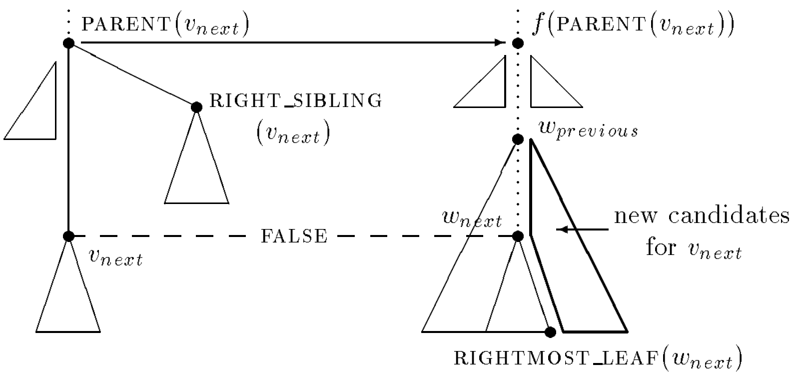

Figure 26: *Interaction, Case (1.b),* w_{next} *is successor of* $w_{previous}$*, but* $P[v_{next}]$ *cannot be completely mapped to* $T[w_{next}]$.

(1.b) If $P[v_{next}]$ *cannot be completely mapped to* $T[w_{next}]$, the algorithm eventually sets

$$\text{STATE}(v_{next}, w_{next}) := \text{FALSE},$$

and goes to the next eligible candidate $\widetilde{w_{next}}$ for v_{next}. Since this cannot be successor of w_{next}, the algorithm sets

$$\widetilde{w_{next}} := \text{NEXT}(\text{LABEL}(v_{next}), \text{RIGHTMOST\_LEAF}(w_{next}))$$

(cf. Case (2) of the backward step and Figure 26). Now we have to distinguish two cases depending on the position of $\widetilde{w_{next}}$.

(1.b.i) If the new candidate $\widetilde{w_{next}}$ *is to the right of* $w_{previous}$, then there is no node with label LABEL(v_{next}) to the right of RIGHTMOST_LEAF(w_{next}) which is successor of $w_{previous}$. Hence there is no suitable candidate for v_{next} among the successors of $w_{previous}$ at all. Now the algorithm is in a situation that is similar to that discussed below in Case (2): it carries out a backward step in which it first checks whether v_{next} has a left sibling or not.

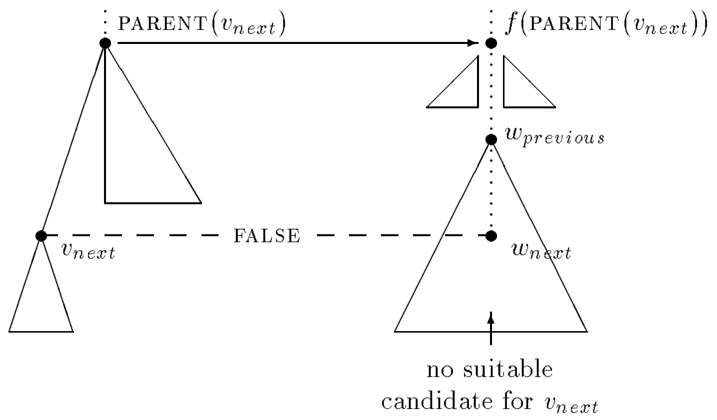

Figure 27: *Interaction, Case (1.b.i),* w_{next} *is successor of* $w_{previous}$*,* $P[v_{next}]$ *cannot be completely mapped to* $T[w_{next}]$*, the new candidate* $\widetilde{w_{next}}$ *is to the right of* $w_{previous}$*, and* v_{next} *has no left sibling.*

If v_{next} has no left sibling, P[PARENT(v_{next})] cannot be completely mapped to $T[f$(PARENT(v_{next}))] (see Figure 27). Hence the algorithm sets

$$\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next}))) := \text{FALSE},$$

and goes to the next eligible candidate for PARENT(v_{next}).

If v_{next} has a left sibling, the algorithm sets

$$\text{RANGE}(v_{next}) := f(\text{LEFT\_SIBLING}(v_{next}))$$

and

$$\text{RANGE}(\text{LEFT\_SIBLING}(v_{next})) := f(\text{LEFT\_SIBLING}(v_{next})).$$

Then it chooses the next eligible candidate for LEFT_SIBLING(v_{next}) (see Figure 28). Note that the new value of RANGE(v_{next}) is to the left of the old one.

(1.b.ii) If the new candidate $\widetilde{w_{next}}$ *is successor* $w_{previous}$, it may be a suitable candidate for v_{next} (see Figure 29). Now the algorithm is in a situation that is similar to the situation we started with in Case (1): it tries to construct an inclusion map from $P[v_{next}]$ to $T[\widetilde{w_{next}}]$, and so on. Eventually it is either able to map $P[v_{next}]$ completely or it finds out that f(LEFT_SIBLING(v_{next})) is not a suitable candidate; hence the algorithm again carries out a backward step to look for a new candidate for LEFT_SIBLING(v_{next}).

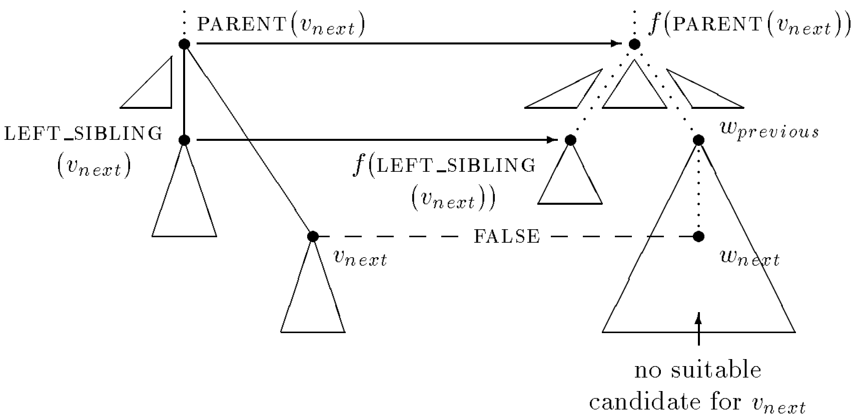

Figure 28: *Interaction, Case (1.b.i), w_{next} is successor of $w_{previous}$, $P[v_{next}]$ cannot be completely mapped to $T[w_{next}]$, the new candidate $\widetilde{w_{next}}$ is to the right of $w_{previous}$, but v_{next} has a left sibling.*

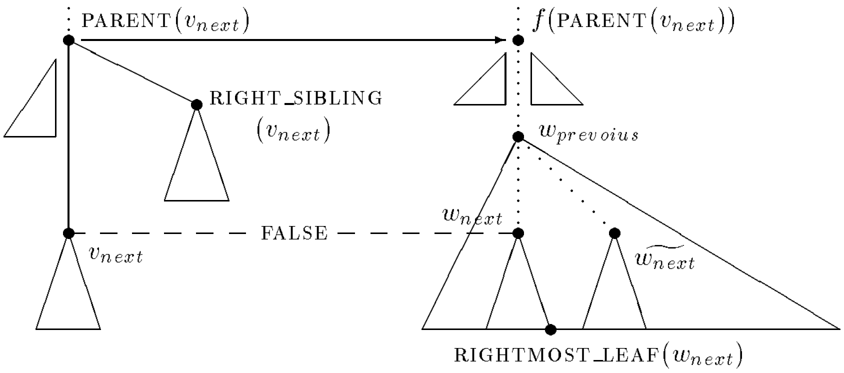

Figure 29: *Interaction, Case (1.b.ii), w_{next} is successor of $w_{previous}$, $P[v_{next}]$ cannot be completely mapped to $T[w_{next}]$, but the new candidate $\widetilde{w_{next}}$ is also successor of $w_{previous}$.*

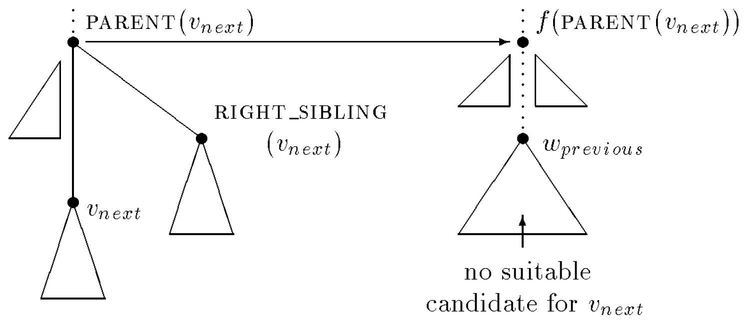

Figure 30: *Interaction, Case (2), w_{next} is to the right of $w_{previous}$.*

(2) If w_{next} *is to the right of* $w_{previous}$, i. e. out of the desired mapping range for v_{next}, then there is no suitable candidate for v_{next} within the desired location, as shown in Figure 30. Since RANGE(v_{next}) has the value $w_{previous}$, in the next iteration of the algorithm a backward step is carried out again. There the algorithm first checks whether v_{next} has a left sibling or not.

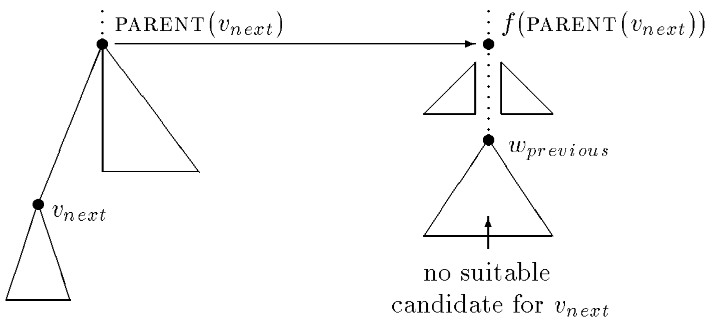

Figure 31: *Interaction, Case (2.a), w_{next} is to the right of $w_{previous}$ and v_{next} has no left sibling.*

(2.a) If v_{next} *has no left sibling*, $P[\text{PARENT}(v_{next})]$ cannot be completely mapped to $T[f(\text{PARENT}(v_{next}))]$, i. e. $f(\text{PARENT}(v_{next}))$ is not a suitable candidate (see Figure 31). Therefore the algorithm sets

$$\text{STATE}(\text{PARENT}(v_{next}), f(\text{PARENT}(v_{next}))) := \text{FALSE},$$

and goes to the next eligible candidate for PARENT(v_{next}).

(2.b) If v_{next} *has a left sibling*, the algorithm sets

$$\text{RANGE}(v_{next}) := f(\text{LEFT\_SIBLING}(v_{next}))$$

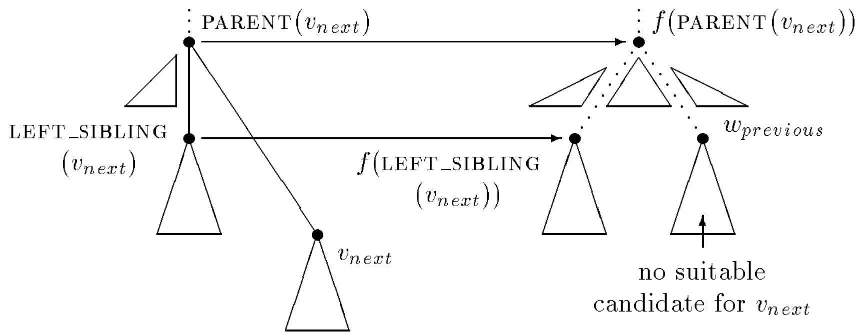

Figure 32: *Interaction, Case (2.b), w_{next} is to the right of $w_{previous}$, but v_{next} has a left sibling.*

and

$$\text{RANGE}(\text{LEFT\_SIBLING}(v_{next})) := f(\text{LEFT\_SIBLING}(v_{next})).$$

Then it chooses a new candidate for LEFT_SIBLING(v_{next}) (see Figure 32). Note that the new value of RANGE(v_{next}) is to the left of the old one.

This completes our illustration of the interaction between forward and backward steps.

5 Implementation of the Algorithm

In this section we sketch an implementation of our algorithm *OrderedTreeInclusion* for solving the ordered tree inclusion problem.

In Figure 33 the main part of the algorithm is given in pseudo-code fashion. Again we assume that the **input** to the algorithm consists two ordered labeled trees, the pattern tree $P = (V_P, E_P)$ and the target tree $T = (V_T, E_T)$, where $V_P = \{v_1, \ldots, v_n\}$, $V_T = \{w_1, \ldots, w_m\}$, and $n \leq m$. In our description we assume that the subscripts of the nodes correspond to their preorder numbers.

First, in line (1), some preprocessing is carried out as described in Subsection (4.3). We omit the details here. In line (2) the root of P is mapped to the root of T. Then the leftmost child of the root of P is considered, and the next eligible candidate for it is chosen in line (3). Now the algorithm is in the start state of the first iteration. In the following we use two binary variables, *ready* and *success*, to indicate whether the algorithm has finished work and whether the construction of an inclusion map has been successful, respectively. Theses variables are initialized in line (4). The following **while**-loop in line (5) corresponds to the iterations of the algorithm. In the **if**-statement in line (6) it is checked whether a forward or a backward step has to be carried out. The corresponding procedures, *ForwardStep* and *BackwardStep*, are described below. Finally, in line (7) the result of the algorithm is given out. If the algorithm has been successful, the constructed inclusion map is given out. Otherwise the algorithm just prints a message that there is no inclusion map.

```
program OrderedTreeInclusion
    begin
(1)     Preprocessing;
(2)     f(v_1) := w_1;
(3)     v_next := v_2;
        w_next := NEXT(LABEL(v_2), w_1);
(4)     ready := false;
        success := false;
(5)     while not ready do
(6)         if w_next is not successor of PARENT(v_next)
                or [(RANGE(v_next) ≠ NIL and w_next is to the right of RANGE(v_next)]
            then
                BackwardStep
            else
                ForwardStep
            fi
        od;
(7)     if success then
            output f
        else
            output "no inclusion map"
        fi
    end
```

Figure 33: *The main part of the algorithm.*

```
procedure ForwardStep
   begin
(1)    if STATE(v_next, w_next) = NIL then
          f(v_next) := w_next;
(1.a)     if LEFTMOST_CHILD(v_next) ≠ NIL then
             w_next := NEXT(LABEL(v_next+1), f(v_next));
             v_next := v_next+1
(1.b)     elsif RIGHT_SIBLING(v_next) ≠ NIL then
             STATE(v_next, w_next) := TRUE;
             w_next := NEXT(LABEL(v_next+1), RIGHTMOST_LEAF(f(v_next)));
             v_next := v_next+1
(1.c)     else
             v_k := v_next;
             while RIGHT_SIBLING(v_k) = NIL and v_k ≠ ROOT(P) do
                STATE(v_k, f(v_k)) := TRUE;
                for each child u of v_k whose RANGE-value has been set do
                   RANGE(u) := NIL
                od;
                v_k := PARENT(v_k)
             od;
             STATE(v_k, f(v_k)) := TRUE;
             for each child u of v_k whose RANGE-value has been set do
                RANGE(u) := NIL
             od;
             if v_k = ROOT(P) then
                ready := true;
                success := true
             else
                w_next := NEXT(LABEL(v_next+1),
                              RIGHTMOST_LEAF(f(LEFT_SIBLING(v_next))));
                v_next := v_next+1
             fi
          fi
(2)    elsif STATE(v_next, w_next) = FALSE then
          w_next := NEXT(LABEL(v_next), RIGHTMOST_LEAF(w_next))
(3)    elsif STATE(v_next, w_next) = TRUE then
          f(v_next) := w_next;
(3.a)     if RIGHT_SIBLING(v_next) ≠ NIL then
             w_next := NEXT(LABEL(RIGHT_SIBLING(v_next)), RIGHTMOST_LEAF(w_next));
             v_next := RIGHT_SIBLING(v_next)
(3.b)     else
             v_next := v_PRE(RIGHTMOST_LEAF(v_next))+1;
             w_next := NEXT(LABEL(v_next), RIGHTMOST_LEAF(f(LEFT_SIBLING(v_next))))
          fi
       fi
   end
```

Figure 34: *The forward step of the algorithm.*

```
procedure BackwardStep
  begin
(1)  if LEFT_SIBLING(v_next) ≠ NIL then
       RANGE(v_next) := f(LEFT_SIBLING(v_next));
       RANGE(LEFT_SIBLING(v_next)) := f(LEFT_SIBLING(v_next));
       v_next := LEFT_SIBLING(v_next);
       w_next := NEXT(LABEL(v_next, f(v_next)))
(2)  else
       STATE(PARENT(v_next), f(PARENT(v_next))) := FALSE;
       if PARENT(v_next) = ROOT(P) then
         ready := true;
         success := false
       else
         for each child u of PARENT(v_next)
               whose RANGE-value has been set do
           RANGE(u) := NIL
         od;
         v_next := PARENT(v_next);
         w_next := NEXT(LABEL(v_next, RIGHTMOST_LEAF(f(v_next))))
       fi
     fi
  end
```

Figure 35: *The backward step of the algorithm.*

In Figure 34 a procedure that implements the forward step of the algorithm is given. There the numbering of the statements corresponds to the cases of the forward step as discussed in Subsection (4.4.2). Finally, in Figure 35, a procedure that implements the backward step of the algorithm is given. The numbering of the statements corresponds to the cases of the backward step, as discussed in Subsection (4.4.3). We omit a detailed description of these procedures here.

6 Analysis of the Algorithm

In this section we prove the correctness of our algorithm and consider its complexity.

6.1 Correctness

In this subsection we show the correctness of the algorithm OrderedTreeInclusion. First we prove that an inclusion map constructed by the algorithm satisfies the conditions of ordered tree inclusion. The next two facts show that the nodes of the pattern tree P are only mapped to feasible candidates of the target tree T.

Fact 2 *In the algorithm* OrderedTreeInclusion *the current candidate w_{next} of the target tree T is always the next eligible candidate for the node v_{next} of the pattern tree P under consideration.*

PROOF. Immediately from the discussion of the forward and the backward step. □

Fact 3 *If a node v_{next} of the pattern tree P is mapped to a candidate w_{next} of the target tree T in the algorithm* OrderedTreeInclusion*, w_{next} is a feasible candidate for v_{next}.*
PROOF. Since w_{next} is the next eligible candidate for v_{next}, its preorder number is greater than that of $f(\text{PARENT}(v_{next}))$ or that of $\text{RIGHTMOST\_LEAF}(f(\text{LEFT\_SIBLING}(v_{next})))$, if v_{next} has a left sibling. Since a forward step is only carried out if w_{next} is successor of $f(\text{PARENT}(v_{next}))$, v_{next} is only mapped to w_{next}, if w_{next} is feasible. □

These facts imply that the algorithm constructs only inclusion maps that satisfy the conditions of ordered tree inclusion.

Corollary 5 *If the value of* $\text{STATE}(v, w)$ *for a match* (v, w) *is set to* TRUE *in the algorithm* OrderedTreeInclusion*, the algorithm has constructed an inclusion map from $P[v]$ to $T[w]$ that satisfies the conditions of ordered tree inclusion.* □

Hence the algorithm has constructed an inclusion map from the pattern tree P to the target tree T that satisfies the conditions of ordered tree inclusion, if it sets the value of $\text{STATE}(v_1, w_1)$ to TRUE.

Now we show that there is actually no inclusion map if the algorithm does not find one.

Fact 4 *If the value of* $\text{STATE}(v, w)$ *for a match* (v, w) *is set to* FALSE *in the algorithm* OrderedTreeInclusion*, there is no inclusion map from $P[v]$ to $T[w]$ that satisfies the conditions of ordered tree inclusion.*
PROOF. Since the value of $\text{STATE}(v, w)$ can be set to FALSE only in Case (2) of the backward step, the assertion immediately follows from Lemma 4. □

Hence there is no inclusion map from the pattern tree P to the target tree T that satisfies the conditions of ordered tree inclusion if the algorithm sets the value of $\text{STATE}(v_1, w_1)$ to FALSE.

Now the correctness of the algorithm follows immediately.

Theorem 1 *The algorithm* OrderedTreeInclusion *is correct.* □

6.2 Complexity

Now we analyze the time complexity of the algorithm OrderedTreeInclusion. This is composed of the time complexity of preprocessing and the time spent in the forward and backward steps of the main part.

A forward step can be carried out in constant time, with the exception of *going up the path* in Case (1.c) of Subsection (4.3.2). During *going up the path* the value of $\text{STATE}(v, w)$ for a match (v, w) is set from NIL to TRUE in every step. Since the value of $\text{STATE}(v, w)$ is not changed afterwards, the total time spent in all forward steps for *going up the path* is at most $O(\#\text{matches})$. Furthermore the value of $\text{STATE}(v, w)$ for a match (v, w) is also not changed after it has been set from NIL to FALSE.

Next we prove an upper bound for the number of forward steps carried out per match.

Lemma 5 *A match* (v_k, w_l) *is considered in at most*

$$|\{w \mid w \textit{ is on the path from } w_l \textit{ to } \text{ROOT}(T) \wedge \text{LABEL}(w) = \text{LABEL}(\text{PARENT}(v_k))\}|$$

forward steps.
PROOF. The match (v_k, w_l) can only be considered if PARENT(v_k) has been mapped to an ancestor of w_l, i. e. a node that is on the path from w_l to ROOT(T) and that has the label LABEL(PARENT(v_k)). It remains to show that the match (v_k, w_l) is considered only once per such a mapping of PARENT(v_k) in a forward step.

Let w_{first} be the first ancestor of w_l, such that PARENT(v_k) is mapped to w_l and thereafter the algorithm considers the match (v_k, w_l) the first time. At that time the value of STATE(v_l, w_k) is NIL. After mapping v_k to w_l the algorithm tries to construct an inclusion map from $P[v_k]$ to $T[w_l]$. In doing so the algorithm considers only nodes which are successors of v_k and w_l, respectively, until the state of (v_k, w_l) is set either to TRUE or to FALSE.

If STATE(v_k, w_l) *is set to* TRUE, there may be carried out some backward steps in the remainder of the phase for the match (PARENT$(v_k), w_{\mathit{first}}$). This does not concern the match (v_k, w_l) unless in one of these backward steps another candidate for v_k is sought. Then the match (v_k, w_l) is dismissed, RANGE(v_k) is set to w_l, and the algorithm looks for another candidate for v_k among the successors of w_l (cf. Case (1) of the backward step). According to Lemma 3, the algorithm does not consider the match (v_k, w_l) in this phase again.

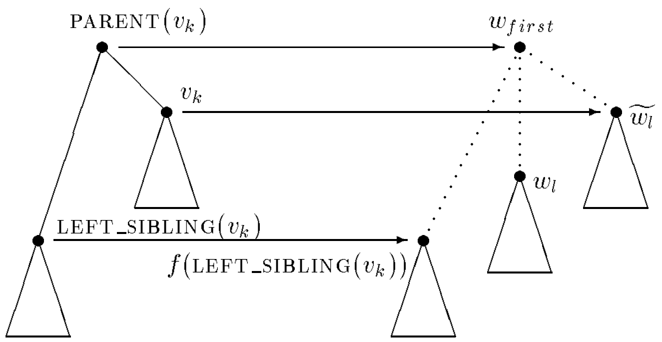

Figure 36: STATE(v_k, w_l) = FALSE.

If STATE(v_k, w_l) *is set to* FALSE, the algorithm seeks for another candidate $\widetilde{w_l}$ for v_k to the right of w_l, as shown in Figure 36. If $\widetilde{w_l}$ is also dismissed afterwards, the algorithm again seeks for another candidate for v_k. If STATE$(v_k, \widetilde{w_l})$ = TRUE holds, it searches among the successors of $\widetilde{w_l}$, and if STATE$(v_k, \widetilde{w_l})$ = FALSE holds, it searches among the nodes that are to the right of $\widetilde{w_l}$. In neither case can the match (v_k, w_l) be considered again.

If eventually no more candidates for v_k are found, the algorithm goes back to the left sibling of v_k and looks for another candidate for this node. But previously RANGE(v_k) has beem set to f(LEFT_SIBLING(v_k)). Hence the match (v_k, w_l) cannot be considered in this phase again. If v_k has no left sibling, the current phase is finished immediately.

At the end of the phase for the match $(\text{PARENT}(v_k), w_{first})$ the algorithm sets $\text{STATE}(\text{PARENT}(v_k), w_{first})$ either to TRUE or to FALSE. Subsequently it may happen that the match $(\text{PARENT}(v_k), w_{first})$ is considered again. If $\text{STATE}(\text{PARENT}(v_k), w_{first}) = \text{TRUE}$ holds, the whole inclusion map from $P[\text{PARENT}(v_k)]$ to $T[w_{first}]$ is taken over immediately. If, on the other hand, $\text{STATE}(\text{PARENT}(v_k), w_{first}) = \text{FALSE}$ holds, the algorithm immediately looks for a new candidate for $\text{PARENT}(v_k)$ to the right of w_{first}. In both cases the match (v_k, w_l) is not considered again.

Thus we have shown that the algorithm considers the match (v_k, w_l) in all phases for the match $(\text{PARENT}(v_k), w_{first})$ together in only one forward step.

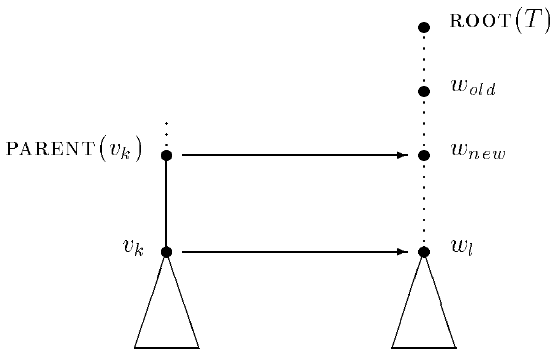

Figure 37: *The location of w_{old} and w_{new}.*

Now, let w_{old} be a previous image of $\text{PARENT}(v_k)$, w_{new} the current image of $\text{PARENT}(v_k)$, and let w_{old} and w_{new} both be on the path from w_l to $\text{ROOT}(T)$, as shown in Figure 37. Note that this situation can only occur if the $\text{STATE}(\text{PARENT}(v_k), w_{old})$ has been set to TRUE.

If the algorithm considers the match (v_k, w_l) in the phase for the match $(\text{PARENT}(v_k), w_{new})$ again, we have to distinguish two cases depending on the value of $\text{STATE}(v_k, w_l)$.

If $\text{STATE}(v_k, w_l) = \text{TRUE}$ holds, the algorithm immediately makes use of the whole inclusion map from $P[v_k]$ to $T[w_l]$ and proceeds with considering the right sibling of v_k, if it exists. By an argumentation similar to that for w_{first} we have that the algorithm does not consider the match (v_k, w_l) in this phase again, even if some backward steps are carried out in the remainder of this phase.

If $\text{STATE}(v_k, w_l) = \text{FALSE}$ holds, the algorithm immediately looks for another candidate for v_k to the right of w_l. Analogously to the argumentation for w_{first} above, the algorithm does not consider the match (v_k, w_l) in this phase again.

If the match $(\text{PARENT}(v_k), w_{new})$ is considered again, the algorithm does not consider the match (v_k, w_l) in the corresponding phase, because $\text{STATE}(\text{PARENT}(v_k), w_{new})$ has then been set either to TRUE or to FALSE.

Thus we have shown that the algorithm considers the match (v_k, w_l) in all phases for the match $(\text{PARENT}(v_k), w_{new})$ together only once. This completes the proof. □

$|\{w \mid w \text{ is on the path from } w_l \text{ to ROOT}(T) \wedge \text{LABEL}(w) = \text{LABEL}(\text{PARENT}(v_k))\}|$ is bounded by DEPTH(T). Note that this bound is generally not very tight. Nevertheless, we can now give an upper bound for the number of forward steps in the algorithm OrderedTreeInclusion.

Corollary 6 *In the algorithm OrderedTreeInclusion at most $O(\#matches \cdot \text{DEPTH(T)})$ forward steps are carried out.* □

Note that there may be some matches which the algorithm will never consider. As the algorithm considers the nodes in ascending preorder, a match (v, w) can only be considered, if the depth of w is at least as large as the depth of v, and if the number of nodes which are to the left of w is at least as large as the number of nodes which are to the left of v.

However, the number of forward steps is also an upper bound for the total number of single mappings of nodes of the pattern tree to nodes of the target tree in the algorithm OrderedTreeInclusion. Hence we also have an upper bound for the number of backward steps, since in every backward step one mapping of a node of the pattern tree to a node of the target tree is dismissed.

Corollary 7 *In the algorithm OrderedTreeInclusion at most $O(\#matches \cdot \text{DEPTH(T)})$ backward steps are carried out.* □

Furthermore, a backward step can be carried out in constant time.

Since the space complexity of the algorithm is dominated by the STATE and the NEXT arrays, we have the following results for the complexity of the algorithm OrderedTreeInclusion.

Theorem 2 *The algorithm OrderedTreeInclusion has a running time of*

$$O(|\Sigma_P| \cdot |T| + \#matches \cdot \text{DEPTH}(T)),$$

where Σ_P is the alphabet of the labels of the pattern tree, and a space complexity of

$$O(|\Sigma_P| \cdot |T| + \#matches).$$

□

7 The Computation of All Inclusion Maps

Our algorithm computes only *one* inclusion map f_0 from the pattern tree P to the target tree T. This map is in the following sense the first one: the root of the pattern tree is mapped to the root of the target tree, the node v_2 of the pattern tree is mapped to the in preorder first suitable candidate w_r for it which is not the root, the node v_3 of the pattern tree is mapped to the first suitable candidate for it with a preorder number greater than that of w_r, and so on. Note that it may be possible that there is a suitable candidate for a node v_s of the pattern tree with an preoder number lower than that of

$f_0(v_s)$. For example, assume that LEFT_SIBLING(v_s) can be mapped to a successor of $f_0(\text{LEFT\_SIBLING}(v_s))$ as well, i. e. the subtree $P[\text{LEFT\_SIBLING}(v_s)]$ can be mapped to a proper subtree of $T[f_0(\text{LEFT\_SIBLING}(v_s))]$. Then it may be possible to map also the subtree $P[v_s]$ to a subtree of $T[f_0(\text{LEFT\_SIBLING}(v_s))]$ such that the node v_s is mapped to a node with a preorder number lower than that of $f_0(v_s)$.

Nevertheless, we can enumerate *all* inclusion maps from the pattern tree to the target tree as follows. Having constructed the inclusion map f_0, we dismiss the mapping of the last node v_n of the pattern tree, and map v_n to the next suitable candidate for it (if any). Thus we have constructed the "next" inclusion map f_1. Now we again dismiss the mapping of v_n and look for next eligible candidate for it. This process is iterated until no more suitable candidate for v_n is found. Then we have enumerated all inclusion maps in which the nodes v_1 to v_{n-1} of the pattern tree are mapped to the nodes $f_0(v_1)$ to $f_0(v_{n-1})$ of the target tree. Now we dismiss the mapping of the node v_{n-1} and look for the next eligible candidate for it. If there is any suitable candidate w_j, we complete the inclusion map. Now we enumerate all inclusion maps in which the nodes v_1 to v_{n-2} are mapped to the nodes $f_0(v_1)$ to $f_0(v_{n-2})$ and v_{n-1} is mapped to w_j as describe above. Afterwards we dismiss the mapping of v_{n-1} to w_j and look for the next eligible candidate for v_{n-1}. Eventually we have enumerated all inclusion maps in which the nodes v_1 to v_{n-2} of the pattern tree are mapped to the nodes $f_0(v_1)$ to $f_0(v_{n-1})$ of the target tree. This process is iterated until all inclusion maps from the pattern tree to the target tree have been enumerated. Note that we can use the same STATE array during the whole enumeration. This could avoid much duplicate work.

8 Conclusion and Further Work

We have presented a new algorithm for the ordered tree inclusion problem that is better than the previous ones in many cases. Next we would like to eliminate the factor DEPTH(T) in the time complexity of our algorithm. Furthermore, we would like to implement our algorithm to see how competetive it is in practice. In this implementation some of the heuristics mentioned above could be applied.

Next we would like to apply the techniques we have used for the ordered tree inclusion to other tree inclusion problems, for example the ordered path and the ordered region inclusion problems. We conjecture that we can solve these problems within the same time and space bounds by modifying our algorithm for the ordered tree inclusion problem appropriately.

Finally we would like to attack the largest common substructure problem in order to beat the complexity resulting from the application of the algorithm of [20] for the tree editing problem to this problem.

Acknowledgement. I would like to thank Prof. Dr. N. Blum for helpful discussions on this work.

References

[1] A. Apostolico and C. Guerra, *The Longest Common Subsequence Problem Revisited*, Algorithmica **2** (1987), pp. 315 - 336.

[2] M. Dubiner, Z. Galil and E. Magen, *Faster Tree Pattern Matching*, Proc. 31st FOCS (1990), pp. 145 - 150.

[3] G. H. Gonnet and F. W. Tompa, *Mind your Grammar - a New Approach to Text Databases*, Proc. of the Conf. on Very Large Databases 1987 (VLDB'87), pp. 339 - 346.

[4] D. S. Hirschberg, *A Linear Space Algorithm for Computing Maximal Common Subsequences*, CACM **18** (1975), pp. 341 - 343.

[5] D. S. Hirschberg, *Algorithms for the Longest Common Subsequence Problem*, JACM **24** (1977), pp. 664 - 675.

[6] C. M. Hoffman and M. J. O'Donnell, *Pattern Matching in Trees*, JACM **29** (1982), pp. 68 - 95.

[7] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Chapter 4, Addison-Wesley, Reading, MA, 1979, pp. 77 - 106.

[8] D. S. Johnson, *The NP-completeness Column: An Ongoing Guide*, J. Algorithms **8** (1987), pp. 285 - 303.

[9] P. Kilpeläinen, G. Linden, H. Mannila and E. Nikunen, *A Structured Document Database System*, in R. Furuta (ed.), *EP'90 - Proc. of the Int. Conf. on Electronic Publishing, Document Manipulation & Typography*, The Cambridge Series on Electronic Publishing, Cambridge University Press, 1990.

[10] P. Kilpeläinen and H. Mannila, *Retrieval from Hierarchical Texts by Partial Patterns*, in R. Korfhage, E. Rasmussen and P. Willet (eds.), SIGIR '93 - Proc. of the 16th Ann. Int. ACM SIGIR Conf. on Research and Development in Informational Retrieval 1993, pp. 214 - 222.

[11] P. Kilpeläinen and H. Mannila, *Query Primitives for Tree-Structured Data*, Proc. 5th CPM (1994), pp. 213 - 225.

[12] P. Kilpeläinen and H. Mannila, *Ordered and Unordered Tree Inclusion*, SIAM J. Comput. **24** (1995), pp. 340 - 356.

[13] D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA, 1969, p. 347.

[14] S. R. Kosaraju, *Efficient Tree Pattern Matching*, Proc. 30th FOCS (1989), pp. 178 - 183.

[15] J. Matousek and R. Thomas, *On the Complexity of Finding Iso- and Other Morphisms for Partial k-Trees*, Discrete Mathematics **108** (1992), pp. 343 - 364.

[16] C. Rick, *A New Flexible Algorithm for the Longest Common Subsequence Problem*, Nordic J. Comput. **2** (1995), pp. 444 - 461.

[17] N. Robertson and P. D. Seymor, *Graph Minors. II. Algorithmic Aspects of Tree-Width*, J. Algorithms **7** (1986), pp. 309 - 322.

[18] K.-C. Tai, *The Tree-to-Tree Correction Problem*, JACM **26** (1979), pp. 422 - 433.

[19] T. Winograd, *Language as a Cognitive Process*, Vol. 1: *Syntax*, Chapter 3, Addison-Wesley, Reading, MA, 1983, pp. 72 - 132.

[20] K. Zhang and D. Shasha, *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems*, SIAM J. Comput. **18** (1989), pp. 1245 - 1262.