Tree Matching Problems with Applications to Structured Text Databases

Pekka Kilpeläinen

Ph.D. dissertation
Department of Computer Science, University of Helsinki
Report A-1992-6, Helsinki, Finland, November 1992

Abstract

Tree matching is concerned with finding the instances, or matches, of a given pattern tree in a given target tree. We introduce ten interrelated matching problems called tree inclusion problems. A specific tree inclusion problem is defined by specifying the trees that are instances of the patterns. The problems differ from each other in the amount of similarity required between the patterns and their instances. We present and analyze algorithms for solving these problems, and show that the computational complexities of the problems range from linear to NP-complete.

The problems are motivated by the study of query languages for structured text databases. The structure of a text document can be described by a context-free grammar, and text collections can be represented as collections of parse trees. Matching-based operations are an intuitive basis for accessing the contents of structured text databases. In "G-grammatical" tree inclusion problems the target tree is a parse tree over a context-free grammar G. We show that a certain natural class of grammars allows solving some of the grammatical inclusion problems in linear time.

Tree inclusion problems are extended by introducing logical variables in the patterns. These variables allow extracting substructures of the pattern matches and posing equality constraints on them. We show that most of the tree inclusion problems with logical variables are NP-hard, and also consider solving their polynomial versions. As an application of these problems we finally show how tree inclusion with logical variables can be used for querying structured text databases, and discuss how the inclusion queries should be evaluated in practice.

Contents

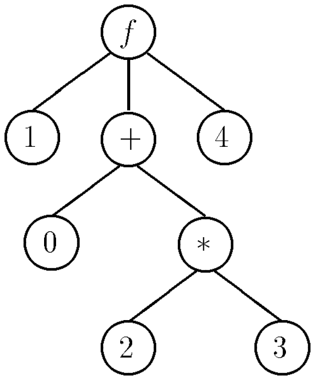
Chapter 1

Introduction

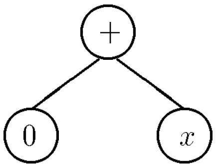
Trees are one of the most important ways of structuring data. They are suitable for representing any information with hierarchical structure. Trees can be used to represent natural language sentences, computer programs, algebraic formulae, molecule structures, and family relationships, just to mention some application areas.

Pattern matching is the process of locating substructures of a larger structure, the *target*, by comparing them against a given form called the *pattern*. Posing pattern matching requests is descriptive by nature in the sense that one describes how the results of the search should look like, instead of expressing how they are to be found. This makes pattern matching a central operation in many declarative programming systems, and also a promising framework for the retrieval of information.

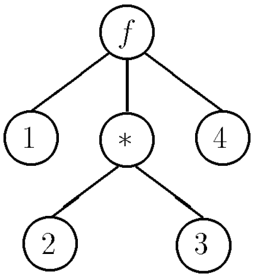
Matching of tree structures and the related unification problem have an important role in the definition and implementation of functional and logic programming languages like Lisp and Prolog. Representing terms as tree structures is typical in those systems. For example, consider a programming system that deals with simplifying arithmetic expressions. The system might be required to simplify a term like $f(1,(0+2*3),4)$ that represents some function f applied to arguments 1, $0+2*3$ and 4. This term could be represented inside the system as the following tree structure.



Assume that the system includes a rewriting rule $0+x \rightarrow x$, which tells that any expression of the form $0+x$ can be simplified to x. Now this rule can be implemented by treating the left-hand side of the rule as the *pattern tree* below.



The earlier tree that represents the object of simplification is then treated as a *target tree*, and the pattern is matched against the target. Matching succeeds against the subtree whose root is labeled by the +-sign. The simplification step can be implemented by replacing that tree by its second subtree that was matched by the x-node of the pattern. The result is as follows.



The above was an example of *classical tree pattern matching*. In this problem the pattern describes its matches rather precisely. For instance, the previous pattern matches only against trees that have exactly two immediate subtrees, whose root label is +, and whose first subtree is 0; the second subtree of the pattern occurrences can be arbitrary.

Sometimes we may need more liberal matching of tree structures than classical tree pattern matching. For example, assume that we have a large collection of parse trees for some natural language sentences. Figure 1.1 is an example of such a parse tree.

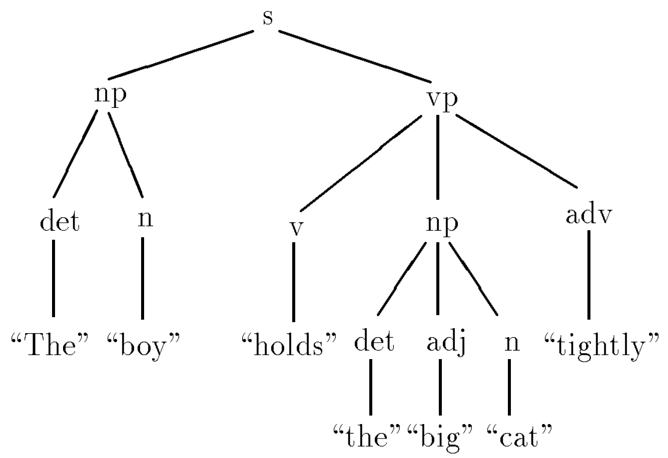


Figure 1.1: A parse tree

Linguists might want to locate sentences from the collection by expressing conditions on the form of their parse trees. For example the following queries on the data might be of interest.

1. Locate those sentences that contain verb "holds", noun "cat", and some adverb.

2. Locate those sentences that contain a noun phrase consisting of determiner "the", adjective "big", and noun "cat", in this order.

3. Extract the nouns that are preceded in a verb phrase by the verb "holds".

4. Extract the words that appear in a sentence both as a verb and a noun.

The richness of natural language leads to very diverse syntactic structures and makes queries like those above difficult to express using classical tree pattern matching. We consider in this thesis extensions of the tree pattern matching problem that allow us to express conditions like those above. We call these extensions *tree inclusion problems*. The first two sample queries above can be treated as tree inclusion problems, where we search for occurrences of a pattern tree. The last two queries above can be treated as inclusion problems extended with *logical variables*. Logical variables can be used to extract substructures of the pattern occurrences and for posing equality constraints on those structures.

The specific application that we have in mind is to apply tree matching to locating structures in *structured text databases*. Various text documents comprise some kind of internal structure. Examples of such documents are manuals for software and equipment, encyclopedias, and dictionaries. A *structured text database system* is a computer system that utilizes the structure of the texts in its task of storing and manipulating text documents. The model for text databases that we consider uses grammars as database schemas and parse trees as database instances. In this setting, pattern trees can be given as queries for locating parts of interest in database trees.

The main theme of the thesis is the study of reasonable variations of pattern matching on tree structures. The contents are organized as follows. Chapter 2 presents the basic notions of trees that are used in the rest of the thesis. Tree inclusion problems are introduced in Chapter 3. In Chapter 4 we consider solving these problems. We give a general solution scheme for the problems, and present and analyze algorithms for solving particular tree inclusion problems. The computational complexities of the problems are shown to range from linear to NP-complete.

In Chapter 5 we discuss application oriented special cases called *grammatical tree inclusion problems*. In a grammatical problem the target is a parse tree over some grammar. We show that the class of so called nonperiodic grammars allow certain tree inclusion problems to be solved in linear time. We also discuss possibilities to preprocess the pattern when the grammar of the target is known. In Chapter 6 we consider extending the tree inclusion problems by allowing the patterns to contain logical variables. Most of the tree inclusion problems with logical variables are shown to be NP-hard. Solving the polynomially solvable problems and some polynomial subclasses of ordered tree inclusion with logical variables are considered at the end of Chapter 6. In Chapter 7 we discuss querying structured text databases using tree inclusion. We give examples of using inclusion queries and outline how they should be evaluated in practice. Chapter 8 is a conclusion.

Chapter 2

Preliminaries

This chapter defines the basic notions of trees and their components used in the rest of the thesis. The concepts are standard and can be found, although with slightly differing notations, in many sources that deal with tree algorithms; [Knu69], [AHU74], and [Tar83] are three basic references.

A *binary relation* on a set D is a subset of the cartesian product $D \times D$. Let R be a binary relation on D. The *transitive closure* of R, denoted by R^+, and the *reflexive transitive closure* of R, denoted by R^*, are defined as follows:

$$\begin{aligned}
R^0 &= \{(x,x) \mid x \in D\} \ , \\
R^{n+1} &= \{(x,y) \mid \exists\, z \in D : (x,z) \in R, (z,y) \in R^n\} \ , \text{ for } n \geq 0, \\
R^+ &= \bigcup_{n>0} R^n \ , \text{ and} \\
R^* &= \bigcup_{n\geq 0} R^n = R^0 \cup R^+ \ .
\end{aligned}$$

A *rooted tree* is a structure $T = (V, E, \mathit{root}(T))$, where V is a finite set of *nodes*, $\mathit{root}(T) \in V$ is a node called the *root* of tree T, and E is a binary relation on V that satisfies the conditions listed below. If $(u,v) \in E$, we say that (u,v) is an *edge* and that node u is the *parent* of node v, denoted by $\mathit{parent}(v)$. The set of edges must satisfy the following:

1. The root has no parent.

2. Every node of the tree except the root has exactly one parent.

3. All nodes are reachable via edges from the root, i.e., $(\mathit{root}(T), v) \in E^*$ for all nodes $v \in V$.

If we omit the special status of the root of a rooted tree and the direction of the edges, we get an *unrooted tree*. We consider only rooted trees if not especially stated otherwise.

Rooted trees are a special case of *directed graphs*; graphs do not pose restrictions on their edges. The nodes of a graph are also called *vertices*. A graph (tree) is *labeled*, if a function *label* from the nodes of the graph (tree) to some alphabet is given.

A *path* in a directed graph is a sequence of edges (v_1, v_2), (v_2, v_3), …, (v_{n-1}, v_n); this is called a *path from* v_1 *to* v_n and its *length* is $n-1$. In a tree there is a unique path from the root to each node. We define the *depth* or the *height* of a tree T, denoted by $depth(T)$, to be one greater than the length of the longest path in T.

The nodes of a rooted tree with a common parent are *children* of their parent, denoted by

$$children(u) = \{v \in V \mid (u,v) \in E\} \ .$$

A node without children is a *leaf*. Nodes which are not leaves are *internal nodes*. Leaves are sometimes called *external nodes*. The *descendants* of a node u are defined by

$$desc(u) = \{v \in V \mid (u,v) \in E^+\} \ ,$$

and the *ancestors* of u are defined by

$$anc(u) = \{v \in V \mid (v,u) \in E^+\} \ .$$

The set $A^i(u) = \{v \in V \mid (v,u) \in E^i\}$ is obviously for each i either empty or a singleton; if it is a singleton, the node v belonging to the set $A^i(u)$ is called the ith *ancestor* of u. A node is its own zeroth ancestor, and the parent is the first ancestor of a node.

A rooted tree is *ordered* if the children of each internal node having k children are uniquely numbered by $1, \ldots, k$. Let u be a node in an ordered tree. The child number i of node u is denoted by $child(u,i)$. If node u has k children ($k > 0$), the node $child(u,1)$ is the *leftmost child* of u, and $child(u,k)$ is the *rightmost child* of u.

Let u be a node in a rooted tree $T = (V, E, root(T))$. The *subtree of* T *rooted by node* u, denoted by $T[u]$, is the tree (V', E', u), where

$$\begin{aligned} V' &= \{u\} \cup desc(u) \ , \text{ and} \\ E' &= E \cap (V' \times V') \ . \end{aligned}$$

The *subtrees of a tree* T are the trees rooted by the nodes of T. If T is ordered or labeled, the labeling of nodes and ordering of children in the subtrees of T are the same as the labeling and ordering in T. The tree rooted by the child number i of a node u is called the *subtree number* i *of node* u. The subtrees of $root(T)$ are the *immediate subtrees of tree* T. A subtree of T is a *proper subtree* of T if it is not rooted by the root of T.

We can omit the ordering of an ordered tree and consider it as an unordered tree. In the sequel, if not stated otherwise, we mean by trees ordered and labeled trees. Trees have a natural representation as linear terms. Let T be a tree with immediate subtrees $T_1, \ldots, T_k$ and $label(root(T)) = a$. Then the *term representation* of T, denoted by $Term(T)$, is defined inductively by

$$Term(T) = a(Term(T_1), \ldots, Term(T_k)) \;;$$

if $k = 0$, we omit the parentheses after a. If s is the term representation of a tree T, we say that T is the *tree represented by term* s, and denote it by $Tree(s)$. Sometimes we refer to tree $Tree(s)$ simply by term s. For an example, see Figure 2.1.

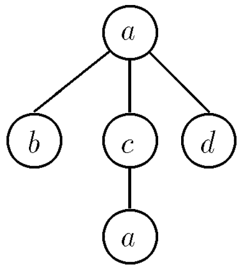


Figure 2.1: The tree represented by the term $a(b, c(a), d)$.

A *forest* is an ordered sequence of trees no two of which have nodes in common. The forest of trees $T_1, \ldots, T_k$ is denoted by $\langle T_1, \ldots, T_k \rangle$. Let $F = \langle T_1, \ldots, T_k \rangle$ be a forest, and u a node in a tree T_i of F. The *subtree of* F *rooted by* u, denoted by $F[u]$, is defined by $F[u] = T_i[u]$. The *subtrees of a forest* F are the subtrees of the trees in F.

The subtrees of a node u with k children in a tree T form the forest $\langle T[child(u, 1)], \ldots, T[child(u, k)] \rangle$. The roots of the trees forming a forest are *siblings* to each other. Let $u = root(T_i)$ in a forest $\langle T_1, \ldots, T_k \rangle$. If $i > 1$, the nodes $root(T_j)$ where $1 \leq j < i$ are *left siblings* of node u, and $root(T_{i-1})$ is the *previous sibling* of u. If $i < k$, the nodes $root(T_j)$ where $i < j \leq k$ are *right siblings* of u, and $root(T_{i+1})$ is the *next sibling* of u. Usually we do not distinguish between a single tree T and the forest $\langle T \rangle$.

The nodes u of a forest $F = \langle T_1, \ldots, T_k \rangle$ where $k > 0$ can be assigned *preorder numbers* $pre(u)$ along the following rules:

1. The preorder number of $root(T_1)$ is 1.
2. The preorder number of the leftmost child of a node u is $pre(u) + 1$.
3. Let node u be the next sibling of node v, and let p be the largest preorder number assigned to the nodes in $F[v]$. Then $pre(u) = p + 1$.

The leaf in a forest F with the smallest preorder number is the *leftmost leaf* of F, and the leaf with the greatest preorder number is the *rightmost leaf* of F.

Postorder numbers are assigned to the nodes of a forest $\langle T_1, \ldots, T_k \rangle$ where $k > 0$ by the following rules:

1. Let u be the leftmost leaf of T_1. Then the postorder number $post(u)$ of u is 1.
2. Let u be an internal node, and let p be the largest postorder number assigned to the descendants of u. Then $post(u) = p + 1$.
3. Let node v be the next sibling of node u. Then the postorder number of the leftmost leaf of $F[v]$ is $post(u) + 1$.

Figure 2.2 is an example of preorder and postorder numbering.

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

Lemma 2.1 Let u and v be nodes in a forest F. Then u is an ancestor of v if and only if $pre(u) < pre(v)$ and $post(u) > post(v)$.

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

Many of the tree inclusion problems in the next chapter can be characterized operationally by *tree edit operations.* Tree edit operations form the basis for the edit distance of trees, which can be used for measuring the similarity of trees. This topic is discussed in [Sel77], [Tai79], and [ZS89].

Let u be a node in a tree T. The effect of the *deletion* of node u from T is as follows. If u is the root of T, the result is the forest of immediate subtrees of T. Otherwise, let u be the ith child of a node v. In this case

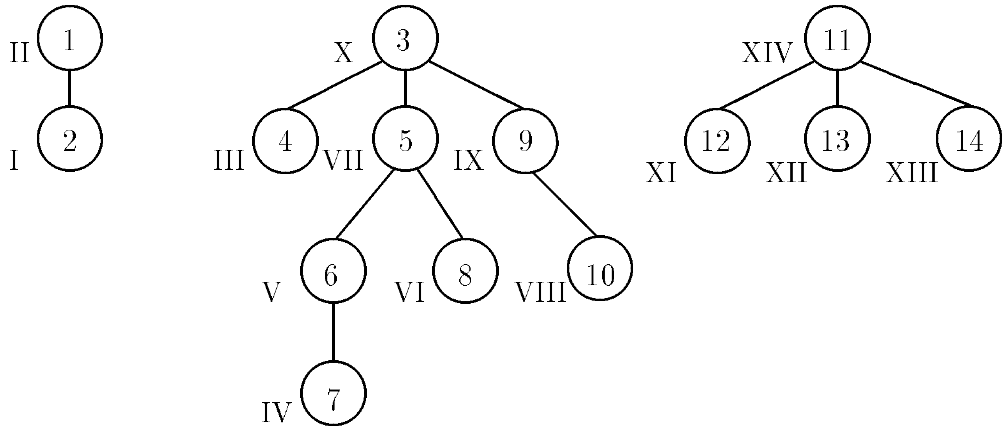


Figure 2.2: Numbering nodes of a forest. The preorder numbers are shown as Arabic numbers inside the nodes, and the postorder numbers are shown as Roman numbers to the left of the corresponding nodes.

the result is tree T with node u removed and the edge from v to u replaced by edges from v to the children of u. The relative order of children does not change: Let the children of v be $v_1, \ldots, v_i, \ldots, v_k$, and the children of u be $u_1, \ldots, u_l$. Then, after the removal of $u = v_i$, the children of v are $v_1, \ldots, v_{i-1}, u_1, \ldots, u_l, v_{i+1}, \ldots, v_k$. For an example, see Figure 2.3.

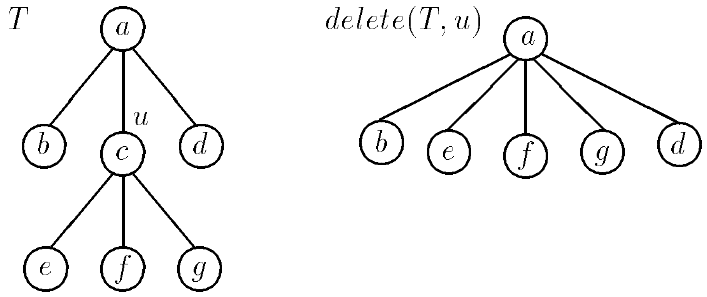


Figure 2.3: The effect of deleting the node u from the tree T.

An *insertion* is the inverse operation of a deletion. Two derived operations can be defined by the deletion of nodes [WJZS91]. *Pruning* at a node u means deleting all the descendants of u. *Cutting* at a node u means pruning at u

and deleting u.

Another basic tree editing operation is the *permutation of siblings*. Let u, v and p be nodes in tree T with $u = child(p, i)$ and $v = child(p, j)$. Permuting u and v changes u to be $child(p, j)$ and v to be $child(p, i)$.

Chapter 3

Tree inclusion problems

The *general tree inclusion problem* given a *pattern tree* P and a *target tree* T is to locate the subtrees of T that are *instances* of P. A specific inclusion problem is defined by fixing the *instance relation* that specifies the instances of each pattern. The pattern is said to *match* or *occur* at the root of the trees that are instances of the pattern. If P matches at a node v of T, we say that v is an *occurrence* of P.

In this chapter we introduce ten interrelated inclusion problems on trees. The problems are introduced by restricting their instance relation step by step. The problems divide into two groups of five problems: the unordered problems, where the left-to-right order of the pattern nodes does not matter, and the ordered problems, where it does matter. The motivation for introducing the problems is to provide means of locating data in a target tree by giving a pattern tree that describes the wanted-for occurrences on an appropriate level of precision.

The problems are introduced in Sections 3.1 through 3.10. Section 3.11 relates the instance relations of the problems together. Solving the problems is considered in Chapter 4.

3.1 Unordered tree inclusion

Tree inclusion problems are defined via embeddings between trees. Let $P = (V, E, root(P))$ and $T = (W, F, root(T))$ be trees. An injective function f from the nodes of P to the nodes of T is an *embedding* of P into T, if it preserves labels and ancestorship. That is, for all nodes u and v of P we require that

1. $f(u) = f(v)$ if and only if $u = v$,

2. $label(u) = label(f(u))$, and

3. u is an ancestor of v in P if and only if $f(u)$ is an ancestor of $f(v)$ in T.

We say that P is an *unordered included tree* of T if there is an embedding of P in T. Figure 3.1 shows an example of unordered tree inclusion.

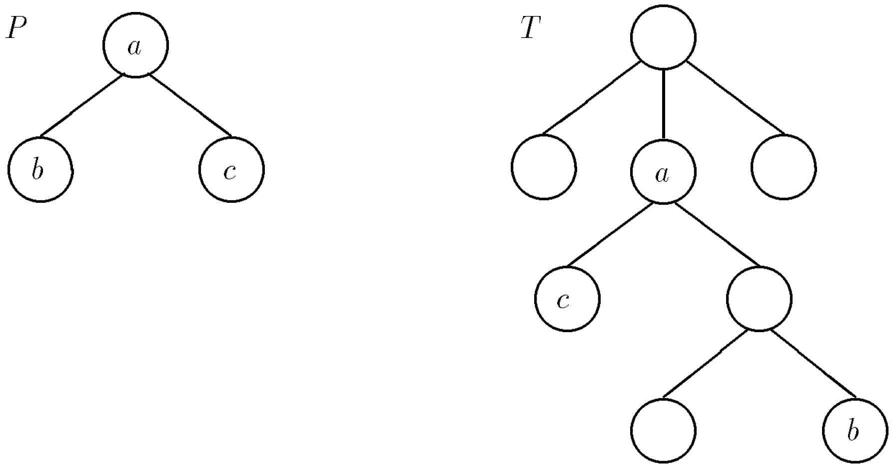


Figure 3.1: Tree P is an unordered included tree of tree T.

Alternative characterizations for the unordered tree inclusion can be given. Intuitively we think that a subset of nodes of the target together with their induced ancestorship relation resembles the pattern. Operationally, an unordered included tree can be obtained from a tree by deleting nodes and permuting siblings.

Example 3.1 The tree $a(b, c(a), d)$ has 42 unordered included trees. They are represented by the terms a, b, c, d, $a(a)$, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(a, b)$, $a(a, d)$, $a(b, a)$, $a(b, c)$, $a(b, d)$, $a(c, b)$, $a(c, d)$, $a(d, a)$, $a(d, b)$, $a(d, c)$, $a(c(a))$, $a(a, b, d)$, $a(a, d, b)$, $a(b, a, d)$, $a(b, c, d)$, $a(b, d, a)$, $a(b, d, c)$, $a(c, b, d)$, $a(c, d, b)$, $a(d, a, b)$, $a(d, b, a)$, $a(d, b, c)$, $a(d, c, b)$, $a(b, c(a))$, $a(c(a), b)$, $a(c(a), d)$, $a(d, c(a))$, $a(b, c(a), d)$, $a(b, d, c(a))$, $a(c(a), b, d)$, $a(c(a), d, b)$, $a(d, b, c(a))$, and $a(d, c(a), b)$. $\square$

If P is an unordered included tree of T, then P is an unordered included tree of every tree that is rooted by an ancestor of $root(T)$. Therefore it is reasonable to consider trees that include P *minimally*. We say that tree T *includes* P *minimally* if T includes P but no proper subtree of T does.

Problem 1 (Unordered tree inclusion problem, UTI)
Given a pattern tree P and a target tree T, locate the subtrees of T that include P minimally. □

A mapping f from the nodes of P to the nodes of T is *root preserving*, if $f(root(P)) = root(T)$. It is immediate that a tree T *includes* P *minimally* if and only if there is an embedding of P in T and every embedding of P in T is root preserving.

3.2 Ordered tree inclusion

The second inclusion problem, the *ordered tree inclusion problem*, results from the previous one by fixing the left-to-right order of nodes. That is, ordered tree inclusion allows us to locate subtrees of the target that contain nodes that agree with the nodes of the pattern with regard to labeling, ancestorship relation, and the left-to-right order induced by the target.

An embedding of a tree P into a tree T is an *ordered embedding* of P into T, if it preserves the left-to-right-order of nodes. That is, for all nodes u and v of pattern P we have that

$$post(u) < post(v) \text{ if and only if } post(f(u)) < post(f(v)) \ .$$

Since f preserves the ancestorship, the above condition is by Lemma 2.1 equivalent to requiring that

$$pre(u) < pre(v) \text{ if and only if } pre(f(u)) < pre(f(v)) \ .$$

Tree P is an *ordered included tree* of T, and T is an *ordered including tree* of P, if there is an ordered embedding of P into T. For an example of ordered tree inclusion see Figure 3.2.

Again, the instance relation can also be characterized operationally: tree P is an ordered included tree of T, if P can be obtained from T by deleting nodes.

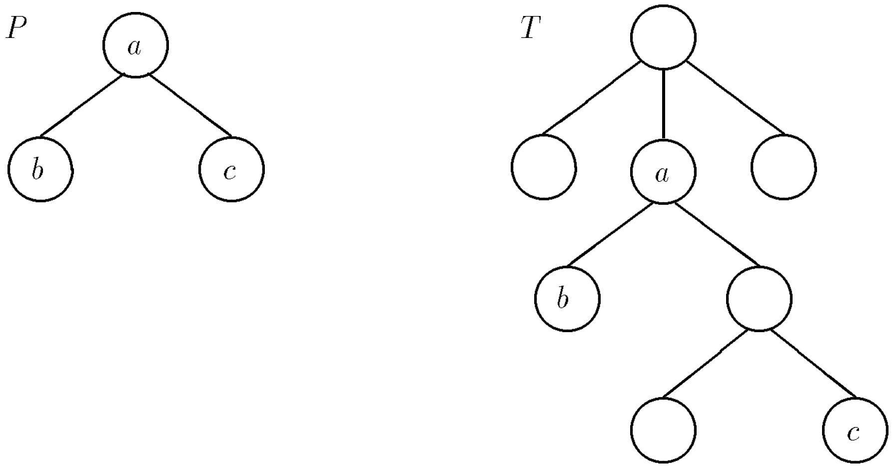


Figure 3.2: Tree P is an ordered included tree of tree T.

Example 3.2 The tree $a(b, c(a), d)$ has 20 ordered included trees. They are a, b, c, d, $a(a)$, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(a, d)$, $a(b, a)$, $a(b, c)$, $a(b, d)$, $a(c, d)$, $a(c(a))$, $a(b, a, d)$, $a(b, c, d)$, $a(b, c(a))$, $a(c(a), d)$, and $a(b, c(a), d)$. □

A tree T *includes tree P with order minimally* if P is an ordered included tree of exactly one subtree of T, namely T itself.

Problem 2 (Ordered tree inclusion problem, OTI)
Given a pattern tree P and a target tree T, locate the subtrees of T that include P with order minimally. □

Again it is immediate that U includes tree P with order minimally if and only if there is an ordered embedding of P into U and every ordered embedding of P into U is root preserving. It is obvious from the definitions that if P is an ordered included tree of T, then P is also an *un*ordered included tree of T. On the other hand, a tree T can include P with order minimally and have a proper subtree that includes P without order minimally: Consider trees $P = a(b, c)$ and $T = a(b, a(c, b))$. Now T includes P with order minimally, but the subtree $a(c, b)$ of T includes P without order minimally.

The ordered tree inclusion problem appears in Exercise 2.3.2-22 of [Knu69]; in the solution Knuth gives a sufficient condition for the existence of an ordered embedding.

3.3 Unordered path inclusion

An embedding f of a tree $P = (V, E, root(P))$ in a tree $T = (W, F, root(T))$ is a *path embedding* if it preserves the parent relation. That is, for all nodes u and v,

$$(u, v) \in E \text{ if and only if } (f(u), f(v)) \in F \ .$$

Pattern tree P is an *unordered path-included tree* of target tree T if there is a path embedding of P in T. For an example see Figure 3.3. Intuitively, a path-included tree of T consists of paths originating from a single node of T.

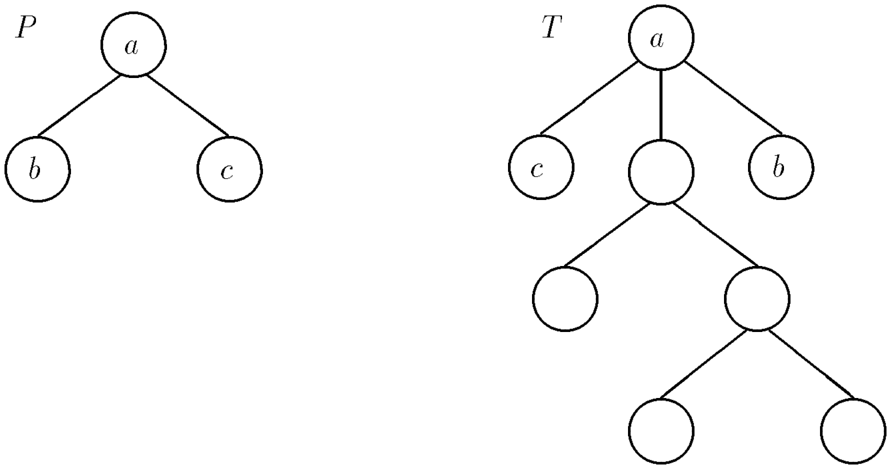


Figure 3.3: Tree P is an unordered path-included tree of tree T.

Let $G = (V, E)$ and $H = (W, F)$ be two directed and labeled graphs. A bijection $f : V \to W$ that satisfies $label(v) = label(f(v))$ for all $v \in V$, and where $(u, v) \in E$ if and only if $(f(u), f(v)) \in F$, is an *isomorphism* between G and H. The graphs are *isomorphic* if there is an isomorphism between them. A graph $G' = (V', E')$ is a *subgraph* of G, if $V' \subseteq V$ and $E' \subseteq E$. Now a tree P is an unordered path-included tree of a tree T if and only if P is isomorphic to a subgraph of T.

Operationally, the unordered path-included trees of T can be obtained from the subtrees of T by cutting and permuting siblings.

Example 3.3 The tree $a(b, c(a), d)$ has 31 unordered path-included trees. They are a, b, c, d, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(b, c)$, $a(b, d)$, $a(c, b)$, $a(c, d)$,

$a(d,b)$, $a(d,c)$, $a(c(a))$, $a(b,c,d)$, $a(b,d,c)$, $a(c,b,d)$, $a(c,d,b)$, $a(d,b,c)$, $a(d,c,b)$, $a(b,c(a))$, $a(c(a),b)$, $a(c(a),d)$, $a(d,c(a))$, $a(b,c(a),d)$, $a(b,d,c(a))$, $a(c(a),b,d)$, $a(c(a),d,b)$, $a(d,b,c(a))$, and $a(d,c(a),b)$. $\square$

As a search problem, we do not want to locate all unordered path including trees; instead, we limit ourselves again to root preserving embeddings.

Problem 3 (Unordered path inclusion, UPI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that there is a root preserving path embedding of P in U. $\square$

It is again immediate from the definitions that if P is an unordered path-included tree of T, then P is also an unordered included tree of T.

3.4 Ordered path inclusion

An ordered embedding f of a tree $P = (V, E, root(P))$ in a tree $T = (W, F, root(T))$ is an *ordered path embedding* if it preserves the parent relation. Tree P is an *ordered path-included tree* of tree T, if there is an ordered path embedding of P in T. Figure 3.4 shows an example of ordered path inclusion.

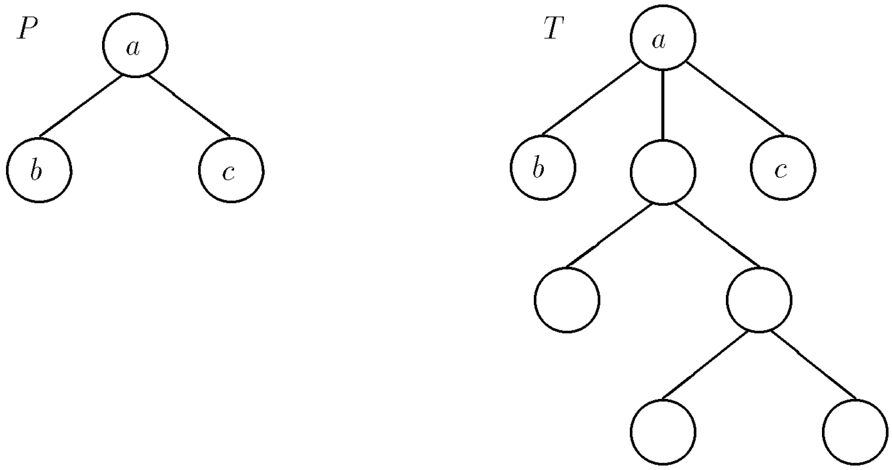


Figure 3.4: Tree P is an ordered path-included tree of tree T.

Intuitively, an ordered path-included tree P of T consists of paths in T having a common start node, and the left-to-right order of the nodes are the same in P and in T. Operationally, the ordered path-included trees of T can be obtained from the subtrees of T by cutting.

Example 3.4 The tree $a(b, c(a), d)$ has 16 ordered path-included trees. They are a, b, c, d, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(b, c)$, $a(b, d)$, $a(c, d)$, $a(c(a))$, $a(b, c, d)$, $a(b, c(a))$, $a(c(a), d)$, and $a(b, c(a), d)$. □

As a search problem, we do not want to locate all ordered path including trees; instead, we restrict ourselves again to root preserving embeddings.

Problem 4 (Ordered path inclusion, OPI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that there is an ordered root preserving path embedding of P in U. □

It is again immediate from the definitions that if P is an ordered path-included tree of T, then P is also an unordered path-included tree of T.

3.5 Unordered region inclusion

Let $P = (V, E, root(P))$ and $T = (W, F, root(T))$ be trees, and let f be a path embedding of P in T. Denote by $f(V)$ the range of f. Embedding f is a *region embedding* of P in T, if whenever $u, v \in f(V)$, and v is a right sibling of u, all nodes of T that are right siblings of u *and* left siblings of v belong to $f(V)$. Pattern tree P is an *unordered region-included tree* of target tree T, if there is a region embedding of P in T. For an example see Figure 3.5.

Intuitively, an unordered region-included tree of T is an integral region of T where the left-to-right order inside the region is irrelevant. Operationally, the unordered region-included trees of T can be obtained from the subtrees of T by first repeatedly cutting at some leftmost and rightmost children, and then permuting the remaining siblings.

Example 3.5 The tree $a(b, c(a), d)$ has 29 unordered region-included trees. They are a, b, c, d, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(b, c)$, $a(c, b)$, $a(c, d)$, $a(d, c)$, $a(c(a))$, $a(b, c, d)$, $a(b, d, c)$, $a(c, b, d)$, $a(c, d, b)$, $a(d, b, c)$, $a(d, c, b)$, $a(b, c(a))$, $a(c(a), b)$, $a(c(a), d)$, $a(d, c(a))$, $a(b, c(a), d)$, $a(b, d, c(a))$, $a(c(a), b, d)$, $a(c(a), d, b)$, $a(d, b, c(a))$, and $a(d, c(a), b)$. □

In the search problem, we restrict ourselves again to root preserving embeddings.

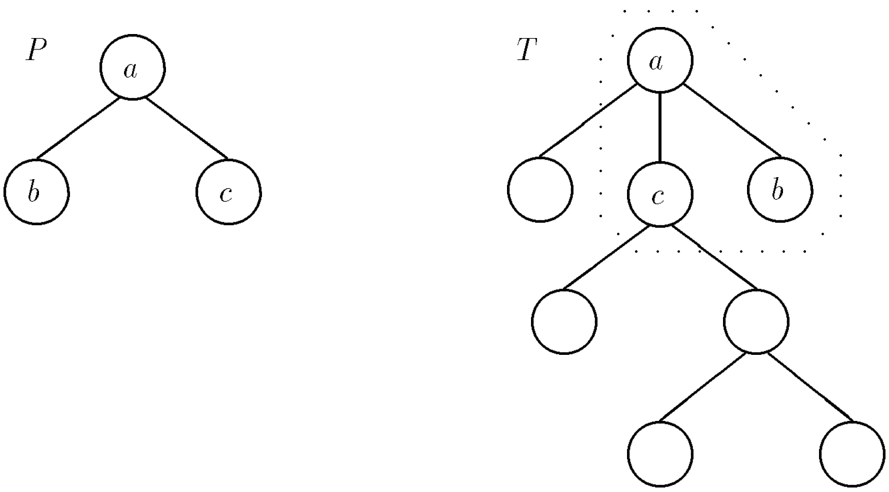


Figure 3.5: Tree P is an unordered region-included tree of tree T.

Problem 5 (Unordered region inclusion, URI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that there is a root preserving region embedding of P in U. $\square$

It is obvious from the definition that an unordered region-included tree of T is also an unordered path-included tree of T.

3.6 Ordered region inclusion

Pattern tree P is an *ordered region-included tree* of target tree T, if there is an *ordered region embedding* of P in T, i.e., an embedding that preserves the parent relation and the order and adjacency of siblings. For an example of ordered region inclusion see Figure 3.6.

Intuitively, an ordered region-included tree of T is an integral region of T. Operationally, the ordered region-included trees of T can be obtained from the subtrees of T by cutting repeatedly at some leftmost and rightmost children.

Example 3.6 The tree $a(b, c(a), d)$ has 15 ordered region-included trees. They are a, b, c, d, $a(b)$, $a(c)$, $a(d)$, $c(a)$, $a(b, c)$, $a(c, d)$, $a(c(a))$, $a(b, c, d)$, $a(b, c(a))$, $a(c(a), d)$, and $a(b, c(a), d)$. $\square$

We restrict the search problem again to root preserving embeddings.

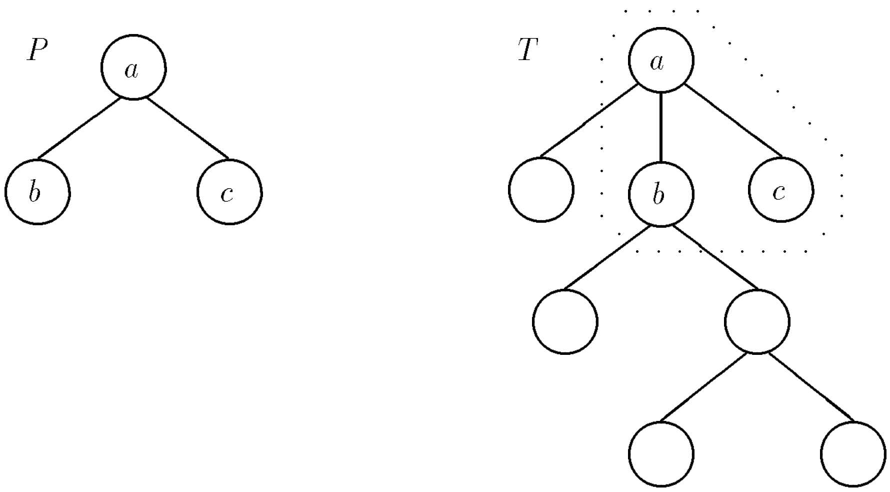


Figure 3.6: Tree P is an ordered region-included tree of tree T.

Problem 6 (Ordered region inclusion, ORI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that there is a root preserving ordered region embedding of P in U. $\Box$

It is evident that an ordered region-included tree of T is also an unordered region-included tree of T.

3.7 Unordered child inclusion

Let P and T be trees. A path embedding f of P in T is a *child embedding*, if it preserves the number of children of the internal nodes. That is, whenever a node u of P has k children, $k > 0$, then $f(u)$ has also k children. Tree P is an *unordered child-included tree* of T, if there is a root preserving child embedding of P in T. Tree P is an *unordered child-included subtree* of T, if P is an unordered child-included tree of a subtree of T, or equivalently, if there is a child embedding of P in T. See Figure 3.7 for an example of unordered child-included subtree.

Operationally, the unordered child-included subtrees of T can be obtained from the subtrees of T by pruning at some nodes, and permuting siblings.

Example 3.7 The tree $a(b, c(a), d)$ has 17 unordered child-included subtrees. They are a, b, c, d, $c(a)$, $a(b, c, d)$, $a(b, d, c)$, $a(c, b, d)$, $a(c, d, b)$,

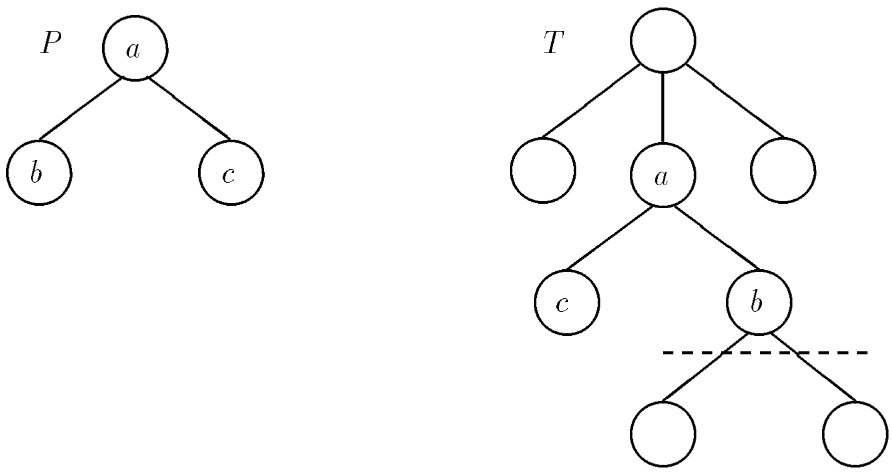


Figure 3.7: Tree P is an unordered child-included subtree of tree T.

$a(d, b, c)$, $a(d, c, b)$, $a(b, c(a), d)$, $a(b, d, c(a))$, $a(c(a), b, d)$, $a(c(a), d, b)$, $a(d, b, c(a))$, and $a(d, c(a), b)$. □

Problem 7 (Unordered child inclusion, UCI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that P is an unordered child-included tree of U. □

Obviously every child embedding is a region embedding and therefore every unordered child-included subtree of T is also an unordered region-included tree of T.

3.8 Ordered child inclusion

A tree P is an *ordered child-included tree* of a tree T if there is an ordered root preserving child embedding of P in T, and P is an *ordered child-included subtree* of T if there is an ordered child embedding of P in T. If f is an ordered child embedding of P in T and u is an internal node of P with k children, then $f(u)$ has also k children, and $f(\mathit{child}(u, i)) = \mathit{child}(f(u), i)$ for all $i = 1, \ldots, k$. For an example see Figure 3.8.

An ordered child-included tree of T can be considered to be a simplified representation of the concept represented by T, where the subconcepts represented by the descendants of pruned nodes have been abstracted away.

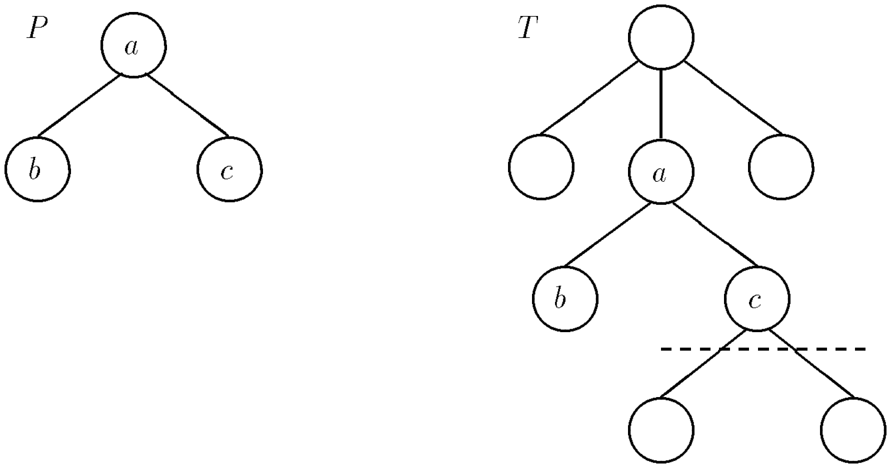


Figure 3.8: Tree P is an ordered child-included subtree of tree T.

Operationally, the ordered child-included subtrees of T can be obtained from the subtrees of T by pruning.

Example 3.8 The tree $a(b, c(a), d)$ has 7 ordered child-included subtrees. They are a, b, c, d, $c(a)$, $a(b, c, d)$, and $a(b, c(a), d)$. □

Problem 8 (Ordered child inclusion, OCI)
Given a pattern tree P and a target tree T, locate the subtrees U of T such that P is an ordered child-included tree of U. □

Obviously every ordered child-included tree of T is also an unordered child-included tree of T.

The ordered child-included subtree problem is usually called the *tree pattern matching* problem. Tree pattern matching has many applications, especially in the implementation of rewriting systems. For this reason, it is the most extensively studied of the inclusion problems presented here. (See for example [HO82].)

3.9 Unordered subtree problem

A tree P is an *unordered subtree* of a tree T, if P is isomorphic to a subtree of T. Operationally, the unordered subtrees of a tree T are obtained from the subtrees of T by permuting siblings. For an example see Figure 3.9.

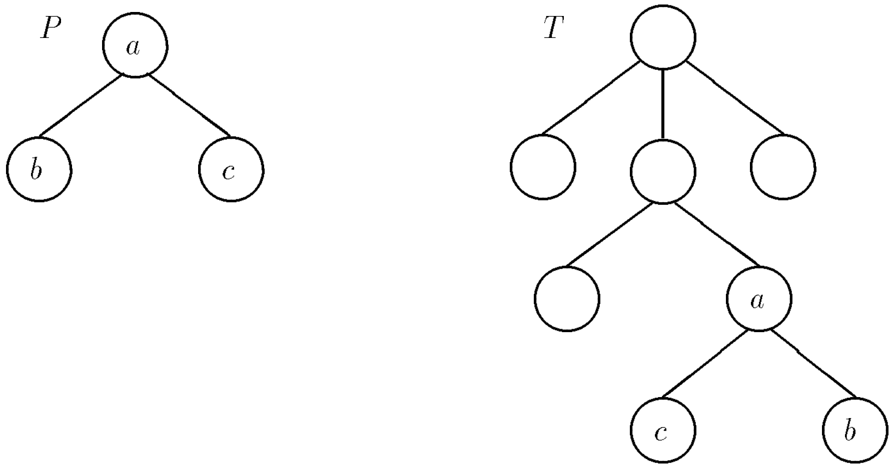


Figure 3.9: Tree P is an unordered subtree of tree T.

Example 3.9 The tree $a(b, c(a), d)$ has 10 unordered subtrees. They are a, b, d, $c(a)$, $a(b, c(a), d)$, $a(b, d, c(a))$, $a(c(a), b, d)$, $a(c(a), d, b)$, $a(d, b, c(a))$, and $a(d, c(a), b)$. □

Problem 9 (Unordered subtree problem, UST)
Given a pattern tree P and a target tree T, locate the subtrees U of T that are isomorphic to P. □

An isomorphism between P and a subtree of T is a child embedding of P in T, and so every unordered subtree of T is also an unordered child-included tree of T. We say that an isomorphism between P and a subtree of T is a *subtree embedding* of P in T just to streamline the terminology.

The unordered subtree problem is sometimes erroneously mixed with the unordered path inclusion problem, as was noted in [Dub90].

3.10 Ordered subtree problem

Let f be an isomorphism between two trees P and U. If f preserves the left-to-right order of the nodes of P we say that f is an *ordered isomorphism* of P in U. If there is an ordered isomorphism between two trees we say that the trees are identical. In order to streamline the terminology we call an ordered isomorphism between P and a subtree of T also an *ordered subtree embedding* of P in T.

The *ordered subtree problem* is the most restricted one of the tree inclusion problems: we search for subtrees of the target that are identical with the pattern. Figure 3.10 shows an instance of the ordered subtree problem.

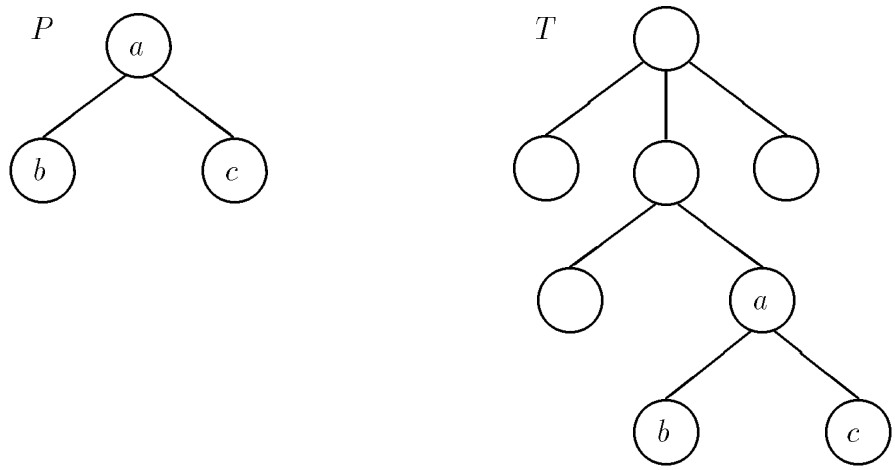


Figure 3.10: Tree T contains a subtree that is identical to tree P.

A tree with n nodes has n ordered subtrees; however some of these may be identical to each other.

Example 3.10 The tree $a(b, c(a), d)$ has 5 ordered subtrees. They are a, b, d, $c(a)$, and $a(b, c(a), d)$. □

Problem 10 (Ordered subtree problem, OST)
Given a pattern tree P and a target tree T, locate the subtrees U of T that are identical to P. □

It is trivial that ordered subtrees of T are also *un*ordered subtrees of T.

3.11 Relating various inclusion problems

After introducing the problems, let us restate the inclusions of the instance relations that were notified in the previous sections. First, the unordered problems form a chain, in which each problem is a special case of the previous one:

Theorem 3.11 Let T be a tree. Then the following inclusions between the sets of unordered included trees of T hold:

1. The set of unordered subtrees of T is a subset of the set of unordered child-included subtrees of T.
2. The set of unordered child-included subtrees of T is a subset of the set of unordered region-included trees of T.
3. The set of unordered region-included trees of T is a subset of the set of unordered path-included trees of T.
4. The set of unordered path-included trees of T is a subset of the set of unordered included trees of T.

$\square$

The corresponding result holds for the ordered problems.

Theorem 3.12 Let T be a tree. Then the following inclusions between the sets of ordered included trees of T hold:

1. The set of ordered subtrees of T is a subset of the set of ordered child-included subtrees of T.
2. The set of ordered child-included subtrees of T is a subset of the set of ordered region-included trees of T.
3. The set of ordered region-included trees of T is a subset of the set of ordered path-included trees of T.
4. The set of ordered path-included trees of T is a subset of the set of ordered included trees of T.

$\square$

Finally, each ordered problem is a special case of the corresponding unordered problem:

Theorem 3.13 Let T be a tree. Then the following inclusions between the sets of ordered and unordered included trees of T hold:

1. The set of ordered subtrees of T is a subset of the set of unordered subtrees of T.

2. The set of ordered child-included subtrees of T is a subset of the set of unordered child-included trees of T.

3. The set of ordered region-included trees of T is a subset of the set of unordered region-included trees of T.

4. The set of ordered path-included trees of T is a subset of the set of unordered path-included trees of T.

5. The set of ordered included trees of T is a subset of the set of unordered included trees of T.

$\square$

Chapter 4

Solving tree inclusion problems

In this chapter we study algorithms for the tree inclusion problems presented in the previous chapter.

Section 4.1 presents a general algorithm schema that can be modified to solve many of the specialized inclusion problems, and some mathematical results that help analyzing variations of the schema. The time complexities of the algorithms are expressed in terms of m and n, where m is the number of nodes in the pattern, and n the number of nodes in the target. Section 4.3 presents algorithms for solving the unordered tree inclusion problem. The time required by the algorithms is superpolynomial in m; this seems almost inevitable, since in Section 4.2 we show that the unordered tree inclusion problem is NP-complete.

Section 4.4 considers solving a group of unordered inclusion problems whose solving and complexity are closely related to computing matchings in bipartite graphs. Those problems are unordered path inclusion, unordered region inclusion, and unordered child inclusion. The corresponding ordered problems, together with ordered tree inclusion, form a group with a common $O(mn)$ upper bound complexity and a conjectured non-linear worst-case lower bound complexity.[1] Solving ordered tree inclusion efficiently is based on *left embeddings*, which are introduced in Section 4.5. Section 4.6 then presents an algorithm for ordered tree inclusion based on left embeddings. Another algorithm for ordered tree inclusion that requires in practical

[1]We are mainly concerned with the worst case running time complexities. The following notation is used for asymptotics: If f and g are functions on nonnegative variables $m, n, \ldots$ we write $f = O(g)$ if there is a constant c such that $f(m, n, \ldots) \leq cg(m, n, \ldots)$ for all sufficiently large values of $m, n, \ldots$. We write $f = \Omega(g)$ if $g = O(f)$, and $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$. If $\lim_{m\to\infty, n\to\infty, \ldots} \frac{f}{g} = 0$, we may write $f = o(g)$.

situations substantially less space is then presented in Section 4.7. Solving ordered path inclusion problems is discussed in Section 4.8. We show there that ordered path inclusion and ordered region inclusion are not easier problems than string matching with don't care symbols, for which no linear time algorithm is known.

The ordered child inclusion problem, i.e., the widely studied classical tree pattern matching differs from the other $O(mn)$ problems, since it has recently been shown to be solvable in $o(mn)$ time. Methods for solving classical tree pattern matching are discussed in Section 4.9. Finally, the easiest group of problems solvable in linear time consists of the subtree problems. This is shown in Section 4.10 as an application of a general condition for tree matching problems to be solvable in linear time.

Section 4.11 summarizes what is known of the computational complexities of the tree inclusion problems.

4.1 A general solution scheme

Solving tree inclusion problems leads us to considering embeddings and inclusions between forests. The definitions for embeddings between forests are similar to the corresponding definitions of embeddings between trees. Let $F = \langle P_1, \ldots, P_k \rangle$ and $G = \langle T_1, \ldots, T_l \rangle$ be forests. An embedding f of F in G is *root preserving* if $f(root(P_i)) \in \{root(T_1), \ldots, root(T_l)\}$ for all $i = 1, \ldots, k$. We say that

1. F is an *included forest* of G if there is an embedding of F in G.
2. F is an *ordered included forest* of G if there is an ordered embedding of F in G.
3. F is a *path-included forest* of G if there is a path embedding of F in G.
4. F is an *ordered path-included forest* of G if there is an ordered path embedding of F in G.
5. F is a *region-included forest* of G if there is a region embedding of F in G.
6. F is an *ordered region-included forest* of G if there is an ordered region embedding of F in G.

Each of the above relations between F and G is denoted by $F \sqsubseteq G$; the meaning of the notation should always be clear from the context.

The following lemma is easy to prove from the definitions of embeddings.

Lemma 4.1 For each of the *unordered* interpretations above, the relation $\sqsubseteq$ is a partial ordering, i.e., it is reflexive, transitive, and antisymmetric up to isomorphism. That is, for all forests F, G and H,

1. $F \sqsubseteq F$,
2. If $F \sqsubseteq G$ and $G \sqsubseteq H$, then $F \sqsubseteq H$, and
3. If $F \sqsubseteq G$ and $G \sqsubseteq F$, then F and G are isomorphic.

For each of the *ordered* interpretations, the relation is a partial ordering up to identity. (As above, but $F \sqsubseteq G$ and $G \sqsubseteq F$ imply that F and G are identical.) $\square$

The equality of root labels is a necessary condition for the existence of any root preserving embedding between trees. The following lemma contains the basic idea of the algorithms that search for root preserving embeddings of a pattern in the subtrees of the target.

Lemma 4.2 Let P and T be trees with $label(root(P)) = label(root(T))$, and let the immediate subtrees of P be $P_1, \ldots, P_k$ and the immediate subtrees of T be $T_1, \ldots, T_l$.

1. There is a root preserving embedding of P in T if and only if the forest $\langle P_1, \ldots, P_k \rangle$ is an included forest of $\langle T_1, \ldots, T_l \rangle$.
2. There is a root preserving ordered embedding of P in T if and only if $\langle P_1, \ldots, P_k \rangle$ is an ordered included forest of $\langle T_1, \ldots, T_l \rangle$.
3. There is a root preserving path embedding of P in T if and only if there is a root preserving path embedding of $\langle P_1, \ldots, P_k \rangle$ in $\langle T_1, \ldots, T_l \rangle$.
4. There is a root preserving ordered path embedding of P in T if and only if there is a root preserving ordered path embedding of $\langle P_1, \ldots, P_k \rangle$ in $\langle T_1, \ldots, T_l \rangle$.
5. There is a root preserving region embedding of P in T if and only if there is a root preserving region embedding of $\langle P_1, \ldots, P_k \rangle$ in $\langle T_1, \ldots, T_l \rangle$.

6. There is a root preserving ordered region embedding of P in T if and only if there is a root preserving ordered region embedding of $\langle P_1, \ldots, P_k \rangle$ in $\langle T_1, \ldots, T_l \rangle$. □

The algorithms for solving various tree inclusion problems can be presented as variations of a single dynamic programming scheme. The scheme is parameterized by a *match predicate* ψ that expresses the required relation between the occurrences of the immediate subtrees of the pattern.

The generic algorithm refers to the nodes by their postorder numbers. The algorithm consists of two loops that process the nodes of the target and the nodes of the pattern in ascending postorder. This means that when two nodes v and w are compared, the proper subtrees of $P[v]$ and $T[w]$ have already been tested against each other. The algorithm scheme uses an array M, where for each target node w the entry $M(w)$ will contain the set of the pattern nodes that match at w.

Algorithm 4.3 General tree matching scheme.

Input: Pattern tree $P = (V, E, root(P))$ and target tree $T = (W, F, root(T))$.

Output: The nodes of W that are occurrences of P.

Method:

for $w := 1, \ldots, n$ **do**
comment: Process the target nodes in postorder;
    $M(w) := \emptyset$;
    for all $v := 1, \ldots, m$ **do**
    comment: Process the pattern nodes in postorder;
        if $label(v) = label(w)$ **then**
            Let $v_1, \ldots, v_k$ be the children of v;
            if there are nodes $w_1, \ldots, w_k \in desc(w)$
            such that $v_i \in M(w_i)$ for $i = 1, \ldots, k$
            and $\psi(w, w_1, \ldots, w_k)$ holds **then**
                $M(w) := M(w) \cup \{v\}$;
            fi;
        fi;
    od;
    if $root(P) \in M(w)$ **then**
    comment: an occurrence found;

```
        output w;
    fi;
od;
```

Algorithm 4.3 can easily be modified to output only the roots of the subtrees that include the pattern minimally. This is achieved by marking the ancestors of each occurrence in the target, and by considering only non-marked target nodes in the outer loop of the algorithm.

The time analysis of the variations of the above schema often leads to expressions of the form

$$\sum_{w \in W} \sum_{v \in V} g(|w|) f(|v|) \ ,$$

where $|w|$ ($|v|$) denotes the number of children of node w (of node v), and f and g are convex functions. In order to facilitate simplifying these expressions, we present the following lemmas.

A function f defined on a closed real interval $[a, b]$ is *convex* if for all $x, y \in [a, b]$ and all $\alpha \in]0, 1[$

$$f(z) \leq \alpha f(x) + (1 - \alpha) f(y) \ , \tag{4.1}$$

when $z = \alpha x + (1 - \alpha) y$. The geometric interpretation of (4.1) is simple: When $x < y$, the graph of f in $[x, y]$ stays below the line segment from $(x, f(x))$ to $(y, f(y))$.

Lemma 4.4 Let f be a convex function that is defined on a real interval $I = [0, b]$, such that $f(0) = 0$. Let $a_1, \ldots, a_n$ be numbers in interval I such that $\sum_{i=1}^{n} a_i \in I$. Then

$$\sum_{1 \leq i \leq n} f(a_i) \leq f(\sum_{1 \leq i \leq n} a_i) \ . \tag{4.2}$$

Proof. Let u, v, and $u + v$ be numbers in $]0, b]$. By assigning in (4.1) 0 for x, $u + v$ for y, u for z, and $v/(u + v)$ for α we get

$$f(u) \leq \frac{u}{u + v} f(u + v) \ . \tag{4.3}$$

Similarly, by substituting v for z and $u/(u + v)$ for α, we get

$$f(v) \leq \frac{v}{u + v} f(u + v) \ . \tag{4.4}$$

From (4.3) and (4.4) we get

$$f(u) + f(v) \leq f(u + v) \ .$$

From this, (4.2) is obtained by induction. □

The following lemma gives a useful test for recognizing many convex functions. For a proof see any book on calculus.

Lemma 4.5 Assume that f is a real-valued function that is defined on a closed interval $[a, b]$, and that is differentiable on $]a, b[$. If the first derivative of f is increasing on $]a, b[$, then f is convex on $[a, b]$. In particular, f is convex if its second derivative exists and is nonnegative in $]a, b[$. □

4.2 Unordered tree inclusion is NP-complete

The existence of an efficient general algorithm for unordered tree inclusion is highly improbable, since the problem appears to be NP-complete.

Tree inclusion problems are special cases of the *minor containment* problem for graphs [RS86, Joh87]. In that problem, given two graphs $G = (V, E)$ and $H = (U, F)$, one has to decide whether G contains H as a *minor*, i.e., is there a subgraph of G that can be converted to H by a sequence of *contractions*. In a contraction, two adjacent vertices and an edge between them are replaced by a single new vertex. All other edges previously incident on either contracted vertex become incident to the new vertex. For trees, a contraction is equivalent to the deletion of a node.

Minor containment is known to be NP-complete even for unrooted trees, when both H and G are given as inputs. (For every *fixed* planar graph H, and therefore for every fixed tree and forest H, there is a polynomial time algorithm for testing whether H is a minor of a given graph G [RS86, Joh87].) This implies that minor containment is NP-complete also for rooted trees, or in our framework that unordered tree inclusion is NP-complete. The proofs of these results have not been published; we prove the NP-completeness of unordered tree inclusion by a reduction from the basic NP-complete problem satisfiability [Coo71, GJ79].

Problem 11 (Satisfiability)
Given a finite collection of clauses C over a finite set of Boolean variables U, decide whether or not there is a satisfying truth assignment for C. □

For proving the NP-completeness of unordered tree inclusion we use the following lemma stating that a slight restriction of satisfiability is still NP-complete.

Lemma 4.6 Let $U = \{u_1, \ldots, u_n\}$ be a set of Boolean variables and $C = \{c_1, \ldots, c_m\}$ be a collection of clauses over U. Now C can be transformed in polynomial time into a collection of clauses $C' = \{c'_1, \ldots, c'_{m'}\}$ such that

1. C is satisfiable if and only if C' is satisfiable, and
2. no negated variable occurs in two clauses of C'.

Proof. Let $D = \{\{u, y_u\}, \{\bar{u}, \bar{y}_u\} \mid u \in U\}$, where y_u is a new variable for each $u \in U$. Clauses $\{u, y_u\}$ and $\{\bar{u}, \bar{y}_u\}$ express exclusive-or of u and y_u, i.e., a truth assignment satisfies them if and only if it assigns opposite values to u and y_u. Let E be the set of clauses obtained from C by replacing each negated occurrence $\bar{u}$ of a variable by y_u, and let $C' = D \cup E$. Now C' is satisfiable if and only if C is satisfiable. The transformation can obviously be done in polynomial time. $\square$

Theorem 4.7 [KM91a] Unordered tree inclusion is an NP-complete problem.

Proof. It is easy to see that all the tree inclusion problems presented in Chapter 3 are in NP: an algorithm can guess a mapping from the pattern nodes to the target nodes and check in polynomial time that it is indeed an embedding.

The completeness for NP is shown by a reduction from satisfiability. Let an instance of satisfiability be given by a collection of variables $U = \{u_1, \ldots, u_n\}$ and a collection of clauses $C = \{c_1, \ldots, c_m\}$ over U. Lemma 4.6 allows us to assume that no negated variable appears in two clauses of C.

Form a pattern tree P and a target tree T as follows. Let $P = (V_P, E_P, 0)$ be the tree given by nodes $V_P = \{0, \ldots, m\}$ and by parent-child edges

$$E_P = \{(0, x) \mid x \in V_P, x \neq 0\} .$$

Let $label(x) = x$ for all $x \in V_P$. The intuition is that the nodes of tree P, excluding the root, represent clauses of C and each of them is labeled by the index of the corresponding clause. Let $T = (V_T, E_T, (0, 0))$ be the tree whose nodes consist of pairs

$$\begin{aligned} V_T &= \{(0,0)\} \cup \\ &\quad \{(u, j) \mid u \in c_j \in C\} \cup \\ &\quad \{(\bar{u}, j) \mid \bar{u} \in c_j \in C\} , \end{aligned}$$

and whose parent-child edges are

$$\begin{aligned} E_T &= \{((0,0),(\bar{u},j)) \mid \bar{u} \in c_j \in C\} \cup \\ &\quad \{((0,0),(u,j)) \mid u \in c_j \in C, \bar{u} \notin \bigcup C\} \cup \\ &\quad \{((\bar{u},j),(u,k)) \mid \bar{u} \in c_j \in C, u \in c_k \in C\} . \end{aligned}$$

The assumption of unique occurrences of negative literals implies that T is indeed a tree, i.e., each node except the root has a unique parent. Let $label((x,y)) = y$ for all nodes $(x,y) \in V_T$. So, tree T has one node corresponding to each literal occurrence in some clause of C, plus an additional root node $(0,0)$. The nodes corresponding to literal occurrences in clause c_j are labeled by j. A node $v = (\bar{u},j)$ corresponding to the negated occurrence of a variable u is the parent of the nodes corresponding to the positive occurrences of u, and the root is the parent of v. The nodes corresponding to the positive occurrences of a variable that does not occur negated in C are children of the root of T. An example of the construction is shown in Figure 4.1.

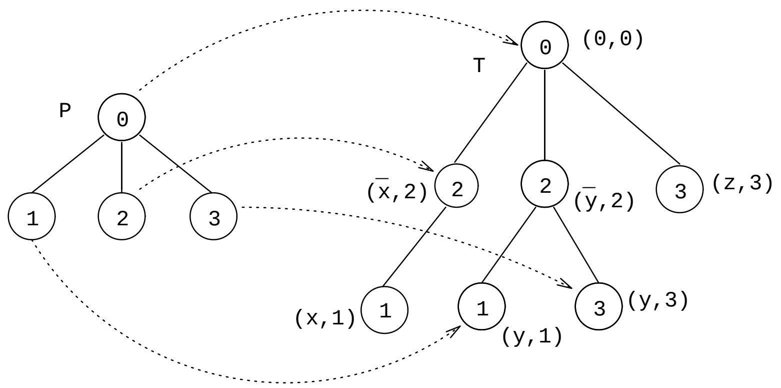


Figure 4.1: Trees for clauses $c_1 = \{x, y\}$, $c_2 = \{\bar{x}, \bar{y}\}$, and $c_3 = \{y, z\}$. The embedding shown by arrows corresponds to satisfying truth assignments that set x to *false* and y to *true*.

Trees P and T can clearly be formed in polynomial time.

Now we claim that there is a satisfying truth assignment for C if and only if P is an unordered included tree of T. First assume that t is a satisfying truth assignment for C. Define a mapping h from the nodes of P onto the nodes of T as follows: For the root of P set $h(0) = (0,0)$ and for other nodes j of P set $h(j) = (l,j)$, where $(l,j) \in V_T$ is some node such that literal l is true under truth assignment t; such a node can be selected because t satisfies at least one literal of every clause $c_j \in C$. Now h is an unordered embedding since it is obviously injective and label preserving, and it cannot map two sibling nodes of P to an ancestor and its descendant in T. Otherwise, by the construction of T, truth assignment t would satisfy both a positive and a negative occurrence of the same variable, which is impossible.

Next assume that h is an unordered embedding of P into T. Set $t(x) =$ *false* if the range of h contains a node $(\bar{x}, j)$ corresponding to a negative occurrence of variable x, and $t(x) =$ *true* if the range of h contains a node (x, j) corresponding to a positive occurrence of variable x. It is easy to see that t is a well defined truth assignment for a subset of variables in C and that t satisfies at least one literal in each clause of C. $\square$

One should not be too disappointed by the above negative result. The next section shows that the apparently unavoidable exponentiality comes from the size of the pattern only; if m, the size of the pattern, is $o(\log\log n)$ the problem is solvable in time $O(n)$.

4.3 An algorithm for unordered tree inclusion

In this section we present an algorithm for solving the unordered tree inclusion problem. The algorithm manipulates match systems consisting of subsets of pattern nodes. In the worst case the size of a match system is exponential in m, i.e., the size of the pattern. The previous section showed that it is unlikely that we can do essentially any better. On the other hand, the exponentiality in m does not cause problems if m is small. We show how to build tables from the pattern in a preprocessing phase. Using these tables the matching can be performed in time that is linear in the size of the target.

The *match system* $S(w)$ for a target node w consists of all the subsets $\{v_1, \ldots, v_k\}$ of pattern nodes such that $\langle P[v_1], \ldots, P[v_k] \rangle$ is an included forest of $T[w]$. The following algorithm computes the match systems for each target node while going through the target in a bottom-up order. Note that if w' is an ancestor of a target node w, then $T[w']$ includes $T[w]$ and therefore $S(w) \subseteq S(w')$. Also, for a pattern node v we have that $\{v\} \in S(w)$ only if *children*$(v) \in S(w)$.

Algorithm 4.8 Unordered tree inclusion algorithm.

Input: Trees $P = (V, E, root(P))$ and $T = (W, F, root(T))$.

Output: The nodes w of T such that there is a root preserving embedding of P in $T[w]$.

Method:

1.   **for** $w := 1, \ldots, n$ **do**
2.   **comment**: Go through the target nodes in postorder;
3. $S := \{\emptyset\}$;
4.   Let $w_1, \ldots, w_l$, $l \geq 0$, be the children of w;
5.   **for** $i := 1, \ldots, l$ **do**
6. $S := \{A \cup B \mid A \in S, B \in S(w_i)\}$;
7.   **od**;
8. $S\Delta := \emptyset$;
9.   **for all** $v \in V$ such that $label(v) = label(w)$ **do**
10.   **if** $children(v) \in S$ **then**
11. $S\Delta := S\Delta \cup \{\{v\}\}$;
12.   **fi**;
13.   **od**;
14.   **if** $\{root(P)\} \in S\Delta$ **then**
15.   **comment**: A match found;
16.   **output** w;
17.   **fi**;
18. $S(w) := S \cup S\Delta$;
19.   **od**;

Algorithm 4.8 computes the match system $S(w)$ for a target node w as follows. Let $w_1, \ldots, w_l$ be the children of w. First the loop on lines 5–7 computes from the match systems of the children of w a set S. The invariant for the loop is central to the correctness of the algorithm. It says that for $i = 0, \ldots, l$ the set S consists of the subsets $\{v_1, \ldots, v_k\}$ of such pattern nodes that $\langle P[v_1], \ldots, P[v_k] \rangle$ is an included forest of the forest $\langle T[w_1], \ldots, T[w_i] \rangle$. For $i = 0$ this is clearly true since then $S = \{\emptyset\}$. Next assume inductively that the invariant holds for $i - 1$ when $0 < i \leq l$, and that $S(w_i)$ is the correct match system for w_i. Now it is rather easy to see that a forest $\langle P[v_1], \ldots, P[v_k] \rangle$ has an embedding in $\langle T[w_1], \ldots, T[w_i] \rangle$ if and only if $\{v_1, \ldots, v_k\} = A \cup B$ for some $A \in S$ and $B \in S(w_i)$.

Next the algorithm computes in a set $S\Delta$ the singleton sets $\{v\}$ of the pattern nodes v for which there is a root preserving embedding of $P[v]$ in $T[w]$; these are exactly those nodes v whose label matches the label of w and whose set of children belongs to S. (Cf. Lemma 4.2.) After this process, $S \cup S\Delta$ is the the match system of w.

Like Algorithm 4.3, also Algorithm 4.8 can easily be modified to report the roots of only those subtrees that include the pattern minimally.

We can restrict S to consist of sets of siblings, since they only affect the insertion of new pattern nodes to the match systems. (See line 10 of the algorithm.) Each match system S is *monotone decreasing*, i.e., if $A \in S$, then $B \in S$ for each $B \subseteq A$. Thus we can represent the match systems by maintaining their maximal elements only. Such a representation is a *Sperner system*, i.e., a system where no set includes another one. We will see in a moment that also the size of a Sperner system on a set of m elements can be exponential in m.

Note that the size of the systems in S depends on P only. If P is fixed, executing line 6 and the loop on lines 9–13 takes constant time. The algorithm examines every target node at most twice; once as w and at most once as a child of w. Therefore we have the following result.

Theorem 4.9 For a fixed pattern P, an instance (P, T) of the unordered tree inclusion problem can be solved in time $O(|T|)$. □

By using preprocessing of the pattern Algorithm 4.8 can be modified to operate in linear time with respect to the size of the target. The preprocessing is based on enumerating the possible values of variable S, and it resembles the preprocessing of the bottom-up tree pattern matching algorithm in [HO82].

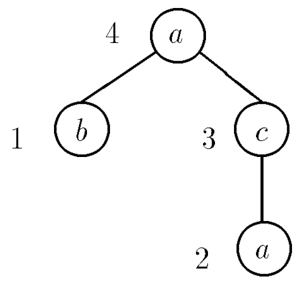
Let $P = (V, E, root(P))$ be the given pattern, and let L be the set of labels. (The labels in P together with a symbol for all other labels is a sufficient representation for L.) In addition to the restrictions above, each match system S is *descendant-closed*. By this we mean that if $u \in A$ for some $A \in S$, then $children(u) \in S$. Denote by $\mathcal{S} = \mathcal{S}(P)$ the collection of the descendant-closed Sperner systems on V such that each set in a system consists of sibling nodes only. We can use an enumeration of $\mathcal{S}$ for representing in arrays two functions $u : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ and $r : \mathcal{S} \times L \to \mathcal{S}$. The value of $u(S_1, S_2)$ is the representation of the element-wise union of S_1 and S_2, that is, the set of the maximal elements in the system

$$\{A \cup B \mid A \in S_1, B \in S_2, A \cup B \text{ is a set of siblings in } P\} .$$

Having this function in an array, executing Line 6 in Algorithm 4.8 takes constant time. The loop on lines 9–13 and line 18 can be replaced by a single assignment if we precompute the function $r : \mathcal{S} \times L \to \mathcal{S}$, where

$$r(S, l) = S \cup \{\{v\} \mid v \in V,\ label(v) = l,\ children(v) \in S,\ v \notin \bigcup S\}\ .$$

As an example, consider the following pattern P.



Now the systems of $\mathcal{S}(P)$ can be enumerated as follows.

1: $\{\emptyset\}$
2: $\{\{1\}\}$
3: $\{\{2\}\}$
4: $\{\{1\}, \{2\}\}$
5: $\{\{2\}, \{3\}\}$
6: $\{\{1\}, \{2\}, \{3\}\}$
7: $\{\{2\}, \{1, 3\}\}$
8: $\{\{2\}, \{4\}, \{1, 3\}\}$

Note that for example the system $\{\{1\}, \{3\}\}$ is excluded, because it is not descendant-closed; node 2, the child of node 3, is missing. System number 8 stands for an occurrence of P, since it contains 4, the root of P.

The element-wise union of the match systems can be performed using the following table u. Only the upper right corner of the table is filled, since the table is symmetric. For example, combining element-wise the systems 5 and 6, i.e., the systems $\{\{2\}, \{3\}\}$ and $\{\{1\}, \{2\}, \{3\}\}$ gives the system $\{\{2\}, \{1, 3\}\}$, because nodes 1 and 3 are the only siblings in P. The number of the result system is 7.

u	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2		2	4	4	7	7	7	8
3			3	4	5	6	7	8
4				4	7	7	7	8
5					5	7	7	8
6						7	7	8
7							7	8
8								8

The table r is in this case as follows (with x standing for labels not appearing in the pattern):

r	a	b	c	x
1	3	2	1	1
2	4	2	2	2
3	3	4	5	3
4	4	4	6	4
5	5	6	5	5
6	6	6	6	6
7	8	7	7	7
8	8	8	8	8

For example, $r(7, a) = 8$, because $label(4) = a$, $children(4) = \{1, 3\} \in \{\{2\}, \{1, 3\}\}$, which is the system number 7, and $\{\{2\}, \{4\}, \{1, 3\}\}$ is the system number 8.

The size of table u is $|\mathcal{S}(P)|^2$ and the size of table r is $O(m\,|\mathcal{S}(P)|)$. These bounds suggest that the preprocessing is reasonable only when the size of $\mathcal{S}(P)$ is not too large. The collection $\mathcal{S}(P)$ depends strongly on the form of the pattern P. If P consists of m nodes that form a path from the root to a single leaf, $\mathcal{S}(P)$ consists of $m + 1$ systems, which are $\{\emptyset\}$, $\{\{1\}\}$, $\ldots$, and $\{\{1\}, \ldots, \{m\}\}$. At the other extreme, the size of $\mathcal{S}(P)$ can be doubly exponential in m. Consider a pattern P consisting of a root node and $m - 1$ leaves. Let $\mathcal{A}$ be the system consisting of the sets of $\lfloor (m-1)/2 \rfloor$ leaves of P. Obviously $|\mathcal{A}| = \binom{m-1}{\lfloor (m-1)/2 \rfloor}$, and every subset of $\mathcal{A}$ belongs to $\mathcal{S}(P)$. Therefore

$$|\mathcal{S}(P)| > 2^{\binom{m-1}{\lfloor (m-1)/2 \rfloor}} .$$

Next we present an upper bound for the size of the collection $\mathcal{S}(P)$. The following results give an upper bound for the number of Sperner systems on m nodes.

Proposition 4.10 [Spe28, Bol86] The size of a Sperner system on an m-element set is at most $\binom{m}{\lfloor m/2 \rfloor}$. $\square$

Proposition 4.11 [BDK+91] The number of set systems $\mathcal{A}$ on an m-element set, satisfying

$$|\mathcal{A}| \leq c \binom{m}{\lfloor m/2 \rfloor} \tag{4.5}$$

is at most

$$2^{\frac{c}{2}\binom{m}{\lfloor m/2 \rfloor} \log m (1+o(1))} .$$

$\square$

The second proposition tells that there is an upper bound of the form $2^{f(m)}$ for the number of set systems $\mathcal{A}$ on an m-element set satisfying (4.5), where $f(m)$ is arbitrarily close to $\frac{c}{2}\binom{m}{\lfloor m/2 \rfloor} \log m$ when m is large enough. Putting these results together gives the following upper bound for the number of Sperner systems on an m-element set, denoted by $\alpha(m)$:

$$\alpha(m) \leq 2^{\frac{1}{2}\binom{m}{\lfloor m/2 \rfloor} \log m (1+o(1))} .$$

Because the systems in $\mathcal{S}(P)$ are Sperner systems, this is also an upper bound for the size of $\mathcal{S}(P)$. As discussed above, a pattern consisting of a root node and $m-1$ leaves brings $\mathcal{S}(P)$ fairly close to this bound.

4.4 Solving unordered path inclusion problems

This section considers solving the problems of unordered path inclusion, unordered region inclusion and unordered child inclusion. These problems are easier than unordered tree inclusion. The reason is that for path inclusion it is sufficient to test the immediate subtrees of the pattern against the trees rooted by the *children* of a target node w when testing if the pattern occurs at w. With tree inclusion we have to examine *descendants* of w, which leads to more combinations to embed the immediate subtrees of the pattern and to more complicated dependencies between the occurrences of pattern subtrees. In Section 4.2 we saw that these dependencies allow representing the satisfiability problem of Boolean formulas as a tree inclusion problem.

The problems are again solved using the scheme of Algorithm 4.3 that stores in an array the information of the matches between all pairs of pattern-target nodes. Solving the match predicate that expresses the required relation between the occurrences of the immediate subtrees of the pattern includes in these problems computing a maximum matching in a bipartite graph.

A *matching* in a graph G is a set of edges of G, no two of which share a vertex. A node is *matched* if it is incident to an edge in the matching. A *maximum matching* is a matching with the maximum number of edges. A directed graph $G = (V, E)$ is *bipartite*, if its vertices can be partitioned in two disjoint sets X and Y such that $E \subseteq X \times Y$.

Problem 12 (Bipartite matching)
Given a bipartite graph G, find a maximum matching in G. $\square$

Now we explain how bipartite matching relates to solving unordered path inclusion problems. Let P be a pattern tree with immediate subtrees $P_1, \ldots, P_k$ rooted by the nodes $v_1, \ldots, v_k$, and let T be a target tree with immediate subtrees $T_1, \ldots, T_l$ rooted by the nodes $w_1, \ldots, w_l$. By Lemma 4.2 there is a root preserving path embedding of P in T if and only if the root labels of P and T match and there is a root preserving path embedding of forest $\langle P_1, \ldots, P_k \rangle$ in forest $\langle T_1, \ldots, T_l \rangle$. The latter condition can be solved as a graph matching problem if the matches between the immediate subtrees of P and T have been computed. Let G be the bipartite graph whose vertices are $v_1, \ldots, v_k, w_1, \ldots, w_l$ with an edge from pattern node v_i to target node w_j if and only if $P[v_i]$ has a root preserving path embedding in $T[w_j]$. Then there is a root preserving path embedding of forest $\langle P_1, \ldots, P_k \rangle$ in forest $\langle T_1, \ldots, T_l \rangle$ if and only if G has a maximum matching of k edges.

Bipartite matching algorithms start with an empty matching of the graph and extend it by searching augmenting paths. An *augmenting path* with respect to a matching M in a bipartite graph $G = (X \cup Y, E)$ is a path p from a node x to a node y, where neither x nor y is matched, the first and the last edge in p are in $E \setminus M$, and the edges in p are alternatively in M and in $E \setminus M$. The number of edges of an augmenting path p in $E \setminus M$ is one greater than the number of edges of p in M. Therefore replacing in a matching M all the edges of an augmenting path p that are in M by the edges of p that are in $E \setminus M$ produces a new matching that has one more edge than M. A bipartite matching algorithm stops when no more augmenting paths can be found. The correctness of these algorithms is based on the following proposition.

Proposition 4.12 A matching M in a graph G is maximum if and only if there is no augmenting path with respect to M.

Proof. One direction follows from the above considerations. For the other direction, see for example [PS82] or [vL90]. $\square$

Each augmenting path can be computed in time proportional to the size of the graph, and each of them extends the matching by one edge. The basic bipartite matching algorithm (see, e.g., [PS82]) finds one augmenting path in one search step; therefore its time complexity is $O(s|E|)$, where s is the size of a maximum matching. The algorithm of Hopcroft and Karp finds in one $O(|E|)$ search step a set of augmenting paths, and the number of these search steps is $O(\sqrt{s})$ [HK73]. This result is utilized in the next algorithm.

Lemma 4.13 The bipartite matching problem on a graph $G = (V, E)$ can be solved in time $O(\sqrt{s}\,|E|)$, where s is the size of a maximum matching. $\square$

The unordered path inclusion algorithms store their results in an $m \times n$ Boolean array a; the value of $a(v, w)$ is set to *true* if and only if $P[v]$ has a root preserving embedding (of appropriate type for the problem at hand) in $T[w]$.

The next algorithm for unordered path inclusion was sketched in [Mat68]. A more complete presentation and analysis of the algorithm can be found in [Rey77] and [Chu87]. (See also [vL90].) Using bipartite matching as a subroutine of this algorithm is mentioned also in [HK73].

Algorithm 4.14 [Mat68, Rey77] Unordered path inclusion algorithm.

Input: Trees $P = (V, E, root(P))$ and $T = (W, F, root(T))$.

Output: The nodes w of T such that there is a root preserving path embedding of P in $T[w]$.

Method:

for $w := 1, \ldots, n$ **do**
comment: Go through the target nodes bottom-up;
    Let $w_1, \ldots, w_l$, $l \geq 0$, be the children of w;
    for $v := 1, \ldots, m$ **do**
    comment: Go through the pattern nodes bottom-up;
        $a(v, w) :=$ **false**;

7. Let $v_1, \ldots, v_k$, $k \geq 0$, be the children of v;
8. Let $G = (X \cup Y, E)$, where
$X = \{v_1, \ldots, v_k\}$, $Y = \{w_1, \ldots, w_l\}$, and
$E = \{(x, y) \mid x \in X, y \in Y, a(x, y) = \textbf{true}\}$;
9. **if** $label(v) = label(w)$ **and**
10. the size of a maximum matching of G is k **then**
11. $a(v, w) :=$ **true**;
12. **fi**;
13. **od**;
14. **if** $a(root(P), w) =$ **true then**
15. **comment**: An occurrence found;
16. **output** w;
17. **fi**;
18. **od**;

Again, the algorithm can easily be modified to report only those subtrees of the target that include the pattern minimally. The correctness of Algorithm 4.14 is based on Lemma 4.2 and on observing that forest $\langle P[v_1], \ldots, P[v_k] \rangle$ has a root preserving embedding in forest $\langle T[w_1], \ldots, T[w_l] \rangle$ if and only if a maximum matching of graph G on line 8 has k edges.

The graph G does not have to be explicitly constructed on line 8 of the algorithm. On line 10 we need to compute a maximum matching of size at most $|v|$ in a graph having at most $|v| \times |w|$ edges. (Notation $|u|$ stands for the number of children of a node u.) This can be done by Lemma 4.13 in time $O(\sqrt{|v|}\,|v||w|)$. Therefore the total complexity of Algorithm 4.14 is

$$O(\sum_{w \in W} (1 + \sum_{v \in V} (1 + |v|^{3/2} |w|))) = O(n + nm + \sum_{v \in V} |v|^{3/2} \sum_{w \in W} |w|) \; . \qquad (4.6)$$

By Lemmas 4.4 and 4.5 expression (4.6) is $O(m^{3/2} n)$. (Note that summing the number of children of each node in a tree of n nodes gives the number of edges in the tree, which is $n - 1$.)

The unordered child inclusion problem can be solved by a slight modification of Algorithm 4.14; if v is an internal pattern node we need to require on line 9 also that w has the same number of children as v. Therefore we have the following result.

Theorem 4.15 The problems of unordered path inclusion and unordered child inclusion can be solved in time $O(m^{3/2} n)$. □

Chung has shown that the above complexity also applies to testing whether a given tree is a subgraph of a given *unrooted* tree [Chu87].

Algorithm 4.14 can be modified to solve unordered region inclusion, too. Let pattern node v have children $v_1, \ldots, v_k$, and let target node w have children $w_1, \ldots, w_l$ where $l \geq k$. At the core of the algorithm we need to test whether the subtrees of v match on k *adjacent* children of w. This is a special restriction to bipartite matching. The instances of the problem consist of a graph with $k + l$ vertices and at most $k \times l$ edges, and the size of a maximum matching is at most k.

The bipartite matching problem with the restriction that the image nodes have to be adjacent siblings can be solved by considering $l - k + 1$ subgraphs that are obtained by restricting at a time to k adjacent sibling nodes of the target. The number of edges in each such subgraph is at most k^2. By repeating the algorithm of [HK73] $(l - k + 1)$ times we get total time $(l - k + 1)O(\sqrt{k}\, k^2) = O(k^{2.5} l)$.

The basic bipartite matching algorithm can be applied incrementally for this version of the problem. Consider the graph $G = (X \cup Y, E)$ where X consists of k children of a pattern node and Y consists of the children $\{w_1, \ldots, w_l\}$ of a target node, and $E = \{(x, y) \mid x \in X, y \in Y, a(x, y) = \mathbf{true}\}$. First we compute a maximum matching for graph G restricted to nodes $X \cup \{w_1, \ldots, w_k\}$. Matching this first subgraph requires at most k augmenting steps using the basic bipartite matching algorithm. As soon as we find a matching of k edges we are done.

Assume then that we have computed a maximum matching M of size p for a subgraph that contains the target nodes $w_i, \ldots, w_{i+k-1}$. Next we consider the subgraph with target nodes $w_{i+1}, \ldots, w_{i+k}$. At least $p-1$ edges of matching M belong to the new subgraph, and they are taken as the matching to be extended by the basic algorithm. At most two augmenting steps can be performed for this matching because matching M was maximum. Therefore, for each of the l target nodes at most two augmenting paths are computed in a subgraph of size k^2. Thus the total time for the bipartite matching is $O(k^2 l)$.

Applying this version of bipartite matching we get that the dominating term in the time complexity of the algorithm for unordered region inclusion is

$$\sum_{v \in V} \sum_{w \in W} |v|^2 |w| = \sum_{v \in V} |v|^2 \sum_{w \in W} |w| \,. \tag{4.7}$$

Like above, we get from (4.7) the upper bound time complexity for unordered region inclusion.

Theorem 4.16 Unordered region inclusion can be solved in time $O(m^2 n)$. □

4.5 Left ordered embeddings

In this section we present the concept of left embeddings that helps us to solve ordered tree inclusion problems efficiently. There may be exponentially many ordered embeddings of a tree P in a tree T. To avoid searching among these embeddings, we develop an algorithm that tries to match P at the root of T by embedding the subtrees of P as deep and as left as possible in T.

In order to discuss the order of images of sibling nodes in an embedding we define the right and left relatives of a node. Let F be a forest, N the set of its nodes, and u a node in F. The set of *right relatives* of u is defined by

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

i.e., the right relatives of u are those nodes that follow u in both preorder and postorder. Correspondingly, the set of *left relatives* of a node u, denoted by $lr(u)$, consists of the nodes that precede u both in preorder and in postorder. (See Figure 4.2.)

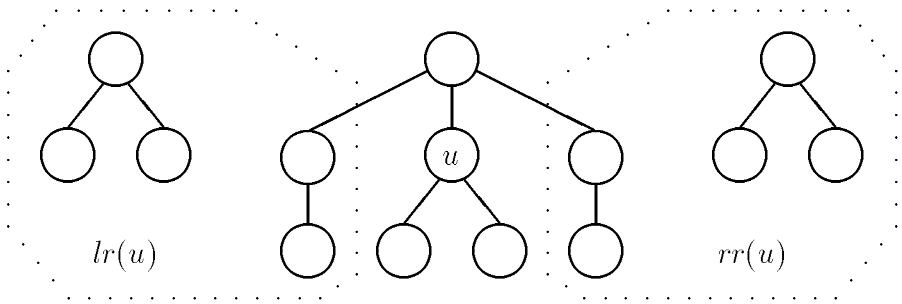


Figure 4.2: The left and right relatives of node u.

Lemma 4.17 Let u, v and x be three nodes in a forest. Then it is not possible that both

$$\begin{array}{ccccc} pre(u) & < & pre(v) & < & pre(x) \quad \text{and} \\ post(x) & < & post(u) & < & post(v) \end{array}$$

hold.

Proof. The above conditions imply by Lemma 2.1 that u and v are ancestors of x, and neither of them is an ancestor of the other, which is impossible. $\square$

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

Lemma 4.18 Let u and v be nodes, and assume $v \in rr(u)$. Then $desc(v) \subset rr(u)$.

Proof. Let a node x be a descendant of v, which is by Lemma 2.1 equivalent to $pre(v) < pre(x)$ and $post(x) < post(v)$. Node u precedes v in both orders which implies especially that $pre(u) < pre(x)$. Now Lemma 4.17 implies also that $post(u) < post(x)$. $\square$

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

Lemma 4.19 Let u and v be nodes in a forest. If $post(u) \leq post(v)$, then $rr(v) \subseteq rr(u)$.

Proof. If $post(u) = post(v)$, then $u = v$ and $rr(u) = rr(v)$. Let $post(u) < post(v)$ and $y \in rr(v)$, which means

$$\begin{array}{ccccc} & & pre(v) & < & pre(y) \quad \text{and} \\ post(u) & < & post(v) & < & post(y) \end{array} \tag{4.8}$$

In order to show that $y \in rr(u)$, it suffices to show that $pre(u) < pre(y)$. If $pre(u) < pre(v)$, we have $pre(u) < pre(y)$ by (4.9); otherwise $pre(u) < pre(y)$ by Lemma 4.17. $\square$

Definition 4.20 Let $F = \langle P_1, \ldots, P_k \rangle$ where $k \geq 1$ and G be forests, and let $\mathcal{E}$ be a collection of embeddings of F in G. An embedding $f \in \mathcal{E}$ is a *left embedding of* $\mathcal{E}$ if for every $g \in \mathcal{E}$

$$post(f(root(P_k))) \leq post(g(root(P_k))) \; .$$

A left embedding of the set of all ordered embeddings of F in G is called a *left ordered embedding*, or simply a *left embedding*, of F in G. $\square$

It is obvious that every finite nonempty set of embeddings has at least one left embedding. Therefore we can concentrate on left embeddings when testing for the existence of an ordered embedding.

Theorem 4.21 Let F and G be forests. There is an ordered embedding of F in G if and only if there is a left embedding of F in G. $\square$

The next theorem presents a method to build left embeddings by proceeding from left to right. The method is applied in the next algorithm.

Theorem 4.22 Let $F = \langle P_1, \ldots, P_k \rangle$ where $k \geq 2$ and G be forests, and let f be a left embedding of $F_1 = \langle P_1, \ldots, P_i \rangle$ in G, where $1 \leq i < k$. Let F_2 be the forest $\langle P_{i+1}, \ldots, P_k \rangle$ and let $\mathcal{E}$ be the set of ordered embeddings g of F_2 in G such that $g(\mathit{root}(P_{i+1})) \in \mathit{rr}(f(\mathit{root}(P_i)))$. Then the following hold:

1. If $\mathcal{E}$ is empty, there is no ordered embedding of F in G.
2. If $\mathcal{E}$ is nonempty and g is a left embedding of $\mathcal{E}$, then $f \cup g$ is a left embedding of F in G.

Proof.

1. Assume counterpositively that g is an ordered embedding of F in G. Let g_1 and g_2 be the restrictions of g in F_1 and F_2, respectively. Then $g_2(\mathit{root}(P_{i+1})) \in \mathit{rr}(g_1(\mathit{root}(P_i)))$, and $\mathit{post}(f(\mathit{root}(P_i))) \leq \mathit{post}(g_1(\mathit{root}(P_i)))$, since f is a left embedding of F_1 in G. Therefore

$$g_2(\mathit{root}(P_{i+1})) \in \mathit{rr}(f(\mathit{root}(P_i)))$$

by Lemma 4.19, which means that $g_2 \in \mathcal{E}$.

2. First we show that $f \cup g$ is an ordered embedding of F in G. It is sufficient to show that $f \cup g$ preserves the relative order of any two nodes x of F_1 and y of F_2. Now $\mathit{root}(P_i)$ is the last node of forest F_1 in postorder. Therefore we have by Lemma 4.19 that $\mathit{root}(P_j) \in \mathit{rr}(x)$ for all $i < j \leq k$, and Lemma 4.18 states that $y \in \mathit{rr}(x)$ also when y is not the root of any of the trees $P_1, \ldots, P_k$. Then we can see that $g(y) \in \mathit{rr}(f(x))$ by applying the same argument to the images of the above nodes.

Then we show that $f \cup g$ is a left embedding of F in G. For this, let h be any ordered embedding of F in G, and let h_1 and h_2 be the restrictions of h to F_1 and F_2, respectively. Now we have that $h_2 \in \mathcal{E}$, and therefore $\mathit{post}(g(\mathit{root}(P_k))) \leq \mathit{post}(h_2(\mathit{root}(P_k)))$. This entails the claim since $g(\mathit{root}(P_k)) = f \cup g(\mathit{root}(P_k))$ and $h_2(\mathit{root}(P_k)) = h(\mathit{root}(P_k))$. $\square$

4.6 Left embedding algorithm

Now we are ready to give an algorithm for searching the nodes w of the target T such that $T[w]$ includes the pattern P with order minimally. The algorithm is another variation of the general bottom-up dynamic programming schema of Algorithm 4.3. As before, the algorithm manipulates nodes as postorder numbers. For example, the minimum of a set of nodes refers to the first node of the set in postorder. The algorithm uses an auxiliary target node 0 that is a left relative of all the other target nodes.

Consider the situation where the algorithm compares a pattern node v against a target node w, and the subtrees of v are $P_1, \ldots, P_k$. First the algorithm searches for the first occurrence in postorder of P_1 among the descendants of node w, if there is any. The algorithm uses a pointer p for traversing the descendants of node w. After finding a left embedding f for the forest $\langle P_1, \ldots, P_i \rangle$ pointer p points at node $f(root(P_i))$. In order to extend f to a left embedding of $\langle P_1, \ldots, P_{i+1} \rangle$ into the subtrees of w we choose the closest right relative of p in postorder that is an occurrence of P_{i+1} and a descendant of w. The central idea of the algorithm is to apply dynamic programming in a way that allows this node to be chosen in *constant time*. This is achieved by using a table e having rows $1, \ldots, m$ and columns $0, \ldots, n-1$. The nodes of P are used as row indices of the table, and the nodes of T are used as column indices and contents of the table. Denote by $R(P,T)$ the collection of root preserving ordered embeddings of tree P into tree T. We compute into table e the values

$$e(v, w) = \min(\{x \in rr(w) \mid \exists\, f \in R(P[v], T[x])\} \cup \{n+1\}) \, , \qquad (4.9)$$

where v is a node of P and w is a node of T excluding the root of T. That is, $e(v,w)$ contains the closest occurrence of $P[v]$ among the right relatives of target node w, or if there is no such node, the value in $e(v,w)$ is greater than any postorder number in T. Columns $0, \ldots, n-1$ suffice for the table because node n, the root of T, has no right relatives.

The final result of the computation can be found on row $root(P)$ of the table. There is a root preserving embedding of P in $T[w]$ for every $w \leq n$ that appears on row $root(P)$, and P is an ordered included tree of T if and only if $e(root(P), 0) \leq n$.

When a root preserving embedding of $P[v]$ in $T[w]$ is found, a pointer q is used for writing value w into $e(v,q)$ for those nodes q that satisfy equation (4.9). Note that those nodes q are left relatives of node w.

Algorithm 4.23 [KM91b] Left embedding algorithm for ordered tree inclusion.

Input: Trees $P = (V, E, root(P))$ and $T = (W, F, root(T))$.

Output: Table e filled so that for all $v \in V$ and $w = 0, \ldots, n-1$

$$e(v, w) = \min(\{x \in rr(w) \mid \exists\ f \in R(P[v], T[x])\} \cup \{n+1\}) \ .$$

Method:

```
1.    for v := 1,...,m do
2.        comment: Initialize row v of e;
3.        for w := 0,...,n−1 do e(v,w) := n+1; od;
4.        Let v_1,...,v_k be the children of v;
5.        q := 0;
6.        for w := 1,...,n do
7.            if label(v) = label(w) then
8.                p := min(desc(w) ∪ {n+1}) − 1;
9.                i := 0;
10.               while i < k and p < w do
11.                   p := e(v_{i+1}, p);
12.                   if p ∈ desc(w) then i := i+1; fi;
13.               od;
14.               if i = k then
15.                   while q ∈ lr(w) do
16.                       e(v,q) := w;
17.                       q := q+1;
18.                   od;
19.               fi;
20.           fi;
21.       od;
22.   od;
```

The loop on lines 10–13 tests whether there is a left embedding of the forest $\langle P[v_1], \ldots, P[v_k] \rangle$ into the forest of subtrees of w. This is by Theorem 4.21 and Lemma 4.2 equivalent to testing whether there is a root preserving ordered embedding of $P[v]$ in $T[w]$, because on these lines $label(v) = label(w)$. The test is successful if and only if all the subtrees of v can be embedded in the loop, i.e., if and only if $i = k$ on line 14.

Next we show that the above test is correct. This is obvious if pattern node v is a leaf. Let then pattern node v have children. If node w is a leaf, $P[v]$ can not be embedded in $T[w]$. In this case p gets value n on line 8 of the algorithm, which prevents the execution of lines 10–19. Otherwise p gets the value $min(desc(w)) - 1$, which is the closest left relative of w. The postorder numbers of the descendants of w are then $\{p+1, p+2, \ldots, w-1\}$. By Lemma 4.18, $desc(w) \subset rr(p)$. Assume that correct values have been computed in the rows of table e corresponding to the children of v; this will be established in Theorem 4.24. The first execution of line 11 finds then in p the leftmost occurrence of $P[v_1]$ among the descendants of w, if there are any. The correctness of the remaining executions of the loop follows from Theorem 4.22.

As an example, consider how Algorithm 4.23 finds the embedding of a tree $P = a(c, e)$ into a tree $T = a(b(c), a(b(d), a(b(e))))$. The trees and the result of the computation are shown in Figure 4.3. Each column of table e is shown to the right of the corresponding node of tree T.

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

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

Finally $v = 3 = root(P)$. Node $w = 7$ is the first node of T with $label(v) = label(w) = a$. The node p preceding $desc(w)$ in postorder is node 4. The first child of v is node number 1. Its image in $rr(p) \supset desc(w)$ is looked up from $e(1, 4)$; value 10 means that there is no embedding of the child into $desc(w)$. Next, a similar failure occurs with $w = 8$ and $p = 2$. Finally $w = 9 = root(T)$ and $label(v) = label(w) = a$, which leads to testing the embedding of the children of v by executing $p := 0$, $p := e(1, 0) = 1$, and $p := e(2, 1) = 5$. The algorithm has found a root preserving embedding of P into T. The value of $w = 9$ is written as the final result into $e(3, 0)$. Since 0 is the only left relative of 9, the computation ends.

Theorem 4.24 Algorithm 4.23 fills the table e correctly.

Proof. Consider a fixed pattern node v. We outline a proof that all

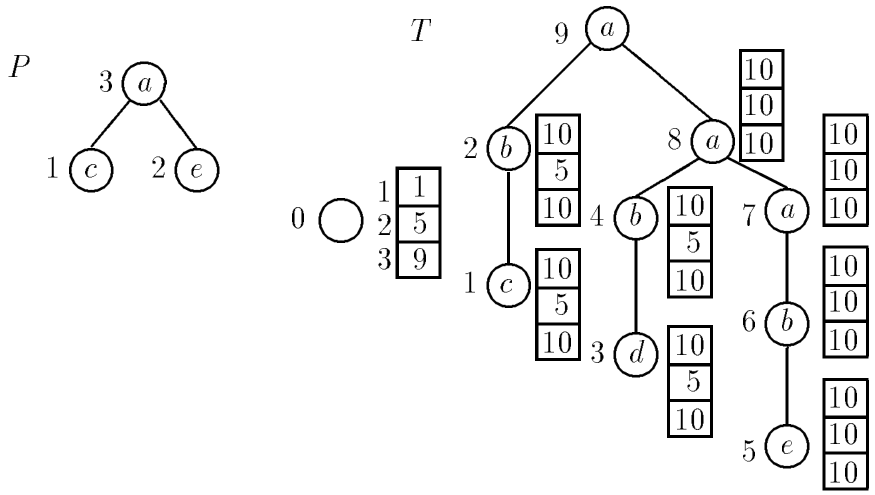


Figure 4.3: The result of applying the algorithm to trees P and T.

columns of row v get correct values in the **while**-loop on lines 15–18 of the algorithm. First, the precondition that all columns up to q have got the correct values is initially true. We show that the following invariant holds on line 16:

$$\exists\, f \in R(P[v], T[w]) \quad \wedge \quad w \in rr(q) \tag{4.10}$$

$$\wedge\ \forall\ 1 \leq x < w : (\exists\, g \in R(P[v], T[x]) \quad \Rightarrow \quad x \notin rr(q))\ . \tag{4.11}$$

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

The previous discussion about the loop on lines 10–13 showed that the loop on lines 15–18 is executed if and only if there is a root preserving ordered embedding of $P[v]$ in $T[w]$. Therefore (4.10) holds when we are on line 16. (Note that $q \in lr(w)$ and $w \in rr(q)$ are equivalent.) When line 16 is executed for the first time (4.11) is vacuously true. By Lemma 4.19 we can strengthen (4.11) into

$$\forall\ 1 \leq x < w, \forall\ y \geq q : (\exists\, g \in R(P[v], T[x]) \Rightarrow x \notin rr(y))\ . \tag{4.12}$$

At the exit from the loop $w \notin rr(q)$ allows further strengthen (4.12) into

$$\forall\ 1 \leq x \leq w, \forall\ y \geq q : (\exists\, g \in R(P[v], T[x]) \Rightarrow x \notin rr(y))\ .$$

This postcondition makes (4.11) true on the subsequent executions of the loop. It also tells that the writing is complete, i.e., value w must not be written into $e(v, y)$ for any column $y \geq q$. $\square$

Theorem 4.25 Algorithm 4.23 requires $\Theta(mn)$ time and space.

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

Time: During every execution of the outermost loop the pointer q may increase in steps of one from 0 to n. Therefore the **while**-loop incrementing q requires $O(n)$ steps per one outermost loop. One execution of the **while**-loop on lines 10–13 requires time $O(1 + |v|)$, where $|v|$ is the number of the children of node v. We get total time

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

The sum $\sum_{v=1}^{m} |v|$ equals the number of edges in tree P, which is $m - 1$. Therefore the total time is $O(n(3m - 1)) = O(mn)$. On the other hand, the algorithm uses obviously time $\Omega(mn)$. $\square$

4.7 A space efficient algorithm for ordered tree inclusion

In an application where the target tree is very large the $\Theta(mn)$ space and time requirements of Algorithm 4.23 may be unacceptable. This section improves on the previous result by presenting an algorithm that solves the ordered tree inclusion problem in $O(m\, depth(T))$ space. The result is useful in practice, since most text databases can be represented by trees having a small constant depth. Moreover, the complexity of the new algorithm is sensitive to the instances of the problem: it runs in time $O(c_P(T)\, n)$, where $c_P(T)$ is the number of the subtrees of P that are included in T. Especially, if no part of the pattern appears in the target, the algorithm runs in time $O(n)$.

Let P be a pattern tree. Call a nonempty sequence $\langle v_1, \ldots, v_k \rangle$ of nodes of P a *sibling interval* of P, if node v_{i+1} is the next sibling of node v_i for all $i = 1, \ldots, k - 1$. The parent of the nodes in a sibling interval is called the *parent of the sibling interval.* A sibling interval is *complete* if it contains all the children of its parent. We say that a sibling interval $\langle v_1, \ldots, v_k \rangle$ can be embedded in a tree T (in a forest F) if there is an ordered embedding of $\langle P[v_1], \dots, P[v_k] \rangle$ in T (in F).

The following algorithm is a variation of Algorithm 4.8. It computes in bottom-up fashion for each target node w a *match set* $M(w)$. The match set $M(w)$ consists of the sibling intervals $\langle v_1, \dots, v_k \rangle$ of P that can be embedded in $T[w]$.

For a target node w the algorithm first computes the set of the sibling intervals $\langle v_1, \dots, v_k \rangle$ that can be embedded in the forest of the subtrees of w. This set is computed from the match sets of the children of w. The computation is based on a combining operation denoted by $\oplus$. Let A and B be the sets of sibling intervals that can be embedded in forests $\langle T_1, \dots, T_i \rangle$ and $\langle T_{i+1}, \dots, T_l \rangle$, respectively. Then $A \oplus B$ is the set of sibling intervals that can be embedded in $\langle T_1, \dots, T_l \rangle$. We defer the implementation of this operation for a moment.

After computing in a variable S the sibling intervals that can be embedded in the descendants of w, the algorithm computes in a set $S\Delta$ the sibling intervals consisting of single pattern nodes that can be mapped in an ordered embedding to w. These are exactly the pattern nodes whose label matches with the label of w and that are either leaves or have the sibling interval consisting of their children in S. After this process $S \cup S\Delta$ is the match set of w. The algorithm is represented as a recursive function that returns a match set and reports the target subtrees that match the pattern.

Algorithm 4.26 Space efficient algorithm for ordered tree inclusion.

Input: Trees $P = (V, E, root(P))$ and $T = (W, F, root(T))$.

Output: The nodes w of T such that $T[w]$ is an ordered including tree of P.

Method: Call function $M(P, T)$;

function $M(P, T)$:

$S := \emptyset$;

Let $T_1, \dots, T_k$, $k \geq 0$, be the immediate subtrees of T;

for $i := 1, \dots, k$ **do**

  $S := S \oplus M(P, T_i)$;

od;

$S\Delta := \emptyset$;

```
7.      for all v ∈ V such that label(v) = label(root(T)) do
8.          if (children(v) = ∅ or ⟨children(v)⟩ ∈ S) then
9.              SΔ := SΔ ∪ {⟨v⟩};
10.         fi;
11.     od;
12.     if ⟨root(P)⟩ ∈ SΔ then
13.         output root(T);
14.     fi;
15.     return S ∪ SΔ;
```

We can form a set of $k(k+1)/2$ sibling intervals from k sibling nodes, but we can do with smaller representations for match sets. A set of intervals is *simple* if it does not have two distinct members one of which is a subinterval of the other. If a sibling interval belongs to a match set, then all of its subintervals belong there, too. Knowing this we can represent the match sets by simple sets, keeping only the maximal intervals in S.

Lemma 4.27 Let P be a tree (or a forest) on m nodes. Then a simple set of sibling intervals of P has at most m members.

Proof. Each node of P can be the leftmost node of at most one sibling interval. □

To measure the amount of "partial inclusion" of a pattern P in a target T we define the P-*content* of T, denoted by $c_P(T)$, as the number of subtrees of P that are ordered included trees of T. Note that $0 \leq c_P(T) \leq |P|$, and P is an ordered included tree of T if and only if $c_P(T) = |P|$. By the same argument as in Lemma 4.27, the size of a simple match set is at most $c_P(T)$. We will see in a moment how the operation $\oplus$ on line 4 and lines 7–15 can be performed in time proportional to the number of elements in the returned match set. Therefore Algorithm 4.26 uses for each target node w time $O(1 + c_P(T[w]))$. By noting that the algorithm uses $O(1 + c_P(T[w]))$ space for each recursive call to inspect the target subtree $T[w]$ we get the following result.

Theorem 4.28 Algorithm 4.26 uses time

$$O\left(\sum_{w \in W} (1 + c_P(T[w]))\right)$$

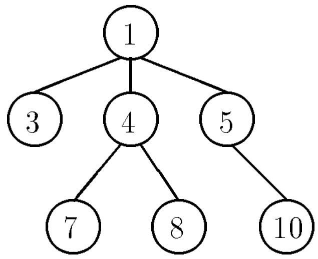


Figure 4.4: The interval numbering for easy recognition of sibling nodes.

and space $O((1 + c_P(T))\, depth(T))$. Especially, the algorithm works in the best case in $O(n)$ time and in $O(depth(T))$ space, and in the worst case in $O(mn)$ time and in $O(m\, depth(T))$ space. $\Box$

Since the size of match sets depends on P only, executing lines 4 and 7–15 with a fixed pattern takes constant time. This observation shows the following result, which follows for time also immediately from Theorem 4.25.

Theorem 4.29 For a fixed pattern P, the ordered tree inclusion problem can be solved in time $O(|T|)$ and in space $O(depth(T))$. $\Box$

We can also perform preprocessing similar to Algorithm 4.8 by enumerating the simple sets of sibling intervals and building tables that allow performing lines 4 and 7–11 in constant time. Again the preprocessing would require exponential time and space; although the number of simple sets of sibling intervals is smaller than the number of all sets of sibling intervals, it can easily be seen to be $\Omega(2^m)$ in the worst case.

Algorithm 4.26 called for efficient representation and manipulation of simple sets of sibling intervals. For the manipulation of sibling intervals we need to be able to recognize sibling nodes of P and to compare their relative order easily. For this reason we number the pattern nodes in the following manner. First, assign 1 to the root of P. Then, during a level-wise traversal number the children of each internal node consecutively starting from $x + 2$, when x is the largest number assigned so far. A pattern node u is the next sibling of a pattern node v if and only if the sibling number of u is one greater than the sibling number of v. In the rest of this section we refer to the pattern nodes by these numbers, which we call *interval numbers*. For an example of this numbering, see Figure 4.4.

A sibling interval $\langle a, a+1, \ldots, b\rangle$ can be represented by the pair $[a, b]$ of interval numbers; a is called the *start point* of the interval, and b is called the *end point* of the interval.

Next we present an algorithm for combining interval sets. Let L be the set of sibling intervals that can be embedded in the forest $\langle T_1, \ldots, T_i\rangle$, and R the set of sibling intervals that can be embedded in the forest $\langle T_{i+1}, \ldots, T_k\rangle$. The intervals in L and R are called *left* and *right* intervals, respectively. A left interval $l = [l.a, l.b]$ and a right interval $r = [r.a, r.b]$ *can be combined* if $l.a \leq r.a \leq l.b + 1$; in this case the combination lr of l and r is the interval $[l.a, \max(l.b, r.b)]$. The result of the combination $L \oplus R$ is such a set of intervals S that

1. for each interval $x \in L \cup R$ there is an interval $y \in S$ that includes x,
2. for each $l \in L$ and $r \in R$ that can be combined there is an interval $y \in S$ that includes the combined interval lr, and
3. each $y \in S$ is either a left interval, a right interval, or a combination of a left and a right interval.

The next algorithm combines two simple sets of intervals yielding a simple set of intervals. The merging-like method is based on representing the sets as lists of intervals sorted by start points. Because the sets are simple the intervals are sorted by the end points, too.

Algorithm 4.30 Combining intervals.

Input: Sorted and simple lists of intervals $L = l_1, \ldots, l_m$ and $R = r_1, \ldots, r_n$.

Output: Sorted and simple representation of $L \oplus R$.

Method:

$i := 1$; $j := 1$;
$l_{m+1} := [\infty, \infty]$; $r_{n+1} := [\infty, \infty]$;
while $i \leq m$ **or** $j \leq n$ **do**
  if $l_i.b + 1 < r_j.a$ **then**
    comment: l_i cannot be combined with r_j;
    output l_i;
    outb := $l_i.b$;
  else comment: $l_i.b + 1 \geq r_j.a$;

```
    if l_i.a ≤ r_j.a then
        while l_i.b + 1 ≥ r_j.a do
        comment: l_i can be combined with r_j;
            x := r_j.b;
            j := j + 1;
        od;
        outb:= max(l_i.b, x);
        output [l_i.a, outb];
    else comment: l_i.a > r_j.a;
        output r_j;
        outb := r_j.b;
        j := j + 1;
    fi;
    while l_i.b ≤ outb and l_i.b + 1 < r_j.a do
    comment: skip those l_i's that are included in output interval
        and cannot be combined with r_j;
        i := i + 1;
    od;
od;
```

It is easy to see that every output interval is either in $L \cup R$ or the combination lr of some $l \in L$ and $r \in R$. It is also straightforward to check that the result intervals are output in strictly increasing order with respect to both their start and end points; therefore the result list is both sorted and simple. The completeness of the result list can be shown by induction on the lengths of the input lists.

Algorithm 4.30 can combine two match sets in time proportional to the number of their elements. Now we continue to show that Algorithm 4.26 can compute the result $S \cup S\Delta$ in time that is proportional to its size. Set $S\Delta$ can be implemented as two sorted lists of intervals consisting of single nodes. A list S_I is used for intervals of internal nodes and a list S_L is used for intervals of leaves. Note that for any nodes u and v of P such that $1 < u < v$ (ordered as interval numbers) we have $parent(u) \leq parent(v)$. Therefore we can form list S_I by a single scan in the sorted list S locating the complete sibling intervals that have a parent whose label equals the label of the root of T. Computing list S_L is done a bit differently. In the preprocessing phase that performs the interval numbering of P, we can link all leaves of P with the same label in a list in increasing order of their interval numbers. Then we can form list S_L from the list of pattern leaves labeled by the label of

$root(T)$. To form the representation of the result $S \cup S\Delta$ we need to merge lists S_I and S_L in S keeping it simple. This is straightforward to do in time proportional to the final length of the result.

These considerations complete the claim that Algorithm 4.26 uses time $O(1 + c_P(T[w]))$ for a target node w.

4.8 Solving ordered path inclusion problems

Ordered path inclusion and ordered region inclusion problems are easier to solve than ordered tree inclusion problems. The reason is that with path inclusion and region inclusion it is sufficient to test the immediate subtrees of the pattern against the trees rooted by the *children* of a target node w when testing if the pattern occurs at w. With tree inclusion we have to examine trees rooted by the *descendants* of w, which offers more combinations to embed the immediate subtrees of the pattern.

Ordered path inclusion and ordered region inclusion can be solved using the familiar algorithm scheme that stores in an array the information of the matches between all pairs of pattern-target nodes. We compare again at the core of the algorithms a pattern node v with children $v_1, \ldots, v_k$ against a target node w with children $w_1, \ldots, w_l$.

The algorithm for ordered *path* inclusion has to find an ordered subsequence $w_{j_1}, \ldots, w_{j_k}$ of $w_1, \ldots, w_l$, where $1 \leq j_1 < \ldots < j_k \leq l$ are such that v_i matches at w_{j_i} for all $i = 1, \ldots, k$. This can be done by scanning the children of w from left to right and using a counter to indicate the number of children of v already matched. This solution yields the following complexity:

$$O(\sum_{w \in W} (1 + \sum_{v \in V} (1 + |w|))) = O(n + nm + m(n-1)) = O(mn) \ .$$

Note that with a minor change we can modify also Algorithm 4.26 to solve the ordered path inclusion problem. The modification is to return from the recursive calls only the set $S\Delta$ that contains essentially the pattern nodes that match at the root of the tested target subtree.

The algorithm for ordered *region* inclusion has to find a continuous sequence $w_j, \ldots, w_{j+k-1}$ of children of w, such that node v_i matches at w_{j+i-1} for all $i = 1, \ldots, k$. This can be done by a method resembling trivial string matching: Starting with $j = 1$, the sequences $v_1, \ldots, v_k$ and $w_j, \ldots, w_{j+k-1}$ are compared from left to right against each other as long as they match, and at a failure j is incremented by one and the comparison is started anew. The complexity of this solution to ordered region inclusion is

$$O(\sum_{w \in W}(1 + \sum_{v \in V}(1 + |v||w|))) = O(mn) \; . \tag{4.13}$$

We state these results as a theorem:

Theorem 4.31 The problems ordered path inclusion and ordered region inclusion are solvable in time $O(mn)$. □

Tree inclusion problems can be considered to be special cases of the editing distance problem for trees [Tai79, ZS89]. *Ordered* tree inclusion problems can be described and solved in the framework of Zhang, Shasha, Wang, and Jeong [ZSW91, WJZS91]. They allow patterns to contain variable length don't care symbols (VLDCs). A *path VLDC* is a pseudo node in the pattern that matches at an arbitrary path in the target. The algorithms of [ZSW91, WJZS91] allow *tree matching with cut*: the instances of the pattern are the subtrees of the target that are identical with the pattern after deleting the nodes matched with the VLDCs and possibly cutting at some nodes. Therefore, the ordered tree inclusion problem can be presented as a matching problem with cut after inserting a path VLDC on each edge of the pattern. Ordered *path* inclusion is the same problem as the tree matching problem with cut.

The algorithms of Zhang, Shasha and Wang [ZSW91, ZSW92] require time

$$O(|P| \cdot |T| \cdot \min(depth(P), leaves(P)) \cdot min(depth(T), leaves(T))) \; ,$$

where $depth(T)$ is the length of the longest root-to-leaf paths in tree T and $leaves(T)$ is the number of leaves in T. Thus our algorithms are faster than theirs by a factor of

$$min(depth(P), leaves(P)) \cdot \min(depth(T), leaves(T)) \; .$$

Nontrivial lower bounds for the complexity of the above problems are not known, but they do not seem to be easily solvable in linear time. We show below how string pattern matching with don't care symbols can be reduced in linear time to ordered path inclusion problems. The time complexity of the best known algorithm for string matching with don't care symbols is $O(polylog(m)\, n)$ [FP74], and the existence of a faster algorithm has been an open problem for almost 20 years. The idea of the reduction was presented in [Kos89], and it was also noted in [Ver92].

Let P be a pattern string $p_1, \ldots, p_m$ and T a target string $t_1, \ldots, t_n$ where $m \leq n$, both over the alphabet $\{0, 1, d\}$. The don't care character d matches at any other character, and the other characters match at themselves and at d. Pattern string P matches target string T at position i if p_j matches at t_{i+j-1} for every $j = 1, \ldots, m$.

Problem 13 (String matching with don't care symbols)
Given a pattern string P and a target string T, both over the alphabet $\{0, 1, d\}$, compute the set of positions at which P matches T. □

The reduction of string matching to ordered path or region inclusion is as follows. Each character c is represented as a term $\beta(c)$ as follows: $\beta(0) = b(a)$, $\beta(1) = b(b)$, and $\beta(d) = b$ for don't care characters in the pattern and $\beta(d) = b(a, b)$ for the don't care characters in the target. Note that a pattern character p matches at a target character t if and only if there is a root preserving path or region embedding of $\mathit{Tree}(\beta(p))$ into $\mathit{Tree}(\beta(t))$. The strings $P = p_1, \ldots, p_m$ and $T = t_1, \ldots, t_n$ are represented as the trees represented by the terms $\alpha(P)$ and $\alpha(T)$. The terms are constructed along the following recursive rule:

$$\begin{aligned} \alpha(c_1, \ldots, c_k) &= a(\beta(c_1), \alpha(c_2, \ldots, c_k)) \text{ when } k > 1 \\ \alpha(c_1, \ldots, c_k) &= \beta(c_1) \text{ when } k = 1 \; . \end{aligned}$$

An example of the reduction is shown in Figure 4.5.

Now it is easy to see that P matches T at position i if and only if there is an ordered root preserving path or region embedding of tree $\mathit{Tree}(\alpha(P))$ in tree $\mathit{Tree}(\alpha(t_i, \ldots, t_n))$. Since $m + n$ is the trivial lower bound for the complexity of inclusion problems, we can state the following theorem, which indicates the difficulty of obtaining linear time algorithms for ordered path inclusion problems.

Theorem 4.32 If ordered path inclusion or ordered region inclusion can be solved in time $O(f(m, n))$, then also string matching with don't care symbols can be solved in time $O(f(m, n))$. □

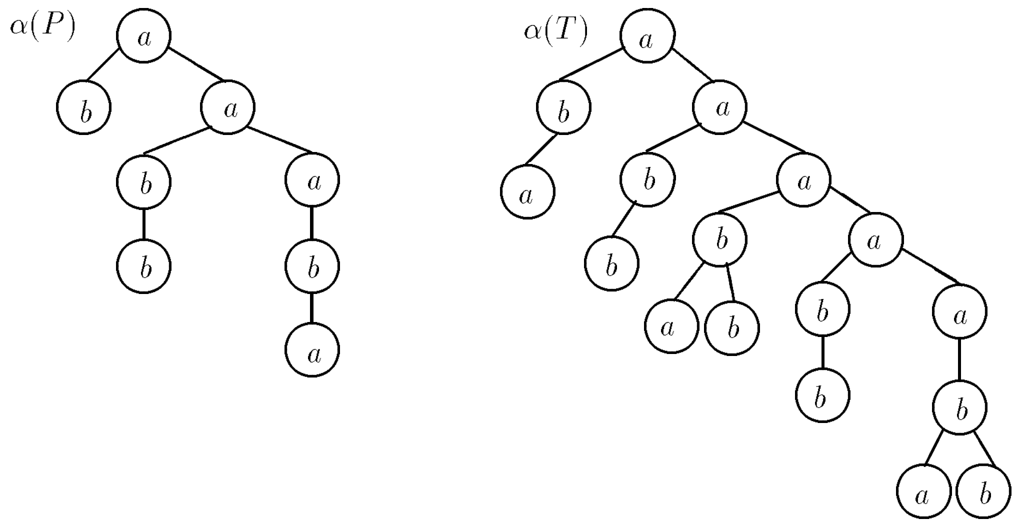


Figure 4.5: Representing the pattern string $P = d10$ and the target string $T = 01d1d$ as trees.

4.9 Classical tree pattern matching

In this section we give a short review of the most important methods for solving the ordered child inclusion problem, i.e., the classical tree pattern matching problem. The problem has been extensively studied, and we only touch upon the various approaches to it. The presentation follows mainly the review in [RR88] and in [RR92].

If we have any knowledge about the structure of the target, we can try to utilize it for solving the matching problem more efficiently than in the general case. We discuss this idea in Chapter 5, where we show how tree pattern matching can be solved efficiently if the target is a parse tree over an appropriate grammar.

The naive or basic tree pattern matching algorithm belongs to the folklore of computing. The basic algorithm traverses the target tree in preorder and compares the pattern to each subtree of the target in turn. In the worst case this algorithm requires $O(mn)$ time, but it can be shown to have a linear expected time behaviour [SF83].

Hoffman and O'Donnell [HO82] have presented advanced bottom-up and top-down tree pattern matching techniques. The key idea of the bottom-up algorithm is to find at each target node w a match set that consists of the subtrees of the pattern that match w. The match sets can be enumerated and coded by the enumeration. The method assumes that the alphabet is *ranked*, i.e., the label of a node determines the number of its children. (The leaves of the pattern that match at arbitrary nodes are represented by *variables*. We treat inclusion problems with variables in Chapter 6.) For each label there is a table whose dimension is equal to the rank of the label. The code assigned to a target node labeled by a is determined using the table of a on the basis of the codes of the match sets assigned to the children of the node. Once the tables have been computed, the matching takes $O(n)$ time. The main drawback of the method is the size of the tables and the time taken by the preprocessing; both are exponential in m.

The top-down tree matching algorithm of [HO82] is based on representing the pattern as a set of *path strings*, i.e., sequences of interleaved labels and child numbers each representing a unique path from the root to a leaf. (A path string begins with the label of the root, which is followed by the number of the child that is the next node on the path, and so on up to the label of the leaf at the end of the path.) The *length* of a path string is the number of child numbers in it. These path strings can be transformed in $O(m)$ time into an Aho-Corasick like multiple string recognition automaton [AC75]. A single final state of the automaton may accept many path strings; their lengths are stored in the corresponding final state. The nodes of the target tree are decorated with counters initialized to zero. The occurrences of the path strings in the target are recognized by running the automaton on the target tree. When a final state that accepts path strings of lengths $\{l_1, \ldots, l_p\}$ is reached at a target node w, the counters of the l_1th, the l_2th, $\ldots$, and the l_pth ancestor of w are incremented by one. The pattern occurs at those nodes whose counter reaches the number of leaves of P. The time required to do the matching is the maximum of n and the number of times a counter is incremented.

The amount of periodicity in the path strings of the pattern is measured by the *suffix index* $s(P)$ of the pattern. The suffix index of P is the maximum number of path strings of P such that they all are suffixes of some path string of P; it can range from 1 to m. The suffix index sets a tight upper bound on the number of counters that are incremented by the top-down algorithm. Therefore the matching time of the top-down algorithm is $O(s(P)n)$.

Recently Kosaraju [Kos89] broke the quadratic $O(mn)$ complexity barrier of tree pattern matching. Dubiner, Galil, and Magen [DGM90] improved the result of Kosaraju by introducing an $O(n\sqrt{m}\, polylog(m))$ time algorithm.

We state this best known upper bound as a theorem.

Theorem 4.33 [DGM90] Classical tree pattern matching can be solved in time $O(n\sqrt{m}\, polylog(m))$. $\square$

The algorithms of [Kos89] and [DGM90] apply sophisticated but complicated techniques like suffix trees of trees and convolutions; therefore they are more of theoretical than practical value. Considerable effort is needed in the algorithms to handle periodic patterns efficiently. In Chapter 5 we consider restricting the periodicity of the target tree, which makes the tree pattern matching problem essentially easier.

In the applications of tree pattern matching the nodes of the pattern are usually either *function symbols* or *variables*. Variables can appear only as leaves, and a variable matches at any node. Function symbols match only at nodes with the same label and the same number of children. This variation can easily be included in the definition of child inclusion problems.

If we require that repeating occurrences of the same variable match only at the roots of identical subtrees, we get another problem that is called either *nonlinear tree pattern matching* or *tree pattern matching with logical variables*. Solving this problem is considered in [RR88]. We discuss tree inclusion problems with logical variables in Chapter 6.

4.10 Subtree problems are solvable in linear time

In this section we show that the unordered and ordered subtree problems can be solved in time linear in the size of the target tree.

An instance relation is *linearly solvable* if there is a constant c such that the question "Is U an instance of P?" can be answered in time bounded by $c|U|$ for all trees P and U. For example, the instance relation of the unordered subtree problem is linearly solvable:

Lemma 4.34 The relation "trees P and U are isomorphic" is linearly solvable.

Proof. A linear time algorithm for the problem, based on lexicographic sorting, is given in [AHU74, p. 84–86]. $\square$

A set of nodes of T is a *candidate set of occurrences* of P, if it is a superset of the set of occurrences of P. A set of nodes N is *k-thin* if any node $n \in N$

has at most $k-1$ ancestors in N. A 1-thin set of nodes is *flat*. That is, a flat set of nodes does not contain two nodes one of which is an ancestor of the other. Note that in tree pattern matching the set of occurrences need not be flat, since the pattern may match both at a node and at some of its descendants.

Lemma 4.35 Assume that for a tree inclusion problem there is a k-thin candidate set of occurrences that can be computed in time $O(kn)$, and that the instance relation is linearly solvable. Then the tree inclusion problem is solvable in time $O(kn)$.

Proof. First compute a k-thin candidate set C in time $O(kn)$. For each node u in C, test whether P matches at u. Since the instance relation is linearly solvable, this requires time at most $c\sum_{u\in C}|T[u]|$ for some constant c. Because C is k-thin, each node of T can belong to at most k different trees rooted by nodes in C, and therefore

$$c\sum_{u\in C}|T[u]| \leq ckn = O(kn) \ .$$

□

The next lemma states the simple result that the instance relation of ordered child inclusion is linearly solvable.

Lemma 4.36 The instance relation "P is an ordered child-included tree of U" is linearly solvable.

Proof. The relation can be tested simply by comparing the corresponding nodes of the trees against each other; at most $\min\{|P|,|U|\}$ nodes of U are examined. □

Now we are ready to state the time complexities of the subtree problems.

Theorem 4.37 The unordered and the ordered subtree problem can be solved in time $O(n)$.

Proof. The roots of the subtrees of size m form a flat candidate set of occurrences. They can be located in time $O(n)$ by traversing the target bottom-up and counting the sizes of its subtrees. The instance relations are linearly solvable by Lemmas 4.34 and 4.36. The result follows from Lemma 4.35. □

Lemma 4.35 is a generalization of an argument used in [Gro91]. Grossi shows that locating subtrees of the target that are identical to the pattern or differ only with regard to don't care labels or up to m mismatching labels can be done in $O(n)$ sequential time. The same idea appears also in the ordered subtree algorithm of [Dub90].

4.11 Summary of complexities

The table below summarizes the complexity results for the tree inclusion problems considered in this chapter. Shorthand NPC stands for NP-complete, and m and n stand for the size of the pattern and for the size of the target, as elsewhere. Notation O denotes an upper bound for the worst-case complexity of a problem, and Θ denotes tight complexity (i.e., one that equals the worst-case lower bound complexity) for the problem.

	incl.	path incl.	region incl.	child incl.	subtree
unordered	NPC	$O(m^{1.5}n)$	$O(m^2n)$	$O(m^{1.5}n)$	$\Theta(n)$
ordered	$O(mn)$			$O(n\sqrt{m}\, polylog(m))$	$\Theta(n)$

Table 4.1: The complexities of the tree inclusion problems

Chapter 5

Grammatical tree inclusion

In this chapter we consider tree inclusion problems with targets that are parse trees over some grammar. Motivation for this comes from text databases modeled by context-free grammars. Context-free grammars are rather commonly used for describing the structure of text documents [GT87, BR84, CIV86, FQA88, QV86, KLMN90].

First in Section 5.1 we discuss describing the structure of targets by context-free grammars. We define *nonperiodic grammars* and argue that they are adequate for describing the structure of many text databases appearing in practice. Then we show that classical tree pattern matching can be solved in linear time on nonperiodic target trees. In Section 5.2 we show that also ordered tree inclusion is solvable in linear time on nonperiodic targets. Section 5.3 discusses utilizing the grammar of the target in inclusion problems. The possibilities include checking patterns for syntactic correctness before actual matching, and transforming inclusion problems to equivalent but easier problems.

More discussion about text databases appears in Chapter 7, where we also give examples of using inclusion patterns as database queries.

5.1 Grammatical targets and tree pattern matching

We define a *grammar* to be a quadruple $G = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$, where $\mathcal{V}$ is the set of *nonterminals*, $\mathcal{T}$ is the set of *terminals*, $\mathcal{P}$ is the set of *productions* and $S \in \mathcal{V}$ is the *start symbol.*

It is useful to allow regular expressions on the right-hand-sides of productions. This leads to fewer nonterminals and seems to be a form easily comprehensible also to nonspecialists. Therefore, we define the productions to be of the form $A \rightarrow \alpha$, where α is a regular expression over $\mathcal{V} \cup \mathcal{T}$. We say that a production $A \rightarrow w$ is an *instance* of $A \rightarrow \alpha$, if w belongs to the regular language defined by α.

As an example we show a grammar for describing the structure of a list of bibliographic references stored in a text database system.

publications	$\rightarrow$	publication*
publication	$\rightarrow$	authors title journal volume year pages
authors	$\rightarrow$	author*
author	$\rightarrow$	initials name
initials	$\rightarrow$	text
name	$\rightarrow$	text
title	$\rightarrow$	text
journal	$\rightarrow$	text
volume	$\rightarrow$	number
year	$\rightarrow$	number
pages	$\rightarrow$	start end
start	$\rightarrow$	number
end	$\rightarrow$	number
text	$\rightarrow$	character*
number	$\rightarrow$	digit*

The obvious productions for nonterminals *character* and *digit* have been excluded. The grammar is allowed to be ambiguous because it will not be used for parsing. Producing string representations of a parse tree and parsing them back to trees can be done using versions of the grammar that are annotated with extra terminal symbols. The methodology is explained in [KLMN90] and in [Nik90].

To define *parse trees* over a grammar G, we define sets $T(G, a)$ for terminals $a \in \mathcal{T}$ and sets $T(G, A)$ for nonterminals $A \in \mathcal{V}$:

$$T(G, a) = \{a\} \text{ for terminals } a \; ;$$

$$\begin{aligned} T(G, A) = \{A(t_1, \ldots, t_n) \mid {} & A \rightarrow B_1, \ldots, B_n \\ & \text{is an instance of a production in } P \\ & \text{and } t_i \in T(G, B_i) \text{ for each } i = 1, \ldots, n\} \; . \end{aligned}$$

Here $A(t_1, \ldots, t_n)$ stands for a tree whose root is labeled by the nonterminal A and whose ith immediate subtree is t_i for all $i = 1, \ldots, n$. That is, elements of $T(G, A)$ represent derivations of terminal strings from the nonterminal A according to G. Finally, the trees that represent derivations from the start symbol of G, i.e., the trees in $T(G, S)$, are the parse trees over G.

A tree inclusion problem is *G-grammatical* if the target is a parse tree over a grammar G. Grammatical inclusion problems are in general not easier than the unrestricted ones. Let T be a tree and let $\{a_1, \ldots, a_l\}$ be the set of labels occurring in T. Consider then the following grammar G

$$\begin{array}{rcl} S & \rightarrow & a_1 \mid a_2 \mid \ldots \mid a_l \\ a_1 & \rightarrow & (a_1 \mid a_2 \mid \ldots \mid a_l)^* \\ & \vdots & \\ a_l & \rightarrow & (a_1 \mid a_2 \mid \ldots \mid a_l)^* \ . \end{array}$$

Now any inclusion problem with target T is G-grammatical.

A grammar is *nonperiodic*[1] if it has no nonterminal A that can derive a string of the form $\alpha A \beta$. A tree T is *k-periodic* if any nonterminal appears at most k times on a single root-to-leaf path in T. A 1-periodic tree is *nonperiodic*, or equivalently, a tree T is nonperiodic if and only if T is a parse tree over some nonperiodic grammar. An inclusion problem (P, T) is *nonperiodic*, if it is G-grammatical for a nonperiodic grammar G (i.e., the target T is nonperiodic).

It is easy to see that nonperiodic grammars with regular expressions in their productions define exactly the regular languages [HU79]. For example, we could describe the previous list of bibliographic references also by a single production of the following form.

$$\begin{array}{lcl} \text{publications} & \rightarrow & ((\text{character}^* \ \text{character}^*)^* \\ & & \text{character}^* \ \text{character}^* \\ & & \text{digit}^* \ \text{digit}^* \ \text{digit}^* \ \text{digit}^*)^* \end{array}$$

The subexpressions *character* and *digit* that simply describe the set of recognized characters and the digits '0'–'9' have been left unspecified. It is clear from this example that using descriptive nonterminals may clarify the logical structure of a text database. Although nonperiodic grammars are

[1] Nonperiodic grammars are usually called *nonrecursive*. We use the term nonperiodic to avoid confusing nonperiodic matching problems with nonrecursive, i.e., undecidable problems.

too weak for modeling programming languages, they are powerful enough to model the structure of most text databases. In practice, nonperiodic grammars with regular expressions in productions support modeling long lists of, say, dictionary articles, but unlimited nesting of structures is of course not possible.

Note that if T is a parse tree over a nonperiodic grammar G, then the height of T is at most $|\mathcal{V}| + 1$. It is known that restricting the height of the pattern improves the running time of the basic algorithm for classical tree pattern matching:

Lemma 5.1 [DGM90] If the height of the pattern is h, then the basic algorithm for tree pattern matching takes time $O(nh)$. $\square$

In a matching problem the height of P is at most the height of T. Thus nonperiodic tree pattern matching can be solved using the basic algorithm in time $O(|\mathcal{V}|\, n)$. Next we show how Lemma 4.35 makes it possible to solve the tree pattern matching problem in time $O(kn)$ for a k-periodic target, and hence G-grammatical tree pattern matching in time $O(n)$ for an arbitrary nonperiodic grammar G.

Lemma 5.2 A k-thin candidate set of occurrences for any tree inclusion problem can be computed in a k-periodic target in time $O(n)$.

Proof. The nodes of the target which have the same label as the root of the pattern form a candidate set of occurrences; they can be located by a simple traversal of the target. The set is k-thin, because the target is k-periodic. $\square$

Now the next theorem follows directly from Lemmas 4.35, 5.2, and 4.36.

Theorem 5.3 The ordered child inclusion problem with k-periodic targets can be solved in time $O(kn)$. $\square$

Theorem 5.3 tells especially that nonperiodic tree pattern matching is solvable in linear time. To obtain the same result for nonperiodic ordered tree inclusion, we show in the next section that the instance relation "U includes P with order minimally" is solvable in linear time, when U is a nonperiodic tree.

5.2 Solving ordered tree inclusion on non-periodic targets

Next we give an algorithm for testing the instance relation of ordered tree inclusion between a pattern P and a nonperiodic tree U. The nonperiodicity of U means that if v is a node of U then there are no nodes labeled by $label(v)$ in the descendants of v. This implies two further facts utilized in the algorithm. First, tree U includes P with order minimally if and only if there is a root preserving ordered embedding of P into U. Second, let N be a nonempty set of nodes of U all of which have the same label. Then the first node of N in the preorder of U and the first node of N in the postorder of U are the same node.

Let $P_1, \ldots, P_k$ be the immediate subtrees of the pattern P rooted by nodes $u_1, \ldots, u_k$. The principle of the algorithm is the same as in Algorithm 4.23: to search for a left embedding of $\langle P_1, \ldots, P_k \rangle$ in U.

Algorithm 5.4 [KM92] Testing the instance relation of ordered tree inclusion with a nonperiodic target.

Input: Trees P and U, where U is nonperiodic.

Output: **true** if U includes P with order minimally; otherwise **false**.

Method: **if** $label(root(P)) = label(root(U))$ **then**
  return emb($root(P), root(U)$);
 else return false fi;

1. **function** emb(u, v);
2. **if** u is a leaf **then return true**;
3. **else** Let $u_1, \ldots, u_k$ be the children of u;
4.   Let p be the first descendant of v in preorder such that $label(p) = label(u_1)$;
   if there is no such node p **then return false**; **fi**;
5. $i := 1$;
6.   **while** $i \leq k$ **do**
7.   **if** emb(u_i, p) **then** $i := i + 1$; **fi**;
8.   **if** $i \leq k$ **then**
9.   Let p be the first node in $rr(p) \cap desc(v)$ in preorder of U such that $label(p) = label(u_i)$;

```
                    if there is no such p then return false; fi;
10.               fi;
11.           od;
12.           return true;
13.       fi;
14.   end;
```

Lemma 5.5 Algorithm 5.4 tests the relation "U includes P with order minimally" correctly for all trees P and all nonperiodic trees U.

Proof. If P consists of a single node u, the claim is obvious. Then assume that the height of P is $h > 0$ and and that the algorithm works correctly with all patterns of height less than h. Let the immediate subtrees of P rooted by $u_1, \ldots, u_k$ be $P_1, \ldots, P_k$. Now the following two invariants can be shown to hold for the loop on lines 6–11. First, before each execution of the loop, $post(p) \leq post(f(u_i))$ for all ordered embeddings f of $\langle P_1, \ldots, P_i \rangle$ into the forest of immediate subtrees of U. Second, after each execution of the loop, the forest $\langle P_1, \ldots, P_{i-1} \rangle$ has a left embedding into the forest of immediate subtrees of U. The correctness of the algorithm follows from these invariants. □

Lemma 5.6 The instance relation "U includes P with order minimally" is linearly solvable for nonperiodic trees U.

Proof. Algorithm 5.4 tests the relation for nonperiodic trees U correctly by Lemma 5.5. We show that there is a constant c such that the algorithm works in time bounded by $c|U|$ for all trees P and U.

Denote by $t(n)$ the maximum time needed to compare the root labels of P and U and to perform the function call emb($root(P), root(U)$), when P and U are trees and $|U| = n$. First, consider testing a single node. Obviously there is a constant c' such that $t(1) \leq c'$. Then assume that $|U| = n > 1$ and $t(m) \leq c'm$ for all targets of size $m < n$. Let n'' be the number of nodes of U that are examined during the traversal on lines 4 and 9 of Algorithm 5.4, excluding the roots of the subtrees of U that are examined in the recursive calls on line 7. Let n' be the total size of the subtrees of U that are examined in the recursive calls. There is a constant c'' such that the traversal can be performed in time $c''n''$. Therefore we get for $n > 1$

$$t(n) \leq c'n' + c''n''$$

Now, since the regions of U examined outside and in the recursive calls do not overlap, $n' + n'' \leq n$. From this we get for all $n > 0$

$$t(n) \leq cn$$

by selecting $c = \max\{c', c''\}$. $\square$

Theorem 5.7 [KM92] The ordered tree inclusion problem can be solved on nonperiodic targets in $O(n)$ time.

Proof. By Lemma 5.2 a flat candidate set of occurrences can be computed in $O(n)$ time, and by Lemma 5.6 the instance relation is linearly solvable. Therefore the problem is solvable in linear total time by Lemma 4.35. $\square$

Periodic targets seem to be more difficult to handle. Trying to apply the approach of Algorithm 5.4 to testing the instance relation with k-periodic targets leads to backtracking with $\Omega(2^k n)$ worst case running times. This means that the specialized approach of this section would be feasible for k-periodic targets if $k < \log m$.

5.3 Preprocessing grammatical patterns

In this section we consider further utilizing the grammar of the target in connection to tree inclusion problems. First, the patterns can be checked syntactically against the grammar in order to test whether they can have occurrences in the target. Second, in some cases the grammar can be used for transforming the problem to an easier but equivalent problem.

In the nonperiodic tree inclusion problems considered above obviously only nonperiodic patterns can have occurrences in the target. More generally, in a G-grammatical inclusion problem one can check, before performing the actual matching, whether P can have an occurrence in a parse tree over G. For example, in a text database application query patterns not passing this test could result in informative diagnostics about the impossibility of locating data using such patterns on a database modeled by grammar G.

An alternative and a more co-operative way of ensuring that only sensible patterns are given through a query interface is to provide the user with *pattern templates*, which the user can edit. This kind of an interface has been implemented in the LQL system [Byr89]. Next we sketch what conditions the patterns have to meet, whether their correct form is guaranteed by a template editor or whether they are checked separately.

In what follows, we can assume that the grammar $G = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ of the target contains only useful nonterminals. That is, every nonterminal in $\mathcal{V}$ appears in a derivation of a string of terminals from the start symbol S of G. (See [HU79, p. 88–90].)

For tree pattern matching one checks that each node of pattern P is labeled by a symbol of grammar G and that the children of each internal node of P correspond to an appropriate production in G. Let u be an internal node of P with label A and with children $u_1, \ldots, u_k$. Checking u is performed by applying to the string of labels in $u_1, \ldots, u_k$ an automaton that recognizes the instances of the right-hand sides of the productions for nonterminal A. These finite automata need to be constructed only once for a grammar. If the automata are deterministic, checking pattern P takes only time $O(|P|)$. If the worst case $O(2^{|G|})$ size of the deterministic automata is prohibitive, it is also possible to construct nondeterministic finite automata for the same task; this can be done in time $O(|G|)$. The children of each pattern node can then be checked by simulating the NFAs, yielding total time $O(|G||P|)$. (See [AHU74].)

For the tree inclusion problem the condition is slightly more complicated. For each node of P labeled by A and having children labeled by $a_1, \ldots, a_k$ the following should hold in G:

$$A \stackrel{*}{\Rightarrow} \beta_0 a_1 \beta_1 \ldots \beta_{k-1} a_k \beta_k \ ,$$

where $\beta_i \in (\mathcal{V} \cup \mathcal{T})^*$. This can be checked in the following manner. For each nonterminal $B \in \mathcal{V}$ let B' be a unique terminal not belonging to $\mathcal{T}$, and for a set of nonterminals N let $N' = \{A' \mid A \in N\}$. For each terminal $t \in \mathcal{T}$ let t' be a unique new nonterminal not belonging to $\mathcal{V}$, and for a set of terminals C let $C' = \{t' \mid t \in C\}$. For a production p denote by p' the production obtained by replacing each terminal t in p by t'. For a set of productions Q denote by Q' the set $\{p' \mid p \in Q\}$. Finally, for grammar $G = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ and a nonterminal $A \in \mathcal{V}$ let $G'_A = (\mathcal{V}'', \mathcal{T}'', \mathcal{P}'', S'')$ be the grammar with

$$\begin{aligned}
\mathcal{V}'' &= \mathcal{V} \cup \mathcal{T}' \\
\mathcal{T}'' &= \mathcal{T} \cup \mathcal{V}' \\
\mathcal{P}'' &= P' \\
&\quad \cup \{A \to (A' \mid \epsilon) \mid A \in \mathcal{V}\} \\
&\quad \cup \{t' \to (t \mid \epsilon) \mid t \in \mathcal{T}\} \\
S'' &= A
\end{aligned}$$

The idea is that grammar G'_A generates the subsequences of the sentential forms that are derivable from nonterminal A in grammar G. (Note that G'_A and G'_B may differ only with regard to the start symbol.) Now checking a pattern node labeled by A and having children labeled by $a_1, \ldots, a_k$ is done by first substituting B' for each nonterminal B in the sequence $a_1, \ldots, a_k$ and then parsing this sequence using G'_A.

Above we have outlined possibilities to check the patterns against the grammar before performing the actual matching. Another promising direction for preprocessing patterns with regard to the grammar, analogical to query optimization in databases, is trying to transform the given inclusion problem to an easier one that still yields the same set of occurrences as the original problem.

For some patterns P and grammars G we may be able to compute a unique sequence of labels on any path between two nodes labeled by a and b in any parse tree over G, when a and b are labels of a node u and its descendant v in pattern P. Such knowledge allows us to complete the ordered tree inclusion pattern by adding nodes labeled by the corresponding sequence of labels between every node-descendant pair u, v in P, and to solve the problem, possibly more efficiently as an ordered path inclusion problem. Another problem that is feasible for this transformation is unordered tree inclusion. It would be convenient to ignore the left-to-right order of subtrees in expressing queries on a grammatical database. Unfortunately, unordered tree inclusion is an NP-complete problem. (See Section 4.2.) If the unordered tree inclusion pattern P and grammar G allow transforming P as above, the problem reduces to an unordered path inclusion problem, which is solvable in polynomial time.

Chapter 6

Tree inclusion with logical variables

In this chapter we consider *tree inclusion with logical variables.* In this extension of tree inclusion the pattern leaves can be labeled by variable symbols. This provides means for extracting subtrees of the target using substitutions to the variables in the pattern. Moreover, labeling various pattern nodes by the same variable symbol is a way of requiring identity of the subtrees matched by those nodes.

The addition of logical variables makes most of our problems NP-hard; only the ordered child inclusion problem and the ordered subtree problem retain their polynomial complexities. The inclusion problems with logical variables are defined in Section 6.1. The NP-hardness proofs are given in Section 6.2. Section 6.3 outlines how the ordered child inclusion problems, i.e., the classical tree pattern matching and the ordered subtree problem with logical variables, can be solved in polynomial time. Some polynomial subclasses of ordered tree inclusion with logical variables are considered in Section 6.4. Section 6.5 is a summary of the computational complexities of the tree inclusion problems with logical variables.

6.1 Logical variables in patterns

So far we have considered tree inclusion problems where the patterns and the targets are trees labeled by symbols without any specific semantics. Now we assume that the set of labels consists of disjoint sets Const of *constant symbols* and Var of *variable symbols*, and extend the inclusion problems by allowing pattern leaves to be labeled by variables.

We denote constants by letters a, b, c, …, and variables by letters x, y, z, …, possibly with subscripts. A node in a pattern is called a *variable node* if it is labeled by a variable symbol; otherwise it is a *constant node*. In both cases we may call nodes labeled by a symbol c simply *c-nodes*. A *pattern with logical variables* is a tree whose internal nodes are constant nodes and whose leaves are labeled by symbols in Const $\cup$ Var.

Let tree $P = (V, E, root(P))$ be a pattern with logical variables and $T = (W, F, root(T))$ a tree. An injective function $f : V \to W$ is a *pattern embedding* of P into T if it preserves ancestorship and the labels of constant nodes, and *respects variables*. Function f respects variables if it maps nodes labeled with identical variables to roots of identical subtrees. That is, f is a pattern embedding, if for all nodes u and v of P we have that

1. $f(u) = f(v)$ if and only if $u = v$,
2. u is an ancestor of v in P if and only if $f(u)$ is an ancestor of $f(v)$ in T,
3. if $label(u) \in$ Const then $label(u) = label(f(u))$, and
4. if $label(v) \in$ Var and $label(u) = label(v)$ then the trees $T[f(u)]$ and $T[f(v)]$ are identical.

Ordered pattern embeddings, *path pattern embeddings*, *region pattern embeddings*, and *child pattern embeddings* are defined analogously to the corresponding tree embeddings in Chapter 3.

Let $G = (V, E)$ and $H = (W, F)$ be directed and labeled graphs. A bijection $f : V \to W$ is a *pattern isomorphism* between G and H, if for all nodes $u, v \in V$

1. $(u, v) \in E$ if and only if $(f(u), f(v)) \in F$,
2. $label(v) = label(f(v))$ if $label(v) \in$ Const, and
3. $label(f(u)) = label(f(v))$ if $label(v) \in$ Var and $label(u) = label(v)$.

A pattern isomorphism between a pattern P with logical variables and a tree T is a *subtree pattern embedding* of P in T. If a subtree pattern embedding preserves the left-to-right order of the nodes, it is an *ordered subtree pattern embedding*.

The *tree inclusion problems with logical variables* are now derived from the corresponding tree inclusion problems simply by replacing the embeddings by the corresponding pattern embeddings. We refer for shortness to ordered and unordered C-embeddings, C-inclusion problems, C-pattern embeddings, and C-inclusion problems with logical variables, where "C-" is empty or stands for a phrase from the list "path", "region", "child", "subtree". The abbreviations for a particular tree inclusion problem with logical variables is produced by suffixing the initials of the corresponding tree inclusion problem by "-V" (referring to "with variables"). For example, the abbreviation of the ordered tree inclusion problem with logical variables is OTI-V, which stems from Ordered Tree Inclusion with Variables. We can further refer to the decision versions ("Is there a C-embedding of P in T?") of the problems by attaching the prefix "D-" to the problem identifiers.

A *binding* is a pair consisting of a variable x and a term t, denoted normally by $x = t$. A *substitution* is a set of bindings $x_i = t_i$, where the variables x_i are distinct. Let $P = (V, E, root(P))$ be a pattern with logical variables and let T be a tree. If f is an ordered (unordered) C-pattern embedding of P into T, the substitution

$$\{x = t \mid x \in \mathrm{Var},\ \exists\, v \in V\ : label(v) = x \text{ and } t = Term(T[f(v)])\}\ ,$$

is called a *solution* to the instance (P, T) of the ordered (unordered) C-tree inclusion problem with logical variables. That is, a solution is a substitution that binds each variable x occurring in the pattern to the term representation of the subtrees matched by the x-nodes. The *answer* to the instance (P, T) of an ordered (unordered) C-inclusion problem with logical variables Π is the set of solutions to the instance (P, T) of Π. We often refer to the instances of problems as "problems". If there is no ordered (unordered) C-embedding of P into T, the answer to the problem (P, T) is the empty set. On the other hand, if P is an ordered (unordered) C-included pattern of T but no nodes of P are labeled by variables, the answer is the singleton set consisting of the empty substitution.

As an example, consider the pattern P and the target T shown in Figure 6.1. Since permutation of siblings does not change the pattern, the answers to the ordered and unordered problems are the same. The answer to the instance (P, T) of the tree inclusion problems with logical variables OTI-V and UTI-V is $\{\{x = b\}, \{x = c\}\}$, which contains two solutions. In the case of the path inclusion problems OPI-V and UPI-V the answer is $\{\{x = b\}\}$. If (P, T) is an instance of region inclusion with logical variables

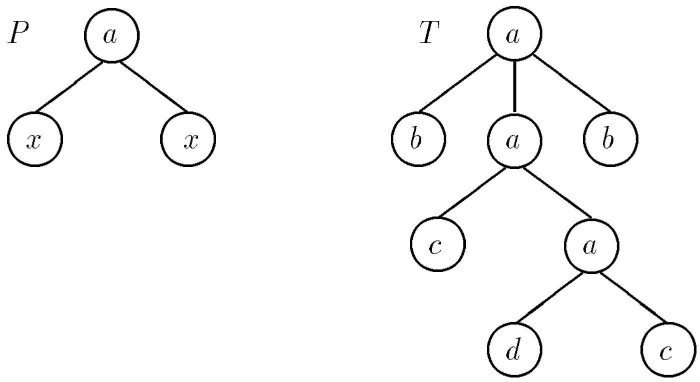


Figure 6.1: An instance of tree inclusion problems with logical variables.

ORI-V or URI-V, the answer is the empty set. This is because the target T does not contain two identical subtrees rooted by adjacent sibling nodes.

We obtain the following relationships between the answers to different tree inclusion problems with logical variables analogically to Section 3.11.

Theorem 6.1 Let tree P be a pattern with logical variables and T a tree. Then the following inclusions hold between the answers to the tree inclusion problems with logical variables:

1. The answer to the ordered (unordered) path inclusion problem with logical variables (P, T) is a subset of the answer to the ordered (unordered) tree inclusion problem with logical variables (P, T).

2. The answer to the ordered (unordered) region inclusion problem with logical variables (P, T) is a subset of the answer to the ordered (unordered) path inclusion problem with logical variables (P, T).

3. The answer to the ordered (unordered) child inclusion problem with logical variables (P, T) is a subset of the answer to the ordered (unordered) region inclusion problem with logical variables (P, T).

4. The answer to the ordered (unordered) subtree problem with logical variables (P, T) is a subset of the answer to the ordered (unordered) child inclusion problem with logical variables (P, T). $\square$

Further, each ordered C-pattern embedding of P into T is also an unordered C-pattern embedding of P into T.

Theorem 6.2 Let tree P be a pattern with logical variables and T a tree. Then the answer to the ordered C-inclusion problem with logical variables (P, T) is a subset of the answer to the unordered C-inclusion problem with logical variables (P, T). □

6.2 Complexity of inclusion problems with logical variables

In this section we show that most of the tree inclusion problems with logical variables are NP-hard in the general case. The intuitive reason for the difficulty of these problems is common to many NP-complete problems: it is difficult to do local choices that are globally consistent. In the case of tree inclusion with logical variables the requirement for global consistency appears in the requirement that embeddings respect variables.

Also string pattern matching becomes difficult if we allow repeating variables in the pattern. Angluin has shown in [Ang80] that it is NP-complete to decide whether a given string can be obtained by substituting strings to the logical variables in a given string pattern. In string pattern matching the difficulty lies in deciding the right lengths for the substrings matched by the variables.

A tree inclusion problem with logical variables may have exponentially many solutions. Consider the ordered tree inclusion problem with logical variables (P, T) with

$$P = \mathit{Tree}(a(x_1, \ldots, x_m)), \text{ and } T = \mathit{Tree}(a(b_1, \ldots, b_n)),$$

where $x_1, \ldots, x_m$ are distinct variable symbols and $b_1, \ldots, b_n$ are distinct constant symbols. The answer to the problem consists of $\binom{n}{m}$ substitutions. Thus tree inclusion problems with logical variables may require in general exponential time in the size of the input.

The decision problems "Is there a C-pattern embedding of P into T?" are in NP, since guessing a mapping and checking whether it is a C-embedding can be done in polynomial time with respect to the size of input.

If no variable appears twice in the pattern, the decision versions of tree inclusion problems with logical variables are not harder than the corresponding tree inclusion problems. They can be solved by straightforward variants of the corresponding tree inclusion algorithms by simply ignoring the labels of the variable nodes.

Next we show that all unordered inclusion problems with logical variables are NP-hard. The proof is based on a pseudo-polynomial reduction from 3-PARTITION.

A decision problem Π is *NP-hard in the strong sense* if a variant of Π where all input numbers are expressed in unary notation is NP-hard. A decision problem Π is *NP-complete in the strong sense* if Π is NP-hard in the strong sense and $\Pi \in$ NP. A *pseudo-polynomial transformation* from a problem Π to a problem Π' is a transformation from Π to Π' that can be computed in time limited by a polynomial over both the length of the instance of Π and the magnitude of the largest number appearing in the instance of Π. For a more rigorous development of these concepts, see [GJ78] or [GJ79].

Lemma 6.3 [GJ79] If Π is NP-hard in the strong sense and there exists a pseudo-polynomial transformation from Π to Π', then Π' is NP-hard in the strong sense. □

Problem 14 (3-PARTITION)

Instance: A set $A = \{a_1, \ldots, a_{3k}\}$, a positive integer bound B and a positive integer size $s(a)$ for each element a of A, such that each size $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = kB$.

Question: Is there a partition of A into disjoint sets $A_1, \ldots, A_k$ such that $\sum_{a \in A_i} s(a) = B$ for each $i = 1, \ldots, k$? □

The constraints of the above problem imply that each subset A_i in the partition $A_1, \ldots, A_k$ contains exactly three elements of A; hence the name 3-PARTITION.

Theorem 6.4 [GJ79] 3-PARTITION is NP-complete in the strong sense. □

We give a pseudo-polynomial transformation from 3-PARTITION to the unordered subtree problem with logical variables (UST-V). Let an instance of 3-PARTITION be given by the set $A = \{a_1, \ldots, a_{3k}\}$, the positive integer bound B and the positive integer sizes $s(a)$ for each element a of A. We represent the instance of 3-PARTITION as an instance of tree inclusion with logical variables as follows. Set A is represented by the leaves of a pattern

$P = (V, E, root(P))$ consisting of a root and kB leaves. Each element a_i of A is represented by a unique variable symbol x_i, called the *label* of a_i. For each element a_i of A we label $s(a_i)$ leaves of P by the label x_i of a_i. The root of P is labeled by 0. A target $T = (W, F, root(T))$ consisting of a root and kB leaves represents the partitioning of A. The leaves of T are divided into k groups of B nodes; the nodes in group i are labeled by i. The root of T is labeled by 0. (See Figure 6.2.)

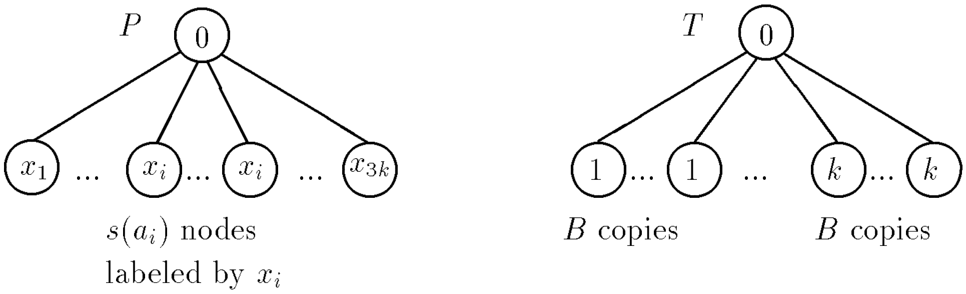


Figure 6.2: Trees P and T corresponding to an instance of 3-PARTITION.

Lemma 6.5 Let a set $A = \{a_1, \ldots, a_{3k}\}$ with a size $s(a)$ for each of its elements a and a bound B form an instance of 3-PARTITION. Let (P, T) be the representation of the instance of 3-PARTITION as an instance of the unordered subtree problem with logical variables. Then there is a pattern isomorphism between P and T if and only if there is a partition of A into disjoint sets $A_1, \ldots, A_k$ such that $\sum_{a \in A_i} s(a) = B$ for each $i = 1, \ldots, k$.

Proof. Let $W_i = \{w \in W \mid label(w) = i\}$ be the set of the nodes of T that are labeled by i. Then $|W_i| = B$ for $i = 1, \ldots, k$.

First assume that there is a partition of A into disjoint sets $\{A_1, \ldots, A_k\}$ such that $\sum_{a \in A_i} s(a) = B$ for each $i = 1, \ldots, k$. For $i = 1, \ldots, k$, let

$$V_i = \{v \in V \mid label(v) = x_j \text{ and } a_j \in A_i\}$$

be the set of the nodes of P that are labeled by the label x_j of some element $a_j \in A_i$. Sets $V_1, \ldots, V_k$ form a partition of the leaves of P in disjoint subsets. For each element a_j there are $s(a_j)$ leaves of P labeled by x_j. Since $\sum_{a \in A_i} s(a) = B$, we have that $|V_i| = B$ for each $i = 1, \ldots, k$. Therefore for each $i = 1, \ldots, k$ there is a bijection f_i between the sets V_i and W_i. Now it is easy to see that the function

$$\bigcup_{i=1}^{k} f_i \cup \{(root(P), root(T))\}$$

is a pattern isomorphism between P and T.

Assume for the converse that there is a pattern isomorphism f between P and T. Let $V_i = \{v \in V \mid f(v) \in W_i\}$ be the set of the nodes of P that f maps to nodes of T labeled by i. For all the leaves v of P denote the label of v by x_v. For each $i = 1, \ldots, k$ let $L_i = \{x_v \in \mathrm{Var} \mid v \in V_i\}$ be the set of variables that are labels of nodes in set V_i. Because f is a variable respecting function from the nodes of P to the nodes of T, the sets $L_1, \ldots, L_k$ form a partition of the set of variables occurring in P. For each $i = 1, \ldots, k$ let

$$A_i = \{a_j \in A \mid x_j \in L_i\}$$

be the set of elements a_j of A whose label x_j belongs to set L_i. Now it is clear that the sets $A_1, \ldots, A_k$ form a partition of A.

Then we show that the total size $\sum_{a \in A_i} s(a)$ of each subset A_i in the partition $A_1, \ldots, A_k$ of A equals B. The leaves of P are labeled so that $s(a_j) = |\{v \in V \mid label(v) = x_j\}|$ for each element a_j of A. Since two sets of pattern nodes labeled by the labels of two different elements of A are disjoint we see that

$$\sum_{a_j \in A_i} s(a_j) = |\bigcup_{a_j \in A_i} \{v \in V \mid label(v) = x_j\}|$$

for each $i = 1, \ldots, k$. Now for each $i = 1, \ldots, k$ we have

$$\begin{aligned} \bigcup_{a_j \in A_i} \{v \in V \mid label(v) = x_j\} &= \bigcup_{x_j \in L_i} \{v \in V \mid label(v) = x_j\} \\ &= \{v \in V \mid x_v \in L_i\} \\ &= V_i \ . \end{aligned}$$

Because f is a bijection, $|V_i| = |W_i| = B$ for each $i = 1, \ldots, k$. $\square$

Note that in the above construction any pattern embedding of P into T is also a pattern isomorphism since both P and T consist of a root node and kB leaves. Therefore we have shown that none of the unordered inclusion problems with logical variables is easier than 3-PARTITION.

Theorem 6.6 The unordered inclusion problems with logical variables are NP-hard. □

Benanav, Kapur, and Narendran have derived closely related results for matching terms built of logical variables and symbols in a ranked alphabet [BKN87]. Especially, they show that *commutative matching* of terms is NP-complete.

We have studied mainly trees as representations for terms. Terms can be represented concisely as directed graphs by sharing the representations of some identical subterms. Applying this idea to the trees of the previous transformation leads to a proof of the NP-completeness of a version of GRAPH HOMOMORPHISM. Given two graphs G and H, the problem of GRAPH HOMOMORPHISM is to decide whether a graph isomorphic to H can be obtained from G by a sequence of identifications of non-adjacent vertices. The effect of identifying two vertices u and v is to replace them by a single new vertex that is adjacent to all the vertices that were previously adjacent to u or v. It is well known that GRAPH HOMOMORPHISM is an NP-complete problem [GJ79].

A *multigraph* is a graph which may have more than one edge between two vertices. Two multigraphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are *isomorphic* if there are bijections $f : V_G \rightarrow V_H$ and $g : E_G \rightarrow E_H$ such that for all vertices u and $v \in V_G$ and all edges $e \in E_G$ we have that e is an edge from u to v if and only if $g(e)$ is an edge from $f(u)$ to $f(v)$. A graph is *acyclic* if none of its nodes can be reached by following a nonempty sequence of edges starting from the node itself. Let u be a vertex in a directed graph G. If u has no incoming edges and all vertices of G are reachable from u we say that u is the *root of graph* G.

Now we can show that GRAPH HOMOMORPHISM is NP-complete also in the version where both graphs G and H are directed acyclic multigraphs with a single root, and whose longest paths contain only two nodes. The proof is rather similar to the previous reduction. Let an instance of 3-PARTITION be given by a set $A = \{a_1, \ldots, a_{3k}\}$, a positive integer bound B and a positive integer size $s(a)$ for each element a of A. Then we build a multigraph G out of a root and a distinct node v_a for each element a of A; v_a is pointed to by $s(a)$ edges starting from the root of G. Multigraph H is built out of a root and k other nodes; each of the non-root vertices of H are pointed to by B edges starting from the root of H. Now it is easy to see that there is a partition of A in k disjoint subsets, each having total size B if and only if G can be made isomorphic to H by identifying non-adjacent vertices of

G. (Identifying two vertices v_{a_1} and v_{a_2} of G corresponds to choosing the elements a_1 and a_2 of A to the same subset in the partition.)

The proof of NP-hardness of the ordered inclusion problems with logical variables OTI-V, OPI-V and ORI-V is by a reduction from the NP-complete problem 3-SATISFIABILITY [GJ79].

Problem 15 (3-SATISFIABILITY, 3SAT)
Given a collection $C = \{c_1, \ldots, c_k\}$ of clauses, each consisting of three literals over a finite set of variables U, decide whether there is a truth assignment for U that satisfies each clause in C. □

The problem 3SAT can be reduced in polynomial time to ordered tree inclusion with logical variables, ordered path inclusion with logical variables, and ordered region inclusion with logical variables. Let an instance of 3SAT be given by the set of clauses $C = \{c_1, \ldots, c_k\}$, where $|c_i| = 3$ for all $i = 1, \ldots, k$. We represent C as the following instance (P, T) of ordered tree inclusion with logical variables. Pattern tree P consists of a root node labeled by 0 and of k immediate subtrees $P_1, \ldots, P_k$. Each pattern subtree P_i is a straightforward representation of clause c_i; for an example see Figure 6.3.

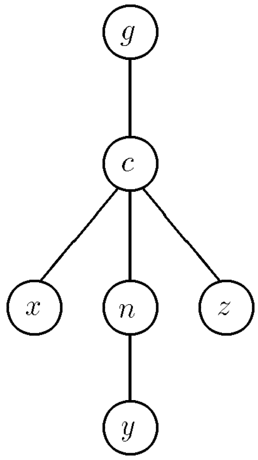


Figure 6.3: The pattern subtree representing clause $\{x, \bar{y}, z\}$.

The target tree T consists also of a root node labeled by 0 and of k immediate subtrees $T_1, \ldots, T_k$. Target subtree T_i is a representation for the group of clauses consisting of the seven instances of clause c_i that evaluate to true. Each immediate subtree of T_i is obtained from the immediate subtree of pattern subtree P_i by substituting zeros and ones for its variables. An example is shown in Figure 6.4.

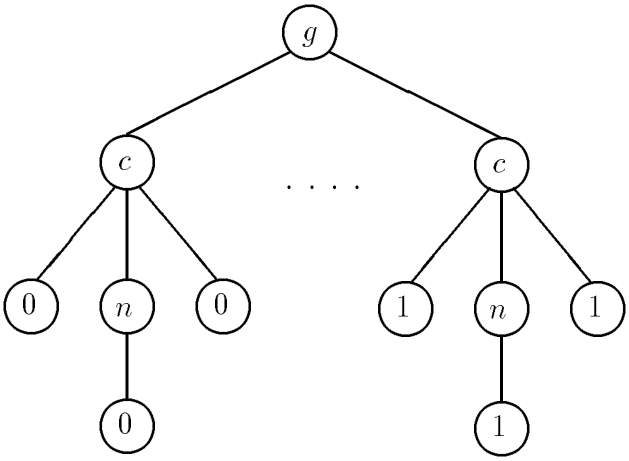


Figure 6.4: The target subtree representing the true instances of clause $\{x, \bar{y}, z\}$.

Now it is easy to check that there is an ordered pattern embedding of P in T if and only if C is satisfiable. Further, there is an ordered pattern embedding of P in T if and only if there is an ordered path pattern embedding of P in T, which holds if and only if there is an ordered region pattern embedding of P in T. We have derived the following result.

Theorem 6.7 The problems

1. ordered tree inclusion with logical variables (OTI-V),
2. ordered path inclusion with logical variables (OPI-V), and
3. ordered region inclusion with logical variables (ORI-V)

are NP-hard. □

6.3 Ordered child inclusion with logical variables

In this section we outline how the ordered child inclusion problems with logical variables OCI-V and OST-V can be solved in the same asymptotic time as the corresponding inclusion problems OCI and OST. We considered solving the ordered child inclusion problem OCI in Section 4.9 and the subtree problems, including the ordered subtree problem OST, in Section 4.10. The new feature in the problems with logical variables is the need to consider only embeddings that respect variables.

Let P be a pattern tree with logical variables and let T be a target tree. In order to test whether an embedding of P in T respects variables we must check whether the subtrees of the target that are matched by similarly labeled variable nodes are identical. Checking whether trees $T[u]$ and $T[v]$ are identical can be done in time $O(\min\{|T[u]|, |T[v]|\})$ by simply comparing the trees node-by-node against each other. If we have preprocessed target T we can decide in constant time whether the trees rooted by two nodes of T are identical. Indeed, repeated occurrences of identical subtrees can be represented by links to a single representative of those trees, leading to a representation of T as a directed acyclic graph. We call the link to the representative of a tree U the *signature* of U. Recognizing identical substructures is called *common subexpression elimination.*[1] Common subexpression elimination in a tree T can be performed in time $O(|T|)$ [DST80]. (See also [FSS90].)

The image of the root of P determines unambiguously the images of all the other pattern nodes in an ordered child embedding of P in T. This means that there are at most n ordered child pattern embeddings into a target of size n. Assume that common subexpression elimination has been performed in the target T. The basic tree pattern matching method can be extended to check the consistency of variable matches simultaneously with the top-down traversal of the pattern. This can be done for example by using an array that stores the bindings for each variable symbol in the pattern. When the algorithm compares the first node labeled by a variable x against a target node u, it stores the signature of $T[u]$ in the slot of x. Then the later comparisons between pattern nodes labeled by x and target nodes v are performed by comparing the signature stored in the slot of x and the signature of $T[v]$. Clearly this requires a total $O(m)$ time at each of the $O(n)$ potential occurrences of the pattern. These considerations lead to the following result.

Theorem 6.8 Ordered child inclusion with logical variables can be solved in $O(mn)$ time. □

---

[1]This process is employed for example in optimizing programming language compilers to produce efficient code for evaluating arithmetic expressions [AU77].

Ramesh and Ramakrishnan call ordered child inclusion with logical variables *nonlinear tree pattern matching.* They have presented in [RR88] and in [RR92] an algorithm for the problem working in time $O(s(P)n)$, where $s(P)$ is the suffix index of the pattern. As discussed in Section 4.9, the suffix index is $\Theta(m)$ in the worst case.

Next we consider solving the ordered subtree problem with logical variables. Let P be a pattern with logical variables consisting of m nodes, and let T be a target tree consisting of n nodes. In subtree problems the pattern instances are of the same size as the pattern. Therefore we can form a flat set of candidate occurrences in $O(n)$ time by selecting those target subtrees that have m nodes. The instance relation of the ordered subtree problem with logical variables is tested between P and a tree U by comparing P and U node-by-node against each other; constant nodes have to be identical and pattern leaves labeled by identical variables have to match at identical target leaves. The method outlined above for checking the consistency of variable matches can be applied also here. Therefore we see that the instance relation of ordered subtree problem with logical variables is linearly solvable. Lemma 4.35 gives then the following result.

Theorem 6.9 The ordered subtree problem with logical variables is solvable in $O(n)$ time. □

6.4 Polynomial subclasses of ordered tree inclusion with logical variables

In this section we consider some cases where the decision version of the ordered pattern inclusion problem D-OTI-V is solvable in polynomial time. We focus on ordered inclusion since we believe it to be a feasible operation for practical data retrieval. In Chapter 7 we show how ordered tree inclusion with logical variables can be applied to querying structured text databases.

Let tree P be a pattern with logical variables. A variable symbol x is *repeating* in P if there are at least two nodes labeled by x in P. A pattern node that is labeled by a non-repeating variable matches at every node. This rule is easy to add to the tree inclusion algorithms of Chapter 4; therefore the decision versions of inclusion problems with logical variables are solvable in the same time as the corresponding tree inclusion problems, as long as no variable occurs more than once in the pattern. We assume in the sequel that the non-repeating variable nodes are handled this way in the algorithms.

A strategy for solving inclusion problems with logical variables is first to substitute signatures of target subtrees to the variables of the pattern (see Section 6.3), and then apply the corresponding tree inclusion algorithm. If the number of different variables occurring in P is limited by a constant k, the ordered decision problems can be solved in polynomial time: Consistent substituting produces at most n^k different instantiations of the pattern, and each of them can be handled in time that is polynomial in m and n using algorithms for ordered tree inclusion problems.

There are some weaker restrictions to ordered tree inclusion with logical variables that also result in polynomially solvable problems. The general strategy is to consider cases where the effect of variable bindings is localized, i.e., the interaction between the substitutions to different variables of the pattern is limited. One way to achieve this is to consider *separable patterns*.

Let F be a forest (or a tree). Recall from Section 4.7 that a sibling interval of F is a nonempty sequence $\langle u_1, \ldots, u_k \rangle$ of nodes of F, where node u_{i+1} is the next sibling of u_i for each $i = 1, \ldots, k-1$. Forest $\langle T_1, \ldots, T_k \rangle$ is a *sibling forest* in F, if there is a sibling interval $\langle u_1, \ldots, u_k \rangle$ of F such that $T_i = F[u_i]$ for each $i = 1, \ldots, k$. For each pattern P and each variable x appearing in P there is a unique sibling forest F in P with the property that F contains all x-nodes but no sibling forest in F other than F itself contains all x-nodes. We call such a forest F the *x-forest of P*. A pattern P is *separable* if for any two distinct repeating variables x and y the x-forest and the y-forest of P are disjoint. Note that if F is an x-forest of a separable pattern P, then no variable label occurs both in F and in any sibling forest G in P that is disjoint from F; in such situations we say that the forests F and G are *variable disjoint*.

Let $\mathcal{E}$ be the set of all ordered pattern embeddings of a forest F of patterns with logical variables into a forest or tree G. Then the left embeddings of $\mathcal{E}$ are called *left pattern embeddings of F in G*. Now the previous results of Section 4.5 about constructing left embeddings extend easily to constructing left pattern embeddings of variable disjoint forests.

Theorem 6.10 Let F be a forest of patterns with logical variables and let T be a tree. There is an ordered pattern embedding of F in T if and only if there is a left pattern embedding of F in T. $\square$

Theorem 6.11 Let $F = \langle P_1, \ldots, P_k \rangle$ where $k \geq 2$ be a forest of patterns with logical variables and let T be a tree. Let $F_1 = \langle P_1, \ldots, P_i \rangle$ and $F_2 = \langle P_{i+1}, \ldots, P_k \rangle$ where $1 \leq i < k$ be two variable disjoint subforests of F. Let

f be a left pattern embedding of F_1 in T and let $\mathcal{E}$ be the set of such ordered pattern embeddings g of F_2 in T that $g(root(P_{i+1})) \in rr(f(root(P_i)))$. Then the following hold:

1. If $\mathcal{E}$ is empty, there is no ordered pattern embedding of F in T.

2. If $\mathcal{E}$ is nonempty and g is a left pattern embedding of $\mathcal{E}$, then $f \cup g$ is a left pattern embedding of F in T. $\square$

Assume that P is a separable pattern containing at least two repeating variables, and that T is a target tree. We outline an efficient method to decide whether there is an ordered pattern embedding of P in T. The overall idea is to compute left pattern embeddings for variable disjoint subforests of the pattern. This can be done applying Algorithm 4.23. The pattern nodes that do not belong to any variable forest are treated exactly as in Algorithm 4.23. Consider then the first node u in postorder of an x-forest F of P. Let p be the parent node of forest F and let the children of p be $p_1, \ldots, p_k$. Let the children p_i and p_j of p be the roots of the leftmost and the rightmost trees of forest F. The situation is depicted in Figure 6.5.

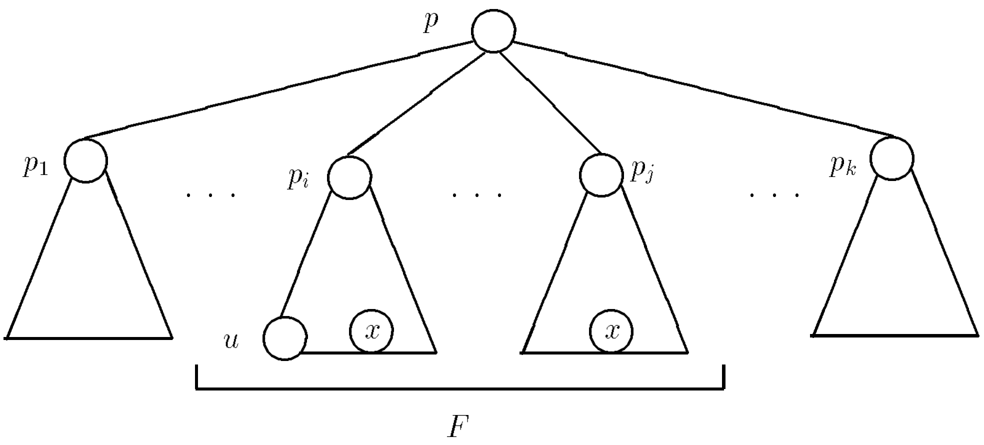


Figure 6.5: The context of an x-forest F in pattern P.

For each target node w we compute a left pattern embedding f_w of the forest $F' = \langle P[p_1], \ldots, P[p_j] \rangle$ in the subtrees of node w. This is done by substituting the signature s of each target subtree in turn for variable x in F, and applying Algorithm 4.23 to the resulting instance of F' and target T. The collection of left embeddings of all the instantiations of F' to the subtrees of node w is the collection of ordered pattern embeddings of F' to the subtrees of w, and f_w is a left embedding of this collection. Note that left embeddings for the forest $\langle P[p_1], \ldots, P[p_{i-1}]\rangle$ have already been computed, and that the only pattern nodes that are needed to compute f_w are p_{i-1} and the nodes of x-forest F.

The total time of applying Algorithm 4.23 to one instantiation of each variable forest of P is $O(mn)$. There are at most n different instantiations, which leads to total time $O(mn^2)$ for the variable forests of P. This computation dominates the $O(mn)$ time used for the pattern nodes outside the variable forests of P. We have the following result.

Theorem 6.12 For a given separable pattern P with logical variables and a tree T we can test in time $O(mn^2)$ whether there is an ordered pattern embedding of P in T. □

6.5 Summary of complexities

The table below summarizes the complexity results for the inclusion problems with logical variables. The notations are explained in Section 4.11.

	incl.	path incl.	region incl.	child incl.	subtree
ordered	NP-hard			$O(mn)$	$\Theta(n)$
unordered	NP-hard				

Table 6.1: The complexities of the inclusion problems with logical variables

We derived relative lower bounds for the complexities of many of the inclusion problems with logical variables. The NP-hard problems can be considered to be in general intractable. We showed in Section 6.4 that ordered tree inclusion with logical variables is solvable in polynomial time when the patterns are separable. One can search for other efficiently solvable subclasses of the problems. Another open question is the complexity of child inclusion with logical variables; both nontrivial lower bounds and more efficient algorithms for the problem could be searched for.

Chapter 7

Applying tree inclusion to structured text databases

In this chapter we show how tree inclusion can be applied to querying structured text databases. Using tree inclusion as a query primitive provides data independence and strong expressiveness with concise query syntax. The chapter is organized as follows. Section 7.1 reviews some methods that are used or proposed for modeling and retrieving computerized text data, and explains our idea of a structured text database. In Section 7.2 we explain how tree inclusion with logical variables can be applied to querying data in a structured text database, and give examples of its usage. We show that tree inclusion owns some power of recursive database queries, and that the primitive is robust with respect to certain modifications of the database. The evaluation of queries is discussed in Section 7.3.

7.1 Text databases

A *text database system* is a system for storing, editing, and querying text documents. In this section we review different approaches to modeling and querying text databases, and describe our idea of a text database.

Efficient locating and accessing of relevant text documents has been actively studied in the area known as *information retrieval (IR)* already since 1960's. Information retrieval has been mainly concerned with identifying those text documents in bibliographic databases that best satisfy the user's information needs. This identification traditionally happens by comparing a set of keywords given by the user against the contents of the database [SM83].

In IR the result of a retrieval is a set of *documents*. Documents can be for example books, scientific reports, or newspaper articles.

We can deviate from information retrieval by allowing more flexible querying and manipulating of sub-parts of the text instead of restricting it to fixed complete documents. This leads to approaches to modeling document databases that we classify as *text-based*, *grammatical*, and *hypertext*.

An obvious approach for storing text documents is to try to apply traditional data models and database techniques. Despite their many virtues, commercial database systems do not seem most appropriate for manipulating text documents, which often have complicated structure and contain free text of variable length. Enhancements to a relational database management system to support document processing have been proposed in [SSL+83].

A *text-based* system views text documents as strings. An example is the PAT™ free text search system. The PAT system has initially been developed for efficiently accessing the computerized version of the Oxford English Dictionary [Tom92]. The query language of the PAT system allows searching any substrings of the text stored in the system [ST92].

In Chapter 5 we touched upon modeling text databases by context-free grammars. A *grammatical* system models the structure of data by grammars. The conceptual view of a document in a grammatical system is a parse tree. A text database system that utilizes and maintains the structure of text documents is called a *structured text database system*.

The *p-string algebra* of Gonnet and Tompa [GT87] has been an influential query language proposal for structured text databases. The p-string algebra is a procedural language for the manipulation of parse trees. Gyssens, Paradaens, and Van Gucht have further developed both an algebraic and a logic-oriented language for querying grammatical databases [GPG89]. Mannila's and Räihä's work [MR90] on query languages for the p-string data model has been an important source of inspiration for this thesis. Ordered tree inclusion appears as a primitive in the calculus of [MR90].

The implementation of the *Lexical Query Language (LQL)* [Byr89] is a system for querying static structured text documents. LQL combines modeling ideas from grammatical text databases and user-interface ideas from the Query-by-Example database query language [Zlo77].

Helsinki Structured Text Database System (HST) [KLMN90] is a prototype text database system that is based on modeling text documents by context-free grammars. HST enables the manipulation of structured documents on two levels of abstraction. On the higher level, the database designer can define *views* for producing different textual appearances out of the document instances in the database [Nik90]. Views are annotated grammars, which resemble syntax directed translation schemas [AU72]. Ad-hoc queries can be given in HST by writing small programs in the *P-string Query Language (PQL)* [KLM+91]. PQL is a procedural derivative of the p-string algebra.

Burkowski has proposed a *containment model* that lies between the text-based and grammatical approaches [Bur92]. Containment model views a text database as a collection of *concordance lists*. A concordance list is a list of contiguous non-overlapping text segments like occurrences of a word or chapters. The containment model provides an algebra for selecting sublists of concordance lists based on relative containment criteria of their text segments. The abstraction level of the containment model is rather close to the concepts of the PAT language. On the other hand, Burkowski describes a text database system based on the containment model that utilizes the structure of the documents and supports interfaces with a hierarchic view of the text.

Hypertext is a way to organize information in a collection of *nodes* connected by associative *links* [SW88]. While grammatical systems view documents as parse trees, hypertext systems view them as graphs. The typical way to search information from a hypertext database is to use a graphical user interface to navigate through the database by following the links.

We take the grammatical approach to a text database and view it as a collection of parse trees over some grammar. Each document is presented in the database as a labeled and ordered tree. The internal nodes of the trees are labeled by nonterminals of the grammar, and they correspond to meaningful document components like titles, headers, chapters, and paragraphs. The leaves of the database trees are text strings contained in the document components that correspond to their parent nodes. For example, the leaf below a node u labeled by *title* would be the text content of the *title*-structure represented by node u. This kind of databases can originate from explicit representations of document structures conforming for example to standards like ODA and SGML [Bro89]. In our model a search is used for locating subtrees of the database. This means that a search can either return complete documents or their parts.

7.2 Querying with inclusion patterns

In this section we show how tree inclusion with logical variables can be applied to querying structured text data. We use ordered tree inclusion with logical variables for several reasons. First, ordered inclusion allows expressing both containment and ordering conditions that are essential in structured text documents. It is characteristic for example of books that their chapters consist of sections consisting of subsections, and so forth, and that the ordering of the chapters and sections is essential. Second, inclusion patterns allow irrelevant structures to be ignored in the queries. Third, we believe that queries based on ordered tree inclusion with logical variables can be implemented efficiently. We discuss the evaluation of inclusion queries in Section 7.3.

The tree inclusion problems with logical variables, as were defined in Chapter 6, are a viable query formalism as such. Variables in the patterns can be used to retrieve parts of the target. We extend now the syntax slightly. A tree pattern can state conditions on the values of the variables that occur *in* the pattern, that is, we can restrict the retrieved parts of data by their *context*. It is often useful also to be able to access parts of the database by their *content*. For this reason we allow patterns of the form

$$x : p \,, \tag{7.1}$$

where x is a variable and p is a term. We say that expression (7.1) *attaches* variable x to the root of $\mathit{Tree}(p)$. This gives a way to attach variables also to the constant nodes of patterns. We say that a variable x *occurs* at pattern node v if v is labeled by x or x is attached to node v. To take this extension into account in the definition of pattern embeddings, we require that all pattern nodes where a variable x occurs have to be mapped to roots of identical subtrees of the target.

We also allow strings as leaves of the patterns and agree that a string s matches at a target leaf l if s is a substring of l. This is a simple way to treat strings in patterns; for practical purposes one can provide e.g. matching based on regular expressions.

A single pattern can state a single structural condition on the data. We often need to express multiple search conditions. For this reason we define a *query* to be a sequence of patterns, all of which are matched or embedded in the database simultaneously.

Before defining the precise meaning of queries we present some examples of applying inclusion patterns to structured text documents. We use in the examples Prolog-like syntax [CM84, SS86], where lower-case identifiers are constants, upper-case identifiers are variables, patterns constituting the query are represented as terms separated by commas and queries are preceded by "?- ". The examples are borrowed from [Byr89]. They consider querying the *Collins-Robert French Dictionary.* The simplified structure of the dictionary used in the examples can be described by the following grammar.

dictionary	$\rightarrow$	eng-fren fren-eng
eng-fren	$\rightarrow$	entry*
fren-eng	$\rightarrow$	entry*
entry	$\rightarrow$	hdw superhom*
superhom	$\rightarrow$	pronunc homograph*
homograph	$\rightarrow$	{ homnum } pos translat

That is, the dictionary consists of the English-French part and the French-English part, in this order. Each part consists of entries formed of a head-word followed by a number of superhomographs; a single written word can have various pronunciations that differ in their meaning; each of them is treated in its own superhomograph. Finally the superhomographs consist of homographs containing possibly a homograph number, the part-of-speech indication (e.g., a verb or a noun), and the translation. The nonterminals for which no productions are shown produce plain text.

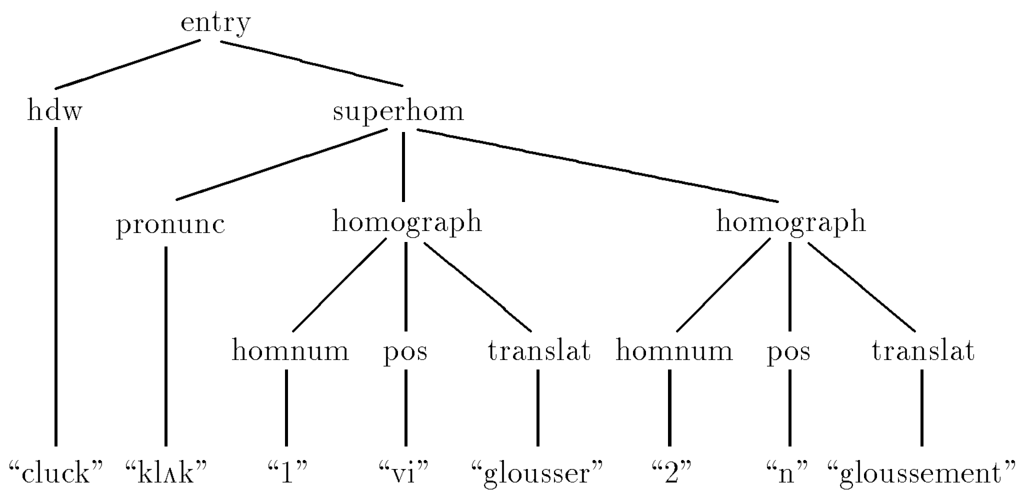


Figure 7.1: The parse tree of the dictionary entry for *cluck*.

For example, the English-French entry for *cluck*, whose representation as a tree is shown in Figure 7.1, has one superhomograph with two homographs. A typical typeset appearance of the entry could be as below.

> **cluck** [klʌk] **1** *vi* glousser. **2** *n* gloussement.

The following query would fetch the whole entry for *cluck*. Note that the query pattern needs to express only those components of the entry that are relevant for fetching it.

> *?- X : entry(hdw("cluck")).*

The following query accesses only the parts-of-speech and translation fields for *cluck*.

> *?- entry(hdw("cluck"), homograph(pos(P), translat(T))).*

Compare the two queries above. The first query describes the wanted-for complete entry X by its partial *content*, whereas the second one describes the wanted-for strings P and T by their partial *context*. Also note how the second query restricts the structures matched by *pos(P)* and *translat(T)* to appear inside the same homograph.

Sometimes we need to express multiple search conditions. For instance, assume that we do not know the relative order of the translations of a word. Then we might give the following query to fetch those English words that can be translated by both *glousser* and *gloussement*. (This feature is a consequence of the decision to restrict to *ordered* inclusion.)

> *?- eng-fren(entry(hdw(X), translat("glousser"))),*
> *eng-fren(entry(hdw(X), translat("gloussement"))).*

Finally, repeating variables allow expressing join-like queries quite easily. For example, the following query shows all English words E that are translations of any French words F that are themselves translations of *capital*.

> *?- eng-fren(entry(hdw("capital"), translat(F))),*
> *fren-eng(entry(hdw(F), translat(E))).*

Next we define the meaning of queries on trees and forests. Let $Q = P_1, \ldots, P_k$ be a query and T a tree or a forest. The nodes of the trees $P_1, \ldots, P_k$ are the *nodes of query* Q. Let f be a function from the nodes of query Q to the nodes of T. If f maps nodes u and v of Q to the roots of identical subtrees of T whenever the same variable occurs at u and v we say that f *respects variables in query* Q. If function f respects variables in Q and the restriction f_i of f to P_i is an ordered pattern embedding of P_i in T for each $i = 1, \ldots, k$ then f is a *query embedding* of Q in T. The substitution consisting of the bindings $x = Term(T[f(v)])$, where x is a variable that occurs at a node v of Q and f is a query embedding of Q in T, is a *solution to query* Q *on* T. The set of solutions to query Q on T is the *answer to query* Q *on* T.

Let T be a tree or a forest. Denote by $st(T)$ the set of all subtrees of T, and by $st^{\circ}(T)$ the set of all proper subtrees of T.

Example 7.1 Let T be a tree, X a variable and a a constant symbol. The answer to the query $Q_1 = X$ on tree T is the set of substitutions

$$\{\{X = t\} \mid \exists\, U \in st(T) : t = Term(U))\} \ .$$

That is, query Q_1 in effect retrieves every subtree of tree T.

The answer to the query $Q_2 = X : a$ on tree T is the set of substitutions

$$\{\{X = t\} \mid \exists\, U \in st(T) : label(root(U)) = a \text{ and } t = Term(U))\} \ .$$

That is, query Q_2 retrieves every subtree of T rooted by an a-node.

The answer to the query $Q_3 = a(X)$ on tree T is the set of substitutions

$$\begin{aligned}\{\{X = t\} \mid \exists\, U \in st(T)\, \exists\, V \in st^{\circ}(U) \quad &: \\ label(root(U)) = a \text{ and } t \quad &= \quad Term(V))\} \ .\end{aligned}$$

That is, query Q_3 retrieves all proper subtrees of the subtrees of T whose root is labeled by a. □

We show in the next example how tree inclusion with logical variables can be used to access arbitrarily deep structures easily. Assume that the database contains information about the structure of various engines. Each engine has a name and consists of named components. The components are either basic components or compound components that consist recursively of other components. The problem is to find out the names for the basic components of a given engine. Problems like this are often used as examples for the need of *recursive database queries* [CGT90]. The database in question can be modeled as a structured text database by the following grammar.

engines	→	engine*
engine	→	engine-name component*
component	→	compound-comp
component	→	basic-comp
compound-comp	→	name component*
basic-comp	→	name

In a database modeled by the above schema the following inclusion query would compute the names for all basic components of vacuum cleaner.

?- engine(engine-name("vacuum cleaner"), basic-comp(X)).

As a query primitive, tree inclusion is tolerant to certain variations in the structure of the data. Let T be a database represented as a tree or a forest, and let M be a modification of T. Denote the modified version of the database by $M(T)$. We say that a query Q on database T is *robust* with respect to modification M if applying query Q on $M(T)$ gives the same result as modifying by M the answer of Q on T. We omit the precise definitions. Still, we note that the inclusion primitive owns some robustness. Let $P = (V, E, root(P))$ and T be trees. If there is an ordered embedding f of P in T, there is also an ordered embedding of P in T', when tree T' is obtained from T by deleting any nodes not belonging to $f(V)$, or by inserting any new nodes in T. Let now T be a structured text database and Q a *simple inclusion query* expressed as $x : p$, where x is a variable and p is a variable-free term. Then the above considerations show that modifying database T by inserting in T or deleting from T any nonterminals or strings that do not appear in P does not essentially change the answer to query Q; the same trees are retrieved, although possibly modified.

A serious query language would need many features in addition to those represented in this chapter. An ability to express negation like "Accept here the structures that are *not* matched by this pattern" and disjunctive conditions like "Accept here the structures matched by this *or* that" would certainly be useful in practice. Another extension that might be useful is the introduction of Prolog-like rules. A rule that consists of a head and a query-like body would generate new structures as instantiations of its head by all the solutions to its body. The rules could also be allowed to be recursive and be given a fixed-point semantics, but the need for such a powerful extension is not clear yet.

7.3 Evaluation of queries

The evaluation of queries poses interesting algorithmic questions that have already been discussed in the previous chapters. We saw in Section 6.2 that answers to inclusion queries may consist of exponentially many solutions. We showed that even the decision version of ordered tree inclusion with logical variables is NP-complete. Here we give some hints for solving queries consisting of a sequence of inclusion patterns. The treatment is rather sketchy.

Let $Q = P_1, \ldots, P_k$ be a query and T a database represented as a tree or a forest. We can solve the query by generating one solution at a time by a strategy that resembles the backtracking-based control of Prolog [War77, Bru82, MW88]. That is, we first find a solution for the first pattern and instantiate the variables in the remaining patterns using it. Then the evaluation continues similarly with the remaining patterns. When no solution to some pattern instance can be found, or when the entire query has been solved, the control backtracks to the most recently solved pattern and tries to seek a new solution for it.

There are two basic kinds of optimizations found in relational query processors, namely algebraic manipulation and cost-estimation strategies [Ull89]. We discussed in Section 5.3 some possibilities for transforming the inclusion queries to more restricted forms by using the grammar of the database. Such methods can be considered to correspond to the optimization of queries through algebraic manipulation. Query optimization algorithms usually take *indices* into account because their usage can render the evaluation of database queries essentially more efficient. Next we discuss some ideas about using indices in the evaluation of inclusion queries.

Lexicographical indices like inverted files [Knu73] and PAT trees [GBYS91] allow efficiently locating occurrences of words in text files. An *inverted file* is a sorted list of keywords together with links to the documents that contain them. The indices in most commercial library systems are inverted files. A PAT *tree* for a text is a Patricia tree [Knu73] built from all suffixes of the text. A PAT tree allows locating the occurrences of any substring s of the text in time that is proportional to the length of s and to the number of the occurrences. (Counting the *number* of those occurrences can be done even in time that depends only on the length of s.) PAT trees are used in the implementation of the PAT$^{\text{TM}}$ text search system.

We saw in Section 5.2 how simple inclusion queries can be solved in $O(n)$ time on a database that is modeled by a nonperiodic grammar. We also claimed that most text databases seem to be natural to model using nonperiodic grammars. However, for large databases no method of answering a query is acceptable if it needs to scan through the entire database. Luckily, most useful query patterns contain string components that restrict the number of possible occurrences drastically. Therefore a reasonable strategy of evaluating such queries is first to locate the occurrences of their string components in the database, and then concentrate on the ancestors of these occurrences in the evaluation of the rest of the pattern. This kind of an approach has been taken in the Maestro project [Mac91], where a prototype retrieval tool for hierarchic text structures has been implemented on top of the full text retrieval system Ful/Text.

Note that the patterns forming a query can be treated in any suitable order. In fact, patterns or pattern instances that do not have any variables in common do not affect each other and can be solved even in parallel. Some orders of matching the patterns may be though much more economical than others. For example, reconsider from Section 7.2 the following query for finding the English counterparts for the French translations of the word "capital".

?- eng-fren(entry(hdw("capital"), translat(F))),
fren-eng(entry(hdw(F), translat(E))).

Now the second pattern

fren-eng(entry(hdw(F), translat(E)))

is likely to have at least one embedding in each entry of the French-English dictionary. Thus solving it first would produce a large number of instantiations of the first pattern to be matched against the database. On the other hand, solving the first pattern first is much better: It has probably a small number of embeddings to the dictionary, and produces therefore a small number of instantiations of the other pattern. For the evaluation of a query consisting of many patterns it can be useful to try to find an ordering that is likely to minimize the number of alternatives to be considered.

We outline a simple heuristic for ordering patterns in the query. The heuristic is inspired by the query planning algorithm of [War81]. The ordering of patterns is based on the frequencies of string occurrences in the database, which can be easily found in a suitable index. Assume that a PAT tree has been built from the strings in the leaves of a database tree (or forest) T. Then the number of occurrences of any string s in the leaves of the database can be extracted from the index in time $\Theta(|s|)$. We call this number the *cost of string* s in T. The *cost of a label* a in T is defined to be the number of times that label a appears in database T. Let $P = a(\ldots)$ be a pattern. Define the *cost of pattern* P in T be the minimum of the cost of label a and the costs of any string appearing in P. The meaning of the cost of a pattern P in T is that it is an estimate for the number of solutions to P on T. The above cost function is a crude one; more complicated cost functions could lead to more accurate estimates.

Now the strategy for optimizing the evaluation order of patterns is simple: select a pattern with the smallest cost and find a solution to the pattern. Then the same process is started anew for the remaining patterns instantiated by that solution.

On the basis of the above considerations we believe that evaluating inclusion queries can be done reasonably efficiently in practice.

Chapter 8

Conclusions

We have studied a collection of tree matching problems called tree inclusion problems. The general motivation for this research comes from the importance of tree structures and from the intuitiveness of pattern matching as an access operation. A specific motivation for these problems is the research of structured text databases and their query languages.

In Chapter 3 we introduced the tree inclusion problems as variations of the general problem of locating instances of a given pattern tree among the subtrees of a given target tree. The common feature of various tree inclusion problems is that the instances contain distinct nodes that correspond to the pattern nodes and have similar hierarchical relationships. The classes of unordered and ordered inclusion problems result, correspondingly, if we either ignore the left-to-right ordering of pattern nodes or require that the pattern instances resemble the pattern also with respect to this ordering.

We offered a unified treatment for the problems by presenting algorithms for most of them in Chapter 4. Most of the algorithms were derived from a single dynamic programming schema. New algorithms were presented for the unordered and ordered tree inclusion problems. We derived upper complexity bounds for the tree inclusion problems from their algorithms. The lower bounds of most of the problems are open. We showed that unordered tree inclusion is an NP-complete problem. The relationship between tree matching and string pattern matching with don't care symbols, which has been noted by Kosaraju, implies that the ordered path inclusion and region inclusion problems cannot be solved asymptotically faster than string pattern matching with don't care symbols. No linear time algorithm is known for string pattern matching with don't care symbols.

The $|P| \times |T|$ space required by the dynamic programming algorithms can be prohibitive in applications with large targets T. In the case of ordered tree inclusion we solved this problem by presenting an algorithm whose space complexity is $O(|P|\ depth(T))$. The result is significant for applications where the patterns can be expected to be small or the targets can be expected to be shallow.

We presented a simple general condition for tree matching problems to be solvable in linear time. In Chapter 5 we considered G-grammatical tree matching problems where the targets are parse trees over some grammar G. The general condition was applied to showing that G-grammatical tree pattern matching and ordered tree inclusion can be solved in linear time with nonperiodic grammars G. Such grammars seem sufficient for modeling many practical text databases. We also outlined how a grammar G can be used to preprocess patterns in G-grammatical matching problems.

In Chapter 6 we extended tree inclusion problems by logical variables. The variables can be used to extract substructures of the pattern instances and to express equality constraints on them. We gave many NP-hardness results for the tree inclusion problems with logical variables and sketched efficient algorithms for the polynomial variants.

The intended application of the inclusion problems is in structured text databases. In Chapter 7 we showed how tree inclusion can be used for querying structured text databases and gave examples of using inclusion queries. Tree inclusion was shown to be a powerful and robust query primitive. We also discussed how inclusion queries can be evaluated efficiently in practice by utilizing lexicographical indices.

We plan to develop a complete query language for structured text databases based on these notions.

Bibliography

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Ang80] D. Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21:46–62, 1980.

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, vol I and II*. Prentice-Hall, 1972.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[BDK+91] G. Burosch, J. Demetrovics, G. O. H. Katona, D. J. Kleitman, and A. A. Sapozhenko. On the number of databases and closure operations. *Theoretical Computer Science*, 78:377–381, 1991.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1&2):203–216, February/April 1987.

[Bol86] B. Bollobás. *Combinatorics*. Cambridge University Press, 1986.

[BR84] F. Bancilhon and P. Richard. Managing texts and facts in a mixed data base environment. In G. Gardarin and E. Gelenbe, editors, *New Applications of Data Bases*, pages 87–107. Academic Press, 1984.

[Bro89] H. Brown. Standards for structured documents. *The Computer Journal*, 32(6):505–514, December 1989.

[Bru82] M. Bruynooghe. The memory management of Prolog implementations. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 83–98. Academic Press, 1982.

[Bur92] F. J. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In N. Belkin, P. Ingwersen, and A. M. Pejtersen, editors, *Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 112–125. ACM Press, June 1992.

[Byr89] R. J. Byrd. LQL user notes: An informal guide to the lexical query language. Technical Report RC 14853 8/17/89, IBM T.J. Watson Research Center, August 1989.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[Chu87] M. J. Chung. $O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees. *Journal of Algorithms*, 8:106–112, 1987.

[CIV86] G. Coray, R. Ingold, and C. Vanoirbeek. Formatting structured documents: Batch versus interactive. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 154–170. Cambridge University Press, 1986.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[DGM90] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. In *Proc. of the Symposium on Foundations of Computer Science (FOCS'90)*, pages 145–150, 1990.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations of the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[Dub90] P. Dublish. Some comments on the subtree isomorphism problem for ordered trees. *Information Processing Letters*, 36:273–275, 1990.

[FP74] M. J. Fischer and M. S. Paterson. String-matching and other products. In *Complexity of Computation*, pages 113–125. SIAM-AMS, 1974.

[FQA88] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing*, 1(1):19–44, 1988.

[FSS90] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Automata, Languages and Programming*, pages 220–234. Springer-Verlag, 1990.

[GBYS91] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Report OED-91-01, UW Centre for the New Oxford English Dictionary and Text Research, 1991.

[GJ78] M. R. Garey and D. S. Johnson. "Strong" NP-completeness results: Motivation, examples and implications. *Journal of the ACM*, 25(3):499–508, July 1978.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[GPG89] M. Gyssens, J. Paradaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. Report, University of Antwerp, Dept. of Math. and Comp. Science, 1989.

[Gro91] R. Grossi. A note on the subtree isomorphism for ordered trees and related problems. *Information Processing Letters*, 39:81–84, 1991.

[GT87] G. H. Gonnet and F. Wm. Tompa. Mind your grammar - a new approach to text databases. In *Proc. of the Conference on Very Large Data Bases (VLDB'87)*, pages 339–346, 1987.

[HK73] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973.

[HO82] C. M. Hoffman and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Joh87] D. S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 8:285–303, 1987.

[KLM+91] P. Kilpeläinen, G. Lindén, H. Mannila, E. Nikunen, and K.-J. Räihä. The data model and query language of the Helsinki structured text database system (HST). Technical report, University of Helsinki, Department of Computer Science, November 1991.

[KLMN90] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured document database system. In R. Furuta, editor, *EP90 – Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, The Cambridge Series on Electronic Publishing. Cambridge University Press, 1990.

[KM91a] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. Report A-1991-4, University of Helsinki, Dept. of Comp. Science, August 1991. To appear in *SIAM Journal on Computing.*

[KM91b] P. Kilpeläinen and H. Mannila. The tree inclusion problem. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91, Proc. of the International Joint Conference on the Theory and Practice of Software Development, Vol. 1: Colloqium on Trees in Algebra and Programming (CAAP'91)*, pages 202–214. Springer-Verlag, 1991.

[KM92] P. Kilpeläinen and H. Mannila. Grammatical tree matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 162–174. Springer-Verlag, 1992.

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

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[Kos89] S. R. Kosaraju. Efficient tree pattern matching. In *Proc. of the Symposium on Foundations of Computer Science (FOCS'89)*, pages 178–183, 1989.

[Mac91] I. A. Macleod. A query language for retrieving information from hierarchic text structures. *The Computer Journal*, 34(3):254–264, 1991.

[Mat68] D. W. Matula. An algorithm for subtree identification. *SIAM Rev.*, 10:273–274, 1968. Abstract.

[MR90] H. Mannila and K.-J. Räihä. On query languages for the p-string data model. In H. Kangassalo, S. Ohsuga, and H. Jaakkola, editors, *Information Modelling and Knowledge Bases*, pages 469–482. IOS Press, 1990.

[MW88] D. Maier and D. S. Warren. *Computing with Logic - Logic Programming with Prolog.* The Benjamin/Cummings Publishing Company, Inc., 1988.

[Nik90] E. Nikunen. Views in structured text databases. Phil.lic. thesis, University of Helsinki, Department of Computer Science, December 1990.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[QV86] V. Quint and I. Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 200–213. Cambridge University Press, 1986.

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

[RR88] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. In T. Lepistö and A. Salomaa, editors, *Automata, Languages and Programming*, pages 473–488. Springer-Verlag, 1988.

[RR92] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295–316, April 1992.

[RS86] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[Sel77] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[SF83] J.-M. Steyaert and P. Flajolet. Patterns and pattern-matching in trees: An analysis. *Information and Control*, 58:19–58, 1983.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[Spe28] E. Sperner. Ein Satz über Untermengen einer endlichen Menge. *Math. Z.*, 27:544–548, 1928.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[SSL+83] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. Document processing in a relational database system. *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.

[ST92] A. Salminen and F. Wm. Tompa. PAT expressions: an algebra for text search. Technical Report OED–92–02, UW Centre for the New Oxford English Dictionary and Text Research, 1992.

[SW88] J. B. Smith and S. F. Weiss. An overview of hypertext. *Communications of the ACM*, 31(7):816–819, July 1988.

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

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[Tom92] F. Wm. Tompa. An overview of Waterloo's database software for the OED. In T. R. Wooldridge, editor, *Historical Dictionary Databases*, pages 125–143. University of Toronto, 1992. Centre for Computing in the Humanities Working Papers 2.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The New Technologies. Computer Science Press, 1989.

[Ver92] R. M. Verma. Strings, trees, and patterns. *Information Processing Letters*, 41:157–161, March 1992.

[vL90] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10. Elsevier Science Publishers B.V., 1990.

[War77] D. H. D. Warren. Implementing Prolog - compiling predicate logic programs. Volumes 1 and 2. D.A.I. research reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[War81] D. H. D. Warren. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 272–281. Computer Society Press, 1981.

[WJZS91] J. T.-L. Wang, K. Jeong, K. Zhang, and D. Shasha. Reference manual for ATBE – a tool for approximate tree pattern matching. Technical Report 551, New York University, Dept. of Comp. Science, Courant Institute of Mathematical Sciences, March 1991.

[Zlo77] M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, December 1989.

[ZSW91] K. Zhang, D. Shasha, and J. T.-L. Wang. Approximate tree matching in the presence of variable length don't cares. Submitted for publication, July 1991.

[ZSW92] K. Zhang, D. Shasha, and J. T.-L. Wang. Fast serial and parallel algorithms for approximate tree matching with VLDC's. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 151–161. Springer-Verlag, 1992.