Determining the Consistency of Partial Tree Descriptions

Manuel Bodirsky[1] and Martin Kutz[2]

[1] Humboldt-Universität zu Berlin, Germany
bodirsky@informatik.hu-berlin.de
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
mkutz@mpi-inf.mpg.de

Abstract. We present an efficient algorithm that checks the consistency of partial descriptions of ordered trees. The constraint language of these descriptions was introduced by Cornell in computational linguistics; the constraints specify for pairs of nodes sets of admissible relative positions in an ordered tree. Cornell asked for an algorithm to find a tree structure satisfying these constraints. This computational problem generalizes the *common-supertree problem* studied in phylogenetic analysis. We present the first polynomial time algorithm for Cornell's problem, which runs in time $O(mn)$, where m is the number of constraints and n the number of variables in the constraint.
keywords: tree descriptions, constraint satisfaction, graph algorithms

1 Introduction

Tree description languages became an important tool in computational linguistics over the last twenty years. Grammar formalisms have been proposed that derive logical descriptions of trees representing the syntax of a string [8, 13, 15]. Membership in a language is then equivalent to the satisfiability of the corresponding logical formula. In computational semantics, the paradigm of *underspecification* aims at manipulating the partial description of tree-structured semantic representations of a sentence rather than at manipulating the representations themselves [9, 14]. One of the key issues in both constraint based grammar and constraint based semantic formalisms is to collect partial descriptions of trees and to *solve* them, i.e., to find a tree structure that satisfies all constraints.

Cornell [6] introduced a simple but powerful tree description language, which contains constraints for *dominance*, *precedence*, and *equality* between nodes, and disjunctive combinations of these (a formal definition is given is Section 2). Cornell also gave a saturation algorithm based on local propagations, which turned out to be incomplete. For an example of a tree description that shows this see Section 3.4 in [3].

In this article we present an efficient algorithm that tests satisfiability of a tree description from Cornell's tree description language and directly constructs a solution to the problem instance. A predecessor of this algorithm, which applies to a restricted language, was presented in [4]. The present algorithm, which solves the general problem of Cornell's full tree description language, runs in time $O(nm)$, where n is the number of variables and m the number of constraints in the input. The performance is achieved by a recursive strategy that works directly on the constraint graph, and avoids local consistency techniques ala [7, 10] that are frequently used in constraint satisfaction.

Another field in computer science where we have to deal with partial information about trees is phylogenetic analysis in computational biology. An *evolutionary tree* for a set of species is a rooted tree, where the leaves are bijectively labeled with the species from the set. Constructing evolutionary trees from biological data is a difficult problem for a variety of reasons (see [11]). Many approaches assume that the evolutionary tree is built from a set of taxa based on the comparison of a single protein or a single position in aligned protein sequences, but very often the resulting tree will be different depending on which particular protein or position is used. Now several trees, each from a different protein or position, must be built and be shown to be "generally consistent" before the implied evolutionary history is considered reliable.

This is the motivation to study the following problem, called the *common-supertree problem* [11]. Given is a set S of trees with common leaf set L. The computational talk is to decide whether there is a tree T on the leaf set L such that every tree in S is a *refinement* of T, i.e., can be obtained by a series of contractions of edges from T. In Section 3, we reduce this problem to a tree-description problem in a fragment of Cornell's language and therefore the algorithm presented here also yields a new algorithm for the common-supertree problem.

2 Tree Descriptions

The trees considered here are always rooted and we consider the edges as directed, pointing away from the root. The *height* of a tree is the length of a longest directed path. By an *ordered tree* we mean a rooted tree with a linear order on the children of each vertex and we use the terms *left* and *right* to compare them.

We follow the notation of [2]. Usually the vertices of a tree are denoted by u, v, w. The expression $u \lhd v$ denotes that u is the father of v and $u \lhd^* v$ (and $v \,{}^*\!\rhd u$) means that u *dominates* v, i.e., u is an ancestor of v in the tree (including $u = v$). We write $u \lhd^+ v$ (and $v \,{}^+\!\rhd u$) if $u \lhd^* v$ and $u \neq v$. If for two vertices u and v neither $u \lhd^* v$ nor $v \lhd^* u$, we say that u and v are *disjoint*, in symbols $u \perp v$. In this situation we distinguish two cases: either u *precedes* or *succeeds* v. A vertex u *precedes* a vertex v (and v *succeeds* u), in symbols $u \prec^+ v$ (or $v \,{}^+\!\succ u$), if there is a common ancestor of u and v in the tree that has two children w_1 and w_2 with $w_1 \lhd^* u$ and $w_2 \lhd^* v$ and such that u is to the left of v. We write $u \prec^* v$ if either $u \prec^+ v$ or $u = v$. Altogether, in an ordered tree, for every pair u, v of vertices exactly one of the following relations holds:

$$u \lhd^+ v, \quad u \,{}^+\!\rhd v, \quad u \prec^+ v, \quad u \,{}^+\!\succ v, \quad u = v.$$

It is important to note that the union of the relations $\lhd^+$ and $\prec^+$ forms a strict linear order on the set of all vertices of an ordered tree, which is easily seen to be the "pre-order" that results from a (recursive) tree traversal that lists each node before its descendants.

We now define partial tree descriptions, due to Cornell [6], that allow to partially describe the structure of an ordered tree using arbitrary disjunctions of these five cases. To distinguish clearly between equality in this language and the common usage of the symbol '=', we denote the former relation by '$\equiv$'.

Definition 1. *A* partial tree description (V, C) *consists of a set of variables* V *and a set* C *of binary constraints of the form* $x\mathcal{R}y$*, where* $x, y \in V$ *and* $\mathcal{R} \subseteq \{\lhd^+, {}^+\!\rhd, \prec^+, {}^+\!\succ, \equiv\}$*. A constraint* $x\mathcal{R}y$ *is* satisfied *by a pair* (T, α)*, where* T *is an ordered tree and* $\alpha\colon V \to T$ *is a mapping from the variables to the vertices of the tree, if* $\alpha(x)R\alpha(y)$ *holds in the tree for* some *relation* $R \in \mathcal{R}$*. A pair* (T, α) *that satisfies all the constraints in* C *is called a* solution *for* C*.*

Note that the mapping α in the above definition is not required to be surjective. In particular, the induced graph on the image of α need not be connected but can be a forest. An instance that has a solution is called *satisfiable* or *consistent*. Figure 1 shows an unsatisfiable instance. Solid arcs stand for the constraint $\{\lhd^+\}$ and dashed arcs for the constraint $\{\prec^+, {}^+\!\succ\}$.

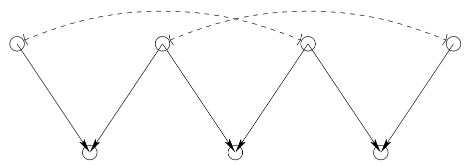

Fig. 1. A constraint graph with no solution.

If the nodes in a tree are the intervals over the real line, we can translate tree descriptions into a fragment of Allen's interval algebra [1]. For this, the set of intervals of the tree is required to be *laminar*, i.e., for any pair of intervals that is not disjoint, one interval must completely contain the other. In other words, intervals must not overlap. Dominance between nodes then corresponds to containment of intervals and disjointness of nodes corresponds to disjointness of intervals. Allen's full interval constraint logic is NP-complete in its unrestricted form, but there are tractable fragments of this logic [5, 12]. The constraint language with dominance, disjointness, and non-overlap constraints does not belong to those tractable fragments. Thus we cannot use the known algorithms for interval constraints to efficiently solve our problem.

3 An Algorithm for a Restricted Signature

We first illustrate the idea of our algorithm for a small fragment of the tree description language. However, as we shall see below, this fragment is already expressive enough to cover the common-supertree problem from phylogenetic analysis mentioned in the introduction. In this reduced constraint language we only allow constraints of the types

$$x\ \{\lhd^+\}\ y \quad \text{and} \quad x\ \{\prec^+, {}^+\succ\}\ y.$$

The first constraint is called *(strict) dominance* and the second *disjointness*, and we use the shorthands $x \lhd^+ y$ and $x \perp y$, respectively, for these two types of literals. Thus, such a set of constraints can be viewed as a graph $(V; \lhd^+, \perp)$ with two types of edges, namely directed edges $\lhd^+$ and undirected edges $\perp$. Observe that the binary relations $\lhd^+$ and $\perp$ are now defined as well on the vertices of a tree as on the nodes of an instance. The reference, however, should always be clear. In pictures we draw this *constraint graph* with two different types of arcs. For a dominance edge $x \lhd^+ y$ we draw a directed arc from x to y, for a disjointness edge $x \perp y$ we draw a dashed line without direction. Figure 1 on the preceding page, for instance, shows such a constraint graph. We emphasize that the two constraints do not allow for equality, i.e., $x \lhd^+ y$ means that in any solution, x must lie strictly above y.

The basic idea behind our algorithm for solving a constraint graph $(V; \lhd^+, \perp)$ is to pick a node x of V as the root of our solution, then to decompose the remainder of V and to recurse on these components, which will become the subtrees below x. In this view, the following definition comes naturally.

Definition 2. *A node x in a set of constraints C is* free *if C has a solution (T, α) in which $\alpha(x)$ is the root of T.*

By definition, a constraint with a free node is satisfiable. We shall soon see that under a simple connectivity condition, the converse is also true: a "connected" satisfiable constraint must have a free node, i.e., a node that dominates all others.

The crucial point here is, of course, in what sense the constraint graph, which contains directed and undirected edges, should be connected. Let us briefly recall some conventions. An *undirected* path in a directed graph may use arcs in any direction, ignoring their orientation. A digraph $D = (V; E)$ is *strongly (weakly) connected* if there is a directed (an undirected) path from a to b for any two vertices $a, b \in V$. The subgraph of D induced by a subset S of its vertices is denoted by $D[S]$. A *strongly (weakly) connected component of D* is a maximal strongly (weakly) connected induced subgraph U of D.

Now, we say that a constraint graph $(V; \lhd^+, \perp)$ is *dominance connected* if the digraph $(V; \lhd^+)$ is weakly connected. This notion of connectivity is exactly the right thing for our desired characterization of free nodes. The following lemma marks the first step.

Lemma 1. *Let C be a set of constraints with a dominance-connected constraint graph $G = (V; \triangleleft^+, \perp)$ and let y and y' be variables in C. In every solution (T, α) of C there exists a variable $x \in V$ such that $\alpha(x) \triangleleft^* \alpha(y)$ and $\alpha(x) \triangleleft^* \alpha(y')$ in T.*

Proof. Since the vertices y and y' are weakly connected in $(V; \triangleleft^+)$ there exists a chain of nodes $(y_0, y_1, \ldots, y_r)$ that starts at $y = y_1$, ends at $y' = y_r$, and is linked by edges $(y_i, y_{i+1}) \in \triangleleft^+ \cup {}^+\triangleright$. We prove by induction on r that for every solution (T, α) of C there exists an index $j \in \{0, \ldots, r\}$ with $\alpha(y_j) \triangleleft^+ \alpha(y_0)$ and $\alpha(y_j) \triangleleft^+ \alpha(y_r)$ in T. If $r = 0$ or $r = 1$ then we can choose x to be either $\alpha(y_0)$ or $\alpha(y_r)$. Otherwise, we can apply the induction hypothesis to the chain $(y_1, \ldots, y_{r-1})$. Thus, there exists a j, $0 \leq j \leq r-1$, such that $\alpha(y_j)$ is a common ancestor of $\alpha(y_1)$ and $\alpha(y_{r-1})$ in T. If $\alpha(y_{r-1}) \triangleleft^+ \alpha(y_r)$ then $\alpha(y_j)$ is also a common ancestor of $\alpha(y_1)$ and $\alpha(y_r)$, so we can choose $x = y_j$. Otherwise, $\alpha(y_r) \triangleleft^+ \alpha(y_{r-1})$ in T. Thus, both $\alpha(y_0)$ and $\alpha(y_r)$ are ancestors of y_{r-1} in T. Since T is a tree, it follows that $\alpha(y_r) \triangleleft^* \alpha(y_j)$ or $\alpha(y_j) \triangleleft^* \alpha(y_r)$ in T. In the first case, we choose $x = y_r$, and in the second one $x = y_j$. □

As promised, we now show that for dominance connected constraints solvability implies the existence of a free node.

Proposition 1. *A satisfiable set of constraints C with a dominance-connected constraint graph has a free node.*

Proof. Assume there is a solution (T, α) for C and consider the vertices v that are topmost in T with respect to $\triangleleft^+$ such that $v = \alpha(x)$ for some node x of C. If there is only one such vertex v, then the subtree rooted at v together with α is also a solution of C. But since there is a node x in C such that $v = \alpha(x)$, this contradicts the assumption that x is not free. If there are two distinct topmost nodes v, v', then v and v' are disjoint in T. Since the constraint graph is dominance-connected, we can apply Lemma 1, and obtain a contradiction to the assumption that v and v' lie topmost in T. □

For the algorithm we need a concise graph-theoretic characterization of free nodes that can be checked efficiently.

Proposition 2. *Let $G = (V; \triangleleft^+, \perp)$ be the constraint graph of a satisfiable set of constraints C. Then a node x of C is free if and only if*

(C1) *there is no arc $y \triangleleft^+ x$ in G and*
(C2) *there is no edge $x \perp y$ in G.*

Proof. If there is another y such that $y \triangleleft^+ x$, then the vertex x cannot be topmost in any solution of C, and thus can not be free. If the vertex x is involved in a disjointness constraint it can also not be free, since the root of a tree is not disjoint to the other nodes in the tree.

Conversely, assume that the node x of a satisfiable set of constraints satisfies (C1) and (C2). The dominance-connected components of $G[V - \{x\}]$ are also satisfiable and must thus by Proposition 2 have free nodes. Then we have the following solution for C: introduce a tree node x and let the solutions of the dominance-connected components become the subtrees of this node. The disjointness constraints between the different dominance-connected components are thus satisfied by construction and it is clear from (C1) and (C2) that all the constraints on x are also satisfied. □

Solve$\big(G = (V; \lhd^+, \perp)\big)$
// receives a dominance-connected constraint graph G
// and constructs a tree T with mapping $\alpha \colon V \to T$

```
pick a node x of G satisfying (C1) and (C2);
if no such free node exists then return "problem has no solution";
create a new tree node r and let α(x) := r;
compute the dominance-connected components C_1, ..., C_k of G − {x};
for i = 1 to k do
    call Solve(G[C_i]) and make the returned root a new child of r;
od
return root r;
```

Fig. 2. The function Solve for constraints of the type $\lhd^+$ and $\perp$.

Figure 2 shows the algorithm for the restricted constraint language with the two relations $\lhd^+, \perp$. For a dominance-connected constraint graph, the function Solve first selects a free node x as root and then links the the recursively computed solutions to the dominance-connected components C_i of the remainder $G - \{x\}$ below the α-image r of x in the correct order.

Theorem 1. *There is an algorithm that decides satisfiability of a given tree description C in the restricted constraint language in $O(nm)$ time, where n is the number of nodes and m the number of constraints in C.*

Proof. We use the algorithm shown in Figure 2. For an instance that is not dominance-connected, we can first introduce a dummy node z together with a $\lhd^+$-arc from z to each other node to make the graph connected. The node z then becomes the root of the output, and since α is not required to be onto the solution will be valid also for the original graph without z.

If the algorithm detects a weakly connected component without a free node, we know by Proposition 1 that the constraints do not have a solution. Otherwise Proposition 2 guarantees that we can proceed and make the node x satisfying (C1) and (C2) the root of our solution. The running time is dominated by the repeated computations of the connected components of constraint graphs. We can use depth first search to compute the connected components in time $O(n + m)$, and this will be done at most n times. We can assume that n is smaller than

$m-1$, otherwise the first call of the algorithm divides the problem into instances where the assumption holds. Hence the algorithm runs in time $O(nm)$. $\square$

We finally return to the common-supertree problem from the introduction. It can be easily formulated with tree description in the restricted language of $\lhd^+$ and $\perp$. Let S be a set of trees over a common leaf set L. The variables of the tree description are the vertices of the trees in S, where the inner nodes of all the trees become different variables but the leaves are represented by the same set of variables for all trees. Tree-edges translate to dominance constraints. Siblings x, y in a tree from S will be related via a $x \perp y$ constraint. The size of the resulting tree-description is clearly $O(\Delta^2 n)$, where n is the total number of nodes in S and Δ denotes the maximum degree in any tree of S. It has a solution if and only if there is a common supertree for the trees from S.

4 Reduction to Four Basic Constraints

We now turn to the general problem of solving a set of constraints with the full power of Cornell's tree description language. The basic recursive idea for the algorithm of picking free nodes remains, but the new constraint types lead to more complicated details.

In order to get control over the 2^5 potential subsets of the relation set $\mathcal{R} \subseteq \{\lhd^+, {}^+\!\rhd, \prec^+, {}^+\!\succ, \equiv\}$, we first show how to express each of them by a smaller language which contains the following four different constraint types only:

$$\begin{aligned} x\ \{\lhd^+, \equiv\}\ y, \quad x\ \{\prec^+, \equiv\}\ y, \\ x\ \{\lhd^+, {}^+\!\rhd, \prec^+, {}^+\!\succ\}\ y, \quad x\ \{\prec^+, {}^+\!\succ, \equiv\}\ y. \end{aligned} \tag{1}$$

We reduce each of the 32 original sets either directly to an intersection of constraints from (1) or build small gadgets with new dummy variables that can simulate the original constraint.

The constraints $\{{}^+\!\rhd, \equiv\}$ and $\{{}^+\!\succ, \equiv\}$ are simply the first and second constraint of (1) flipped, and the singletons $\{\lhd^+\}$, $\{\prec^+\}$, and $\{\equiv\}$ can be written as intersections of these. The two extremal sets $\{{}^+\!\rhd, \lhd^+, \prec^+, {}^+\!\succ, \equiv\}$ and $\emptyset$ are not needed since the former imposes no restrictions on the tree and the latter is, by definition, unsatisfiable. If we show how to express the constraints

$$\begin{aligned} x\ \{\lhd^+, {}^+\!\rhd, \equiv\}\ y, \quad x\ \{\lhd^+, \prec^+, {}^+\!\succ, \equiv\}\ y, \\ x\ \{\lhd^+, \prec^+, \equiv\}\ y, \quad x\ \{\lhd^+, {}^+\!\succ, \equiv\}\ y \end{aligned} \tag{2}$$

with those in (1) we are done, since the remaining constraints are easily representable as flippings and intersections of constraints from (1) and (2).

For each constraint $x\ \{\lhd^+, {}^+\!\rhd, \equiv\}\ y$, we introduce a new variable z and replace it by the two constraints $x\ \{\lhd^+, \equiv\}\ z$ and $y\ \{\lhd^+, \equiv\}\ z$. By the properties of a tree, these two constraints imply $x\ \{\lhd^+, {}^+\!\rhd, \equiv\}\ y$. Conversely, every solution for the original constraints can be modified to satisfy also the new constraints.

Constraints of the form $x\ \{\triangleleft^+, \prec^+, \equiv\}\ y$ are replaced by the two constraints $x\ \{\triangleleft^+, \equiv\}\ z$ and $z\ \{\prec^+, \equiv\}\ y$, where z is a new variable. Similarly, we can replace $x\ \{\triangleleft^+, {}^+\!\succ, \equiv\}\ y$ by the two constraints $x\ \{\triangleleft^+, \equiv\}\ z$ and $z\ \{{}^+\!\succ, \equiv\}\ y$. Finally, we replace $x\ \{\triangleleft^+, \prec^+, {}^+\!\succ, \equiv\}\ y$ by $x\ \{\triangleleft^+, \equiv\}\ y$ and $z\ \{\prec^+, {}^+\!\succ, \equiv\}\ y$, where z is again a new variable.

Thus we can express all constraints in Cornell's language with the four basic binary relations from (1). Our algorithm works on tree descriptions consisting of such basic constraints only. We therefore introduce special names and notation for these four types:

- $x\ \{\triangleleft^+, \equiv\}\ y$: the *dominance* constraint $x \triangleleft^* y$,
- $x\ \{\prec^+, \equiv\}\ y$: the *precedence* constraint $x \prec^* y$,
- $x\ \{\prec^+, {}^+\!\succ, \equiv\}\ y$: the *disjoint-or-equal* constraint $x \doteq y$,
- $x\ \{\triangleleft^+, {}^+\!\triangleright, \prec^+, {}^+\!\succ\}\ y$: the *inequality* constraint $x \not\equiv y$.

Note that from now on, the notions of *dominance* and *precedence* relation are to be considered non-strict, i.e., they stand for reflexive relations.

5 Constraint Graphs and Freeness

Our algorithm for the general language will be similar to the one we discussed for the restricted setting. Like there, we will recursively construct a solution by selecting root by root and decomposing remainders into components that become subtrees. The first important difference is that now several variables may map to the same tree node. Hence, we shall need a generalized notion of freeness.

Definition 3. *A subset S of nodes in a set of constraints C is called* free *if there is a solution (T, α) such that $C = \alpha^{-1}(r)$, where r is the root of T.*

Like in the restricted setting, we want to show that under a certain connectedness condition free sets must exists and further have to find practical characteristics of free sets to identify them algorithmically. To this end, we adapt the concept of the constraint graph to the changed situation. The simple idea of weak connectedness with respect to dominance edges must be replaced by a more complex definition based on an auxiliary graph that contains dominance and precedence arcs. We define the directed *P-graph* (V, P) on V with arc set

$$P := \{xy \mid C \text{ contains } x \prec^* y \text{ or } x \triangleleft^* y \text{ or } y \triangleleft^* x\}$$

and call a set of constraints C *dominance connected* if its P-graph is strongly connected.

The following proposition is the analog of Proposition 1 for the restricted case.

Proposition 3. *Any satisfiable set of constraints with strongly connected P-graph contains a free set.*

Proof. Suppose (T, α) is a solution of C. We consider the set S of nodes in C that map to the minimum of the linear order $\lhd^+ \cup \prec^+$ in T, i.e., the nodes that map to the leftmost and topmost vertex u in T. If the vertex u dominates all vertices in $\alpha(V)$ then S is actually a free set. We claim that this must always be the case. So assume for contradiction that there is a variable $y \in V$ such that $\alpha(y)$ is not dominated by u. Since the P-graph of C is strongly connected it contains a path $y, x_1, x_2, \ldots, x_k$ from $y \notin S$ to some $x_k \in S$. The path $\alpha(y), \alpha(x_1), \alpha(x_2), \ldots, \alpha(x_k)$ must eventually enter the subtree rooted at u, with $\alpha(x_j)$, say. Then $\alpha(x_{j-1})$ comes before u in $\lhd^+ \cup \prec^+$, contradicting the minimality of u. □

For the analogy to Proposition 2, the graph-theoretic characterization of free sets, we define a directed graph (V, D) (called the *D-graph*) that is based on the $\lhd^*$ relation but in addition also takes precedences into account. Precisely, we let

$$D := \{xy \mid C \text{ contains } x \lhd^* y \text{ or } x \doteq y \text{ or } x \prec^* y \text{ or } y \prec^* x\}.$$

We remark that the digraphs (V, D) and (P, D) have an interesting symmetry: D is based on dominance constraints and contains bidirected precedence constraints, while P is the union of the precedence constraints and all bidirected dominance constraints. However, a deeper reason for this similarity is elusive. The purposes of the two digraphs are very different in nature.

The characterization of Proposition 2 now becomes a similar statement about this D-graph, with the old conditions (C1) and (C2) merged into one, (C). On the other hand, inequality now has to be handled explicitly, which gives a second condition, (I), again.

Proposition 4. *Let C be a satisfiable constraint. Then S is a free set of nodes if and only if the following two conditions hold:*

(C) *there is no edge $xy \in D$ such that $x \notin S$ and $y \in S$ and*
(I) *there is no pair $x, y \in S$ such that $x \not\doteq y$.*

Proof. The two conditions are clearly necessary because a free set denotes the root of a tree, which dominates all other vertices, and because a constraint $x \not\doteq y$ explicitly forbids to map x and y to the same tree node.

For the other implication, let S be a set of nodes that satisfies conditions (C) and (I). Fix an arbitrary linear extension of the acyclic structure on the strongly connected components of the P-graph without S. Because C is satisfiable, each of the subgraphs of C induced by these components has a solution. Introduce a new vertex and add these solutions as subtrees according to this linear order. The resulting tree satisfies all constraints in the subgraphs and in S and all dominance, precedence, and inequality constraints between the subgraphs and between the subgraphs and S. □

6 The Algorithm

We now turn the Propositions 4 and 3 on free sets of nodes into an algorithm for Cornell's full language. From a given input with constraints $\lhd^*$, $\prec^*$, $\doteq$, and $\not\doteq$, we first create the D-graph and P-graph and then recursively create a tree by decomposing the vertex set into dominance-connected components and extracting free sets as roots.

Figure 3 shows the algorithm, which consists of two procedures, Solve and Solve_con. Initially, for a given instance C, we call Solve(C), which partitions the variable set into the strongly connected components of P-graph. If the P-graph of C is strongly connected, the procedure Solve_con can be applied to C. The algorithm contains a statement *choose*, that influences *which* free set of nodes is selected for the solution. We discuss the issue of how to find such a free set in the running-time analysis below.

Solve(C):

> Compute the scc's of the P-graph of C,
> and let $C_1, \ldots, C_k$ be a linear extension of their acyclic structure
> **create** a new vertex r with children
> Solve_con($C[C_i]$), for $1 \leq i \leq k$, in this order
> **return** r

Solve_con(C):

> **precondition:** The P-graph of C is strongly connected
> **if** no set of nodes satisfying (C) and (I) exists
> **then return** "problem has no solution"
> **else choose** a set of nodes S satisfying (C) and (I)
> **let** r be the root of the tree Solve($C[V - S]$) and set $\alpha(S)$ to r
> **return** r

Fig. 3. The function Solve for the full tree description language.

Proposition 5. *There is an algorithm that decides satisfiability for a given tree description C in time $O(nm)$, where n is the number of nodes and m the number of constraints in C.*

Proof. We use algorithm Solve of Figure 3. In Solve, the strongly connected components of the precedence graph can be computed in linear time in the input size. We then call the procedure Solve_con on the strongly connected components. Thus, if it returns "problem has no solution," the constraints were really unsatisfiable by Proposition 3 on page 8. Otherwise, Proposition 4 guarantees that we can construct a solution by selecting a free set as root. Because of (C1'), a free set must be the union of strongly connected components of the D-graph into which no D-arcs enter. Any strongly connected component contained in a free set is a free set as well. These strongly connected components can easily be found by depth-first search in time $O(n + m)$.

We will have to perform the strongly connected components on each level of the recursion. Since we take out at least one vertex in each call of Solve_con and since in all calls of Solve_con (except possibly the first call) the overall running time is in $O(nm)$. $\square$

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. R. Backofen, J. Rogers, and K. Vijay-Shanker. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information*, 4:5–39, 1995.
3. M. Bodirsky. Constraint satisfaction with infinite domains. Dissertation, Humboldt-Universität zu Berlin, 2004.
4. M. Bodirsky and M. Kutz. Pure dominance constraints. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS'02), LNCS 2285*, pages 287–298, Antibes - Juan le Pins, 2002.
5. H.-J. Bürckert and B. Nebel. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.
6. T. Cornell. On determining the consistency of partial descriptions of trees. In *32nd Annual Meeting of the Association for Computational Linguistics (ACL'94)*, pages 163–170, 1994.
7. R. Dechter and P. van Beek. Local and global relational consistency. *Journal of Theoretical Computer Science*, 173(1):283–308, 1997.
8. D. Duchier and S. Thater. Parsing with tree descriptions: a constraint-based approach. In *Sixth International Workshop on Natural Language Understanding and Logic Programming (NLULP'99)*, pages 17–32, Dec. 1999.
9. M. Egg, A. Koller, and J. Niehren. The Constraint Language for Lambda Structures. *Journal of Logic, Language, and Information*, 10:457–485, 2001.
10. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery*, 29(1):24–32, 1982.
11. D. Gusfield. *Algorithms on strings, trees, and sequences. Computer Science and Computational Biology.* Cambridge University Press, New York, 1997.
12. P. Jeavons, P. Jonsson, and A. A. Krokhin. Reasoning about temporal relations: The tractable subalgebras of Allen's interval algebra. *Journal of the ACM*, 50(5):591–640, 2003.
13. M. P. Marcus, D. Hindle, and M. M. Fleck. D-theory: Talking about talking about trees. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL'83)*, pages 129–136, 1983.
14. M. Pinkal. Radical underspecification. In *Proceedings of the 10th Amsterdam Colloquium*, pages 587–606, 1996.
15. J. Rogers and V. Shanker. Reasoning with descriptions of trees. In *Proceedings of the 30th Meeting of the Association for Computational Linguistics (ACL'92)*, pages 72–80, 1992.