The *tree* Constraint

Nicolas Beldiceanu[1], Pierre Flener[2,3], and Xavier Lorca[1]

[1] École des Mines de Nantes, LINA FREE CNRS 2729,
FR-44307 Nantes Cedex 3, France
{Nicolas.Beldiceanu,Xavier.Lorca}@emn.fr

[2] Department of Information Technology, Uppsala University,
Box 337, SE–751 05 Uppsala, Sweden
Pierre.Flener@it.uu.se

[3] The Linnaeus Centre for Bioinformatics, Uppsala University,
Box 598, SE–751 24 Uppsala, Sweden

Abstract. This article presents an arc-consistency algorithm for the *tree* constraint, which enforces the partitioning of a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ into a set of vertex-disjoint anti-arborescences. It provides a necessary and sufficient condition for checking the *tree* constraint in $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ time, as well as a complete filtering algorithm taking $\mathcal{O}(|\mathcal{V}| \cdot |\mathcal{E}|)$ time.

1 Introduction

Graph partitioning constraints were already considered from an early stage of constraint programming research as natural shortcuts for expressing constraints on a graph. This was for instance the case of the *Hamiltonian circuit* and *spanning tree* constraints of ALICE [11]. Later on, this was also the case for the *cycle* [3] and *path* constraints [5,14,15], which were respectively introduced in some later version of CHIP [7] and Ilog Solver [12]. But curiously, despite its study within the Operations Research and algorithm design communities [6,13], the problem of partitioning a digraph into a set of vertex-disjoint anti-arborescences[4] was so far ignored by the constraint programming community. This problem has a lot of practical applications, for instance in VLSI circuit design. The application that motivated us is the construction of a supertree from given trees with overlapping leaf sets, such that the ancestor relationships of the given trees are preserved. This is an important issue in phylogeny and has applications in molecular biology and linguistics [1], such as the construction of the Tree of Life [4]. See the description of future work in Section 4 for how this phylogenetic problem relates to the problem described here.

This paper addresses the mentioned digraph partitioning problem from a constraint programming perspective.[5] We should stress that, as usual within

[4] A digraph $\mathcal{A}$ is an *anti-arborescence* with *anti-root* r iff there exists a path from all vertices of $\mathcal{A}$ to r and the undirected graph associated with the digraph $\mathcal{A}$ is a tree.

[5] The term "tree constraint" exists in the constraint programming community, but the *tree processing* problem defined in [1] assembles a set of trees in one single tree according to some *dominance* constraints.

constraint programming, our goal is not partitioning a given digraph $\mathcal{G}$ into vertex-disjoint anti-arborescences, but rather first to find out whether this is possible at all or not, and second to detect those arcs of $\mathcal{G}$ that do not belong to any partitioning. Throughout this article, we use for simplicity the term *tree* rather than the term *anti-arborescence*.

The *tree* constraint has the form *tree*($\texttt{NTREE}$, $\texttt{VERTICES}$), where $\texttt{NTREE}$ is a integer variable[6] and $\texttt{VERTICES}$ is a collection of n items, each item consisting of the following attributes:

- $\texttt{index}$ is an integer between 1 and n.
- $\texttt{father}$ is an integer variable whose domain is a subset of the values of the interval $[1, n]$.

The i-th item of the $\texttt{VERTICES}$ collection is denoted $\texttt{VERTICES}[i]$. Furthermore, $\texttt{VERTICES}[i].\texttt{attr}$ represents the value of attribute $\texttt{attr}$ of $\texttt{VERTICES}[i]$. A collection of n items, each having p attributes $a_1, a_2, ..., a_p$ is denoted by:

$$\{(a_1 - v_{11}, ..., a_p - v_{1p}), (a_1 - v_{21}, ..., a_p - v_{2p}), ..., (a_1 - v_{n1}, ..., a_p - v_{np})\}$$

In order to define the *tree* constraint we first introduce the digraph associated with any instance of the *tree* constraint. We then define the meaning of the *tree* constraint as a graph property that must hold on the digraph associated with a ground[7] instance of the *tree* constraint.

Definition 1. ***Digraph associated with a* tree *constraint***
To any tree($\texttt{NTREE}$, $\texttt{VERTICES}$) *constraint we associate the digraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, *where:*

- *To each item* $\texttt{VERTICES}[i]$, $(1 \leq i \leq n)$, *of the* $\texttt{VERTICES}$ *collection corresponds a vertex of* $\mathcal{V}$ *denoted by* v_i. *Observe that* $|\mathcal{V}| = n$.
- *For every pair of items* ($\texttt{VERTICES}[i]$,$\texttt{VERTICES}[j]$), *where* i *and* j *are not necessarily distinct, there is an arc from* v_i *to* v_j *in* $\mathcal{E}$ *(i.e.,* $(v_i, v_j) \in \mathcal{E}$*) if* $j \in \texttt{dom}(\texttt{VERTICES}[i].\texttt{father})$. *Let:*

$$m = |\mathcal{E}| = \sum_{i=1}^{n} |dom(\texttt{VERTICES}[i].\texttt{father})|$$

Observe that each vertex of the digraph $\mathcal{G}$ associated with a ground instance of the *tree* constraint has exactly one successor.

Definition 2. *A ground instance of a* tree($\texttt{NTREE}$, $\texttt{VERTICES}$) *constraint* holds *if* $\texttt{VERTICES}[i].\texttt{index} = i$, $(1 \leq i \leq n)$, *and if its associated digraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *verifies the two following conditions:*

- $\mathcal{G}$ *consists of* $\texttt{NTREE}$ *connected components.*

---

[6] A integer variable V is a variable that ranges over a finite set of integers denoted by $dom(V)$. $min(V)$ and $max(V)$ respectively denote the minimum and the maximum value of V.

[7] An instance such that all its integer variables are fixed.

- *Each connected component of $\mathcal{G}$ does not contain any circuit involving more than one vertex.*

The `index` and the `father` attributes of an item can be respectively interpreted as the unique identifier of that item and as the successor of that item in the partionning into trees.

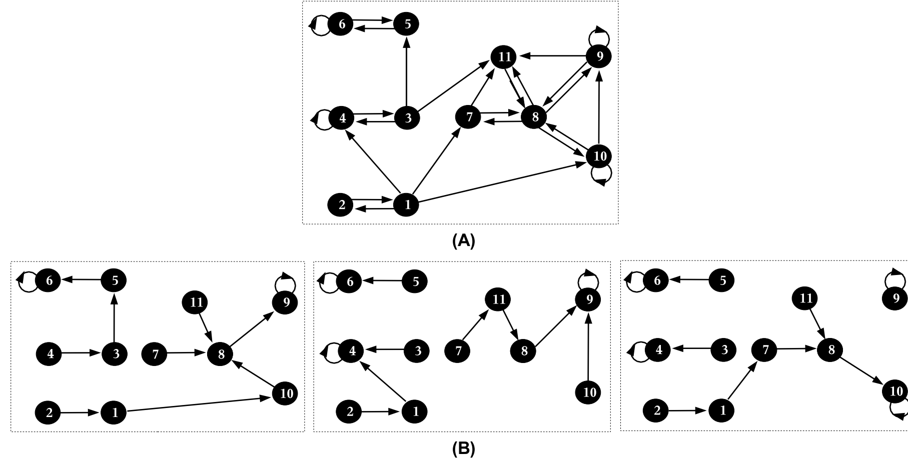

Fig. 1. (A) A digraph $\mathcal{G}$ and **(B)** three possible vertex-disjoint tree partitionings of $\mathcal{G}$.

Example 1. For the digraph depicted by part (A) of Figure 1, a *tree* constraint is stated as *tree*(NTREE, VERTICES) where:

$$\begin{aligned}\texttt{VERTICES} = \{&(\texttt{index}-1, \texttt{father}-F_1), (\texttt{index}-2, \texttt{father}-F_2),\\ &(\texttt{index}-3, \texttt{father}-F_3), (\texttt{index}-4, \texttt{father}-F_4),\\ &(\texttt{index}-5, \texttt{father}-F_5), (\texttt{index}-6, \texttt{father}-F_6),\\ &(\texttt{index}-7, \texttt{father}-F_7), (\texttt{index}-8, \texttt{father}-F_8),\\ &(\texttt{index}-9, \texttt{father}-F_9), (\texttt{index}-10, \texttt{father}-F_{10}),\\ &(\texttt{index}-11, \texttt{father}-F_{11})\},\end{aligned}$$

$dom(\texttt{NTREE})$, $dom(F_1)$, $dom(F_2)$, $dom(F_3)$, $dom(F_4)$, $dom(F_5)$, $dom(F_6)$, $dom(F_7)$, $dom(F_8)$, $dom(F_9)$, $dom(F_{10})$, $dom(F_{11})$ respectively are $\{1,2,3,4,5\}$, $\{2,4,7,10\}$, $\{1\}$, $\{4,5,11\}$, $\{3,4\}$, $\{6\}$, $\{5,6\}$, $\{8,11\}$, $\{7,9,10,11\}$, $\{8,9,11\}$, $\{8,9,10\}$, $\{8\}$.

Part (B) of Figure 1 shows three possible solutions of the vertex-disjoint partitioning of $\mathcal{G}$ with respectively 2, 3 and 4 trees. Observe that, as stated by the second condition of Defintion 2, there is no circuit involving more than one vertex. In order to achieve arc-consistency we have to prune NTREE as well as $F_1, F_2, \ldots, F_{11}$ in the following way:

- We want to find out that 1 and 5 are not feasible numbers of trees for partitioning $\mathcal{G}$, then $dom(\texttt{NTREE}) = \{2,3,4\}$.

– According to the previous restriction of $dom(\texttt{NTREE})$, we restrict the domains of F_1, F_6, F_8 respectively to $dom(F_1) = \{4, 7, 10\}$, $dom(F_6) = \{6\}$ and $dom(F_8) = \{9, 10\}$.

Example 1 will be used throughout this article in order to illustrate the different propositions.

The *tree* constraint was introduced within a catalogue of global constraints [2, page 74] but no filtering algorithm was known. The contribution of this article is an $\mathcal{O}(n \cdot m)$ arc-consistency filtering algorithm for the *tree* constraint.

The rest of the article is organised as follows: Section 2 provides a necessary and sufficient condition for partitioning the digraph $\mathcal{G}$ associated with a *tree* constraint according to a given set $dom(\texttt{NTREE})$ of potential numbers of trees. Section 3 shows how to exploit this necessary and sufficient condition in order to prune NTREE as well as the father variables. Finally, Section 4 concludes this article and outlines future work.

2 Checking Feasibility

This section first gives a necessary and sufficient condition for the *tree* constraint to hold. Second, it sketches an $\mathcal{O}(n+m)$ algorithm for evaluating that condition. Before presenting it, we introduce some terminology regarding the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ associated with a *tree* constraint, as well as a lower and upper bound on the number of trees needed for partitioning $\mathcal{G}$:

– To each instance of a *tree*(NTREE, VERTICES) constraint we associate the *reduced digraph* $\mathcal{G}_r$ derived from $\mathcal{G}$ in the following way: to each strongly connected component of $\mathcal{G}$ we associate a vertex of $\mathcal{G}_r$; to each arc of $\mathcal{G}$ that connects different strongly connected components corresponds an arc in $\mathcal{G}_r$. A strongly connected component of $\mathcal{G}$ that corresponds to a sink of $\mathcal{G}_r$ is called a *sink component*.
– A vertex v of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $(v, v) \in \mathcal{E}$ is called a *potential root*. The arc (v, v) is called a *loop*. A strongly connected component of $\mathcal{G}$ that contains at least one potential root is called a *rooted component*.
– A vertex u of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a *door* of the strongly connected component associated with u iff there exists $(u, v) \in \mathcal{E}$ such that u and v do not belong to the same strongly connected component of $\mathcal{G}$.
– A *connecting arc* (u, v) of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an arc of $\mathcal{E}$ such that u and v do not belong to the same strongly connected component. Similary, a *non-connecting arc* (u, v) of $\mathcal{E}$ is an arc such that u and v belong to the same strongly connected component.
– A vertex v of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a *winner* if v is a *door* or if $(v, v) \in \mathcal{E}$, i.e., a potential root.
– *Enforcing an arc* (u, v) of $\mathcal{G}$ corresponds to removing from $\mathcal{G}$ all arcs (u, w) such that $w \neq v$.

Example 2. Figure 2 illustrates the previous terms according to the digraph introduced in part (A) of Figure 1. In part (A), the *winners* correspond to the *doors* and *potential roots*. The *connecting arcs* and the *loops* are depicted by a black line, while the other arcs are depicted by a dotted line. S_2, S_3, S_4 are *rooted components* while S_3, S_4 are *sink components*. Part (B) depicts the reduced digraph associated with $\mathcal{G}$. To each strongly connected component $\mathcal{S}_i$ of $\mathcal{G}$ corresponds a vertex $\mathcal{R}_i$ of $\mathcal{G}_r$. Observe that R_3 and R_4 represent *sink* vertices.

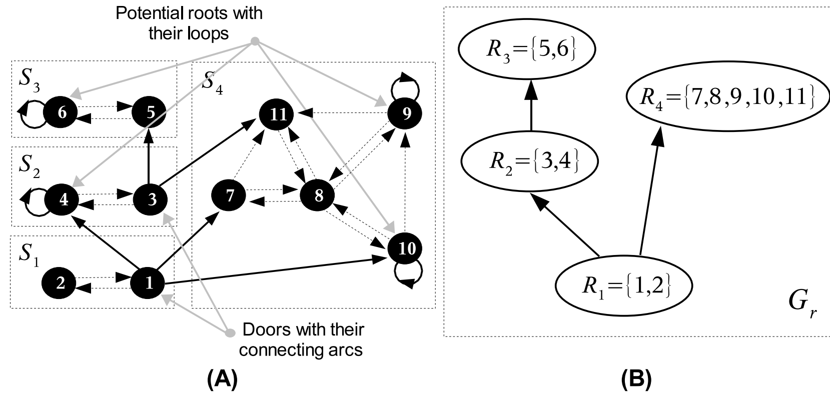

Fig. 2. (A) The digraph $\mathcal{G}$ and its strongly connected components S_1, S_2, S_3, S_4. **(B)** The reduced digraph $\mathcal{G}_r$ associated with $\mathcal{G}$.

We now present a lower and upper bound on the number of trees that can possibly cover a given digraph $\mathcal{G}$ associated with a *tree* constraint. For this purpose, we name by `MINTREE` the number of sinks of $\mathcal{G}_r$ and by `MAXTREE` the number of potential roots of $\mathcal{G}$.

Proposition 1. *A lower bound on the minimum number of trees for partitioning the digraph $\mathcal{G}$ associated with a* tree *constraint is the number of sinks in $\mathcal{G}_r$ (i.e.,* `MINTREE`*).*

Proof. Proposition 1 stems from the fact that there is no path between two vertices that belong to two distinct sink components of $\mathcal{G}$. □

Proposition 2. *An upper bound on the maximum number of trees for partitioning the digraph $\mathcal{G}$ associated with a* tree *constraint is the number of potential roots of $\mathcal{G}$ (i.e.,* `MAXTREE`*).*

Proof. Since each tree has a distinct root, we cannot have more trees than the number of potential roots. □

We now state the necessary and sufficient condition to verify on a *tree* constraint in order to have at least one solution.

Proposition 3. ***Necessary and sufficient condition for a* tree *constraint***
A constraint tree($\mathtt{NTREE}, \mathtt{VERTICES}$) *has at least one solution iff the following two conditions both hold:*

(1) All sink components of $\mathcal{G}$ are rooted components,
(2) $dom(\mathtt{NTREE}) \cap [\mathtt{MINTREE}, \mathtt{MAXTREE}] \neq \emptyset$.

Proof. We first prove that the conjunction of (1) and (2) is a necessary condition. If a sink component of $\mathcal{G}$ is not a rooted component, then there will be at least one circuit in $\mathcal{G}$ among a subset of vertices associated with this component, and the *tree* constraint cannot hold. Moreover, if $dom(\mathtt{NTREE}) \cap [\mathtt{MINTREE}, \mathtt{MAXTREE}] = \emptyset$ then $max(\mathtt{NTREE}) < \mathtt{MINTREE}$ or $min(\mathtt{NTREE}) > \mathtt{MAXTREE}$. And we know, from Propositions 1 and 2, that the *tree* constraint then has no solutions. Secondly, we prove that the conjunction of (1) and (2) is sufficient. For this purpose, we show, in a two step construction, that for each value t in $[\mathtt{MINTREE}, \mathtt{MAXTREE}]$, there exists at least one vertex-disjoint partitioning of $\mathcal{G}$ into t distincts trees. Step 1 selects t root vertices and chooses for each strongly connected component of $\mathcal{G}$ the vertex that will be the root of a tree or that will be attached to another component. Step 2 constructs for each strongly connected component a spanning forest.

STEP 1

- We choose one potential root r for each sink component of $\mathcal{G}$ and we enforce the loop (r, r) on r. Let $\mathcal{R}_1$ denote the set of thus selected roots.
- If $t > \mathtt{MINTREE}$ then we choose a set $\mathcal{R}_2$ of $t - \mathtt{MINTREE}$ potential roots in $\mathcal{G}$, distinct from $\mathcal{R}_1$, and we enforce a loop for each vertex of $\mathcal{R}_2$.
- For all strongly connected components for which we did not enforce a loop, we choose one vertex v that is a door and we enforce a connecting arc starting from v. Let $\mathcal{R}_3$ denote the set of thus selected doors.

STEP 2

For a given strongly connected component $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ of $\mathcal{G}$:

- Let $\mathcal{H}_\mathcal{S} = \mathcal{V}_\mathcal{S} \cap (\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3)$,
- Let $\mathcal{L}_\mathcal{S} = \mathcal{V}_\mathcal{S} - \mathcal{H}_\mathcal{S}$.

For each strongly connected component $\mathcal{S}$ of $\mathcal{G}$, we call the function introduced in Lemma 1 (see the Appendix), $\mathtt{TreeCovering}(\mathcal{S}, \mathcal{H}_\mathcal{S}, \mathcal{L}_\mathcal{S}, \emptyset)$, in order to build a vertex-disjoint partitioning of $\mathcal{S}$ with $|\mathcal{H}_\mathcal{S}|$ trees and having their roots in $\mathcal{H}_\mathcal{S}$.

Thus, we have shown how to build a vertex-disjoint partitioning of $\mathcal{G}$ with t trees, for all $t \in [\mathtt{MINTREE}, \mathtt{MAXTREE}]$. And, since $dom(\mathtt{NTREE}) \cap [\mathtt{MINTREE}, \mathtt{MAXTREE}] \subseteq [\mathtt{MINTREE}, \mathtt{MAXTREE}]$, we know that there exists at least one solution for the *tree* constraint. □

Proposition 4. *The worst-case complexity for checking the necessary and sufficient condition for a* tree *constraint (i.e., Proposition 3) is $\mathcal{O}(n + m)$ time.*

Proof. Evaluating the worst-case complexity for implementing Proposition 3 is done by analysing the following items:

1- Computing the strongly connected components of $\mathcal{G}$ takes $\mathcal{O}(n + m)$ time with Tarjan's algorithm [16].
2- Checking that each sink component of $\mathcal{G}$ contains at least one potential root takes $\mathcal{O}(n)$ time.
3- Checking that $dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$ takes $\mathcal{O}(1)$ time.

Observe that the worst-case complexity makes the following hypotheses on the complexity of the primitives that access the domains of the variables:

- In item 3, we assume that we can get the minimum and maximum values of a integer variable in $\mathcal{O}(1)$ time.
- Since item 1 uses depth-first search, we need to iterate over the successors of a vertex of $\mathcal{G}$. This is done by iterating through the potential values of a `father` variable. In order to achieve $\mathcal{O}(n+m)$ time, getting the next successor (i.e., the next value of a `father` variable) needs to be done in $\mathcal{O}(1)$ time. Therefore we assume that a domain is represented by a list of intervals. □

3 Domain Filtering

This section first shows how to prune the domains of the `father` variables $F_1, F_2, \ldots, F_n$ and of the variable `NTREE` from the digraph $\mathcal{G}$ associated to a *tree* constraint. All the pruning rules are derived from the necessary and sufficient condition given by Proposition 3. Then it proves the completeness of the previous pruning rules and finally sketches an $\mathcal{O}(n \cdot m)$ arc-consistency filtering algorithm.

The pruning rules remove some arcs of the digraph $\mathcal{G}$ associated with a *tree* constraint. Observe that since there is a one to one correpondence between the arcs of $\mathcal{G}$ and the `father` variables and their respective domains, removing an arc (u, v) from $\mathcal{G}$ is equivalent to removing value v from the domain of variable F_u.

3.1 Filtering for a *tree* Constraint

We first present a proposition that restricts the domain of `NTREE` according to condition (2) of Proposition 3.

Proposition 5. *The domain of* `NTREE` *is restricted by the two following rules:*

- *If* $max(\texttt{NTREE}) > \texttt{MAXTREE}$ *then* $max(\texttt{NTREE}) = \texttt{MAXTREE}$ *(3)*
- *If* $min(\texttt{NTREE}) < \texttt{MINTREE}$ *then* $min(\texttt{NTREE}) = \texttt{MINTREE}$ *(4)*

Proof. Conditions 3 and 4 are respectively derived from Propositions 2 and 1. □

Example 3. We illustrate how to prune the domain of `NTREE` according to the digraph $\mathcal{G}$ depicted by part (A) of Figure 1. As $\mathcal{G}$ contains 2 sink components and 4 potential roots, `MINTREE` and `MAXTREE` are respectively equal to 2 and 4. Therefore, assuming that $dom(\texttt{NTREE}) = \{1, 2, 3, 4, 5\}$, Proposition 5 removes the values 1 and 5 from $dom(\texttt{NTREE})$.

Before presenting the next proposition, we need to introduce the notion of *strong articulation point* given by Gondran and Minoux in [9, page 175].

Definition 3. *A* strong articulation point *of a strongly connected digraph $\mathcal{G}$ is a vertex such that if we remove it, $\mathcal{G}$ is broken into at least two strongly connected components.*

The withdrawal of a strong articulation point p, in a strongly connected component $\mathcal{S}$ of the digraph $\mathcal{G}$ associated with a *tree* constraint, creates two types of strongly connected components:

- let $\Delta^p_{out} = \{\mathcal{S}^1_{out}, \ldots, \mathcal{S}^l_{out}\}$ be the possibly empty set of new strongly connected components from which we can reach, by a path that does not contain p, a *winner* of $\mathcal{S}$.
- let $\Delta^p_{in} = \{\mathcal{S}^1_{in}, \ldots, \mathcal{S}^q_{in}\}$ be the possibly empty set of new strongly connected components from which we cannot reach, by a path that does not contain p, a *winner* of $\mathcal{S}$.

Property 1. Let p be a strong articulation point of a strongly connected component of $\mathcal{G}$. Then p belongs to all paths from any vertex of Δ^p_{in} to any vertex of Δ^p_{out}.

Proposition 6. *An outgoing arc (p, v) of a strong articulation point p that reaches a vertex v of a strongly connected component of Δ^p_{in} never belongs to any solution of a* tree *constraint.*

Proof. For any outgoing arc (p, v) of a strong articulation point p, if v belongs to a strongly connected component of Δ^p_{in}, then, by Property 1, every path from v to any vertex of a strongly connected component of Δ^p_{out} contains p. Thus, enforcing (p, v) creates a circuit with some vertices of Δ^p_{in} and p. Therefore, (p, v) never belongs to any solution of a *tree* constraint. □

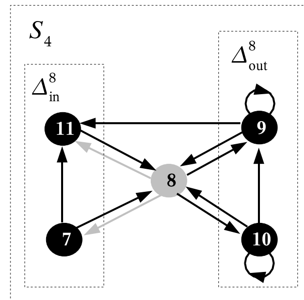

Fig. 3. Pruning according to a strong articulation point.

Example 4. Figure 3 illustrates Proposition 6 on the strongly connected component S_4 of the digraph $\mathcal{G}$ depicted by part (A) of Figure 1. Vertex 8 is a strong articulation point since its removal breaks S_4 into four strongly connected components. 9 and 10 are potential root vertices, and since 7 and 11 have neither loops nor connecting arcs, we have $\Delta^8_{out} = \{\{9\},\{10\}\}$ and $\Delta^8_{in} = \{\{7\},\{11\}\}$. Then, from Proposition 6, the arcs $(8,7)$ and $(8,11)$ (drawn in gray in Figure 3) are infeasible.

We now introduce a final proposition that allows us to prune according to the fact that we have to build a vertex-disjoint partitioning compatible with $dom(\texttt{NTREE})$.

Proposition 7. *Let $\mathcal{C} = dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}]$. For each strongly connected component $\mathcal{S}$ of $\mathcal{G}$:*

1. *If $\mathcal{S}$ is a sink component of $\mathcal{G}$ that contains one single potential root r, then all the outgoing arcs of r, except the loop (r,r), are infeasible.*
2. *Otherwise:*
   - 2.1. *If $\mathcal{C} = \{\texttt{MAXTREE}\}$ then, for each potential root r of $\mathcal{S}$, all the non-loop arcs (r,v) $(v \neq r)$ are infeasible.*
   - 2.2. *If $\mathcal{C} = \{\texttt{MINTREE}\}$ and $\mathcal{S}$ is a non-sink component then all the loops of $\mathcal{S}$ are infeasible.*
   - 2.3. *If there exists a single winner w in $\mathcal{S}$, which is a door, then all the non-connecting arcs (w,v) are infeasible.*

Proof. For item 1, let r be the potential root of $\mathcal{S}$ and assume that we enforce an outgoing arc (r,v) $(v \neq r)$. Then, as $\mathcal{S}$ does not contain any doors, we cannot leave $\mathcal{S}$ and thus create a circuit involving at least two vertices. Item 2.1 (respectively 2.2) is a direct consequence of Proposition 2 (respectively Proposition 1). For item 2.3, assume that we have a single door w in $\mathcal{S}$ and consider that no potential root belongs to $\mathcal{S}$. If we do not take a connecting arc of w, then we can never leave $\mathcal{S}$ and therefore we create a circuit in $\mathcal{S}$ involving at least two vertices. Thus, the *tree* constraint cannot hold. $\square$

Example 5. In order to illustrate Proposition 7 on the digraph depicted by part (A) of Figure 1, we consider the following three cases:

- First, we do not assume any restriction on $dom(\texttt{NTREE})$. Then, in this context, item 1 removes the arc $(6,5)$, while item 2.3 removes $(1,2)$.
- If NTREE is equal to MAXTREE (i.e., 4) then, in addition to the arcs removed by items 1 and 2.3, item 2.1 removes the arcs $(4,3)$, $(9,8)$, $(9,11)$, $(10,8)$ and $(10,9)$ since a loop is enforced for each of the vertices 4, 9 and 10.
- If NTREE is equal to MINTREE (i.e., 2) then, in addition to the arcs removed by items 1 and 2.3, item 2.2 removes the arc $(4,4)$.

3.2 Arc-consistency

Now, we prove that Propositions 5, 6 and 7 characterise all the arcs that do not belong to any solution of a *tree* constraint. For this purpose, we assume that the necessary and sufficient condition of Proposition 3 holds.

Proposition 8. *If Proposition 3 holds then the two following equivalences lead to the completeness of the pruning rules:*

- *Let $t \in dom(\texttt{NTREE})$, then t is incompatible with a* tree *constraint if and only if t is pruned by Proposition 5.*
- *Let $(u, v) \in \mathcal{E}$, then (u, v) is incompatible with a* tree *constraint if and only if (u, v) is removed by at least one proposition among Propositions 6 and 7.*

Proof. We first prove that Proposition 5 removes all infeasible values in the domain of NTREE. Indeed, we have completeness since Proposition 3 enforces for each $t \in [\texttt{MINTREE},\ \texttt{MAXTREE}]$ the existence of a vertex-disjoint partitioning of $\mathcal{G}$ with t trees.

Second, we prove that Propositions 6 and 7 remove all infeasible values for the `father` variables. Now, for each arc (u, v) that was not pruned by Propositions 6 or 7, we show how to build a vertex-disjoint partitioning of $\mathcal{G}$ with t trees, where $t \in dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}]$.

STEP 1

Let $dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] = [\texttt{mintree}, \texttt{maxtree}]$,

A1 **Selecting a root in each sink component of $\mathcal{G}$**:
For each sink component $\mathcal{S}$ of $\mathcal{G}$, if u is a potential root of $\mathcal{S}$ and $u = v$ then (u, v) has to be enforced. Otherwise, we select a potential root r of $\mathcal{S}$ different from u and we enforce the loop (r, r). Observe that item 1 of Proposition 7 garanties us to find a potential root different from u. Let $\mathcal{R}_1$ denote the set of thus selected roots in the different sink components of $\mathcal{G}$.

A2 **Completing the set of roots in order to get** $\texttt{mintree}$ **or** $\texttt{mintree} + 1$ **trees:**

CASE 1: $\texttt{mintree} > \texttt{MINTREE}$
Since we have to build at least mintree trees, we choose to build exactly mintree trees. Therefore, we enforce a set $\mathcal{R}_2$ of $\texttt{mintree} - \texttt{MINTREE}$ potential roots distinct from $\mathcal{R}_1$. Observe that if $u = v$ and u does not belong to any sink component then u must belong to $\mathcal{R}_2$.

CASE 2: $\texttt{mintree} = \texttt{MINTREE}$

- If $u = v$ and u does not belong to any sink component then $\texttt{MINTREE} < \texttt{MAXTREE}$ and we have to enforce the loop (u, u), and $\mathcal{R}_2 = \{u\}$. Therefore, we choose to build $\texttt{mintree} + 1$ trees.
- Otherwise, we build mintree trees and $\mathcal{R}_2 = \emptyset$.

A3 **Selecting a *door* in the strongly connected components that do not contain a vertex of $\mathcal{R}_1 \cup \mathcal{R}_2$.**
For all the strongly connected components $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ for which no loops are enforced (i.e., $\mathcal{V}_\mathcal{S} \cap (\mathcal{R}_1 \cup \mathcal{R}_2) = \emptyset$):

- If $u \in \mathcal{V}_\mathcal{S}$ and (u, v) is a connecting arc, then (u, v) is enforced. Observe that if u is the only door of $\mathcal{S}$ then $v \notin \mathcal{V}_\mathcal{S}$ by item 2.3 of Proposition 7.
- Otherwise, if $u, v \in \mathcal{V}_\mathcal{S}$ or $u \notin \mathcal{V}_\mathcal{S}$ then a door w, different from u, is chosen and we enforce one of its connecting arcs.

Let $\mathcal{R}_3$ denote the set of thus selected doors.

STEP 2

For a given strongly connected component $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$:

- Let $\mathcal{H}_\mathcal{S} = \mathcal{V}_\mathcal{S} \cap (\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3)$.
- Let $\mathcal{L}_\mathcal{S} = \mathcal{V}_\mathcal{S} - \mathcal{H}_\mathcal{S}$.
- Let $\mathcal{A}_\mathcal{S} = \{(u, v)\}$ if $u, v \in \mathcal{V}_\mathcal{S}$, otherwise $\mathcal{A}_\mathcal{S} = \emptyset$.

For each strongly connected component $\mathcal{S}$ of $\mathcal{G}$, we call the function introduced in Lemma 1 (see the Appendix), `TreeCovering`$(\mathcal{S}, \mathcal{H}_\mathcal{S}, \mathcal{L}_\mathcal{S}, \mathcal{A}_\mathcal{S})$, in order to build a vertex-disjoint partitioning of $\mathcal{S}$ with $|\mathcal{H}_\mathcal{S}|$ trees that includes (u, v) if $u, v \in \mathcal{V}_\mathcal{S}$.

Thus, for each arc $(u, v) \in \mathcal{E}$ that is not pruned by Propositions 6 or 7, we have shown how to build a vertex-disjoint partitioning of $\mathcal{G}$ with t trees, where $t \in dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}]$. □

3.3 Polynomial Arc-Consistency Algorithm

We show how to process all the pruning rules in $\mathcal{O}(n \cdot m)$ time. Two parts are distinguished, the first one only considers the pruning according to the strong articulation points (Proposition 6), the second one considers the pruning of `NTREE` (Proposition 5) and the pruning related to Proposition 7.

Then, in the first part, we are interested in Proposition 6 and we propose the `TreeFiltering` algorithm below. For this purpose we have to detect all the strong articulation points of a strongly connected component of $\mathcal{G}$:

- Finding the strong articulation points takes at the maximum $\mathcal{O}(n \cdot m)$ time because we have not found a more efficient algorithm than withdrawing a vertex and testing if the remaining subgraph is strongly connected or not.
- Detecting the arcs to be pruned takes $\mathcal{O}(n \cdot m)$ time because for each of the strongly connected components $\mathcal{S}$ we have to withdraw each strong articulation point p detected:
  - we have to search Δ^p_{in}, thanks to a depth-first search beginning from the winners of $\mathcal{S}$.
  - we mark the vertices reachable from at least one winner as vertices of Δ^p_{out}.
  - we remove the arcs, that reach Δ^p_{in} vertices from p, according to Proposition 6.

Now, in the second part, it is straightforward that the pruning related to Proposition 5 is carried out in constant time. Moreover, the pruning related to the general Proposition 7 consists of four steps:

- Items 1 and 2.3 of Proposition 7: the time complexity of these steps lies in the construction of the reduced digraph, which takes $\mathcal{O}(m+n)$ time.
- Item 2.1 of Proposition 7: all the potential roots have to be detected, that takes $\mathcal{O}(n)$ time.
- Item 2.2 of Proposition 7: we have to detect all the non-sink components of $\mathcal{G}$; then the time complexity lies in the construction of the depth-first search in $\mathcal{O}(n+m)$ time.

`TreeFiltering`$(\mathcal{G}) : \mathcal{R}$.
Input : the digraph $\mathcal{G}$.
Output : $\mathcal{R}$ the set of prunable arcs of $\mathcal{G}$.
$\mathcal{R} \leftarrow \emptyset$; \\ *the set of arcs pruned.*
$W \leftarrow \{u \mid u \; is \; a \; winner\}$;
For each vertex scc **of** $\mathcal{G}$ **do**
  $\Psi_{scc} \leftarrow \emptyset$; \\ *the set of the strong articulation points of scc.*
  \\ *we detect the strong articulation points (s.a.p.) of scc.*
  For each vertex u **of** scc **do**
    if scc **without** u **is not strongly connected then** $\Psi_{scc} \leftarrow \Psi_{scc} \cup \{u\}$;
  \\ *we search infeasible arcs.*
  For each vertex u **of** Ψ_{scc} **do** `PruneArcs`$(u, scc, W, \mathcal{R})$;

`PruneArcs`$(u, scc, W, \mathcal{R}) : \mathcal{R}$
Input : a strongly connected component scc, a strong articulation point u of scc, the set W of winners and the set $\mathcal{R}$.
Output : $\mathcal{R}$ the set of prunable arcs, increased by those discovered in scc.
For each vertex v **of** scc **do** $reach[v] \leftarrow false$;
\\ *detecting the blocks obtained by the withdrawal of each s.a.p. of scc.*
$\Sigma^u_{scc} \leftarrow \{C^u_1, ..., C^u_m\}$;
\\ *we search in each block the infeasible arcs.*
For each $C^u_i \in \Sigma^u_{scc}$ **do**
  $search[C^u_i] \leftarrow false$;
  For each $w \in W$ **such that** $(w \in C^u_i \wedge \neg reach[w]) \vee (w = u \wedge \neg search[C^u_i])$ **do**
    `Visit`$(w, u, reach[\,])$;
  $search[C^u_i] \leftarrow true$;
  \\ *withdrawal of infeasible outgoing arcs of u.*
  If $\exists (u,v) \in \mathcal{E}$ **such that** $v \in C^u_i \wedge \neg reach[v]$ **then**
    $W \leftarrow W \cup \{u\}$;
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(u,v)\}$;

`Visit`$(v, u, reach[\,]) : reach[\,]$
Input : a *winner* v, a strong articulation point u and a boolean table $reach[\,]$.
Output : the table $reach[\,]$.
$reach[v] \leftarrow true$;
For each $v \neq u$ **and** $w \in pred[v]$ **such that** $\neg reach[w]$ **do** `Visit`$(w, u, reach[\,])$;

`TreeFiltering` **algorithm**

Example 6. We present a trace of `TreeFiltering` according to the strongly connected component depicted by Figure 3:

- In `TreeFiltering`: $\mathcal{R} \leftarrow \emptyset$; $W \leftarrow \{9, 10\}$; $\Psi_{\mathcal{S}_4} \leftarrow \{8\}$; `PruneArcs`$(8, \mathcal{S}_4, W, \mathcal{R})$.
- In `PruneArcs`: for each $u \in \{7, 8, 9, 10, 11\}$ do $reach[u] \leftarrow false$; $\Sigma^8_{\mathcal{S}_4} \leftarrow \{\{7\}, \{9\}, \{10\}, \{11\}\}$. We process a depth-first search with the recursive function `Visit`(). Finally, vertices 7 and 11 are not reachable from 9 or 10, then $\Delta^8_{in} = \{\{7\}, \{11\}\}$ and $\Delta^8_{out} = \{\{9\}, \{10\}\}$, thus $\mathcal{R} \leftarrow \{(8, 7), (8, 11)\}$ according to Proposition 6.

4 Conclusion and Perspectives

This article provides an arc-consistency algorithm description and a necessary and sufficient condition for the *tree* constraint.

On the one hand, the necessary and sufficient condition for the *tree* constraint consists in two conditions checked in $\mathcal{O}(n + m)$ time (Proposition 3). On the other hand, the key point of the arc-consistency algorithm is the detection and processing (Proposition 6) of the strong articulation points. Unfortunately, to our knowledge, the existence of an $\mathcal{O}(m)$ algorithm is an open problem, thus the current complexity is $\mathcal{O}(n \cdot m)$ time. Furthermore, note that it would be possible to get a relaxed $\mathcal{O}(n+m)$ time algorithm using a subset of the strong articulation points. A natural choice for such a subset is the set of articulation points[8] provided without the orientation of the arcs of the digraph associated with the *tree* constraint. An implementation of this relaxed algorithm was carried out in *Choco* [10] with the version 1.0 available to `http://choco.sf.net`.

Future work will address the phylogenetic problem of constructing a supertree from given trees with overlapping leaf sets, such that the ancestor relationships of the given trees are preserved. This problem can be modelled in terms of a generalisation of the *tree* constraint, where the `VERTICES` collection has been augmented with an optional attribute giving the direct ancestors of the considered vertex in the given trees, but with `NTREE` $= 1$. Notice that this takes a number of integer variables that is *linear* in the number of vertices, and hence linear in the number of species (the leaves of the given trees), rather than a number quadratic in the number of species as in a previous constraint-programming approach to supertree construction [8]. The advantages of deploying constraint programming on this phylogenetic problem, as opposed to the purely algorithmic approaches advocated so far (see [4] for instance), are that any combination of biological side constraints (on branch lengths, speciation dates, nested species, etc) to the otherwise purely combinatorial problem can be incorporated without having to design a new algorithm each time, that *all* the supertrees can be enumerated without having to generalise an otherwise deterministic algorithm, and that an explanation of why the given trees are incompatible can be provided when no supertree is found. In the latter case, the supertree problem

[8] An *articulation point* [9, page 16] of an undirected graph $\mathcal{G}$ is a vertex v of $\mathcal{G}$ such that if we remove it, the number of connected components of $\mathcal{G}$ deprived of v increases.

can also be re-cast as an optimisation problem, and constraint programming will facilitate experiments with emerging cost functions.

Acknowledgements. This project was partially funded by grant HPRI-CT-2001-00153 within the *Human Research Potential and Socio-Economic Knowledge Base: Access to Research Infrastructures* (ARI) programme of the European Commission, when the first author visited the second author at The Linnaeus Centre for Bioinformatics. Finally, the implementation of the constraint would not have been possible without the relevant advice of Hadrien Cambazard and Guillaume Rochart regarding *JChoco*.

References

1. E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An efficient graph algorithm for dominance constraints. *Journal of Algorithms*, 48(1):194–219, May 2003. Special Issue of SODA 2001.
2. N. Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical report, SICS T2000:01, Sweden, January 2000.
3. N. Beldiceanu and E. Contejean. Introducing global constraint in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. O. R. Bininda-Emonds, editor. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life*. Kluwer, 2004.
5. H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. *JNPC*, pages 107–120, 2004. In French.
6. Z.-Z. Chen. Efficient algorithms for acyclic colorings of graphs. *Theoretical Computer Science*, 230(1-2):75–95, 2000.
7. M. Dincbas, P. V. Hentenryck, H. Simonis, T. G. A. Aggoun, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Int. Conf. on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, 1988.
8. I. P. Gent, P. Prosser, B. M. Smith, and W. Wei. Supertree construction using constraint programming. In F. Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 837–841. Springer-Verlag, 2003.
9. M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, Paris, 2nd edition, 1985. In French.
10. F. Laburthe and the OCRE group. CHOCO: implementing a CP kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, Sept. 2000.
11. J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.
12. J.-F. Puget. A C++ Implementation of CLP. In *Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages 256–261, Singapore, November 1994.
13. A. Roychoudhury and S. Sur-Kolay. Efficient algorithms for vertex arboricity of planar graphs. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 37–51. Springer-Verlag, 1995.
14. M. Sellmann. *Reduction techniques in Constraint Programming and Combinatorial Optimization*. PhD thesis, University of Paderborn, 2002.

15. M. Sellmann. Cost-based filtering for shortest path constraints. In *9th international Conference on the Principles and Practice of Constraint Programming (CP)*, volume 2833 of *LNCS*, pages 694–708. Springer-Verlag, 2003.
16. R. Tarjan. Depth-first search and linear graph algorithms. In *SIAM J. Comput.*, volume 1, pages 146–160, 1972.

Appendix

Lemma 1 is used in the proofs of Propositions 3 and 8. It presents a constructive method for building a vertex-disjoint partitioning into anti-arborescences of a particular digraph depicted by the assumptions 1, 2 and 3.

Lemma 1. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a digraph such that:*

(1) Let $\mathcal{H}, \mathcal{L} \subseteq \mathcal{V}$ such that $\mathcal{H} \cup \mathcal{L} = \mathcal{V}$ and $\mathcal{H} \cap \mathcal{L} = \emptyset$.
(2) For each $v \in \mathcal{L}$ there exists a path from v to at least one vertex of $\mathcal{H}$.
(3) Let $\mathcal{A} \subset \mathcal{E}$ such that $|\mathcal{A}| \leq 1$ and if $|\mathcal{A}| = 1$ then $\mathcal{A} = \{(u, v)\}$.

The following algorithm computes $|\mathcal{H}|$ vertex-disjoint anti-arborescences and having their roots in $\mathcal{H}$:

`TreeCovering`$(\mathcal{G}, \mathcal{H}, \mathcal{L}, \mathcal{A}) : \mathcal{F}$
Input *: $\mathcal{G}, \mathcal{H}, \mathcal{L}, \mathcal{A}$.*
Output *: $\mathcal{F}$, the set of vertex-disjoint anti-arborescences.*

$\mathcal{F} \leftarrow \emptyset$;
While $\mathcal{L} \neq \emptyset$ ***do***
  If $\mathcal{A} \neq \emptyset$ ***and*** $\exists h \in \mathcal{H}$ ***such that*** $(v, h) \in \mathcal{E}$ ***then***
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{(v, h), (u, v)\}$;
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{u, v\}$;
    $\mathcal{L} \leftarrow \mathcal{L} - \{u, v\}$;
  Else
    Let $w \in \mathcal{L}$ ***and*** $h \in \mathcal{H}$ ***such that*** $(w, h) \in \mathcal{E}$;
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{(w, h)\}$;
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{w\}$;
    $\mathcal{L} \leftarrow \mathcal{L} - \{w\}$;

Proof. By assumption (2), we know that from every vertex $w \in \mathcal{L}$ there exists a path to at least one vertex of $\mathcal{H}$. Thus we make sure that $\mathcal{L}$ will become an empty set, i.e., that all vertices of $\mathcal{V}$ are covered. □