Combining Tree Partitioning, Precedence, and Incomparability Constraints

Nicolas Beldiceanu[1], Pierre Flener[2⋆], 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, and
The Linnaeus Centre for Bioinformatics
Uppsala University, Box 337, SE – 751 05 Uppsala, Sweden
Pierre.Flener@it.uu.se

Abstract. The *tree* constraint partitions a directed graph into node-disjoint trees. In many practical applications that involve such a partition, there exist side constraints specifying requirements on tree count, node degrees, or precedences and incomparabilities within node subsets. We present a generalisation of the *tree* constraint that incorporates such side constraints. The key point of our approach is to take partially into account the strong interactions between the tree partitioning problem and all the side constraints, in order to avoid thrashing during search. We describe filtering rules for this extended *tree* constraint and evaluate its effectiveness on three applications: the Hamiltonian path problem, the ordered disjoint paths problem, and the phylogenetic supertree problem.

1 Introduction

Graph partitioning problems are involved in many practical applications such as vehicle routing, mission planning, DNA sequencing, or phylogeny. However, in most real life applications, graph partitioning problems are subjected to various side constraints. This paper proposes a global constraint for partitioning a given digraph into a set of node-disjoint trees under side constraints such as the number of trees, the degree of each node, and a partial order between nodes. For instance, degree constraints can be used to get tree partitions limited to paths (which are "unary" trees) or binary trees. Similarly, a partial order between nodes can be used to express a set of precedence constraints inherent in many tree or path problems [26]. Moreover, a partial order also expresses a set of incomparability constraints (also known as dominance constraints [8]) that can be used to force some nodes to belong to distinct tree branches or trees. Examples of such problems include the construction of a phylogenetic supertree from given species trees [1, 7, 17, 23, 31], possibly under degree constraints requiring the resulting tree to be binary or precedence constraints requiring some species to be nested, as well as digraph partitioning by paths [6, 11, 26], trees [5], or cycles [10].[1]

⋆ Some of this work was done while a Visiting Faculty Member and Erasmus Exchange Teacher at Sabancı University in İstanbul, Turkey, during the academic year 2006/07.

[1] Searching for a cycle in a digraph can be modelled as searching for an elementary path if one of the nodes of the digraph is duplicated.

In [3], we proposed a complete polynomial characterisation of the *tree* constraint, which partitions a given digraph into a forest of node-disjoint trees. This paper extends the original *tree* constraint with the following useful side constraints:

– *Precedence constraints*: a node u *precedes* a node v if there exists a directed path from u to v.
– *Incomparability constraints*: two nodes u and v are *incomparable* if there is no directed path from u to v or from v to u.
– *Degree constraints* that restrict the in-degrees of the nodes in the tree partition.
– *Constraints on the number of proper trees*, where a *proper tree* is a tree involving at least two nodes.

We will show that combining these side constraints is mostly equivalent to solving an NP-hard problem. As this is generally the case for heterogeneous combinatorial problems, a disjoint treatment of each additional restriction is not efficient and leads to repeatedly discovering the same inconsistencies; this phenomenon is called *thrashing*. Thus, the main contribution of this paper is a set of necessary structural conditions combining the input graph with the graphs associated with the precedence and incomparability constraints. These conditions focus on the strong interaction between the side constraints (i.e., precedences, incomparabilities, and degrees) in order to get an improved filtering algorithm that reduces thrashing. As a consequence, the corresponding extended *tree* constraint can directly handle partitioning problems as varied as (extensions to) the phylogenetic supertree and ordered disjoint paths problems, which were previously addressed by ad-hoc approaches.

The rest of this paper is then organised as follows. First Section 2 recalls the necessary background in graph theory. Next, Section 3 first recalls the original *tree* constraint and then shows how to represent the extended version including additional restrictions. Then, Section 4 states the theoretical complexity of the extended *tree* constraint. Section 5 details the additional restrictions related to precedence and incomparability constraints. For each, necessary conditions for partitioning the digraph $\mathcal{G}$ into trees are provided, and pruning rules are derived from these necessary conditions. Next, Section 6 shows that managing the interaction between these constraints allows us to improve the necessary conditions and derive new pruning rules. Section 7 concludes the theoretical part of the paper by a synthetic overview of all theorems and algorithms on the different aspects of the extended *tree* constraint. Section 8 then presents our experimental results with the extended *tree* constraint, including the problems of constructing a supertree of several phylogenetic species trees, possibly under side constraints, and constructing an ordered simple path with mandatory nodes. Finally, Section 9 reviews related work, discusses future work, and concludes.

2 Background: Graph Theory

Most of the filtering algorithms involved in graph-based constraints are directly dependent on classical notions of graph theory. The *tree* constraint introduced in [3] perfectly illustrates this remark, but we can also mention the *allDifferent* and *global cardinality* constraints [27, 28]. Thus, we now introduce the standard notions of graph theory that will be used throughout this paper.

Definition 1 (Digraph). *A* directed graph *(also called* digraph*) $\mathcal{G}$ is a pair $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of objects, called* nodes *(or* vertices*), and $\mathcal{E}$ is a binary relation on $\mathcal{V} \times \mathcal{V}$ that defines a set of ordered pairs of nodes. The elements of $\mathcal{E}$ are called the* arcs *of the digraph.*

Definition 2 (Graph). *A* graph *$\mathcal{G}$ is a pair $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of objects, called* nodes *(or* vertices*), and $\mathcal{E}$ is a binary relation on $\mathcal{V} \times \mathcal{V}$ that defines a set of unordered pairs of distinct nodes. The elements of $\mathcal{E}$ are called the* edges *of the digraph.*

A graph can be represented by a digraph in the following way: for any edge (i, j) of the graph, we have to build two arcs (i, j) and (j, i) in the equivalent digraph. So, in the rest of this section, definitions are provided in the context of directed graphs.

Definition 3 (Partial digraph). *A* partial digraph *$\mathcal{G}' = (\mathcal{V}, \mathcal{S})$ is a digraph induced by a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{S} \subseteq \mathcal{E}$.*

Definition 4 (Source and sink nodes). *Given a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a node $j \in \mathcal{V}$ is a* source *node of $\mathcal{G}$ if for any node $i \in \mathcal{V}$ with $i \neq j$ we have that $(i, j) \notin \mathcal{E}$. Symmetrically, a node $j \in \mathcal{V}$ is a* sink *node of $\mathcal{G}$ if for any node $k \in \mathcal{V}$ with $k \neq j$ we have that $(j, k) \notin \mathcal{E}$.*

Definition 5 (Elementary Path). *Given a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an* elementary path *of length $q > 0$ is a sequence of arcs $\alpha = (u_1, u_2, \ldots, u_q)$ of $\mathcal{G}$ such that the final node of each arc in the sequence coincides with the initial node of the succeeding arc in the sequence, and the sequence does not contain the same node twice.*

Definition 6 (Elementary Cycle). *An* elementary cycle *is a path that begins and ends at the same node, and all the other nodes used are different.*

In the following, we simply use the terms 'path' and 'cycle' rather than the terms 'elementary path' and 'elementary cycle'. Next, we provide some operations that can be applied on digraphs. Given three digraphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$, $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, and $\mathcal{G}_3 = (\mathcal{V}_3, \mathcal{E}_3)$:

- $\mathcal{G}_1 \cup \mathcal{G}_2$ denotes the *union* of $\mathcal{G}_1$ and $\mathcal{G}_2$, that is the graph $(\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2)$.
- $\mathcal{G}_1 \cap \mathcal{G}_2$ denotes the *intersection* of $\mathcal{G}_1$ and $\mathcal{G}_2$, that is the graph $(\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cap \mathcal{E}_2)$.
- $\mathcal{G}_1 \subseteq \mathcal{G}_2$ denotes the *inclusion* of $\mathcal{G}_1$ in $\mathcal{G}_2$, which holds if $\mathcal{G}_1 \cup \mathcal{G}_2 = \mathcal{G}_2$ and $\mathcal{G}_1 \cap \mathcal{G}_2 = \mathcal{G}_1$.
- $\mathcal{G}_1 \setminus \mathcal{V}_2$ denotes the *restriction* of $\mathcal{G}_1$ to the nodes of $\mathcal{V}_1 \setminus \mathcal{V}_2$, precisely $\mathcal{G}_1 \setminus \mathcal{V}_2 = (\mathcal{V}_1 \setminus \mathcal{V}_2, \{(i, j) \in \mathcal{E}_1 \mid i \notin \mathcal{V}_2 \wedge j \notin \mathcal{V}_2\})$.
- $\mathcal{G}_1 \setminus \mathcal{E}_2$ denotes the restriction of $\mathcal{G}_1$ to the arcs of $\mathcal{E}_1 \setminus \mathcal{E}_2$, precisely $\mathcal{G}_1 \setminus \mathcal{E}_2 = (\mathcal{V}_1, \{(i, j) \in \mathcal{E}_1 \mid (i, j) \notin \mathcal{E}_2\})$.
- $TC(\mathcal{G})$ denotes the *transitive closure* of $\mathcal{G}$, that is the graph $(\mathcal{V}, \mathcal{E}')$ such that for all v, w in $\mathcal{V}$ there is an edge (v, w) in $\mathcal{E}'$ iff there is a non-empty path from v to w in $\mathcal{G}$.

– $TR(\mathcal{G})$ denotes the *transitive reduction* of $\mathcal{G}$, that is the smallest graph (under arc inclusion) such that $TC(\mathcal{G}) = TC(TR(\mathcal{G}))$.

Now, we are in position to define classical structures (or properties) related to digraphs.

Definition 7 (Connected component). *A* connected component *of an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a set of nodes $\mathcal{C} \subseteq \mathcal{V}$ such that, for any pair of nodes $(u, v) \in \mathcal{C}$, there exists a path from u to v, or from v to u, in the graph defined by $(\mathcal{C}, \{(i,j) \in \mathcal{E} \mid i \in \mathcal{C} \wedge j \in \mathcal{C}\})$. For a given node i, the* maximum connected component *(under node inclusion) containing node i is denoted by $CC(i)$.*

Definition 8 (Strongly connected component). *A* strongly connected component *of a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a set of nodes $\mathcal{S} \subseteq \mathcal{V}$ such that, for any pair of nodes $(u, v) \in \mathcal{S}$, there exist a path from u to v and a path from v to u in the digraph defined by $(\mathcal{S}, \{(i,j) \in \mathcal{E} \mid i \in \mathcal{S} \wedge j \in \mathcal{S}\})$. For a given node i, the* maximum strongly connected component *(under node inclusion) containing node i is denoted by $SCC(i)$.*

Definition 9 (Sink component). *A* sink component *$\mathcal{S}$ of a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a strongly connected component of $\mathcal{G}$ such that no node in $\mathcal{G} \setminus \mathcal{S}$ is reachable by a path from any node in $\mathcal{S}$.*

Definition 10 (Reduced digraph). *The* reduced digraph *$\mathcal{G}_r$ is derived from a given digraph $\mathcal{G}$ by associating to each strongly connected component of $\mathcal{G}$ a vertex of $\mathcal{G}_r$, and to each arc of $\mathcal{G}$ that connects two different strongly connected components an arc in $\mathcal{G}_r$.*

Definition 11 (Dominator [21]). *Given two distinct nodes i, j of a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that there is at least one path from i to j, a node d is a* dominator *of j with respect to i if and only if there is no path from i to j in $\mathcal{G} \setminus \{d\}$. The set of dominator nodes of j with respect to i is denoted by $DOM_{\langle \mathcal{G}, i \rangle}(j)$.*

Finally, we introduce the notion of tree. In the context of directed graphs, the proper term that denotes a tree is *anti-arborescence*, but in the following, we only use the term tree.

Definition 12 (Tree and Proper Tree). *A digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a* tree *if and only if:*

– *$\mathcal{G}$ is connected (i.e., $\mathcal{G}$ is composed of a single connected component).*
– *$\mathcal{G}$ does not contain any cycles.*
– *Each node of $\mathcal{G}$ has exactly one successor, except the* root*, which is the single sink node in $\mathcal{G}$.*

A proper tree *is an tree with at least two nodes [12].*

Definition 13 (Forest). *A digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a* forest *if and only if each connected component of $\mathcal{G}$ is a tree.*

3 Extending the Original *tree* Constraint

Graph partitioning problems can mostly be reduced to the search of a partial graph respecting a set of properties, induced by an initial graph that defines the problem. So, finding a suitable representation of the initial graph and all the properties associated with the problem has to be done.

In constraint programming, three kinds of approaches to graph problems are generally proposed. The first one associates to each possible edge (respectively arc) a Boolean variable such that it is set to *true* if and only if the edge (respectively arc) is present in the digraph. A second representation associates to each node of the digraph an integer variable[2] whose domain represents the set of potential direct successors in the digraph. Finally, the most recent representation directly deals with graph variables. Such a composite type of variables was proposed in [24, 15]. It combines the representation of each node by an integer variable and the representation of each arc by a Boolean variable.

This section first recalls the initial *tree* constraint introduced in [3]. Next, it presents the extended *tree* constraint, showing how to extend the initial constraint modelling in order to deal with additional restrictions.

3.1 The Original *tree* Constraint

The original *tree* constraint is modelled by an integer variable NTREE specifying the number of trees in the partition and by a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which the node set $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of integer variables and the arc set $\mathcal{E}$ gives the domains $dom(v_i) = \{j \mid (v_i, v_j) \in \mathcal{E}\}$ of these variables, so that they represent the direct successor relation of the partition. A *tree*$(\mathtt{NTREE}, \mathcal{G})$ constraint specifies that its digraph $\mathcal{G}$ should be a forest of NTREE trees. Formally:

Definition 14 (Solution of a *tree* constraint). *A* ground instance *of a* tree$(\mathtt{NTREE}, \mathcal{G})$ *constraint is said to be a* solution *if and only if:*

- *The digraph $\mathcal{G}$ consists of* NTREE *connected components.*
- *Each connected component of $\mathcal{G}$ has no cycle involving more than one node (notice that each component contains exactly one node that has a self-loop and that corresponds to the root of that tree).*

Given a digraph $\mathcal{G}$ containing a set of *potential roots*, where a potential root of $\mathcal{G}$ is a node that can be the root of a tree in a solution of the *tree* constraint, the minimum number of trees (MINTREE) for partitioning the digraph $\mathcal{G}$ of a *tree* constraint is the number of sink nodes of the reduced digraph of $\mathcal{G}$, and the maximum number of trees (MAXTREE) for partitioning the digraph $\mathcal{G}$ is the number of potential roots in $\mathcal{G}$. We are now in position to recall the filtering algorithm of the *tree* constraint. This algorithm is the first step of the extended constraint detailed in this paper. When NTREE has to reach MAXTREE, the algorithm enforces, for each potential root, a loop on itself (that represents the fact that it is a root). In the case where NTREE has to reach MINTREE, the

[2] An *integer variable* V is a variable ranging over a finite set of integers denoted by $dom(V)$; $\min(V)$ and $\max(V)$, respectively, denote the minimum and maximum values of $dom(V)$.

algorithm removes, for each potential root that does not belong to a sink strongly connected component $\mathcal{G}$, the loop on itself. Finally, for any `NTREE`, the main filtering rule associated with the constraint is based on the detection of dominator nodes of the digraph. Then, the filtering algorithm has to detect each node j of $\mathcal{G}$ such that there exists a node i for which j dominates all the potential roots of $\mathcal{G}$ according to i. The infeasible arcs in $\mathcal{G}$ for a *tree* constraint are the outgoing arcs (j, k), where j is a dominator node, such that there is no path from i to a potential root of $\mathcal{G}$ using the arc (j, k).

3.2 Modeling the Extended *tree* Constraint

In order to extend the original *tree* constraint according to the additional restrictions proposed in the introduction, we have to represent each restriction in the context of graphs. The restrictions involved in the extended version of the constraint can be classified in three distinct categories:

– Restrictions related to admissible arcs in $\mathcal{G}$: this is the case for the precedence and incomparability constraints, which respectively enforce a node to precede another one in any admissible tree partition, and enforce two nodes to be located on distinct paths in any admissible tree partition.
– Restrictions related to the accessibility of each node in $\mathcal{G}$: this is the case for the degree constraints, which restrict the in-degree of each node in any admissible tree partition.
– Restrictions related to the number of trees allowed to cover $\mathcal{G}$: this is the case for the constraint on the number of proper trees in any admissible tree partition.

From this classification, the following natural representation of each restriction emerges: the precedence and incomparability constraints can be modeled by a digraph $\mathcal{G}_{prec}$ and a graph $\mathcal{G}_{inc}$ respectively, the degree constraints can be modelled by associating to each node i of the given digraph $\mathcal{G}$ an integer variable $\mathtt{D}_i$ representing the in-degree of i, and the constraints on the numbers of trees and proper trees can be modeled by integer variables `NTREE` and `NPROP` respectively. Before upgrading Definition 14 accordingly, we need to introduce some notions.

Definition 15 (Required digraph and possible digraph). *Given an extended* tree *constraint and its digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$:*

– *An arc (i, j) of $\mathcal{G}$ is an* R-arc *(a* required arc*) if i has only j as successor in $\mathcal{G}$; otherwise (i, j) is a* P-arc *(a* possible arc*).*
– *A node i is an* R-succ *if all its outgoing arcs are R-arcs; otherwise i is a* P-succ*.*
– *A node i is an* R-pred *if all its incoming arcs are R-arcs; otherwise i is a* P-pred*.*
– *The* required digraph $\mathcal{G}_{req}$ *contains all arcs that must be in the partition. Formally, $\mathcal{G}_{req} = (\mathcal{V}, \mathcal{E}_{req})$, where $\mathcal{E}_{req}$ is the set of all R-arcs in $\mathcal{G}$.*
– *The* possible digraph $\mathcal{G}_{pos}$ *contains all arcs that may be in the partition. Formally, $\mathcal{G}_{pos} = (\mathcal{V}_{pos}, \mathcal{E}_{pos})$, where $\mathcal{V}_{pos}$ contains all the nodes that are incident to at least one P-arc, and $\mathcal{E}_{pos}$ is the set of all P-arcs in $\mathcal{G}$.*

Definition 15 ensures that $\mathcal{G}_{req}$ and $\mathcal{G}_{pos}$ completely define the digraph $\mathcal{G}$. We are now in position to define an admissible solution for an extended *tree* constraint.

Definition 16 (Solution of an extended *tree* constraint). *A ground instance of an extended* tree *constraint is said to be a* solution *if and only if:*

- *The digraph $\mathcal{G}_{req}$ consists of* `NTREE` *connected components.*
- *$\mathcal{G}_{req}$ contains* `NPROP` *connected components involving at least two nodes.*
- *Each connected component of $\mathcal{G}_{req}$ has no cyle involving more than one node (notice that each component contains exactly one node that has a self-loop and that corresponds to the root of that tree).*
- *For each arc (i, j) in $\mathcal{G}_{prec}$, there exists a path in $\mathcal{G}_{req}$ from node i to node j.*
- *For each edge (i, j) in $\mathcal{G}_{inc}$, there exists neither a path in $\mathcal{G}_{req}$ from node i to node j, nor a path in $\mathcal{G}_{req}$ from node j to node i.*
- *Each node i of $\mathcal{G}_{req}$ has exactly $\mathtt{D}_i$ predecessors in $\mathcal{G}_{req}$ that are distinct from i.*

3.3 Representing the Extended *tree* Constraint in Practice

We could provide a trivial representation of the extended *tree* constraint by:

$$\textit{tree}(\mathtt{NTREE}, \mathtt{NPROP}, \mathcal{G}, \mathcal{G}_{prec}, \mathcal{G}_{inc}, \mathtt{D}, \mathtt{F})$$

This representation is based on seven parameters respectively representing the numbers of trees and proper trees by two integer variables `NTREE` and `NPROP`, the digraphs $\mathcal{G}$ and $\mathcal{G}_{prec}$ by two $n \times n$ adjacency matrices, the graph $\mathcal{G}_{inc}$ by another $n \times n$ adjacency matrix, the degree restrictions by a vector `D` of n integer variables whose i^{th} element gives the in-degree of node i of $\mathcal{G}$, and the tree partition by a vector `F` of n integer variables whose i^{th} element gives the successor (father) of node i of $\mathcal{G}$ in the tree partition. Such a representation approach generally leads to a proliferation of parameters because a parameter is added each time a new restriction is introduced (e.g., see the *cycle*, *cumulative*, and *diffn* constraints in CHIP [14], which respectively have 17, 12, and 10 parameters).

Since the three graphs $\mathcal{G}$, $\mathcal{G}_{prec}$, and $\mathcal{G}_{inc}$ involve exactly the same set of nodes $\mathcal{V}$, and since the vectors `D` and `F` are also over $\mathcal{V}$, a more elegant and compact representation of the extended *tree* constraint rather has the form:

$$\textit{tree}(\mathtt{NTREE}, \mathtt{NPROP}, \mathtt{NODE})$$

where `NTREE` and `NPROP` are two integer variables respectively representing the numbers of trees and proper trees in the forest, and `NODE` is a collection of n nodes $\mathtt{NODE}[1], \ldots, \mathtt{NODE}[n]$. Each node $v_i = \mathtt{NODE}[i]$ has the following attributes, which complete the representation of the constraint:

- `L` is a unique integer in $[1, n]$. It can be interpreted as the *label* of v_i.
- `F` is an integer variable whose domain consists of node labels, i.e. elements in $[1, n]$. It can be interpreted as the *unique successor* (or *father*) of v_i.
- `P` is a possibly empty set of node labels, i.e., integers in $[1, n]$. It can be interpreted as the set of *mandatory descendants* (or *precedences*) of v_i.
- `I` is a possibly empty set of node labels, i.e., integers in $[i + 1, n]$. It can be interpreted as the set of nodes that are *incomparable* with v_i.

– D is an integer variable in $[0, n-1]$. It can be interpreted as the *in-degree* of v_i. Notice that the in-degree constraint ignores the possible self-loop on the node v_i.

The aim of this representation of the constraint is to encapsulate the three original graphs $\mathcal{G}$, $\mathcal{G}_{prec}$, and $\mathcal{G}_{inc}$ in a compact and expressive way. We now show how to read off these graphs from the collection NODE of the extended *tree* constraint. However, in the rest of this paper, we reason directly on these graphs rather than on the NODE structure. First, we extract the digraph $\mathcal{G}$, i.e., the digraph to partition, from the NODE structure:

Definition 17 (Associated digraph). *The* associated digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *of a* tree *constraint is defined by:*

$$\mathcal{V} = \{v_i \mid i \in [1, n]\} \textit{ and } \mathcal{E} = \{(v_i, v_j) \mid j \in dom(\texttt{NODE}[i].\texttt{F})\}$$

Note that filtering an inconsistent value j in $dom(\texttt{NODE}[i].\texttt{F})$ is equivalent to removing the arc (i, j) from $\mathcal{G}_{pos}$.

Next, we extract the digraph $\mathcal{G}_{prec}$,which represents the precedence constraints, from the NODE structure:

Definition 18 (Precedence digraph). *Given a* tree *constraint with associated digraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$*, the* precedence digraph $\mathcal{G}_{prec}$ *corresponds to the following transitive reduction:*

$$TR(\mathcal{V}, \{(i, j) \in \mathcal{V}^2 \mid i \in \texttt{NODE}[j].\texttt{P}\})$$

Finally, we extract the graph $\mathcal{G}_{inc}$, which represents the incomparability constraints, from the NODE structure:

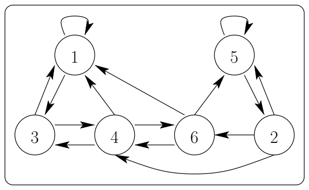
Definition 19 (Incomparability graph). *The* incomparability graph $\mathcal{G}_{inc}$ *of a* tree *constraint with associated digraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *is defined by:*

$$(\mathcal{V}, \{(i, j) \in \mathcal{V}^2 \mid i \in \texttt{NODE}[j].\texttt{I}\})$$

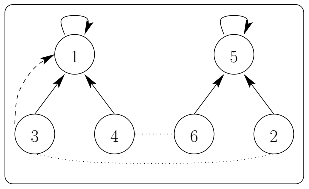
3.4 Example of an Extended *tree* Constraint

Figure 1a and the first three columns of Table 1 represent a directed graph $\mathcal{G}$ for which the four solutions in Figures 1b to 1e are obtained from the *tree* constraints in the columns of Parts (b) to (e), respectively, of Table 1:

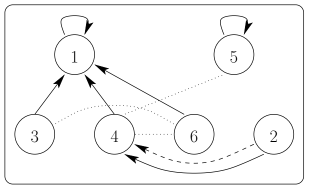
– Part (b): $dom(\texttt{NTREE}) = \{2\} = dom(\texttt{NPROP})$, $dom(\texttt{NODE}[i].\texttt{D}) = [0, 2]$ for each node v_i of $\mathcal{G}$, the node v_3 has to precede node v_1, and the node pairs (v_2, v_3) and (v_4, v_6) are incomparable.
– Part (c): $dom(\texttt{NTREE}) = \{2\}$, $dom(\texttt{NPROP}) = \{1\}$, $dom(\texttt{NODE}[i].\texttt{D}) = [0, 3]$ for each node v_i of $\mathcal{G}$, the node v_2 has to precede node v_4, and the node pairs (v_3, v_6), (v_4, v_5), and (v_4, v_6) are incomparable.
– Part (d): $dom(\texttt{NTREE}) = \{2\}$, $dom(\texttt{NPROP}) = \{1\}$, $dom(\texttt{NODE}[i].\texttt{D}) = [0, 1]$ for each node v_i of $\mathcal{G}$, the arcs (v_2, v_3), (v_2, v_4), (v_2, v_6), (v_3, v_1), and (v_6, v_1) represent precedence constraints restricting the digraph $\mathcal{G}$, and the node pairs (v_3, v_5) and (v_4, v_5) are incomparable.



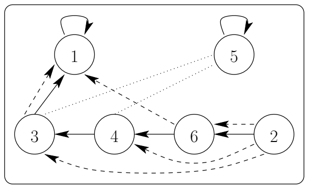
(a) The original digraph $\mathcal{G}$.



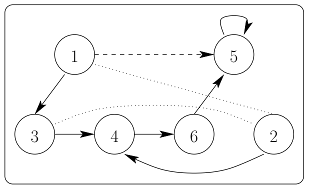
(b) $\mathtt{D} \leq 2$, $\mathtt{NTREE} = 2$, $\mathtt{NPROP} = 2$.



(c) $\mathtt{D} \leq 3$, $\mathtt{NTREE} = 2$, $\mathtt{NPROP} = 1$.



(d) $\mathtt{D} \leq 1$, $\mathtt{NTREE} = 2$, $\mathtt{NPROP} = 1$.



(e) $\mathtt{D} \leq 2$, $\mathtt{NTREE} = 1$, $\mathtt{NPROP} = 1$.

Fig. 1: (a) The associated digraph $\mathcal{G}$ of the *tree* constraint detailed in Section 3.4. (b,c,d,e) Plain arcs depict four solutions according to the precedence, incomparability, and degree constraints described in Parts (b) to (e), respectively, of Table 1. Dashed arcs depict precedence constraints, while dotted edges represent incomparability constraints.

– Part (e): $dom(\mathtt{NTREE}) = \{1\} = dom(\mathtt{NPROP})$, $dom(\mathtt{NODE}[i].\mathtt{D}) = [0, 2]$ for each node v_i of $\mathcal{G}$, the node v_1 precedes node v_5, and the node pairs (v_1, v_2) and (v_2, v_3) are incomparable.

Moreover, in each digraph depicted by Figures 1a to 1e, the different kinds of arcs represent distinct constraints:

– The plain arcs depict the arcs of the digraph to partition, i.e., the digraph $\mathcal{G}$.
– The dashed arcs depict the precedence constraints. In other words, the digraph induced by dashed arcs represents the precedence digraph $\mathcal{G}_{prec}$.
– The dotted edges depict the incomparability constraints. This means that the graph induced by dotted edges represents the incomparability graph $\mathcal{G}_{inc}$.

The rest of this paper does not directly deal with degree constraints or the NPROP variable. However, in practice, the degree constraints are globally maintained by a specialised global cardinality constraint, called *global cardinality no loop* in [2] and discussed in [5, 4, 22]. Similarly, the propagation related to the NPROP integer variable is

v_i	NODE$[i]$.L	NODE$[i]$.F	Part (b)			Part (c)		
			NODE$[i]$.P	NODE$[i]$.I	NODE$[i]$.D	NODE$[i]$.P	NODE$[i]$.I	NODE$[i]$.D
v_1	1	$\{1,3\}$	-	-	$[0,2]$	-	-	$[0,3]$
v_2	2	$\{4,5,6\}$	-	$\{3\}$	$[0,2]$	$\{4\}$	-	$[0,3]$
v_3	3	$\{1,4\}$	$\{1\}$	-	$[0,2]$	-	$\{6\}$	$[0,3]$
v_4	4	$\{1,3,6\}$	-	$\{6\}$	$[0,2]$	-	$\{5,6\}$	$[0,3]$
v_5	5	$\{2,5\}$	-	-	$[0,2]$	-	-	$[0,3]$
v_6	6	$\{1,4,5\}$	-	-	$[0,2]$	-	-	$[0,3]$

v_i	NODE$[i]$.L	NODE$[i]$.F	Part (d)			Part (e)		
			NODE$[i]$.P	NODE$[i]$.I	NODE$[i]$.D	NODE$[i]$.P	NODE$[i]$.I	NODE$[i]$.D
v_1	1	$\{1,3\}$	-	-	$[0,1]$	$\{5\}$	$\{2\}$	$[0,2]$
v_2	2	$\{4,5,6\}$	$\{3,4,6\}$	-	$[0,1]$	-	$\{3\}$	$[0,2]$
v_3	3	$\{1,4\}$	$\{1\}$	$\{5\}$	$[0,1]$	-	-	$[0,2]$
v_4	4	$\{1,3,6\}$	-	$\{5\}$	$[0,1]$	-	-	$[0,2]$
v_5	5	$\{2,5\}$	-	-	$[0,1]$	-	-	$[0,2]$
v_6	6	$\{1,4,5\}$	$\{1\}$	-	$[0,1]$	-	-	$[0,2]$

Table 1: The first three columns describe the associated digraph $\mathcal{G}$ of the *tree* constraint depicted by Figure 1a. The columns of Parts (b) to (e) show the precedence, incomparability, and degree constraints associated with the digraph $\mathcal{G}$ that lead to the *tree* partitions depicted by Figures 1b to 1e, respectively.

discussed in [4, 22]. First, in the following, Section 4 details the theoretical complexity of each part composing the extended *tree* constraint. Then, Section 5.1 provides necessary conditions for partitioning the digraph $\mathcal{G}$ of a *tree* constraint into trees according to a potential number of trees and a set of precedence constraints. Since achieving arc consistency for even this simplified constraint is already NP-hard, some pruning rules are derived from these necessary conditions. Next, Section 5.2 proposes necessary conditions for partitioning $\mathcal{G}$ into trees according to a potential number of trees and a set incomparability constraints. Section 6.1 completes those results by considering the interaction between precedence and incomparability constraints, and new necessary conditions are introduced to ensure that there exists at least one tree partitioning of $\mathcal{G}$ according to the precedence and incomparability constraints. Finally, Section 6.2 improves the necessary conditions introduced in Sections 5.1 to 6.1 by deriving new precedence constraints from the interaction between the digraph $\mathcal{G}$ and the existing precedence and incomparability constraints.

4 Theoretical Complexity of the Extended *tree* Constraint

This section discusses the theoretical complexity of each restriction involved in the extended *tree* constraint. Particularly, we point out that the propagation of the generalised arc consistency in an extended *tree* constraint is an NP-hard problem. However, notice that in the case of an extended *tree* constraint involving only incomparability constraints, we do not classify the theoretical complexity. Indeed, to the best of our knowledge, the complexity of finding a tree partition according to a set of incomparability constraints between nodes is an open question.

4.1 The *tree* Constraint Is Only Constrained by the `NPROP` Variable

We study the theoretical complexity of propagating an extended *tree* constraint to generalised arc consistency, in the context where only the `NPROP` variable is constrained. In other words, the extended *tree* constraint has the following characteristics:

- $dom(\texttt{NPROP}) \subseteq dom(\texttt{NTREE})$;
- There does not exist any precedence constraint between the nodes of $\mathcal{G}$;
- There does not exist any incomparability constraint between the nodes of $\mathcal{G}$;
- The in-degree variables associated with each node of $\mathcal{G}$ have domains included in $[0, n]$.

In this context, we show that the extended *tree* constraint cannot be propagated to reach generalised arc consistency in polynomial time.

Theorem 1. *Propagating generalised arc consistency for the extended* tree *constraint according to* `NPROP` *is NP-hard.*

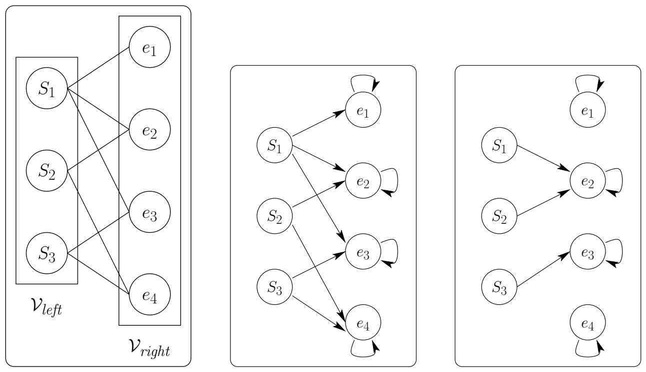
Proof. First, we give a polynomial certificate for the extended *tree* constraint according to the number of proper trees `NPROP`, i.e., a deterministic polynomial-time algorithm that checks if a ground instance of an extended *tree* constraint is a solution satisfying the constraint. We start by checking that the ground instance is a forest containing exactly `NTREE` trees, i.e., the digraph $\mathcal{G}_{req}$ (1) consists of `NTREE` connected components and (2) does not contain any cycles involving more than one node. Next, we check that exactly `NPROP` trees of the forest contains at least two nodes.

Second, we show that any instance of the NP-complete hitting set problem[3] [16] can be polynomially reformulated as an extended *tree* constraint. A classical graph-based representation of the hitting set problem uses a bipartite graph $\mathcal{B} = (\mathcal{V}_{left}, \mathcal{V}_{right}, \mathcal{E})$ defined in the following way (see Figure 2):

- A node of $\mathcal{V}_{left}$ is associated with each subset of $\mathcal{C}$;
- A node of $\mathcal{V}_{right}$ is associated with each element of $\mathcal{S}$;
- There exist an edge in $\mathcal{E}$ between a node of $\mathcal{V}_{left}$ and a node of $\mathcal{V}_{right}$ iff the corresponding element with the node of $\mathcal{V}_{right}$ belongs to a subset associated with a node of $\mathcal{V}_{left}$.

A solution satisfying the hitting set problem is an assignment of each node of $\mathcal{V}_{left}$ to a node of $\mathcal{V}_{right}$ such that at most k nodes of $\mathcal{V}_{right}$ are used to assign all $\mathcal{V}_{left}$ nodes. A polynomial-time transformation of the hitting set problem into an extended *tree* constraint is based on a digraph $\mathcal{G}$ defined by $(\mathcal{V}_{left} \cup \mathcal{V}_{right}, \{(i,j) \in \mathcal{V}_{left} \times \mathcal{V}_{right} \mid i = j \vee (i,j) \in \mathcal{E}\})$. Then, the hitting set problem is reduced to finding a tree partition of $\mathcal{G}$ in at most k proper trees, i.e., satisfying an extended *tree* constraint with $\texttt{NPROP} \leq k$. □

[3] Given a collection of subsets $\mathcal{C}$ of a set $\mathcal{S}$ and an integer k, the hitting set problem checks if there exists a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $|\mathcal{S}'| \leq k$ and $\mathcal{S}'$ contains at least one element of each subset in $\mathcal{C}$.



(a) Bipartite graph $\mathcal{B}$. (b) Digraph $\mathcal{G}$. (c) A solution.

Fig. 2: A hitting set problem represented by an extended *tree* constraint, with $\texttt{NPROP} = 2$ and $|\mathcal{V}_{left}| = |\mathcal{C}|$.

4.2 The *tree* Constraint Only Involves Precedence Constraints

We study the theoretical complexity of propagating an extended *tree* constraint to generalised arc consistency, when only precedence constraints are represented. In other words, we have the following assumptions:

- $dom(\texttt{NTREE}) \subset dom(\texttt{NPROP})$;
- There exist precedence constraints between the nodes of $\mathcal{G}$;
- There does not exist any incomparability constraint between the nodes of $\mathcal{G}$;
- The in-degree variables associated with each node of $\mathcal{G}$ have domains included in $[0, n]$.

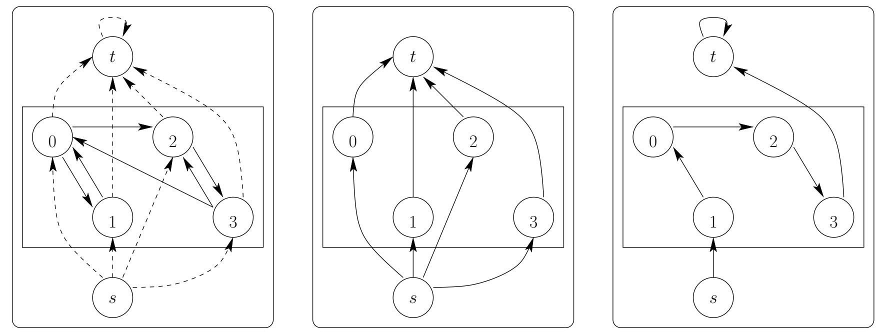
In this context, we show that the extended *tree* constraint cannot be propagated to reach generalised arc consistency in polynomial time.

Theorem 2. *Propagating generalised arc consistency for the extended* tree *constraint according to precedence constraints is NP-hard.*

Proof. First, we give a polynomial certificate for the extended *tree* constraint according to precedence constraints, i.e., a deterministic polynomial-time algorithm that checks if a ground instance of an extended *tree* constraint with precedence constraints is a solution. We start by checking that the ground instance is a forest containing exactly NTREE trees, i.e., the digraph $\mathcal{G}_{req}$ (1) consists of NTREE connected components and (2) does not contain any cycles involving more than one node. Next, for each arc (u, v) of $\mathcal{G}_{prec}$, we check if there is a path from u to v in $\mathcal{G}_{req}$; this can be done by a depth-first search from u in $\mathcal{G}_{req}$.

Second, we show that any instance of the NP-complete Hamiltonian path problem[4] [16] for a digraph $(\mathcal{V}, \mathcal{E})$ can be polynomially reformulated as a *tree* constraint.

[4] Given a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the *Hamiltonian path problem* checks if there exists an elementary path in $\mathcal{G}$ containing all the nodes of $\mathcal{V}$.



(a) The digraph $\mathcal{G}$ of the *tree* constraint. (b) The precedence digraph $\mathcal{G}_{prec}$ of the *tree* constraint. (c) A solution.

Fig. 3: Reduction of the Hamiltonian path problem to an extended *tree* constraint, with $\texttt{NTREE} = 1$. Figure 3a depicts the digraph $\mathcal{G}$ of the constraint. The nodes involved in the rectangle depicts the digraph $\mathcal{H}$ for which an Hamiltonian path is searched. Figure 3b depicts the precedence digraph $\mathcal{G}_{prec}$ of the constraint. Finally, Figure 3c provides a solution of the constraint.

For this purpose, we have to define the digraph $\mathcal{G}$ and the precedence digraph $\mathcal{G}_{prec}$ (see Figure 3). Let $\mathcal{G} = (\mathcal{V} \cup \{s, t\}, \mathcal{E}')$ with $\mathcal{E}'$ being a superset of $\mathcal{E}$ such that s is the only source node in $\mathcal{G}$, s is a predecessor of all the nodes of $\mathcal{V}$, t is the only sink node of $\mathcal{G}$ with a self-loop, and t is a successor of all the nodes of $\mathcal{V}$. Let $\mathcal{G}_{prec} = (\mathcal{V} \cup \{s, t\}, \{(s, j) \mid j \in \mathcal{V}\} \cup \{(i, t) \mid i \in \mathcal{V}\}$. The Hamiltonian path problem then consists of finding a tree partition of $\mathcal{G}$ with one tree (i.e., $\texttt{NTREE} = 1$) according to the set of precedence constraints defined by $\mathcal{G}_{prec}$. $\square$

4.3 The *tree* Constraint Only Involves In-Degree Constraints

We now study the theoretical complexity of propagating an extended *tree* constraint to generalised arc consistency, in the context where only the in-degree variables, associated with each node of the digraph $\mathcal{G}$, are constrained. In other words, the extended *tree* constraint has the following characteristics:

- $dom(\texttt{NTREE}) \subset dom(\texttt{NPROP})$;
- There does not exist any precedence constraint between the nodes of $\mathcal{G}$;
- There does not exist any incomparability constraint between the nodes of $\mathcal{G}$;
- Some in-degree variables associated with some nodes of $\mathcal{G}$ have domains strictly included in $[0, n]$.

The extended *tree* constraint cannot be propagated to reach generalised arc consistency in polynomial time. The reduction from the Hamiltonian path problem is straightforward.

5 Precedence and Incomparability Constraints in a Tree Partitioning

This section details how to handle the precedence and incomparability constraints respectively provided by the `P` and `I` attributes of the `NODE` collection. Since achieving arc consistency for even this simplified constraint is already NP-hard (and the complexity is not known in the case of incomparabilities), necessary conditions are proposed for each one and filtering rules are derived from these conditions.

5.1 Combining Tree Partitioning and Precedence Constraints

From Definition 18 of Section 3.3, which proposes a digraph model of precedence constraints (via a digraph $\mathcal{G}_{prec}$), this section studies the precedence restriction by, first, providing an upper bound on `NTREE`, next, considering necessary conditions related to the existence of a solution, and finally, introducing filtering derived from these conditions.

Definition 20 (Contracted precedence digraph). *Given a* tree *constraint with digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of precedence constraints represented in the directed acyclic graph (dag) $\mathcal{G}_{prec}$, the* contracted precedence digraph $\mathcal{G}^H_{prec}$ *of $\mathcal{G}$ is defined by:*

- *There is a node of $\mathcal{G}^H_{prec}$ for each connected component of the required digraph $\mathcal{G}_{req}$.*
- *There is an arc between two nodes of $\mathcal{G}^H_{prec}$ iff there is an arc in $\mathcal{G}_{prec}$ between two nodes of the corresponding connected components of $\mathcal{G}_{req}$.*

Without loss of generality, we assume that $\mathcal{G}_{prec}$ does not contain any self-loops, i.e., that a node of $\mathcal{G}_{prec}$ cannot precede itself. Moreover, observe that any non-loop required arc of $\mathcal{G}_{req}$ corresponds to a precedence constraint. This leads us to maintaining the following two invariants:

$$\forall (u,v) \in \mathcal{E}_{req} : u \neq v \Rightarrow (u,v) \in \mathcal{E}_{prec} \quad (1)$$

$$\mathcal{G}_{req} \setminus \{(u,v) \in \mathcal{E}_{req} \mid u = v\} \subseteq \mathcal{G}_{prec} \quad (2)$$

They state that the required digraph without its self-loops must always be a sub-graph of the precedence digraph.

Upper Bound on the Number of Trees According to Precedence Constraints: We now provide an upper bound on the number of trees that partition the digraph $\mathcal{G}$ of a *tree* constraint, according to the precedence digraph $\mathcal{G}_{prec}$. A first upper bound, given in [3] for the “pure” *tree* constraint (i.e., the *tree* constraint without any side constraints) and denoted by `MAXTREE`, is the number of potential roots of $\mathcal{G}$. Since this bound does not consider the precedence constraints, we now provide a tighter bound that considers $\mathcal{G}_{prec}$ as well. The basic idea is to count among the connected components of $\mathcal{G}_{prec}$ those that are necessarily connected to another one. For this purpose, a $\{0,1\}$-value out_i is associated with each connected component $CC(i)$ of $\mathcal{G}_{prec}$, depending on whether there is a node in $CC(i)$ whose potential father nodes are *all* outside $CC(i)$:

$$out_i = \begin{cases} 1 & \text{if } \exists u \in CC(i) : \forall v \in dom(\texttt{NODE}[u].\texttt{F}) : v \notin CC(i) \\ 0 & \text{otherwise} \end{cases}$$

Thus, $CC(i)$ will have to be merged with another connected component when $out_i = 1$.

Proposition 1. *Given a* tree *constraint and its precedence digraph $\mathcal{G}_{prec}$ such that $\mathcal{G}_{prec}$ contains k connected components, an upper bound on* NTREE *is:*

$$\texttt{MAXTREE}_{prec} = k - \sum_{i=1}^{k} out_i$$

Proof. By the definition of out_i. □

Proposition 2. $\texttt{MAXTREE}_{prec}$ *is tighter than* MAXTREE*.*

Proof. Let $p = \texttt{MAXTREE}$ be the number of potential roots of $\mathcal{G}$. We know that:

$$k - p \leq \sum_{i=1}^{k} out_i \leq k$$

Thus:

$$0 \leq k - \sum_{i=1}^{k} out_i \leq p$$

Hence, $0 \leq \texttt{MAXTREE}_{prec} \leq \texttt{MAXTREE}$. □

Filtering a *tree* Constraint According to Precedence Constraints: A necessary condition on the existence of solutions of a *tree* constraint involving precedence constraints is made of four conjuncts, each maintaining a property of the sought tree partition: first, we have to maintain a compatible number of trees allowed to cover the digraph; second, we have to ensure that the partition is cycle-free; third, we have to ensure that the partition is compatible with all the precedence constraints; finally, we have to ensure that each tree is rooted on a potential root.

Theorem 3. *If there is a solution to an extended* tree *constraint with precedences, then the following conditions hold:*

1. ***Number of Trees:*** $dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}_{prec}] \neq \emptyset$*, where* MINTREE *is the number of sink components in $\mathcal{G}$ and $\texttt{MAXTREE}_{prec}$ is the quantity introduced by Proposition 1.*
2. ***Cycle-free:*** *$\mathcal{G}_{prec}$ has no cycles.*
3. ***Compatibility:*** *The transitive closure of $\mathcal{G}_{prec}$ is included within the transitive closure of $\mathcal{G}$ (i.e., $TC(\mathcal{G}_{prec}) \subseteq TC(\mathcal{G})$).*

4. ***Compatible Root:*** *For each sink component $\mathcal{S}$ of $\mathcal{G}$, at least one node is both a node with a self-loop in $\mathcal{G}$ and a sink in $\mathcal{G}_{prec}$.*

Proof. A proof is provided for each condition:

1. If $dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}_{prec}] = \emptyset$, then $\max(\texttt{NTREE}) < \texttt{MINTREE}$ or $\min(\texttt{NTREE}) > \texttt{MAXTREE}_{prec}$. If the first inequality holds, then there exists a sink component of $\mathcal{G}$ that does not contain any potential root; also, we know that there is no path between two nodes of distinct sink components of $\mathcal{G}$. Naturally, Proposition 1 ensures that if the second inequality holds, then the *tree* constraint has no solution.
2. Assume $\mathcal{G}_{prec}$ contains a cycle and there is a solution. Then there exist paths $P = \langle u_1, \ldots, u_k \rangle$ and $P' = \langle u_k, \ldots, u_1 \rangle$ in $\mathcal{G}_{prec}$. The path P enforces that node u_1 precedes node u_k in any solution. Similarly, the path P' enforces that u_k precedes u_1 in any solution. But then there is no solution satisfying at the same time P and P': a contradiction.
3. Assume $TC(\mathcal{G}_{prec}) \not\subseteq TC(\mathcal{G})$. Then there exists at least one arc (u, v) in $TC(\mathcal{G}_{prec})$ that is not in $TC(\mathcal{G})$. This means there exists at least one precedence constraint that cannot be satisfied.
4. Assume on the contrary that there exists a sink component S such that each node r with a self-loop (i.e., node r is a potential root) has at least one successor in $\mathcal{G}_{prec}$. Since each sink component contains at least one potential root (see the necessary and sufficient condition for the pure *tree* constraint [3]), there is a contradiction with the fact that each node r has at least one successor in $\mathcal{G}_{prec}$.

Each condition is necessary, hence their conjunction is necessary. □

Based on the necessary condition expressed by Theorem 3, Algorithm 1 filters a *tree* constraint according to its precedence constraints. Algorithm 1 begins with an initialisation (STEP 1) that computes the connected components of $\mathcal{G}_{req}$ and marks the single sink node of each component (by definition of $\mathcal{G}_{req}$, each connected component of $\mathcal{G}_{req}$ is a tree). This can be done in $O(n + m)$ time. Next, STEP 2 is a normalisation step that updates the precedence digraph $\mathcal{G}_{prec}$ as follows: for each non-root node u of a connected component $CC(u)$ of $\mathcal{G}_{req}$, any precedence constraint (u, v) is replaced by $(r(u), v)$ in $\mathcal{G}_{prec}$, where $r(u)$ is the root of $CC(u)$ (indeed, observe that every path from u to v visits $r(u)$ before reaching v). This can be done in $O(m)$ time by iterating through all precedences. Next, STEP 3 checks the feasibility of the constraint according to the precedence constraints (Theorem 3); this can be done in $O(nm)$ time by computing the transitive closure of $\mathcal{G}_{prec}$ and $\mathcal{G}$. STEP 4 updates the domain of $\texttt{NTREE}$ according to the lower bound $\texttt{MINTREE}$, defined in [3], and the upper bound $\texttt{MAXTREE}_{prec}$, given in Proposition 1; this can be done in $O(n+m)$ time. Next, STEP 5 checks for each arc (u, v) of $\mathcal{G}_{pos}$ its compatibility (i.e., whether it does not create any cycle, is not transitive, and can reach a potential root) with the contracted precedence digraph $\mathcal{G}_{prec}^{H}$; this can be done in $O(m)$ time by injecting the arc $(CC(u), CC(v))$ in the depth-first-search tree of $\mathcal{G}_{prec}^{H}$. Finally, STEP 6 updates the normal form (i.e., the transitive reduction) of $\mathcal{G}_{prec}$ according to $\mathcal{G}_{true}$, which can be done in $O(nm)$ time.

Algorithm 1 Filtering of the *tree* constraint according to precedence constraints.

```
 /* STEP 1: Initialisation */
 1. foreach node $u \in \mathcal{V}$ do
 2.  $CC(u) \leftarrow$ maximum connected component of $\mathcal{G}_{req}$ containing $u$;
 3.  $r(u) \leftarrow$ the single sink node of $CC(u)$;
 /* STEP 2: Normalisation of the precedence digraph */
 4. foreach arc $(u,v) \in \mathcal{G}_{prec}$ such that $u \neq r(u)$ and $v \notin CC(u)$ do
 5.  replace $(u,v)$ in $\mathcal{G}_{prec}$ by $(r(u),v)$;
 /* STEP 3: Checking feasibility */
 6. if the tree constraint has no solution (see Theorem 3) then
 7.  report failure and exit;
 /* STEP 4: Updating the domain of NTREE */
 8. $dom(\texttt{NTREE}) \leftarrow dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}_{prec}]$;
 /* STEP 5: Maintaining the cycle-free, compatibility, and
            compatible-root conditions*/
 9. foreach arc $(u,v)$ of $\mathcal{G}_{pos}$ do
10.  remove $(u,v)$ from $\mathcal{G}_{pos}$ if one of the following holds:
11.    a. $(CC(u),CC(v))$ would create a cycle in $\mathcal{G}^H_{prec}$;
12.    b. $(CC(u),CC(v))$ would be a transitive arc in $\mathcal{G}^H_{prec}$;
13.    c. $u = v$ and $CC(u)$ is not a sink of $\mathcal{G}^H_{prec}$;
 /* STEP 6: Re-normalisation of the precedence digraph */
14. $\mathcal{G}_{prec} \leftarrow TR(\mathcal{G}_{prec} \cup \mathcal{G}_{req})$;
```

Lemma 1. *Algorithm 1 never removes an arc from $\mathcal{G}_{pos}$ or a value from $dom(\texttt{NTREE})$ that belongs to a solution to a* tree *constraint.*

Proof. If a value k of $dom(\texttt{NTREE})$ is removed (STEP 4) but there exists a partition of $\mathcal{G}$ in k trees satisfying the *tree* constraint, then MINTREE is not a lower bound on NTREE or $\texttt{MAXTREE}_{prec}$ is not an upper bound on NTREE. This is impossible because of Condition 1 of Theorem 3.

If an arc (u,v) of $\mathcal{G}_{pos}$ is removed but there exists a partition of $\mathcal{G}$ that contains it, then this is only due to STEP 5. Conditions a, b, c of this step are respectively derived from Conditions 2, 3, 4 of Theorem 3. However, in Algorithm 1, we only consider the digraph $\mathcal{G}^H_{prec}$ instead of $\mathcal{G}_{prec}$ that allows us to ignore connected components of $\mathcal{G}_{req}$, i.e., the pieces of trees already built. □

5.2 Combining Tree Partitioning and Incomparability Constraints

From Definition 19 of Section 3.3, which proposes a graph model of incomparability constraints (via a graph $\mathcal{G}_{inc}$), this section studies the incomparability restriction by, first, considering necessary conditions related to the existence of a solution, and next, introducing filtering rules derived from these conditions.

A necessary condition on the existence of solutions of a *tree* constraint involving incomparability constraints is made of two conjuncts: first, we have to ensure that the

Algorithm 2 Filtering of the *tree* constraint according to incomparability constraints.

```
1. if the tree constraint has no solution (see Theorem 4) then
2.     report failure and exit;
3. foreach arc $e \in \mathcal{G}_{pos} \cap \mathcal{G}_{inc}$ do remove $e$ from $\mathcal{G}_{pos}$;
```

partition violates none of the incomparability constraints involved in the incomparability graph $\mathcal{G}_{inc}$; second, we have to maintain the reachability of a potential root for each non-potential-root node. First, let $inc(u)$ denote the set of nodes of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that are incomparable with node u. In other words, $inc(u)$ denotes the neighbors of u in $\mathcal{G}_{inc}$. Next, we can formally introduce the necessary condition:

Theorem 4. *If there is a solution to an extended* tree *constraint with incomparabilities then, the following conditions hold:*

1. ***Compatibility:*** $TC(\mathcal{G}_{req}) \cap \mathcal{G}_{inc}$ *has no arcs.*
2. ***Reachability:*** *For each node u of $\mathcal{G}$, there is at least one path reaching a potential root of $\mathcal{G}_{pos}$ that does not contains any node of $inc(u)$.*

Proof. A proof is provided for each condition:

1. Assume there exists a path from a node u to a node v in $\mathcal{G}_{req}$. If u and v are incomparable, then the *tree* constraint cannot be satisfied.
2. Assume there exists a node u of $\mathcal{G}$ such that for any node r with a self-loop and any path P from u to r, there is a node v in P such that u and v are incomparable. Then the *tree* constraint cannot be satisfied since u cannot reach a potential root without violating an incomparability constraint.

Each condition is necessary, hence their conjunction is necessary. □

Based on the necessary condition expressed by Theorem 4, Algorithm 2 first (lines 1 and 2) checks the feasibility of the *tree* constraint according to its incomparability constraints (Theorem 4); this can be achieved in $O(nm)$ time due to the computation of the transitive closure of $\mathcal{G}_{req}$ and depth-first search for each node u in the digraph $\mathcal{G} \setminus inc(u)$. Next, line 3 detects infeasible arcs of $\mathcal{G}_{pos}$ that belong to $\mathcal{G}_{inc}$; this can be done in $O(m)$ time.

Lemma 2. *Algorithm 2 never removes an arc from $\mathcal{G}_{pos}$ that belongs to a solution to a* tree *constraint.*

Proof. Assume two nodes u and v are incomparable and the arc (u, v) of $\mathcal{G}_{pos}$ is added to $\mathcal{G}_{req}$: this is a contradiction because the incomparability constraint between u and v is violated. □

6 Managing Interaction Between Precedence and Incomparability

This section shows how to improve the treatment of the precedence and incomparability constraints introduced in Section 5. This improvement is based on two aspects of these restrictions. The first one reasons on the information provided by the interaction between precedence and incomparability constraints, while the second one shows how to derive new precedence constraints hidden in the interaction between the existing precedence and incomparability constraints with the digraph $\mathcal{G}$ of the extended *tree* constraint.

6.1 Combining Precedence and Incomparability Constraints

We now study the interaction between precedence and incomparability constraints in order to provide two necessary conditions as well as a filtering algorithm directly derived from these conditions.

A necessary condition on the existence of solutions of a *tree* constraint combining precedence and incomparability constraints is now exhibited. It is made of two conjuncts: first, we have to ensure that the partition satisfies all the precedence constraints (involved in $\mathcal{G}_{prec}$) and violates none of the incomparability constraints (involved in $\mathcal{G}_{inc}$); second, we have to maintain the reachability of a potential root for each non potential-root node.

Theorem 5. *If there is a solution to a* tree *constraint involving both precedence and incomparability constraints, then the following conditions hold:*

1. ***Compatibility:*** $TC(\mathcal{G}_{prec}) \cap \mathcal{G}_{inc} = \emptyset$.
2. ***Reachability:*** *For each edge* (u, v) *in* $\mathcal{G}_{inc}$*, there does not exist a node* w *such that the arcs* (w, u) *and* (w, v) *both belong to* $TC(\mathcal{G}_{prec})$.

Proof. A proof is provided for each condition:

1. This condition is directly derived from Condition 1 of Theorem 4. Indeed, any incomparability constraint between two nodes of $\mathcal{G}_{prec}$ that belong to the same path in $\mathcal{G}_{prec}$ leads to a contradiction with the *tree* constraint.
2. Assume there exists a node w such that the arcs (w, u) and (w, v) belong to $TC(\mathcal{G}_{prec})$ and assume $(u, v) \in \mathcal{G}_{inc}$. The nodes u and v then belong to the same path in any solution and there is a contradiction with $(u, v) \in \mathcal{G}_{inc}$.

Each condition is necessary, hence their conjunction is necessary. □

Algorithm 3 first checks the feasibility of the *tree* constraint according to precedence and incomparability constraints (Theorem 5); this can be done in $O(nm)$ time due to the computation of the transitive closure of $\mathcal{G}_{prec}$ (assuming that computing the union and intersection of two graphs takes $O(m)$ time). Then, lines 3 to 6 take $O(m^2)$ time. Indeed, for each arc (u, v) of $\mathcal{G}_{pos}$, condition (a) checks the compatibility of (u, v) in the precedence digraph according to the incomparability graph. Condition (b) detects the arcs of $\mathcal{G}_{pos}$ that violate the reachability condition introduced by Theorem 5.

Algorithm 3 Filtering of the *tree* constraint according to precedence and incomparability constraints.

```
1. if the tree constraint has no solution (see Theorem 5) then
2.  report failure and exit;
3. foreach arc $(u,v) \in \mathcal{G}_{pos}$ do
4.  remove $(u,v)$ from $\mathcal{G}_{pos}$ if one of the following holds:
5.    a. $TC(\mathcal{G}_{prec} \cup \{(u,v)\}) \cap \mathcal{G}_{inc} \neq \emptyset$;
6.    b. $\exists (u,u_d),(v_a,v) \in TC(\mathcal{G}_{prec}) : (u_d,v_a) \in \mathcal{G}_{inc}$;
```

Lemma 3. *Algorithm 3 never removes an arc from $\mathcal{G}_{pos}$ that belongs to a solution to a* tree *constraint.*

Proof. Case a of Algorithm 3 is directly derived from Condition 1 of Theorem 5. Assume an arc (u,v) of $\mathcal{G}_{pos}$, such that $TC(\mathcal{G}_{prec} \cup \{(u,v)\}) \cap \mathcal{G}_{inc} = \emptyset$, is removed but there exists a solution to the *tree* constraint containing (u,v). Then there is a contradiction because the equality $TC(\mathcal{G}_{prec} \cup \{(u,v)\}) \cap \mathcal{G}_{inc} = \emptyset$ ensures that the arc (u,v) violates at least one incomparability constraint.

Case b of Algorithm 3 is intuitively derived from Condition 2 of Theorem 5. If such an arc (u,v) is added to $\mathcal{G}_{req}$, then node u has to precede at the same time nodes u_d and v; this means there exists a path in any solution satisfying the constraint, starting from node u, reaching first node v (because, by assumption, $(u,v) \in \mathcal{E}_{req}$) and next node u_d. Moreover, we know that node v_a precedes node v, hence, by transitivity, there exists a path from v_a to u_d in any solution satisfying the constraint: there is a contradiction with the incomparability constraint between nodes u_d and v_a. □

Theorem 6. *Algorithm 3 filters a* tree *constraint in $O(m^2)$ time.*

Proof. See the discussion above of the algorithm. □

6.2 Deriving New Precedence Constraints

Graph properties related to $\mathcal{G}$, $\mathcal{G}_{prec}$, and $\mathcal{G}_{inc}$ lead to the derivation of new precedence constraints that come from the strong interaction between the constraints induced by these three graphs.

Using the concept of dominator in digraphs (Definition 11), we propose a simple rule that reveals new precedence constraints. Given the associated digraph $\mathcal{G}$, let S be a strongly connected component of $\mathcal{G}$, and let u and v be two nodes of S such that (u,v) is an arc of $\mathcal{G}_{prec}$. For each dominator d of $DOM_{\langle S,u\rangle}(v)$, the arcs (u,d) and (d,v) are new precedence constraints because any path from u to v in $\mathcal{G}$ reaches node d before reaching node v. Then, adding (u,d) and (d,v) to $\mathcal{G}_{prec}$ leads the arc (u,v) to become transitive, thus (u,v) is removed from $\mathcal{G}_{prec}$. Lines 2 to 5 of Algorithm 4 detect such a pattern in $\mathcal{G}$ in $O(mn)$ time, whereas the dominators are computed in $O(n^2)$ time [13]. In practice, the dominator nodes in $\mathcal{G}$ are computed dynamically according to potential roots.

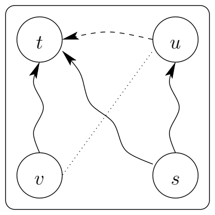


Fig. 4: The dashed precedence arc (u,t) can be added to $\mathcal{G}_{prec}$. The dotted edge depicts an incomparability constraint between the nodes u and v. The plain curly arcs depict paths in $\mathcal{G}_{prec}$.

Algorithm 4 Deriving new precedence constraints.

```
/* Deriving precedence constraints from $\mathcal{G}$ */
1. compute the dominator nodes of $\mathcal{G}$ according to potential roots;
2. foreach dominator $d$ of $\mathcal{G}$ do
3.  foreach $(u,v) \in \mathcal{E}_{prec}$ such that $d \in DOM_{\langle \mathcal{G},u \rangle}(v)$ do
4.     add the arcs $(u,d)$ and $(d,v)$ to $\mathcal{G}_{prec}$;
5.     remove the arc $(u,v)$ from $\mathcal{G}_{prec}$;
/* Deriving precedence constraints from $\mathcal{G}_{inc}$ and $\mathcal{G}_{prec}$ */
6. foreach node $s$ with at least two successors in $TC(\mathcal{G}_{prec})$ do
7.  foreach successor $u$ of $s$ such that $\exists v : (u,v) \in \mathcal{G}_{inc}$ do
8.     if there exists a node $t \in \mathcal{G}_{prec}$ such that the arcs
       $(v,t)$ and $(s,t)$ are in $TC(\mathcal{G}_{prec})$ then add the arc $(u,t)$ to $\mathcal{G}_{prec}$;
```

The interaction between the precedence constraints (via $\mathcal{G}_{prec}$) and the incomparability constraints (via $\mathcal{G}_{inc}$) can also reveal new precedence constraints. Given four distinct nodes u, v, s, and t, assume there exists an edge (u,v) in $\mathcal{G}_{inc}$ and the arc set $\{(v,t),(s,u),(s,t)\}$ is in $TC(\mathcal{G}_{prec})$. Then the arc (u,t) can be added to $\mathcal{G}_{prec}$. In Figure 4, nodes t and u cannot be incomparable because both are descendants of node s. Moreover, node u cannot be reached from s after reaching t, as otherwise nodes u and v belong to the same path and their incomparability is not respected. Thus, the only way of ordering t and u consists in adding a precedence constraint from u to t. Lines 6 to 8 of Algorithm 4 detect such a pattern in $TC(\mathcal{G}_{prec})$ in $O(mn)$ time.

7 Synthetic Overview of the *tree* Constraint

Table 2 summarises the theoretical results of this article. It is divided into four horizontal parts. The first part shows that the upper bound on NTREE has been improved over [3] without any overhead. The second part points out some necessary conditions that can be evaluated in polynomial time. The third part provides polynomial-time filtering rules derived from the previous necessary conditions. The last part recalls how each instantiation of a father variable leads to updating the precedence digraph as well as the incomparability graph. For each set of propositions and algorithms, an upper bound on

	Interaction	*Effects*	*Related Theorems, Propositions, and Algorithms*	*Time Complexity*
Bounds	$\mathcal{G}$	$\min(\texttt{NTREE})$	Proposition 1 of [3]	$O(n+m)$
	$\mathcal{G}_{prec}$	$\max(\texttt{NTREE})$	Proposition 1	
Feasibility	$\mathcal{G}_{prec}$	$fail$	Theorem 3	$O(mn)$
	$\mathcal{G}_{inc}$		Theorem 4	
	$\mathcal{G}_{prec}$ & $\mathcal{G}_{inc}$		Theorem 5	
Direct Filtering	$\mathcal{G}_{prec}$	$\mathcal{G}$	Algorithm 1	$O(mn)$
	$\mathcal{G}_{inc}$		Algorithm 2	
	$\mathcal{G}_{prec}$ & $\mathcal{G}_{inc}$		Algorithm 3	$O(m^2)$
Internal Derivations	$\mathcal{G}$ & $\mathcal{G}_{prec}$	$\mathcal{G}_{prec}$	Algorithm 4	$O(mn)$
	$\mathcal{G}_{inc}$ & $\mathcal{G}_{prec}$			

Table 2: Summary of the *tree* constraint.

Algorithm 5 General filtering skeleton

```
1. if the tree constraint is feasible then
2.  Update $\mathcal{G}$, $\mathcal{G}_{prec}$, $\mathcal{G}_{inc}$ according to the internal derivations;
3.  Detect infeasible values in the variable domains
    according to Algorithms 1, 2 and 3;
4.  Remove the infeasible values detected by statement 3;
5.  if at least one value was removed then go to statement 1
    else exit;
6. else generate a failure and exit.
```

the time complexity is provided, where n and m respectively denote the numbers of nodes and arcs in the digraph $\mathcal{G}$. Let m_{prec} and m_{inc} respectively denote the numbers of arcs and edges in $\mathcal{G}_{prec}$ and $\mathcal{G}_{inc}$. Notice that $m_{prec} \leq m$ (because $\mathcal{G}_{prec}$ is an acyclic digraph without any transitive arc) and $m_{inc} \approx m$. Thus, the time complexity of the filtering algorithms is provided below only in terms of the number m of arcs in $\mathcal{G}$.

A frequent problem with the combination of different kinds of filtering is to ensure that the same fixpoint is reached independently of the ordering of the filtering rules. Thus, Algorithm 5 is based on a saturation loop such that if the constraint is feasible (line 1), then first all the data structures are updated (line 2), next each filtering rule is applied and inconsistent values are recorded but not immediately removed (line 3), then inconsistent values are removed (line 4), and finally a new iteration begins if a value was removed (line 5), because a value removal modifies $\mathcal{G}$ and/or NTREE. Thus, line 3 ensures that for a given iteration, all the filtering rules are applied on the same data (i.e., the same digraph $\mathcal{G}$ and the same NTREE).

8 Experimental Results

We now report on several experiments we have conducted to evaluate the extended *tree* constraint. First, in Section 8.1, we discuss our experiments on real-life instances of (an extension of) the biological problem of constructing phylogenetic supertrees, and show that the *tree* constraint significantly outperforms the previous constraint programming approach. Then, in Section 8.2, we present our results on the routing problem of constructing ordered simple paths with mandatory nodes. Finally, in Section 8.3, we report on the performance on random instances for the Hamiltonian path problem.

All experiments were performed with the Choco constraint programming system (which is a Java library) on an Intel Pentium 4 CPU with 3GHz and a 1GB RAM, but with 512MB allocated to the Java Virtual Machine.

8.1 The Phylogenetic Supertree Problem

One objective of phylogeny is to construct the genealogy of the species, called the *tree of life*, whose leaves represent the contemporary species and whose internal nodes represent extinct species that are not necessarily named. An important problem in phylogeny is the construction of a supertree [7] that is compatible with several given trees. There are several definitions of tree compatibility in the literature:

Definition 21 (Strong, weak, and stable compatibility).

- *A tree $\mathcal{T}$ is* strongly compatible *with a tree $\mathcal{T}'$ if $\mathcal{T}'$ is topologically equivalent to a subtree of $\mathcal{T}$ that respects the node labelling. [23]*
- *A tree $\mathcal{T}$ is* weakly compatible *with a tree $\mathcal{T}'$ if $\mathcal{T}'$ can be obtained from $\mathcal{T}$ by a series of arc contractions.*[5] *[31]*
- *A tree $\mathcal{T}$ is* stably compatible *with a set $\mathcal{S}$ of trees if $\mathcal{T}$ is weakly compatible with each tree in $\mathcal{S}$ and each internal node of $\mathcal{T}$ can be labelled by at least one corresponding internal node of some tree in $\mathcal{S}$.*

For the supertree problem, strong and weak compatibility coincide if and only if all the given trees are binary [23]. The existence of solutions is not lost when restricting weak compatibility to stable compatibility.

For example, the trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of Figure 5 have $\mathcal{T}$ and $\mathcal{T}'$ as supertrees under both weak and strong compatibility. As shown, all the internal nodes of $\mathcal{T}'$ can be labelled by corresponding internal nodes of the two given trees, but this is not the case for the father of b and g in $\mathcal{T}$. Hence $\mathcal{T}$ and four other such supertrees are debatable because *they speculate about the existence of extinct species* that were not in any of the given trees. Consider also the three small trees in Figure 6: $\mathcal{T}_3$ and $\mathcal{T}_4$ have $\mathcal{T}_4$ as a supertree under weak compatibility, as it suffices to contract the arc $(3, 2)$ to get $\mathcal{T}_3$ from $\mathcal{T}_4$. However, $\mathcal{T}_3$ and $\mathcal{T}_4$ have no supertree under strong compatibility, as the most recent common ancestor of b and c, denoted by $mrca(b, c)$, is the same as $mrca(a, b)$ in $\mathcal{T}_3$, namely 1,

[5] The *contraction* of an arc $a = (v, w)$ is the replacement of v and w by a single node whose incident arcs are those of v and w other than a.

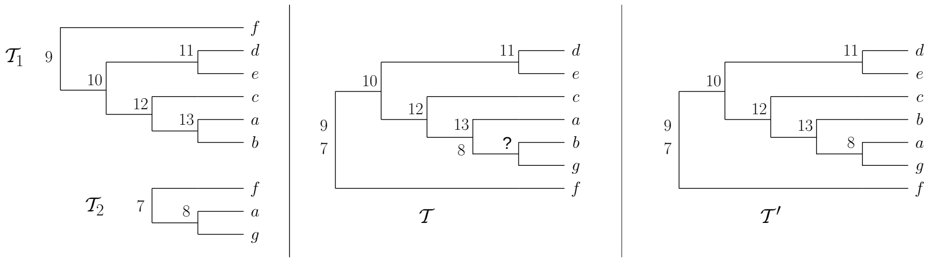


Fig. 5: Supertree problem instance and two of its solutions

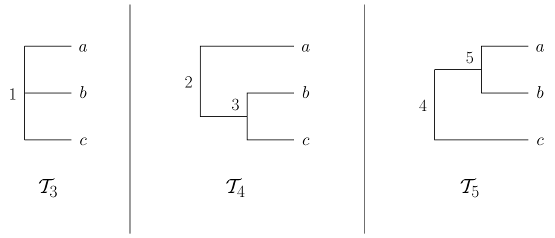


Fig. 6: Three small phylogenetic trees

but not the same in $\mathcal{T}_4$, as $mrca(b, c) = 3$ is an evolutionary descendant of $mrca(a, b) = 2$. Also, $\mathcal{T}_4$ and $\mathcal{T}_5$ have neither weakly nor strongly compatible supertrees.

Under strong compatibility, a first supertree algorithm was given in [1], with an application for database management systems; it takes $O(\ell^2)$ time, where ℓ is the number of leaves in the given trees. Derived algorithms have emerged from phylogeny, for instance *OneTree* [23]. The first constraint program was proposed in [17], using standard, non-global constraints. Under weak compatibility, a phylogenetic supertree algorithm can be found in [31] for instance. Under stable compatibility, the algorithm from computational linguistics of [9] has supertree construction as a special case.

Under stable compatibility, the supertree problem for trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$ can be modelled by an extended *tree* constraint, such that:

– The digraph $\mathcal{G}$ is the complete digraph $(\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V} = \mathcal{N}(\mathcal{T}_1) \cup \cdots \cup \mathcal{N}(\mathcal{T}_k)$ and edge set $\mathcal{E} = \{(u, v) \mid u, v \in \mathcal{V}\}$, where $\mathcal{N}(\mathcal{T})$ denotes the set of nodes of tree $\mathcal{T}$.
– The precedence digraph $\mathcal{G}_p = (\mathcal{V}, \mathcal{E}_p)$ is dictated by $\mathcal{T}_1, \ldots, \mathcal{T}_k$.
– The incomparability constraints are generated from the incomparable nodes of each tree $\mathcal{T}_1, \ldots, \mathcal{T}_k$.
– `NTREE` $= 1 =$ `NPROP`, i.e., the partition of $\mathcal{G}$ must consist of exactly one proper tree.
– All the leaves of $\mathcal{T}_1, \ldots, \mathcal{T}_k$ that are not internal nodes of any tree $\mathcal{T}_i$ must remain leaves and thus have an in-degree of zero.
– All the other nodes of $\mathcal{V}$ have degrees 1 or 2 if a binary supertree is requested, and within $[1, n-1]$ otherwise, where $n = |\mathcal{V}|$.

A smallest-domain heuristic is used to select a father variable at each waking up of the *tree* constraint (i.e., each time the solver instantiates a variable), and the value selection heuristic favours, for a selected node v_i, a father v_j such that there exists a minimum-length path from v_i to v_j or vice-versa in the precedence digraph $\mathcal{G}_{prec}$. The latter heuristic is based on the following intuition: the longer a maximum path from v_i to v_j in $\mathcal{G}_{prec}$, the lower the chances of satisfying all the precedence constraints involved in this path.

Table 3 compares the performance of our Choco constraint model, under stable compatibility, with an improvement (now also written in Choco) by Prosser of the constraint model, under strong compatibility, of [17] (available at `http://www.dcs.gla.ac.uk/~pat/supertrees/`). The statistics are until the first solution is found or until the absence of solutions is established. For a given instance, the column 'satisfiable' indicates the existence of supertrees, and, between parentheses, the existence of binary supertrees. Also, 'n/a' stands for 'not applicable'. There are 17 leaf species in the two spider trees S_1 and S_2, which were taken from study S1x6x97c14c42c30 in TreeBASE (see `http://www.treebase.org/`); they feature side constraints on nested species and one of these trees is not binary, hence there is no binary supertree. Only our model can accommodate that side constraint without reformulation. There are 23 leaf species in the two cat trees C_1 and C_2, which were taken from biology journals; one of them is not binary, hence there is no binary supertree. There are 129 leaf species across the seven seabird trees A to G, which were taken from an ornithology journal [19]; only the trees A, E, and G are not binary, and only the listed subsets of at least three of these seven trees have stably compatible supertrees.

Table 3: Real-life phylogenetic supertree construction

instance	#species	name	$\lvert\mathcal{G}\rvert$	satisfiable	#fails	time (ms)
$S_1 + S_2$	17	*tree*	18	yes (no)	0	48
		Prosser	n/a	yes	1	155
$C_1 + C_2$	23	*tree*	26	yes (no)	0	75
		Prosser	n/a	yes	2	254
$A + B$	30	*tree*	52	yes (no)	0	302
		Prosser	n/a	yes	32	648
$A + C$	34	*tree*	63	yes (no)	0	406
		Prosser	n/a	no	0	810
$A + D$	47	*tree*	85	yes (no)	0	398
		Prosser	n/a	yes	0	1972
$A + E$	95	*tree*	191	yes (no)	0	10393
		Prosser	n/a	n/a	n/a	out of memory
$A + F$	33	*tree*	58	yes (no)	0	127
		Prosser	n/a	yes	4	710
$A + G$	49	*tree*	82	yes (no)	0	409
		Prosser	n/a	yes	9	2135
$B + C$	32	*tree*	65	no (no)	0	32
		Prosser	n/a	no	1770	12866

instance	#species	name	$\lvert\mathcal{G}\rvert$	satisfiable	#fails	time (ms)
$B + D$	43	*tree*	85	yes (yes)	0	301
		Prosser	n/a	yes	3	1683
$B + E$	95	*tree*	195	no (no)	0	892
		Prosser	n/a	n/a	n/a	out of memory
$B + F$	33	*tree*	62	yes (no)	0	144
		Prosser	n/a	yes	0	606
$B + G$	44	*tree*	81	yes (no)	0	1440
		Prosser	n/a	yes	35	1765
$C + D$	52	*tree*	101	yes (no)	0	630
		Prosser	n/a	yes	0	2979
$C + E$	96	*tree*	203	yes (no)	0	27180
		Prosser	n/a	n/a	n/a	out of memory
$C + F$	38	*tree*	74	yes (no)	0	393
		Prosser	n/a	yes	3	979
$C + G$		*tree*	93	yes (no)	0	1530
	49	Prosser	n/a	no	0	2371
$D + E$	104	*tree*	220	no (no)	0	1126
		Prosser	n/a	n/a	n/a	out of memory
$D + F$	46	*tree*	91	yes (yes)	0	630
		Prosser	n/a	yes	4	1776
$D + G$	59	*tree*	112	yes (no)	0	910
		Prosser	n/a	no	35	4403
$E + F$	96	*tree*	199	no (no)	0	1035
		Prosser	n/a	n/a	n/a	out of memory
$E + G$	100	*tree*	211	no (no)	0	1211
		Prosser	n/a	n/a	n/a	out of memory
$F + G$	43	*tree*	83	no (no)	0	62
		Prosser	n/a	n/a	n/a	$> 5 \cdot 10^5$
$A + C + E$	99	*tree*	215	yes (no)	0	49224
		Prosser	n/a	n/a	n/a	out of memory
$A + B + D + F$	72	*tree*	139	yes (no)	0	8139
		Prosser	n/a	yes	59	4811
$A + B + D + G$	82	*tree*	157	no (no)	0	347
		Prosser	n/a	n/a	n/a	out of memory
$A + C + D + F$	76	*tree*	150	yes (no)	0	8690
		Prosser	n/a	no	0	2553
$A + C + D + G$	86	*tree*	168	yes (no)	0	12650
		Prosser	n/a	n/a	n/a	out of memory

Our model generates some symmetric solutions (as some internal nodes are fathers of only one node, which is also internal, so that their roles can be inverted), whereas the model of [17] generates unique solutions modulo all symmetries. For instance, for the two given trees of Figure 5, we get 36 supertrees instead of the 8 actually captured by stable compatibility, due to the interchangeability of the internal nodes 8 and 10 and the interchangeability of the internal nodes 7 and 9. By contracting in a post-processing step all arcs (v, w) where node w has only one incoming arc, we obtain 10 supertrees, as some mirror symmetry remains.

However, our model has only $\Theta(\ell)$ domain variables, where ℓ is the number of leaves in the given trees, whereas the model of [17] has $\Theta(\ell^2)$ domain variables. The runtime and memory consequences thereof on large instances can be clearly observed in Table 3.

Specialised $O(\ell^2)$ runtime supertree algorithms, such as the ones of [9, 23], of course systematically and drastically outperform our constraint model (even the hardest of the considered instances take less than 100 ms). However, they do not provide the flexibility of a constraint programming approach, as every combination of the currently emerging biological side constraints (on nested species or relative ancestral divergence dates, say) and objective functions (when switching to an optimisation version of the problem) requires a new algorithm. For instance, the extended *tree* constraint directly accommodates the nested-species side constraint, as seen for the spider trees, but the algorithms of [9, 23] cannot do that.

8.2 The Ordered Simple Path Problem with Mandatory Nodes

We now evaluate the *tree* constraint on the *ordered disjoint paths problem* (ODP), which consists in partitioning a given (di)graph into a given number of mutually node-disjoint paths [16, page 217], subject to precedence constraints between nodes. The extended *tree* constraint can directly deal with this problem, but, in order to compare with [26], our evaluation is done on a restriction of this problem, called the *ordered simple path problem with mandatory nodes* (OSPMN), which consists in finding an elementary path containing a set of mandatory nodes in a given order.

The OSPMN problem can be modelled by an extended *tree* constraint whose digraph $\mathcal{G}$ is to be covered, but enriched by a loop on each node, while the set of mandatory nodes is contained in a connected component of the precedence digraph $\mathcal{G}_{prec}$ such that each mandatory node succeeds the first node of the path and precedes the final node of the path, and if there exist precedence constraints between two mandatory nodes, then an arc is added between them. All the other nodes represent connected components of size 1 in $\mathcal{G}_{prec}$. An ordered-path heuristic is used to select a father variable at each waking up of the *tree* constraint. This heuristic forces an incremental building of the path by selecting as new variable to instantiate the value chosen at the previous step: if an arc (v_i, v_j) is enforced at a given step, then an arc starting from v_j is selected at the next step.

Table 4 shows that our model compares favourably, in the number of failures, with the results reported in [26] for an equivalent hardware, until the first solution is found or until the absence of solutions is established. In [26], the used constraints *domReachability*, *noCycle*, and *allDifferent* are implemented in the *Gecode*(*CP*(*Graph*)) C++ library [30], which explains why the computation times are nevertheless very similar.

OSPMN instances	*tree*			*DomReachability+Path* [26]	
	wake-ups	failures	time	failures	time
$SPMN\_22$	7	0	52	5	110
$SPMN\_22\_full$	3	0	36	0	70
$SPMN\_52b$	20	0	1115	6	920
$SPMN\_52\_full$	6	0	562	0	580
$SPMN\_52order\_a$	6	0	592	0	500
$SPMN\_52order\_b$	1	0	17	4	280

Table 4: Results for the OSPMN instances in [26]

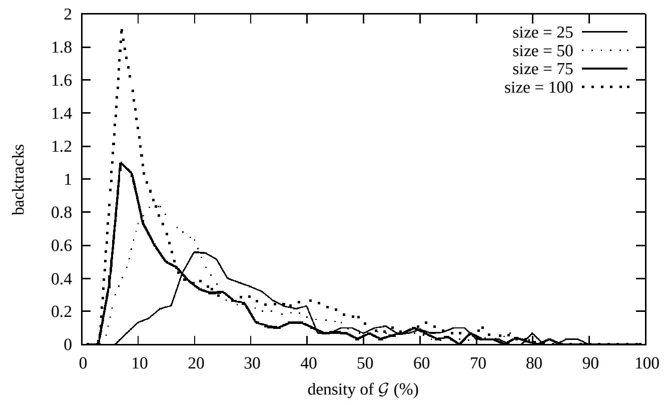
8.3 Random Instances for the Hamiltonian Path Problem

Our model for the Hamiltonian path problem uses an extended *tree* constraint where each node of the given digraph has an in-degree of one, except the origin of the path, which has an in-degree of zero. Precedence constraints are added so that the origin of the path precedes all the other nodes, and all these nodes precede the destination of the path.

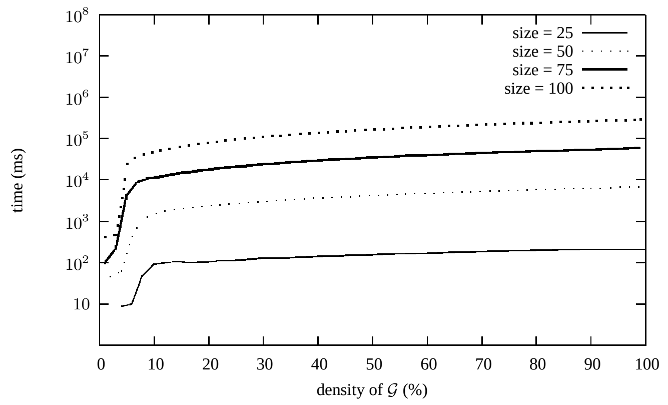
For each density[6] among $\{0\%, 10\%, \ldots, 100\%\}$, a total of 50 random connected digraphs of sizes 25, 50, 75, and 100 were generated, the origin and destination nodes being randomly chosen. Figure 7a shows that the instances are harder for a density of 8% to 22%. Intuitively, the denser a graph, the higher the probability of existence of a Hamiltonian path [20, 25]. If there are no arcs, then no Hamiltonian path can exist. If the graph is complete, then the existence of such a path is guaranteed. From this observation, the left-hand side of the density interval $[8\%, 22\%]$ of Figure 7a is easy (according to the number of backtracks) because the probability of existence of a Hamiltonian path is low, and the right-hand side of this interval is also easy because this probability is high. Figure 7b confirms the theoretical runtime complexity announced in Table 2 of Section 7. The interval of more complex instances (in terms of backtracks) is not discernable because even if it is easy (in terms of backtracks) to provide a Hamiltonian path for a dense graph, the runtime complexity of the filtering algorithms depends directly on the *number* of arcs in the graph. In practice, a more efficient *tree* constraint would require an efficient trigger for each filtering algorithm. This is not surprising, and randomised algorithms have been proposed for some global constraints, such as *AllDifferent* and *Global Cardinality* [18]. Finally, an average of only 1.9 backtracks for graphs of size 100 may seem strange for a well-known NP-hard problem. This observation rather highlights the difficulty of generating hard randomised instances for the Hamiltonian path problem. Obviously, there do exist pathological graphs, such as Tutte's graph [32],[7] for which we need 120 backtracks to prove that there exists no Hamiltonian cycle.

[6] The *density* of an n-node m-arc digraph is m/n^2.

[7] *Tutte's graph* is a non-Hamiltonian 3-connected cubic graph of size 47.



(a) Backtracks in terms of the density of the graph



(b) Runtime in terms of the density of the graph

Fig. 7: Evaluation of the *tree* constraint on the Hamiltonian path problem.

9 Conclusion

The *tree* and *path* constraints have been unified within a single global constraint. Moreover, we have shown how to handle in a uniform way a variety of side constraints, namely precedence, incomparability, and degree constraints, which often occur in the context of path and tree problems. The resulting global constraint can thus tackle a large variety of graph partitioning problems related to paths or trees.

Our experiments, particularly on dense graphs, point to an important topic for future research, namely the finding of efficient filters that avoid triggering heavy algorithms when there is obviously nothing to prune (particularly for expensive filtering algorithms like the one of the global cardinality constraint [18]). Moreover, using fully incremental algorithms in order to maintain during search some graph properties such as the transitive closure, the strongly connected components, etc, seems a crucial point to improve the scalability and runtime complexity [29]. However, coming up with fully dynamic algorithms for complex graph properties that need to maintain and synchronise many data structures seems quite challenging, even if such algorithms exist for many graph properties considered independently.

In the case of the phylogenetic supertree problem, we point out that our approach naturally provides a lot of information on the structure of the supertrees, by means of the precedence digraph $\mathcal{G}_{prec}$. However, there remain two questions. First, is it possible to provide a complete filtering algorithm for this problem, even if solving its model in terms of precedence and incomparability constraints is NP-hard in general? Second, is it possible to see the precedence digraph as a canonical form of the set of compatible supertrees for a given set of trees?

Acknowledgements

We are grateful for the constructive discussions with Manuel Bodirsky, Vincent Moulton, Patrick Prosser, and Luis Quesada. Many thanks also to the anonymous referees for their useful comments.

References

1. A. Aho, Y. Sagiv, T. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal of Computing*, 10(3):405–421, 1981.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Research Report T2005-08, Swedish Institute of Computer Science, 2005.
3. N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
4. N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, Sweden, 2006. Available at http://www.it.uu.se/research/publications/reports/2006-020/.
5. N. Beldiceanu, P. Flener, and X. Lorca. Partitionnement de graphes par des arbres sous contraintes de degré. In *Deuxièmes Journées Francophones de Programmation par Contraintes (JFPC'06)*, pages 35–42, 2006. In French.
6. N. Beldiceanu and X. Lorca. Necessary condition for path partitioning constraints. In *Proceedings of CP-AI-OR'07*, volume 4510 of *LNCS*, pages 141–154. Springer-Verlag, 2007.
7. O. Bininda-Emonds, J. Gittleman, and M. Steel. The (super)tree of life: Procedures, problems, and prospects. *Annual Reviews of Ecological Systems*, 33:265–289, 2002.
8. M. Bodirsky, D. Duchier, S. Miehle, and J. Niehren. A new algorithm for normal dominance constraints. In *Proceedings of SODA'04*, pages 59–67, 2004.
9. M. Bodirsky and M. Kutz. Determining the consistency of partial tree descriptions. *Artificial Intelligence*, 171:185–196, 2007.
10. E. Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes*. PhD thesis, University of Paris 13, France, March 1999. In French.
11. H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. In *Proceedings of the Dixièmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets (JNPC'04)*, pages 107–120, 2004. In French.
12. A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

13. K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 31(4):1–10, 2001.
14. COSYTEC. *CHIP Reference Manual*, release 5.1 edition, 1997.
15. G. Dooms, Y. Deville, and P. E. Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *Proceedings of CP'05*, volume 3709 of *LNCS*, pages 211–225, 2005.
16. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1978.
17. I. Gent, P. Prosser, B. Smith, and W. Wei. Supertree construction with constraint programming. In *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 837–841, 2003.
18. I. Katriel. Expected-case analysis for delayed filtering. In *Proceedings of CP-AI-OR'06*, volume 3990 of *LNCS*, pages 119–125. Springer-Verlag, 2006.
19. M. Kennedy and R. D. Page. Seabird supertrees: Combining partial estimates of procellariiform phylogeny. *The Auk, A Quarterly Journal of Ornithology*, 119:88–108, 2002.
20. J. Komlós and E. Szemerédi. Limit distribution for the existence of a Hamilton cycle in a random graph. *Discrete Mathematics*, 43:55–63, 1983.
21. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.
22. X. Lorca. *Contraintes de Partitionnement de Graphe*. PhD thesis, Université de Nantes, École des Mines, Nantes, France, 2007. In French.
23. M. Ng and N. Wormald. Reconstruction of rooted trees from subtrees. *Discrete Applied Mathematics*, 69:19–31, 1996.
24. C. L. Pape, L. Perron, J.-C. Régin, and P. Shaw. Robust and parallel solving of a network design problem. In *Proceedings of CP'02*, volume 2470 of *LNCS*, pages 633–648. Springer-Verlag, 2002.
25. L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14:359–364, 1976.
26. L. Quesada. *Solving Constrained Graph Problems Using Reachability Constraints Based on Transitive Closure and Dominators*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.
27. J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *Proceedings of AAAI'94*, pages 362–367, 1994.
28. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI'96*, pages 209–215, 1996.
29. G. Richaud, X. Lorca, and N. Jussien. A portable and efficient implementation of global constraints: The *tree* constraint case. In S. Abreu and V. S. Costa, editors, *Proceedings of CICLOPS'07*, Porto, Portugal, September 2007.
30. C. Schulte, M. Lagerkvist, and G. Tack. *Gecode*, 2006. Available at http://www.gecode.org/.
31. M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9:91–116, 1992.
32. W. T. Tutte. On Hamiltonian circuits. *Journal of the London Mathematical Society*, 21:98–101, 1946.