Revisiting the `tree` Constraint

Jean-Guillaume Fages and Xavier Lorca

École des Mines de Nantes, INRIA, LINA UMR CNRS 6241,
FR-44307 Nantes Cedex 3, France
`{Jean-guillaume.Fages,Xavier.Lorca}@mines-nantes.fr`

Abstract. This paper revisits the `tree` constraint introduced in [2] which partitions the nodes of a n-nodes, m-arcs directed graph into a set of node-disjoint anti-arborescences for which only certain nodes can be tree roots. We introduce a new filtering algorithm that enforces generalized arc-consistency in $O(n + m)$ time while the original filtering algorithm reaches $O(nm)$ time. This result allows to tackle larger scale problems involving graph partitioning.

1 Introduction

In the recent history of constraint programming, global constraints constitute a powerful tool for both modeling and resolution. Today still, the most commonly used global constraints are based on an intensive use of concepts stemming from graph theory. Of these, the most important are cardinality constraints [12, 13] and automaton based constraints [8, 9, 7]. More generally, the reader should refer to the catalogue of constraints [1] to gain a more complete idea of the graph properties used in global constraints. In the same way, difficult problems modeled and solved thanks to graph theory have been successfully tackled in constraint programming and, more particularly, thanks to global constraints. This mainly consists of constraints around graph and subgraph isomorphism [17, 19], search paths in graphs [11, 10, 16], even minimum cost spanning trees [14] and graph partitioning constraints like the `tree` constraint [2] Such a constraint is mainly involved in practical applications like vehicle routing, mission planning, DNA sequencing, or phylogeny.

The `tree` constraint enforces the partitioning of a directed graph $G = (V, E)$ into a set of L node-disjoint anti-arborescences, where $|V| = n$, $|E| = m$ and L is an integer variable. In [2], it is shown that Generalized Arc-Consistency (GAC) can be enforced in $O(nm)$ time, while feasibility can be checked in $O(n + m)$ time. The bottleneck of the filtering algorithm relies on the computation of strong articulation points which, at this moment, could not be performed in linear time. However, based on the works of [15, 3], Italiano et. al. [5] solved this open problem by giving an $O(n + m)$ time algorithm for computing strong articulation points of a directed graph G. Their main contribution is the link they made between the concept of strong articulation point in a directed graph and the concept of dominator in a directed flow graph. This recent improvement in graph theory made us revisit the `tree` constraint to see whether the complete filtering algorithm could now be computed in linear time or not.

In this paper, Section 2 first recalls some basic notions of graph theory as well as the integration of graph theory in a classical constraint programming framework. Next, Section 3 proposes a short survey about the `tree` constraint. First, a decomposition of the constraint is discussed in Section 3.1. Next, Section 3.2 shows that dominator nodes were already used in a different way to enforce reachability between nodes. Finally, Section 3.3 proposes a brief sum up of the original `tree` constraint filtering algorithm. This leads us to discuss the relevance of the approach showing that strong articulation points is a much too strong notion for this problem whereas the notion of dominators in a flow graph perfectly fits our needs. Thus, Section 4 presents a new complete filtering algorithm, closer to the definition of a tree, that runs in $O(n + m)$ worst-case time complexity. Finally, Section 5 concludes the paper with a short evaluation of the algorithm to illustrate how large scale problems can be tackled thanks to this new filtering algorithm.

2 Graph Theory Definitions

A *graph* $G = (V, E)$ is the association of a set of nodes V and a set of edges $E \subseteq V^2$. $(x, y) \in E$ means that x and y are connected. A *directed graph* (*digraph*) is a graph where edges are directed from one node to another, it is generally noted $G = (V, A)$. $(x, y) \in A$ means that an arc goes from x to y but does not provide information about whether an arc from y to x exists or not.

A *connected component* (CC) of a digraph $G = (V, A)$ is a maximal subgraph $G_{CC} = (V_{CC}, A_{CC})$ of G such that a chain exists between each pair of nodes. A *strongly connected component* of a digraph $G = (V, A)$ is a maximal subgraph $G_{SCC} = (V_{SCC}, A_{SCC})$ of G such that a directed path exists between each pair of nodes. A graph is said (strongly) *connected* if it consists in one single (strongly) connected component.

A *strong articulation point (SAP)* of a digraph $G = (V, A)$ is a node $s \in V$ such that the number of strongly connected components of $G \backslash \{s\}$ is greater than the one of G. In other words $s \in V$ is a SAP of G if there exists two nodes x and y in V, $x \neq s$, $y \neq s$, such that each path from x to y goes through s and a path from y to x exists.

A *flow graph* $G(r)$ of a directed graph $G = (V, E)$ is a graph rooted in the node $r \in V$ which maintains the following property: for each node $v \in V$ a directed path from r to v exists. A *dominator* of a flow graph $G(r)$ where $G = (V, A)$ is a node $d \in V$, $d \neq r$ such that there exists at least one node $v \in V$, $v \neq d$, for which all paths from the root r to v go through d. We extend this notion to arcs as following: an *arc-dominator* of a flow graph $G(r)$ where $G = (V, A)$ is an arc $(x, y) \in A$, $x \neq y$, such that there exists at least one node $v \in V$, for which all paths from the root r to v go through (x, y). This definition can easily be simplified into: an arc $(x, y) \in A$ is an *arc-dominator* of $G(r)$ if and only if $x \neq y$ and all paths from the root r to y go through (x, y).

A *tree* $T = (V, E)$ is an acyclic, connected and undirected graph. One of its directed variants is the *anti-arborescence*, a directed graph $T = (V, A)$ such that every node $v \in V$ has exactly one successor $w \in V$ and with one root $r \in V$, such that $(r, r) \in A$ and for each node $v \in V$, a directed path from v to r exists. It can be seen that an anti-arborescence can be transformed into a tree easily. In the paper we study the case of an anti-arborescence but use for simplicity the term tree rather than anti-arborescence. Thus the following definitions will be used as references:

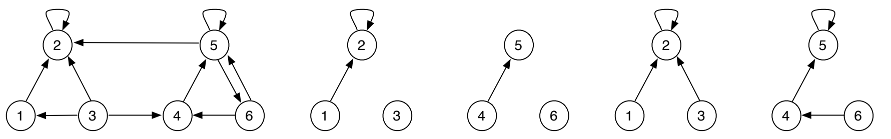
Definition 1. *A* tree $T = (X, Y)$ *is a connected digraph where every node* $v \in X$ *has exactly one successor* $w \in X$ *and with one root* $r \in X$ *such that* $(r, r) \in Y$ *and for each node* $v \in X$*, a directed path from* v *to* r *exists.*

Definition 2. *Given a digraph* $G = (V, A)$*, a* tree partition *of* G *is a subgraph* $P = (V, A_2)$*,* $A_2 \subseteq A$*, such that each connected component is a tree.*

Then, the two previous definitions directly provide the proposition:

Proposition 1. *Given a digraph* $P = (V, A_2)$*, subgraph of a digraph* $G = (V, A)$*, and a set* $R = \{r | r \in V, (r, r) \in A_2\}$*, then* P *is a tree partition of* G *if and only if each node in* V *has exactly one successor in* P *and for each node* $v \in V$ *there exists a node* $r \in R$ *such that a directed path from* v *to* r *exists in* P*.*

In a constraint programming context a *solution* to the `tree` constraint is a tree partition of an input graph $G = (V, A)$. A *Graph Variable* is used to model the partitioning of G. It is composed of two graphs: the *envelope*, $G_E = (V, A_E)$, contains all arcs that potentially occur in at least one solution (Figure 1(a)) whereas the *kernel*, $G_K = (V, A_K)$, contains all arcs that occur in every solution (Figure 1(b)). It has to be noticed that $A_K \subseteq A_E \subseteq A$. During the resolution, filtering rules will remove arcs from A_E and decisions that add arcs to A_K will be applied until the Graph Variable is instantiated, i.e. when $A_E = A_K$ (Figure 1(c)). The problem has no solution when $|A_E| < |V|$.



(a) Envelope $G_E = (V, A_E)$ (b) Kernel $G_K = (V, A_K)$ (c) A solution $(A_E = A_K)$

Fig. 1. A graph variable associated with a directed graph $G = (V, A)$

Definition 3. *An instantiated* Graph Variable *is a tree partition of a digraph* G*, if and only if its kernel* G_K *is a tree partition of* G*. A partially instantiated* Graph Variable *can lead to a tree partition of a digraph* G*, if and only if there exists at least one tree partition of its envelope* G_E*.*

In the following, a node $r \in V$ is a *root* if and only if $(r, r) \in A_K$. It is called a *potential root* if $(r, r) \in A_E$.[1]

3 The `tree` Constraint: A survey

This section introduces first a decomposed constraint programming model for the `tree` constraint. Such a model does not ensure any consistency level for the constraint. Next, we show how the `DomReachability` constraint can be used as a propagator for the `tree` constraint. Finally, we recall the initial GAC filtering algorithm of the constraint.

3.1 A Basic Decomposition

Our objective in this section is to convince the reader of the importance in proposing a global constraint for directed tree partitioning. In order to do so, we will introduce a broken down model for this problem. It is necessary to define an integer variable L characterizing the number of trees admitted in the partition. Next, taking $G = (V, E)$ a graph of order n, we link to each node i an integer variable v_i of enumerated domain $[1; n]$ defining the successor of i in G, an integer variable r_i of bounded domain $[0; n-1]$ defining the height of i in a solution and a boolean variable b_i characterizing the root property for node i. Note that the set of decision variables (to be used for branching) is v and that r is introduced to prevent from the creation of circuits. As such, the problem can be defined in the following way:

$$v_i = j \wedge i \neq j \Rightarrow r_i > r_j, \ \forall i \in [1; n] \tag{1}$$

$$b_i \Leftrightarrow v_i = i, \ \forall i \in [1; n] \tag{2}$$

$$L = \sum_{i=1}^{n} b_i \tag{3}$$

The correctness of the model is proved when we consider the following cases: (1) the model does not accept a solution containing more than L well-formed trees, (2) the model does not admit a solution containing fewer than L well-formed trees, (3) the model does not accept a solution containing a circuit and (4) the model does not accept a solution containing a single node without a loop. Given $G = (V, E)$, a directed graph of order n and F a partition of G into α well-formed trees:

- case (1), let us suppose that $\alpha > L$ so that if $\alpha > \sum_{i=1}^{n} b_i$ this means that there are more well-formed trees in F than nodes which are potential roots, which is impossible according to constraint (2);
- case (2), let us suppose that $\alpha < L$ so that if $\alpha < \sum_{i=1}^{n} b_i$ this means that F contains more loops than trees so that some contain more than one loop. However, since each node has exactly one successor, it is a contradiction;

[1] Notice that in this definition, a root is also a potential root.

- case (3), if there is $f \in F$ such that f contains a cycle, then, there are two nodes i and j in f such that we have an elementary path from i to j and an elementary path from j to i and, consequently, according to the constraint(1) we have $r_i < r_j$ and $r_j < r_i$: it is a contradiction;
- case (4) is obvious because each variable must be fixed to a value, which is equivalent to saying that each node must have exactly one successor.

3.2 The `DomReachability` Constraint

Luis Quesada et. al. introduced the `DomReachability` constraint [11, 10, 16] for solving path partitioning problems. It uses a similar graph variable description [4] .Their constraint maintains structural relationships between a flow graph, its dominator tree and its transitive closure. In particular, it can ensure that all nodes are reachable from a given source node, which is very close to the concept of a tree. Plus, and as it will be shown in the next subsection, dominance relationships are very useful information in that context. Thus one could think that it could be a good propagator for solving tree partitioning problems. We will show that the use of such a constraint is not appropriate.

`DomReachability` runs in $O(nm)$ worst case time complexity. This cost is due to the algorithm used for maintaining the transitive closure, which is not necessary for tree partitioning. The first algorithm [11] consisted in performing one DFS per node whereas the current algorithm [16] works on a reduced graph.

`DomReachability` does not enable to build a tree partition directly. As it needs a single source node it can only compute a single tree. To get a tree partition of cardinality k, then a trick would consist in adding a fictive source node s to the input graph and declare that each of its successors is a root node of a tree and add a propagator which would ensure that the outdegree of s is k. In the same way, even if one single tree is expected, when the root node is not known in advance then it is necessary to use the previous trick with $k = 1$.

`DomReachability` does not ensure GAC over tree partitioning. This is pretty obvious because the reachability property does not exclude cycles whereas tree properties do. Nevertheless they use `DomReachability` for path partitioning through the global constraint Simple Path with Mandatory Nodes (SPMN) [11, 10, 16]. As tree partitioning is a polynomial relaxation of path partitioning, a complete filtering over tree partitioning could be expected. However, to the best of our knowledge, SPMN does not reap the benefit of dominators by missing the following major pruning rule: if i dominates j then the arc (j, i) does not belong to any solution which, regarding their notation, can be expressed as:

$$\frac{\langle i,j\rangle \in Edges(Min(EDG))}{Edges(Max(FG)) := Edges(Max(FG)) \setminus \{\langle j,i\rangle\}} \tag{4}$$

For a better understanding of this rule, it has to be considered that they work on arborescences instead of anti-arborescences.

3.3 The original `tree` Constraint

This part is a fast sum up of the tree constraint described in [2]. The constraint is composed of two main algorithms. The first one enables to check whether a partially instantiated *Graph Variable* can lead to at least one solution or not. This algorithm can be run in $O(n+m)$ worst case time complexity. The second algorithm enables to prune every arc that does not belong to any solution in $O(nm)$ worst case time complexity. It is the focus of this paper.

Feasibility Condition. The original paper defines an integer variable $L = [\underline{\ell}; \overline{\ell}]$ that represents the cardinality of the tree partition and two bounds: $\underline{\ell}^*$, the number of sink components[2], and $\overline{\ell}^*$, the number of potential roots. Those two bounds can easily be evaluated in linear time: all the strongly connected components of G_E can be computed in $O(n+m)$ using Tarjan's algorithm [18]. Thus, the sink components can be detected in $O(m)$ time, which provides $\underline{\ell}^*$. Moreover a simple breadth first search exploration of G_E enables to compute $\overline{\ell}^*$.

The feasibility condition can be decomposed into two parts. The first one is directly related to the number of trees allowed into the partition, while the second one is related to the definitions of directed tree: $dom(L) \cap [\underline{\ell}^*; \overline{\ell}^*] \neq \emptyset$, and all sink components of G_E must contain at least one potential root. The original paper provides the proof of sufficiency of those conditions which can obviously be checked in linear time.

Complete Filtering Algorithm. The complete filtering algorithm can be split into two propagators: bound filtering and structural filtering. The bound filtering focuses on the cardinality of the expected partition whereas the structural filtering ensures the generalized arc-consistency over tree partitioning properties. Both algorithms are complementary and form together a complete filtering algorithm for tree constraint.

The bound filtering is pretty simple. First of all, it consists in ensuring that $dom(L) \cap [\underline{\ell}^*; \overline{\ell}^*] \neq \emptyset$ by removing the values of L that are out of range. Secondly, it consists in pruning infeasible arcs when L is instantiated to one of its extrema: If $dom(L) = \{\underline{\ell}^*\}$ then any potential root in a non sink component is infeasible and thus removed from the envelope; If $dom(L) = \{\overline{\ell}^*\}$ then any potential root must be instantiated as a root thus all their outgoing arc that are not a loop are removed from the envelope.

The structural filtering detects all arcs that belong to no tree partition. For this purpose, several notations are required: A *door* is a vertex $v \in V$ such that there exists $(v, w) \in A_E$ where w does not belong to the same strongly connected component as v. A *winner* is a vertex $v \in V$ which is a potential root or a door. Let's consider S, a strongly connected component of G_E, and p a strong articulation point in S; Δ^p is the set of the new strongly connected components obtained by the removal of p from S: $\Delta^p = \{S_i | S_i$ strongly connected component of

[2] the number of strongly connected components with no outgoing arcs

$S \setminus \{p\}\}$. Δ^p_{in} is the subset of Δ^p such that all paths from each of its strongly connected component to any winner of S go through p. Δ^p_{out} is the subset of Δ^p such that a path exists from each of its strongly connected component to a winner of S without going through p. Remark, $\Delta^p_{in} \bigoplus \Delta^p_{out} = \Delta^p$. Pruning is then performed according to three following rules:

1. If a sink component of G_E contains one single potential root r, then all the outgoing arc of r except the loop (r, r) are infeasible.
2. If a strongly connected component $C \subseteq G_E$ contains no potential root but one single door d, then all arcs $(d, v), v \in C$ are infeasible.
3. An outgoing arc (p, v) of a strong articulation point p of G_E that reaches a vertex v of a strongly connected component of Δ^p_{in} is infeasible.

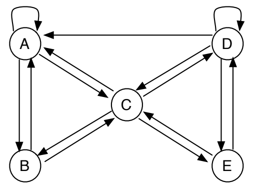
Rules 1 and 2 are obvious. Rule 3 basically means that enforcing such an arc would lead to some strongly connected components with no winners, thus sinks with no potential roots which is a contradiction.

About time complexity, pruning among rules 1 and 2 can easily be performed in linear time but when the paper was published, computing efficiently the strong articulation points of a digraph remained an open problem and the worst case time of the pruning procedure was thus $O(nm)$. In response to that claim, Italiano et. al. [5] recently showed an $O(m+n)$ worst case time complexity algorithm for computing strong articulation points of a digraph. This work enabled to fasten the pruning in practice. However, the theoretic time complexity remains $O(nm)$: rule 3 needs to withdraw strong articulation points one by one and compute new strongly connected components each time. The strongly connected components can be computed in $O(n+m)$ time using Tarjan's algorithm but there can be up to n strong articulation points, thus the total processing has a $O(n^2 + nm) = O(nm)$ worst case time complexity.

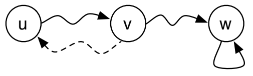
We will now show that the concept of strong articulation point is not well appropriate and propose a new formulation of the pruning conditions based on dominance relationship.

4 Linear Time Algorithm for the `tree` Constraint

The contribution of this paper relies on a new formulation of the filtering rule related to the strong articulation points. The first point to notice is that, given a strong articulation point p of a strongly connected component $SCC_i \subseteq G_E$, Δ^p_{in} may be empty (Figure 2(a)), thus the initial algorithm may perform several expensive and useless computations. The second important point is that the initial filtering algorithm does not require the concepts of doors, winners, strong articulation points and strongly connected components. Actually, their use, which implies paths in two directions, is not natural in our context because only a one way path from each node to a root is required. For instance, given three nodes u, v and w in V such that w is the unique potential root reachable from u. If every path from u to w requires v, then any path from v to u has to be forbidden (Figure 2(b)).



(a) A strong articulation point C such that $\Delta_{in}^{C} = \emptyset$



(b) Main pruning rule

Fig. 2. Structural pruning observations

The filtering rule proposed by the initial algorithm can be reformulated by: Any arc $(x, y) \in SCC_i \subseteq G_E, x \neq y$ is infeasible if and only if all paths from y to any winner of SCC_i go through x. However, as a winner in a strongly connected component is either a potential root, or a door that can thus lead to a potential root (each sink has at least one potential root), it can be rephrased: Any arc $(x, y) \in SCC_i \subseteq G_E, x \neq y$ is infeasible if and only if all paths from y to any potential root of G_E go through x. Moreover, assume (x, y) is an arc of the digraph G_E and there exists a path from y to x then x and y belong to the same strongly connected component, so the condition can be simplified in the following way: Any arc $(x, y) \in G_E, x \neq y$ is infeasible if and only if all paths from y to any potential root of G_E go through x. This condition is much closer to definition 1, it can be noted that it is also quite similar with the dominance definition: Let R be the set of potential roots, i.e. $R = \{v | v \in V, (v, v) \in A_E\}$. Let us consider the graph $G_{ES} = (V \cup s, A_E \cup S)$ where $s \notin V$ and $S = \bigcup_{r \in R}((r, s) \cup (s, r))$. Let the digraph G_{ES}^{-1} be the inverse graph of G_{ES} (obtained by reversing the orientation of arcs of G_{ES}). The previous definition can be transposed into: $(x, y) \in G_E, x \neq y$, *is infeasible if and only if* x *is a dominator of* y *in the flow graph* $G_{ES}^{-1}(s)$. The main interest is that algorithms do exist to find dominators of a flow graph in linear time [15, 3].

4.1 Feasibility and filtering conditions

We now consider a partially instantiated graph variable $GV = (G_E, G_K)$ that represents a subgraph of an input directed graph $G = (V, A)$. We have $G_E = (V, A_E)$, $G_K = (V, A_K)$ and $A_K \subseteq A_E \subseteq A$ (Section 2).

Proposition 2. *Given a partially instantiated graph variable* GV *of a digraph* G*, there exists a tree partition of* G *if and only if for each node* $v \in V$ *the two following conditions hold:*

1. $|\{(v, w) | (v, w) \in A_E\}| \geq 1$ *and* $|\{(v, w) | (v, w) \in A_K\}| \leq 1$
2. *there exists a potential root* $r \in V$ *such that a directed path from* v *to* r *exists in* G_E.

Proof. If there exists a node $v \in V$ such that $|\{(v,w) \mid (v,w) \in A_K\}| > 1$ then v has more than one successor in all solutions, if $|\{(v,w) \mid (v,w) \in A_E\}| < 1$ then v has no successors in A_E and thus in all solutions because $A_K \subseteq A_E$. Both cases are in contradiction with Proposition 1 thus cannot lead to any tree partitioning. If there exists a node $v \in V$ such that v can reach no potential root with a directed path in G_E then, as $A_K \subseteq A_E$, v cannot reach any root in any solution which is in contradiction with Proposition 1 and thus cannot lead to any tree partitioning.

Let us now suppose that conditions 1 and 2 are respected. Let us instantiate all potential roots $r \in V$, i.e. add all arcs $(r,r) \in A_E$ to A_K and delete all other outgoing arcs of r from A_E. At this step condition 2 still holds so for each node $v \in V$ there exists a potential root $r \in V$ such that a path from v to r exists in G_E. Let's add that path in G_K (by adding its arcs to A_K). The result of this procedure is an instantiated *Graph Variable* that is a tree partition of G. □

Remark 1. Condition 1 did not appear in previous models because they used integer variables which immediately ensured that property.

Proposition 3. *Given a partially instantiated graph variable GV of a digraph G, if there exists a tree partition of G then, an arc $(x,y) \in A_E$, $x \neq y$, does not belong to any solution if and only if one of the following conditions holds:*

1. *there exists a node $w \in V, w \neq y$, such that $(x,w) \in A_K$,*
2. *all directed paths in G_E from y to any potential root $r \in V$ go through x.*

Proof. Let x and w be two nodes in V such that $(x,w) \in A_K$ then w is a successor of x in every solution. Definition 1 implies that w is the unique successor of x, thus any arc $(x,y) \in A_E$ such that $y \neq w$ belongs to no solution. Let x and y be two distinct nodes of V such that all directed paths in G_E from y to any potential root $r \in V$ go through x and that $(x,y) \in A_E$. Then Proposition 1 implies that there will be a directed path from y to x in every solution. Using arc $(x,y) \in A_E$ would thus create a cycle which is in contradiction with definition 1 so (x,y) belong to no solution.

Let now suppose that there exists an arc $(x,y) \in A_E$ which belongs to no solution and such that conditions 1 and 2 are both false. Condition 1 is false if and only if $(x,y) \in A_K$ or no outgoing arc of x belongs to A_K. If $(x,y) \in A_K$ then all solutions contains arc (x,y) which is a contradiction so it can be supposed that no outgoing arc of x belongs to A_K. As condition 2 is false then at least one directed path exists in G_E from y to any potential root without going through x. If (x,y) is added to A_K then the two conditions of Proposition 2 hold, thus the graph variable GV still lead to at least one solution which is a contradiction. □

Proposition 4. *Given a partially instantiated graph variable GV of a digraph G, if each infeasible arc has been removed from GV, then an arc $(x,y) \in A_E$, $x \neq y$, belongs to all solutions if and only if all paths in G_E from x to any potential root go through the arc (x,y).*

Proof. Given an arc $(x, y) \in A_E$, if all paths from x to any potential root go through (x, y), as a path should exist from x to a potential root in each solution, (x, y) belongs to all solutions. Given an arc $(x, y) \in A_K$, then x has only one single successor y in G_E, otherwise all infeasible arcs have not been pruned (because each node should have exactly one successor). Thus all outgoing paths of x go through (x, y) and as the problem is feasible at least one path from x to any potential root exists. □

4.2 Filtering Algorithm

Keeping the previous notations about the graph variable GV, we assume that any graph is represented by two arrays of lists: successors and predecessors of nodes. The list of index i in the successors array represents the successors of the node $i \in V$. In order to make the complexity study easier, we introduce several basic notations: $n = |V|$, $m = |A|$, $m_E = |A_E|$ with $m_E \leq m$ (Section 2).

Proposition 5. *An $O(m + n)$ worst case time complexity algorithm exists to check whether GV can lead or not to a tree partition of G.*

Proof. Considering the list representation of the graph, condition 1 of Proposition 2 can be checked in $O(n)$ by computing the size of the list of successors of each node, once in G_E and once in G_K.

Let R be the set of potential roots, i.e. $R = \{v | v \in V, (v, v) \in A_E\}$. Let's consider the graph $G_{ES} = (V \cup s, A_E \cup S)$ where $s \notin V$, $A_E \cap S = \emptyset$ and $S = \bigcup_{r \in R}((r, s) \cup (s, r))$. Let the digraph G_{ES}^{-1} be the graph inverse of G_{ES} (obtained by reversing the orientation of arcs of G_{ES}). A simple Depth-First Search (DFS) exploration of G_{ES}^{-1} from node s will check whether each node $v \in V$ is reachable or not from s using directed paths in G_{ES}^{-1}. So it checks whether each node $v \in V$ can reach or not a potential root using a directed path in G_E. Thus it checks condition 2 of Proposition 2. The time complexity of a DFS of a graph of m_E arcs is $O(m_E)$. The total worst case time complexity of this algorithm is so $O(n + m_E)$. As $A_E \subseteq A$, $m_E \leq m$. Thus this algorithm runs in $O(n + m)$ worst case time complexity. □

Proposition 6. *If the graph variable GV, associated with the digraph G to partition, can lead to a tree partition of G, an $O(m + n)$ worst case time complexity algorithm exists to detect and remove all the arcs $(x, y) \in A_E$ that do not belong to any tree partition of G.*

Proof. In this context pruning an arc (x, y) consists in removing it from A_E. We will now describe such an algorithm, which relies upon two main steps that respectively correspond to conditions 1 and 2 of Proposition 3.

Condition 1: for each node $v \in V$, either v has one successor in G_K or v has no successor in G_K. If v has one successor w in G_K then the list of successors of v in G_E is cleared and (v, w) is put back into G_E. This is done in constant time so the whole complexity of step 1 is $O(n)$.

Condition 2: let us consider the graph G_{ES}^{-1} previously described. Then condition 2 of Proposition 3 to ensure that the arc $(x, y) \in G_E$ belongs to no solution is equivalent to "x dominates y in the flow graph $G_{ES}^{-1}(s)$". Several algorithms enable to find immediate dominators in a flow graph [15, 3] in $O(n + m_E)$ worst case time complexity. Let's compute I the dominance tree of the flow graph $G_{ES}^{-1}(s)$ with one of those algorithms.

Then a node $p \in V$ dominates $v \in V$ in G_{ES}^{-1} if and only if p is an ancestor of v in I. Such a query can be answered in constant time thanks to a $O(n)$ space and $O(n + m)$ time preprocessing. Let's create two n-size arrays *opening* and *closure*, perform a depth first search of I from s and record pre-order and post-order numbers of each node in respectively *opening* and *closure*. Then, p is an ancestor of v if and only if: $opening[p] < opening[v]$ and $closure[p] > closure[v]$. There are at most m_E requests (one for each arc) so the whole worst case time complexity of step two is $O(n + m)$. □

Proposition 7. *If the graph variable GV, associated with the digraph G to partition, can lead to a tree partition of G and if all its infeasible arcs have been pruned, an $O(m + n)$ worst case time complexity algorithm exists to add all the arcs $(x, y) \in A_E, x \neq y$, that belong to all tree partitions of G, into A_K.*

Proof. In this context enforcing an arc (x, y) consists in adding it to A_K. Let's consider the previously introduced flow graph $G_{ES}^{-1}(s)$. It should be noticed that the condition for enforcing an arc $(x, y) \in A_E$ of Proposition 4 is equivalent to: $(x, y) \in A_E$ *belong to all solutions if and only if* (y, x) *is an arc-dominator in* $G_{ES}^{-1}(s)$. Thus computing arc-dominators of $G_{ES}^{-1}(s)$ is all we have got left.

In [5] and [10], it is suggested to insert a fictive node inside each arc of the input graph and then compute dominators (in linear time). If a dominator is a fictive node, then the corresponding arc is an arc-dominator. Thus, the total processing time remains in $O(n+m)$ worst case time complexity. However, (y, x) is an arc-dominator of a flow graph $G(s)$ if and only if y is the immediate dominator of x in $G(s)$ and for each predecessor p of x such that $p \neq y$, x dominates p in $G(s)$. Thus we present an alternative method which we claim to be faster in practice and less space-consuming.

We assume that the pruning algorithm has been performed. Thus the dominance tree I, of G_{ES}^{-1}, is already computed and the preprocessing for ancestor relationships in I introduced in Proposition 6 has been done. A Depth First Search (DFS) exploration of G_{ES}^{-1} is then performed from the node s. For each encountered arc (y, x), such that $(x, y) \in I$ and $(x, y) \notin A_K$, for each predecessor p of x, a request to know whether x is an ancestor of p in I is computed. If one of those queries return false then (y, x) is not an arc-dominator of $G_{ES}^{-1}(s)$. Otherwise (x, y) can be enforced i.e. added into A_K. This algorithm computes $O(m)$ constant time queries. It is thus in $O(n + m)$ worst case time complexity. □

Remark 2. In our problem, each node has exactly one successor and all infeasible arcs are detected. Thus it is not useful to compute arcs that belong to all solutions explicitly: they will be deduced from the filtering algorithm. However in the general case, unlike integer variables, graph variables enable a node to have 0 or many outgoing arcs. Then the identification of arcs that belong to all solutions cannot be immediately deduced from pruning and thus provides important additional information.

Proposition 8. *Given the input graph G and an integer variable L, a partition of G into L trees, if one exists, can be found within $O(nm)$ worst case time complexity.*

Proof. Each of the n nodes must have exactly one successor. Then $n \leq m$. If the decision used in the propagation engine is *"enforce an arc $a \in A_E$, $a \notin A_K$"* then, as the pruning is complete the number of nodes in the tree search is $O(n)$. Plus it has just been shown that each propagation runs in $O(n + m)$ worst case time complexity. Thus the total solving time is $O(n(n + m)) = O(nm)$ worst case time complexity. □

4.3 Implementation details

The new `tree` constraint consists in the conjunction of 3 propagators: `OneSucc` enforces that each node must have exactly one successor; `Ntrees` controls the cardinality of the partition as described in [2]; `TreeProps` ensures a complete filtering over tree partitioning properties, which is the focus of the paper.

Algorithm 1 `TreeProps` propagator of the `tree` constraint

```
Require: two digraphs $G_E = (V_E, A_E)$, $G_E^{-1}$ s.t. $s \notin V_E$ is its unique source
Ensure: each arc of $G_E$ that does not belong to any solution has been removed
1: $T_E \leftarrow$ dominatorTree($G_E^{-1}, s$); {dominance tree of $G_E^{-1}$}
2: int[] opening, closure $\leftarrow$ ancestorPreProcess($T_E, s$); {pre/post-order of $T_E$}
3: for all node $v \in V_E$ do
4:     for all node $w \in A_E$.getSucc($v$) s.t. $w \neq v$ do
5:         if opening[$v$] < opening[$w$] $\wedge$ closure[$v$] > closure[$w$] then
6:             $A_E \leftarrow A_E \setminus \{(v, w)\}$;
7:         end if
8:     end for
9: end for
```

The structural filtering of `TreeProps` is based on the `dominatorTree`(G, s) algorithm which computes the immediate dominators of the flow graph $G(s)$ and return its dominance tree. It can be run in $O(n+m)$ worst case time [15, 3]. However, for practical reasons, the current implementation uses the Lengauer-Tarjan algorithm [6] which runs in $O(m\alpha(n, m))$ worst case time complexity, where $\alpha(n, m)$ is the inverse of the Ackermann function and thus grows very slowly. The function `ancestorPreProcess`(T, s) returns the pre-order and the post-order (starting from node s) labels of the nodes involved in the tree T. This can be done in $O(n)$ time if T involves n nodes.

5 Experimental study

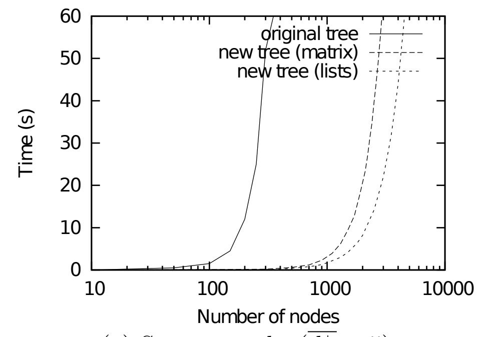
This section enables to compare several previously exposed models: the decomposition model of Section 3.1, the original `tree` constraint of Section 3.3 and the new linear time algorithm introduced by Section 4. Each algorithm consists in providing a tree partition of a randomly generated input graph. For practical interest, two cases of the new tree constraint have been tested. Both uses exactly the same constraint implementation but one uses graph variables with a matrix representation whereas the second one uses adjacency lists. Technically, the matrix representation uses `bitset` arrays instead of `boolean` matrix. For homogeneity reasons, all those experiments use the same branching strategy which consists in enforcing a randomly selected arc. All algorithms are implemented within the Choco open source Java Library. The source code is not included in the current distribution but is available on demand. The experiment has been performed on a Mac mini 4.1, with a 2.4 Ghz Intel core 2 duo processor and 3 Go of memory allocated to the Java Virtual Machine.

As the study focuses on structural filtering, the cardinality of the expected partition has not been restricted and input graphs were generated connected. We note d the density of the input graph and $\overline{d^+}$ the average outdegree of its nodes. We have $d = \frac{m}{n^2} = \frac{\overline{d^+}}{n}$. For each parameters combination $(n, \overline{d^+})$ thirty graphs have been randomly generated and partitioned into trees. Then, for each method, the number of solved instances and their mean solving time have been recorded. The time limit has been set to one minute. This enables to get information about the stability of those algorithms and about the relevance of our measures.

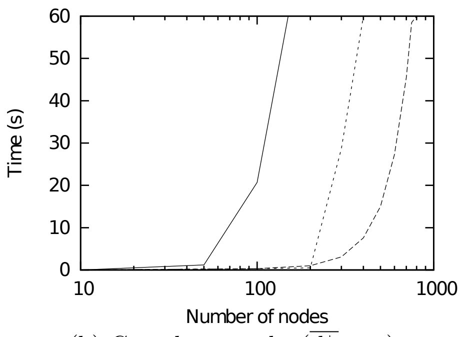
Instances		Decomp.		Original		Matrix		List	
n	$d+$	time	solved	time	solved	time	solved	time	solved
50	5	1.1	80	0.5	100	0	100	0	100
	20	0.1	100	1.3	100	0	100	0	100
	50	0.1	100	1.2	100	0	100	0	100
150	5	2.6	60	4.5	100	0	100	0	100
	20	3.1	80	11.3	100	0	100	0	100
	50	0.6	87	25.3	100	0.1	100	0.2	100
	150	2.8	100	-	0	0.3	100	1.4	100
300	5	0.1	20	51.6	100	0.1	100	0	100
	20	0.4	47	-	0	0.2	100	0.3	100
	50	1.7	53	-	0	0.5	100	1	100
	300	17.7	77	-	0	2.4	100	30	100

Table 1. Stability and performance study

Table 1 highlights that all approaches with a complete pruning are stable whereas the decomposition is unreliable. The computation time (column *time*) is provided in seconds, while the *solved* column denotes the percentage of solved instances. The decomposition is the worst choice for sparse graphs whereas it is better than the original `tree` constraint for dense graphs. This is due to the fact that the decomposition pruning is faster and that the denser the input graph, the higher the chance of any given arc to belong to at least one solution.



(a) Sparse graphs ($\overline{d^+} = 5$)



(b) Complete graphs ($\overline{d^+} = n$)

Fig. 3. Scalability and data structure

Figure 3 shows that the new `tree` constraint clearly outperforms the previous version by an important scaling factor. It can solve problems with up to 750 vertices when the graph is complete and up to 4500 vertices when the graph is sparse. Those results confirm the complexity of $O(nm)$ in theory (Proposition 8) and $O(nm\alpha(n,m))$ in practice (Section 4.3). Moreover they highlight the impact of the chosen data structure according to the input graph density. We observed a critical density $d_c = \frac{35}{n}$: when $d < d_c$ a list representation should be preferred whereas a matrix representation should be more relevant for denser graphs.

The last experiments we provide concerns scalability. For this purpose, the time limit is increased from one minute to two minutes and we observe the size of the graphs which can be treated within this time. In the case of sparse digraphs, a list representation in our algorithm improves the size of the largest treated digraph by 31% (up to 5900 nodes), while the original approach only allows to handle digraphs about 17% bigger (about 350 nodes). In the case of complete digraphs, a matrix representation in our algorithm improves the results by 20% (up to 900 nodes), while the original approach reaches 28% (about 160 nodes).

6 Conclusion

In this paper we have presented a non incremental linear time filtering algorithm that ensures generalized arc consistency for the `tree` constraint. Its correctness and worst case time complexity have been demonstrated and enforced by an experimental study. Even with an implementation in $O(m\alpha(n,m))$ of the filtering phase (due to the Lengauer-Tarjan algorithm) the constraint gains a mean scale factor of approximately ten. Moreover, two different types of data structures have been tested: matrix and adjacency lists. We experimentally showed that the lists representation clearly outperforms the matrix representation for sparse graphs and that this trend reverses when the input graph density grows enough. All those results are encouraging for further works as path partitioning. Also, we might work on incremental algorithms, however the dominance property seems too global to let us hope in significant improvements.

References

1. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global Constraint Catalog: Past, Present and Future. *Constraints*, 12(1):21–62, 2007.
2. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *CPAIOR'05*, volume 3524 of *LNCS*, pages 64–78, 2005.
3. A.L. Buchsbaum, H. Kaplan, A. Rogers, and J.R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20:1265–1296, 1998.
4. G. Dooms, Y. Deville, and P. Dupont. CP(graph): Introducing a graph computation domain in constraint programming. In *CP'05*, volume 3709 of *LNCS*, pages 211–225, 2005.
5. G. F. Italiano, L. Laura, and F. Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. In *COCOA'10*, volume 6508 of *LNCS*, 2010.
6. T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, 1(1), 1979.
7. J. Menana and S. Demassey. Sequencing and counting with the `multicost-regular` constraint. In *CPAIOR'09*, volume 5547 of *LNCS*, pages 178–192, 2009.
8. G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP'04*, volume 3258 of *LNCS*, pages 482–495, 2004.
9. G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP'04*, volume 3258 of *LNCS*, pages 482–495, 2004.
10. L. Quesada. *Solving constrained graph problems using reachability constraints based on transitive closure and dominators*. PhD thesis, Université Catholique de Louvain, 2006.
11. L. Quesada, P. van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL'06*, volume 3819 of *LNCS*, pages 73–87, 2006.
12. J-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.
13. J-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI'96*, pages 209–215, 1996.
14. J-C. Régin. Simpler and incremental consistency checking and arc consistency filtering algorithm for the weighted tree constraint. In *CPAIOR'08*, volume 5015 of *LNCS*, pages 233–247, 2008.
15. P.W. Lauridsen S. Alstrup, D. Harel and M. Thorup. Dominators in linear time. *SIAM J. Comput.*, 28(6):2117–2132, 1999.
16. M. Sellmann. Cost-based filtering for shortest path constraints. In *CP'03*, volume 2833 of *LNCS*, pages 694–708, 2003.
17. S. Sorlin and C. Solnon. A global constraint for graph isomorphism problems. In *CPAIOR'04*, volume 3011 of *LNCS*, pages 287–301, 2004.
18. R.E. Tarjan. Depth-first search and linear graph algorithms. In *SIAM J. Comput.*, volume 1, pages 146–160, 1972.
19. S. Zampelli, Y. Devilles, C. Solnon, S. Sorlin, and P. Dupont. Filtering for subgraph isomorphism. In *CP'07*, volume 4741, pages 728–742, 2007.