

Undirected Forest Constraints

Nicolas Beldiceanu¹, Irit Katriel² *, and Xavier Lorca¹

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

² BRICS**, University of Aarhus, Åbogade 34, Århus, Denmark. irit@daimi.au.dk

Abstract. We present two constraints that partition the vertices of an undirected n -vertex, m -edge graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ into a set of vertex-disjoint trees. The first is the *resource-forest* constraint, where we assume that a subset $R \subseteq \mathcal{V}$ of the vertices are *resource* vertices. The constraint specifies that each tree in the forest must contain at least one resource vertex. This is the natural undirected counterpart of the *tree* constraint [1], which partitions a directed graph into a forest of directed trees where only certain vertices can be tree roots. We describe a hybrid-consistency algorithm that runs in $\mathcal{O}(m + n)$ time for the *resource-forest* constraint, a sharp improvement over the $\mathcal{O}(mn)$ bound that is known for the directed case. The second constraint is *proper-forest*. In this variant, we do not have the requirement that each tree contains a resource, but the forest must contain only *proper* trees, i.e., trees that have at least two vertices each. We develop a hybrid-consistency algorithm for this case whose running time is $\mathcal{O}(mn)$ in the worst case, and $\mathcal{O}(m\sqrt{n})$ in many (typical) cases.

1 Introduction

Constraints that describe graph properties were considered from an early stage of constraint programming research. Some examples are the *Hamiltonian circuit* and *spanning tree* constraints of ALICE [2] that were later followed by the *cycle* [3] and *path* constraints [4], which were, respectively, introduced in later versions of CHIP [5] and Ilog Solver [6]. A more recent example is the *tree*(NTREE, VER) constraint [1], which receives an integer variable NTREE and a graph described by the vertex-list VER. Some of the vertices are specified as “possible roots” and the constraint determines that the graph consists of NTREE directed trees, each of which is rooted at a “possible root”.

A natural network design problem is the following. We are given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $R \subseteq \mathcal{V}$ is a set of vertices that correspond to a certain resource, e.g., a printer. The remaining vertices represent the tasks (clients/users). The problem is to cover the vertices of the graph with trees (networks) such that every tree contains at least one vertex from R (every network has a printer). We could replace each undirected edge by two anti-parallel directed arcs and then use the *tree* constraint which can be filtered in $\mathcal{O}(mn)$ time. However, undirected graphs are often much simpler than directed graphs. Indeed, we will show that the *resource-forest* constraint, the undirected counterpart of the *tree* constraint, can be filtered in $\mathcal{O}(m + n)$ time.

* Supported by the Danish Research Agency (grant # 272-05-0081).

** Basic Research in Computer Science, funded by the Danish National Research Foundation.

We then turn to another variant of the problem, the *proper-forest*(NTREE, VER) constraint which specifies that the graph is a forest of NTREE *proper trees* as defined by A. CAYLEY in 1889 [7]: A proper tree is a connected, cycle-free graph with at least two vertices. Note that with the *proper-forest* constraint the issue of resources does not exist (or, equivalently, all vertices are resource vertices). It can apply to the design of fault-tolerant networks, where each network needs to contain at least two computers so that each computer can back up the other. The *proper-forest* variant appears to be more complex than the *resource-forest* one; we show a filtering algorithm for *proper-forest* whose running time is $\mathcal{O}(mn)$ in the worst case, and is dominated by the complexity of determining which edges of the graph belong to at least one maximum cardinality matching. As we will see, the worst case occurs when the domain of NTREE is ground and contains a certain value. In all other cases, the algorithm's bottleneck is finding a maximum matching in the graph, which can be done in $\mathcal{O}(m\sqrt{n})$ time.

Since both constraints involve integer and set variables, our filtering algorithms achieve *hybrid-consistency*, which is a type of consistency suitable for this context, introduced by Bessi re *et al.* [8]. It will be formally defined in Section 2, but intuitively hybrid-consistency means that every integer variable is arc-consistent and every set variable is bound-consistent.

The rest of the paper is organized as follows. Section 2 provides the necessary background on constraint programming and graph theory. Section 3 introduces the *resource-forest* and *proper-forest* constraints. Sections 4 and 5, respectively, present filtering algorithms for the *resource-forest* and *proper-forest* constraints. Finally, Section 6 contains a summary of the known results on filtering tree-partitioning constraints.

2 Preliminaries

In this section we recall some of the constraint programming and graph theory terminology that we use in the rest of the paper.

Definition 1. An integer variable V ranges over a finite set of integers denoted by $\mathcal{D}(V)$. The extremal values in $\mathcal{D}(V)$ are denoted by $\min(V)$ and $\max(V)$.

Definition 2. The domain of a set variable V is a set of sets of integers. It is specified by its lower bound \underline{V} and its upper bound \overline{V} and contains all sets that contain \underline{V} and are contained in \overline{V} . When the set variable V is ground we have that $\underline{V} = \overline{V}$. The values in \underline{V} are the mandatory values of V and the values in $\overline{V} \setminus \underline{V}$ are its potential values.

Definition 3 (Hybrid-consistency [8]). A constraint \mathcal{C} defined on the integer variables V_1^d, \dots, V_l^d and the set variables V_{l+1}^s, \dots, V_n^s is hybrid-consistent iff:

1. For every pair (V^d, v) such that V^d is an integer variable of \mathcal{C} and $v \in \mathcal{D}(V^d)$, there exists at least one solution to \mathcal{C} in which V^d is assigned the value v .
2. For every pair (V^s, v) such that V^s is a set variable of \mathcal{C} , if $v \in \underline{V}^s$ then v belongs to the set assigned to V^s in all solutions to \mathcal{C} and if $v \in \overline{V}^s \setminus \underline{V}^s$ then v belongs to the set assigned to V^s in at least one solution and is excluded from this set in at least one solution.

Definition 4. Graph theoretic terms [9] :

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph. A path in \mathcal{G} is a sequence of vertices, such that every two consecutive vertices are joined by an edge. A path is simple if every vertex appears on it at most once. A bridge in \mathcal{G} is an edge $e \in \mathcal{E}$ whose removal increases the number of maximal connected components of \mathcal{G} . A matching in \mathcal{G} is a set $M \in \mathcal{E}$ of edges such that every vertex in \mathcal{V} is incident on at most one edge from M .

3 The *resource-forest* and *proper-forest* Constraints

In this section, we define and motivate the *resource-forest* and the *proper-forest* constraints, introduce their corresponding graphs, define them formally and provide examples that illustrates the semantics of the constraint as well as the problem of filtering them to hybrid-consistency.

In many graph-partitioning problems, the vertex set of the graph is the union of a set of *resource* vertices and a set of *task* vertices. Independently of the pattern used to cover the graph, this distinction between the two types of vertices comes from the need that each partition has to contain at least one resource vertex. This distinction between resource and task vertices was already introduced in the *cycle* constraint [3]. An example of application for the *cycle* constraint is the vehicle routing problem which consists in allocating a set of trucks (resources) to deliver goods to a set of shops (tasks). The *resource-forest* constraint, on the other hand, can be used to model the problem of allocating hardware resources in a network. Here, a resource represents a piece of hardware (e.g., a printer) and a task represents a computer. The solution (forest) is a network in which each computer is connected with at least one printer.

In 1889, A. CAYLEY [7] introduced the definition of a *tree* as a connected graph without cycles which contains at least two vertices. We will call Cayley's tree a *proper tree*. A *proper forest*, then, is a set of proper trees. The *proper-forest* constraint partitions the vertices of an undirected graph into a set of vertex-disjoint proper trees.

Formally, each of the *resource-forest*(NTREE, VER) and *proper-forest*(NTREE, VER) constraints is defined on an integer variable NTREE and an array VER which is essentially an adjacency-list representation of a graph. Each item $v \in \text{VER}$ has the following attributes, which complete the description of the graph:

- I is an integer between 1 and n , which can be interpreted as the *label* of v .
- N is a set variable whose elements are integers (vertex labels) between 1 and n . The lower and upper bounds of N can respectively be interpreted as the *set of mandatory neighbors* and the *set of mandatory or potential neighbors* of v .
- R (only for the *resource-forest* constraint) is a boolean flag which is true if the vertex is a resource vertex and false if it is a task vertex.

Notation: For each $1 \leq i \leq n$, $\text{VER}[i]$ is the i -th item of the VER collection, while $\text{VER}[i].\text{I}$, $\text{VER}[i].\text{N}$, and $\text{VER}[i].\text{R}$, respectively, denote the I, N and R attributes of $\text{VER}[i]$.

When speaking of global constraints, it is often convenient to reason about a graph that models the constraint rather than directly about the constraint (see, e.g., the *cycle* [3], *path* [10, 4], and *alldifferent* [11] constraints). In the case of the *resource-forest*

and *proper-forest* constraints, the graph model is obvious: It is the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which the vertices represent the elements of VER and the edges represent the neighborhood relations between them. Each edge of the graph has a *type* (solid or dotted) which indicates whether it represents a mandatory (solid) or a potential (dotted) neighborhood relation.

Since it can be easily achieved by a linear-time preprocessing step, we will assume in the rest of this paper that the associated graph does not contain loops and that the N sets of the vertices are symmetric, i.e., $i \in \text{VER}[j].\underline{N} \Leftrightarrow j \in \text{VER}[i].\underline{N}$ (in this case we will say that i and j are *mandatory neighbors*) and $i \in \text{VER}[j].\overline{N} \Leftrightarrow j \in \text{VER}[i].\overline{N}$ (in this case, if i and j are not mandatory neighbors then they are *possible neighbors*). Note that the preprocessing step may find that the constraint has no solution. This can happen if $i \in \text{VER}[i].\underline{N}$ (there is a mandatory loop) or $\exists i, j : i \in \text{VER}[j].\underline{N} \wedge j \notin \text{VER}[i].\overline{N}$ (i is a mandatory neighbor of j but j is not a possible neighbor of i). Formally, the graph associated with a *resource-forest* or a *proper-forest* constraint is defined as follows.

Definition 5. For a resource-forest(NTREE, VER) or a proper-forest(NTREE, VER) constraint, the associated graph is the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{v_i : i \in [1, n]\}$ and $(v_i, v_j) \in \mathcal{E}$ iff $i \in \text{VER}[j].\overline{N} \wedge j \in \text{VER}[i].\overline{N}$. We distinguish between solid and dotted edges: The edge $(v_i, v_j) \in \mathcal{E}$ is solid if i and j are mandatory neighbors and dotted if i and j are potential neighbors. Finally, we denote the number of edges in the graph, $|\mathcal{E}|$, by m .

In the case of the resource-forest constraint, we distinguish between resource vertices and task vertices; the set R of resource vertices is $\{v_i : \text{VER}[i].\text{R} = \text{true}\}$. All vertices in $\mathcal{V} \setminus \text{R}$ are task vertices.

The *resource-forest* constraint specifies that its associated graph is a forest where each tree contains at least one resource and the *proper-forest* constraint specifies that its associated graph is a proper forest. Formally:

Definition 6. A ground resource-forest(NTREE, VER) constraint is satisfied iff the following conditions hold:

- (1) $\forall i \in [1, n] : \text{VER}[i].\text{I} = i$,
- (2) $\forall i, j \in [1, n] : i \in \text{VER}[j].\text{N} \Leftrightarrow j \in \text{VER}[i].\text{N}$ (i.e., the neighborhood relation is symmetric),
- (3) The associated graph \mathcal{G} consists of NTREE maximal connected components such that each component contains at least one vertex from R and does not contain any cycles.

Definition 7. A ground proper-forest(NTREE, VER) constraint is satisfied iff the following conditions hold:

- (1) $\forall i \in [1, n] : \text{VER}[i].\text{I} = i$,
- (2) $\forall i, j \in [1, n] : i \in \text{VER}[j].\text{N} \Leftrightarrow j \in \text{VER}[i].\text{N}$ (i.e., the neighborhood relation is symmetric),
- (3) The associated graph \mathcal{G} is a forest of NTREE (vertex-disjoint) proper trees.

The following example will be used throughout the paper.

Example 1. Part (A) of Figure 1 shows the input graph \mathcal{G} , where the mandatory edges are solid and the rest are dotted. Parts (B) and (C) of the figure show two possible solutions to the *resource-forest* constraint on this graph, one with two trees and the other with three trees. Parts (B) and (D) show two solutions to the *proper-forest* constraint on this graph, with two and seven proper trees, respectively.

A hybrid-consistency algorithm for the *resource-forest* constraint on \mathcal{G} should discover that regardless of the contents of the domain of NTREE, the edge marked by M_r , i.e., the edge (6, 8), is mandatory and that the edge (5, 7) (marked by F_r) is forbidden. Furthermore, it should set the domain of NTREE to be the intersection of its previous value with $\{2, 3\}$. If, in the input, $\mathcal{D}(\text{NTREE}) = \{2\}$, the algorithm should also discover that the edge marked by J_r , i.e., the edge (13, 15), is mandatory. Section 4 will justify this pruning.

A hybrid-consistency algorithm for the *proper-forest* constraint on \mathcal{G} should discover that the edge (13, 15) (marked by M_p) is mandatory and that the edge (5, 7) (marked by F_p) is forbidden. Next, it should set the domain of NTREE to be the intersection of its previous value with $\{2, 3, 4, 5, 6, 7\}$. If, in the input, $\mathcal{D}(\text{NTREE}) = 2$, the algorithm should discover that the edge marked by J_p , i.e., the edge (6, 8), is mandatory. Finally, if $\mathcal{D}(\text{NTREE}) = \{7\}$ in the input, the algorithm should discover that the edges marked by I_p , i.e., (2, 3), (3, 5), (4, 7), (6, 8), (11, 13), and (12, 14), are forbidden. Section 5 will justify this pruning.

Before we can describe the filtering algorithms for the *resource-forest* and *proper-forest* constraints, we need to define the mandatory graph $\mathcal{G}_{\text{TRUE}}$ and the possible graph $\mathcal{G}_{\text{MAYBE}}$ associated with a graph \mathcal{G} . An example appears in Figure 2.

Definition 8. (Mandatory graph) *Given a resource-forest or proper-forest constraint and its associated graph \mathcal{G} , the graph $\mathcal{G}_{\text{TRUE}}$ contains all edges that must be in the forest. Formally, $\mathcal{G}_{\text{TRUE}} = (\mathcal{V}, \mathcal{E}_{\text{TRUE}})$, where $\mathcal{E}_{\text{TRUE}}$ is the set of solid edges in \mathcal{G} .*

Definition 9. (Possible graph) *Given a resource-forest or proper-forest constraint and its associated graph \mathcal{G} , the graph $\mathcal{G}_{\text{MAYBE}}$ contains the subgraph induced by the vertices that are not incident on mandatory edges. Formally, $\mathcal{G}_{\text{MAYBE}} = (\mathcal{V}_{\text{MAYBE}}, \mathcal{E}_{\text{MAYBE}})$ where $\mathcal{V}_{\text{MAYBE}}$ contains all vertices that are isolated in $\mathcal{G}_{\text{TRUE}}$ and $\mathcal{E}_{\text{MAYBE}} = \mathcal{E} \cap (\mathcal{V}_{\text{MAYBE}} \times \mathcal{V}_{\text{MAYBE}})$.*

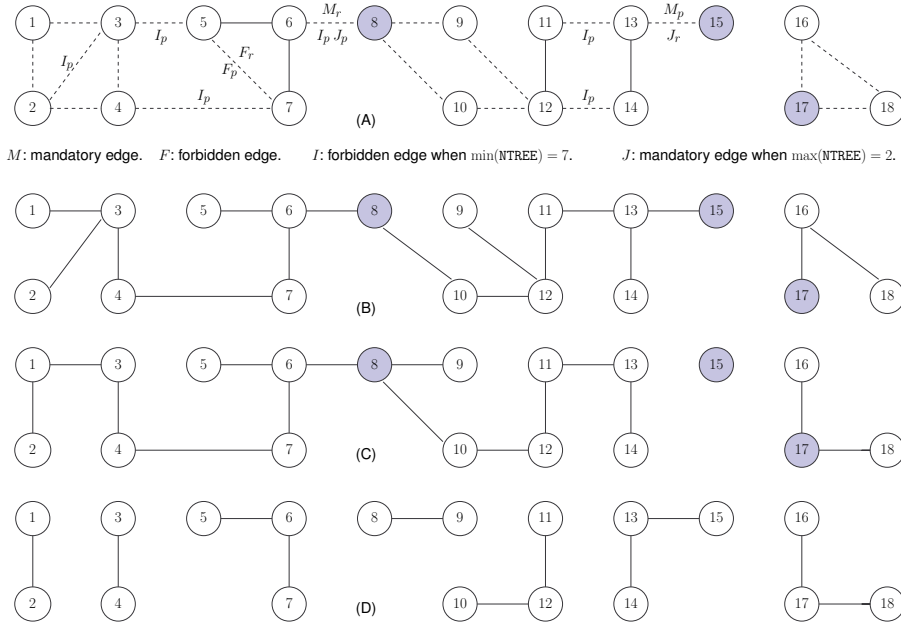
4 Filtering the *resource-forest* Constraint

4.1 Checking Feasibility of the *resource-forest* Constraint

Theorem 1 specifies necessary and sufficient conditions for the existence of a solution to a *resource-forest* constraint. The first two conditions ensure that it is possible to partition the graph into a forest with a resource in every tree and the third ensures that the number of trees in the forest is within the domain of NTREE.

Theorem 1. *There is a solution to the resource-forest(NTREE, VER) constraint iff the following conditions hold:*

- (1) $\mathcal{G}_{\text{TRUE}}$ does not contain any cycles.



M : mandatory edge. F : forbidden edge. I : forbidden edge when $\min(\text{NTREE}) = 7$. J : mandatory edge when $\max(\text{NTREE}) = 2$.

Fig. 1. (A) An undirected graph with 3 (grayed) resource vertices (B) a solution with 2 trees for the *resource-forest* and *proper-forest* constraints (C) a solution with 3 trees for the *resource-forest* constraint (D) a solution with 7 proper trees for the *proper-forest* constraint. Notice that each kind of edges (M , F , I and J) are indicated by p in the case of the *proper-forest* and by r in the case of the *resource-forest*.

- (2) Every maximal connected component of \mathcal{G} contains at least one resource vertex.
- (3) $\mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] \neq \emptyset$, where MINTREE is the number of maximal connected components in \mathcal{G} and MAXTREE is the number of maximal connected components of $\mathcal{G}_{\text{TRUE}}$ that contain at least one resource vertex.

Proof. Sufficiency: To prove that the three conditions are sufficient, we assume that they hold and show that for every value $k \in [\text{MINTREE}, \text{MAXTREE}]$, we can construct a spanning forest of \mathcal{G} with k trees, each of which contains a resource vertex.

Case 1: $k = \text{MAXTREE}$. Let $\mathcal{T} = \{C_1, \dots, C_p\}$ be the maximal connected components of $\mathcal{G}_{\text{TRUE}}$. By definition, exactly MAXTREE of them contain at least one resource vertex. By Condition (2), every component which does not contain a resource vertex is connected by a path of \mathcal{G} to a component which does. To obtain a solution of k trees, we merge every component that does not contain any resource vertices with one that does, and output a spanning tree of each component.

Case 2: $k < \text{MAXTREE}$. We first construct a forest of MAXTREE trees as in Case 1 and then merge trees until there are k trees: While there are too many trees, select two trees which are connected by an edge e and merge them by including e in the forest. Since MINTREE is the number of maximal connected components in \mathcal{G} , as long as $k > \text{MINTREE}$ we are guaranteed to find two trees that can be merged.

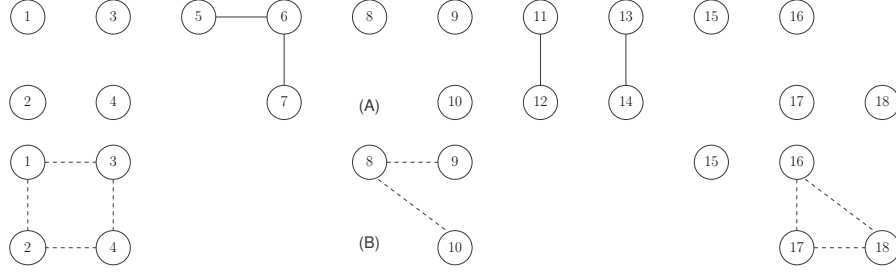


Fig. 2. The graphs (A) \mathcal{G}_{TRUE} and (B) \mathcal{G}_{MAYBE} associated with the graph of Figure 1, Part (A).

Necessity: Clearly, if \mathcal{G}_{TRUE} contains a cycle, the solution cannot be a forest. If $\mathcal{D}(NTREE) \cap [\text{MINTREE}, \text{MAXTREE}] = \emptyset$ then we have $\max(NTREE) < \text{MINTREE}$ or $\min(NTREE) > \text{MAXTREE}$. In either case, the constraint is infeasible: We cannot create less than MINTREE trees because a tree must be connected. To see that we cannot create more than MAXTREE trees, note a connected component of \mathcal{G}_{TRUE} cannot be broken, so every component can contribute at most one tree. Furthermore, the vertices of a component that does not contain a resource vertex must belong to the same tree as the vertices of a component that does contain a resource vertex. In other words, a component cannot contribute a tree to the forest if it does not contain a resource vertex. \square

4.2 Hybrid-consistency of the *resource-forest* Constraint

Figure 3 shows the algorithm for filtering a *resource-forest* constraint to hybrid-consistency. First, it verifies that the constraint has at least one solution, using the characterization of Theorem 1. Lines 3 to 7 prune with respect to the fact that a solution must be a forest (while ignoring the cardinality of the forest and the condition on the resources). In Line 7 the algorithm removes any dotted edge (u, v) where u and v are connected by solid edges; since the solid edges must be in the solution, this dotted edge would create a cycle (e.g., the edge $(5, 7)$ in Part (A) of Figure 1). In Lines 8 to 10 it identifies dotted edges which must be in the solution because removing them would separate one or more vertices from the resources, and makes them solid (e.g., the edge $(6, 8)$ in Part (A) of Figure 1). Line 11 narrows the domain of $NTREE$.

Lines 12 to 17 are executed only when the domain of $NTREE$ is ground. In this case, the number of trees in a solution is fixed and if it is equal to MINTREE (as defined in the statement of Theorem 1), all bridges of \mathcal{G} are mandatory and are turned solid, because otherwise the number of connected components of the graph, and therefore also the number of trees in any solution, is strictly larger than MINTREE (e.g., the edge $(13, 15)$ in Part (A) of Figure 1). On the other hand, if the value of $NTREE$ is equal to MAXTREE , then every maximal connected component of \mathcal{G}_{TRUE} which contains a resource should contribute a tree to the solution, so a dotted edge between two such components must not be in the forest and is removed.

1. **if** the constraint has no solution (see Theorem 1) **then**
2. report failure and exit;
3. Compute the maximal connected components (CCs) of \mathcal{G}_{TRUE} .
4. **foreach** $v \in \mathcal{V}$ **do**
5. $C(v) \leftarrow$ the CC of \mathcal{G}_{TRUE} that contains v ;
6. **foreach** dotted edge $(u, v) \in \mathcal{E}$ **do**
7. **if** $C(u) = C(v)$ **then** remove (u, v) from the graph;
8. **foreach** dotted edge e **do**
9. **if** removing e creates a CC of \mathcal{G} without resource vertices **then**
10. Turn e into a solid edge;
11. $\mathcal{D}(\text{NTREE}) \leftarrow \mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}]$;
12. **if** $\mathcal{D}(\text{NTREE}) = \{\text{MINTREE}\}$ **then**
13. **foreach** dotted edge e that is a bridge of \mathcal{G} **do**
14. Turn e into a solid edge;
15. **if** $\mathcal{D}(\text{NTREE}) = \{\text{MAXTREE}\}$ **then**
16. **foreach** dotted edge $e = (u, v)$ where $C(u) \neq C(v)$ **and**
 both $C(u)$ and $C(v)$ contain a resource vertex **do**
17. Remove e from the graph;

Fig. 3. A hybrid-consistency algorithm for the *resource-forest* constraint.

4.3 Correctness

In the full version of this paper, we prove that the algorithm achieves hybrid-consistency by showing that:

1. We did not remove an edge from the graph or a value from NTREE that belong to a solution.
2. Every remaining edge in \mathcal{G} and value in $\mathcal{D}(\text{NTREE})$ participates in at least one solution, and every remaining dotted edge is excluded from at least one solution.
3. Every edge that we turned from dotted to solid participates in all solutions.

4.4 Complexity

The steps of the algorithm excluding Lines 8 to 10 require cycle detection, computing maximal connected components and identifying bridges, all of which can be done in linear time [12, p.18]. We now show that Lines 8 to 10 also take linear time. Clearly, an edge whose removal creates a maximal CC without a resource is a bridge of \mathcal{G} . But not all bridges have this property. We create a reduced graph \mathcal{H} by contracting every biconnected component of \mathcal{G} into a single vertex. The graph \mathcal{H} is a tree whose edges are exactly the bridges of \mathcal{G} . We will say that a vertex of \mathcal{H} is a resource vertex if one of the vertices of \mathcal{G} that were contracted into it is a resource vertex, i.e., if the biconnected component it represents has a resource. We need to identify which of the edges of \mathcal{H} are edges whose removal would create a connected component of \mathcal{H} without resources. In other words, we have reduced our problem to the same problem on trees. This holds for an edge if one of its endpoints is the root of a subtree without resources. We select an

arbitrary resource vertex of \mathcal{H} and perform a DFS traversal of \mathcal{H} starting at this vertex. Whenever we backtrack from a vertex v , we communicate to its parent p whether a resource was encountered in the subgraph rooted at v . If not, then the edge (p, v) is turned into a solid edge (if it is not solid already).

Thus, we have shown:

Theorem 2. *The algorithm of Figure 3 filters the resource-forest constraint to hybrid-consistency in $\mathcal{O}(m + n)$ time.*

5 Filtering the *proper-forest* Constraint

5.1 Checking Feasibility of the *proper-forest* Constraint

Theorem 3 specifies the conditions for the existence of a solution to a *proper-forest* constraint. The first two conditions ensure that it is possible to partition the graph into a proper forest and the third ensures that the number of proper trees in the proper forest is within the domain of NTREE.

Theorem 3. *There is a solution to the proper-forest(NTREE, VER) constraint iff the following conditions hold:*

- (1) \mathcal{G} does not have isolated vertices,
- (2) \mathcal{G}_{TRUE} does not contain any cycles,
- (3) $\mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] \neq \emptyset$, where MINTREE is the number of maximal connected components in \mathcal{G} and MAXTREE is the number of maximal connected components of cardinality at least two in \mathcal{G}_{TRUE} plus the cardinality of a maximum cardinality matching in \mathcal{G}_{MAYBE} .

Proof. Sufficiency: To prove that the three conditions are sufficient, we assume that they hold and show that for every value $k \in [\text{MINTREE}, \text{MAXTREE}]$, we can construct a spanning forest of \mathcal{G} with k proper trees. We begin with $k = \text{MAXTREE}$ and proceed to an arbitrary $k \in [\text{MINTREE}, \text{MAXTREE}]$.

- Let $\mathcal{T} = \{T_1, \dots, T_p\}$ be a maximum spanning forest of \mathcal{G}_{TRUE} , i.e., each T_i is an isolated vertex from \mathcal{G}_{TRUE} or a spanning tree of a maximal connected component of \mathcal{G}_{TRUE} . Observe that a tree T_i of cardinality one is also a vertex of \mathcal{G}_{MAYBE} .
- To construct a spanning forest of cardinality MAXTREE, compute a maximum cardinality matching \mathcal{M} in \mathcal{G}_{MAYBE} and modify \mathcal{T} as follows:
 - By definition of \mathcal{G}_{MAYBE} , each matching edge connects two singletons T_i and T_j of \mathcal{T} . Merge them into a tree of cardinality two.
 - For every vertex $u \in \mathcal{V}_{MAYBE}$, corresponding to a tree T_i of cardinality one which is unmatched by \mathcal{M} , select a neighbor v of u and include the edge (u, v) in the spanning forest. In other words, merge the tree T_i with the tree T_v to which v belongs. Condition 1 guarantees that this is possible. It is easy to see that the forest consists of exactly MAXTREE trees.

- If $k < \text{MAXTREE}$, merge proper trees until there are k proper trees: While there are too many proper trees, select two proper trees that are connected by an edge e from $\mathcal{G}_{\text{MAYBE}}$ and merge them by including e in the proper forest. Since MINTREE is the number of maximal connected components in \mathcal{G} , as long as $k > \text{MINTREE}$ we are guaranteed to find two proper trees that can be merged.

Necessity: It remains to show that all three conditions are necessary. Clearly, if \mathcal{G} contains an isolated vertex v , then v does not belong to a subgraph of \mathcal{G} which is a proper tree and if $\mathcal{G}_{\text{TRUE}}$ contains a cycle, the solution must contain a cycle so it cannot be a proper forest. Finally, if $\mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] = \emptyset$ then we have $\max(\text{NTREE}) < \text{MINTREE}$ or $\min(\text{NTREE}) > \text{MAXTREE}$. In either case, the constraint does not have a solution: We cannot create less than MINTREE proper trees because a proper tree must be connected. To see that we cannot create more than MAXTREE proper trees, note that the number of proper trees that $\mathcal{G}_{\text{TRUE}}$ can contribute is at most the number of connected components it has (we cannot break a connected component of $\mathcal{G}_{\text{TRUE}}$) and that a vertex of $\mathcal{G}_{\text{MAYBE}}$ can either form a new proper tree with another vertex from $\mathcal{G}_{\text{MAYBE}}$, or be merged into a previously existing proper tree (and not contribute to the tree-count). Clearly, a maximum cardinality matching contributes the largest possible number of proper trees from $\mathcal{G}_{\text{MAYBE}}$. \square

5.2 Hybrid-consistency of the *proper-forest* Constraint

Figure 4 shows the algorithm for filtering a *proper-forest* constraint to hybrid-consistency. First, it verifies that the constraint has at least one solution, using the characterization of Theorem 3, and exits if the constraint is inconsistent. Lines 3 to 7 and 18 to 19 prune with respect to the fact that a solution must be a proper forest (while ignoring the cardinality of the forest). In Line 7 the algorithm removes any dotted edge (u, v) (e.g., the edge (5, 7) in Part (A) of Figure 1) where u and v are connected by a path of solid edges; since the solid edges must be in the solution, this dotted edge would create a cycle. Line 19 identifies dotted edges which must be in the solution because removing them would isolate a vertex, and makes them solid (e.g., the edge (13, 15) in Part (A) of Figure 1). Line 8 narrows the domain of NTREE .

Lines 9 to 17 are executed only when the domain of NTREE is ground. In this case, the number of trees in a solution is fixed and if it is equal to either MINTREE or MAXTREE (these values are defined in the statement of Theorem 3), more filtering is possible: If $\mathcal{D}(\text{NTREE}) = \{\text{MINTREE}\}$ then all bridges of \mathcal{G} are mandatory and are turned solid, because otherwise the number of connected components of the graph, and therefore also the number of trees in any solution, is strictly larger than MINTREE . In the example in Part (A) of Figure 1, the edge (6, 8) (which is marked by J_p) is mandatory when $\mathcal{D}(\text{NTREE}) = \{2\}$. If $\mathcal{D}(\text{NTREE}) = \{\text{MAXTREE}\}$ then three sets of edges are forbidden and are removed from the graph; we will show that including any one of these edges in a solution would reduce the number of trees we can construct to strictly less than MAXTREE . For example, if $\mathcal{D}(\text{NTREE}) = \{7\}$, the edges marked by I_p in Part (A) of Figure 1 are removed: (11, 13) and (12, 14) in Line 15, (2, 3) in Line 16, and (3, 5), (4, 7) and (8, 6) in Line 17.

1. **if** the constraint has no solution (see Theorem 3) **then**
2. report failure and exit;
3. Compute the maximal connected components (CCs) of \mathcal{G}_{TRUE} .
4. **foreach** $v \in \mathcal{V}$ **do**
5. $C(v) \leftarrow$ the CC of \mathcal{G}_{TRUE} that contains v ;
6. **foreach** dotted edge $(u, v) \in \mathcal{E}$ **do**
7. **if** $C(u) = C(v)$ **then** remove (u, v) from the graph;
8. $\mathcal{D}(\text{NTREE}) \leftarrow \mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}]$;
9. **if** $\mathcal{D}(\text{NTREE}) = \{\text{MINTREE}\}$ **then**
10. **foreach** dotted edge (u, v) that is a bridge of \mathcal{G} **do**
11. Turn (u, v) into a solid edge;
12. **if** $\mathcal{D}(\text{NTREE}) = \{\text{MAXTREE}\}$ **then**
13. **foreach** dotted edge (u, v) **do**
14. remove (u, v) from \mathcal{G} if one of the following holds:
15. a. $|C(u)| > 1$, $|C(v)| > 1$, and $C(u) \neq C(v)$.
16. b. $(u, v) \in \mathcal{E}_{MAYBE}$ but does not belong to any maximum matching in \mathcal{G}_{MAYBE} .
17. c. $|C(u)| > 1$ and v is saturated in every maximum cardinality matching of \mathcal{G}_{MAYBE} .
18. **foreach** dotted edge $(u, v) \in \mathcal{E}$ **do**
19. **if** u is a leaf in G **then** turn (u, v) into a solid edge;

Fig. 4. A hybrid-consistency algorithm for the *proper-forest* constraint.

5.3 Correctness

To prove correctness of the algorithm, we will show that:

1. We did not remove an edge from the graph or a value from NTREE that belong to a solution (Lemma 1).
2. Every remaining edge in \mathcal{G} and value in $\mathcal{D}(\text{NTREE})$ participates in at least one solution, and every remaining dotted edge is excluded from at least one solution (Lemma 2).
3. Every edge that we turned from dotted to solid participates in all solutions (Lemma 3).

Lemma 1. *The algorithm in Figure 4 never removes an edge from the graph or a value from $\mathcal{D}(\text{NTREE})$ that belongs to a solution.*

Proof. Let (u, v) be an edge that was removed by the algorithm and assume that there is a solution S that contains (u, v) . If (u, v) was removed in Line 7, then there is a path of solid edges from u to v , and these edges are also in the solution. But then the forest S contains a cycle, a contradiction. So we must assume that (u, v) was removed in Lines 13 to 17. In this case, we know that the number of trees in S is equal to MAXTREE (the number of CCs of \mathcal{G}_{TRUE} of cardinality at least two plus the cardinality of a maximum matching in \mathcal{G}_{MAYBE} .) We will show that if there still is a solution S' when the constraint is on the graph \mathcal{G}' , which is obtained from \mathcal{G} by turning (u, v) into a solid edge, then this violates Theorem 3 because S' has more than MAXTREE'

trees (where $\text{MAXTREE}'$ is the MAXTREE value for \mathcal{G}'). If (u, v) was removed in Line 15, then $\text{MAXTREE}' = \text{MAXTREE} - 1$ because u and v are not in $\mathcal{G}_{\text{MAYBE}}$ and two non-trivial connected components of $\mathcal{G}_{\text{TRUE}}$ have been merged. The solution $S' = S$ has MAXTREE (i.e., $> \text{MAXTREE}'$) trees. If (u, v) was removed in Line 16, then u and v form a size-2 maximal connected component in $\mathcal{G}'_{\text{TRUE}}$. This increases MAXTREE by one. On the other hand, the cardinality of a maximum matching in $\mathcal{G}'_{\text{MAYBE}} = \mathcal{G}_{\text{MAYBE}} \setminus \{u, v\}$ is two less than that of $\mathcal{G}_{\text{MAYBE}}$, because otherwise (u, v) belongs to a maximum matching in $\mathcal{G}_{\text{MAYBE}}$. So $\text{MAXTREE}' = \text{MAXTREE} - 1$ and $S' = S$ is a solution with MAXTREE trees. Finally, if (u, v) was removed in Line 17, then turning (u, v) into a solid edge inserts v into the CC of u in $\mathcal{G}'_{\text{TRUE}}$. Since v is not in $\mathcal{G}'_{\text{MAYBE}}$, the cardinality of a maximum matching in $\mathcal{G}'_{\text{MAYBE}}$ is one less than that of $\mathcal{G}_{\text{MAYBE}}$ (otherwise $\mathcal{G}_{\text{MAYBE}}$ has a maximum matching in which v is not saturated, a contradiction). Again, we get that $\text{MAXTREE}' = \text{MAXTREE} - 1$ and $S' = S$ is a solution with MAXTREE trees. If the algorithm removes a useful value from $\mathcal{D}(\text{NTREE})$, this clearly contradicts Theorem 3. \square

Lemma 2. *After applying the algorithm of Figure 4, every remaining edge in \mathcal{G} and value in $\mathcal{D}(\text{NTREE})$ participates in at least one solution, and every remaining dotted edge is excluded from at least one solution.*

Proof. We have already shown in the proof of Theorem 3 that every value in $[\text{MINTREE}, \text{MAXTREE}]$ participates in a solution. We now show that every remaining edge (u, v) belongs to the forest in at least one solution. First, we construct a solution S with MAXTREE trees as before. If (u, v) belongs to the forest, we are done. Otherwise, let $S' = S \cup (u, v)$. If S' is not a solution to the constraint, it can be either because it is not a forest or because the number of trees in S' is not in $\mathcal{D}(\text{NTREE})$. If it is not a forest, it is because inserting (u, v) creates a cycle, but in this case (u, v) should have been removed in Line 7. So the number of trees, which is $\text{MAXTREE} - 1$, is not in $\mathcal{D}(\text{NTREE})$. If there is a value in $\mathcal{D}(\text{NTREE})$ which is smaller than the number of trees in S' , we can merge trees as in the proof of Theorem 3 until we have a solution. Otherwise, it must be that $\mathcal{D}(\text{NTREE}) = \{\text{MAXTREE}\}$. But in this case, (u, v) should have been removed in Line 15.

It remains to show that every dotted edge is excluded from at least one solution. Let (u, v) be a dotted edge. We have shown above that there exists a solution S that uses (u, v) . Let $S' = S \setminus \{(u, v)\}$. If S' is a solution for the *proper-forest* constraint, we are done. Assume, then, that S' is not a solution. This can be either because it contains an isolated vertex or because it consists of too many trees.

Assume that the removal of (u, v) created a singleton tree with the vertex u . Since (u, v) was not turned into a solid edge in Line 19, we know that u has another neighbor $n_u \neq v$, to which it is linked by a dotted edge. If v did not become a singleton tree or the number of trees was strictly larger than MINTREE , we can merge u into a neighboring tree and obtain a new solution that does not use the edge (u, v) .

However, if the number of trees was exactly MINTREE and both u and v were isolated, merging each of them into a pre-existing tree would leave us with $\text{MINTREE} - 1$ trees. Fortunately, this case is not possible. Indeed, assume that it has occurred. We know that n_u belongs to a tree of the solution which contains at least two vertices and

which does not contain u or v . So the connected component of u (and v) in \mathcal{G} contributes at least two trees to the solution, and the solution must have more than MINTREE trees.

Finally, assume that after the removal of (u, v) we do not have any singleton trees, but we do have one tree too many. If possible, merge two trees by a dotted edge other than (u, v) . If this is not possible, then it is because the number of trees is equal to the number of connected components in $\mathcal{G} \setminus \{e\}$, i.e., to MINTREE. If (u, v) was a bridge, it would have turned into a solid edge in Line 11. So there is a cycle that contains (u, v) . Since we are not able to merge two trees after the deletion of (u, v) , it must be that for every dotted edge on the cycle, both endpoints belong to the same tree. This implies that u and v are in the same tree after the deletion of the edge (u, v) , which means that there is a cycle in the forest S , a contradiction. \square

Lemma 3. *Every edge that the algorithm of Figure 4 turned from dotted to solid participates in all solutions.*

Proof. Assume otherwise, i.e., there exists an edge (u, v) that was turned from dotted to solid but which is excluded from a solution S . If (u, v) turned into a solid edge in Line 19 of the algorithm then \mathcal{G} has an isolated vertex, so S cannot be a proper forest. So the transformation must have occurred in Line 11. In this case, we know that the domain of NTREE is ground and contains only MINTREE, i.e., the number of trees in S is equal to the number of connected components of \mathcal{G} . But then any bridge of \mathcal{G} , and hence the edge (u, v) , must belong to S ; a contradiction. \square

5.4 Complexity

The complexity of checking whether the constraint has a solution is dominated by the complexity of computing MAXTREE (all the rest can be done in linear time). To find MAXTREE we need to find the cardinality of a maximum matching in \mathcal{G}_{MAYBE} and the best known upper bound for this is $\mathcal{O}(m\sqrt{n})$ time [13]. Lines 3 to 9 take linear time: We need to compute the connected components of \mathcal{G}_{TRUE} and traverse the dotted edges, spending a constant amount of time for each edge. Finding all bridges of \mathcal{G} in Line 10 and detecting leaves in Line 19 also takes linear time.

So far, the bottleneck is the feasibility test which takes $\mathcal{O}(m\sqrt{n})$ time. If the domain of NTREE is ground and contains only the value MAXTREE, we need to execute also Lines 13 to 17. Line 15 is trivial. For Line 16, we need to determine which edges of the graph belong to at least one maximum cardinality matching. For bipartite graphs, this can be done in linear time once a single maximum matching is known [11]. However, we need to perform this task on arbitrary graphs. In Section C of the Appendix we describe an algorithm that does this in $\mathcal{O}(mn)$ time.

Finally, for Line 19 we need an algorithm that receives a graph and a maximum matching and detects which vertices of the graph are saturated in every maximum matching. We show a linear-time solution in Section B of the Appendix.

Theorem 4. *The algorithm of Figure 4 filters the proper-forest constraint to hybrid-consistency in $\mathcal{O}(mn)$ time if $\mathcal{D}(\text{NTREE}) = \{\text{MAXTREE}\}$ and in $\mathcal{O}(m\sqrt{n})$ time otherwise.*

6 A Summary of Known Results on *tree* Covering Constraints

This section highlights the commonalities and differences between the constraints *tree* [1], *resource-forest* and *proper-forest*. All three constraints are defined on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, directed or undirected, with $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$.

Figure 5 summarizes the best known running times for checking feasibility and for achieving hybrid-consistency for each constraint. Figure 6 summarizes the main graph properties used to determine relevant bounds on the number of trees allowed to cover a given graph as well as the conditions for the existence of well-formed trees according to the definition of each constraint. The last table indicates that four basic graph properties completely define these constraints: connected components (in undirected graphs), strongly connected components (in digraphs), maximum matchings, existence of cycles. For each of the constraints, necessary conditions and filtering rules were deduced with known algorithms (e.g. dfs, maximum matching, connected component detection, etc.) as well as new algorithms (e.g., identifying vertices which are saturated in any maximum matching in an undirected graph). Observe that the lower and upper bounds MINTREE and MAXTREE for the *proper-forest* constraint exactly correspond to the lower and upper bounds on the number of connected components that appears in [14].

<i>Graph Pattern</i>	Trees	Undirected trees	
	<i>tree</i>	<i>proper-forest</i>	<i>resource-forest</i>
Checking feasibility	$\mathcal{O}(n + m)$	$\mathcal{O}(m\sqrt{n})$	$\mathcal{O}(n + m)$
Hybrid-consistency	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$ [worst], $\mathcal{O}(m\sqrt{n})$ [typical]	$\mathcal{O}(n + m)$

Fig. 5. Best known upper bounds for three *tree* covering constraints.

<i>Graph Pattern</i>	Trees	Undirected trees	
	<i>tree</i>	<i>proper-forest</i>	<i>resource-forest</i>
MINTREE	$ SCC_{sink}(G) $	$ CC(\mathcal{G}) $	$ CC(\mathcal{G}) $
MAXTREE	$ R_{potential}(\mathcal{G}) $	$ CC(\mathcal{G}_{TRUE}) + \mu(\mathcal{G}_{MAYBE})$	$ CC(\mathcal{G}_{TRUE}) $ with at least one <i>resource</i>
Well-formed trees	at least one potential root in each <i>SCC</i> of \mathcal{G}	no cycle in \mathcal{G}_{TRUE} no isolated vertex in \mathcal{G}	no cycle in \mathcal{G}_{TRUE} one <i>resource</i> vertex in each $CC(\mathcal{G}_{TRUE})$
Compatible number of trees	$\mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] \neq \emptyset$		

Fig. 6. Graph properties characterizing solutions to the three *tree* covering constraints.

Notation: For a graph H , the number of maximal CCs in H is $|CC(H)|$, the maximum cardinality of a matching in H is $\mu(H)$, the number of sink SCCs of H is $|SCC_{sink}(H)|$ and the number of potential roots in H is $|R_{potential}(H)|$.

References

1. N. Beldiceanu, P. Flener, and X. Lorca. The *tree* Constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, May 2005.
2. J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.
3. N. Beldiceanu and E. Contejean. Introducing global constraint in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. M. Sellmann. Cost-based filtering for shortest path constraints. In *CP 2003*, volume 2833 of *LNCS*, pages 694–708. Springer-Verlag, 2003.
5. M. Dinbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Int. Conf. on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, 1988.
6. J.-F. Puget. A C++ Implementation of CLP. In *Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages 256–261, Singapore, November 1994.
7. A. Cayley. A theorem on trees. *Quart. J. Math.*, 23:376–378, 1889.
8. C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. The *range* and *roots* Constraints: Specifying Counting and Occurrence Problems. In *IJCAI-05*, pages 60–65, 2005.
9. C. Berge. *Graphes*. Dunod, New York, 2nd edition, 1985. In French.
10. M. Sellmann. *Reduction techniques in Constraint Programming and Combinatorial Optimization*. PhD thesis, University of Paderborn, 2002.
11. J.-C. Régim. A filtering algorithm for constraints of difference in CSP. In *AAAI-94*, pages 362–367, 1994.
12. M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, Paris, 2nd edition, 1985. In French.
13. S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *FOCS 1980*, pages 17–27, New York, 1980. IEEE.
14. N. Beldiceanu, T. Petit, and G. Rochart. Bounds of Graph Characteristics. In P. van Beek, editor, *CP 2005*, volume 3709 of *LNCS*, pages 742–746. Springer-Verlag, 2005.

Appendix

A Omitted Parts of the Proof of Correctness of the Algorithm of Figure 3

Lemma 4. *The algorithm in Figure 3 never removes an edge from the graph or a value from $\mathcal{D}(\text{NTREE})$ that belongs to a solution.*

Proof. Assume that there is a solution S which contains an edge (u, v) which was removed in Line 7. Then there is a path of solid edges from u to v , and these edges are also in the solution. But then the forest contains a cycle, a contradiction.

If an edge (u, v) was removed in Line 17 then $\text{NTREE} = \text{MAXTREE}$, i.e., the number of trees in the forest is equal to the number of maximal connected components of $\mathcal{G}_{\text{TRUE}}$ that contain a resource vertex. Since each such connected component cannot be broken in the solution, we get that no two of them can be merged, because then the number of trees in the forest is less than NTREE . So (u, v) cannot belong to any solution.

If the algorithm removes a useful value from $\mathcal{D}(\text{NTREE})$, this clearly contradicts Theorem 1. \square

Lemma 5. *After applying the algorithm of Figure 3, every remaining edge in \mathcal{G} and value in $\mathcal{D}(\text{NTREE})$ participates in at least one solution, and every remaining dotted edge is excluded from at least one solution.*

Proof. Let S be a solution with MAXTREE trees that is obtained by the construction in the proof of Theorem 1.

Let (u, v) be an edge that was not removed from \mathcal{G} . If it is in S , we are done. Otherwise, it must be a dotted edge (all solid edges are in S).

Case 1: If u and v are in different trees of S , then $S' = S \cup (u, v)$ is still a forest (i.e., does not contain cycles). Clearly, every tree in S' contains at least one resource (because every tree in S does). If S' cannot be in a solution, it must be because it has less than MINTREE trees. Since it has $\text{MAXTREE} - 1$ trees, we get that $\text{MINTREE} = \text{MAXTREE}$, i.e., the number of CCs in $\mathcal{G}_{\text{TRUE}}$ that contain a resource is equal to the number of CCs in \mathcal{G} . This implies that every CC of \mathcal{G} contains at most one CC of $\mathcal{G}_{\text{TRUE}}$ that has a resource. So in S , either the tree of u or the tree of v does not contain a resource, contradicting our assumption that S is a solution.

Case 2: Now assume that u and v are in the same tree of S . Since (u, v) was not removed in Line 7, we know that u and v are not in the same CC of $\mathcal{G}_{\text{TRUE}}$. So in the path connecting them in S there is a dotted edge (u', v') . Let S' be the forest obtained from S by removing (u', v') and inserting (u, v) in its place, i.e., $S' = (S \setminus \{(u', v')\}) \cup \{(u, v)\}$. All solid edges still belong to S' , which has the same number of trees as S and induces the same partition of the vertices to trees. So S' is a solution that contains (u, v) .

Next, we need to show that any remaining value $k \in \mathcal{D}(\text{NTREE})$ belongs to a solution. If the algorithm does not change the number of CCs of $\mathcal{G}_{\text{TRUE}}$ that have a resource or the number of CCs of \mathcal{G} , then the claim then follows from Theorem 1. Assume that it

does change these values. The number of CCs of \mathcal{G}_{TRUE} which have a resource cannot increase because solid edges are not removed or turned to dotted edges. If this number decreased, it is because two previous such components were merged into one, and this can only happen if a dotted edge connecting them turned into a solid edge. However, a dotted edge can only turn into a solid edge if one of its endpoints belongs to a CC of \mathcal{G} with no resources or if it is a bridge of \mathcal{G} and $\mathcal{D}(\text{NTREE}) = \{\text{MINTREE}\}$. Clearly, this edge belongs to every solution with $k = \text{MINTREE}$ trees. As for the number of CCs of \mathcal{G} , it cannot decrease because we do not add edges to \mathcal{G} . It also cannot increase because an edge that is removed (in Line 7) is never a bridge.

It remains to show that every dotted edge (u, v) is excluded from some solution. We have shown above that there exists a solution S that uses (u, v) . Let $S' = S \setminus \{(u, v)\}$. If S' is a solution, we are done. Assume, then, that S' is not a solution. This can be either because it contains a tree without a resource or because it consists of too many trees. If the removal of (u, v) created a tree T without a resource, then since (u, v) was not turned into a solid edge in Line 10, we know that this tree belongs to a CC that contains a resource, so we can reconnect it with a resource by adding the appropriate edges to the forest. On the other hand, if every tree of S' contains a resource but S' consists of too many trees, then the number of trees in S is equal to the maximum value in $\mathcal{D}(\text{NTREE})$, which is at most MAXTREE . Assume that it is not possible to merge two trees by adding a dotted edge other than (u, v) . Then the number of trees in S is equal to the number of connected components in \mathcal{G} , i.e., to MINTREE . So $\mathcal{D}(\text{NTREE}) = \{\text{MINTREE}\}$. If (u, v) was a bridge, it would have turned into a solid edge in Line 14. So there is a cycle that contains (u, v) . Since we are not able to merge two trees after the deletion of (u, v) , it must be that for every dotted edge on the cycle, both endpoints belong to the same tree. This implies that u and v are in the same tree after the deletion of the edge (u, v) , which means that there is a cycle in the forest S , a contradiction. \square

Lemma 6. *Every edge that the algorithm of Figure 3 turned from dotted to solid participates in all solutions.*

Proof. Assume otherwise, i.e., there exists an edge (u, v) that was turned from dotted to solid but which is excluded from a solution S . If (u, v) turned into a solid edge in Line 10 of the algorithm then \mathcal{G} has a maximal connected component that does not contain a resource vertex, so S cannot be a resource-forest. So the transformation must have occurred in Line 14. In this case, we know that the domain of NTREE is ground and contains only MINTREE , i.e., the number of trees in S is equal to the number of connected components of \mathcal{G} . But then any bridge of \mathcal{G} , and hence the edge (u, v) , must belong to S ; a contradiction. \square

B Identifying Vertices That Are Saturated By Every Maximum Cardinality Matching (Omitted Part)

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and let $\mathcal{M} \subseteq \mathcal{E}$ be a maximum matching in G . Let $\mathcal{N}(\mathcal{M})$ be the set of vertices which are matched by \mathcal{M} and $\overline{\mathcal{N}(\mathcal{M})} = \mathcal{V} \setminus \mathcal{N}(\mathcal{M})$ the complement of $\mathcal{N}(\mathcal{M})$, i.e., the set of vertices which are *not* matched by \mathcal{M} .

We wish to identify the set \mathcal{S} of vertices which are saturated in all maximum matchings in \mathcal{G} . Clearly, if $|\mathcal{M}| = |\mathcal{V}|/2$ then $\mathcal{S} = \mathcal{V}$. Otherwise, $\overline{\mathcal{N}(\mathcal{M})} \neq \emptyset$.

Lemma 7. *A vertex v is not in \mathcal{S} iff there exists an alternating path $P = (u_1, u_2, \dots, u_\ell = v)$ from a vertex $u_1 \in \overline{\mathcal{N}(\mathcal{M})}$ to v such that $\ell = 1$ or the last edge $(u_{\ell-1}, u_\ell)$ on the path is in \mathcal{M} .*

Proof. Let \mathcal{A} be the proposition $v \notin \mathcal{S}$, and \mathcal{B} be the proposition $\exists P = (u_1, u_2, \dots, u_\ell)$ from $u_1 \in \overline{\mathcal{N}(\mathcal{M})}$ to v such that $\ell = 1$ or $(u_{\ell-1}, u_\ell) \in \mathcal{M}$. We have to show that $\mathcal{A} \Leftrightarrow \mathcal{B}$, for this purpose we first prove that $\neg\mathcal{B} \Rightarrow \neg\mathcal{A}$ and next we show that $\neg\mathcal{A} \Rightarrow \neg\mathcal{B}$:

- Let us suppose that $\forall P = (u_1, u_2, \dots, u_\ell)$ from $u_1 \in \overline{\mathcal{N}(\mathcal{M})}$ to v such that $\ell \neq 1$ and $(u_{\ell-1}, u_\ell) \notin \mathcal{M}$ (i.e. $\neg\mathcal{B}$). If $u_\ell = v \notin \mathcal{S}$ then for any alternating path $P = (u_1, u_2, \dots, u_\ell)$ such that $u_1, u_\ell \in \overline{\mathcal{N}(\mathcal{M})}$, there exists an augmenting alternating path such that $u_1, u_\ell \in \mathcal{N}(\mathcal{M})$, thus $u_\ell = v \notin \mathcal{S}$ is impossible and as a consequence $u_\ell = v \in \mathcal{S}$ (i.e. $\neg\mathcal{A}$).
- Let us suppose that $u_\ell = v \in \mathcal{S}$ (i.e. $\neg\mathcal{A}$) then $\forall P = (u_1, u_2, \dots, u_\ell)$ from $u_1 \in \overline{\mathcal{N}(\mathcal{M})}$ to v is such that $\ell \neq 1$ and $(u_{\ell-1}, u_\ell) \notin \mathcal{M}$ (i.e. $\neg\mathcal{B}$). Indeed, if $u_\ell = v = u_1$ then $v \in \overline{\mathcal{N}(\mathcal{M})}$ which is impossible because $v \in \mathcal{S}$. Moreover, if $(u_{\ell-1}, u_\ell) \in \mathcal{M}$ then for each path P , there exists an alternating path such that $(u_1, u_2) \in \mathcal{M}$ and $(u_{\ell-1}, u_\ell) \notin \mathcal{M}$ then all adjacent edges of $u_\ell = v$ are non-matched which is impossible because $u_\ell = v \in \mathcal{S}$.

□

The algorithm initializes $\text{SAT} = \mathcal{N}(\mathcal{M})$ and then removes vertices from SAT if there is an alternating path as in the statement of Lemma 7. The alternating paths will be identified by a DFS traversal of the graph, which begins at a vertex which is not saturated in \mathcal{M} and proceeds along alternating paths. Whenever the traversal reaches a vertex $v \in \text{SAT}$ by a matching edge, the algorithm removes v from SAT.

In an unrestricted DFS traversal, each vertex is flagged “visited” or “unvisited”. Since we allow the DFS to proceed only along alternating paths, the status of each vertex is now determined by two flags. The first determines whether it was visited by a matching edge and the second determines whether it was visited by a non-matching edge. The algorithm is shown in Figure 7. The root of the DFS traversal is a non-matched vertex. Each non-matched vertex becomes the root in its turn, but note that (as with ordinary DFS) the flags are not reset, so we do not know (or care) which non-matched vertex was the root when a certain vertex was visited. The recursive function DFS receives the graph \mathcal{G} , the matching \mathcal{M} , the current set SAT, the current vertex v and a Boolean parameter called “matching” which specifies whether the edge by which this vertex was reached is a matching or a non-matching edge.

The following lemma states that the algorithm correctly identifies \mathcal{S} .

Lemma 8. *When the algorithm in Figure 7 terminates, $\text{SAT} = \mathcal{S}$, i.e., the variable SAT contains the set of vertices which are matched in every maximum matching in \mathcal{G} .*

Proof. Case 1: $v \notin \mathcal{S}$. Then by Lemma 7, there exists an alternating path P from an unmatched vertex u to v , such that P is empty or ends with a matching edge. We prove by induction on the number of non-matching edges ℓ on P that $v \notin \text{SAT}$. In the base

procedure AlwaysSaturated(\mathcal{G}, \mathcal{M})

1. $SAT = \mathcal{N}(\mathcal{M});$
2. **forall** $v \in \mathcal{V}$ **do**
3. visited_by_matching_edge(v) $\leftarrow no;$
4. visited_by_non_matching_edge(v) $\leftarrow no;$
5. **forall** root $r \in \overline{\mathcal{N}(\mathcal{M})}$ **do**
6. DFS($\mathcal{G}, \mathcal{M}, SAT, r, yes$);

procedure DFS($\mathcal{G}, \mathcal{M}, SAT, v, matching$)

1. **if** matching = yes **then** (* v is a root or was reached by a matching edge *)
2. **if** visited_by_matching_edge(v) = yes **then** return;
3. $SAT \leftarrow SAT \setminus \{v\}.$
4. visited_by_matching_edge(v) $\leftarrow yes;$
5. **forall** $(v, u) \in \mathcal{E} \setminus \mathcal{M}$ **do**
6. DFS($\mathcal{G}, \mathcal{M}, SAT, u, no$);
7. **else** (* v was reached by a non-matching edge *)
8. **if** visited_by_non_matching_edge(v) = yes **then** return;
9. visited_by_non_matching_edge(v) $\leftarrow yes;$
10. **if** $\exists (v, u) \in \mathcal{M}$ **then**
11. DFS($\mathcal{G}, \mathcal{M}, SAT, u, yes$);

Fig. 7. Algorithm for detecting always-saturated vertices.

case, $\ell = 0, v \in \overline{\mathcal{N}(\mathcal{M})}$ and v will be visited as a root, with the variable “matching” set to “no”. It will then be removed from SAT.

For the induction step, assume that the algorithm correctly identified all vertices which are not in \mathcal{S} and which are reachable from an unmatched vertex by an alternating path that ends with a matching edge and contains $\ell - 1$ non-matching edges. Let v be a vertex which is reachable from a non-matched vertex r by an alternating path $P' = (r = u_1, u_2, \dots, u_{2\ell} = v)$, which traverses ℓ non-matching edges. By the induction hypothesis, the vertex $u_{2\ell-2}$ was removed from SAT. So $u_{2\ell-1}$ was visited by a non-matching edge, and hence $u_{2\ell}$ was visited by a matching edge and was therefore also removed from SAT.

Case 2: $v \in SAT$. Then by Lemma 7, v is not unmatched by \mathcal{M} so it is inserted into SAT. It is easy to see that if it is later removed from SAT, this is because it is reached through a matching edge by an alternating path from an unmatched vertex. But then by Lemma 7 it is not in \mathcal{S} , a contradiction.

□

C Identifying Edges That Do Not Belong To Any Maximum Cardinality Matching (Omitted Part)

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and let $\mathcal{M} \subseteq \mathcal{E}$ be a maximum matching in \mathcal{G} . We wish to identify the set of useful edges, i.e., edges which belong to at least one maximum

matching in \mathcal{G} . We therefore need to find edges that belong to either an alternating path starting with a non-matching edge and ending with a matching edge, or an alternating cycle.

For each vertex u , we identify the set of vertices that are reachable from u by a simple alternating path that begins and ends with non-matching edges. This can be done by a suitable DFS traversal, which can also identify the edges on alternating paths beginning with a non-matching edge incident to u .

We now describe how this information can be used to determine whether a particular non-matching edge belongs to an alternating cycle. Let (x, y) be a non-matching edge. It can belong to an alternating cycle only if there are u, v such that $(x, u) \in M$ and $(y, v) \in M$. Assume that there exists a simple alternating path between u and v that begins and ends with non-matching edges. Since x and y are incident on matching edges connecting them with the path's endpoints, the path cannot visit either one of them: If the path leaves x (y) by the matching edge, it reaches u (v) by a matching edge. Since it must end with a non-matching edge, it must continue and enter v by a non-matching edge. So the path visits u (v) twice, i.e., it is not simple. By symmetry, it also cannot enter x (y) by a matching edge. But it is an alternating path, so it cannot both enter and leave by non-matching edges. Hence, it does not visit x and y .

In other words, the simple path and the path (u, x, y, v) combine into a simple alternating cycle. Clearly, if there is an alternating cycle containing (x, y) , then it must contain the path (u, x, y, v) .

To summarize, the algorithm performs n DFS traversals, one from each vertex, and determines which vertex-pairs are connected by a simple alternating path that begins and ends with non-matching edges. This takes $O(mn)$ time. Then, using this information, the algorithm traverses the non-matching edges and for each such edge (x, y) , if there are matching edges (x, u) and (y, v) , it checks whether this 2-path can be completed to an alternating cycle. This can be done in constant time per edge after proper preprocessing.