The *Minimum Spanning Tree* Constraint

Grégoire Dooms[1,⋆] and Irit Katriel[2,⋆⋆]

[1] Department of Computing Science and Engineering
Université catholique de Louvain, Belgium
dooms@info.ucl.ac.be
[2] BRICS⋆⋆⋆, University of Aarhus, Denmark
irit@daimi.au.dk

Abstract. The paper introduces the $MST(G, T, W)$ constraint, which is specified on two graph variables G and T and a vector W of scalar variables. The constraint is satisfied if T is a minimum spanning tree of G, where the edge weights are specified by the entries of W. We develop algorithms that filter the domains of all variables to bound consistency.

1 Introduction

Complex constraints and variable types simplify the modelling task while providing the constraint solver with a better view of the structure of the CSP. The recently introduced CP(Graph) framework [4] allows the programmer to define *graph variables*, i.e., variables whose assigned values are graphs. The usefulness of graph variables depends on the existence of filtering algorithms for constraints defined on them. That is, a constraint solver that supports graph variables ideally supports a collection of constraints that describe fundamental graph properties.

In this paper we introduce the $MST(G, T, W)$ constraint, which is specified on two graph variables G and T and a vector W of scalar variables. The constraint is satisfied if T is a minimum spanning tree (MST) of G (i.e., the minimum-weight connected subgraph that contains all nodes of G), where the positive weights of the edges in G (and hence also in T) are specified by the entries of W.

Finding the MST of a graph takes almost-linear time, but several interesting variants of the MST problem, such as minimum k-spanning tree [8] (finding a minimum-weight tree that spans any set of k nodes) and Steiner tree [10] (finding a minimum-weight tree that spans a given set of nodes) are known to be NP-hard. Such problems can be modelled by a combination of MST and other constraints.

In other applications, there is uncertainty in the input, e.g., when the exact weight of an edge is not known, but can be assumed to belong to a given interval of values. In the widely-studied MST sensitivity analysis problem, we are given a fixed graph g with fixed weights and an MST t of g. We need to determine, for each edge, the amount by which its weight can be changed without violating $t = MST(g)$. The robust spanning tree problem [1] addresses uncertainty from a different point of view. Its input is a graph

⋆ Supported by the Walloon Region, project BioMaze (WIST 315432).
⋆⋆ Supported by the Danish Research Agency (grant # 272-05-0081).
⋆⋆⋆ Basic Research in Computer Science, funded by the Danish National Research Foundation.

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 152–166, 2006.

with a set of possible edge weights for every edge. A *scenario* is a selection of a weight for each of the edges. For a spanning tree t and a scenario s, the *regret* of t for s is the difference between the weight of t and the weight of the MST of the graph under scenario s. The output is a spanning tree that minimizes that worst-case regret.

Finally, in inverse parametric optimization problems [7] we are interested in finding parameters of an optimization problem, given its solution. For instance, our MST constraint allows us to receive a tree and determine which graphs have this tree as their MST.

Our results are summarized in the table below. We look at various restrictions of the MST constraint, and develop a bound consistency algorithm for each of them. Let n and m be, respectively, the number of nodes and edges in the upper bound of the domain of G (and hence also of T). Let $Sort(m)$ be the time it takes to sort m edge-weights and α the slow-growing inverse-Ackerman function. As the table indicates, our algorithm computes bound consistency for the most general case in cubic time but whenever the domain of one of the variable is fixed, bound consistency can be computed in almost-linear time. The table indicates in which section of the paper each case is handled.

	Fixed Edge-Weights		**Non-Fixed Edge-Weights**	
	Fixed Graph	Non-Fixed Graph	Fixed Graph	Non-Fixed Graph
Fixed Tree	MST verification $O(m+n)$ [11]	$O(m+n)$ [Section 4.1]	$O(m\alpha(m,n))$ [Section 5.1]	$O(m\alpha(m,n))$ [Section 5.2]
Non-Fixed Tree	$O(Sort(m)+ m\alpha(m,n))$ [Section 4.2]	$O(Sort(m)+ m\alpha(m,n))$ [Section 4.3]	$O(Sort(m)+ m\alpha(m,n))$ [Section 5.3]	$O(mn(m+\log n))$ [Section 5.4]

Related Work. Filtering algorithms have been developed for several constraints on graphs, such as Sellmann's shorter paths constraints [14], unweighted forest constraints (directed [2] and undirected [13]) and the Weight-Bounded Spanning-Tree constraint [5], $WBST(G,T,W,I)$, which specifies that T is a spanning tree of G of weight at most I, where W is again a vector of edge weights. Although $WBST$ is semantically close to MST, the structure of the solution set is different and the bound consistency algorithms that we have developed are very different from the ones described in this paper. Perhaps the most obvious difference between the two constraints is that for $WBST$, in the most general case (when all three variables are not fixed) it is NP-hard to decide whether a solution exists (and hence also to filter the constraint to bound consistency). For MST, it is possible to do this in polynomial time. From the application point of view, $WBST$ is a cost-based filtering constraint (i.e., an optimization constraint) and it can be used in conjunction with the MST constraint to get more pruning.

In [1], I. Aron and P. Van Hentenryck addressed the robust spanning tree problem with interval data. They describe an algorithm that partially solves a special case of the

filtering problem that we address in this paper. More precisely, for the case in which the graph is fixed, they show how to detect which edges must be removed from the upper bound of the tree domain. Finally, our filtering algorithms apply techniques that were previously used in King's MST verification algorithm [11] and Eppstein's algorithm for computing the k smallest spanning trees [6] of a graph.

Roadmap. The rest of the paper is structured as follows. Section 2 contains some preliminaries, definitions and conventions used throughout the paper. In Section 3 we describe a preprocessing step and invariants that are maintained during execution of the algorithms described in subsequent sections. Sections 4 and 5 form the core of the paper and describe the bound consistency algorithms for increasingly complicated cases.

2 Preliminaries and Notation

2.1 Set and Graph Variables

The domain $D(x)$ of a *set variable* x is specified by two sets of elements: The set of elements that must belong to the set assigned to x (which we call the *lower bound* of the domain of x and denote by $\underline{D}(x)$) and the set of elements that may belong to this set (the *upper bound* of the domain of x, denoted $\overline{D}(x)$). The domain itself has a lattice structure corresponding to the partial order defined by set inclusion. In other words, for a set variable x with domain $D(x) = [\underline{D}(x), \overline{D}(x)]$, the value $v(x)$ that is assigned to x in any solution must be a set such that $\underline{D}(x) \subseteq v(x) \subseteq \overline{D}(x)$.

A *graph variable* can be modelled as two set variables V and E with an inherent constraint specifying that $E \subseteq V \times V$. Alternatively, the domain $D(G)$ of a graph variable G can be specified by two graphs: A lower bound graph $\underline{D}(G)$ and an upper bound graph $\overline{D}(G)$, such that the domain is the set of all subgraphs of the upper bound which are supergraphs of the lower bound. We will assume the latter because it is more convenient when describing filtering algorithms.

2.2 Assumptions and Conventions

The constraint considered in this paper is defined on graph variables as well as scalar variables. The latter represent weights and are assigned numbers. We will assume that the domain of each scalar variable is an interval, represented by its endpoints. The number of entries in the vector W is equal to the number of edges in G. We will abstractly refer to $W[e]$ as the entry that contains the weight of edge e, ignoring implementation details. We also allow those intervals to be filtered to an empty interval when the corresponding edge does not belong to any solution, ignoring the fact that some constraint systems do not allow empty domains.

When the cardinality of the domain of a variable is exactly 1, we say that the variable is *fixed*. We will denote fixed variables by lowercase letters and non-fixed variables by capital letters.

2.3 Bound Consistency

For a constraint that is defined on variables whose domains are intervals, specified by their endpoints, computing bound consistency amounts to shrinking the variable domains as much as possible without losing any solutions. If the domain is an interval from a total order (which is the case for the entries of W), the bound consistent domain is specified simply by the smallest and the largest value that the variable can assume in a solution.

On the other hand, when the domain is an interval of a partial order (which is the case with a graph variable), the lower bound is the intersection of all values that the variable can assume and the upper bound is the union of all such values. Note that the domain endpoints might not be values that the variable can assume in a solution. Since a bound consistency algorithm may only remove elements from a variable domain but never add new ones, filtering the domain of a graph variable to bound consistency amounts identifying which of the nodes and edges in its upper bound graph must belong to the graph in all solutions (and placing them in the lower bound graph) as well as which nodes and edges may not belong to the graph in any solution (and removing them from the upper bound graph).

3 Side Constraints and Invariants

Throughout the paper, we will assume that the following filtering tasks have been performed in a preprocessing step: First, the algorithm applies the bound consistency algorithms described in [4,5] for the constraints $Nodes(G) = Nodes(T)$, $Subgraph(T, G)$ (which specifies that T is a subgraph of G) and $Tree(T)$ (which specifies that T is connected and acyclic). Filtering for the $Tree(T)$ constraint removes from $\overline{D}(T)$ arcs whose endnodes belong to the same connected component of $\underline{D}(T)$, and enforces that T is connected: If there are two nodes in $\underline{D}(T)$ that do not belong to the same connected component of $\overline{D}(T)$ then the constraint has no solution. Otherwise, any bridge or cut-node in $\overline{D}(T)$ whose removal disconnects two nodes from $\underline{D}(T)$ is placed in $\underline{D}(T)$.

Note that the conjunction of $Subgraph(T, G)$ and $Tree(T)$ enforces $Connected(G)$ and that we may assume that bound consistency for $Subgraph(T, G)$ is maintained dynamically at no asymptotic cost, as described in [5].

Finally, since $\underline{D}(T)$ is contained in the MST in any solution, we reduce the problem to the case in which $\underline{D}(T)$ is initially empty, as follows. We contract all edges of $\underline{D}(T)$ in g and obtain the graph g'. For any MST t' of g', the edge-set $t' \cup \underline{D}(T)$ is a minimum-weight spanning tree of g that contains $\underline{D}(T)$.

4 Fixed Edge Weights

In this section we assume that the edge weights are fixed. That is, the domain of W contains exactly one vector. We separately handle each of the three subcases where at least one other variable is not fixed.

4.1 Non-fixed Graph and Fixed Tree

We first tackle the simple case $MST(G, t, w)$, where $dom(G) = [\underline{D}(G), \overline{D}(G)]$ is not fixed while the minimum spanning tree and the edge weights are.

After applying the filtering described in Section 3 for $Subgraph(T, G)$ and node-set equality, it remains to enforce that t is an MST of the value assigned to G. This means that we need to remove from $\overline{D}(G)$ any edge (u, v) which is not in t and which is lighter than all edges on the path p in t between u and v; by the cycle property, the heaviest edge on the cycle $p \cup \{(u, v)\}$ (which is in t), cannot belong to any MST, so the cycle must not be in G. The only way to exclude the cycle is to remove (u, v) from $\overline{D}(G)$. This can be done as follows in linear time: For each edge $e = (u, v)$ in $\overline{D}(G) \setminus t$, compute the maximum edge weight $w*$ of an edge on the path from u to v in t using King's algorithm [11], which receives a weighted tree and, in linear time, constructs a data structure that supports constant-time queries of the form "which is the heaviest edge on the tree path between u and v?" If $w(e) < w*$, e may not belong to G. If it is in $\underline{D}(G)$ the constraint has no solution. Otherwise, remove it from $\overline{D}(G)$.

4.2 Fixed Graph and Non-fixed Tree

We turn to the case $MST(g, T, w)$ where the variables G and W of the constraint are fixed. The tree T is constrained to be a minimum spanning tree of the given graph g and the bound-consistency problem amounts to finding the union and the intersection of all MSTs of g.

Analysis of g to filter $D(T)$. We now describe a variant of Kruskal's algorithm [12] that constructs an MST of g while partitioning its edge-set into the sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$, defined as follows.

Definition 1. *Let g be a connected graph. The sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$ contain, respectively, the edges that belong to all, some or none of the MSTs of g.*

For an unconnected graph g whose maximal connected components are $g_1, \ldots, g_k$, we extend this definition to be the union of the respective set for each maximal connected component of g. Formally,

$$Mandatory(g) = \cup_{i=1}^{k} Mandatory(g_i),$$
$$Possible(g) = \cup_{i=1}^{k} Possible(g_i),$$
$$Forbidden(g) = \cup_{i=1}^{k} Forbidden(g_i),$$

As in Kruskal's original version, the algorithm begins with a set of n singleton nodes and grows a forest by repeatedly inserting a minimum weight edge that does not create a cycle. The difference is that instead of considering one edge at a time, in each iteration we extract from the queue all edges of minimal weight, determine which of them are mandatory, possible or forbidden, and only then attempt to insert them into the forest. Let t_k be the forest constructed by using edges of weight less than k and let E_k be the

set of edges of weight k. Let $(u, v) \in E_k$ and let $C(u)$ and $C(v)$ be the connected components in t_k of u and v, respectively. If $C(u) = C(v)$, then by the cycle property (u, v) does not belong to any MST of g (i.e., $(u, v) \in Forbidden(g)$). If $C(u) \neq C(v)$ and (u, v) is a bridge in $t_k \cup E_k$, then by the cut property (u, v) belongs to all MSTs of g (i.e., $(u, v) \in Mandatory(g)$).

The running time of this algorithm is $O(Sort(m) + m\alpha(m, n))$ where $Sort(m)$ is the time required to sort the edges by weight and α is the inverse-Ackerman upper bound of the union-find data structure [16] that represents the trees of the forest. When a batch of edges is extracted from the queue we need to perform a bridge computation in the graph composed of these edges to distinguish between possible and mandatory edges. Bridge detection takes time which is linear in the number of edges [15] and each edge of g participates in one bridge computation.

Filtering the Domain of T. We are now ready to use the results of the analysis of g to filter the domain of T. This entails the following steps: (1) For each mandatory edge $e \in Mandatory(g)$, if $e \notin \overline{D}(T)$ then there is no solution. Otherwise, place e in $\underline{D}(T)$. (2) For each forbidden edge $e \in Forbidden(g)$, if $e \in \underline{D}(T)$ then there is no solution. Otherwise, remove e from $\overline{D}(T)$.

Since $Mandatory(g)$ and $Forbidden(g)$ are disjoint, the two steps have no effect on each other, so they may be applied in any order, and it suffices to apply each of them only once. But could we achieve more filtering by repeating the whole algorithm again, from the preprocessing step through the analysis of g to the filtering steps? We will now show that we cannot.

Let e be an edge that was placed in $\underline{D}(T)$ in the first filtering step. Then $e \in Mandatory(g)$, which means that it belongs to all MSTs of g. Let t_1 and t_2 be the two trees that e merges together when it is inserted into the forest by our variant of Kruskal's algorithm on g. Then the edges that were extracted from the queue before e do not contain an edge between t_1 and t_2, because in that case e would have been placed in either $Possible(g)$ or $Forbidden(g)$. This means that if e is in $\underline{D}(T)$ from the start, all edges lighter than e would be classified as before. Clearly, the edges heavier than e see the same partition of the graph into trees whether e is in $\underline{D}(T)$ or not. Since e is mandatory, the edges that have the same weight as e do not belong to a path between e's endpoints that uses only edges with weight at most equal to that of e. Hence, placing e in $\underline{D}(T)$ cannot change their classification.

Now, let e be an edge that was removed from $\overline{D}(T)$ in the second filtering step. Then $e \in Forbidden(g)$, which means that it does not belong to any MST of g. Then its removal from $\overline{D}(T)$ does not have any effect on the classification of other edges as $Forbidden$, $Possible$ or $Mandatory$.

We do not need to apply the filtering steps of Section 3. The nodes of T are fixed so we cannot detect new cut nodes. Clearly an impossible edge does not disconnect $\overline{D}(T)$ (otherwise it would be mandatory by definition). We show that the pruning of $\overline{D}(T)$ cannot creates new bridges then these bridges are already in $Mandatory(g)$. We consider an impossible edge e is removed and creates a bridge in $\overline{D}(T)$. As it is impossible, its endpoints are connected by a path composed of edges lighter than e. Then when the bridge e' of weight k was processed, the edge e was not in the graph g_k, and e' was a bridge in g_k. Hence it was classified as mandatory.

In conclusion, if we apply the algorithm again, the analysis of g would classify all edges in the same way as before. In other words, applying the algorithm again will not result in more filtering.

4.3 Non-fixed Graph and Tree

We now turn to the case $MST(G, T, w)$, in which both the graph and the tree are not fixed (but the edge weights are). Recall that we begin by applying the preprocessing step described in Section 3.

Analyzing $D(G)$ to filter $D(T)$. The main complication compared to the fixed-graph case is in the analysis of the set of graphs described by $D(G)$ in order to filter $D(T)$. We extend the definition of the sets $Mandatory$, $Possible$ and $Forbidden$ for a set of graphs as follows:

Definition 2. *For a set S of graphs, the set $Mandatory(S)$ contains the edges that belong to every MST of any connected graph in S, the set $Forbidden(S)$ contains the edges that do not belong to any MST of a connected graph in S and the set $Possible(S)$ contains all other edges in the union of the graphs in S.*

We need to identify the sets $Mandatory(D(G))$ and $Forbidden(D(G))$. We will show that it suffices to analyze the two bounds of the graph domain, namely the graphs $\underline{D}(G)$ and $\overline{D}(G)$.

Lemma 1 (Downgrade lemma). *The addition of an edge to a graph can only downgrade the state of the other edges of this graph. Here, downgrading means staying the same or going from "mandatory" to "possible" to "forbidden". Formally: Let $g^+ = g \cup \{e = (u, v)\}$ where $e \notin g$ and let $k = w(e)$. Then:*

$$\forall a \in g: \ \big(a \in Mandatory(g^+) \Rightarrow a \in Mandatory(g)\big) \wedge \\ \big(a \in Possible(g^+) \Rightarrow a \in Mandatory(g) \cup Possible(g)\big)$$

Proof. We compare the classification of the edges obtained by running the algorithm of Section 4.2 twice in parallel, one copy called A running on the graph g and one copy called A^+ running on g^+. Clearly, as long as the edge e is not popped from the queue, the edges are classified in the same way in both graphs.

If e is an impossible edge of g^+, then the edges popped after it will still be classified in the same way in both graphs. Otherwise, it can affect the fate of the edges that are popped at the same time or later:

Case 1. If u and v belong to trees that are also merged by another edge e' with weight equal to that of e, then both e and e' are classified in g^+ as possible, while in g the edge e' was classified as either possible or mandatory (depending on whether it was the only path connecting these trees in the batch). After this, the partition of the nodes of the graphs into trees is the same for g and g^+, and the algorithms classify the remaining edges in the same way.

Case 2. Otherwise, A leaves u and v in different trees while A^+ merges the two trees. At some point, both algorithms see a batch whose insertion connects u and v in g. These edges will be classified by A as either possible (if there is more than one) or mandatory (if there is only one). On the other hand, A^+ will classify them as impossible because their endpoints already belong to the same tree. In all other steps, both algorithms classify the edges in the same way. □

Note that the addition of a node of degree 1 and its incident edge to a graph does not change the status of the other edges in the graph. This edge just becomes mandatory.

We now show how to identify the sets $Forbidden(D(G))$ and $Mandatory(D(G))$. The set $Possible(D(G))$ consists of the remaining edges in $\overline{D}(G)$. Recall that the set $Forbidden(\underline{D}(G))$ is the union of the forbidden edges of each maximal connected component of $\underline{D}(G)$.

Theorem 1. *The set $Forbidden(D(G))$ of edges that do not belong to any MST of a connected graph in $D(G)$ is*

$$Forbidden(D(G)) = Forbidden(\underline{D}(G)).$$

Proof. A direct consequence of the downgrade lemma. □

We now turn to computing the $Mandatory$ set. The following lemma states that the mandatory edges belong to the mandatory set of $\underline{D}(G)$. Note that this does not follow from the definition of $Mandatory(G)$, because if $\underline{D}(G)$ is not connected, it does not belong to $D(G)$.

Lemma 2. *The set of edges that belong to all MSTs of all connected graphs in $D(G)$ is contained in the set of mandatory edges for $\underline{D}(G)$, i.e.,*

$$Mandatory(D(G)) \subseteq Mandatory(\underline{D}(G)).$$

Proof. If $\underline{D}(G)$ is empty, $Mandatory(D(G)) = \emptyset$. Otherwise, there are two cases: If $\underline{D}(G)$ is a connected graph, then it belongs to $D(G)$ and the lemma holds by definition. Otherwise, we may assume that $\overline{D}(G)$ is connected. This follows from the pruning rules of the connected constraint in the preprocessing step: Otherwise either the constraint has no solution or $\underline{D}(G)$ is empty or we can remove all but one connected component of $\overline{D}(G)$. Let $\overline{D}(G)'$ be the graph obtained from $\overline{D}(G)$ by contracting every connected component of $\underline{D}(G)$.

Let g be a minimal connected graph in $D(G)$, i.e., a graph in $D(G)$ such that the removal of any node or edge from g results in a graph which is either not connected or not in $D(G)$. Then g consists of the union of $\underline{D}(G)$ with a set t' of additional nodes and edges in $\overline{D}(G) \setminus \underline{D}(G)$. The set of mandatory edges of g, then, is the union of $Mandatory(\underline{D}(G))$ and the mandatory edges in t'.

Since we assume that the preprocessing step was performed, we know that every edge in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one connected graph in $D(G)$: Otherwise, it is a bridge in $\overline{D}(G)$ that connects two mandatory nodes and was not included into $\underline{D}(G)$ in the preprocessing step, a contradiction. We get that the intersection of the mandatory sets over all minimal connected graphs in $D(G)$ is equal to $Mandatory(\underline{D}(G))$.

Since every connected graph in $D(G)$ is a supergraph of at least one minimal connected graph of $D(G)$, we get by the downgrade lemma that the mandatory set for any graph is contained in the mandatory set for some minimal graph. This concludes the proof. □

Theorem 2. *The set $Mandatory(D(G))$ of edges that belong to all MSTs of graphs in $D(G)$ is:*

$$Mandatory(D(G)) = Mandatory(\overline{D}(G)) \cap Mandatory(\underline{D}(G))$$

Proof. If an edge is present in all MSTs of all connected graphs in $D(G)$ then it is present in all MSTs of $\overline{D}(G)$ and all MSTs of the minimal connected graphs of $D(G)$. Hence, by lemma 2, $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G)) \supseteq Mandatory(D(G))$. Assume that there exists an edge e in $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G))$ which is not in $Mandatory(D(G))$. Then there is a graph $g \in D(G)$ such that e is not in $Mandatory(g)$. Since $g \subset \overline{D}(G)$, we can obtain $\overline{D}(G)$ by a series of edge insertions. One of these insertions turned e from a non-mandatory edge into a mandatory one, in contradiction to the downgrade lemma. □

Example 1. Assume that the domain of G is the graph shown in Figure 1, where the solid edges are in $\underline{D}(G)$ and the dashed edge is in $\overline{D}(G) \setminus \underline{D}(G)$. Then $Mandatory(\overline{D}(G))$ contains the edges weighted 1 and 2, while the edge of weight 3 is forbidden. On the other hand, $Mandatory(\underline{D}(G))$ contains the edges weighted 2 and 3, because the edge of weight 1 is not in $\underline{D}(G)$. Hence, only the edge of weight 2 is mandatory in all graphs of $D(G)$.

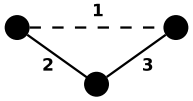

Fig. 1. The domain of G in Example 1

Filtering the Domains of G and T. Using the results of the previous sections, we derive a simple algorithm to filter the domains of T and G to bound consistency.

As before, we begin by applying the preprocessing step of Section 3. We then proceed as follows.

Step 1: We filter $D(T)$ according to $Mandatory(D(G))$ and $Forbidden(D(G))$ as in Section 4.2. By Theorems 1 and 2, we have these two sets if we know $Forbidden(\underline{D}(G))$, $Mandatory(\underline{D}(G))$ and $Mandatory(\overline{D}(G))$. The latter can be computed by applying the algorithm described in Section 4.2 to both bounds of $D(G)$. Once these sets have been computed, we use them as in Section 4.2 to filter $D(T)$.

Step 2: To filter $D(G)$, we need to identify edges that cannot be in G because otherwise T would not be the minimum spanning tree of G. An edge (u, v) has this property iff on every path P in $\overline{D}(T)$ between u and v there is an edge which is heavier than (u, v).

Example 2. To illustrate this condition, assume that the domain of T is as described in Figure 2 and that $\overline{D}(G)$ contains the edge (u, v) with weight 2. Although u and v are not connected in $\underline{D}(T)$, any path from u to v in a tree in $D(T)$ contains an edge which is heavier than (u, v), so (u, v) must not be in G. On the other hand, if the weight of the edge (x, y) is 1, then (u, v) can be in G if $(x, y) \in T$.

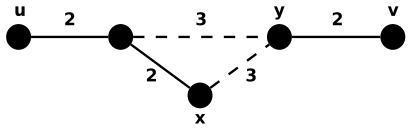

Fig. 2. The domain of T in Example 2. Edges of $\underline{D}(T)$ and solid and those of $\overline{D}(T) \setminus \underline{D}(T)$ are dashed.

To find these edges, we apply a modified version of the algorithm of Section 4.2 to $\overline{D}(G)$: We reverse the contraction of the edges of $\underline{D}(T)$ and create a sorted list of all edges of $\overline{D}(G)$, including those of $\underline{D}(T)$. As before, we begin with a graph H that contains the nodes of $\overline{D}(G)$ as singletons and at each step we extract from the queue the batch B of all minimum-weight edges. We first insert the edges of $B \cap \overline{D}(T)$ into H and contract each of them by merging the connected components that its endpoints belong to. Then, we remove from $\overline{D}(G)$ every edge in $B \setminus \overline{D}(T)$ that connects two different connected components of H; If any of these edges are in G, then at least one of them must belong to the MST of G. But they cannot be in T, so we must make sure that they are not in G either.

After applying Step 2, we need to apply Step 1 again. However, we show that after doing so we have reached a fixpoint. We first show that the second application of the first step does not change $\overline{D}(T)$. Since the second step depends only on $\overline{D}(T)$, we are done if we use the $\overline{D}(G)$ computed by the second step to update $\underline{D}(T)$.

Consider the impact of the removal of a non-tree edge e (i.e., an edge which is not in $\overline{D}(T)$) from $\overline{D}(G)$ during Step 1. The set $Forbidden$ is not affected because it depends only on $\underline{D}(G)$. Assume that the removal of e causes the insertion of an edge e' into $\underline{D}(G)$. Then e' was a bridge in $\overline{D}(G) \setminus \{e\}$. But since $e \notin \overline{D}(T)$, e' is a bridge in $\overline{D}(T)$ so it already was in $\underline{D}(T)$, and hence also in $\underline{D}(G)$, a contradiction. This proves that we have reached a fixed-point.

5 Handling Non-fixed Weights

We now turn to the case where the edge weights are not fixed, i.e., the domain of each entry in W is an interval of numbers, specified by its endpoints. Once again, we begin with simple cases where some of the variables are fixed and gradually build up to the most general case.

5.1 Fixed Graph and Tree

When the graph and tree are fixed ($MST(g, t, W)$), the filtering task is to compute the minimum and maximum weight that each edge can have such that t is an MST of g. [1] A non-tree edge (u, v) must not be lighter than any edge on the path in t between u and v. We can apply King's algorithm to t, while assuming that each tree edge has the minimum possible weight. Then, if the data structure returns the edge e' for the query $e = (u, v)$, we filter the domain of the weight of e, denoted $W[e]$, by setting $D(W[e]) \leftarrow D(W[e]) \cap [\underline{D}(W[e']), \infty]$.

For a tree edge e, let $t_1(e)$ and $t_2(e)$ be the two trees obtained by removing e from t. Then e must not be heavier than any other edge in g that connects a node from $t_1(e)$ and a node from $t_2(e)$. Let $r(e)$ be the minimum weight edge between $t_1(e)$ and $t_2(e)$ in the graph $g \setminus e$. Assuming that every non-tree edge has the maximal possible weight, we can find the $r(e)$'s for all tree edges within a total of $O(m\alpha(m, n))$ time [6,17]. Then, for every $e \in t$ we set $D(W[e]) \leftarrow D(W[e]) \cap [-\infty, \overline{D}(W[r(e)])]$.

Now, if there is an entry $W[e]$ in the weights vector with $D(W[e]) = \emptyset$, the constraint has no solution.

5.2 Fixed Tree and Non-fixed Graph

When the graph is not fixed but the tree is, the node-set of the graph is determined by the tree. After filtering $D(G)$ to equate the node-sets and to contain all edges of t, we have that the endpoints of all non-tree edges, i.e., edges of $\overline{D}(G) \setminus t$, belong to t.

We apply the filtering step of the previous section to the weights of the non-tree edges. If this results in $D(W[e]) = \emptyset$ for some edge e, there are now two options: If $e \in \overline{D}(G) \setminus \underline{D}(G)$ we remove e from $\overline{D}(G)$, and if $e \in \underline{D}(G)$ then the constraint has no solution.

Next, we filter the weights of tree edges by applying the algorithm of the previous section on $\underline{D}(G)$. That is, for each tree edge e we find the weight of the lightest edge $r(e)$ in $\underline{D}(G)$ that connects $t_1(e)$ and $t_2(e)$, and shrink $D(W[e])$ as before. To see why it suffices to consider $\underline{D}(G)$, note that an edge e' in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one graph in $D(G)$, and in this graph, of course, e may be heavier than e'.

5.3 Fixed Graph and Non-fixed Tree

In Section 4.2, we handled the same problem with fixed weights by a variant of Kruskal's algorithm that required sorting the edges by weight. Since one weight interval can now overlap another, there is no longer a total order of the edge weights. We will show how to adapt the Kruskal-based algorithm to this case.

Phase 1: First, the algorithm considers edges in g. Instead of a list of edge-weights, we create a list of the endpoints of these edges' domains. We sort them in non-decreasing

[1] This problem is tightly related to the *MST sensitivity analysis problem*, where we are given a graph with fixed edge weights and its MST and need to determine, for each edge, the amount by which its weight can be perturbed without changing the property that the tree is an MST of the graph.

order, breaking ties in favor of lower bounds. Now, $\underline{D}(W[e])$ or $\overline{D}(W[e])$ is between $\underline{D}(W[e'])$ and $\overline{D}(W[e'])$ in the list if and only if $D(W[e]) \cap D(W[e']) \neq \emptyset$.

We then sweep over this list and examine each domain endpoint in turn. We say that an edge e is *unreached* before $\underline{D}(W[e])$ was processed, *open* if $\underline{D}(W[e])$ was already processed but $\overline{D}(W[e])$ was not, and *closed* after $\overline{D}(W[e])$ was processed. We maintain a graph H which is initially a set of n singleton nodes. In addition, we maintain a union-find data structure UF that represents the connected components of the subgraph H' of H that contains only closed edges. Initially, the union-find data structure also has n singletons. During the sweep, when processing $\underline{D}(W[e])$ where $e = (u, v)$, we mark e as *open*. If $Find(UF, u) = Find(UF, v)$, we place e in the *forbidden set*. Otherwise, we insert it into H. When processing $\overline{D}(W[e])$ where $e = (u, v)$, we mark e as *closed*. If e is a bridge in H, we place it in the *mandatory set* Finally, we perform $Union(UF, u, v)$.

After the sweep, all edges of the mandatory set are included in $\underline{D}(T)$ and all of the impossible edges are removed from $\overline{D}(T)$. Naturally, if this violates $\underline{D}(T) \subseteq \overline{D}(T)$, the constraint is inconsistent.

Phase 2: Next, the algorithm filters the weights of all edges. For a non-tree edge e, i.e., an edge $e \in g \setminus \overline{D}(T)$, the weight must be high enough so that e does not belong to any MST of g. In other words, the weight of e must be higher than the maximum weight of an edge on the tree path connecting its endpoints. In Section 4.1 we mentioned that King's algorithm can find the desired threshold when the tree is fixed. But how do we find the MST in $D(T)$ that minimizes the weight of the heaviest edge on the path between u and v? Clearly, the desired weight is the lower bound of the domain of this edge's weight. The following lemma implies that it suffices to find any MST in $D(T)$ (while assuming that each edge has the minimum possible weight), and apply King's algorithm to this MST.

Lemma 3. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let u and v be two nodes in g, let p_1 be the path between u and v in t_1 and let p_2 be the path between u and v in t_2. Then*

$$\max_{e \in p_1} w(e) = \max_{e \in p_2} w(e)$$

Proof. Consider the symmetric difference $p_1 \oplus p_2$ of the two paths. It consists of a collection of simple cycles, each of which consists of a subpath of p_1 and a subpath of p_2. Let c be one of these cycles. By the cycle property, any MST of g excludes a maximum weight edge from c (see Figure 3). Hence, there are at least two maximum edge weights c, one in $c \cap p_1$ and one in $c \cap p_2$. Since this is true for every cycle, we get that the maximum weights on each of the paths must be equal. □

For an edge $e \in \underline{D}(T)$, i.e., an edge which belongs to all MSTs, the weight must not be so high that there is a cycle in g on which this is the heaviest edge. In other words, we need to find an MST t of g that contains $\underline{D}(T)$ and which maximizes the minimum weight of a non-tree edge r_e that together with t forms a cycle that contains e. In Section 5.1 we mentioned that the desired threshold for all tree edges can be found

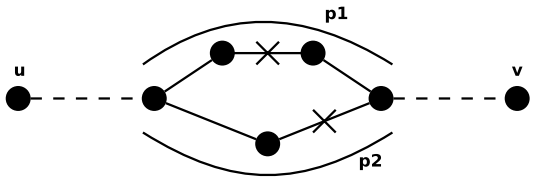

Fig. 3. Illustration of the proof of lemma 3. A cycle and its p_1 and p_2 subpaths, the maximum weight edges excluded from each MST.

in $O(m\alpha(m,n))$ time when the tree is fixed. Once again, we show that it suffices to apply the same algorithm to one of the MSTs in $D(T)$. Clearly, the desired weight is the upper bound of the domain of an edge weight. Furthermore, reducing the weight of any edge in g can only decrease the value of $w(r_e)$. The following lemma implies that it suffices to find any MST of g that contains $\underline{D}(T)$ (while assuming that each edge has the maximum possible weight), and use this MST to compute the thresholds for all tree edges.

Lemma 4. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let e be an edge in $t_1 \cap t_2$, let r_1 be the minimum weight edge that together with t_1 closes a cycle that contains e and let r_2 be the minimum weight edge that together with t_2 closes a cycle that contains e. Then*

$$w(r_1) = w(r_2)$$

Proof. It is known (see, e.g., Lemma 3 in [6]) that each of $t_1 \cup \{r_1\} \setminus \{e\}$ and $t_2 \cup \{r_2\} \setminus \{e\}$ is an MST of $g \setminus \{e\}$. Since all MSTs of a graph have equal weight, $w(r_1) = w(r_2)$. □

The time complexity of the algorithm described in this section is $O(Sort(m))$ to sort the endpoints of the domains of the edge weights and $O(m\alpha(m,n))$ for the modified Kruskal algorithm, incremental connectivity [16] and bridge computation [18], two MST computations and filtering of the edge weights.

5.4 General Case: Non-fixed Graph and Non-fixed Tree

We now turn to the most general case, in which all variables are not fixed. In the case of the $WBST$ constraint, we were able to find efficient bound consistency algorithms for special cases, but the most general case is NP-hard. In contrast, we will show that the MST constraint is not NP-hard in its most general form. The naïve bound consistency algorithm that we sketch in this section has a running time of $O(mn(m+\log n))$, which cannot be considered practical. We leave it as an open problem to find a more efficient method to approach the general case.

As before, we apply the filtering steps that follow from equality of the node-sets, inclusion of T in G and the connectedness of G and T. The main complication compared to the cases considered in the previous sections is in filtering the domains of the edge weights, because now the node-sets of the graph and the tree are not fixed. Again, we need to filter the lower bound weight of non-tree edges and the upper bound weight of tree edges.

Filtering the weights of non-tree edges. An edge $e = (u, v)$ in $\overline{D}(G) \setminus \overline{D}(T)$ cannot belong to the tree and therefore it must not be lighter than the maximum weight edge on the tree path from u to v. Let t be a tree and let $t(u, v)$ be the maximum weight edge on the path in t between u and v. We need to determine the minimum possible value of $t(u, v)$ over all trees in $D(T)$. Clearly, this weight would be a lower bound of the domain of some edge weight.

We set edge weights of all edge in $\overline{D}(T)$ to their lower bounds and contract the edges of $\underline{D}(T)$. In the remaining graph, we need to find a simple path from u to v that minimizes the maximum weight of an edge along it. This can be done in $O(m + n \log n)$ time by a dynamic programming approach that uses a variation of Djikstra's shortest-paths algorithm [3], where the sum of edge weights is replaced by a maximum computation. Since this computation needs to be repeated m times, once for every non-tree edge, the total time is $O(m(m + n \log n))$.

Filtering the weights of tree edges. A tree edge e must not be heavier than any non-tree edge that connects two nodes u and v such that e is on the tree path between u and v. To find the maximum possible weight of an edge $(u, v) \in \underline{D}(T)$, we will show how to check, for each edge $(x, y) \in \overline{D}(G) \setminus \underline{D}(T)$, whether there is a tree $t \in D(T)$ that contains (u, v), such that (x, y) is the minimum weight edge connecting the two components of $t \setminus \{(u, v)\}$.

Let $E' = (\underline{D}(T) \setminus \{(u, v)\}) \cup \{e | e \in \underline{D}(G) \cap \overline{D}(T) \wedge w(e) < w(x, y)\}$. If t exists, then each edge of E' is either in t or connects nodes that belong to the same connected component of $t \setminus \{(u, v)\}$. So we compute connected components of $G' = (Nodes(\overline{D}(G)), E')$. If x and y are in the same component, there is no such t. Otherwise, contract each connected component and merge the component of u with the component of v. Let G'' be the resulting graph. For a node $w \in G'$, we will refer to the node of G'' that represents the connected component of w by $CC(w)$. We will say that a node in G'' is mandatory if it represents a component in G' that contains at least one node from $\underline{D}(T)$.

Insert into G'' all the edges of $\overline{D}(T)$ which are not heavier than (x, y). It remains to determine whether we can make a tree that spans $CC(x)$, $CC(y)$, $CC(u) = CC(v)$ and all the mandatory nodes of G'', such that $CC(u)$ is on the path from $CC(x)$ to $CC(y)$. To do this, root G'' at the $CC(u)$ and find its dominators (in linear time) [9]. If $CC(x)$ and $CC(y)$ have a common dominator, such a tree does not exist. Otherwise, find two disjoint paths, one from $CC(u)$ to $CC(x)$ and one from $CC(u)$ to $CC(y)$. Then add edges to the tree to make it span all mandatory nodes.

This test takes linear time, and needs to be repeated at most m times for every edge in $\underline{D}(T)$, i.e., $O(mn)$ times. The total running time is therefore $O(m^2 n)$.

6 Conclusion

We have shown that it is possible to compute bound consistency for the MST constraint in polynomial time. For the special cases in which at least one of the variables is fixed, we found linear or almost-linear time algorithms. For the most general case, our upper bound is cubic and relies on a brute-force algorithm. It remains open whether the techniques we used for the simpler cases can be generalized to an efficient solution for MST in its most general form.

References

1. I. D. Aron and P. Van Hentenryck. A constraint satisfaction approach to the robust spanning tree problem with interval data. In *UAI*, pages 18–25, 2002.
2. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *CP-AI-OR 2005*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
3. E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. 1959.
4. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *CP 2005*, volume 3709 of *LNCS*, pages 211–225. Springer-Verlag, 2005.
5. G. Dooms and I. Katriel. Graph constraints in constraint programming: Weighted spanning trees. Research Report 2006-1, INGI, Belgium, 2006.
6. D. Eppstein. Finding the k smallest spanning trees. In *SWAT '90*, pages 38–47, London, UK, 1990. Springer-Verlag.
7. D. Eppstein. Setting parameters by example. *SIAM J. Comput.*, 32(3):643–653, 2003.
8. N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *FOCS 1996*, page 302, Washington, DC, USA, 1996. IEEE Computer Society.
9. L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *SODA 2004*, pages 869–878. SIAM, 2004.
10. R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*. Volume 53 of Annals of Discrete Mathematics., North Holland, Amsterdam, 1992.
11. V. King. A simpler minimum spanning tree verification algorithm. In *WADS*, volume 955 of *LNCS*, pages 440–448. Springer, 1995.
12. J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. American Mathematical Society*, 7:48–50, 1956.
13. I. Katriel N. Beldiceanu and X. Lorca. Undirected forest constraints. In *CP-AI-OR 2006*, volume 3990 of *LNCS*, pages 29–43. Springer-Verlag, 2006.
14. M. Sellmann. Cost-based filtering for shorter path constraints. In F. Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 694–708. Springer, 2003.
15. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
16. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
17. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
18. J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.