Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint

Jean-Charles Régin
regin@ilog.fr

ILOG Sophia Antipolis
Les Taissounières HB2,
1681 route des Dolines,
06560 Valbonne, France

Abstract. The weighted spanning tree contraint is defined from a set of variables X and a value K. The variables X represent the nodes of a graph and the domain of a variable $x \in X$ the neighbors of the node in the graph. In addition each pair $(variable, value)$ is associated with a cost. This constraint states that the graph defined from the variables and the domains of the variables admits a spanning tree whose cost is less than K. Efficient algorithms to compute a minimum spanning tree or to establish arc consistency of this constraint have been proposed. However, these algorithms are based on complex procedures that are rather difficult to understand and to implement. In this paper, we propose and detail simpler algorithms for checking the consistency of the constraint and for establishing arc consistency. In addition, we propose for the first time incremental algorithms for this constraint, that is algorithms that have been designed in order to be efficiently maintained during the search for solution.

1 Introduction

In this paper, we consider the weighted spanning tree constraint (wst constraint).

Several filtering algorithms for constraints based on graph theory and particularly on trees have been proposed. For instance, the robust spanning tree problem[1] with interval data has been addressed in [2]; the "tree" constraint has been studied in [3] (this constraint enforces the partionning of a digraph into a set of vertex-disjoint anti-arborescences), and recently, the "Not-Too-Heavy Spanning Tree" constraint has been introduced in [7]. This constraint is defined on undirected graph G and a tree T and it specifies that T is a spanning tree of G whose total weight is at most a given value I, where the edge weights are defined by a vector. The wst constraint is a simplified form of this constraint.

[1] From [2]:the robust spanning tree problem, given an undirected graph with interval edge costs, amounts to finding a tree whose cost is as close as possible of that minimum spanning tree under any possible assignment of costs.

In order to define it without introducing set variables or graph variables, we recall first the definition of a spanning tree and then we present the neighbor variables representation of a graph in CP.

A tree is a connected and acyclic graph. A tree $T = (X', E')$ is a spanning tree of $G = (X, E)$ if $X' = X$ and $E' \subseteq E$. In addition, if each edge of G is associated with a cost then the cost of a spanning tree of G is the sum of the costs of the edges of the tree.

The neighbor variables representation of a graph G consists of a variable set X corresponding to the nodes of G (i.e. x_i is associated with the node i in G and conversely) such that the domain of a variable x_i is equivalent to the neighbors of i in G (i.e. $j \in D(x_i) \Leftrightarrow j \in N(i)$ of G). Then, there is an equivalence between the cost of an edge in G and the cost of a value of a variable (i.e. $cost(i, j) = cost(x_i, j)$).

The weighted spanning tree constraint (wst constraint) is a constraint defined on the neighbor representation of a graph G each of whose edges has an associated cost, and associated with a global cost K. This constraint states that there exists in G a spanning tree whose cost is at most K.

This kind of constraint is not often present directly in real world applications, but it is used frequently as a lower bound of a more complex problems like hamiltonian path or node covering problems. For instance the minimum spanning tree is a well known bound of the traveling salesman problem.

It is straightforward to see that checking the consistency of this constraint is equivalent to finding a minimum spanning tree and to check if its cost is less than K. Moreover, arc consistency filtering algorithms are based on the computation for every edge e of the cost of the minimum spanning tree subject to the condition that the tree must contain e [7]. These two problems were solved for a long time. The search for a minimum spanning tree can be solved by several methods and we will consider here the Kruskal's algorithm. The second problem is close of another problem called "Sensitivity Analysis of Minimum Spanning Trees" [14]. The best algorithms solve this problem in linear time. Unfortunately they are quite complex to understand and to implement (see [6] or [11] for instance).

Therefore, in this paper, we propose a simpler and easy to implement consistency checking and filtering algorithms for the wst constraint, because, currently, there is no CP Solver which contains such a constraint. This algorithm is based on the creation of a new tree while running Kruskal's algorithm for computing an mst. Then, we find lowest common ancestors (LCA) in this tree by using the equivalence between the LCA and the range minimum query problem. A recent simple preprocessing leads to an $O(1)$ algorithm to find any LCA.

In addition, we will consider an important aspects of the algorithms which is usually ignored: the incremental aspect. This aspect is quite important in CP as shown for instance in[12] because the consistency checking algorithms and the filtering algorithms are systematically called during the search for solution. Thus, it is worthwhile to design algorithms that are able to exploit the previous computations in order to solve more quickly the problems they consider. In this

case and because the algorithms are called very often with only few modifications between two calls, any real saving is beneficial in practice.

The paper is organized as follows: First, we recall some concepts of graph theory and constraint programming. Then, we formally define the propositions on which the consistency and the arc consistency of the weighted spanning tree constraint are based. Next, we introduce a new data structure named tree of connected components which will lead us to propose a simple algorithm to establish arc consistency. Afterwards, we modify this algorithm in order to maintain it efficiently during the search for solution when some modifications happen or when a backtrack occurs. At last, we conclude.

2 Preliminaries

2.1 Graph Theory

A **tree** is a connected and acyclic graph. A tree $T = (X', E')$ is a spanning tree of $G = (X, E)$ if $X' = X$ and $E' \subseteq E$. The edges of E' are the **tree edges** of T and the edges of $E - E'$ are the **nontree edges** of T. A **forest** is a disjoint union of trees.

There are different methods to traverse all the nodes of a tree, we recall the one we will use in this paper: **the inorder traversal**. To traverse a non-empty binary tree in inorder, perform the following operations: 1. Traverse the left subtree in inorder. 2. Visit the root. 3. Traverse the right subtree in inorder.

2.2 Constraint Programming

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$. on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is called a **tuple on** $X(C)$. A value a for a variable x is often denoted by (x, a). Let C be a constraint. A tuple τ on $X(C)$ is **valid** if $\forall (x, a) \in \tau, a \in D(x)$. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $(x, a) \in \tau$. A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

Definition 1 *A* **weighted spanning tree constraint** *is a constraint C defined on X the neighbor variable representation of a graph G, and associated with* `cost` *a cost function on the edge of G, and an integer K such that*
$T(C) = \{ \tau$ *such that* τ *is a tuple on* $X(C)$
and the graph defined by τ *is a tree whose cost is less than* $K\}$
It is denoted by $wst(X, cost, K)$.

3 Consistency Checking

Proposition 1 *The constraint $wst(X, cost, K)$ is consistent if and only if the graph G defined by X has a minimum spanning tree T^* with $cost(T^*) \leq K$.*

We propose to use Kruskal's algorithm for searching for a mst. Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. The algorithm starts with a forest where each node in the graph is a separate tree. Then, it adds edges which join two nodes belonging to different trees of the forest and merges the two trees into one. The particularity of the algorithm is that the edges are selected in regards to their costs. For each step the edge which connects two distinct trees and whose cost is minimum is selected. Thus, Kruskal's algorithm can be easily implemented by traversing the edges in nondecreasing order of their costs and by adding edges connecting two disjoint trees, until all nodes of the graph are in the same connected component. The main issue to obtain an efficient implementation is to detect whether two nodes belong to the same tree or not. This operation can be efficiently performed by using the well known union-find data structure of Tarjan [15]. By combining the path compression and the union by rank heuristics, m operations on the union-find performed on a set of n elements run in $O(m\alpha(m, n))$ time [15], where $\alpha(m, n)$ is a functional inverse of Ackerman's function. Thus, we have:

Property 1 *If the list of edges ordered by non decreasing cost is available then Kruskal's algorithm can be implemented in $O(m\alpha(m, n))$.*

Algorithm 1: Kruskal's algorithm for computing a minimum spanning tree

```
getCCRoot(i): return find(i)
mergeCC(r_i, r_j): union(r_i, r_j)
initCC(n): for i = 1 to n do makeSet(i)
addEdge(ccT, T, r_i, r_j, {i, j})
    mergeCC(r_i, r_j)
1   updateCCTree(ccT, r_i, r_j, {i, j})
    add {i, j} to T

MinimumSpanningTree(nonDecrEC): (mst,ccTree)
    initCC(n)
2   initCCTree(ccT, n)
    T ← ∅
    for each {i, j} ∈ nonDecrEC while |T| < n − 1 do
        r_i ← getCCRoot(i); r_j ← getCCRoot(j)
        if r_i ≠ r_j then addEdge(ccT, T, r_i, r_j, {i, j})
    return (T, ccT)
```

Algorithm 1 is a possible implementation of Kruskal's algorithm using the union-find data structure. The algorithm returns the largest forest that can be built. At

this point, we recommend to ignore lines 2 and 1 and parameter ccT. Functions MAKESET, FIND and UNION are the classical union-find functions:
MAKESET(x) :{ $p[x] \leftarrow x$; $rank[x] \leftarrow 0$ }
FIND(x): { if $p[x] \neq x$ then $p[x] \leftarrow$ FIND($p[x]$) endif; return $p[x]$ }
LINK(x, y) :{ if $rank[x] > rank[y]$ then $p[y] = x$ else $p[x] = y$ endif
. if $rank[x] = rank[y]$ then $rank[y] \leftarrow rank[y] + 1$ endif }
UNION(x, y): {LINK(FIND(x),FIND(y)) }

4 Arc Consistency Filtering Algorithm

For each nontree edge $\{i, j\}$, we have to find the cost of a minimum spanning tree subject to the condition that the tree must contain the edge $\{i, j\}$. First, we recall the Optimality Conditions of a mst:

Theorem 1

- **[Path Optimality Condition]** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following path optimality conditions: for every nontree edge $\{i, j\}$ of G, $cost(i, j) \geq cost(u, v)$ for every edge $\{u, v\}$ contained in the path in T^* connecting nodes i and j.*
- **[Cut Optimality Condition]** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following cut optimality conditions: for every tree edge $\{i, j\}$ of G, $cost(i, j) \leq cost(u, v)$ for every edge $\{u, v\}$ contained in the cut formed by deleting edge $\{i, j\}$ from T^*.*

We will call $\{i, j\}$-tree, a tree which must contain the edge $\{i, j\}$. Then:

Property 2 *Let $G = (X, E)$ be a graph, $\{i, j\} \in E$ be an edge of G, and v be the minimum of the edge costs minus 1. Then, a minimum spanning $\{i, j\}$-tree of G is the mst of G when the cost of $\{i, j\}$ is equal to v. The cost of the minimum spanning $\{i, j\}$-tree is then equal to the cost of the mst plus $cost(i, j) - v$.*

proof: Since $\{i, j\}$ is the edge with the minimum cost when its cost is equal to v, then it will necessary be a tree edge of any mst. $\odot$

For the sake of clarity we will consider that T^* is a minimum spanning tree of G. The filtering algorithm is based on the following Proposition [7]:

Proposition 2 *Let $\{i, j\}$ be a nontree edge of G, and $\{u, v\}$ be the edge with the maximum cost contained in the path in T^* connecting nodes i and j.*
The tree T corresponding to the tree T^* in which the edge $\{u, v\}$ has been replaced by the edge $\{i, j\}$ is a minimum spanning $\{i, j\}$-tree of G.

proof: If the edge $\{i, j\}$ is added to the tree then a cycle is created and the Path Optimality Condition implies that the edge of the cycle having the largest cost must be removed. Since an $\{i, j\}$-tree is wanted and from Property 2, we consider that $\{i, j\}$ has the smallest cost. So the edge that must be removed is $\{u, v\}$ because it has the largest cost. Thus a tree T is obtained. This tree satisfies the Path Optimality Condition for

all the nontree edges because T^* does and $\{i,j\}$ is considered as having the smallest cost. T also satisifies the path optimality condition for $\{u,v\}$. $\odot$

Let $minEC(T)$ and $maxEC(T)$ be the cost of the edge of T having respectively the minimum and the maximum cost. We deduce two corollaries:

Corollary 1 *All the nontree edges $\{i,j\}$ such that*
(i) $cost(i,j) > K - cost(T^*) + maxEdgeCost(T^*)$ are not consistent with C
(ii) $cost(i,j) \leq K - cost(T^*) + minEdgeCost(T^*)$ are consistent with C

So, we can immediately delete all the edges satisfying Corollary 1.(i) and avoid studying the edges satisfying Corollary 1.(ii). For the other edges, we have:

Definition 2 *Let $\{i,j\}$ be a nontree edge of G which does not satisfy Corollary 1, and $\{u,v\}$ be the edge with the maximum cost contained in the path in T^* connecting nodes i and j. Then, $\{u,v\}$ is called a* **support** *of $\{i,j\}$, and $S(u,v)$ is the list of nontree edges that are supported by $\{u,v\}$.*

Proposition 3 *Let $\{i,j\}$ be an nontree edge of T^* which does not satisfy Corollary 1, $\{u,v\}$ be the support of $\{i,j\}$.*
$\{i,j\}$ is consistent with C if and only if $cost(i,j) \leq K - cost(T^*) + cost(u,v)$.

We propose to efficiently compute the supports by introducing a new tree while running Kruskal's algorithm,

4.1 Tree of Connected Components Merges

Kruskal's algorithm proceeds by merging disjoint trees. Each time an arc is added to the spanning tree, two trees are merged together. We propose to explicitly represent these operations by creating a specific tree called: **connected component tree** or **ccTree**. Every merge is represented by a node in the ccTree.

A bottom-up creation of this tree is used. The leaves correspond to the nodes of the graph, because, in Kruskal's algorithm, initially each node defines a tree. Each time an edge is added to the mst by Kruskal's algorithm, a new ccTree node is created. This ccTree node has two children: one for each tree (so the ccTree is binary) that have been merged. Each tree created in Kruskal's algorithm has a pointer to the ccTree node which represents it. The ccTree contains at most $2n-1$ nodes. Figure 1 gives a minimum spanning tree of a graph and Figure 2 shows a tree of connected components obtained after running Kruskal's algorithm on this graph. The ccTree involves the following data:

- $ccT.size$: the current number of nodes of the tree
- $ccT.p[r]$: the ccTree leaf corresp. to the canonical element of node r of G
- $ccT.left[k]$ and $ccT.right[k]$: the left and the right child of the ccTree node k
- $ccT.parent[k]$: the parent of the ccTree node k
- $ccT.Gedge[k]$: the edge of G which lead to the creation of the ccTree node k
- $ccT.inorder[i]$: the i^{th} ccTree node visited by the inorder traversal
- $ccT.pos[k]$: the inorder index of the ccTree node k
- $ccT.height[k]$: the height (distance from the root) of ccTree node k

```
0
633 0
257 390 0
 91 661 228 0
412 227 169 383 0
150 488 112 120 267 0
 80 572 196  77 351  63 0
134 530 154 105 309  34 29  0
259 555 372 175 338 264 232 249 0
505 289 262 476 196 360 444 402 495 0
353 282 110 324  61 208 292 250 352 154 0
324 638 437 240 421 329 297 314 95  578 435 0
 70 567 191  27 346  83  47  68 189 439 287 254 0
211 466  74 182 243 105 150 108 326 336 184 391 145 0
268 420  53 239 199 123 207 165 383 240 140 448 202  57 0
246 745 472 237 528 364 332 349 202 685 542 157 289 426 483 0
121 518 142  84 297  35  29  36 236 390 238 301  55  96 153 336 0
```

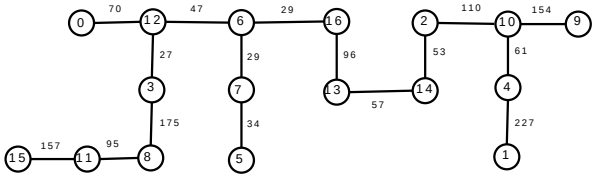
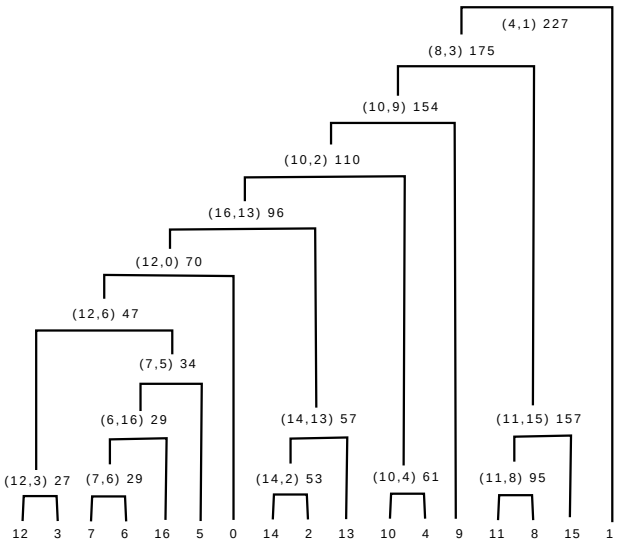


Fig. 1. The lower triangular matrix of problem gr17 of the TSPLIB and a Minimum Spanning Tree of this Graph



indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(12,3)	(7,6)	(6,16)	(7,5)	(12,6)	(14,2)

indice	23	24	25	26	27	28	29	30	31	32
value	(14,13)	(10,4)	(12,0)	(11,8)	(16,13)	(10,2)	(10,9)	(11,15)	(8,3)	(4,1)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
H	10	9	10	6	10	9	10	8	10	7	10	5	10	4	10	9	10	8	10	3	10	9	10	2	10	1	10	9	10	8	10	0	10
inorder	12	17	3	21	7	18	6	19	16	20	5	25	0	27	14	22	2	23	13	28	10	24	4	29	9	31	11	26	8	30	15	32	1

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
value	12	32	16	2	22	10	6	4	28	24	20	26	0	18	14	30	8	1	5	7	9	3	15	17	21	11	27	13	19	23	29	25	31

Fig. 2. Tree of Connected Components Merges of mst of problem gr17. The nodes that are not leaves contain the edge of G and its cost. Array of indices, array H, inorder and Pos are also represented.

The creation of the ccTree can be easily done while running Kruskal's algorithm as shown by Algorithm 1 (See Line 1.). Function UPDATECCTREE (ccT: tree, r_i , r_j, $\{i, j\}$) creates a new node in the ccTree whose children are $p[r_i]$ and $p[r_j]$ and with $Gedge = \{i, j\}$, Function INITCCTREE creates n leaves corresponding to the node of G, and Function INORDERTREETRAVERSAL performs an inorder tree traversal of the ccTree.

Once the ccTree is built, the support of any edge $\{i, j\}$ is the Gedge associated with the ccTree node created when the tree containing i and the tree containing j have been merged together. This node is the least common ancestor of the ccTree node i and the ccTree node j.

Definition 3 Lowest Common Ancestor *(LCA)*
For nodes u and v of tree T , query $LCA_T(u, v)$ returns the lowest common ancestor of u and v in T , that is, it returns the node farthest from the root that is an ancestor of both u and v.

Proposition 4 *Let i and j be two nodes of G, and ccT be the connected component tree built while running Kruskal's algorithm on G.*
Then, $ccT.Gedge[LCA_{ccT}(i, j)]$ is the edge that merged the tree of i and the tree of j while running Kruskal's algorithm on G.

proof: First, note that a node i of G corresponds to the node i of ccT which is leaf. From the definition of ccT each node which is not a leaf corresponds to the merge of two trees. Therefore, the lowest common ancestor of two leaves of ccT corresponds to the merge of two disjoint trees of G that are identified by the extremities of the edge merging them. The *Gedge* data associated with this ccTree node contains it. $\odot$

Corollary 2 *Let T^* be a mst of G, ccT be the connected component tree built while running Kruskal's algorithm and $\{i, j\}$ be a nontree edge of T^*. Then $\{u, v\} = ccT.Gedge[LCA_{ccT}(i, j)]$ is the support of $\{i, j\}$.*

proof: By definition of the ccTree and the LCA, the LCA corresponds to the arc with the greatest cost in the path from i to j in T^*.$\odot$

The are several methods to solve directly the LCA problem, starting with [1] and improved by [10] and [13]. Unfortunately these methods are complex especially when the binary tree is not well balanced, which happens in our case. Another approach to solve the LCA problem in a non direct way has been introduced in [9]: the LCA problem is linearly equivalent to the Range Minimum Query Problem. Thus, by efficiently solving the RMQ problem, we obtain an efficient solution of the LCA problem.

Definition 4 Range Minimum Query *(RMQ)*
Let A be a length n array of numbers. For indices i and j between 1 and n, query $RMQ_A(i, j)$ returns the index of the smallest element in the subarray $A[i, ..., j]$.

We will use the simple and nice transformation of LCA to RMQ proposed by [8]: "Let T be a rooted binary tree with n nodes.

- First perform an inorder tree walk in T and store it in an array $inorder[1, n]$.

• Store the heights of each node: $H[i]$ is the height of node $inorder[i]$ in T.
• Let Pos be the inverse array of $inorder$, i.e., $inorder[Pos[i]] = i$. It is easy to see that $LCA_T(v, w) = inorder[RMQ_H(Pos[v], Pos[w])]$: the elements in $inorder$ between $Pos[v]$ and $Pos[w]$ are exactly the nodes encountered between v to w during an inorder tree walk in T, so the RMQ returns the position k in H of the shallowest such nodes. As the LCA of v and w must be encountered between v and w during the inorder tree walk, $LCA(v, w) = inorder[k]$".

Figure 2 gives an example of ccTree, inorder traversal, H and Pos arrays. For instance, $LCA_T(10, 6) = inorder[RMQ_H(Pos[10], Pos[6])]$ which is equal to $inorder[RMQ_H(20, 6)] = inorder[19] = 28$ that is the index of the edge $\{10, 2\}$.

Now, the goal is to solve some RMQ requests as fast as possible. Harel and Tarjan [10] have shown that if several requests will be made then it is worthwhile to spend some time on preprocessing the tree in order to answer future queries faster. In [8] an $O(n)$ preprocessing is given, and with it any RMQ problem request for two values can be answered in $O(1)$. Unfortunately, this algorithm is quite complex and the authors doubt about its advantages in practice. Thus, we will use the much simpler algorithm proposed by [4]. It has a simple preprocessing step which is in $O(n \log(n))$ and solve each problem $RMQ(i, j)$ in $O(1)$ with only computing the minimum of two values. It is based on the fact that the RMQ problem for two values i and j can be easily solved if we have previously solved the RMQ problems for four values i, u, v, j such that $i \leq v \leq u \leq j$:

Property 3 *[4] Given i, j, u, v four integers such that $i \leq v \leq u \leq j$,*
$r_{iu} =$RMQ(i, u) and $r_{vj} =$RMQ(v, j). Then,
If $A[r_{iu}] \leq A[r_{vj}]$ then RMQ$(i, j) = r_{iu}$ *else* RMQ$(i, j) = r_{vj}$

Then, the nice idea is to work only with intervals whose length is a power of two, because any interval $[i, j]$ can be splitted into two such intervals:

Corollary 3 *[4] Given i, j two integers such that $i \leq j$, and $k = \lfloor \log_2(i) \rfloor$,*
$r_1 =$RMQ$(i, i + 2^k - 1)$ and $r_2 =$RMQ$(j - 2^k + 1, j)$. Then,
If $A[r_1] \leq A[r_2]$ then RMQ$(i, j) = r_1$ *else* RMQ$(i, j) = r_2$

If all the intervals whose length is a power of two are precomputed, then:

Corollary 4 *[4] Let A be an array of n values, and $M[i][k] =$RMQ$(i, i+2^k-1)$ with $i = 1..n$ and $k = 0..\lfloor \log_2(n) \rfloor$. Then, each* RMQ$(i, j)$*, with $1 \leq i < j \leq n$ can be computed in $O(1)$.*

The number of intervals $[i, p]$ with $p \leq n$ and whose length is a power of 2 is in $O(\log(n))$. Since there are n starting values, the overall complexity is in $O(n \log(n))$. Algorithm 2 is a possible implementation of the RMQ Problem. Note that this algorithm uses the arrays $Log2Array$ and $Pow2Array$ which contain respectively for a value k the result of mathematical operations: $\lfloor \log(k) \rfloor$ and 2^k. The values of these arrays can be computed in $O(n + \log(n))$ and this can be done once for all when the constraint is defined.

Algorithm 2: AC Filtering Algorihtm based on LCA Problem

```
PRECOMPUTERMQ(Rmq, n)
    for i = 1 to n do M[i][0] ← i
    for j = 1 to Log2Array[n] do
        for i = 1 to n − Pow2Array[j + 1] do
            minL ← Rmq.M[i][j − 1]
            minR ← Rmq.M[i + Pow2Array[j − 1]][j − 1]
            if Rmq.A[minL] ≤ Rmq.A[minR] then Rmq.M[i][j] ← minL
            else Rmq.M[i][j] ← minR
RANGEMINIMUMQUERY(Rmq, i, j): Integer
    logWidth ← Log2Array[j − i + 1]
    minL ← Rmq.M[i][logWidth]
    minR ← Rmq.M[j − Pow2Array[logWidth] + 1][logWidth]
    if Rmq.A[minL] ≤ Rmq.A[minR] then return minL
    else return minR
LOWESTCOMMONANCESTOR(ccT, Rmq, i, j): Integer
    p_i ← ccT.pos[i]; p_j ← ccT.pos[j]
    return ccT.inorder[RANGEMINIMUMQUERY(Rmq, p_i, p_j)]
COMPUTEALLSUPPORTS(ccT, SE)
    REDUCECCTREE(ccT, nonIncrEC)
    INORDERTREETRAVERSAL(ccT)
    Rmq.A ← ccT.height
    PRECOMPUTERMQ(Rmq, ccT.num)
    for each {i, j} ∈ SE do
        lca ← LOWESTCOMMONANCESTOR(ccT, Rmq, i, j)
        {u, v} ← ccT.Gedges[lca]
        APPEND {i, j} to S(u, v); support(i, j) ← {u, v}
COMPUTEPENDINGEDGES(nonIncrEC, ccT): return ∅
COMPUTEENTERINGEDGES(nonIncrEC, T_1, T_2): Edge Set
    return {{i, j} ∈ nonIncrEC s.t. cost(i, j) > K − cost(T_2) + minEC(T_2)}
ACFILTER(nonIncrEC, oldT, oldccT, R, T, ccT)
    for each {i, j} ∈ nonIncrEC while cost(i, j) > K − cost(T) + maxEC(T)
    do DELETEEDGE({i, j}, nonIncrEC)
    SE ← COMPUTEENTERINGEDGES(nonIncrEC, oldT, T)
    SE ← SE ∪ COMPUTEPENDINGEDGES(R, oldccT)
    COMPUTEALLSUPPORTS(ccT, SE)
    for each {u, v} ∈ T do
        for each {i, j} ∈ S(u, v) while cost(i, j) > K − cost(T) + cost(u, v) do
            DELETEEDGE({i, j}, nonIncrEC)
```

The preprocessing step is in $\theta(n \log(n))$ because the computation needs to be systematically done. However, it can be transformed into a maximum complexity because we can consider less than n nodes. The nodes that are not an extremity of an edge for which we need to compute a support are not needed in the ccTree,

so we can remove them. In order to maintain a binary tree, after a removal each node having only one child is contracted that is the node is deleted and its child becomes the child of its father. These operations have an amortized cost of $O(1)$ per removal. Thus, the number of nodes of the $ccTree$ is less than or equal to $2n$ and so the complexity of the preprocessing step of the RMQ Problem is in $O(n \log n)$. Function REDUCECCTREE implements this idea.

The main function for implementing an AC filtering algorithm are given in Algorithm 2. The first call of a weighted spanning tree constraint can be implemented as follows (we consider that the set of edges has been sorted first):

$(T, ccT) \leftarrow$ MINIMUMSPANNINGTREE($nonDecrEC$)
if $|T| < n - 1$ **or** $cost(T) > K$ then trigger a failure
ACFILTER($nonIncrEC - T, \emptyset, \emptyset, \emptyset, T, ccT$)

Proposition 5 *Arc consistency of the weighted spanning tree constraint can be established in* $O(n + m + n \log(n))$

5 Maintenance during the Search

First, we consider the incremental aspects of the problem, that is we study the computation of the consistency of the constraint or the establishement of arc consistency when some modifications happen. Then, we will consider the problem of the restoration of the data structures when a backtrack occurs.

Note that the list of ordered edges is easy to maintain because we have just to manage the deletion of elements. So if any edge knows its previous and its next element in the ordered list then it can be removed from that list in $O(1)$.

There are two possible events: either a nontree edge is removed or a tree edge is removed. In the first case, the minimum spanning tree remains a minimum spanning tree and the condition of consistency or arc consistency remain satisified (See Propositions 1 and 2).So, there is nothing to do. This case may happen frequently because there are m edges and only $n - 1$ tree edges. The latter case is more complex and deserves a careful study, because a new spanning tree must be computed, so the ccTree may change and the lists of supported values also. This is the purpose of the next section.

5.1 Consistency Checking

If we accept an $O(n)$ complexity when some modifications happen, there is no need to maintain the union find and the ccTree data structures. In fact, each involves at most $2n$ elements. The new minimum spanning tree can be built from the current one by using its tree edges, and some computations can be saved if we rerun Kruskal's algorithm:

Proposition 6 *Let* $T^* = (X, A)$ *be a mst of* G *and* $\{i, j\}$ *a tree edge. There exists a mst of* $G - \{i, j\}$ *containing the set of edges* $A - \{i, j\}$.

proof: Let be $\{u, v\}$ be the edge with the minimum cost contained in cut forming by deleting $\{i, j\}$ from T^*. Let T be the tree corresponding to T^* where $\{i, j\}$ has been replaced by $\{u, v\}$ then T satisifies the Cut Optimality Condition of $G - \{i, j\}$ and so is a minimum spanning tree of $G - \{i, j\}$ and T contains the edges $A - \{i, j\}$. $\odot$

Proposition 7 *Let $T^* = (X, A)$ be a mst of G and $R = \{r_1, ..., r_k\}$ be a subset of the tree edge set. There exists a mst of $G - R$ containing the set of edges $A - R$ and a set $S = \{s_1, ..., s_k\}$ of edges such that for each $i = 1..k$ $r_i \leq s_i$.*

proof: by induction on the number of element of R. From Prop. 6, this is true for 1 that is for $R = r_1$, because the cost of the mst of $G - r_1$ is greater than the cost of T^* so $s_1 \geq r_1$. Suppose it is true until i, that is for $R = r_1, ...r_i$. This means that we can build a tree T containing the edges of $A - \{r_1, ...r_i\}$. Now from Prop.6 if the edge r_{i+1} is removed then we can build another tree that wil contain the edges of T minus r_{i+1}. This tree will also contain an arc s_{i+1} such that $cost(s_{i+1}) \geq cost(r_{i+1})$ because T is a mst of $G - \{r_1, ..., r_i\}$. Therefore this is true for $i+1$ and the proposition holds. $\odot$

Consider that the sets A and R of edges are ordered w.r.t. the cost of the edges. While traversing the edges of E to build the new mst T, we can add the edges of $A - R$ and avoid considering some edges of E. Suppose that we search for an edge s_i replacing the edge r_i and that we have found replacement edges for all the edges of R smaller than r_i. If s_i is smaller than r_{i+1} then we can immediatly add to T all the edges of $A - R$ between r_i and r_{i+1} and we can search for a replacement of r_{i+1} from that position in E (See Algorithm 3.).

5.2 AC Filtering Algorithm

The computation of a new mst changes the boundaries of Corollary 1. Thus, some edges can be immediately deleted and some supports must be computed for the first time for some other edges, named **entering** edges. In addition, the ccTree has been rebuilt when checking the consistency, so some support lists may be no longer correct. Consider ccT^* the ccTree associated with the old mst T^* and ccT the ccTree associated with the new mst T. We need first to run again the preprocessing of the RMQ problems for ccT. Then, we need to identify the edges for which their support is no longer valid or for which the validity must be verified. These edges are called **pending** edges, These are the edges belonging to any support list $S(u, v)$ where the node of ccT^* associated with the edge $\{u, v\}$ or a descendant of this node in ccT^* is associated with an edge of G which has been removed. Once these lists have been identified, it is necessary to compute the supports for all the edges contained in these lists and then to recompute new lists of supports. Then, all the lists of supports can be checked. This is required because the cost of the mst changed and so some edges that were consistant may become inconsistant. These checks of consistency of edges within support lists can be greatly improved if the elements are sorted, due to the structure of Proposition 3, so we need to sort the elements contained in the union of all the unvalid lists of support. Fortunately, it is possible to achieve such a sort in a very efficient way:

Algorithm 3: Recomputation of a mst after modifications.

```
RecomputeMST(T, R, nonDecrEC): (mst, ccTree)
    initCC(n)
    initCCTree(ccT, n)
    A is the edge set of T; newT is empty
    {u, v} ← first(A)
    while {u, v} ≤ last(A) do
        ne ← next(A, {u, v})
        if {u, v} ∈ (A − R) then
            r_i ← getCCRoot(i); r_j ← getCCRoot(j)
            addEdge(ccT, newT, r_i, r_j, {i, j})
        else
            cpt ← cpt + 1
            for each {i, j} ∈ nonDecrEC from {u, v} while cpt > 0 do
                r_i ← getCCRoot(i); r_j ← getCCRoot(j)
                if {i, j} ≥ ne then
                    if {i, j} = ne then addEdge(ccT, newT, r_i, r_j, {i, j})
                    ne ← next(A, ne)
                else
                    if r_i ≠ r_j then
                        addEdge(ccT, newT, r_i, r_j, {i, j})
                        cpt ← cpt − 1
        {u, v} ← ne
    return (newT, ccT)
```

Proposition 8 *Let $G = (X, E)$ be a graph where E is ordered, and OE be the array of ordered indices of E (i.e. $OE[e] = k$ means that the edge e in in the k^{th} position in E). Let F be a subset of E and $n = |X|, m = |E|, m' = |F|$. Then, we can sort the elements of F with the same order as for E in $O(n + m')$.*

proof : Consider a Least Significant Digit Radix Sort and b a base (or radix) used to represent numbers. Such a sort is able to sort an array of numbers ranging from 0 to $\Delta - 1$ in $\log_b(\Delta)$ calls to a stable sort [5]. A stable sort like counting sort [5] is able to sort num numbers ranging from 0 to $b - 1$ in $O(num + b)$. Therefore the time complexity of a radix sort can be expressed as: $\log_b(\Delta) \times O(num + b)$. The edge set E is already sorted, and we can access for each edge to its position in E, so instead of considering the value associated with each element, it is equivalent to consider the position of the element in E. There are m possible positions, so to order F we need to order elements taking their value in $[0..m - 1]$. With a Radix Sort combined with a counting sort we can sort F in $\log_b(m) \times O(m' + b)$, because $\Delta = m$ and $num = m'$ in our case. If we use n as base b then we have $\log_n(m) \times O(m' + n)$. We have $m \leq n^2$ so $\log_n(m) \leq \log_n(n^2) = 2\log_n(n) = 2$. Therefore $\log_n(m) \times O(m' + n)$ is equivalent to $2 \times O(m' + n)$, that is $O(m' + n)$. $\odot$

Algorithm 4: Incremental AC Filtering Algorithm

```
computeEnteringEdges(nonIncrEC, T1, T2): Edge Set
    if cost(T2) ≥ cost(T1) then
        return {{i, j} ∈ nonIncrEC s.t.
        K − cost(T2) + minEC(T2) < cost(i, j) ≤ K − cost(T1) + minEC(T1)}
    else
        return {{i, j} ∈ nonIncrEC s.t.
        K − cost(T2) + maxEC(T2) < cost(i, j) ≤ K − cost(T1) + maxEC(T1)}
computePendingEdges(R, ccT): Edge Set
    SE ← ∅
    add to UN the nodes of ccT associated with edges of R
    for each x ∈ UN do
        SE ← SE ∪ S(ccT.Gedge[x])
        S(ccT.Gedge[x]) ← ∅
        if ccT.parent[x] ∉ UN then add(ccT.parent[x], UN)
    sort(SE)
    return SE
```

When used during the search for a solution the consistency checking and the arc consistency filtering of a wst constraint can be implemnted as follows (See also Algorithm 4). Let R be the set of edges of T that are deleted:

$(newT, newccT) \leftarrow$ RecomputeMST$(T, R, nonDecrEC)$
if $|newT| < n - 1$ **or** $cost(newT) > K$ then trigger a failure
ACFilter$(nonIncrEC - newT, T, ccT, R, newT, newccT)$
$T \leftarrow newT$; $ccT \leftarrow newccT$

5.3 Restoration

Algorithm 5: Restoration of a mst.

```
RestoreMST(T, P, R, nonDecrEC): (mst, ccTree)
    initCC(n)
    initCCTree(ccT, n)
    A is the edge set of T; newT is empty
    for each {i, j} ∈ (A − R) ∪ P do
        r_i ← getCCRoot(i); r_j ← getCCRoot(j)
        addEdge(ccT, newT, r_i, r_j, {i, j})
    return (newT, ccT)
```

There are two possible ways to deal with backtracks: either the state is exactly restored or an equivalent state is defined [12]. With a boundary based constraint the optimal solution at a node n may not be an optimal solution for the ancestors of n, therefore it is needed to restore the same state when a backtrack occurs. For the wst constraint, it means that we need to save all the modifications affecting the current minimum spanning tree. Then, we can easily restore the previous spanning tree because we know the set P of edges that have been deleted at a

given search node and the set R of edges that have been added to the mst for this node (See Algorithm 5). All the restored edges must also be added to the pending edges. Here is a possible procedure to restore the previous state:

$(newT, newccT) \leftarrow$ RestoreMST$(T, P, R, nonDecrEC)$
if $|newT| < n - 1$ **or** $cost(newT) > K$ then trigger a failure
ACFilter$(nonIncrEC - newT, T, ccT, R, newT, newccT)$
$T \leftarrow newT$; $ccT \leftarrow newccT$

6 Conclusion

In this paper we have presented simpler algorithms for checking the consistency and for establishing arc consistency of the weighted spanning tree constraint. We have detailed, by giving the pseudo-code, several versions of these algotihms that are able to exploit the modifications that happen during the search for a solution in order to save some computations . The complexity of all the proposed filtering algorithms neither exceeds $O(m + n \log(n))$ which is quite good.

References

1. A. Aho, J. Hopcroft, and J. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976.
2. I. Aron and P. Van Hentenryck. A constraint satisfaction approach to the robust spanning tree problem with interval data. In *Proc. of UAI*, pages 18–25, 2002.
3. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *Proceedings of CPAIOR05*, pages 64–78, 2005.
4. M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57:75–94, 2005.
5. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
6. B. Dixon, M. Rauch, and R. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.
7. G. Dooms and I. Katriel. The not-too-heavy spanning tree constraint. In *Proceedings of CPAIOR07*, pages 59–70, 2007.
8. J. Fischer and V. Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM'06)*, pages 36–48, 2006.
9. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of STOC*, pages 135–143, 1984.
10. D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
11. G. Manku. An o(m + n log* n) algorithm for sensitivity analysis of minimum spanning trees, 1994. citeseer.ist.psu.edu/manku94om.html.
12. J-C. Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *Proceedings of CP'05*, pages 520–533, 2005.
13. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
14. R. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters*, 14(1):30–33, 1982.
15. R.E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.