Weighted Spanning Tree Constraint with Explanations

Diego de Uña[1], Graeme Gange[1], Peter Schachte[1], and Peter J. Stuckey[1,2]
{d.deunagomez@student.,gkgange@,schachte@,pstuckey@}unimelb.edu.au

[1] Department of Computing and Information Systems – The University of Melbourne
[2] National ICT Australia, Victoria Laboratory

Abstract. Minimum Spanning Trees (MSTs) are ubiquitous in optimization problems in networks. Even though fast algorithms exist to solve the MST problem, real world applications are usually subject to constraints that do not let us apply such methods directly. In these cases we confront a version of the MST called the "Weighted Spanning Tree" (WST) in which we look for a spanning tree in a graph that satisfies other side constraints and is of minimum cost. In this paper we implement this constraint using a lower bound and learning to accelerate the search and thus reduce the solving time. We show that having this propagator is tremendously beneficial for solvers and we show the benefits of learning.

1 Introduction

Given a connected weighted graph $G = (N, E)$, the Minimum Spanning Tree (MST) T of G is a connected acyclic sub-graph of G that contains all the nodes in N and is of minimum weight. Finding the MST of a graph can be done using Kruskal's algorithm (among others) which is $\mathcal{O}(|N|log(|E|))$. Nevertheless, many interesting variants of the MST are NP-hard. In these variants, there are side constraints that make these algorithms unusable.

Some examples where side constraints make the MST problem NP-hard are the capacitated MST [5, 17], the degree-constrained MST [14], the min-degree MST [2], the constrained MST [19], or the diameter-constrained MST [1]. These and other variants can be found in the real world. For instance, cable layout for offshore wind farms [13] combines the capacitated MST, the degree-constrained MST and an extra constraint disallowing cable crossing.

In Constraint Programming (CP), the Weighted Spanning Tree (WST) constraint is defined as follows: given a graph $G = (N, E)$ and a weight function ws that maps every edge $e \in E$ to an integer called the *weight* of e, find a tree T that is a subgraph of G, spans all nodes in N and is of cost at most w (w being an integer variable). The decisions made by this constraint are Boolean variables c_e representing for each edge $e \in E$ whether it is chosen to be part of T or not. Let $B = \{c_e | e \in E\}$, we write the constraint $wst(N, E, ws, B, w)$. Because this is a constraint, it can be used in combination with other constraints and therefore applied to the above stated optimization problems.

The first appearance of this constraint was in [9] (called “Not-too heavy” spanning tree). Their work was followed up in [21] with a simpler algorithm for propagation maintaining the same strength for the propagator. They re-named the constraint “WST”, which is the term that we will use. Here the propagation was proved to be arc-consistent. Later, in [22], the *ccTree* data-structure was improved to decrease the complexity of their algorithms. Similar work was done in [8], although this constraint forced to solution to be a *minimum* spanning tree. The contribution of these papers are the filtering algorithms they provide, but no implementation or experiments are reported. Nonetheless, in Constraint Programming, constant factors in the complexity are crucial and the asymptotic complexity of their algorithms gives only partial information on performance. Also, no previous work explored the use of explanations in this useful constraint.

In this paper we present our implementation of the WST constraint in the CP solver Chuffed [7]. We use learning [16] to accelerate the search. We show that the explanations on this global constraint are tremendously beneficial in practice. We compare our implementation to the one available in the Choco3 CP solver [18] and show the benefits of learning.

We illustrate the use of this constraint on the Diameter-Constrained MST (DCMST) problem, because it has been recently addressed in Constraint Programming by [15] and has a large number of applications in wireless network routing [3], telecommunications [26], distributed mutual exclusion in computer networks [20] and data compression [4]. For this problem there has been work on both approximation and exact algorithms. In approximations, [11] presented an approach using Variable Neighbourhood Search, followed by another heuristic approach [12]. For exact solutions [23] presented a Mixed Integer Programming formulation of the problem that was later improved in [10]. The latest exact algorithm was presented by [15] using CP and it outperforms all other approaches known to the authors. Our approach to the DCMST is also CP, so it is only comparable to the last one. Nevertheless, the solver they used is not the same as ours, and thus comparisons (especially in time) should be considered with care.

Section 2 briefly introduces Lazy Clause Generation. Section 3 describes our algorithms and implementation of the WST constraint, including the computation of explanations. Section 4 summarizes our experimental results on the DCMST.

2 Lazy Clause Generation

Lazy Clause Generation (LCG, [16]) is a technique by which CP solvers can learn from what they have explored in the search space. Constraints can be transformed into a number of clauses over Booleans, and this is typically how SAT solvers work. The idea of LCG is to make propagators generate these clauses on the fly when they propagate. These clauses capture the reason for propagation and thus we say they “explain” propagation. These *explanations* are then given to the solver that uses them as a way of remembering what propagators inferred. This way, we run a relatively expensive algorithm once, do an inference and remember it for the rest of the solving process. On the other hand, propagators might need to do some extra computation to compute the explanations.

3 WST propagator with explanations

In our problem, the decision variables are the edges: which edges form the tree and which edges do not. We say that an edge is *mandatory* if it has been set to be part of the tree. We call *forbidden* the edges that have been set to not be part of the tree. Other edges are *undecided*. Let M be the set of mandatory and F the set of forbidden edges.

Here we present our WST propagator with explanations. We first introduce a novel lower bound with explanations followed by a propagation rule (which was already introduced in [21]).

We define a *substitute edge* of an edge e in a spanning tree $T = (N, T_E)$ as any edge e_s such that $(N, T_E \backslash \{e\} \cup \{e_s\})$ is a spanning tree. Also, following the definition of [21], given a tree T and a non-tree edge $e = (i, j)$, let e' be the edge of maximum cost in the path from i to j in T. Then e' is called the *support* of e.

3.1 Lower bound with explanations

Assume we are looking for a solution of cost w lower than K. When we branch (i.e. we make a decision) we can compute a lower bound of the problem that will tell us if a better solution can exist in this branch. If that's not the case, we can stop the search. This is known as branch-and-bound.

The most accurate lower bound for w in the WST propagator is naturally the MST of the graph given the decisions so far. That is, the tree $T^* = (N, E^*)$ of minimum weight W_{T^*} such that $M \subseteq E^*$ and $E^* \cap F = \emptyset$.

It is easy to see that applying Kruskal's algorithm where the edges in M have been pre-added and the edges in F are not used yields T^*.

Now, if $W_{T^*} \geq K$ then no solution of cost lower than K exists in the current search space, and we can cut the search. A trivially correct explanation is $\bigwedge_{e \in F} \neg c_e \wedge \bigwedge_{e \in M} c_e \Rightarrow W_{T^*} \geq K$, but it is possible to build a better explanation.

Let F_c be the set of forbidden edges e_F such that $T^* \cup \{e_F\}$ forms a cycle where e_F is not the most expensive edge. Let M_S be the set of edges $e \in M$ having some substitute e' such that $ws(e') < ws(e)$. Let S_S be a mapping $M_S \mapsto E$ from each edge in M_S to the substitute of minimum weight for that edge. We then select a subset $M_H \subseteq M_S$ such that the inequality $\sum_{e \in M_H} (ws[S_S[e]] - ws[e]) + W_{T^*} \geq K$ holds. Note that multiple such sets M_H may exist.

A better explanation is given by Theorem 1.

Theorem 1. *A correct explanation for the failure of* $wst(N, E, ws, B, w)$ *is:*

$$\bigwedge_{e \in F_c} \neg c_e \wedge \bigwedge_{e \in M \backslash M_H} c_e \Rightarrow W_{T^*} \geq K$$

Proof. **Forbidden edges:** Clearly, $F_c \subseteq F$. Let $e = (u, v) \in F \backslash F_c$. By definition of F_c, e is the most expensive edge in the cycle formed by $T^* \cup \{e\}$. Because the queue in Kruskal's algorithm is sorted in increasing order, the path P between u and v in T^* was already built before considering e. Therefore, whether e is forbidden or not does not affect the cost of P and consequently does not affect W_{T^*} and the explanation $\bigwedge_{e \in F_c} \neg c_e \wedge \bigwedge_{e \in M} c_e \Rightarrow W_{T^*} \geq K$ holds.

Mandatory edges: By construction, M_H is a set of edges that, when removed and substituted by the best possible edge available, the cost of the tree is still higher than K. Therefore, the edges in M_H do not need to be in the explanation for it to hold. □

Note that because several sets M_H may exist, different explanations can be computed. Evaluating which explanation is better than another is highly dependent on the instance of the problem. We ran different tests and could not determine a way of choosing M_H that dominated others in all cases. In our final implementation we start by putting the cheapest edges in M_H.

The algorithm to detect failure and compute the explanation is Algorithm 1. To construct explanations, we use the Rerooted-Union-Find data structure described in [25]. This is a modification of a classic Union-Find that allows the user to retrieve paths between nodes. Lines 7 to 9 pre-add all the mandatory edges. Lines 10 to 20 follow the classic Kruskal's algorithm with some modifications. Lines 12 and 17 add to the explanation any forbidden edge that should have been used. Lines 14 and 15 compute the cheapest substitute for each mandatory edges (if any). Once the tree T^* is computed, we build M_H in lines 22 and 23, leaving all the other mandatory edges (that have substitutes) in the explanation in line 25. The final explanation for $W_{T^*} \geq K$ is the set X. The complexity of the algorithm is $\mathcal{O}(|E|(|N| + log(|E|)))$.

The same explanations can be used for failure if $cost > K$:

$$\bigwedge_{e \in F_c} \neg c_e \wedge \bigwedge_{e \in M \backslash M_H} c_e \wedge [\![w < K]\!] \Rightarrow \mathit{false}$$

In the example of Figure 1, we are looking for a solution of cost less than $K = 27$. W_{T^*} is 31, so we must fail. When we consider edge e_1, the fact that it is mandatory causes no trouble, as there is no other substitute to this edge that would connect h. When we consider e_4, we must skip it because it is forbidden, which means that we will use a more expensive edge to reach c (here e_8). When considering e_6, e and g are already connected by a path containing the mandatory edges e_7, e_9 and e_{10} and the undecided edge e_3. Therefore e_6 is the substitute of all of them. We later compute that: $W_{T^*} - ws[e_7] + ws[e_6] = 30 > K$, then $30 - ws[e_9] + ws[e_6] = 28 > K$ and lastly $28 - ws[e_{10}] + ws[e_6] = 21 < K$. Therefore, the explanation will be $\neg c_{e_4} \wedge c_{e_{10}} \wedge [\![w < K]\!] \Rightarrow \mathit{false}$.

3.2 Propagation rule with explanations

We use the propagation rule exposed in Proposition 3 of [21], that is: given the best possible tree T^* and an upper bound for the solution K such that $W_{T^*} < K$,

Algorithm 1 Computing the lower bound with explanation.

```
procedure MANDATORY_KRUSKAL($G = (N, E), M, F, K$)
    $Q \leftarrow sort(E)$
    $uf \leftarrow RerootedUF()$
    $c \leftarrow 0,\ cost \leftarrow 0$
    $X \leftarrow \emptyset, sub \leftarrow array(|E|, nil)$
    for all $e = (u, v) \in M$ do                                  $\triangleright$ Pre-add mandatory edges
        $uf.unite(u, v)$
        $c \leftarrow c + 1;\ cost \leftarrow cost + ws[e]$
    for all $e = (u, v) \in Q$ do                                          $\triangleright$ (in order)
        if $\neg uf.connected(u, v) \wedge e \in F$ then
            $X \leftarrow X \cup \{\neg c_e\}$                    $\triangleright$ Should add $e$, but it is forbidden
        else if $uf.connected(u, v)$ then
            for all $ep \in uf.path(u, v)$ do
                $sub[ep] = min\_w(sub[ep], e)$
                if $ws[ep] > ws[e] \wedge e \in F$ then
                    $X \leftarrow X \cup \{\neg c_e\}$                         $\triangleright$ e would be cheaper
        else if $c < |N| - 1 \wedge \neg uf.connected(u, v)$ then
            $uf.unite(u, v)$
            $c \leftarrow c + 1;\ cost \leftarrow cost + ws[e]$
    for all $e \in M \wedge sub[e] \neq nil$ do
        if $cost - ws[e] + ws[sub[e]] \geq K$ then
            $cost \leftarrow cost - ws[e] + ws[sub[e]]$                          $\triangleright$ $e \in M_H$
        else if $ws[sub[e]] \neq ws[e]$ then
            $X \leftarrow X \cup \{c_e\}$                                        $\triangleright$ $e \notin M_H$
    return $[X \Rightarrow [\![w \geq cost]\!]]$
```

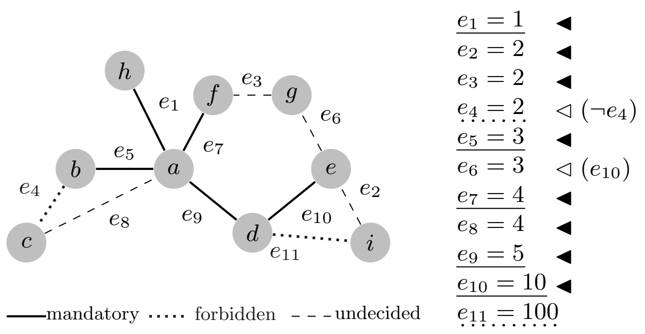

Fig. 1. Example of a graph during solving. The weights are indicated on the right. Symbol '◀' indicates an edge that was used in the solution, whereas '◁' indicates edges that should have been used (accompanied by the explanation).

for any non-tree edge e^* of support e', e^* can be part of the solution if and only if $W_{T^*} - ws[e'] + ws[e^*] < K$. That is, e^* is a valid substitute of e'. If this is not the case, we must remove e^* from the possible edges since using it would increase the weight of T^* above the upper bound K. It is easy to see that the previous explanation applies as well in this case. Let $M'_H = ((M \backslash M_H) \backslash \{e'\}) \cup \{e^*\}$.

$$\bigwedge_{e \in F_c} \neg c_e \wedge \bigwedge_{e \in M'_H} c_e \wedge [\![w < K]\!] \Rightarrow \mathit{false}$$
$$\Leftrightarrow \bigwedge_{e \in F_c} \neg c_e \wedge \bigwedge_{e \in M'_H \backslash e^*} c_e \wedge [\![w < K]\!] \Rightarrow \neg e^*$$

We execute this rule after the previously described algorithm in the case where no failure is detected.

4 The Diameter Constrained Minimum Spanning Tree

The DCMST is formally defined as follows: given a graph $G = (N_G, E_G)$ find a sub-graph $T = (N_T, E_T)$ of G such that T is a tree, $N_T = N_G$ and the longest distance between any two nodes in T is at most D (called the *diameter* of T). The distance between two nodes u and v is the number of nodes in the path from u to v.

4.1 Modeling DCMST

This problem is separated in two cases whether D is even or odd. If D is even, then there exists a node r that is the root of T and the height of all the other nodes has to be at most $\lfloor D/2 \rfloor$. If D is odd, there exists an edge $e = (a, b)$ that acts as the root of the tree (e is therefore in the tree), meaning that the height of a and b is zero and all the other nodes must have at most height $\lfloor D/2 \rfloor$. Notice that r, a and b are not given in the input: these are variables.

We used the same model as [15] with the only addition of our propagator. The matrix *adj* gives for each node the set of neighbour nodes. For the DCMST-specific constraints, we use an array of heights of nodes h, and an array of parenthood of nodes p. Two variables a and b are the end-nodes of the edge that acts as root in the odd case, or are both the root in the even case (in that case, $a = b$). The model is $\mathit{minimize}(w)$ such that:

$$\mathit{wst}(N, E, ws, B, w) \tag{1}$$
$$D \ \mathit{mod}\ 2 = 0 \Leftrightarrow a = b \tag{2}$$
$$(h[a] = 0 \wedge p[a] = b) \wedge (h[b] = 0 \wedge p[b] = a) \tag{3}$$
$$\forall n \in N \backslash \{a, b\},\ h[n] = h[p[n]] + 1 \tag{4}$$
$$\forall n \in N \backslash \{a, b\},\ p[n] \in \mathit{adj}[n] \tag{5}$$
$$\forall e = (u, v) \in E,\ c_e \Leftrightarrow p[u] = v \vee p[v] = u \tag{6}$$

Constraint 2 states that in the even case a and b are the same node (the root r). Constraint 3 forces a and b to be at height 0 and be each others parents. Constraint 4 makes every node (other than the root(s)) be one level below its parent. Constraint 5 forces each node to chose a parent that is adjacent to it. Constraint 6 links the edge variables of the graph with the parenthood relations.

Although our main intention is to compare the improvement that the WST propagator and explanations bring to the solver, we also compare our work to [15] (we name their results "NRS") as their results are the state of the art in DCMST as far as we can tell. They used a Pentium 4, 2.8GHz and 2GB of RAM.

For better comparison, we implemented the exact same search strategy they describe in their paper (Section 3, Figure 2). First, for each node n we compute the sum s_n of the shortest paths from n to any other node. Then, we associate to each pair of nodes (a, b) the minimum of s_a and s_b, noted $s_{(a,b)} = min(s_a, s_b)$. The search is as follows. Start by taking each pair of nodes (a, b) in increasing order of $s_{(a,b)}$. Then for each possible value of the height (from 1 to $\lfloor D/2 \rfloor$), remove that value from the domain of all the nodes (when possible) taking the nodes in decreasing order of the shortest path to either a or b. Here "shortest path" is in weight of the edges.

They use a dominance rule in the search, which we converted into a dominance-breaking constraint [6] in our model, for ease of implementation: $\forall \{e_1 = (u, v), e_2 = (u, y)\} \in E^2,\ ws[e_1] < ws[e_2] \wedge h[v] \leq h[y] \Rightarrow p[u] \neq y$. This states that if it is cheaper to connect u to v than to y and the height of v is lower than the height of y, we can connect u to v with a lower cost. This is because if using e_2 does not violate the diameter constraint, neither does e_1.

4.2 Experimental results

We run our experiments on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM. We used 5 minutes as the time limit. The results from NRS are extracted from their paper where they used the solver IBM OPL. Benchmarks can be found in [24]. We give different versions in CHUFFED: NOPROP uses learning but does not use our propagator, NOEXPL uses our propagator without learning , EXPL uses the propagator with explanations from Section 4, and NAIVEEXPL uses our propagator with naive explanations (i.e. all fixed elements in the graph are in the explanation). All use the same strategy.

As we can see in Table 1, the use of the propagator is absolutely beneficial. The total time is improved by 48.02% when using the propagator without explanations against no propagator. Furthermore, our version with explanation (EXPL) is 90.5% faster than the version without explanations (NOEXPL) and 95.1% faster than the version with no propagator at all. Also, our total time is 36.6% shorter than NRS. Most of the tests with NOPROP and CHOCO3 got to the optimal solution, but timed-out when proving optimality. This also illustrates the need for this propagator.

In CP, the number of nodes represents the size of the search space explored before proving optimality. Here we see an obvious dominance of EXPL as it almost always has less nodes than other versions. It also has an improvement on the total number of nodes for all benchmarks of 99.4% over NOPROP and 99.0% over NOEXPL. Additionally, it has an improvement of 96% over NRS.

The comparison between EXPL and NAIVEEXPL shows that computing our explanations is worthwhile. The NAIVEEXPL uses the same algorithms as described throughout this paper, only the explanations contain all the fixed c_n

Instance			NoProp		NoExpl		NaiveExpl		Expl		Choco3		NRS (IBM OPL)	
$\|N\|$	$\|E\|$	D	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
15	105	4	6825	0.45	784	0.23	448	0.2	**447**	0.19	6256	1.92	1044	**0.08**
15	105	5	35322	2.28	1921	0.39	1003	0.39	**1001**	0.38	301269	44.36	2850	**0.22**
15	105	6	133259	10.31	5997	0.63	2235	0.48	**2101**	0.45	160445	37.06	6960	**0.28**
15	105	7	258317	22.91	5873	0.54	2312	0.41	**2221**	0.41	2182510	300	8240	**0.38**
15	105	9	493166	39.80	6049	0.47	1968	0.21	**1731**	**0.19**	2623006	300	11743	0.47
15	105	10	550536	40.93	24259	1.95	2872	0.39	**2831**	**0.29**	898948	156.63	11830	0.41
20	190	4	192965	20.39	2651	1.86	**1261**	1.70	1266	1.69	200651	82.51	3143	**0.20**
20	190	5	1869837	186.76	9387	4.85	4452	4.56	**4432**	4.48	2064050	300	18283	**1.06**
20	190	6	2585912	300	49673	14.97	9018	4.55	**8462**	4.08	862115	300	35383	**2.03**
20	190	7	2661381	300	16690	4.67	**4252**	2.04	4288	1.97	1857850	300	19142	**0.97**
20	190	9	2433234	300	157236	34.43	6336	1.71	**5972**	**1.60**	1738525	300	119906	5.01
20	190	10	2628419	300	315618	52.61	9050	3.96	**8645**	**3.56**	1067170	300	151969	6.08
25	300	4	1898689	300	20202	58.53	**6166**	17.10	6217	17.1	592738	300	28842	**1.48**
25	300	5	2415919	300	86662	93.57	32787	80.62	**26547**	68.88	1553235	300	37608	**2.83**
25	300	6	2262702	300	402861	300	16150	17.66	**15147**	**15.99**	847691	300	534222	39.14
25	300	7	2045173	300	449104	210.13	76272	87.72	**61098**	63.68	1448142	300	812957	**56.06**
25	300	9	1929801	300	462886	300	21195	18.66	**19724**	**17.43**	1270399	300	2655810	114.14
25	300	10	1961836	300	620555	261.71	**21453**	11.86	21565	**11.50**	586552	300	1126130	55.47
20	50	4	14548	0.48	1219	**0.05**	558	0.05	558	**0.05**	4489	0.61	**389**	**0.05**
20	50	5	55748	2.58	307392	10.87	2258	0.26	**2227**	0.24	426762	40.91	3611	**0.17**
20	50	6	52217	2.34	68384	0.75	1574	0.10	**1475**	**0.08**	41892	5.28	2678	0.13
20	50	7	66676	3.45	25043	0.68	1381	0.12	**1238**	**0.09**	1389117	133	1975	0.14
20	50	9	274583	16.59	14016	0.33	**1117**	0.06	1261	**0.06**	3820792	300	13040	0.45
20	50	10	310688	18.93	**410**	**0.01**	564	0.03	564	0.02	329333	42.87	17937	0.64
40	100	4	3426079	300	45199	6.27	**13766**	4.41	13901	**4.30**	1180714	300	130480	5.44
40	100	5	3261615	300	9596291	300	36496	14.22	**26970**	9.69	3196955	300	161961	**7.31**
40	100	6	4836734	300	8161773	300	10708	2.59	**5037**	**0.84**	1851687	300	91022	4.72
40	100	7	4709441	300	5979528	300	38153	11.03	**18504**	**4.49**	2989047	300	778669	34.38
40	100	9	3646022	300	4468371	300	88837	25.12	**36572**	**7.08**	2873734	300	769161	40.16
Total			47017644	4868.2	31306034	2530.50	414642	312.21	**302002**	**240.81**	38365804	6245.15	7556985	379.84

Table 1. Comparison in time (seconds) and nodes for the DCMST models.

and c_e. This makes the explanations more strict and thus less reusable. As we would expect, this makes the explanations much longer: the average length in the explanations for NaiveExpl is 128.88 literals, whereas the length of our explanations is 73.18 literals in average. We see the consequences of this in the Table 1: naive explanations most often slow down the solving step. The version Expl is 22.9% faster and has 27.2% less nodes.

We observe that our propagator dominates specially when the diameter is big. This is because in that case, the lower bound is more accurate as it violates fewer diameter constraints. When the diameter is small, Algorithm 1 is not aware of it and just computes an MST thus rapidly violating the diameter constraints.

5 Conclusion

In this paper we have given an efficient algorithm to compute explanations for the lower bound for the WST constraint, and we have implemented an already existing propagation rule in our solver. Our major contribution is the computation of explanations that, as we can see in the experiments, are absolutely beneficial to solve large instances of optimization problems on spanning tree.

Acknowledgement. Diego de Uña thanks "la Caixa" Foundation for partially funding his Ph.D. studies at The University of Melbourne.

References

1. Achuthan, N., Caccetta, L., Caccetta, P., Geelen, J.: Algorithms for the minimum weight spanning tree with bounded diameter problem. Optimization: techniques and applications 1(2), 297–304 (1992)
2. Akgün, İ., Tansel, B.Ç.: Min-degree constrained minimum spanning tree problem: New formulation via miller–tucker–zemlin constraints. Computers & Operations Research 37(1), 72–82 (2010)
3. Bala, K., Petropoulos, K., Stern, T.E.: Multicasting in a linear lightwave network. In: INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE. pp. 1350–1358. IEEE (1993)
4. Bookstein, A., Klein, S.T.: Compression of correlated bit-vectors. Information Systems 16(4), 387–400 (1991)
5. Chandy, K.M., Lo, T.: The capacitated minimum spanning tree. Networks 3(2), 173–181 (1973)
6. Chu, G., Stuckey, P.J.: Dominance breaking constraints. Constraints 20(2), 155–182 (2015)
7. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
8. Dooms, G., Katriel, I.: The minimum spanning tree constraint. In: Principles and Practice of Constraint Programming-CP 2006, pp. 152–166. Springer (2006)
9. Dooms, G., Katriel, I.: The not-too-heavy spanning tree constraint. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 59–70. Springer (2007)
10. Gruber, M., Raidl, G.R.: A new 0–1 ILP approach for the bounded diameter minimum spanning tree problem. In: Gouveia, L., Mourão, C. (eds.) Proceedings of the 2nd International Network Optimization Conference 2005. vol. 1, pp. 178–185. Lisbon, Portugal (2005), https://www.ac.tuwien.ac.at/files/pub/gruber-05.pdf
11. Gruber, M., Raidl, G.R.: Variable neighborhood search for the bounded diameter minimum spanning tree problem. In: Hansen, P., Mladenovi, N., Pérez, J.A.M., Batista, B.M., Moreno-Vega, J.M. (eds.) Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search. Tenerife, Spain (2005), https://www.ac.tuwien.ac.at/files/pub/gruber-05a.pdf
12. Gruber, M., Raidl, G.R.: (meta-) heuristic separation of jump cuts in a branch&cut approach for the bounded diameter minimum spanning tree problem. In: Matheuristics, pp. 209–229. Springer (2010)
13. Klein, A., Haugland, D., Bauer, J., Mommer, M.: An integer programming model for branching cable layouts in offshore wind farms. In: Modelling, Computation and Optimization in Information Systems and Management Sciences, pp. 27–36. Springer (2015)
14. Narula, S.C., Ho, C.A.: Degree-constrained minimum spanning tree. Computers & Operations Research 7(4), 239–249 (1980)
15. Noronha, T.F., Ribeiro, C.C., Santos, A.C.: Solving diameter-constrained minimum spanning tree problems by constraint programming. International Transactions in Operational Research 17(5), 653–665 (2010)
16. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009), http://dx.doi.org/10.1007/s10601-008-9064-x
17. Papadimitriou, C.H., Vazirani, U.V.: On two geometric problems related to the travelling salesman problem. Journal of Algorithms 5(2), 231–246 (1984)

18. Prud'homme, C., Fages, J.G., Lorca, X.: Choco3 Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2014), http://www.choco-solver.org
19. Ravi, R., Goemans, M.: The constrained minimum spanning tree problem. Algorithm TheorySWAT'96 pp. 66–75 (1996)
20. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Transactions on Computer Systems (TOCS) 7(1), 61–77 (1989)
21. Régin, J.C.: Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 233–247. Springer (2008)
22. Régin, J.C., Rousseau, L.M., Rueher, M., van Hoeve, W.J.: The weighted spanning tree constraint revisited. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 287–291. Springer (2010)
23. dos Santos, A.C., Lucena, A., Ribeiro, C.C.: Solving diameter constrained minimum spanning tree problems in dense graphs. Springer (2004)
24. de Uña, D.: Weighted spanning tree benchmarks (2015), http://people.eng.unimelb.edu.au/pstuckey/wst/
25. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Steiner tree problems with side constraints using constraint programming. In: Proceedings of the Thertieth AAAI Conference on Artificial Intelligence. p. to appear. AAAI Press (2016)
26. Wang, S., Lang, S.: A tree-based distributed algorithm for the k-entry critical section problem. In: Parallel and Distributed Systems, 1994. International Conference on. pp. 592–597. IEEE (1994)