The Weighted Spanning Tree Constraint Revisited*

Jean-Charles Régin[1], Louis-Martin Rousseau[2], Michel Rueher[1], and Willem-Jan van Hoeve[3]

[1] I3S, CNRS, University of Nice-Sophia Antipolis
[2] CIRRELT, University of Montreal
[3] Tepper School of Business, Carnegie Mellon University

1 Introduction

The *weighted spanning tree constraint*, or *wst*-constraint, is defined on an edge-weighted graph G and a value K. It states that G admits a spanning tree with weight at most K [3, 4]. It can be applied to network design problems as well as routing problems, in which it serves as a relaxation. In this work, we assume that we can represent the mandatory and possible edges that can belong to a solution to the *wst*-constraint, e.g., using a subset-bound set variable as in [3].

Dooms and Katriel [3] consider a version of the *wst*-constraint in which the weights of the edges are also variable. They propose several filtering algorithms, including one for the version of the *wst*-constraint that we consider in this paper. Subsequently, a more practical and incremental filtering algorithm for this constraint was proposed by Régin [4].

In this work, we extend the algorithm of Régin [4] in several ways. First, we revisit the computation of the 'replacement cost' of tree edges, and present an algorithm with an almost linear time complexity. Second, we take mandatory edges into account; that is, edges that belong to every spanning tree having a weight at most K or that are imposed by the user. Third, we discuss the incremental behavior of the algorithms when mandatory edges are introduced.

2 Existing Approaches

The task of propagating the *wst*-constraint consists of a check for consistency, the removal (filtering) of inconsistent edges from the domain of possible edges, and potentially fixing edges that must belong to every solution. An important practical aspect is the incrementality of the algorithms, i.e., efficiently re-using data structures and solutions from one propagation event to the next. The consistency of the *wst*-constraint can easily by verified by finding a minimum-spanning tree in the graph, using a classical method such as Prim's algorithm or Kruskal's

* This work was partially supported by the European Community's 7th Framework Programme (FP7/2007-2013). It was started when L.-M. Rousseau and W.-J. van Hoeve were visiting the University of Nice-Sophia Antipolis (June/July 2009).

algorithm. Assuming that the edges are sorted by non-decreasing weight, this can be done in almost linear time [1]. Identifying inconsistent edges (that cannot participate in a spanning tree of weight at most K) is more involved, however. Dooms and Katriel [3] observed that inconsistent edges can be detected as follows [5]. Let T be a minimum spanning tree, and let (i, j) be a non-tree edge that we wish to evaluate. We now find the maximum-weight edge on the unique i-j path in T. If replacing that maximum-weight edge with (i, j) yields a tree of weight more than K, (i, j) is inconsistent. Similar reasoning can be applied to determine whether a tree edge is mandatory, i.e., when replacing it would yield always a tree of weight more than K [3]. Therefore, the detection of inconsistent and mandatory edges amounts to computing the 'replacement cost' of the edges. Régin [4] also applies the replacement cost for non-tree edges to detect inconsistent edges, but tree edges (and mandatory edges) were not considered.

Several algorithms have been proposed to compute the replacement cost of the edges, for example by Tarjan [5] and Dixon, Rauch, and Tarjan [2]. These algorithms allow to compute all replacement costs in time $O(m\alpha(m, n))$ on a graph with n nodes and m edges, where $\alpha(m, n)$ is the inverse Ackermann function stemming from the complexity of the 'union-find' algorithm [6]. Other approaches, such as those referenced by [3] are based on (or resemble) the algorithms of [5] or [2]. Even though these algorithms allow to find the replacement costs in almost linear time theoretically, the added complexity may not offset the potential savings in practice, as argued by Tarjan [5]. Moreover, it is not obvious how to apply the algorithms incrementally. Therefore, Régin proposed a different algorithm running in $O(n + m + n \log n)$ time [4]. We next briefly describe the main components of this algorithm for later use.

Régin [4] applies Kruskal's algorithm to find a minimum spanning tree. That is, we start from a forest consisting of all nodes in the graph. We then successively add edges, whereby each added edge joins two separate trees. We ensure that the next selected edge has minimum weight among all edges whose extremities are not in the same tree. We use a so-called *ccTree* ('connected component tree') to represent these merges. The leaves of the ccTree are the original graph nodes, while the internal nodes of the ccTree represent the merging of two trees (or connected components), defined in the order in which the edges were added to the tree. An internal node thus represents the edge with which two components have been merged; see Figure 1a and 1b for an example. Therefore, the ccTree contains $n - 1$ internal nodes, where n is the number of nodes in the graph. The computation of the replacement cost of a non-tree edge (i, j) can now be done by finding the lowest common ancestor (LCA) of nodes i and j in the ccTree: the weight of (i, j) minus the weight of the edge corresponding to the LCA is exactly the replacement cost of (i, j). We refer to [4] for further details.

3 Computing the Replacement Cost of Tree Edges

We next present an algorithm that computes the replacement costs of tree edges in time $O(m\alpha(m, n))$. This is the same time complexity as the algorithm proposed by Tarjan [5]. We note that the latter algorithm follows as a corollary from a generic (and relatively complex) algorithm presented in [5]. Our contribution is a description of a more practical algorithm, specific to the problem of computing replacement costs, having the same time complexity. We will apply the algorithm to detect mandatory edges.

Let $G = (V, E)$ be the graph under consideration, with a 'weight' function $w : E \to \mathbb{R}$, and let T be a minimum spanning tree of G. For a subset of edges $S \subseteq E$, we let $w(S)$ denote $\sum_{e \in S} w(e)$. The *replacement cost* of an edge e in T is defined as $w(T_{\not e}) - w(T)$, where $T_{\not e}$ is a minimum spanning tree of $G \setminus e$. It represents the marginal increase of the weight of the minimum spanning tree if e is not used. It can be shown that the new minimum spanning tree can be obtained by replacing e with exactly one other edge, which is called the *replacement edge*. In fact, the replacement cost of e is the weight of its replacement edge minus the weight of e itself.

Let us first describe a basic algorithm for computing the replacement costs for tree edges. We start by computing a minimum spanning tree T, and we label all tree edges as 'unmarked'. We then consider the edges of the graph, ordered by non-decreasing weight. If we encounter a non-tree edge (i, j), we do the following. First, observe that there is a unique i-j path in T, and (i, j) serves as replacement edge for all unmarked edges on this path. Therefore, we will mark a tree edge as soon as we have identified its first replacement edge. For example, in Figure 1, the first non-tree edge that we consider is $(3, 4)$. We thus label the tree edges $(1, 3)$ and $(1, 4)$ as marked, with associated replacement cost 1 and 2, respectively. The next non-tree edge is $(1, 2)$, which is used to mark tree edge $(2, 4)$ with associated replacement cost 2 (edge $(1, 4)$ is already marked).

It can be shown that this basic algorithm computes the replacement costs of all tree edges. Unfortunately, its time complexity is rather high: we may need up to n steps to identify the unmarked edges, which gives an overall time complexity of $O(mn)$. Fortunately, we can efficiently reduce this complexity by *contracting* the marked edges of the tree, i.e., we merge the extremities of marked tree edges. This contraction will be performed by using a 'union-find' data structure [6, 1].

First, we root the minimum spanning tree, i.e., we designate an arbitrary root node, and we organize the nodes in a directed tree with parent information. In addition, each node is associated with a pointer p to its parent in the union-find data structure. Initially the pointer p of every node points to the node itself. When an unmarked edge is discovered, we 'contract' the edge by letting the pointer p now point to its father. We then apply the classical 'find' function, associated with its classical updates. That is, the pointers of the union-find data structure are used to traverse the path between the two extremities of a non-tree edge. Note that we move up in parallel in the tree from the two extremities. We stop when the same node is reached by the two traversals (one from each extremity). For example in Figure 1, suppose we let node 1 be the root of the tree. After processing the first non-tree edge $(3, 4)$, the updated pointers are $p(3) = 1$ and $p(4) = 1$. For the next non-tree edge $(1, 2)$, the algorithm directly proceeds from the parent of 2 (node 4) to $p(4)$, which is node 1.

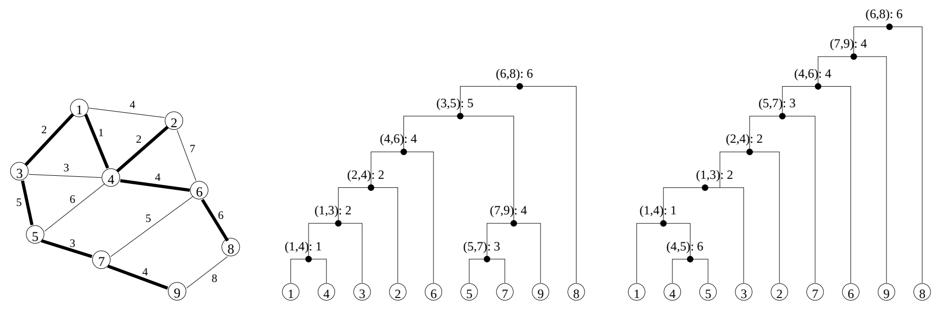
The advantage of this method is that it is easy to implement. Moreover, we will have at most $n-1$ contractions because the tree contains $n-1$ edges. In addition we will have at most m requests, thus we obtain the classical union-find complexity of $O(m\alpha(m,n))$. We note that the replacement cost for tree edges can be used to identify mandatory edges: an edge is mandatory if its replacement cost is higher than $K - w(T)$, see also [3].

4 Mandatory Edges and Incrementality

We next consider the implications of introducing mandatory edges on the maintenance of the minimum spanning tree, in the context of the propagation algorithms of Régin [4]. First, observe that we need to update our minimum spanning tree, and other necessary data structures when both tree edges and non-tree edges become mandatory. If a non-tree edge becomes mandatory (for example as a result from inference by other constraints), we clearly need to find a new minimum spanning tree that includes this edge. If a tree edge becomes necessary, it can remain in the tree, but we do need to update the data structures to forbid this edge from being used as a replacement edge. Recall that the main data structure used in [4] is the ccTree. We propose two different methods to update the minimum spanning tree and the ccTree upon the addition of mandatory edges. The first method is based on recomputation. The second method is based on 'repairing' the current minimum spanning tree and ccTree.

The first method can be implemented in a straightforward manner by using the existing algorithms. Namely, we can associate an appropriate low weight value to the mandatory edges, which will then be added first to the minimum spanning tree (assuming that we use Kruskal's algorithm, that adds the edges ordered by non-decreasing weight). After all mandatory edges have been added, the algorithm will proceed with the other edges. Then we can rebuild the ccTree by considering the tree edges in the order of addition, which takes $O(n)$ steps. The advantage of this approach is that the mandatory edges will never appear as an LCA to compute the replacement cost of a non-tree edge, except for the special case when the edge under consideration forms a cycle with mandatory edges only. In fact, we can avoid such special cases by removing all non-tree edges between the nodes in each component formed by the mandatory edges. This first method works well when several edges have become mandatory during one propagation event. When only a few edges become mandatory, our second method will be more efficient.

The second method rebuilds a new ccTree from the existing one. Consider a mandatory (non-tree) edge (i,j). When this edge enters the minimum spanning tree, it will replace the LCA of i and j in the ccTree, i.e., the LCA disappears. As a result, we need to re-build the ccTree up to the point of the previous LCA. That is, we need to revisit the order of the nodes along the paths from i and j to the LCA. Without loss of generality, we assume that i has been added to the ccTree before j. We start by merging i and j (this is the mandatory edge). Then, we proceed by going up the i-LCA path and j-LCA path, starting from



a. MST b. The ccTree c. Updated ccTree

Fig. 1. The minimum spanning tree (MST, in bold) for a small example (a.), its ccTree (b.), and the updated ccTree after the addition of the mandatory edge $(4, 5)$ (c.)

i and j, respectively. Let c_i and c_j be the current node on the i-LCA path and j-LCA path, respectively. As long as the weight of the parent of c_i is at least the weight of the parent of c_j, we let c_i be its parent and continue. If the weight of the parent of c_j is less than the weight of the parent of c_i, we insert c_j between c_i and its parent. In other words, c_j has as 'left' child c_i, and as 'right' child its subtree in the path from j to the LCA (which is always a single node). We then update c_j to be its original parent in the j-LCA path, and repeat the process until the two paths are fully combined (i.e., we reach the position of the previous LCA). Figure 1 provides an example of our second method. To the example presented in Figure 1.a, we introduce the mandatory edge $(4, 5)$. From the ccTree in Figure 1.b, we determine that the LCA for nodes 4 and 5 is the internal node marked with edge $(3, 5)$ with weight 5, which will disappear from the ccTree. Execution of our second method yields the repaired ccTree, depicted in Figure 1.c. The main benefit of this second method is that it needs to update the minimum spanning tree (and the ccTree) only *locally*. In the worst case, its time complexity may be $O(n)$, but the expected time complexity is much lower.

Bibliography

[1] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[2] 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.

[3] G. Dooms and I. Katriel. The "not-too-heavy spanning tree" constraint. In *CP-AI-OR'07*, volume 4510 of *LNCS*, pages 59–70. Springer, 2007.

[4] J.-C. Régin. Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In *Proceedings of CPAIOR*, volume 5015 of *LNCS*, page 233. Springer, 2008.

[5] R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[6] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.