Steiner Tree Problems with Side Constraints Using Constraint Programming

Diego de Uña[1] and **Graeme Gange**[1] and **Peter Schachte**[1] and **Peter J. Stuckey**[1,2]
[1] Department of Computing and Information Systems – The University of Melbourne
[2] National ICT Australia, Victoria Laboratory
{d.deunagomez@student.,gkgange@,schachte@,pstuckey@}unimelb.edu.au

Abstract

The Steiner Tree Problem is a well know NP-complete problem that is well studied and for which fast algorithms are already available. Nonetheless, in the real world the Steiner Tree Problem is almost always accompanied by side constraints which means these approaches cannot be applied. For many problems with side constraints, only approximation algorithms are known. We introduce here a propagator for the tree constraint with explanations, as well as lower bounding techniques and a novel constraint programming approach for the Steiner Tree Problem and two of its variants. We find our propagators with explanations are highly advantageous when it comes to solving variants of this problem.

1 Introduction

The Steiner Tree Problem (STP) is a combinatorial problem on graphs. Given a non-empty graph $G = (V, E)$ and a subset of its nodes $T \subseteq V$ called *terminals*, a Steiner Tree $ST = (V_{ST}, E_{ST})$ is a tree such that $T \subseteq V_{ST} \subseteq V$ and $E_{ST} \subseteq E$. That is to say, ST spans all the nodes in T. We call the non-terminal nodes *Steiner nodes*.

Following the definition of (Dreyfus and Wagner 1971), the STP is an NP-complete problem (proved by (Karp 1972)) stated as follows: given a graph G and a weight function ws, find the Steiner Tree of minimal weight where the total weight is the sum of the weights of the edges in E_{ST} given by ws.

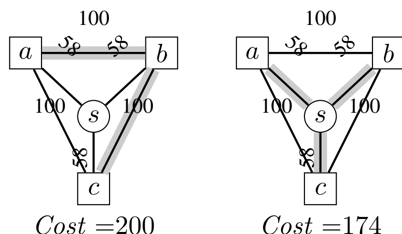

Figure 1: Example of two Steiner trees (highlighted) of the same graph where s is the only Steiner node and a, b and c are terminals.

The STP has been well studied because of its applications in computer networks, VLSI design, transportation and other network problems (Winter 1987), (Hwang, Richards, and Winter 1992). Nevertheless, in many of these applications, the STP is not *pure*. In most real world applications there are side constraints that affect the topology of the solution. Sometimes we need terminal nodes to be leafs of the tree (e.g.: in VLSI or phylogenetics). In other cases, we only want to use nodes that can be connected in multiple ways to increase the reliability of the network (Agrawal, Klein, and Ravi 1995). In transportation we find the hierarchical STP where terminals are divided in size (by city population).

In this paper we present a novel Constraint Programming (CP) approach using explanations to solve the STP that allows any kind of side constraint. We implemented this in the CHUFFED CP solver (Chu 2011). Section 2 introduces some notions from CP along with the model that we use. Section 3 presents some basic data structures. Section 4 presents the implementation of a tree propagator with explanation. Section 5 explores two lower bounds to prune the search space. Section 6 shows the experimental results solving the pure STP and two of its variants (comparing different versions of our propagator against the CHOCO3 solver).

1.1 Previous work

To our knowledge, the state-of-the-art in pure Steiner Tree problems was reached by (Polzin and Daneshmand 2001), and little room for improvement was left by their work. Nonetheless, their ideas focused on the *pure* STP. Indeed, several of the techniques they used to solve the problem (called "reductions" in their paper) are only valid for the pure version. Reductions remove edges or nodes based on the fact that they cannot be part of the *minimum* Steiner tree. We are searching for a Steiner tree that is minimum given a set of side constraints, thus the reductions they used are not valid in our setting.

It is worth noting that the Steiner Tree Problem resembles the Minimum Spanning Tree problem (MST), but it is substantially different in terms of computational complexity. The NP-completeness of the STP comes from not knowing which nodes we need to span. Existing MST or Weighted-Spanning Tree constraints (Dooms and Katriel 2006), (Régin 2008) do not apply in our case as they either look for a *minimum* spanning tree (not allowing side constraint) or try to include all nodes in the tree.

There has also been work in "tree" constraints that are actually more focused on finding forests in (un)directed graphs

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

(Beldiceanu, Flener, and Lorca 2005), (Beldiceanu, Katriel, and Lorca 2006), (Fages and Lorca 2011). Although their constraints could be used for the STP, they are not tailored for it as there is no easy way to encode that all terminals must be in the same tree in the forest, and that we do not care about the cost of the other trees in the forest (nor how many there are).

All the other work we are aware of for solving STP with variations relies on approximation algorithms for each variation (Mehlhorn 1988), (Beasley 1989), (Robins and Zelikovsky 2000), (Garg, Konjevod, and Ravi 2000), (Kim et al. 2002), (Chalermsook and Fakcharoenphol 2005). Our intention in this work is to use CP as a framework to model any variation of the STP to find exact solutions.

2 A CP approach to the STP

Most of the previous work done with exact algorithms for the STP used Mixed Integer Programming (Aneja 1980), (Current, ReVelle, and Cohon 1986). We approach this problem using CP. This allows us to have a more flexible and reusable model. Using global constraints we specialize the solver to tackle the problem in a much more efficient way than using composition of elementary constraints.

2.1 CP, propagation and explanations

In this section we define what propagators and explanations are for the clarity of the paper. A more thorough definition can be found for instance in (Francis and Stuckey 2014).

Constraint Satisfaction Problems (CSP) consist of constraints over a set of variables each with a domain set of possible values. A valid solution to a CSP is a valuation of these variables such that all constraints are satisfied.

A complete search would assign all the possible values to each variable in turn and cover the whole search space to find the optimal solution, backtracking only when no more branching is possible. Since this induces a prohibitive cost, we use constraint propagation during the search.

A propagation solver interleaves propagation and search. The former is a process by which *propagators* remove values from the domains of the variables when they cannot be part of a solution given the previously decided variables. This reduces the search space and, if the domain of a variable becomes empty, detects failure. The latter splits the domain of a variable to generate sub-problems (branches) and tries to solve them. This process stops once all variables have been assigned a value. If a conflict is detected, the solver fails and backtracks to the decision causing it.

Learning is done with clauses created by propagators that capture the reasons for the propagations they do (which means they are universally true). When a propagator infers changes in domains of variables, it gives a set of clauses (or *explanations*) to the solver that "explain" the propagation. The solver will reuse them to make the same inferences again without having to pay the cost of propagation. This is called Lazy Clause Generation (Ohrimenko, Stuckey, and Codish 2009) as the clauses are generated during the solving step.

2.2 Model

To model the STP we use a variable c_e for each edge e indicating whether e is chosen to be in the solution tree and, similarly, a variable c_n for each node n. Other side constraints can specify which nodes are terminals by simply setting the value of the corresponding c_n variable.

Let $ends$ be a map from edges to their end-nodes and adj a map from nodes to their incident edges. Additionally, ws gives the weight of each edge. The variable w is the weight of the tree. The model used to solve the problem is simply:

$$minimize(w) \text{ such that:}$$

$$steiner_tree(\{c_n | n \in V\}, \{c_e | e \in E\}, adj, ends, w, ws)$$

where $steiner_tree$ is the global constraint we will define later that constrains the solution to be a Steiner tree of weight w.

Moreover we can add the extra constraint stating that the solution has one more node than edges (which is true for any tree): $\sum_{e \in E} c_e = \sum_{n \in V} c_n - 1$. The solver can find a conflict with this constraint before other propagation is able to detect a failure. We found this empirically advantageous.

Assigning values to the variables implicitly builds a graph $G_s = (V_s, E_s) = (\{c_n | c_n = true\}, \{c_e | c_e = true\})$. We say that an edge e is an *in-edge* if at the current stage of the search c_e is $true$ (and we draw it as '—' in the following figures), *out-edge* ('.....') if c_e is $false$ and *unknown-edge* ('- - -') for an unassigned edge. Similarly we define the terms *in-node* ('●'), *out-node* (never drawn, for clarity) and *unknown-node* ('○'). Clearly, all terminal nodes are in-nodes. Eventually, G_s will become the solution.

3 Preliminaries

Here we present a few data-structures and algorithms that we will be using later.

3.1 Re-rooting Union-Find data structure

In order to implement the tree propagator we will use a modified version of the classic union-find data structure that will help us to retrieve paths between nodes efficiently.

The typical union-find data structure (UF) builds a directed forest of nodes when the method $unite(u, v)$ is applied. Then we can retrieve the root of each tree by using the method $find(u)$. In our implementation, we will have a method $path(u, v)$ that will return the nodes in the path from u to v (or an empty path if they are not connected).

To do so, we modify the $unite$ procedure: we first make u and v become the root of their respective trees (by inverting some of the parenthood relations in the trees), then we make u the parent of v. To retrieve the path we modify the $find$ method to return the nodes it goes through (we can later map pairs of nodes into edges). Calling this method on two nodes a and b in the same connected component allows us to find the path between those nodes. The worst case complexity of this query is linear in the number of nodes in the graph, although in practice it is much closer to the length of the path between the two nodes queried.

3.2 Graph contraction

Given a graph G and the decisions made so far in the search, we define a contraction function $cont : (V, E) \longmapsto (V', E')$ that contracts all the in-nodes connected by in-edges into one new "in-node" and removes all the out-edges. In other words, the connected components (CCs) of G_s are contracted and the out-edges removed. Clearly E' contains only unknown-edges. By analogy, we call "in-nodes" the nodes of G' built from in-nodes of G.

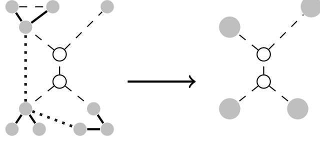

Figure 2: Example of the application of function *cont*

We will use this contraction later when computing the lower bounds.

4 Tree propagator with explanations

We first start by implementing a tree propagator. This propagator will wake whenever a variable c_n or c_e is fixed, meaning that a node (resp. edge) becomes an in-node or out-node (resp. in/out-edge). The purpose of the propagator is to ensure that the final G_s can be a tree (i.e. a connected acyclic sub-graph of G).

Table 1 presents a list of the rules applied (in order) in each case. In the following subsections we detail them.

Event	Rules
Node addition n	1. $reachable(n)$ 2. $articulations(n)$ 3. $cycle_prevent(n)$ 4. $steiner_node(n)$
Node removal n	1. $coherent(n)$
Edge addition $e = (u, v)$	1. $coherent(e)$ 2. $cycle_detect(e)$ 3. $\forall n \in CC_{G_s}(u), cycle_prevent(n)$ 4. $UF.unite(u, v)$
Edge removal $e = (u, v)$	1. $reachable(u)$ 2. $articulations(u)$ 3. $\forall n \in \{u, v\}, steiner_node(n)$

Table 1: List of algorithms in applied in our tree propagator. $CC_{G_s}(u)$ is the connected component (CC) of G_s containing u.

4.1 The *reachable* algorithm

Given an in-node n, the $reachable(n)$ algorithm ensures that n can be connected through in-edges or unknown-edges to other in-nodes. If n is not reachable from some other in-node, then n cannot be part of G_s as it would not be a tree. In this situation the solver must fail and backtrack.

Figure 3 gives an example of two unreachable in-nodes.

First, to detect the failure, we run a depth first search (DFS) starting at n. We will mark all the nodes visited as $blue$. This DFS will traverse in-edges and unknown-edges only. Then we look for any non-blue in-node o. If such a node o exists, we fail.

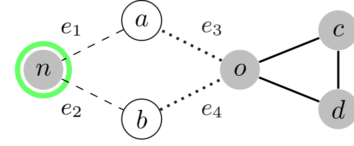

Figure 3: Example of unreachable nodes: n was added to G_s. Edges e_3 and e_4 being out-edges, we cannot reach o.

Explaining this failure requires finding a minimal set of out-edges such that if any of them was *in* or *unknown*, there would still be a solution in the current search space.

To find those edges, we run another DFS from the found target node o marking all the nodes reached as $pink$. During this DFS we allow traversal through all edges except the ones having one blue end-node.

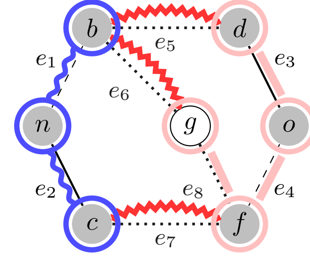

Figure 4: Example of the blue (on the left, ‘ ’,‘ ’) and pink (on the right, ‘ ’,‘ ’) DFS to detect failure. The zigzag edges (‘ ’) explain failure. Note that e_8 is not needed in the explanation for it to hold.

Let OE be the set of out-edges encountered during the pink DFS that have one blue extremity (we do not cross them). If at least one of them was allowed to be used, the pink DFS would have reached the blue nodes thus showing that G_s could still be connected. Therefore, this set of edges explains the un-reachability of o from n. The final explanation is: $(c_n \wedge c_o \wedge \bigwedge_{e \in OE} \neg c_e) \Rightarrow fail$.

4.2 The *articulations* algorithm

We assume that the $reachable$ algorithm succeeded as there would be no solution otherwise. In such a graph there is at least one bi-connected component (bi-CC). A bi-CC is a sub-graph that is bi-connected (i.e. every pair of nodes is connected by at least two paths). The nodes between two bi-CCs are known as *articulations*. Also, if a bi-CC contains only one edge, then that edge is a *bridge*.

Since G_s needs to be a tree, it must be connected. We can then propagate that any articulation (resp. bridge) that is in the path between two in-nodes u and v is an in-node (resp. in-edge), otherwise u and v would be disconnected.

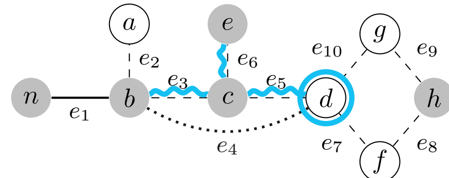

Figure 5: Example of bridges (‘ ’) and articulations (‘ ’)

To find the articulations, we modify Tarjan's algorithm for finding bi-CCs (Tarjan 1972) starting at an in-node. Recall Tarjan's algorithm performs a DFS in the graph while marking nodes with their depth and their "lowpoint". The

lowpoint of a node u is the node v with lower depth that has been reached from the recursive calls of the DFS starting at u. By lemma 5 in Tarjan's paper: $u = parent(v) \wedge depth(lowpt(v)) \geq depth(u) \Rightarrow u\ is\ an\ articulation$.

In our version, the DFS will start at an in-node and will not be allowed to cross out-edges. Also, we are only interested in articulations/bridges that are in the path between two in-nodes. To identify only these ones we use a stack S that records all the in-nodes reached whilst performing the DFS. If after a recursive call in the DFS we detect an articulation a and the top of S is different than when we reached node a then a is a required articulation for the two top-most nodes of S. Additionally, if the last discovered bi-CC had only one edge, then that edge is a required bridge.

To explain this bridges and articulations we need the two in-nodes c_1 and c_2 that required them: we extract those nodes from S while performing the DFS (let c_1 be the top of S). We also need all the out-edges that could have connected c_1 and c_2. Indeed, if those edges were available, then we would not have found articulations or bridges. We do this in two steps after Tarjan's DFS. First, we run a DFS from c_1 that does not go through out-edges (nor the articulation or bridge). This gives a set R of reachable nodes. Then we run a second DFS from c_1 this time allowing to cross out-edges (but not the articulation or bridge). We add any out-edge adjacent to a node not in R but visited during Tarjan's DFS to a set OE.

The explanations is $(c_{c_1} \wedge c_{c_2} \wedge \bigwedge_{e \in OE} \neg c_e) \Rightarrow a$, where a is either the articulation or the bridge we found.

In our example in Figure 5 (starting at n) we would first find that $c_c \wedge c_h \Rightarrow c_d$, then $c_c \wedge c_h \wedge \neg c_{e_4} \Rightarrow c_{e_5}$ and $c_e \wedge c_c \Rightarrow c_{e_6}$ and finally $c_b \wedge c_c \wedge \neg c_{e_4} \Rightarrow c_{e_3}$. Note, e_2 is not a bridge since it is not in the path between two in-nodes.

4.3 The *cycle_detect* and *cycle_prevent* algorithms

Given a new in-edge $e = (u, v)$, the $cycle_detect(e)$ algorithm ensures that e does not form a cycle in G_s. If it does, there must be a pre-existing path p from u to v that we can retrieve using the UF. If p exists, we stop the search and we give the reason $(c_e \wedge \bigwedge_{e' \in p} c_{e'}) \Rightarrow fail$.

Furthermore, given a node n, the algorithm $cycle_prevent(n)$ removes any edge adjacent to n that would form a cycle if it was added to G_s. We can safely propagate the decision that they must be out-edges as G_s would contain a cycle otherwise.

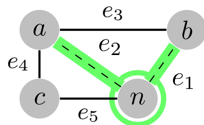

Figure 6: Example of two potential cycles. After adding n we can safely remove e_1 and e_2 because adding them would create a cycle.

To perform this operation, we simply remove any edge $e = (n, o) \in adj[n]$ such that n and o are connected by in-edges. The minimal reason required for this removal is the set of in-edges forming a path p from n to o in G_s. That is: $\bigwedge_{e_{in} \in p} c_{e_{in}} \Rightarrow \neg c_e$. Again, we use our UF to retrieve p.

Note how in the event of the addition of a new node we only need to perform this operation at the new node (the incrementality of the propagator ensures that new possible cycles could only appear from the new node). For a new edge, though, we have to look at all the in-nodes in the CC of G_s containing the edge e (as the new potential cycle can appear anywhere in the CC) so we perform this operation while traversing the CC with a DFS.

4.4 The *steiner_node* checks

Let the degree of a Steiner node n (noted $deg_s(n)$) be the number of edges incident to n that are not out-edges. Let $R_n = \{e | e \in adj[n, e] \wedge \neg c_e\}$.

A Steiner node is only useful if it is part of the path between two in-nodes, otherwise it will just be increasing the cost and not bringing any advantage to the tree. We can use this premise to do STP-specific propagations. Given a node n, $steiner_node(n)$ will verify two conditions:

- if $deg_s(n) = 1$, we will fail with the following reason: $(c_n \wedge \bigwedge_{R_n} \neg c_e) \Rightarrow fail$.
- if $deg_s(n) = 2$ (edges e_1 and e_2) we can safely propagate that the solution will only contain n if both edges are also in G_s. We force them in G_s giving the explanation: $\forall b \in \{c_{e_1}, c_{e_2}\}, (c_n \wedge \bigwedge_{R_n} \neg c_e) \Rightarrow b$.

4.5 The *coherent* checks

Whenever a node is removed or an edge is added, we must make sure that G_s is still sound. The $coherent$ checks enforce that for a node $n, \forall e \in adj[n], \neg c_n \Rightarrow \neg c_e$ and for an edge $e = (u, v)$, $(c_e \Rightarrow c_u) \wedge (c_e \Rightarrow c_v)$.

5 Lower bounding for the STP

Given a solution of cost K, a lower bound allows us to prune the search space by proving that no better solution exists in a branch. This is known as branch-and-bound.

5.1 Shortest-paths based lower bound (SPLB)

Consider the graph $G' = (V', E')$ obtained by applying the function $cont$ defined in section 3.2 to G. Let S be the set of in-nodes in G'. We claim that, in G', the following is a lower bound for the STP.

$$If\ |S|\ is\ even: \quad LB(G') = \frac{1}{2} \sum_{u \in S} spc_{G'}(u)$$

$$If\ |S|\ is\ not\ even:$$

$$LB(G') = \Big(\frac{1}{2} \sum_{u \in S} spc_{G'}(u)\Big) - \min_{u \in S} \big(spc_{G'}(u)\big)$$

where $spc_{G'}(u)$ is the weight of the shortest path between u and its closest in-node in G' (proof in appendix A).

We extend this lower bound to a lower bound of the STP in the current graph by adding the weight of the in-edges that were contracted. We call this lower bound SPLB.

$$\text{SPLB}(G_s) = \Big(\sum_{e \in E_s} ws[e] \Big) + LB(cont(G))$$

5.2 Computing SPLB

Because computing this lower bound can be expensive if we do it every time, we implemented it in an incremental way.

In order to avoid having a contracted version of the graph, we will consider that all in-edges have weight zero and we will only compute the shortest paths between a *representative* in-node from each CC of G_s (we use the roots of the UF as representatives).

Let sp_C be a map from representatives to the cost of its shortest path to another representative and sp_E a map from representatives to the edges used in that shortest path.

We compute SPLB by adding the following steps after the previously described propagations.

Node addition A new node may change all the shortest paths between CCs of G_s, so we need to recompute them all. We use Dijkstra's algorithm starting at a representative while recording the paths and update sp_C and sp_E.

Edge removal All paths using the removed edge (recorded in sp_E) must be recomputed with Dijkstra's algorithm.

Edge addition A new edge merges two CCs of G_s, so we use the shortest path of the two of them as the new shortest path of the resulting CC (other than the path between them). We also add the weight to a variable m_w. Eventually, the lower bound will be the sum of m_w and the sum of all the costs recorded in sp_C.

5.3 Explaining SPLB

The lower bound will prune the search space by making the solver fail and backtrack. As with other propagations, we need to explain the failure.

All edges that are in G_s must be part of the explanation since they bring weight to the lower bound. We must also include the out-edges that have been in a shortest path at some stage. Indeed, if we remove an edge from a shortest path, we find a second shortest path of higher weight. Therefore, deleting those edges causes the lower bound to increase. We record them in a set sp_R whenever we remove them. The resulting explanation is:

$$\left([\![w < K]\!] \wedge \bigwedge_{e \in E_s} c_e \wedge \bigwedge_{e \in spR} \neg c_e \right) \Rightarrow \mathit{fail}$$

where $[\![w < K]\!]$ is the clause stating that we want a tree of cost less than K (which is the cost of the best solution found so far). This explanation states that no better solution can be found given the edges in G_s and the out-edges that could have lowered the weight of the solution.

5.4 Linear program lower bound (LPLB)

Previous work in the pure STP used a linear program (LP) lower bounding technique to compute the solution to the problem (Polzin and Daneshmand 2001). This lower bound remains valid for any variant of the STP.

Cut formulation of the STP Any cut of G that separates the nodes in two partitions $V = \{W, \bar{W}\}$ such that both contain at least one terminal must have at least one edge crossing from W to $\bar{W}$ that is part of the solution.

From this observation derives the cut formulation of the STP introduced by (Aneja 1980):

$$\mathit{minimize}\Big(\sum_{e \in E} ws[e] * c_e \Big) \text{ such that:}$$
$$\forall\, W, \sum_{e \in \delta(W)} c_e \geq 1$$

where $\delta(W)$ is the set of edges with exactly one end-node in W.

The linear relaxation of this problem, which we call LPLB, makes use of real variables $x_e \in 0..1$ for each edge e instead of the Boolean variables c_e, and is solvable in linear time. Solving this yields a lower bound.

5.5 Computing LPLB

Following the work of (Polzin and Daneshmand 2001), we implemented this lower bound using row generation and we solve it using CPLEX 12.4. In order to generate the rows, we use the Edmonds-Karp's maximum flow algorithm (Edmonds and Karp 1972). Each run of this algorithm gives us a minimum cut that we add to the LP. We also implemented the DUAL-ASCEND algorithm described by (Wong 1984) for the initial set of rows in the LP. In order to make this incremental, we run the flow computation and re-optimise LPLB after all the rules in Section 4 are applied.

5.6 Explaining LPLB

Part of the explanation are the in-edges, as before. Also any in/out-edge in $R = \{e | rc(c_e) \neq 0\}$ that has non-zero reduced cost[1] (rc) is part of the explanation. This is because edges with zero reduced cost would not change the value of the lower bound if they changed theirs, so they do not contribute to the lower bound.

$$\left([\![w < K]\!] \wedge \bigwedge_{e \in E_s} c_e \wedge \bigwedge_{e \in R, \neg c_e} \neg c_e \wedge \bigwedge_{e \in R, c_e} c_e \wedge \right) \Rightarrow \mathit{fail}$$

6 Experimental results

We modelled the pure STP and two variations in MINIZINC and solved them with the CHUFFED solver. We used the latest CHOCO3 (Prud'homme, Fages, and Lorca 2014) solver as a comparison since it includes the most up to date implementation of the CP(Graph) framework (Dooms, Deville, and Dupont 2005).

We ran all our tests with the SPLB and LPLB lower bounds as well as without lower bound (NOLB), no tree propagator at all (NOPROP) and the LPLB with no learning *(n.l.)* to compare the benefits of the lower bounds and learning. We also tested a version called SP+LPLB where SPLB runs first and if it does not prune, we run LPLB.

The benchmarks used in this study are from the SteinLib (Koch, Martin, and Voß 2000). Note that a number of benchmarks have been solved to optimality in the *pure* STP but not with side constraints. We used the test-sets ES10FST (15 instances of 12 to 24 nodes), ES20FST (15 instances of 27 to 57 nodes) and B (11 instances of 50 to 75 nodes).

[1] In LP, the reduced cost indicates how much the objective function coefficient of a variable must be reduced before the variable will be positive in the optimal solution.

We used the same search strategy in all the implementations. The order of the variables is: edges sorted by decreasing weight, then nodes in arbitrary order. The value strategy is: try assigning the values $\{false, true\}$ in that order to each variable. This is the strategy that gave the best results.

All tests were run on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz, 15.6GB of RAM machine. We used 5 hours as the time-out for all the tests and the geometric average (including timed-out instances) to summarize the results. Time is indicated in seconds and the number of unsolved instances (if any) appears in parentheses in the tables.

Table 2 shows the results for the pure STP. Subsections 6.1 and 6.2 present the models for the two variants followed by the results tables.

Pure STP		Conflicts	Nodes	Propagations	Time
ES10FST	SPLB	12	15	123	**0.01**
	LPLB	11	14	101	**0.01**
	LPLB (n.l.)	12	22	112	**0.01**
	SP+LPLB	**10**	**13**	**99**	**0.01**
	NOLB	13	138	134	**0.01**
	NOPROP	10686	1079	797998	1.75 (4)
	CHOCO3	130	138	198	**0.01**
ES20FST	SPLB	729	816	6383	1.35 (2)
	LPLB	492	534	3723	0.49
	LPLB (n.l)	632	1080	4636	**0.47**
	SP+LPLB	**477**	**520**	**3626**	0.74
	NOLB	746	831	7016	0.67 (2)
	NOPROP	604055	604200	43168067	1065 (11)
	CHOCO3	45434	46175	67780	1.10 (3)
B	SPLB	55979	83120	531033	22.25 (2)
	LPLB	**8097**	**9157**	**53731**	**5.17**
	LPLB (n.l)	18000	37096	110711	6.99 (1)
	SP+LPLB	8102	10372	68814	7.11
	NOLB	114538	192343	1460071	17.80 (2)
	NOPROP	177364998	179890423	18203144792	18000 (11)
	CHOCO3	80406257	80406288	152270771	3339.66 (5)

Table 2: Results for the pure Steiner Tree problem.

6.1 The Grade of Service STP (GoSST)

In computer networks, computers have bit-rate requests that need to be matched by the network. This has important real world applications (e.g. video distribution described by (Maxemchuk 1997)). Moreover, networks are not unlimited: each edge has a maximum capacity cap. We call the capacity of a path p the minimum of the capacities of the edges in p. Let d be the demand of a terminal node n.

The goal of the GoSST (Du and Hu 2008) is to find a minimum Steiner tree network such that for each pair of terminals, there is at least one path of capacity higher than the minimum of the demand of the two terminals.

To model this problem, we use Boolean variables $P_{i,j}[k]$ indicating whether an edge k is in the path from i to j. Then we add the following constraints to the model in section 2:

$$\forall \{u,v\} \in T^2, \Big(\forall n \in V,\ \big(\sum_{e \in adj[n]} P_{u,v}[e]\big) \in \{0,2\}\Big) \quad (1)$$

$$\forall \{u,v\} \in T^2, \qquad \Big(\sum_{e \in adj[u]} P_{u,v}[e] \quad = \quad 1\Big) \quad (2)$$

$$\forall e \in E,\ \forall \{u,v\} \in T^2,$$
$$\Big(\neg c_e \vee \big(cap[e] < min(d[u], d[v])\big) \Rightarrow \neg P_{u,v}[e]\Big) \quad (3)$$

These constraints are: (1) a node n must provide either no edge or two edges to each path, (2) all terminals must contribute with one edge to each path of which they are an extremity, (3) for any pair of terminals, out-edges or edges with lower capacity than their demand cannot be in the path connecting them.

GoSST		Conflicts	Nodes	Propagations	Time
ES10FST	SPLB	5	1124	7423	**0.01**
	LPLB	**4**	1173	6635	0.02
	LPLB (n.l.)	**4**	1178	6599	**0.01**
	SP+LPLB	**4**	**1121**	**6428**	**0.01**
	NOLB	5	1178	7839	0.05
	NOPROP	5	1178	7839	0.05
	CHOCO3	65	69	67358	0.13
ES20FST	SPLB	82	18587	465484	2.32
	LPLB	49	17659	247446	**1.72**
	LPLB (n.l)	55	18122	263924	1.75
	SP+LPLB	**44**	**17644**	**241253**	1.83
	NOLB	105	18972	573647	2.56
	NOPROP	159	20637	1455869	9.36
	CHOCO3	3373	3433	22745514	55.07 (4)
B	SPLB	16713	144110	33533492	47.63 (2)
	LPLB	11330	135492	18648420	**37.61** (2)
	LPLB (n.l)	20376	207503	27413586	43.19 (2)
	SP+LPLB	**9550**	**122314**	**16627051**	42.67 (2)
	NOLB	15343	191803	27413586	50.96 (2)
	NOPROP	96642	466502	331019078	330.34(3)
	CHOCO3	184248	184307	4937258941	14790.80 (10)

Table 3: Results for the Grade of Service Steiner Tree Problem.

The state of the art in this problem is an approximation algorithm (Karpinski et al. 2003).

6.2 The Terminal Steiner Tree Problem (TSTP)

The terminal STP (Lin and Xue 2002) is a small variation of the original problem used in VLSI and phylogenetic studies. In such environments, we might need a terminal to be a leaf. This only affects the degree of the terminals and can be achieved by adding the following constraint to the original model in section 2:

$$\forall t \in T, \sum_{e \in adj[t]} c_e = 1$$

TSTP		Conflicts	Nodes	Propagations	Time
B	SPLB	33	60	730	1.37
	LPLB	29	46	565	**0.30**
	LPLB (n.l)	405	573	5570	2.29 (1)
	SP+LPLB	**26**	**44**	**524**	0.40
	NOLB	55	101	1354	1.62
	NOPROP	20644413	21174280	2637608115	5040.19 (8)
	CHOCO3	398026	384652	628843	26.27 (3)

Table 4: Results for the Terminal Steiner Tree Problem. We only used the set B because most benchmarks in the other sets were unsatisfiable.

Again, all the work in this problem is in approximation algorithms (Drake and Hougardy 2004), (Chen 2011).

6.3 Concluding remarks

We can clearly see that both our propagator and the explanations are greatly beneficial to solve the problems faster and with less nodes[2]. Also, LPLB is overall the best in time, although SP+LPLB is usually better in all the other measures.

[2]In CP, *nodes* is the number of nodes in the search tree. Therefore, the less nodes, the better the search is.

This is because having both lower bounding techniques in the same propagator has a higher runtime cost when the weaker lower bound (SPLB) is not good enough to stop the search and we need to run LPLB.

The contributions of this work are a new tree propagator with explanations, a new lower bound with explanations for the STP and explanations for the already existing LP lower bound. All this, put together, forms the Steiner Tree Propagator in graphs that we present here.

References

Agrawal, A.; Klein, P.; and Ravi, R. 1995. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing* 24(3):440–456.

Aneja, Y. P. 1980. An integer linear programming approach to the Steiner problem in graphs. *Networks* 10(2):167–178.

Beasley, J. E. 1989. An SST-based algorithm for the Steiner problem in graphs. *Networks* 19(1):1–16.

Beldiceanu, N.; Flener, P.; and Lorca, X. 2005. The tree constraint. In *Integration of AI and OR Techniques in constraint programming for combinatorial optimization problems*. Springer. 64–78.

Beldiceanu, N.; Katriel, I.; and Lorca, X. 2006. Undirected forest constraints. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 29–43.

Chalermsook, P., and Fakcharoenphol, J. 2005. Simple distributed algorithms for approximating minimum Steiner trees. In *Computing and Combinatorics*. Springer. 380–389.

Chen, Y. H. 2011. An improved approximation algorithm for the terminal Steiner tree problem. In *Computational Science and Its Applications-ICCSA 2011*. Springer. 141–151.

Chu, G. G. 2011. *Improving combinatorial optimization*. Ph.D. Dissertation, The University of Melbourne.

Current, J. R.; ReVelle, C. S.; and Cohon, J. L. 1986. The hierarchical network design problem. *European Journal of Operational Research* 27(1):57–66.

Dooms, G., and Katriel, I. 2006. The minimum spanning tree constraint. In *Principles and Practice of Constraint Programming-CP 2006*. Springer. 152–166.

Dooms, G.; Deville, Y.; and Dupont, P. 2005. CP (Graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming-CP 2005*. Springer. 211–225.

Drake, D. E., and Hougardy, S. 2004. On approximation algorithms for the terminal Steiner tree problem. *Information Processing Letters* 89(1):15–18.

Dreyfus, S. E., and Wagner, R. A. 1971. The Steiner problem in graphs. *Networks* 1(3):195–207.

Du, D., and Hu, X. 2008. Grade of service Steiner tree problem. In *Steiner Tree Problems In Computer Communication Networks*. River Edge, NJ, USA: World Scientific Publishing Co., Inc. 111–139.

Edmonds, J., and Karp, R. M. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM* 19(2):248–264.

Fages, J.-G., and Lorca, X. 2011. Revisiting the tree constraint. In *Principles and Practice of Constraint Programming–CP 2011*. Springer. 271–285.

Francis, K. G., and Stuckey, P. J. 2014. Explaining circuit propagation. *Constraints* 19(1):1–29.

Garg, N.; Konjevod, G.; and Ravi, R. 2000. A polylogarithmic approximation algorithm for the group Steiner tree problem. *Journal of Algorithms* 37(1):66 – 84.

Hwang, F.; Richards, D.; and Winter, P. 1992. *The Steiner Tree Problem*. Annals of Discrete Mathematics. Elsevier Science.

Karp, R. 1972. Reducibility among combinatorial problems. In Miller, R.; Thatcher, J.; and Bohlinger, J., eds., *Complexity of Computer Computations*, The IBM Research Symposia Series. Springer US. 85–103.

Karpinski, M.; Măndoiu, I. I.; Olshevsky, A.; and Zelikovsky, A. 2003. *Improved approximation algorithms for the Quality of Service steiner tree problem*. Springer.

Kim, J.; Cardei, M.; Cardei, I.; and Jia, X. 2002. A polynomial time approximation scheme for the grade of service Steiner minimum tree problem. *Journal of Global Optimization* 24(4):437–448.

Koch, T.; Martin, A.; and Voß, S. 2000. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin.

Lin, G., and Xue, G. 2002. On the terminal Steiner tree problem. *Information Processing Letters* 84(2):103–107.

Maxemchuk, N. F. 1997. Video distribution on multicast networks. *Selected Areas in Communications, IEEE Journal on* 15(3):357–372.

Mehlhorn, K. 1988. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters* 27(3):125 – 128.

Ohrimenko, O.; Stuckey, P.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14(3):357–391.

Polzin, T., and Daneshmand, S. V. 2001. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics* 112(13):263 – 300. Combinatorial Optimization Symposium, Selected Papers.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2014. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Régin, J.-C. 2008. 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*. Springer. 233–247.

Robins, G., and Zelikovsky, A. 2000. Improved Steiner tree approximation in graphs. In *SODA*, 770–779. Citeseer.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1(2):146–160.

Winter, P. 1987. Steiner problem in networks: A survey. *Netw.* 17(2):129–167.

Wong, R. 1984. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming* 28(3):271–287.