Steiner Tree Problems with Side Constraints Using Constraint Programming

Diego de Uña¹ and Graeme Gange¹ and Peter Schachte¹ and Peter J. Stuckey¹,²
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 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 \( w \), 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 \( w \).

![Figure 1: Two Steiner trees (highlighted) of the same graph where s is the only Steiner node and a, b and c are terminals.](image)

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), (Région 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 ac-
tually more focused on finding forests in (un)directed graphs (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 (Angea 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:

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

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

where $\text{steiner_tree}$ 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 $G_s = (V_s, E_s) = (\{c_n|c_n = \text{true}\}, \{c_e|c_e = \text{true}\})$. 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 $\text{unite}(u, v)$ is applied. Then we can retrieve the root of each tree by using the method $\text{find}(u)$. In our implementation, we will have a method $\text{path}(u, v)$ 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 $\text{unite}$ 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 $\text{find}$ 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 (CC) 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 $\text{cont} : (V, E) \rightarrow (V', E')$ 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 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 $G$. We will make use of this in Section 5.

![Figure 2: Example of the application of function \text{cont}](image)

4 Tree Propagator with Explanations

We first start by implementing a tree propagator. This propagator will wake whenever a variable $c_o$ 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.

<table>
<thead>
<tr>
<th>Event</th>
<th>Rules</th>
</tr>
</thead>
</table>
| Node addition | 1. reachable($n$)  
2. articulations($n$)  
3. cycle_prevent($n$)  
4. steiner_node($n$) |
| Node removal  | 1. coherent($n$) |
| Edge addition | 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  | 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 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 $o$.](image)

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.](image)

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: $(e_n \land c_o \land \bigwedge_{e \in OE} \neg c_e) \Rightarrow \text{fail}$.

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.

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: $u = \text{parent}(v) \land$
depth(lowpt(v)) ≥ depth(u) ⇒ 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 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 these 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 \((c_1 \land c_2) \land \bigwedge_{e \in OE} \neg c_e) \Rightarrow a\), where \( a \) is either the articulation or the bridge we found.

In our example in Figure 5 (starting at \( n \)) we first find that \( c_e \land c_o \Rightarrow c_n \), then \( c_e \land c_k \land \neg c_4 \Rightarrow c_e \) and \( c_e \land c_c \Rightarrow c_n \) and finally \( c_l \land c_e \land \neg c_4 \Rightarrow c_n \). 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 \((e \land \bigwedge_{e \in E} c_e) \Rightarrow \text{fail}\).

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 potential cycles created by \( e_1 \) and \( e_2 \).

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: \( \bigwedge_{c_n \in p} c_e \Rightarrow \neg c_e \). 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 \( R_n = \{ e \mid e \in adj[n] \land \neg c_e \} \).

A Steiner node is only useful if it is part of the path between two in-nodes, otherwise it increases the cost and does not bring any advantage to the tree. We can use this premise to do STP-specific propagations. Given a node \( n \):

- if \( deg_s(n) = 1 \), we will fail with the following reason: \((c_n \land \bigwedge_{e \in R_n} \neg c_e) \Rightarrow \text{fail}\).
- 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: \( \forall b \in \{ e_1, e_2 \}, (c_n \land \bigwedge_{e \in R_n} \neg c_e) \Rightarrow b \).

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 \( e = (u, v) \), \((c_e \Rightarrow c_u) \land (c_e \Rightarrow c_v)\).

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 \( G' \) obtained by applying the cont function (Sec. 3.2) to \( G \). Let \( S \) be the set of in-nodes in \( G' \). We claim that, in \( G_s \), the following is a lower bound for the STP:

\[
LB(G') = \left\{ \begin{array}{l} 
\frac{1}{2} \sum_{u \in S} \text{spc}_{G'}(u), \text{if } |S| \text{ is even} \\
\frac{1}{2} \sum_{u \in S} \text{spc}_{G'}(u) - \min_{u \in S} \text{spc}_{G'}(u), \text{otherwise} 
\end{array} \right.
\]

where \( \text{spc}_{G'}(u) \) is the weight of the shortest path between \( u \) and its closest in-node in \( G' \) (see proof in extended version of this paper, available on author’s websites).

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.

\[
\text{SPLB}(G_s) = LB(\text{cont}(G)) + \sum_{e \in E_s} \text{ws}[e]
\]
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 $s_{PC}$ be a map from representatives to the cost of its shortest path to another representative and $s_{PE}$ 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 $s_{PC}$ and $s_{PE}$.

Edge Removal All paths using the removed edge (recorded in $s_{PE}$) 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 $m_{uw}$. Eventually, the lower bound will be the sum of $m_{uw}$ and the sum of all the costs recorded in $s_{PC}$.

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 $s_{PR}$ whenever we remove them. The resulting explanation is:

$$
\left( [w < K] \land \bigwedge_{e \in E_s} c_e \land \bigwedge_{e \in s_{PR}} \neg c_e \right) \Rightarrow \text{fail}
$$

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):

$\text{minimize} \sum_{e \in E} w[e] \cdot c_e$ 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 $L_{PLB}$, 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 $L_{PLB}$ 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.

$$
[w < K] \land \bigwedge_{e \in E_s} c_e \land \bigwedge_{e \in R, rc(e) < 0} \neg c_e \land \bigwedge_{e \in R, c_e} c_e \land \Rightarrow \text{fail}
$$

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

\(^1\)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.
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).

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.

### 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 \(v\).

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 Steiner tree network. Therefor, we can model this problem by adding the following constraints to the model in section 2:

\[
\forall e \in E, \forall \{u, v\} \in T^2, \quad -c_v \lor (cap[e] < min(d[u], d[v])) \Rightarrow \neg P_{u,v}[e] \quad (3)
\]

These constraints are: (1) a node \(v\) 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.

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: \(\forall v \in T^2, \sum_{e \in E} d(e) c_e = 1\).

The state of the art is still very limited (Kernighan 2008) and many benchmarks in the other sets were proven unsatisfiable too fast to show any significant result.

### 6.3 Concluding Remarks

We can see clearly that both our propagator and the explanations are greatly beneficial to solve the problems faster and
with fewer nodes (i.e. a smaller search space). Also, LPLB is overall the best in time, although SP+LPLB is usually better in all the other measures. 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.

Acknowledgements

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

References


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


Appendices

A Proofs

Shortest-path based lower bound (SPLB) (Section 5.1 of the paper). Consider the graph $G' = (V', E')$ obtained by applying the function $cont$ defined in section 3.2 of the paper 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:

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

If $|S|$ is not even:

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

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

Lemma 1. Given a graph $G = (V, E)$ with an even size set of terminals $S$ ($|S| = k \geq 2$) and a Steiner tree $T = (V_T, E_T)$ that spans all the terminals, there exists a path $P$ between two terminals $u$ and $v$ such that:

1. $P$ contains only two terminals (namely $u$ and $v$).
2. $P$ contains at most one node of degree more than 2 in $T$.

Proof. We prove this by induction.

Basis: When $G$ contains only two terminals $u$ and $v$, $T$ only contains nodes of degree 2 and two leaves ($u$ and $v$ of degree 1). In this case, the only path between $u$ and $v$ verifies the 2 conditions of the lemma.

Inductive step: Assume the lemma is valid for a tree with $k' \geq 2$ terminals. Let’s prove it for $k = k' + 2$ (i.e. we added two terminals). Let us arbitrarily choose two terminals $u$ and $v$, and let $P$ be some path in $T$ between these two terminals. If $P$ satisfies the 2 conditions of the lemma, then we finish. Otherwise, either $P$ contains three terminal nodes or $P$ contains two nodes $x$ and $y$ of degree greater than 2.

In the first case, $w$ is in between $u$ and $v$ in $T$ (i.e. the path is somewhat like $u---w---v$). The subtree containing both $u$ and $w$ (but not $v$) has $k'' < k$ terminals and the lemma holds for it (by hypothesis). Thus, the lemma holds for $k$ terminals.

In the second case, when nodes $x$ and $y$ of degree greater than 2 in $T$ are in the path between terminals $u$ and $v$, we can apply a similar reasoning. Here, $x$ and $y$ are in between $u$ and $v$ in $P$ (i.e. the path is somewhat like $u---x---y---v$). Then the third edge incident to $x$ that is not in $P$ must lead to some other terminal $w$. Thus, the subtree containing both $u$ and $w$ has $k'' < k$ terminals and the lemma holds for it (by hypothesis). Thus, the lemma holds for $k$ terminals.

Lemma 2. Given a graph $G = (V, E)$ with an even size set of terminals $S$ ($|S| \geq 2$) and a Steiner tree $T = (V_T, E_T)$ that spans all the terminals, there exists a path $P$ (let $E_P$ be the set of edges in $P$) between two terminals $u$ and $v$ such that there is a Steiner tree $T' = (V'_T, E'_T)$ for $G' = (V, E \setminus E_P)$ with terminals $S' = S \setminus \{u, v\}$ and $E'_T \cap E_P = \emptyset$.

Proof. We choose $P$ satisfying Lemma 1. Then if we remove $E_P$ from $G$, we are left with the graph $G'$ and terminals $S'$. Let’s prove that $T'$ exists in $G'$.

Because $P$ has only one node of degree greater than 2, the connectivity of $T$ does not rely on any edge in $E_P$. Therefore, removing $E_P$ from $E$ does not disconnect any terminal in $S'$ from $T$. For this reason, if $T$ was a valid Steiner tree connecting the terminals in $S$, $G'$ contains a tree $T'$ that does not use the edges in $E_P$ that is connected and spans all the terminals in $S'$.

Theorem 1. $LB(G')$ is a valid lower bound for the Steiner Tree problem in $G'$.

Proof. Let $T$ be the Steiner tree of minimum cost $W$ in $G'$ (i.e. $T$ contains all the nodes in $S$). Let’s prove that $LB(G')$ is indeed a lower bound by verifying that $LB(G') \leq W$.

If $|S|$ is even: Because $T$ is connected, there is a path form every node in $S$ to any other node in $S$. We name $p_{a,b}$ the path in $T'$ between two in-nodes $a$ and $b$ (i.e. $(a, b) \in S^2$). We construct a set of paths $P$ as follows: we choose a path $p_{u,v}$ as in Lemma 1 (for some pair of nodes $(u, v) \in S^2$) and remove all its edges from $T$ (and $G'$) and add $p_{u,v}$ to $P$. By lemma 2, the remaining tree is still a Steiner tree and so we can do this until $T$ does not contain any node in $S$. We can indeed pair them because $|S|$ is even. Note that $P$ is a set of edge-disjoint paths (since we choose a path after removing the previous ones, thus making the edges unavailable).

Clearly, after having done this, $T$ is either empty or contains some edge that would have been needed to connect the paths in $P$ to each other in such a way that they form a tree. Therefore, the sum of the costs of the paths in $P$ is lower than $W$. Let $P_E$ be the set of all the edges in $P$: $P_E = \bigcup_{p \in P} edges(p)$. Thus:

$$\sum_{e \in P_E} we \leq W \quad (1)$$

We now choose the paths in $P$ to be of minimal weight (i.e. the shortest paths). The equation 1 still holds as the cost of the shortest paths can only be lower or equal to the cost of the paths chosen in the previous step.

For ease of computation, we compute the shortest paths from each in-node to some other in-node using $spc_{G'}(u)$. For this reason, every edge in $P$ is counted twice using this computation. Thus:

$$\frac{1}{2} \sum_{u \in S} spc_{G'}(u) \leq \sum_{e \in P_E} we \quad (2)$$

By dividing the cost by 2, we get the same lower bound.

If $|S|$ is not even: In this case, we choose some terminal $v$ that we ignore (we subtract the cost of the path containing it). Let $G'' = (V'' \setminus \{v\}, E'' \setminus incident_edges(v))$. This new graph $G''$ has an even number of terminals. Let $W''$ be the cost of the best Steiner tree for $G''$. Then, by applying the even case on $G''$, $LB(G'')$ is a valid lower bound for $G''$ (i.e. $LB(G'') \leq W''$). Because we ignore one terminal, $W'' \leq W$. And since $LB(G') \leq LB(G'')$ (because we subtract a non-negative value to it), $LB(G') \leq W$. Therefore $LB(G')$ is also a valid lower bound.
B Pseudo-code of algorithms

These pseudo-codes correspond to the algorithms presented in Section 4.

First, a series of DFS algorithms that vary on which type of edges are traversed is available in Algorithm 1.

Algorithm 1 Auxiliary DFS functions.

1: function BLUEDFS\( (n, \text{visited}[]) \)
   \( \triangleright \) DFS starting in \( n \), going through in-edges only.
   \( \triangleright \) Returns the set of visited nodes.
2: \( \text{visited}[o] \leftarrow \text{false} \)
3: for all \( e = (n, o) \in \text{adj}[n] \) do
4:   if \( \neg\text{isOutEdge}(e) \land \neg\text{visited}[o] \) then
5:     \( \text{BLUEDFS}(o) \)
6: return \( \text{visited} \)
7: end function

1: function PINKDFS\( (n, \text{visited}[], \text{blue}[], \text{OE}) \)
   \( \triangleright \) DFS starting in \( n \), going on all edges but stopping at \( \text{blue} \) nodes.
   \( \triangleright \) Returns the set of out-edges reaching a \( \text{blue} \) node.
2: \( \text{visited}[o] \leftarrow \text{false} \)
3: for all \( e = (n, o) \in \text{adj}[n] \) do
4:   if \( \neg\text{visited}[o] \) then
5:     if \( \text{blue}[o] \) then \( \text{OE.add}(e) \)
6: else \( \text{PINKDFS}(o, \text{blue}, \text{OE}) \)
7: return \( \text{OE} \)
8: end function

1: function ISLANDDFS\( (n, \text{visited}[], r) \)
   \( \triangleright \) DFS starting at \( n \), not using \( r \) nor out-edges.
   \( \triangleright \) Returns the set of visited nodes.
2: \( \text{visited}[o] \leftarrow \text{false} \)
3: for all \( e = (n, o) \in \text{adj}[n] \) do
4:   if \( \neg\text{isOutEdge}(e) \land \neg\text{visited}[o] \) then
5:     if \( \text{isNode}(r) \land r \neq o \lor (\text{isEdge}(r) \land r \neq e) \) then
6:        \( \text{ISLANDDFS}(o) \)
7: return \( \text{visited} \)
8: end function

1: function OUTEDGESDFS\( (n, \text{visited}[], r, \text{island}[], \text{reach}[], \text{OE}) \)
   \( \triangleright \) DFS starting in \( n \), not using \( r \). Stops recursion when reaching a non \text{island} node that is in \text{reach}.
   \( \triangleright \) Returns the set of out-edges reaching a said nodes.
2: \( \text{visited}[o] \leftarrow \text{false} \)
3: for all \( e = (n, o) \in \text{adj}[n] \) do
4:   if \( \text{reach}[o] \land \neg\text{island}[o] \) then
5:     \( \text{OE.add}(e) \)
6: else if \( \neg\text{visited}[o] \) then
7:   if \( \text{isNode}(r) \land r \neq o \lor (\text{isEdge}(r) \land r \neq e) \) then
8:      \( \text{OUTEDGESDFS}(o, r, \text{island}, \text{reach}, \text{OE}) \)
9: return \( \text{OE} \)
10: end function

Algorithm 2 presents the \text{reachable} algorithm described in 4.1. It first runs a \text{BLUEDFS} and if failure is detected, we run a \text{PINKDFS} to compute the explanations.

Algorithm 2 Algorithm \text{reachable}

1: procedure \text{REACHABLE}(n) \( \triangleright n \) is an in-node
2:   \( \text{blue} \leftarrow \text{BLUEDFS}(n, []) \)
3:   \( \text{OE} \leftarrow [] \)
4: for all \( o \in \text{nodes} \) do
5:   if \( \neg\text{blue}[o] \) then \( \triangleright \) Not visited: run DFS
6:     \( \text{OE} \leftarrow \text{PINKDFS}(o, [], \text{blue}, \emptyset) \)
7: break
8: if \( \text{OE} \neq \emptyset \) then
9: \( \text{fail_reason}((\forall e' \in \text{OE} e') \land n \land o) \)
10: end procedure

The two algorithms in 3 correspond to the two algorithms for cycle detection and prevention. Both are based on a simple query to the Union-Find data structure.

Algorithm 3 Algorithms for cycles

1: procedure \text{CYCLE_DETECT}(e) \( \triangleright e \) is an in-edge
2: \( (u, v) \leftarrow \text{endnodes}[e] \)
3: \( \text{path} \leftarrow \text{UF.path}(u, v) \)
4: if \( \text{path} \neq [] \) then
5: \( \text{fail_reason}(e, \forall e' \in \text{path} e') \)
6: end procedure

1: procedure \text{CYCLE_PREVENT}(n) \( \triangleright n \) is an in-node
2: for all \( e = (n, v) \in \text{adj}[n] \) do
3: \( \text{path} \leftarrow \text{UF.path}(u, v) \)
4: if \( \text{path} \neq [] \) then
5: \( \text{fix_false}(e, \forall e' \in \text{path} e') \)
6: end procedure

Algorithm 4 corresponds to the Section 4.2 of the main paper. The algorithm is based on Tarjan’s algorithm for bi-connected components. We add a stack \( S \) that keeps track of the visited in-nodes. This lets us know whether an articulation (or a bridge) is between two in-nodes. The variable \( \text{tempExpN} \) and \( \text{tempExpE} \) build partial explanations. These sets record, for each required node/edge, an in-node that could only be reached through that given node/edge. The variable \( \text{result} \) adds another node to those explanations. This new node is an in-node on the opposite side of the articulation of bridge. This gives, for each required articulation or bridge, the two in-nodes that require that node or edge. Finally, these explanations are completed with out-edges by running two different DFS algorithms (\text{ISLANDDFS} and \text{OUTEDGESDFS}).
Algorithm 4 Algorithm articulations

1: function FIND_BRIDGES(u, visited[], S, s)
   ▷ S is the stack of visited in-nodes
   ▷ s is the stack of traversed edges.
2:     visited[u] ← true
3:     depth[u], low[u], count ← count + 1
4:     if isInNode(u) then S.push(u)
5:     prevTop ← S.top(); topChanged ← nil
6:     for all e ∈ { adj[u] | ¬isOut(e) } do
7:         v ← otherEndNode(e, u)
8:         if ¬visited[v] then
9:             s.push(e); parent[v] ← u
10:                FIND_BRIDGES(v, visited, S, s)
11:     if ¬isInNode(v) then topChanged ← S.top()
12:     if low[v] ≥ depth[u] then ▷ u is an articulation
13:         lastEdge ← nil; counter ← 0
14:         while lastEdge ≠ e do ▷ Unfold bi-CC
15:             lastEdge ← s.top(); s.pop()
16:         counter ← counter + 1
17:         if counter = 1 ∧ ¬isIn(e) then ▷ Bridge
18:             tmpExplE.add((e, S.top()))
19:         else if ¬isIn(u) then ▷ Articulation
20:             tmpExplN.add((u, S.top()))
21:     while S.top() ≠ prevTop do
22:         S.pop(); ▷ Recover previous state
23:     low[u] ← min(low[u], low[v])
24:     else ▷ Normal Tarjan execution
25:         s.push(e)
26:         low[u] ← min(low[u], depth[v])
27:     if isIn(u) then ▷ Recover explanations
28:         for all (req, cause) ∈ tmpExplE ∪ tmpExplN do
29:             result.add( (req, cause, u) )
30: else if topChanged ≠ nil then ▷ Notify above level that we hit a node
31:     S.push(topChanged)
32: return (result, visited)
33: end function

1: procedure ARTICULATIONS(n)
2:     h ← new Stack(), s ← new Stack()
3:     (arts, tarjan_v) ← FIND_BRIDGES(n, [], h, s)
4:     for all (ar, c1, c2) ∈ arts do
5:         reachable ← ISLANDDFS(c1, ar)
6:         eo ← OUTEDGESDFS(c1, ar, reachable, tarjan_v, Ø)
7:         fix_true(ar, c1 ∧ c2 ∧ ∨ e′ ∈ eo)
8: end procedure