

5.337 tree

	DESCRIPTION	LINKS	GRAPH
Origin	N. Beldiceanu		
Constraint	tree(NTREES, NODES)		
Arguments	NTREES : <code>dvar</code> NODES : <code>collection(index-int, succ-dvar)</code>		
Restrictions	$NTREES \geq 1$ $NTREES \leq NODES $ <code>required(NODES, [index, succ])</code> $NODES.index \geq 1$ $NODES.index \leq NODES $ <code>distinct(NODES, index)</code> $NODES.succ \geq 1$ $NODES.succ \leq NODES $		
Purpose	Cover a digraph G by a set of trees in such a way that each vertex of G belongs to one distinct tree. The edges of the trees are directed from their leaves to their respective roots.		
Example	$2, \left\langle \begin{array}{l} \text{index} - 1 \quad \text{succ} - 1, \\ \text{index} - 2 \quad \text{succ} - 5, \\ \text{index} - 3 \quad \text{succ} - 5, \\ \text{index} - 4 \quad \text{succ} - 7, \\ \text{index} - 5 \quad \text{succ} - 1, \\ \text{index} - 6 \quad \text{succ} - 1, \\ \text{index} - 7 \quad \text{succ} - 7, \\ \text{index} - 8 \quad \text{succ} - 5 \end{array} \right\rangle$		
	<p>The <code>tree</code> constraint holds since the graph associated with the items of the <code>NODES</code> collection corresponds to two trees (i.e., $NTREES = 2$): each tree respectively involves the vertices $\{1, 2, 3, 5, 6, 8\}$ and $\{4, 7\}$. They are depicted by Figure 5.603.</p>		
	<pre> graph TD 1 --- 5 1 --- 6 5 --- 2 5 --- 3 5 --- 8 7 --- 4 </pre>		
Symmetry	Items of <code>NODES</code> are <code>permutable</code> .		

Remark

Given a complete digraph of n vertices as well as an unrestricted number of trees NTREES, the total number of solutions of the corresponding `tree` constraint corresponds to the sequence A000272 of the On-Line Encyclopedia of Integer Sequences [357].

Extension of the `tree` constraint to the *minimum spanning tree* constraint is described in [127, 316, 319].

Algorithm

An `arc-consistency` filtering algorithm for the `tree` constraint is described in [38]. This algorithm is based on a necessary and sufficient condition that we now depict.

To any `tree` constraint we associate the digraph $G = (V, E)$, where:

- To each item `NODES[i]` of the `NODES` collection corresponds a vertex v_i of G .
- For every pair of items (`NODES[i]`, `NODES[j]`) of the `NODES` collection, where i and j are not necessarily distinct, there is an arc from v_i to v_j in E if j is a potential value of `NODES[i].succ`.

A strongly connected component C of G is called a *sink component* if all the successors of all vertices of C belong to C . Let `MINTREES` and `MAXTREES` respectively denote the number of sink components of G and the number of vertices of G with a loop.

The `tree` constraint has a solution if and only if:

- Each sink component of G contains at least one vertex with a loop,
- The domain of `NTREES` has at least one value within interval `[MINTREES, MAXTREES]`.

Most likely, the worst case complexity of the algorithm proposed in [38] may be enhanced by using the linear time algorithm presented in [197] for computing the `strong articulation points` of the digraph G .

Reformulation

The `tree` constraint can be expressed in term of (1) a set of $|\text{NODES}|^2$ reified constraints for avoiding circuit between more than one node and of (2) $|\text{NODES}|$ reified constraints and of one sum constraint for counting the trees:

1. For each vertex `NODES[i]` ($i \in [1, |\text{NODES}|]$) of the `NODES` collection we create a variable R_i that takes its value within interval `[1, |NODES|]`. This variable represents the *rank* of vertex `NODES[i]` within a solution. It is used to prevent the creation of circuit involving more than one vertex as explained now. For each pair of vertices `NODES[i]`, `NODES[j]` ($i, j \in [1, |\text{NODES}|]$) of the `NODES` collection we create a reified constraint of the form `NODES[i].succ = NODES[j].index \wedge $i \neq j \Rightarrow R_i < R_j$` . The purpose of this constraint is to express the fact that, if there is an arc from vertex `NODES[i]` to another vertex `NODES[j]`, then R_i should be strictly less than R_j .
2. For each vertex `NODES[i]` ($i \in [1, |\text{NODES}|]$) of the `NODES` collection we create a 0-1 variable B_i and state the following reified constraint `NODES[i].succ = NODES[i].index $\Leftrightarrow B_i$` in order to force variable B_i to be set to value 1 if and only if there is a loop on vertex `NODES[i]`. Finally we create a constraint `NTREES = $B_1 + B_2 + \dots + B_{|\text{NODES}|}$` for stating the fact that the number of trees is equal to the number of loops of the graph.

Systems

`tree` in **Choco**.

See also

common keyword: `cycle`, `graph_crossing`, `map` (*graph partitioning constraint*), `proper_forest` (*connected component, tree*).

implied by: `binary_tree`.

related: `global_cardinality_low_up_no_loop`, `global_cardinality_no_loop` (*can be used for restricting number of children since discard loops associated with tree roots*).

shift of concept: `stable_compatibility`, `tree_range`, `tree_resource`.

specialisation: `binary_tree` (*no limit on the number of children replaced by at most two children*), `path` (*no limit on the number of children replaced by at most one child*).

uses in its reformulation: `tree_range`, `tree_resource`.

Keywords

constraint type: graph constraint, graph partitioning constraint.

filtering: strong articulation point, arc-consistency.

final graph structure: connected component, `tree`, `one_succ`.

Arc input(s)	NODES
Arc generator	<code>CLIQUE</code> \mapsto <code>collection(nodes1, nodes2)</code>
Arc arity	2
Arc constraint(s)	<code>nodes1.succ = nodes2.index</code>
Graph property(ies)	<ul style="list-style-type: none"> • <code>MAX_NSCC</code> \leq 1 • <code>NCC</code> = <code>NTREES</code>

Graph model

We use the graph property `MAX_NSCC` \leq 1 in order to specify the fact that the size of the largest strongly connected component should not exceed one. In fact each root of a tree is a strongly connected component with one single vertex. The second graph property `NCC` = `NTREES` enforces the number of trees to be equal to the number of connected components.

Parts (A) and (B) of Figure 5.604 respectively show the initial and final graph associated with the **Example** slot. Since we use the `NCC` graph property, we display the two **connected components** of the final graph. Each of them corresponds to a tree. The **tree** constraint holds since all strongly connected components of the final graph have no more than one vertex and since `NTREES` = `NCC` = 2.

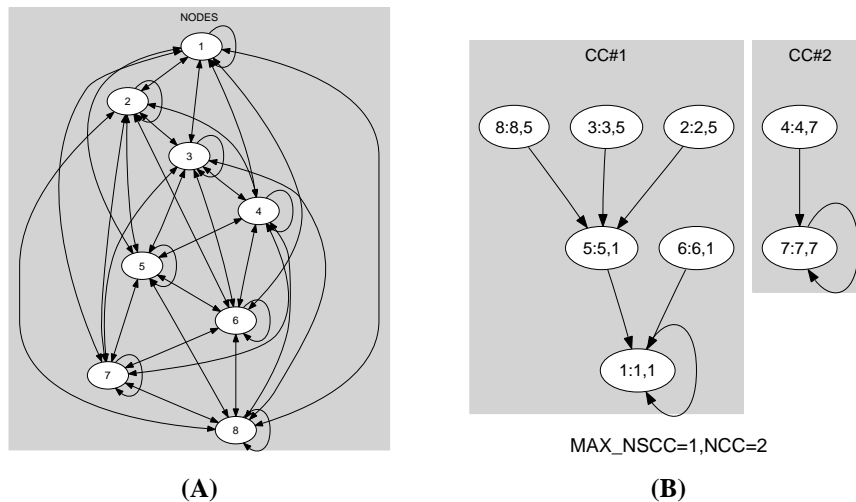


Figure 5.604: Initial and final graph of the tree constraint