MODULE *SpanTree*

This is an algorithm to compute a spanning tree of an undirected graph with a given root. Look up "spanning tree" on the Web to see what that means. You may find pages for finding spanning trees of graphs with weighted edges. The algorithm here effectively assumes each edge has weight 1.

A rooted tree is usually described by a set of nodes with a parent/child relation, where the root is the oldest ancestor of all other nodes. The algorithm computes this relation as a function *mom* where $mom[n]$ equals the parent of node n, except that if n is the root then $mom[n] = n$. If the graph is not connected, then the rooted tree does not contain nodes of the graph that have no path to the root. Such nodes n will have $mom[n] = n$.

A simple algorithm to compute the rooted spanning tree computes a function *dist* where $dist[n]$ is the distance of node n from the root. Initially, $dist[n]$ equals 0 if n is the root and otherwise equals infinity. The algorithm repeatedly performs the following action. It chooses an arbitrary node n that has a neighbor m such that $dist[n] > dist[m] + 1$, and it sets $dist[n]$ to $dist[m] + 1$.

For simplicity, we assume that we're also given a number *MaxCardinality* that's greater than or equal to the number of nodes, and we use *MaxCardinality* instead of infinity. For a reason to be given below, we also modify the algorithm as follows. For a node n with $dist[n] > dist[m\} + 1$, instead of setting $dist[n]$ to $dist[m] + 1$ the algorithm sets it to an arbitrary number d such that $dist[n] > d \geq dist[m\} + 1$.

EXTENDS *Integers; FiniteSets*

We represent the graph by a set of *Nodes* of nodes and a set *Edges* of edges. We assume that there are no edges from a node to itself and there is at most one edge joining any two nodes. We represent an edge joining nodes m and n by the set $\{m, n\}$. We let *Root* be the root node.

CONSTANTS *Nodes; Edges; Root; MaxCardinality*

This assumption asserts mathematically what we are assuming about the constants.

$$\begin{aligned}
\text{ASSUME}\ & \land\ Root \in Nodes \\
& \land\ \forall e \in Edges : (e \subseteq Nodes) \land (Cardinality(e) = 2) \\
& \land\ MaxCardinality \in Nat \\
& \land\ MaxCardinality \geq Cardinality(Nodes)
\end{aligned}$$

This defines $Nbrs(n)$ to be the set of neighbors of node n in the graph–that is, the set of nodes joined by an edge to n.

$$Nbrs(n) \triangleq \{m \in Nodes : \{m; n\} \in Edges\}$$

The spec is a straightforward TLA+ spec of the algorithm described above.

VARIABLES *mom; dist*

$$vars \triangleq \langle mom; dist \rangle$$

$$\begin{aligned}
TypeOK \triangleq\ & \land\ mom \in [Nodes \rightarrow Nodes] \\
& \land\ dist \in [Nodes \rightarrow Nat]
\end{aligned}$$

$$\begin{aligned}
Init \triangleq\ & \land\ mom = [n \in Nodes \mapsto n] \\
& \land\ dist = [n \in Nodes \mapsto \text{IF } n = Root \text{ THEN } 0 \text{ ELSE } MaxCardinality]
\end{aligned}$$

$$\begin{aligned}
Next \triangleq\ & \exists\, n \in Nodes : \\
& \quad \exists\, m \in Nbrs(n) : \\
& \quad\quad \land\ dist[m] < 1 + dist[n]
\end{aligned}$$

$$\begin{aligned}&\land \exists\, d \in (dist[m]+1) \,..\, (dist[n]-1) :\\ &\qquad \land\ dist' \ = [dist \ \text{EXCEPT}\ ![n] \ = d]\\ &\qquad \land\ mom' = [mom \ \text{EXCEPT}\ ![n] = m]\end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \text{WF}_{vars}(Next)$$

The formula WF_*vars*(*Next*) asserts that a behavior must not stop if it's possible to take a *Next* step. Thus, the algorithm must either terminate (because *Next* equals FALSE for all values of $dist'$ and mom') or else it continues taking *Next* steps forever. Don't worry about it if you haven't learned how to express liveness in TLA+.

A direct mathematical definition of exactly what the function *mom* should be is somewhat complicated and cannot be efficiently evaluated by *TLC*. Here is the definition of a postcondition (a condition to be satisfied when the algorithm terminates) that implies that *mom* has the correct value.

$$\begin{aligned}&PostCondition \triangleq\\ &\quad \forall\, n \in Nodes :\\ &\qquad \lor\ \land\ n = Root\\ &\qquad\quad\ \land\ dist[n] = 0\\ &\qquad\quad\ \land\ mom[n] = n\\ &\qquad \lor\ \land\ dist[n] = MaxCardinality\\ &\qquad\quad\ \land\ mom[n] = n\\ &\qquad\quad\ \land\ \forall\, m \in Nbrs(n) : dist[m] = MaxCardinality\\ &\qquad \lor\ \land\ dist[n] \in 1 \,..\, (MaxCardinality - 1)\\ &\qquad\quad\ \land\ mom[n] \in Nbrs(n)\\ &\qquad\quad\ \land\ dist[n] = dist[mom[n]] + 1\end{aligned}$$

ENABLED *Next* is the TLA+ formula that is true of a state iff (if and only if) there is a step satisfying *Next* starting in the state. Thus, ¬ENABLED *Next* asserts that the algorithm has terminated. The safety property that algorithm should satisfy, that it's always true that if the algorith has terminated then *PostCondition* is true, is asserted by this formula.

$$Safety \triangleq \Box((\lnot \text{ENABLED } Next) \Rightarrow PostCondition)$$

This formula asserts the liveness condition that the algorithm eventually terminates

$$Liveness \triangleq \Diamond(\lnot \text{ENABLED } Next)$$

These properties of the spec can be checked with the model that should have come with this file. That model has *TLC* check the algorithm satisfies properties *Safety* and *Liveness* for a single simple graph with 6 nodes. You should clone that model and change it to try a few different graphs. However, this is tedious. There are two better ways to have *TLC* check the spec. The best is to try it on all graphs with a given number of nodes. The spec with root file *SpanTreeTest* does this. It can very quickly check all graphs with 4 nodes. It takes about 25 minutes on my laptop to check all graphs with 5 nodes. *TLC* will probably run out of space after running for a long time if I tried it for all graphs with 6 nodes. The spec *SpanTreeRandom* tests the algorithm for a randomly chosen graph with a given set of nodes. This allows you easily to repeatedly check different graphs.

As a problem, you can now specify an algorithm that is a distributed implementation of this algorithm. We can view the algorithm in the current module as one in which a node n sets its value of $dist[n]$ by directly reading the values of $dist[m]$ from all its neighbors m. Your problem is to write an algorithm in which nodes learn the values of $dist[m]$ from a neighbor m by receiving messages sent by m. The root r sends an initial message informing its neighbors that $dist[r] = 0$. Subsequently, each node n sends a message containing $dist[n]$ to all its neighbors whenever its value of $dist[n]$ changes.

Your algorithm should have variables mom and $dist$ that implement the variables of the same name in the current algorithm. (Hence, it should implement the current algorithm with a trivial refinement mapping assigning to every variable and constant the variable or constant of the same name.) You can use TLC to check that your algorithm does indeed implement the algorithm in the current module.

You may not know how to write a suitable liveness condition for your algorithm. (To find out how, you would have to look through the available TLA+ documentation.) In that case, just write a safety specification of the form $Init \land \Box[Next]\_vars$ and modify formula $Spec$ of the current module by comment out the $\land \mathrm{WF\_}vars(Next)$ conjunction so it too becomes a safey spec.

When writing your algorithm, you should realize why the $Next$ action in the current module doesn't just set $dist[n]$ to $dist[m]+1$ rather than allowing it to be set to any value in $(dist[m]+1) \,..\, (dist[n]-1)$. If you don't see why, use TLC to find out for you.

* Modification History
* Last modified $Mon\ Jun$ 17 05:52:09 PDT 2019 by $lamport$
* Created $Fri\ Jun$ 14 03:07:58 PDT 2019 by $lamport$