Distributed Spanning Tree Algorithms Coded in IOA: Challenge Problems for Software Analysis and Synthesis Methods Ver. 1.3

Chris Luhrs
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

October 7, 2000

Abstract

This paper presents a collection of IOA programs describing distributed algorithms. These programs solve several problems involving network graph searching. The programs are intended to be used as test examples for formal analysis tools such as simulators, model checkers, and interactive theorem provers and for synthesis tools such as automatic code generators.

In addition, the paper presents formal descriptions of several data types that will be used in the programs. For one algorithm, we also present a proof of correctness.

1 Introduction

The way datatypes are passed from higher automata to lower automata is not present at correct though. This is because it is at present unclear what the exact syntax is for types. The composition style is also out of date. These problems should be fixed as soon as I have front end support to check them. There are also general typos in the paper that I haven't found yet.

[[[Repeat what's in the abstract, expand. Cite IOA papers.]]]

2 Graph Data Types

The spanning tree algorithms use graph data types and operations not built into IOA. Therefore, we write definitions of these graph data types using the Larch Shared Language (LSL). For this paper, we give a parameterization of each function along with a precise English definition of what it does. To make these descriptions useful for theorem-proving purposes descriptions of the algorithms would have to be written in formal logic in the assertion section of the description.

To make the functions work in a compiled version of the IOA programs, they would have to be implemented in some lower level language such as Java. Where appropriate, we add formal assertions about the data types in general.

The two graph data types used in the algorithms presented, GraphWithRoot and WeightedGraphWithRoot, are built up in layers from more general graph types. Each successive data type is a specific case of a previous data type, adding more restrictions and functions.

2.1 Graph

The most basic graph data type is described by Graph.lsl. It uses Set.lsl, which describes the theory of finite sets. Stating Graph(Node, Edge, Graph) essentially guarantees that the types Node, Edge, and Graph are in a correspondence such that every function in the introduces section and every assertion in the assertions section are valid. Essentially, this means a Graph is a set of Nodes and Edges. Edges are directed; they have a root node and a source node. If an edge is in a graph so are its root and source. This is essentially the only constraint on a graph data type satisfying Graph.lsl. Edges may have the same node as target and source, and there may be several edges between two nodes.

```
Graph(Node, Edge, Graph) : trait
 %A directed graph. Edges have a source and a target. Multiple edges
 %may connect the same nodes in the same direction. An edge may go
 %from a node to itself.

  includes Set(Node for E), Set(Edge for E)

  introduces
    empty:                    -> Graph
     %Returns a graph with no nodes and no edges.

    target: Edge              -> Node
     %Returns the target of an edge.

    source: Edge              -> Node
     %Returns the source of an edge.

    addnode: Node, Graph      -> Graph
     %addnode(n, g) returns a graph identical to g except it also
     %contains n. If g already contained n, then g is returned.

    addedge: Edge, Graph      -> Graph
     %If source(e) and target(e) are both in g then addedge(e, g) is
     %identical to g but with e added. Else addedge(e, g) = g.

    delnode: Node, Graph      -> Graph
     %If n is in g then delnode(n, g) is a graph identical to g
     %except without n and all edges connected to n. Else
     %delnode(e, g) = g.

    deledge: Edge, Graph      -> Graph
     %If e is in g then deledge(e, g) is a new graph identical to g
     %except with e removed. Else deledge(e, g) = g.

    makeedge: Node, Node      -> Edge
     %makeedge(n1, n2) is an edge e such that source(e) = n1 and
```

```
    %target(e) = n2.

   nodes: Graph              -> Set[Node]
    %Returns the set of all nodes in a graph.

   edges: Graph              -> Set[Edge]
    %Returns the set of all edges in a graph.

   isedge: Node, Node, Graph -> Bool
    %isedge(n1, n2, g) returns true if and only if there is an edge
    %e in g such that source(e) = n1 and target(e) = n2.

   neighborsin: Node, Graph  -> Set[Node]
    %neighborsin(n1, g) returns the set of all nodes n2 such that
    %there is an edge TO n1 FROM n2 in g.

   neighborsout: Node, Graph -> Set[Node]
    %neighborsin(n1, g) returns the set of all nodes n2 such that
    %there is an edge FROM n1 TO n2 in g.

   ispath: Node, Node, Graph -> Bool
    %ispath(n1, n2, g) returns true if and only if there is a directed
    %path from n1 to n2 in g.

   isconnected: Graph        -> Bool
    %isconnected(g) returns true if and only if there is a directed
    %path from any node to any other node in g.

   reverseedge: Edge         -> Edge
    %reverseedge(e1) returns an edge e2 such that target(e1) = source(e2)
    %target(e2) = source(e1).

 asserts with e : Edge, g : Graph
   (e \in edges(g) => source(e) \in nodes(g) /\ target(e) \in nodes(g))
    %An edge cannot be in a graph unless its endpoints are.
```

2.2 SimpleConnectedGraph

The first refinement of Graph.lsl is SimpleConnectedGraph.lsl. In addition to the requirements of Graph.lsl, SimpleConnectedGraph.lsl also requires that every graph be bidirectional in the sense that for every edge in a graph there is an edge that is identical except target and source are reversed. There can be only one such pair of edges between any two nodes. There are no edges from a node to itself, and the graph is connected. These requirements are stated formally in the assertions section of SimpleConnectedGraph.lsl. In order to have a way to construct SimpleConnectedGraphs without breaking these constraints, we need two new constructors: connectnode and biaddedge. addedge and addnode from graph.lsl are not sufficient since they will either result in an unconnected graph or a directed edge. We also add a way of removing bidirectional edges.

```
SimpleConnectedGraph(Node, Edge, SimpleConnectedGraph) : trait
 %All edges are undirected in the sense that if there is an edge from
 %node1 to node2 there is an edge from node2 to node1.

  includes Graph(Node, Edge, SimpleConnectedGraph for Graph)

  introduces

    neighbors : Node, SimpleConnectedGraph         -> Set[Node]
     %neighbors(n1, g) is the set of nodes n2 such that there is an
     %edge between n1 and n2.

    connectnode : Node, Node, SimpleConnectedGraph ->
                    SimpleConnectedGraph
     %If n1 is in g and n1 does not equal n2 connectnode(n1, n2, g)
     %returns a graph identical to g except n2 (if it was not already
     %there) and a bidirectional edge between n1 and n2 are added.
     %If n1 is not in g, n1 = n2, or n1 and n2 are already connected
     %then g is returned.

    biaddedge : Edge, SimpleConnectedGraph         -> SimpleConnectedGraph
     %If source(e) and target(e) are both in g biaddedge(e, g) returns a
     %graph identical to g except e and reverse(e) are added. If source(e)
     %or target(e) is not in g then g is returned. If there is already a
     %bidirectional edge between source(e) and target(e) then the original
     %bidirectional edge is removed and e and reverse(e) are added.

    bideledge : Edge, SimpleConnectedGraph          -> SimpleConnectedGraph
     %If e is in g bideledge(e, g) returns a graph identical to g but
     %with e and reverseedge(e) removed. Else g is returned.

  asserts with n1, n2 : Node, e1, e2 : Edge, g1 : SimpleConnectedGraph

    (e1 \in edges(g1) /\ e2 \in edges(g1)) /\ ((target(e1) = target(e2)
     /\ source(e1) = source(e2)) => e1 = e2);
     %There is only one edge in each direction between each pair of nodes.

    ~(n1 \in neighbors(n1, g1));
     %There are no edges from a node to itself.

    e1 \in edges(g1) => reverseedge(e1) \in edges(g1);
     %All edges are undirected.
```

```
  isconnected(g1)
   %All graphs are connected.
```

2.3 GraphWithRoot

GraphWithRoot is the first data type used in our IOA code. It is essentially a pair of a graph satisfying SimpleConnectedGraph.lsl and a node in that graph. GraphWithRoot.lsl provides a constructor for building GraphWithRoots and two selectors to pull the node or the graph of a GraphWithRoot.

```
GraphWithRoot(Node, Edge, GraphWithRoot) : trait
 %Graph with root is a pair of a node, the root, and a simple
 %connected graph such that the root is in the graph.

  includes
    SimpleConnectedGraph(Node, Edge, SimpleConnectedGraph)

  introduces

    graphwithroot : Node, SimpleConnectedGraph -> GraphWithRoot
     %Graphwithroot(n, g) is a graph with root such that n is the root
     %and g is the graph provided n is in g. If n is not in g then
     %graphwithroot(n, g) has n as its root and the graph containing
     %the single node n as its graph.

    graphwithroot : Node                       -> GraphWithRoot
     %graphwithroot(n) is a GraphWithRoot with n set as its root and
     %the graph containing the single node n as its graph.

    __.root : GraphWithRoot                    -> Node
     %returns the root of a GraphWithRoot.

    __.g : GraphWithRoot                       -> SimpleConnectedGraph
     %returns the graph of a GraphWithRoot.

  asserts with g1 : GraphWithRoot
    g1.root \in nodes(g1.g)
     %The root of a GraphWithRoot is in the graph of a GraphWithRoot.
```

2.4 WeightedGraph

WeightedGraph.lsl adds the requirement that every edge have some sort of weight associated with it. In the case of the Bellman-Ford algorithms described in sections 5 and 6, the weight is a real number representing distance. The weight of an edge may represent some other property such as color though. WeightedGraph.lsl includes an additional selector that returns the weight of an edge as well as a constructor that creates an edge with a specified root, source, and weight.

Note that while Graph.lsl does not state that an edge has weight, it does not exclude it from having one either. In fact, edges in WeightedGraphs may have properties other than weight, but other selectors would be required to make these accessible.

```
WeightedGraph(Node, Edge, WeightedGraph, Weight) : trait

  includes Graph(Node, Edge, WeightedGraph for Graph)

  introduces
    weight : Edge                    -> Weight
     %Returns the weight of an edge.

    wmakeedge: Node, Node, Weight   -> Edge
     %wmakeedge(n1, n2, w) returns an edge e such that source(e)=n1,
     %target(e)=n2 and weight(e)=w.
```

2.5 SimpleWeightedGraph

SimpleWeightedGraph.lsl essentially combines the constraints of SimpleConnectedGraph and WeightedGraph. It provides another constructor that both obeys the requirements of simple connected graphs and allows the weight of an edge to be specified.

```
SimpleWeightedGraph(Node, Edge, Graph, Weight) : trait

  includes SimpleConnectedGraph(Graph for SimpleConnectedGraph),
    WeightedGraph(Graph for WeightedGraph)

  introduces
    waddedge : Node, Node, Weight, Graph -> Graph
     %Like connectedge in SimpleConnectedGraph.lsl, but the weight of
     %the edge added is specified.

    wconnectnode : Node, Weight, Graph    -> Graph
     %Like connectnode in SimpleConnectedGraph.lsl, but the weight of
     %the edge used to connect the node is specified.
```

2.6 WeightedGraphWithRoot

WeightedGraphWithRoot.lsl is the data type used in the Bellman-Ford algorithm. Essentially, it makes a GraphWithRoot from a SimpleWeightedGraph.

```
WeightedGraphWithRoot(Node, Edge, Graph, Weight) : trait

  includes SimpleWeightedGraph,
   GraphWithRoot(Graph for GraphWithRoot)
```

3 Simple Spanning Tree Algorithm

The first algorithm we look at is a simple spanning tree algorithm.

3.1 Overview

A tree with a root is passed into the automaton, as well as the *NodeType*, *EdgeType*, and *GraphType* for that graph. The algorithm creates two types of automata: *Node* and *Channel*. One *Node* automaton is created for every node in the graph that is taken in as a parameter. Each *Node* is told only who its neighbors are and whether or not it is the root. For each directed edge, one *Channel* automaton is created. Thus, there is one channel in each direction for every pair of connected nodes.

The *Channel* automaton acts in a standard manner. When a $send(i, j, m)$ occurs, $Channel_{(i,j)}$ puts m at the end of its queue. A $receive(i, j, m)$ action is enabled when the first message in the queue is m and removes m from the queue. In this algorithm, all messages are *null*.

The algorithm starts with an *initialize* procedure in the root node. *initialize* adds all of its neighbors to the set of nodes to send a message to, *outgoing*. As long as a node is in *outgoing*, root may send a message to it. If it does so then that node is removed from *outgoing*.

When a node that is not root receives a message it checks to see if it already has a parent. If it does not, the node will make the sender of the message its parent. It will then send messages to all of its neighbors except its parent. Every node except root will only send a message if it has a parent, so if a node has a parent assigned to it, it will in fact be connected to a tree containing root. The algorithm proceeds until every node is assigned a parent and sends a message to all its neighbors.

3.2 Type Declaration

```
%This program emulates an algorithm for producing a spanning tree.

%Typing issues still here.

type NullType = enumeration of null
type ParentType = union of node : NodeType, n : NullType
```

3.3 Channel Automaton

```
automaton Channel(IndexType : Type, i, j : IndexType, MessageType: Type)

  signature
    input send(const i, const j, message : MessageType)
    output receive(const i, const j, message : MessageType)

  states
    queue : Seq[MessageType] := {}
```

```
  transitions

    input send(i, j, message)
      eff queue := queue |- message

    output receive(i, j, message)
      pre queue ~= {} /\ head(queue) = message
      eff queue := tail(queue)
```

3.4 Node Automaton

```
automaton SpanningNode(i: NodeType, neighbors : Set[NodeType],
 isroot : Bool)

  signature
    input receive(j : NodeType, const i, m : NullType) where j \in neighbors
    output send(const i, j : NodeType, m : NullType)
    internal initialize(const i)

  states
    parent : ParentType := n(null),
    outgoing : Set[NodeType] := { },
    initialized : Bool := false

  transitions

    input receive(j, i, m)
      eff
        if ~isroot /\ (parent = n(null))
          then parent := node(j);
          outgoing := (neighbors - {parent.node}) fi

    output send (i, j, m)
      pre j \in outgoing
      eff outgoing := delete(j, outgoing)

    internal initialize(i)
      pre isroot /\ ~initialized
      eff
        outgoing := neighbors;
        initialized := true
```

3.5 The Complete System

```
automaton SpanningTree(GraphType : Type,
 EdgeType : Type, graph : GraphType)

  assumes GraphWithRoot(NodeType, EdgeType, GraphType)

  compose

    SpanningNode(i, neighbors(i, graph.g), false)
      for i : NodeType where i \in delete(graph.root, nodes(graph.g)) ;

    SpanningNode(graph.root, neighbors(graph.root, graph.g), true);

    Channel(NodeType, i, j, NullType )
      for i, j : NodeType where \E e : EdgeType
       (e \in edges(graph.g) /\ i = source(e) /\ j = target(e))
```

3.6 Proof

We now a present a proof of correctness of the algorithm described above. We approach the proof as a series of steps divided into safety and liveness conditions. The safety section ends by proving the algorithm produces a spanning tree containing root at every step. The liveness section proves the algorithm terminates and eventually produces a spanning tree containing all nodes. To simplify notation, we ignore the union constructors in ParentType. Hence, we say *null* instead of $n(null)$.

We begin with a lemma.

Lemma (Queue lemma). *Let A be an automaton, with a sequence s as one of its states. Let $\{a(x)\}$ be a task of A for which $a(x)$ is enabled exactly when* $\text{head}(s) = x$ *and sets s to* $\text{tail}(s)$ *for all x. Assume all other actions may only append elements to the end of s. Then if y is the nth element of s in some state of a fair execution of A where n is an integer then $a(y)$ occurs in that fair execution of A.*

Proof. We prove the statement by induction on n. If $n = 1$ then $\text{head}(s) = y$. In this case, $a(y)$ is enabled, and no other action of the task $\{a(x)\}$ can be enabled. Since no other action can change $\text{head}(s)$ and we are considering a fair execution, $a(y)$ must occur.

Now assume the statement is true if there are only k elements before y, and that there are $k+1$ elements before y in s. Let $\text{head}(s) = z$. Then $a(z)$ is the and only action enabled in its task and will remain so until $a(z)$ occurs. When it does s is set to $\text{tail}(s)$. Other actions may only change append elements to s. In particular, no action may change the elements of s before y. Thus, when $s := \text{tail}(s)$, y becomes the kth element of s. By inductive hypothesis, $a(y)$ must occur, and the statement is true. □

Let the derived state variable *ptree* be an undirected graph defined as follows: *ptree* contains *root*, every node with parent not equal to *null*, and an edge between every node i with non-null parent and $parent_i$.

3.6.1 Safety Properties

Theorem 3.1. *Every $receive(i, j, m)$ is preceded by a $send(i, j, m)$. Furthermore, at most one $receive(i, j, m)$ occurs for each $send(i, j, m)$.*

Proof. If $receive(i, j, m)$ is a viable input action in $Node_j$ then i is in $neighbors_j$. Thus, $Channel_{(i,j)}$ exists. Thus, for a $send(i, j, m)$ to occur, $queue_{(i,j)}$ must be non-empty. This means that head($queue_{(i,j)}$) is well-defined and head($queue_{(i,j)}$) is actually the first element of $queue_{(i,j)}$. Thus, for $send(i, j, m)$ to occur $queue_{(i,j)}$ must have m in it. $queue_{(i,j)}$ is initially empty and the only way to add a message to $queue_{(i,j)}$ is by a $send(i, j, m)$. When a $receive(i, j, m)$ occurs, $queue_{(i,j)}$ is set to tail($queue_{(i,j)}$), removing the first message from the queue. Thus, for each m added to $queue_{(i,j)}$ only one $receive(i, j, m)$ can occur. Since each $send(i, j, m)$ adds m only once to $queue_{(i,j)}$, only one $receive(i, j, m)$ can occur for each $send(i, j, m)$. ☐

Theorem 3.2. *$parent_i$ is non-null or i is root if*

1. *$outgoing_i$ is non-empty*
2. *$queue_{(i,j)}$ is non-empty*
3. *i is in ptree*

Proof. The proof is by induction on the number of actions in *ptree*. In the initial state, $outgoing_i$ and $queue_{(i,j)}$ are empty for all i and j, and *root* is the only node in *ptree*. Thus, the statement is true. Now assume that the statement is true in some state and consider the next action that occurs. An $initialize(i)$ can only occur if i is *root* and only changes $outgoing_i$. *root* is in *ptree* by definition. Thus, if the statement is true before an *initialize* occurs, it will be true after it occurs. A $send(i, j, m)$ adds only to $queue_{(i,j)}$. $outgoing_i$ must contain j for $send(i, j, m)$ to be enabled. Thus, in the pre-state of a $send(i, j, m)$, i is in *ptree* by inductive hypothesis, so a $send(i, j, m)$ will keep the statement true if it was true before. A $receive(i, j, m)$ can only make $outgoing_j$ non-empty and set $parent_j$ to i. For it to be enabled, $queue_{(i,j)}$ must be non-empty. Thus, in the pre-state $parent_i$ is non-null or i is *root*. In either case, i is already in *ptree*, so $receive(i, j, m)$ will not add i to *ptree*. If $parent_j$ is non-null $outgoing_i$ will be set to $neighbors_i$. In this case though, if j is not *root* then $receive(i, j, m)$ will set $parent_j$ to i. Thus, if the statement is true before a $receive(i, j, m)$ occurs it will be true after. ☐

Theorem 3.3. *If an edge or node is in ptree in state s then it is in ptree in any state reachable from s.*

Proof. *ptree* is dependent only on parent variables. In particular, a node or edge is in *ptree* because $parent_i = j$ where j is not null. The only action that changes $parent_i$ is $receive(k, i, m)$ for some k, and $receive(k, i, m)$ only changes $parent_i$ if it is *null*. Thus, once $parent_i$ is set it does not change, and any node or edge in *ptree* because of the value of $parent_i$ will still be in *ptree*. ☐

Theorem 3.4. *ptree is a tree containing root in any reachable state.*

To prove this theorem we will use the following graph theoretic result:

Lemma. *If a tree is altered by adding a node not in the tree and an edge between that node and a node originally in the tree, the resulting graph is still a tree.*

Proof. The proof is by induction on the number of actions. In the initial state, $ptree$ is the graph consisting only of root since $parent_i = null$ for all i. Now assume $ptree$ is a tree containing $root$ in state s. Now consider what can happen after a single transition. $send(i, j, m)$ and $initialize(i)$ do not change $ptree$ since they do not change parent variables, so receives are the only actions that have to be looked at. $receive(i, j, m)$ can only change $parent_j$ and will only change $parent_j$ if it is currently $null$. Thus, we may assume $parent_j$ is null. If a $receive(i, j, m)$ occurs then $queue_{(i,j)}$ is non-empty so i is in $ptree$.

If j is $root$ then a $receive(i, j, m)$ will not change $parent_j$, so we may assume j is not $root$. Assume $parent_k = j$ for some node k. Then a $receive(j, k, m)$ must have occurred, so $queue_{(i,j)}$ was non-empty when it did. Thus, $parent_j$ was and still is non-null. This is a contradiction so $parent_k$ is not j for any k.

Since $parent_j$ is $null$ and j is not the parent of any other node, j is not in $ptree$ in state s. Thus, when $recieve(i, j, m)$ occurs $parent_j$ will be set to i. This will add j and an edge between i and j to $ptree$ by definition of $ptree$. Since i is already in $ptree$, this will change $ptree$ by adding a node not in $ptree$ and an edge between that node and a node already in $ptree$. Since $ptree$ was a tree containing $root$ before this occurred, it must still be a tree containing $root$. □

Theorem 3.5. *If e_1 is an edge in ptree then there is an edge e_2 in graph.g that has the same endpoints.*

Proof. For an edge between i and j to be in $ptree$, $parent_i = j$ or $parent_j = i$. Assume without loss of generality that $parent_j = i$. The only action that can change, $parent_j$ to i is a $receive(i, j, m)$, so i must be in $neighbors_j$, so i is a neighbor of j in $graph.g$. Thus, by definition, there is an edge between them in $graph.g$. □

Theorem 3.6. *ptree is a subtree of $graph_g$. (When we say this we assume that two edges are the same if they have the same endpoints.)*

Proof. $ptree$ is a tree in any reachable state s by 3.4. By 3.5, $ptree$ contains only edges of $graph.g$. Thus, $ptree$ is a subtree of $graph.g$. □

3.6.2 Liveness Properties

Theorem 3.7. *For each i and j, $send(i, j, m)$ occurs at most once.*

Proof. $send(i, j, m)$ is enabled exactly when j is in $outgoing_i$ and $m = null$. $outgoing_i$ is initially empty, and when a $send(i, j, m)$ occurs, j is removed from $outgoing_i$. Thus, the number of times $send(i, j, m)$ occurs is no more than the number of times j is added to $outgoing_i$. The only actions that can add nodes to $outgoing_i$ are $initialize(i)$ and $receive(k, i, m)$ for some node k.

If i is $root$ then $receive(k, i, m)$ will not change $outgoing_i$. An $initialize(i)$ can only occur if $initialized_i$ is $false$. When an $initialize(i)$ occurs it sets $initilialized_i$ to $true$. No other action alters $initialized_i$, so an $initialize(i)$ can only occur once. In the case where i is root at most one $initialize(i)$ can occur, and no $receive(k, i, m)$ will change $outgoing_i$. Thus, $outgoing_i$ can only have nodes added to it at most once.

If i is not $root$, then $initialize(i)$ is never enabled. A $receive(k, i, m)$ only affects $outgoing_i$ if $parent_i = null$. If a $receive(k, i, m)$ occurs when $parent_i$ is $null$ then $parent_i$ is set to k.

Thus, if a $receive(k, i, m)$ has already occurred for some k, another $receive(k, i, m)$ for any k will not affect $outgoing_i$ since $parent_i$ will not be $null$. In either case, any given node can only be added to $outgoing_i$ once, so a $send(i, j, m)$ can occur at most once for any ordered pair of i and j. ☐

Theorem 3.8. *For any fair execution of SpanningTree, eventually a state will be reached where no action is enabled.*

Proof. We will prove this by bounding the number of each type of action that can occur. $initialize(i)$ can only be enabled if i is $root$. $initialize(i)$ sets $initialized_i$ to $true$, and no other action alters the value of $initialized_i$. Since $initialized_i$ must be $false$ for $initialize(i)$ to be enabled, $initialize(i)$ can occur at most once.

By 3.7, $send(i, j, m)$ can occur at most once for each ordered pair of i and j. Thus, the number of send actions is bounded by $|nodes(graph_g)|^2$. Furthermore, there is no more than one receive action for each send action by 3.1, so the number of receive actions is also bounded. Thus, total number of transitions in any execution of Spanningtree is finite. Thus, in any fair execution of Spanningtree no actions are enabled in the final state. ☐

Theorem 3.9. *For any fair execution, if a $send(i, j, m)$ occurs then a $receive(i, j, m)$ must occur.*

Proof. The only action that adds elements to $queue_{(i,j)}$ is $send(i, j, m)$, and it adds only one. $queue_{(i,j)}$ is initially empty so, it always contains finitely many elements. By 3.8, the algorithm terminates so every task partition induces the same set of possible executions. Thus, the queue lemma applies. ☐

Theorem 3.10. *For all fair executions of SpanningTree, there is a state s such that ptree contains all nodes in graph.g.*

Proof. Assume $ptree$ never contains all the nodes in $graph.g$. A node i is in $ptree$ if it is $root$ or if $parent_i$ is non-null. Once a node is in $ptree$ it remains in $ptree$, so the number of nodes in $ptree$ monotonically increases. Let n be the maximum number of nodes in $ptree$. Assume n is less than the number of nodes in $graph.g$. $ptree$ is always non-empty since it contains $root$. Since $graph.g$ is connected there must be an edge in $graph.g$ between a node i that is eventually in $ptree$ and a node j that never is. Then j is in $neighbors_i$ by definition.

Consider the case where i is $root$. $initialized_i$ is initially $false$ and the only action that can set it to $true$ is $initialize(i)$. $initialize(i)$ is enabled as long as $initialized_i$ is $false$ so an $initialize(i)$ must occur because the execution is finite. When $initialize(i)$ occurs, j is placed in $outgoing_i$ since j is in $neighbors_i$. Thus, $send(i, j, null)$ is enabled. It is enabled as long as j is in $outgoing_i$. The only action that could remove j from $outgoing_i$ is a $send(i, j, m)$, so a $send(i, j, m)$ must occur. Then by 3.9, $receive(i, j, m)$ must also occur. When it does if $parent_j$ is not $null$ then $parent_j$ will be set to i, putting j in $ptree$. This is a contradiction.

Now assume i is not $root$. i is in $ptree$, so $parent_i$ is non-null by 3.2. Since $parent_i$ is initially $null$, a $receive(k, i, m)$ must have occurred while $parent_i$ was $null$. When it did $outgoing_i$ was set to $neighbors_i$. Thus, j was put in $neighbors_i$. As long as j is in $outgoing_i$ $send(i, j, null)$ is enabled. As in the previous case, a $send(i, j, m)$ must eventually occur setting $parent_j$ to i if it is still $null$. Again, this is a contradiction since j will then be in $ptree$. Therefore, $ptree$ eventually contains all nodes in $graph.g$. ☐

Theorem 3.11. *For all fair executions of SpanningTree, there is a states s such that ptree is a spanning tree of graph.g in s and all states following s.*

Proof. There is a state s where all nodes of *graph.g* are in ptree by 3.10. In all states, *ptree* is a subtree of *graph.g* by 3.6 , so *ptree* is a spanning tree of *graph.g* in this state. Once an edge or node is in *ptree* it is not removed by 3.3, so *ptree* must contain this spanningtree in all states following s. Since any addition of an edge to a spanning tree results in a cycle, *ptree* must remain constant for the remainder of the execution. □

4 Terminating Spanning Tree Algorithm

4.1 Overview

This algorithm takes in a connected, undirected graph with a specified root. The algorithm will produce a spanning tree of the graph with the specified root, and then the root will output a completed spanning tree.

A *Node* automaton is created for each node in the input graph. Each *Node* automaton is passed the specifications for the graph data type. In addition, each node knows its neighbors and whether or not it is the root of the graph. We use the same *Channel* automaton as in the previous algorithm to connect *Node* automata that have edges between them, one in each direction. The message type passed to *Channel* is a union of the edge type and two special messages. *wakeup* messages are used when the spanning tree is being built to search for children. *end* messages are used when the edge sets of each subtree are being passed back to tell the root of the subtree that it has received all its edges.

When a node receives a message via it adds the message to its queue of incoming messages, *inmessages*. There are three separate procedures for processing received messages, one for each message type. Each of these is enabled when the appropriate message type is at the head of *inmessages* and removes the message when it processes it.

The algorithm begins with the root node being initialized. When initialized, *wakeup* messages for all the neighbors of root are added to the root's message queue, *outgoing*. Every other node has an empty queue until it receives a *wakeup* message. When a node receives a *wakeup* message it does several things. If the receiving node is not root and the *wakeup* message is the first one received, the receiving node makes the sending node its parent. It also adds a wakeup message for every neighbor but its parent to *outgoing*. Regardless of whether or not the *wakeup* is the first one received, the receiving node also adds the sending node to its set of nodes it knows not to be children, *notchildren*. This is because each node sends *wakeup* messages to all of its neighbors but its parent.

Eventually, one node will become a leaf of the tree. It will know that it is a leaf because it will receive *wakeup* messages from all its neighbors. When this happens it will report back to its parent that it is a leaf. It does this by sending an edge connecting itself to its parent, and then an end message. The leaf will then output the subtree it is the root of, a single node tree.

In general, each node waits until all its neighbors have told it they are either not its child or have passed back their entire subtrees. When a child is done passing its subtree the child sends an *end* message. When the parent processes the *end* message, it adds the child to its set of the node's children who have finished sending messages, *children*. When every neighbor is in either *children* or *notchildren*, the node is ready to begin constructing its subtree and passing back the edges of its subtree to its parent. The internal operation $tell(e, i)$ moves edges from the ingoing queue, a set of the edges the node has received, and creates outgoing messages to the parent node. When *ingoing* is empty (and every child has reported), the procedure *donetelling* adds two more messages to *outgoing*. One is a message consisting of the edge from the node to its parent. The second is an *end* message.

The procedure buildtree makes a simple rooted graph from the edges in treeset. It does this by starting with a rooted graph consisting only of itself. It then picks an edge that has one of its nodes already in the subtree graph, and applies connectnode, a procedure in GraphWithRoot.lsl, to add the edge. The edge must already have one node in the subtree due to a property of the GraphWithRoot data type's specification. It is asserted that a GraphWithRoot is a SimpleConnectedGraph. Connectnode, a constructor for SimpleConnectedGraphs, must thus always return a simple connected graph. Thus, if connectnode tries to add an edge that does not have one of its endpoints in the graph already it will just return the original graph. As a result, the edge would be removed from treeset without it being added to the graph. When all the edges have been removed from treeset and added to the tree, *complete* will output the nodes subtree. In the case of the root, this will be the whole spanning tree.

4.2 Type Declaration

```
%See comments in introduction

type NullType = enumeration of null
type ParentType = union of node : NodeType, n : NullType
type SpecialMessage = enumeration of wakeup, end
type FullMessageType = union of special : SpecialMessage,
 edge : EdgeType
type NodeAndMessage = tuple of node : NodeType, message : FullMessageType
```

4.3 Channel Automaton

```
automaton Channel(IndexType : Type, i, j : IndexType, MessageType : Type)

  signature
    input send(const i, const j, message : MessageType)
    output receive(const i, const j, message : MessageType)

  states
    queue : Seq[MessageType] := {}

  transitions

    input send(i, j, message)
      eff queue := queue |- message

    output receive(i, j, message)
      pre queue ~= {} /\ head(queue) = message
```

```
    eff queue := tail(queue)
```

4.4 Node Automaton

```
automaton Node(i: NodeType, neighbors : Set[NodeType], isroot : Bool)

  assumes GraphWithRoot(NodeType, EdgeType, GraphType)

  signature

    input
      receive(j : NodeType, const i, message : FullMessageType)
       where j \in neighbors
    output
      send(const i, j : NodeType, message : FullMessageType),
      finished(complete : GraphType, const i)
    internal
      receivewakeup(const i, j : NodeType),
      receiveend(const i, j : NodeType),
      receiveedge(const i),
      initialize(const i),
      tell(edge : EdgeType, const i),
      donetelling(const i),
      buildtree(edge : EdgeType, const i)

  states
    parent : ParentType := n(null),
    notchildren : Set[NodeType] := {}, %neighbors known to not be children
    children : Set[NodeType] := {}, %children done sending edges
    treeset : Set[EdgeType], %edges to be added to this nodes subtree
    tree : GraphType := graphwithroot(i), %permanent copy of node's subtree
    reported : Bool := false, %true when finished sending subtree
    done : Bool := false, %true when a complete tree is outputted
    ready : Bool := true, %controls initialization of root
    inmessages : Seq[NodeAndMessage] := {},
    inedges : Set[EdgeType] := {}, %stores edges while accumulating
    outmessages: Seq[NodeAndMessage] := {} %queue for send

  transitions

    input receive(j, i, message)
      eff inmessages := inmessages |- [j, message]

    internal receivewakeup(i, j)
      pre [j, special(wakeup)] = head(inmessages)
      eff
        if ~isroot /\ parent = n(null) then
```

```
    parent := node(j);
    for k : NodeType in delete(parent.node, neighbors) do
      outmessages := outmessages |- [k, special(wakeup)] od fi;
   notchildren := insert(j, notchildren);
   inmessages := tail(inmessages)

internal receiveend(i, j)
  pre [j, special(end)] = head(inmessages)
  eff
    children := insert(j, children);
    inmessages := tail(inmessages)

internal receiveedge(i)
  pre tag(head(inmessages).message) = edge
  eff
    inedges := insert(head(inmessages).message.edge, inedges);
    treeset := insert(head(inmessages).message.edge, treeset);
    inmessages := tail(inmessages)

output send(i, j, message)
  pre outmessages ~= {} /\ head(outmessages) = [j, message]
  eff outmessages := tail(outmessages)

output finished(complete, i)
  pre ~done /\ children \U notchildren = neighbors
   /\ complete = tree /\ treeset = {}
  eff done := true

internal tell(edge, i)
  pre children \U notchildren = neighbors /\ edge \in inedges
   /\ ~isroot
  eff
    inedges := delete(edge, inedges);
    outmessages := outmessages |- [parent.node, edge(edge)]

internal donetelling(i)
  pre outmessages = {} /\ ~reported /\
   children \U notchildren = neighbors /\ ~isroot
  eff
    outmessages := outmessages
     |- [parent.node, edge(makeedge(parent.node, i))];
    outmessages := outmessages |- [parent.node, special(end)];
    reported := true

internal initialize(i)
  pre isroot /\ ready
```

```
    eff
      for k : NodeType in neighbors do
        outmessages := outmessages |- [k, special(wakeup)] od;
        ready := false

  internal buildtree(edge, i)
    pre edge \in treeset /\ source(edge) \in nodes(tree.g)
    eff
      tree := graphwithroot(tree.root, connectnode(target(edge),
       source(edge), tree.g));
      treeset := delete(edge, treeset)
```

4.5 The Complete System

```
automaton SpanningTree(GraphType : Type, graph: GraphType)

  assumes GraphWithRoot(NodeType, EdgeType, GraphType)

  compose
    Node(i, neighbors(i, graph.g),false)
      for i : NodeType where i \in (nodes(graph.g) - {graph.root});

    Node(graph.root,
     neighbors(graph.root, graph.g), true);

    Channel(NodeType, i, j, FullMessageType)
      for i, j : NodeType where \E e : EdgeType (e \in edges(graph.g) /\
       (source(e) = i /\ target(e) = j))
```

5 Unsynchronized Distributed Bellman-Ford Algorithm

5.1 Overview

Our next algorithm adds the issue of weight. We now want an algorithm to find the spanning tree such that the distance from each node to the root along the tree is minimal.

This algorithm takes in a graph of type WeightedGraphWithRoot such that the weight type is Real. Each node is told its neighbors and the weights of the edges connected to it. Every node except root starts with its *pathlength* and *parent* set to null.

The algorithm begins with the root adding a message to each of its neighbors to its *queue* consisting of length 0. This message corresponds to the fact that the path from the root to itself has length 0. In general, each message from node i will consist of the best known distance from root to i.

When a node j receives a message from a node i it sees if the path created by taking the path from root to i and then going from i to j is smaller than its current pathlength. It does this by adding the pathlength given to it in the message to the weight of the edge from i to j, which is part of $j's$ specification. If this length is smaller, or j currently has no parent, it makes i its parent and sets its pathlength to the sum of the pathlength it was sent and the weight of the edge between i and j. Then it adds a message announcing that it has a new minimal pathlength to each of its neighbors.

A synchronous version of this algorithm in which all the nodes pass their pathlength information at the same time will produce the minimum spanning tree in n-1 rounds of each node sending messages to all its neighbors where n is the number of nodes. Thus, the total number of messages required is proportional to the number of nodes times the number of edges. This can be seen by proving that after the kth round any minimal path to a node traversing k or fewer nodes will be found by induction. The asynchronous version of the algorithm does not come close to this efficiency and actually make take time on order of $2^{\frac{n}{2}}$. It can, however, be shown that every node will eventually know the minimal pathlength to it and its parent along this path. No node will know this has happened, but messages will cease to be passed from node to node.

5.2 Type Declaration

```
%See comments in introduction
type NullType = enumeration of null
type ParentType = union of n : NullType, node : NodeType
type PathLengthType = union of n : NullType, length : Real
```

5.3 Channel Automaton

```
automaton Channel(Index : Type, i, j : Index, MessageType : Type)

  signature
    input send(const i, const j, message : MessageType)
    output receive(const i, const j, message : MessageType)

  states
    queue : Seq[MessageType] := {}

  transitions

    input send(i, j, message)
      eff queue := queue |- message

    output receive(i, j, message)
      pre head(queue) = message
      eff queue := tail(queue)
```

5.4 Node Automaton

```
automaton Node(i : NodeType, neighbors : Set[NodeType],
 weight : Array[NodeType, Real], isroot : Bool)

  signature
    input receive(j : NodeType, const i : NodeType, message : Real)
    output send(const i, j : NodeType, message : Real)
    internal initialize(const i)

  states
    parent : ParentType := n(null),
    pathlength : PathLengthType := n(null),
    queue : Array[NodeType, Seq[Real]] := constant({}),
    initialized : Bool := false

  transitions
    input receive(j, i, message)
      eff
        if (tag(pathlength) = n) \/
         ((message + weight[j]) < pathlength.length) then
          pathlength := length(message + weight[j]);
          if ~isroot
            then parent := node(j) fi;
          for k : NodeType in delete(parent.node, neighbors) do
            queue[k] := queue[k] |- pathlength.length od fi

    output send(i, j, message)
      pre queue[j] ~= {} /\ head(queue[j]) = message
      eff queue[j] := tail(queue[j])

    internal initialize(i)
      pre isroot /\ ~initialized
      eff pathlength := length(0);
          for j : NodeType in neighbors do
            queue[j] := queue[j] |- pathlength.length od;
          initialized := true
```

5.5 The Complete System

```
automaton BellmanFord(EdgeType : Type,
 GraphType : Type, graph : GraphType)

  assumes WeightedGraphWithRoot(NodeType, EdgeType, GraphType, Real)

  compose
    Node(i, neighbors(i, graph.g), weights, false)
```

```
  for i : NodeType, weights : Array[NodeType, Real]
   where i \in delete(graph.root, nodes(graph.g)) /\ \A e : EdgeType
   (\A j : NodeType ((e \in edges(graph.g)
   /\ target(e) = i /\ source(e) = j) => weights[j] = weight(e)));

Node(graph.root, neighbors(graph.root, graph.g), weights,
 true)
  for weights : Array[NodeType, Real] where \A e : EdgeType
   \A j : NodeType (((e \in edges(graph.g) /\ target(e) = graph.root
   /\ source(e) = j) => weights[j] = weight(e)));

Channel(NodeType, i, j, Real)
  for i, j : NodeType where
   \E e : EdgeType (e \in edges(graph.g) /\ source(e) = i
   /\ target(e) = j)
```

6 Synchronized Version of the Bellman-Ford Algorithm

6.1 Overview

The previous algorithm is inefficient in that its running time grows exponentially while a version of the algorithm in a synchronized model only takes time proportional to the number of nodes times the number of edges. The goal is then to force the algorithm to behave as it did in the synchronous case, running in a series of rounds. All the messages in a given round cannot be made to occur simultaneously. It is, however, sufficient to have each node not progress to the next round until it and all its neighbors have performed all actions from the previous round. We can achieve this by having nodes send messages to their neighbors stating when they are done with a round of messages.

Each node begins by initializing itself. When it does so, it sets its current round to 1 and makes itself active. If the node is root it also sets its pathlength to 0 and adds a message to each neighbor containing a pathlength of 0.

During each round two things happen at a given node. If a node's pathlength has changed since the last round it sends a message to each of its neighbors, stating what its pathlength is. In addition, each node receives pathlengths from each of its neighbors. As before, if i receives a pathlength from j such that the pathlength plus the weight of the edge from j to i is less than $i's$ current pathlength to root, i replaces its old pathlength. It does not, however, add more messages to its queue until the next round.

When a node has sent all the pathlength messages it will for the round (either one to each neighbor or none if its pathlength has not changed since last round), a $finishround$ action is enabled. The node sends an ok message to each of its neighbors saying that it is ready to move on to the next round. It also increments its own readiness to the next round. When a node has set its own readiness to k and has received an $ok(k)$ from each of its neighbors, $nextround$ is enabled. A $nextround(i, k)$, begins round k of messages in i adding a pathlength message to each of its neighbors if i has a non-null pathlength.

After i finishes round $n-1$ where n is the number of nodes it sets $done$ to true, indicating that it has sent all the messages it is going to. It sends an $ok(n)$ to each of its neighbors. When

i has received an $ok(n)$ from all its neighbors a $report(i, parent, pathlength)$ occurs. A *report* announces to the external environment that the node has completed determining its parent and pathlength. Alternately, we could have the nodes report their subtree of the spanning tree back to their parent as we did in the second algorithm. This would result in root having a complete copy of the spanning tree.

6.2 Type Declaration

```
type NullType = enumeration of null
type ParentType = union of n : NullType, node : NodeType
type PathLengthType = union of n : NullType, length : Real
type MType = union of w : Real, ok : Int, n : NullType
```

6.3 Channel Automaton

```
automaton Channel(Index : Type, i, j : Index, MessageType : Type)

  signature
    input send(const i, const j, message : MessageType)
    output receive(const i, const j, message : MessageType)

  states
    queue : Seq[MessageType] := {}

  transitions
    input send(i, j, message)
      eff queue := queue |- message
    output receive(i, j, message)
      pre queue ~= {} /\ head(queue) = message
      eff queue := tail(queue)
```

6.4 Node Automaton

```
automaton Node (i : NodeType, neighbors : Set[NodeType], weights :
 Array[NodeType, Real], isroot : Bool, nodecount : Int)

  signature
    input
      receive(j : NodeType, const i, message : MType)
    output
      send(const i, j : NodeType, message : MType),
      report(const i, parent : ParentType, pathlength : PathLengthType)
    internal
      initialize(const i),
      nextround(const i, newround : Int),
      finishround(const i)
```

```
states
  round : Int := 0,
  readiness : Array[NodeType, Int] := constant(1),
   %the round i's neighbors are ready to start
  parent : ParentType := n(null),
  pathlength : PathLengthType := n(null),
   %the minimum pathlength from root to i currently found
  outgoing : Array[NodeType, MType] := constant(n(null)),
   %an array of the messages to each of i's neighbors if a message exists
  active : Bool := false,
   %true if i is currently sending messages in a round
  initialized : Bool := false,
   %true if i has performed an initialization action
  done : Bool := false,
   %true if i has completed its last round of messages
  reported : Bool := false,
   %true if i has announced it is done sending messages
  changed : Bool := false
   %true if the value of pathlength was changed in the previous round

transitions

  input receive(j, i, message)
    eff
      if tag(message) = ok then
        readiness[j] := message.ok
      else
        if tag(parent) = n \/
          ((weights[j] + message.w) < pathlength.length) then
          parent := node(i);
          pathlength := length(weights[j] + message.w);
          changed := true fi fi

  output send(i, j, message)
    pre outgoing[j] = message
    eff outgoing[j] := n(null)

  internal initialize(i)
    pre ~initialized
    eff
      round := 1;
      initialized := true;
      active := true;
      if isroot then
        pathlength := length(0);
```

```
        for j : NodeType in neighbors do
          outgoing[j] := w(0) od fi

  internal nextround(i, newround)
    pre outgoing = constant(n(null)) /\ \A j : NodeType (j \in neighbors =>
     newround = readiness[j]) /\ readiness[i] = newround
    eff
      round := newround;
      active := true;
      if changed then
        for j : NodeType in neighbors do
          outgoing[j] := w(pathlength.length) od fi;
      changed := false

  internal finishround(i)
    pre outgoing = constant(n(null)) /\ active
    eff
      if round = nodecount - 1 then
        done := true
      else
        readiness[i] := readiness[i] + 1 fi;
      for j : NodeType in neighbors do
        outgoing[j] := ok(round + 1) od;
      active := false

  output report(i, parent, pathlength)
    pre done /\ ~reported /\
     \A j : NodeType (j \in neighbors => readiness[j] = nodecount)
    eff
      reported := true
```

6.5 The Whole System

```
automaton SynchBellmanford(EdgeType : Type, GraphType : Type,
 graph : GraphType)

  assumes WeightedGraphWithRoot(NodeType, EdgeType, GraphType, Real)

  compose
    Node(i, neighbors(i, graph.g), weights, false, size(nodes(graph.g)))
      for i : NodeType, weights : Array[NodeType, Real]
       where i \in (nodes(graph.g) - {graph.root}) /\ \A e : EdgeType
       (\A j : NodeType
       ((e \in edges(graph.g) /\ target(e) = i /\ source(e) = j) /\
       weights[j] = weight(e)));
```

```
Node(graph.root, neighbors(graph.root, graph.g),
 weights, true, size(nodes(graph.g)))
  for weights : Array[NodeType, Real] where
   \A e : EdgeType \A j : NodeType (((e \in edges(graph.g) /\
   target(e) = graph.root /\ source(e) = j) /\
   weights[j] = weight(e)));

Channel(NodeType, i, j, MType)
  for i, j : NodeType where \E e : EdgeType
   (e \in edges(graph.g) /\ source(e) = i /\ target(e) = j)
```