Spanning Tree

Some facts about spanning trees in undirected graphs:

Acyclic
Number of edges is 1 less than the number of nodes
Connected
Root has no parent
Any connected subgraph of a graph with (number of nodes)-1 edges is a tree

Applications

Broadcast
Convergecast
Build a loop-free topology for bridges
Min-cost communication (e.g., min latency/energy in a cycle-free structure)

BFS Spanning Tree

The tree is defined by the parent pointers; it will only be a BFS tree if the network is synchronous. Otherwise, it can be any tree rooted at the start node.

Min-weight spanning tree

Minimize total weight of all edges in the tree. We will show a synchronous algorithm first, then a much more complex asynchronous algorithm.

Assume: G=(V,E) undirected; weights on edges are known by adjacent processes; processes have UID's; n (size of graph) is known

Each node will decide which of its adjacent edges is or is not in the tree.

Theory

A spanning tree of G is a tree connecting all nodes, with edges selected from G. A spanning forest is a collection of trees spanning all the nodes, with edges selected from G. All MST algorithms are special cases of a general strategy.

1) Start with trivial spanning forest of n separate nodes.
2) Merge components along edges that connect components until all are connected (but no cycles).

The trick is to make sure that we merge only along edges that are minimum weight outgoing edges of some component.

Justification:
Lemma 4.3. (p 64) Let (V_i, E_i) $1 \le i \le k$ be a spanning forest, with $k>1$
Fix any i. Let e be an edge of smallest weight among the set of edges with exactly one endpoint in V_i. Then there is a spanning tree for G that

1) Includes $\cup_j E_j$
2) Includes e

3) Has min weight among all spanning trees that include $\cup_j E_j$

Proof:
Suppose this is false, then there is some T that is a spanning tree for G and includes $\cup_j E_j$, doesn't contain e, and has weight strictly less than the weight of any spanning tree that includes both $\cup_j E_j$ and e.

We can construct T' by adding e to T. This contains a cycle, which must contain another edge outgoing from the same component (V_i,E_i). The weight of e' must be greater than or equal to the weight of e. Remove e' from T'. The result is a spanning tree that contains both $\cup_j E_j$ and e and has weight no greater than T'. This contradicts the choice of T.

Strategy for MST

See page 65. Repeatedly:
Choose component i
Choose any least-weight outgoing edge from i and add (merging two components)

Sequential MST are special cases:
Prim-Dijkstra adds 1 more node on each iteration
Kruskal adds min weight edge globally

Distributed version: We want to choose edges concurrently for multiple components – but (if multiple edges have the same weight) this can produce cycles – example 4.4.1. Assume all edge weights are distinct (we can get the same effect by breaking ties with UID's.

Lemma 4.4. If all edge weights are distinct, then there is a unique MST.

The concurrent strategy:
At each stage, suppose (inductively) that the forest produced so far is part of the unique MST. Each component chooses a least-cost outgoing edge. Each of these is in the (unique) MST, by Lemma 4.3. So, all are ok – add them all.

The synchronous algorithm

Main idea: The algorithm proceeds by *levels*. It starts at level 0, with each component containing 1 node and no edges. At level 1, nodes connect along their minimum-weight outgoing edge (MWOE) to their nearest neighbor. Each component now has at least 2 nodes. At level k, with each component having at least 2^k nodes, each component selects an edge to connect it to its nearest neighbor, so that at level k+1 each component has at least 2^{k+1} nodes.
Finding the MWOE: To make this work, we also need a way to identify the MWOE of a component. The algorithm requires that a leader has been chosen for each component. The leader's UID serves as the component ID. We'll discuss the choice of leader shortly. This leader broadcasts a search request (using BFS) for the MWOE. Each node

determines its own MWOE by sending a *test* message along each outgoing edge, asking whether the neighbor is in the same component (using the UID of the leader). Among those edges not leading to the same component, the node chooses the MWOE and responds with the edge and its weight in a convergecast to the leader. Then the leader can pick the overall MWOE.
Combining components: Once the leaders have picked MWOE's, they tell the processes adjacent to the MWOE's to mark them as selected as part of the spanning tree.
Leader selection: To select a new leader, note that there is a unique edge e that is the common MWOE of two of the level k components in the group. Let the new leader be the endpoint of e having the larger UID. Note the leader election is efficient, because the leader can identify himself by asking his neighbor along the MWOE if that edge is also the neighbor's MWOE.
Propagating the leader id: Use BFS to propagate the leader ID to all components.
Synchronizing component levels: Note that nodes must be at the same level before they can determine accurately whether they are in the same component; for this reason, a pre-determined number of rounds is used for each level, based on the size of the network. This is why the size of the network must be known.
Termination: When the *search* request fails to find any outgoing edges, the algorithm is complete and the leader can use BFS to inform the other nodes.

Adaptation for asynchronous case

Difficulties (p 511)
1) Since we can't run by rounds to keep nodes at the same level, determination of whether two nodes are in the same component may be done incorrectly.
2) Inefficiencies may arise due to unbalanced combining of components
3) Components at higher levels might interfere with searches for MWOE by components at lower levels (how?)

Strategy
Maintain a number at each node representing the level of its component.
Define two combining operations, *merge* and *absorb*.
merge is for components at the same level that have a common MWOE (the new leader and component ID are chosen accordingly).
absorb is for components at different levels in which the lower level component has a MWOE leading to the higher level component. Its level is then raised – this is not a new component.

This strategy guarantees that the components at level k have at least 2^k nodes. Also, any sequence of legal merges and absorbs leads eventually to a single component, ie, a spanning tree.

Cisco Spanning Tree (by Radia Perlman)

Not a min-weight spanning tree, but a shortest-distance-to-root spanning tree.

Algorhyme

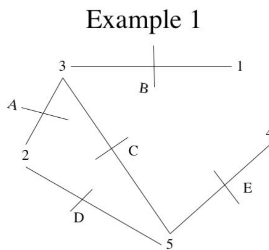
Radia Perlman, the inventor of the algorithm, summarized it in a poem titled "Algorhyme"_(adapted from

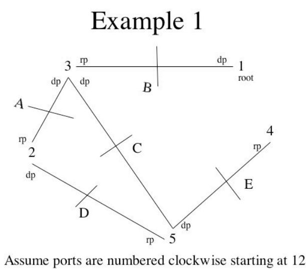
"Trees", by Joyce Kilmer):

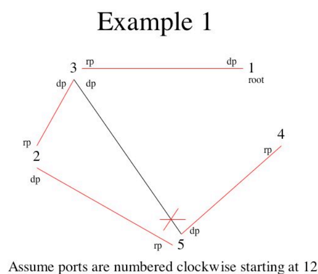
I think that I shall never see
A graph more lovely than a tree.
A tree whose crucial property
Is loop-free connectivity.
A tree which must be sure to span
So packets can reach every LAN.
First the Root must be selected
By ID it is elected.
Least cost paths from Root are traced
In the tree these paths are placed.
A mesh is made by folks like me
Then bridges find a spanning tree.

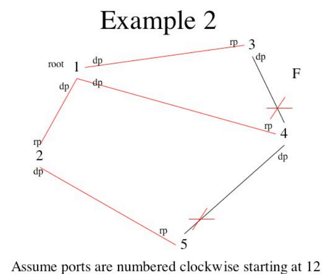
Note that BFS computes a spanning tree (the parent pointers identify the edges), once a root has been selected. How many edges are there in a spanning tree?

Example









How does the Cisco STP work

When the protocol stabilizes, the state should be as follows:

1) **Root bridge**: The process (switch) with the lowest MAC address (or lowest combined priority+MAC address) is the root. This uses a leader election algorithm.

2) **Root ports**: Each bridge has one root port. The root port on each bridge is the port of the bridge with the smallest distance from the root. If two ports are equidistant from the root, then the one going to the bridge with the lower MAC address is the root port. This uses a breadth-first search, if we assume rounds; however, if the network is asyncrhonous, it's more complicated.
3) **Designated ports**: Each network segment (connecting bridges) has a designated port. Messages put on that network segment are forwarded to the rest of the network through the designated port. The designated port is on the bridge closest to the root. If there is a tie, it is on the bridge with the lowest MAC address. If the bridge selected by this rule has multiple ports on a network, it is the port with the lowest id.

Algorithm

All processes send a BPDU (Basic Protocol Data Unit) at each round (actually, default is every 2 seconds – but we will describe this as a synchronous algorithm, running in rounds). The BPDU contains the id (MAC address) of the sending process, the id of the process it thinks is the root, and the distance from the sending process to its presumed root.

id:Int
id:Int
cost:Int

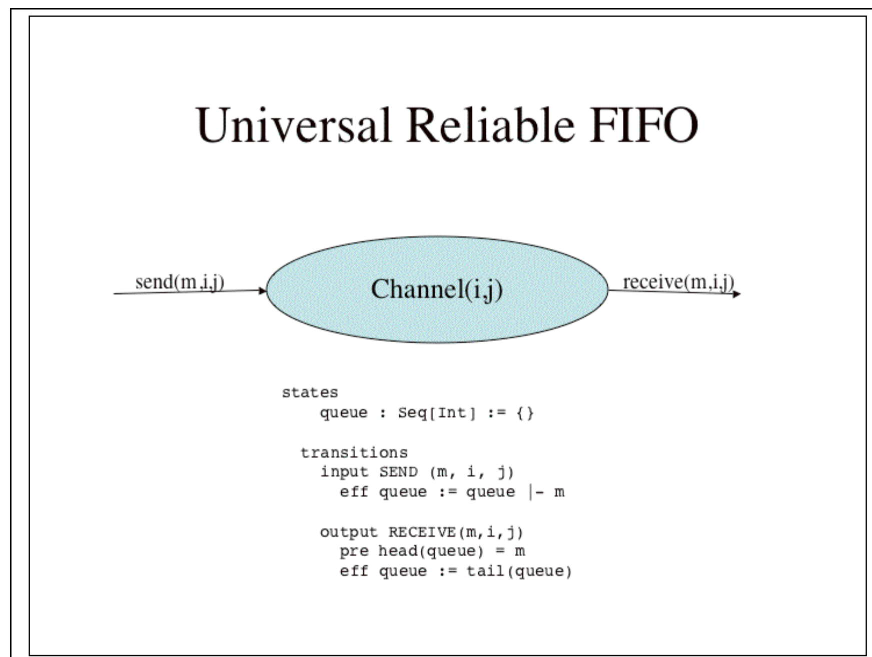
When a process receives a BPDU, it compares the ID of the root (designated by the neighbor that sent the BPDU) to the local value for the ID of the root. If the new root ID is lower, it replaces its local root ID with the new one and adds one to the cost in the incoming BPDU and makes that its cost to the root. If it receives different BPDU's having different roots, it uses the "best," i.e., the one with the lowest root ID and the lowest distance (if root Ids are the same).
Finally it designates the port to which the sending neighbor is connected as the root port. (Effectively choosing its parent).

Designated ports are then chosen: for each network it is on, the bridge checks incoming BPDUs. If its own cost is lowest, or if it is tied with another bridge on cost and its id is smaller, it is the designated bridge, and its port on that network segment is the designated port. If it has multiple ports on the network segment, it chooses the one with the lowest id.

Relationship to BFS:
If we ignore all BPDU's except those carrying the ID of the eventual root, it will look just like BFS. This is the basic idea behind simulation, which I introduce next.

Hierarchical proofs

This is an important strategy for complex algorithms. We formulate the algorithm in a series of levels. For example, we could have a high-level centralized algorithm (easy to prove, almost a specification), then do a simple but inefficient decentralized version, then do an optimized version.
High level version:



Simulation Relations

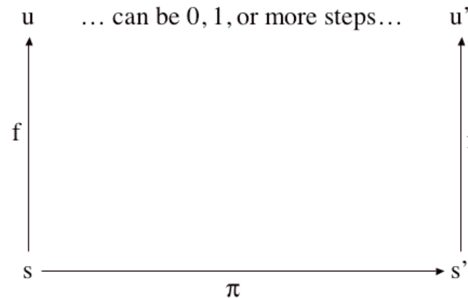
Lower levels are harder to understand, so we relate them to higher levels with a simulation invariant rather than trying to deal with them directly. This is similar to the simulation relation for synchronous algorithms:
Run them side by side.
Define an invariant relating the states.
The invariant is called a *simulation relation* and is usually shown via induction.

Show that for each execution of the lower-level algorithm, there exists a related execution of the higher-level algorithm.

Definition: Assume A and B have the same external signature. Let f be a binary relation on states(A)xstates(B).
Notation: $(s,u) \in f$ or $u \in f(s)$
Then f is a simulation relation from A to B provided that

1. If $s \in start(A)$ then $f(s) \cap start(B) \neq \emptyset$
2. If s,u are reachable states of A and B respectively, with $u \in f(s)$, and if (s, π, s') is a transition of A, then there is an execution fragment α of B starting with u, and ending with $u' \in f(s')$, with $trace(\alpha) = trace(\pi)$.

Simulation Relation



Theorem. If there's a simulation relation from A to B then $traces(A) \subseteq traces(B)$.
Proof: Take any execution of A, and iteratively construct the corresponding execution of B.

Example proof

Example: Implementing the reliable FIFO channel with TCP (sort of)

Altered spec

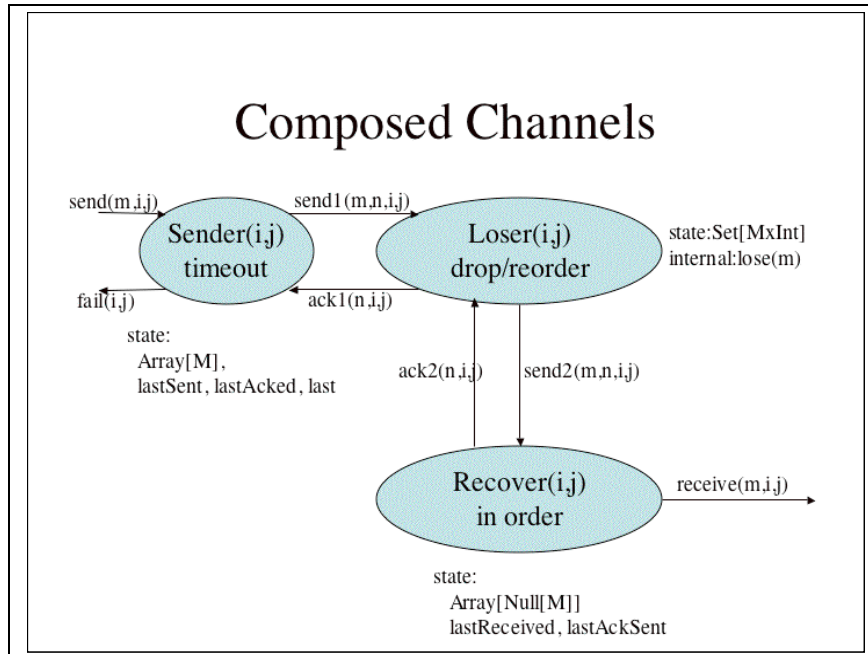
Reliable FIFO assumes that all messages are eventually delivered. When we deal with real networks, we know that cables can break and as a result a channel entirely disappears for a period of time. So the Reliable FIFO spec can't be implemented in a real network; we have to modify it somewhat to address reality.

Add fail(i.j) transition to Reliable FIFO: all messages up to some point get sent in order. At some point there is a fail. No more messages can be sent after some point, but a subset of the messages in the queue can be received.

either there is a fail action or the cause is surjective.

if a send `cause(π)` is in `range(cause)` then so is every earlier send

Implementation
This is an adaptation of TCP.



Automata Descriptions:
`Sender(i,j)` puts new message in `Sender(i,j).Array[last+1]` and increments `last`. It sends messages in `Sender(i,j).Array[lastAcked+1...last]` repeatedly until it receives `ack1(n,i,j)`. Then it sets `lastAcked` to `max(lastAcked, n)`.

`Loser(i,j)` puts messages in `Loser(i,j).Mset` when `send1(m,i,j)` happens. The precondition for `send2(m,i,j)` and for `delete(m,i,j)` is that `m ∈ Loser(i,j).Mset`. The effect of `delete(m,i,j)` is to remove the message from the multi-set, but `send2(m,i,j)` causes no change. This means that `Loser(i,j)` delivers each message to `Recover(i,j)` 0 or more times.

`Recover(i,j)` puts each message in its proper position in `Recover(i,j).Array`.
`Recover(i,j).ack2(n,i,j)` is enabled when there are no holes in `Recover(i,j).Array[1..n-1]` and `n > lastAckSent`. Its effect is to update `lastAckSent` to `n`.
`Recover(i,j).receive(m,i,j)` is enabled if `m = Recover(i,j).Array[lastReceived+1]` and `lastAckSent > lastReceived+1`.
Simulation relation
We are comparing the composition of `Sender`, `Loser`, and `Recover` with all actions hidden except `send(m,i,j)`, `receive(m,i,j)`, and `fail(i,j)`. We need to show that the traces of this composition are contained in the traces of `Channel(i,j)`, modified by the fail action.

The important parts of the state are:

```
Sender(i,j).Array
Recover(i,j).Array
lastReceived
lastAcked
```

Facts:
Just to develop some intuition about what's going on:

`Recover(i,j).Array` is a prefix of `Sender(i,j).Array[1...lastSent]` (with holes)
`Recover(i,j).Array[1...lastReceived]` extends
`Sender(i,j).Array[1...lastAcked]`

`Recover(i,j).Array[1...lastAckSent]` is the messages already delivered to the receiving application plus those that are ready to be delivered

The simulation relation
The correspondence is that `Channel(i,j).Queue` corresponds to any state with all messages in `Channel(i,j).Queue` sent and no messages in `Channel(i,j).Queue` received (yet) – that is,

`Channel(i,j).Queue` corresponds to all states such that there exists `n` with
`Send(i,j).Array[n...last] = Channel(i,j).Queue` and
`n = Recover(i,j).lastReceived+1`

Proof that the above is a simulation relation:
We must show that the relation is preserved by all actions. Consider actions of the composition:
`send` adds a message to `Channel(i,j).Queue` and to `Sender(i,j).Array[last+1]` – this preserves the simulation relation
`send1, send2` don't change any of the relevant parts of the state
`receive` increases `Recover(i,j).lastReceived` and `Recover(i,j).lastAckSent`; also removes an element from `Channel(i,j).Queue` – this preserves the simulation relation
`ack2` and `ack1` have no effect on relevant parts of state
`fail` can happen any time but after `fail`, `Sender` quits doing anything - the only other thing that can happen is that messages get through the system and are received. In `Channel`, no more messages can be sent but some subset of the messages already in the channel can be delivered.