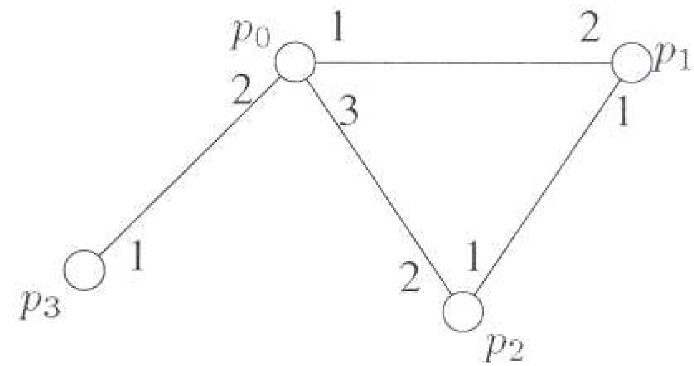
Basic Algorithms

Formal Model of Message-Passing Systems



- There are n processes in the system: p_0, .., p_{n-1}
- Each process is modeled as a state machine.
- The state of each process is comprised by its local variables and a set of arrays. For instance, for p_0, the state includes six arrays:
  - $inbuf_0[1]$, …, $inbuf_0[3]$: contain messages that have been sent to p_0 by p_1, p_2 and p_3, respectively, but p_0 has not yet processed.
  - $outbuf_0[1]$, …, $outbuf_0[3]$: messages that have been sent by p_0 to p_1, p_2, and p_3, respectively, but have not yet been delivered to them.

Formal Model of Message-Passing Systems

- The state of process p_i excluding the $outbuf_i[l]$ components, comprises the accessible state of p_i.
- Each process has an initial state in which all inbuf arrays are empty.
- At each step of a process, all messages stored in the inbuf arrays of the process are processed, the state of the process changes and a message to each other neighboring process can be sent.
- A configuration is a vector $C = (q_0, .., q_{n-1})$ where q_i represents the state of p_i.
  - The states of the outbuf variables in a configuration represent the messages that are in transit on the communication channels.
  - In an initial configuration all processes are in initial states.

Formal Model of Message-Passing Systems

- **Computation event, comp(i)**
  - Represents a computation step of process p_i in which p_i's transition function is applied to its current accessible state.
- **Delivery Event, del(i,j,m)**
  - Represents the delivery of message m from processor p_i to processor p_j (i.e., message m is placed in one of the inbuf buffers of p_j)
- The behavior of a system over time is modeled as an **execution**, which is a sequence of configurations alternating with events.
- This sequence must satisfy a variety of conditions.
  - Safety condition
    - Holds in every finite prefix of the execution (it states that nothing bad has happened yet)
  - Liveness condition
    - Holds a certain number of times (it states that eventually something good must happen)

Formal Model of Message-Passing Systems Complexity Measures

- The message complexity of an algorithm for either a synchronous or an asynchronous message-passing system is the maximum, over all executions of the algorithm, of the total number of messages sent.
- The time complexity of an algorithm for a *synchronous message-passing system* is the maximum number of rounds, in any execution of the algorithm, until the algorithm has terminated.

Formal Model of Message-Passing Systems Complexity Measures

Measuring the time complexity of asynchronous algorithms

- A timed execution is an execution that has a nonnegative real number associated with each event, the time at which that event occurs.
- The times must start at 0, must be strictly increasing for each individual processor, and must increase without bound if the execution is infinite.
- We define the delay of a message to be the time that elapses between the computation event that sends the message and the computation event that processes the message.
- **Assumption**: The maximum message delay in any execution is one unit of time.
- The time complexity of an *asynchronous algorithm* is the maximum time until termination until termination among all timed executions of the algorithm in which every message delay is at most one time unit.

Broadcast on a Spanning Tree

- A distinguished processor, p_r, has a message <M> it wishes to send to all other processors.
- Copies of the message are to be sent along a tree which is rooted at p_r, and spans all the processors in the network.
- The spanning tree is maintained in a distributed fashion:
  - Each processor has a distinguished channel that leads to its parent, as well as a set of channels that lead to its children.

Algorithm 1 Spanning tree broadcast algorithm.

Initially $\langle \mathsf{M} \rangle$ is in transit from p_r to all its children in the spanning tree.

Code for p_r:

upon receiving no message: // first computation event by p_r
    terminate

Code for p_i, $0 \leq i \leq n-1$, $i \neq r$:

upon receiving $\langle \mathsf{M} \rangle$ from parent:
    send $\langle \mathsf{M} \rangle$ to all children
    terminate

Broadcast on a Spanning Tree

State of process p_i, $i \in \{0, \ldots, n-1\}$

- a variable $parent_i$,which holds either a processor index or nil
- a variable $children_i$,which holds a set of processor indices
- a variable $terminated_i$,which indicates whether p_i is in a terminated state
- the inbuf and outbuf tables of p_i

Initial State

- all terminated variables are false.
- The inbuf tables are empty, for all processes.
- The outbuf tables are empty for all processes other than p_r; $outbuf_r[j]$ contains M for all $j \in children_r$.

Complexities?

- Communication Complexity?
- Time Complexity?

Broadcast on a spanning tree - Time Complexity

Synchronous System

- **Lemma**: In every execution of the broadcast algorithm in the synchronous model, every process at distance t from p_r in the spanning tree receives <M> in round t.
- **Proof**: By induction on the distance t of a process from p_r.
- t = 1. Each child of p_r receives <M> from p_r in the first round.
- Assume that every process at distance $t-1 \geq 1$ from p_r receives the message <M> in round t-1.
- Let p be any process in distance t from p_r. Let p' be the parent of p in the spanning tree. Since p' is at distance t-1 from p_r, by the induction hypothesis, p' receives <M> in round t-1. By the description of the algorithm, p receives <M> from p' in the next round.

Broadcast on a spanning tree - Time Complexity

Asynchronous System

- **Lemma**: In every execution of the broadcast algorithm in an asynchronous model, every process at distance t from p_r in the spanning tree receives <M> in time t.
- **Proof**: By induction on the distance t of a process from p_r.
- t = 1. From the description of the algorithm, <M> is initially in transit to each process p_i at distance 1 from p_r. By the definition of time complexity for the asynchronous model, p_i receives <M> by time 1.
- Assume that every process at distance $t-1 \geq 1$ from p_r receives the message <M> by time t-1.
- Let p be any process in distance t from p_r. Let p' be the parent of p in the spanning tree. Since p' is at distance t-1 from p_r, by the induction hypothesis, p' receives <M> by time t-1. By the description of the algorithm, p receives <M> from p' by time t.

Broadcast on a spanning tree

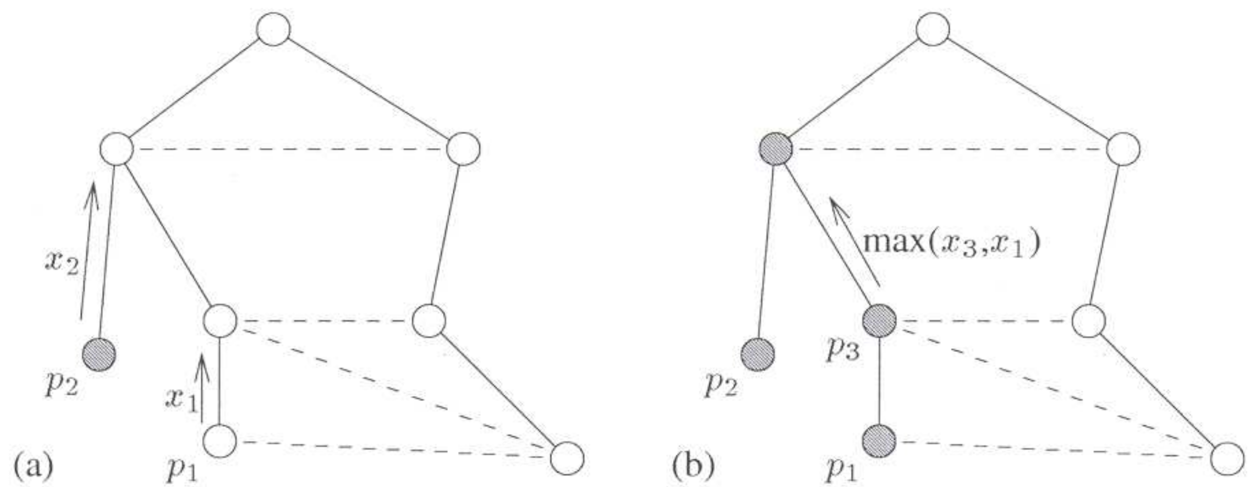
- **Theorem 1:** There is a synchronous broadcast algorithm with message complexity n-1 and time complexity d, when a rooted spanning tree with depth d is known in advance.

- **Theorem 2:** There is an asynchronous broadcast algorithm with message complexity n-1 and time complexity d, when a rooted spanning tree with depth d is known in advance.

Convergecast

Problem

- Collect information from the nodes of the tree to the root.
- Each processor p_i starts with a value x_i.
- We wish to forward the maximum value among these values to the root p_r.



- **Theorem:** There is an asynchronous convergecast algorithm with message complexity n-1 and time complexity d, when a rooted spanning tree with depth d is known in advance.

Flooding and Building a Spanning Tree

Problem

- Broadcast without a preexisting spanning tree, starting from a distinguished processor p_r.

Solution

- **Flooding**
- Assume that m is the number of edges and n is the number of processes. How many messages does the flooding algorithm send?
- Can we modify the flooding algorithm to construct a spanning tree?

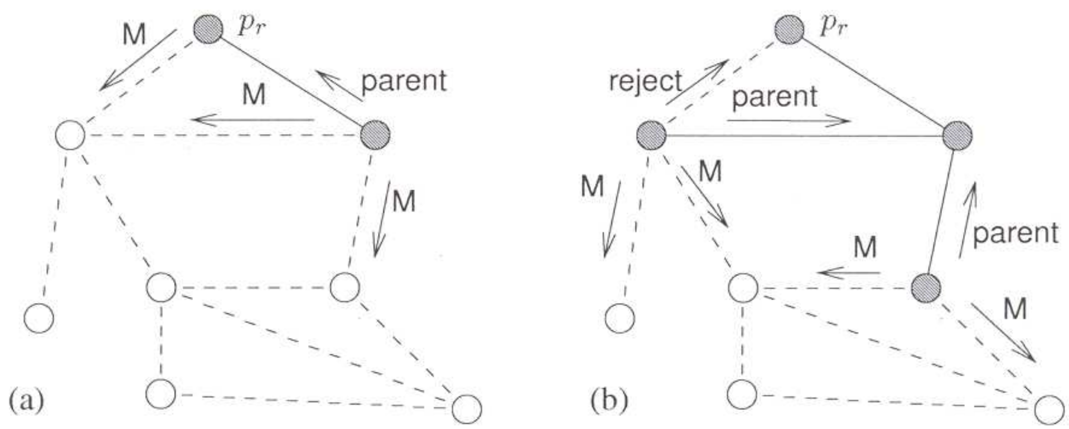
Flooding and Building a Spanning Tree: The F-SpanningTree Algorithm

Modified flooding algorithm to construct a spanning tree: code for processor p_i, $0 \leq i \leq n-1$.

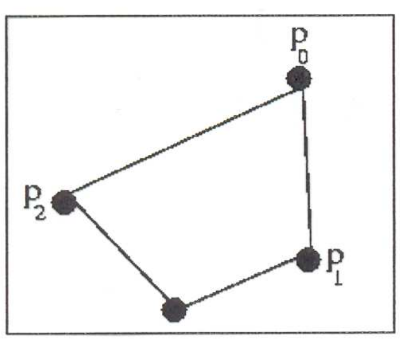
Initially $parent = \perp$, $children = \emptyset$, and $other = \emptyset$.

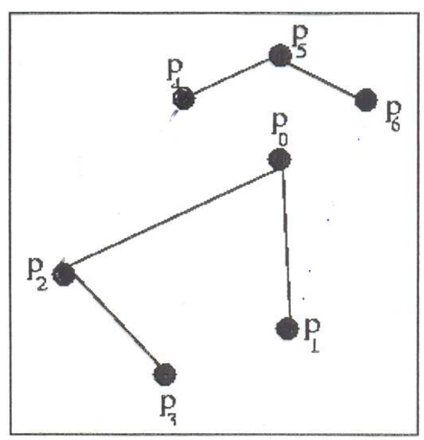
```
upon receiving no message:
    if p_i = p_r and parent = ⊥ then
        send ⟨M⟩ to all neighbors
        parent := p_i

upon receiving ⟨M⟩ from neighbor p_j:
    if parent = ⊥ then
        parent := p_j
        send ⟨parent⟩ to p_j
        send ⟨M⟩ to all neighbors except p_j
    else send ⟨already⟩ to p_j

upon receiving ⟨parent⟩ from neighbor p_j:
    add p_j to children
    if children ∪ other contains all neighbors except parent then
        terminate

upon receiving ⟨already⟩ from neighbor p_j:
    add p_j to other
    if children ∪ other contains all neighbors except parent then
        terminate
```

The F-SpanningTree Algorithm



Two steps in the construction of the spanning tree.





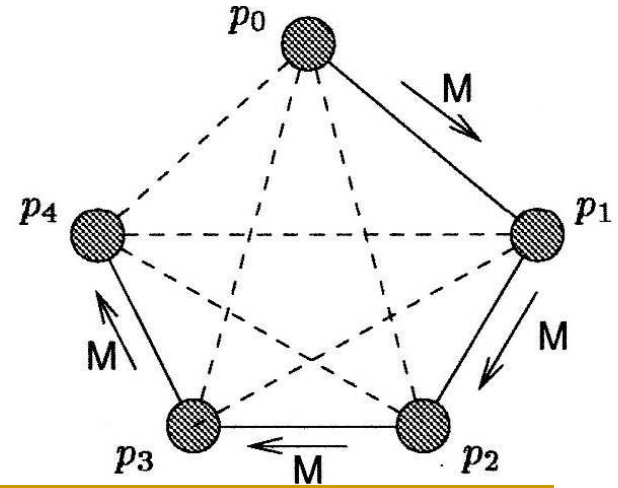
Correctness

Why is every node reachable from the root?

Why is there no cycle?

The F-SpanningTree Algorithm

- **Theorem:** There is an asynchronous algorithm to find a spanning tree of a network with m edges and diameter D, given a distinguished node, with message complexity O(m) and time complexity O(D).
- What kind of tree is the output of F-SpanningTree when the system is synchronous?
- **Theorem:** In every execution of F-SpanningTree in the synchronous model, the algorithm constructs a BFS tree rooted at p_r.
- What kind of tree can be the output of F-SpanningTree when the system is asynchronous?



Synchronous Systems

- We define a directed spanning tree of a directed graph G = (V,E) to be a rooted tree that consists entirely of directed edges in E, all edges directed from parents to children in the tree, and that contains every vertex of G.
- A directed spanning tree of G with root node p_r is breadth-first provided that each node at distance d from p_r in G appears at depth d in the tree (that is at distance d from p_r in the tree).
- ✓ Every strongly connected digraph has a breadth-first directed spanning tree.

- Given that the G is a strongly connected directed graph and given that we have a distinguished node p_r, how can we design a synchronous algorithm that computes the directed BFS tree?
- How can a process learn which nodes are its children?
- What is the communication complexity of the algorithm in this case?
- What is the time complexity of the algorithm in this case?
- How can p_r learn that the construction of the spanning tree has terminated?

Constructing a Depth-First Search Spanning Tree for a Specified Root

Brief Description

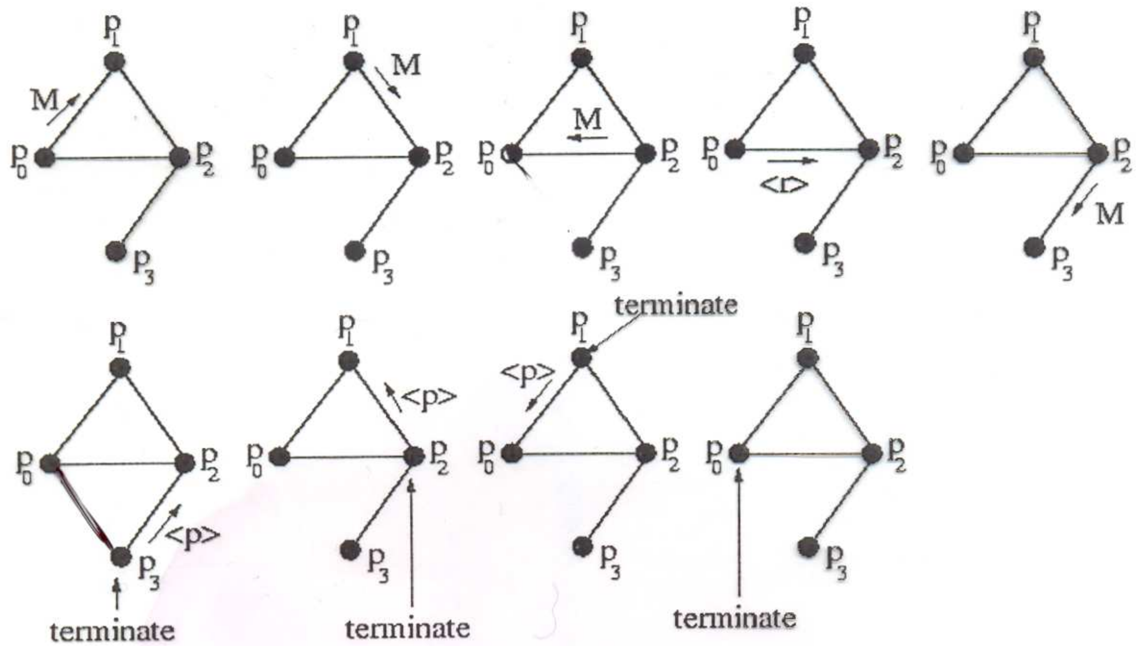
- Each node maintains a set, called unexplored, of "unexplored" neighboring nodes and a set of nodes that will be its children in the constructed spanning tree.
- Initially, the root sends <M> to one of its neighbors and deletes this neighbor from unexplored.
- When a node p_i receives <M> for the first time from some node p_j, p_i marks p_j as its parent node in the spanning tree. Then, p_i chooses one of the nodes in unexplored and forwards <M> to it. If p_i does not receive <M> for the first time, it sends a message of type <already> to p_j and removes p_j from unexplored. If unexplored is empty, p_i sends a message of type <parent> to its parent node.
- When a node p_i receives a message of type <parent> or <already>, it sends <M> to one of the nodes in unexplored. If p_i has received <M> or a message of type <parent> or <already> from all its neighbors, p_i terminates.

Constructing a Depth-First Search Spanning Tree for a Specified Root: The DFS-ST Algorithm

Algorithm 3 Depth-first search spanning tree algorithm for a specified root: code for processor p_i, $0 \le i \le n-1$.

```
Initially parent = ⊥, children = ∅, unexplored = all neighbors of p_i

upon receiving no message:
    if p_i = p_r and parent = ⊥ then                    // root wakes up
    parent := p_i
    explore()

upon receiving ⟨M⟩ from p_j:
    if parent = ⊥ then                       // p_i has not received ⟨M⟩ before
        parent := p_j
        remove p_j from unexplored
        explore()
    else
        send ⟨already⟩ to p_j                            // already in tree
        remove p_j from unexplored
upon receiving ⟨already⟩ from p_j:
    explore()

upon receiving ⟨parent⟩ from p_j:
    add p_j to children
    explore()

procedure explore():
    if unexplored ≠ ∅ then
        let p_k be a processor in unexplored
        remove p_k from unexplored
        send ⟨M⟩ to p_k
    else
        if parent ≠ p_i then send ⟨parent⟩ to parent
        terminate                    // DFS subtree rooted at p_i has been built
```

The DFS-ST Algorithm



$unexplored_0 = \{2\}$	$parent_0 = nil$	$children_0 = \{\}$
$unexplored_1 = \{0,2\}$	$parent_1 = nil$	$children_1 = \{\}$
$unexplored_2 = \{0,1,3\}$	$parent_2 = nil$	$children_2 = \{\}$
$unexplored_3 = \{2\}$	$parent_3 = nil$	$children_3 = \{\}$

The DFS-ST Algorithm

Correctness

- **Lemma**: In every execution of DFS-ST in the asynchronous model, DFS-ST constructs a DFS tree of the network rooted at p_r.

Communication Complexity

- **Lemma**: The communication complexity of DFS-ST is O(m).
- **Proof**: Each node/process sends <M> at most once in each of the edges that are incident to it.
- Each node that receives <M> sends at most one message as a response on each of the edges that are incident to it.
- Thus, the number of messages sent is at most 2m.

The DFS-ST Algorithm

Time Complexity

Lemma: The time complexity of DFS-ST is O(m).

Proof

- Since the time pr executes its first step and before pr terminates, there is always exactly one message in transit.
- No more than two messages are ever send on each edge.
- There are m edges in the graph.

Theorem: There is an asynchronous algorithm to find a depth-first search spanning tree of a network with m edges and n nodes, given a distinguished node, with message complexity O(m) and time complexity O(m).

Constructing a DFS Spanning Tree without a Specified Root

- How can we build a spanning tree when there is no distinguished node?

Brief Description

- Each processor that wakes up spontaneously attempts to build a DFS spanning tree with itself as the root, using a separate copy of DFS-ST.
- If two DFS trees try to connect to the same node, the node will join the DFS tree whose root has the higher identifier.
- p_m: the node with the maximal identifier among the nodes that wake up spontaneously.

Constructing a DFS Spanning Tree without a Specified Root

Algorithm 4 Spanning tree construction: code for processor p_i, $0 \le i \le n-1$.

Initially $parent = \bot$, $leader = -1$, $children = \emptyset$, $unexplored =$ all neighbors of p_i

```
upon receiving no message:
    if parent = ⊥ then                                   // wake up spontaneously
        leader := id
        parent := p_i
        explore()

upon receiving ⟨leader,new-id⟩ from p_j:
    if leader < new-id then                              // switch to new tree
        leader := new-id
        parent := p_j
        children := ∅
        unexplored := all neighbors of p_i except p_j
        explore()
    else if leader = new-id then
        send ⟨already,leader⟩ to p_j                     // already in same tree
                    // otherwise, leader > new-id and the DFS for new-id is stalled

upon receiving ⟨already,new-id⟩ from p_j:
    if new-id = leader then explore()

upon receiving ⟨parent,new-id⟩ from p_j:
    if new-id = leader then                              // otherwise ignore message
        add p_j to children
        explore()

procedure explore():
    if unexplored ≠ ∅ then
        let p_k be a processor in unexplored
        remove p_k from unexplored
        send ⟨leader,leader⟩ to p_k
    else
        if parent ≠ p_i then send ⟨parent,leader⟩ to parent
        else terminate as root of spanning tree
```

Constructing a DFS Spanning Tree without a Specified Root

Correctness

- <leader> messages with leader id m are never dropped because of discovering a larger leader id, by definition of m.
- <already> messages with leader id m are never dropped because they have the wrong leader id.
- <parent> messages with leader id m are never dropped because they have the wrong leader id.
- messages with leader id m are never dropped because the recipient has terminated.
- Thus, the instance of DFS-ST for leader id m completes, and correctness of DFS-ST implies correctness of Algorithm 4.

- Message complexity?
- Time complexity?

Constructing a DFS Spanning Tree without a Specified Root

- Theorem: Algorithm 4 finds a spanning tree of a network with m edges and n nodes, with message complexity O(nm) and time complexity O(m).

Leader Election in Rings

The Leader Election Problem

- Each process should eventually decide that it is either the leader or it is not the leader.
- Exactly one process should decide that it is the leader.
- The leader process may be responsible for achieving synchronization in future activities of the system:
- token re-creation
- recovery from deadlock
- play the role of the root node in the construction of a spanning tree, etc.

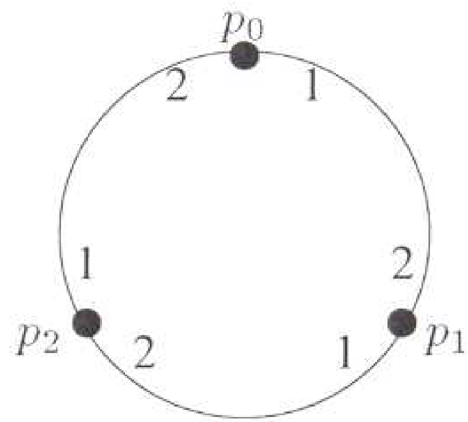
The Leader Election Problem – More formally

- An algorithm is said to solve the leader election problem if it satisfies the following conditions:
  - The terminated states are partitioned into elected and not-elected states. Once a process enters an elected (respectively, not-elected) state, its transition function will only move it to another (or the same) elected (respectively, not-elected) state.
  - In every admissible execution, exactly one process (the leader) enters an elected state and all the remaining processes enter a not-elected state.

The Leader Election Problem

Assumptions

- Ring topology
- The n processes have a consistent notion of left and right



  - For every i, $1 \leq i \leq n$, p_i's channel to p_{i+1} is labeled 1, also known as left or clock-wise, and p_i's channel to p_{i-1} is labeled 2, also known as right or counter-clock-wise (addition and subtraction here are modulo n).

Model - Rings

- An algorithm is anonymous if the processes do not have unique identifiers that can be used by the algorithm.
  - Every process has the same state machine.
- Otherwise, the algorithm is called eponymous (or non-anonymous).
- If n is not known to the algorithm, the algorithm is called uniform
  - The algorithm looks the same for every value of n.
- In an anonymous non-uniform algorithm, for each value of n, there is a single state machine, but there can be different state machines for different ring sizes.
  - n can be explicitly present in the code.

Leader Election in Anonymous Synchronous Rings

Theorem: There is no non-uniform anonymous algorithm for leader election in synchronous rings.

Lemma: For every round k of the admissible execution of an anonymous leader election algorithm in a ring, the states of all the processors at the end of round k are the same.

Proof: By induction on k.

- **Base case:** Straightforward since all processes begin in the same state.
- **Induction Hypothesis:** Assume the lemma holds for round k-1.
- **Induction Step:** Since all processes are in the same state in round k-1, they all send the same messages m_l to the left and m_r to the right.
- In round k, all processes receive message m_r on its left edge and m_l on its right; because they execute the same program, they are in the same state at the end of round k.

Leader Election in Eponymous Asynchronous Rings

An $O(n^2)$ Algorithm

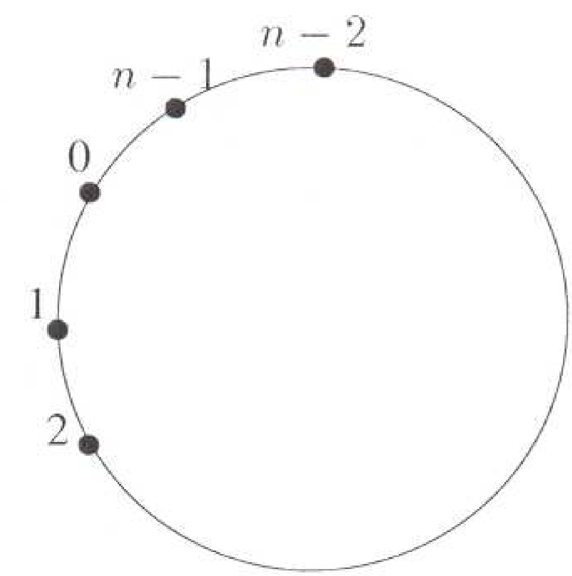
Description of the algorithm:

- Each process sends a message with its identifier to its left neighbor and then waits for messages from its right neighbor.
- When is receives such a message, it checks the identifier in the message:
  - If it is greater than its own identifier, it forwards the message to the left.
  - Otherwise, it shallows the message.
- If a processor receives a message with its own identifier, it declares itself a leader by sending a termination message to its left neighbor and terminating.
- A processor that receives the termination message, forwards it to the left and terminates as non-leader.

Leader Election in Eponymous Asynchronous Rings

Communication Complexity?

- No process sends more than n messages.
- Is there an execution at which $\Theta(n^2)$ messages are sent?



An Algorithm with Communication Complexity O(nlogn) – Main Ideas

- The k-neighborhood of a process p_i in the ring is the set of processes that are at distance at most k from p_i in the ring (either to the left or to the right).

Main Ideas

- The algorithm works in phases:
  - k^{th} phase, $k \geq 0$: a process tries to become a winner for the phase; a process becomes a winner if it has the largest id in its 2^{k}-neighborhood.
  - Only processes that are winners in the k^{th} phase continue to compete in the $(k+1)^{st}$ phase.

An Algorithm with Communication Complexity O(nlogn)- Description

- In phase k, a process p_i that is a phase k-1 winner sends <probe> messages with its identifier to the 2^k-neighborhood (one in each direction).
- A <probe> is shallowed by a processor if it contains an identifier that is smaller than its own identifier.
- If the message arrives at the last process in the neighborhood, then that last process sends back a <reply> message to p_i.
- If p_i receives replies from both directions, it becomes a phase k winner, and it continues to phase k+1.
- A processor that receives its own <probe> message terminates the algorithm as the leader and sends a termination message around the ring.

An Algorithm with Communication Complexity O(nlogn) - Pseudocode

Algorithm 5 Asynchronous leader election: code for processor p_i, $0 \leq i < n$.

Initially, *asleep* = true

```
upon receiving no message:
    if asleep then
        asleep := false
        send ⟨probe,id,0,1⟩ to left and right

upon receiving ⟨probe,j,k,d⟩ from left (resp., right):
    if j = id then terminate as the leader
    if j > id and d < 2^k then                          // forward the message
        send ⟨probe,j,k,d + 1⟩ to right (resp., left)   // increment hop counter
    if j > id and d ≥ 2^k then                          // reply to the message
        send ⟨reply,j,k⟩ to left (resp., right)
                                          // if j < id, message is swallowed

upon receiving ⟨reply,j,k⟩ from left (resp., right):
    if j ≠ id then send ⟨reply,j,k⟩ to right (resp., left)   // forward the reply
    else                                                     // reply is for own probe
        if already received ⟨reply,j,k⟩ from right (resp., left) then
            send ⟨probe,id,k + 1,1⟩ to left and right !      // phase k winner
```

- A message of type <probe> contains the id j of the process that sends it, the phase number k and a hop counter d.
- A message of type <reply> contains the id j and the number of the current phase k.

An Algorithm with Communication Complexity O(nlogn) - Analysis

- **Lemma:** For each k ≥ 1, the number of processes that are phase k winners is at most n/(2^k+1).
- **Proof:**
  - Between two winners of phase k there are 2k other processes in the ring.
- **Remarks**
  - There is just one winner after log(n-1) phases.
  - The total number of messages is:
  - 5n + Sum_{k=1}^{\lceil log(n-1) \rceil+1} 4*2^k*n/(2^{k-1}+1) < 5n + 8n(logn+2)
- **Theorem:** There is an asynchronous leader election algorithm whose message complexity is O(nlogn).

Leader Election in Synchronous Rings

- The reception of no message in a round is a piece of information. Does this help?

An O(n) Upper Bound

The Non-Uniform Algorithm

- Elects the processor with the minimal identifier as the leader.
- It works in phases, each consisting of n rounds.
- In phase $i \geq 0$, if there is a processor with id i, it is elected as a leader and the algorithm terminates.
- Phase i includes rounds ni+1, ni+2, …, ni+n.
- At the beginning of phase i, if a process has id i, and it has not terminated yet, the process sends a message around the ring and terminates as a leader.
- If the process does not have id i, and it receives a message in phase i, it forwards the message and terminates as the non-leader.

Leader Election in Synchronous Rings - The Uniform Algorithm

- Processes wake up either spontaneously in an arbitrary round or upon receiving a message from some other processor.
- Messages that originate from different processes are forwarded at different rates.
  - A message that originates at a processor with identifier i is delayed 2^i-1 rounds at each processor that receives it, before it is forwarded clockwise to the next processor (slow message).
- There is a wake-up phase.
  - Each process that wakes up spontaneously sends a “wake-up” message around the ring (fast message).
- A process that receives a wake-up message before starting the algorithm does not participate in the algorithm and will only act as a relay, forwarding or shallowing messages.
- The leader is elected among the set of participating processes.

Algorithm 6 Synchronous leader election: code for processor p_i, $0 \le i < n$.

```
Initially waiting is empty and status is asleep

let R be the set of messages received in this computation event
S := ∅                                                   // the messages to be sent

if status = asleep then
    if R is empty then                                   // woke up spontaneously
        status := participating
        min := id
        add ⟨id,1⟩ to S                                  // first phase message
    else
        status := relay
        min := ∞

for each ⟨m,h⟩ in R do
    if m < min then
        become not elected
        min := m
        if (status = relay) and (h = 1) then             // m stays first phase
            add ⟨m,h⟩ to S
        else                                             // m is/becomes second phase
            add ⟨m,2⟩ to waiting tagged with current round number
    elseif m = id then become elected
                                          // if m > min then message is swallowed

for each ⟨m,2⟩ in waiting do
    if ⟨m,2⟩ was received 2^m − 1 rounds ago then
        remove ⟨m⟩ from waiting and add to S

send S to left
```

Leader Election in Synchronous Rings – The Uniform Algorithm

- **Lemma 1:** Only the process with the smallest id among the participating processes receives its own message back.
- To calculate the number of messages sent during an admissible execution of the algorithm we divide them into three categories:
  - **Category 1:** First phase messages (fast messages)
  - **Category 2:** Second phase messages (slow messages) sent before the message of the eventual leader enters its second phase (i.e., as long as it is fast).
  - **Category 3:** Second phase messages sent after the message of the eventual leader enters its second phase.
- **Lemma 2:** The total number of messages in the first category is at most n.
- **Proof:** At most one 1^{st} phase message is forwarded by each process.

Leader Election in Synchronous Rings - The Uniform Algorithm

- Let r be the first round in which some process starts executing the algorithm, and let p_i be one of these processes.
- **Lemma 3:** If a process p_j is in (clock-wise) distance k from p_i, then a first-phase message is received by p_j no later than round r+k.
- **Lemma 4:** The total number of messages in the second category is at most n.
- **Proof:** The message of the future leader enters in its 2nd phase at most n rounds after the first message of the algorithm is sent.
- Thus, a message <i> of the 2nd category is sent at most $n/2^i$ times.
- Worst Case: All processes participate and the identifiers are as small as possible (that is, 0, ..., n-1).
- Then, the number of messages of category 2 is Sum_{i=1 to n-1} $n/2^i \leq n$.

Leader Election in Synchronous Rings – The Uniform Algorithm

- Let p_i be the eventual leader and let p_j be any other participating process ($p_i < p_j$).
- At most $n * 2^{idi}$ rounds are needed for $\langle id_i \rangle$ to return to $p_i \rightarrow$ messages of the 3rd category are sent only during $n*2^{idi}$ rounds.
- Message $\langle id_j \rangle$ is forwarded at most:
  - $n*2^{idi} / 2^{idj} = n / 2^{idj-idi}$
- Hence, the total number of messages transmitted in this category is at most:
  - Sum_{j=0 to n-1} $n/2^{idj-idi}$.
- In the worst case, all processes participate and the identifiers are as small as possible. Then, the total number of messages is:
  - Sum_{j=0 to n-1} $n/2^{j} \leq 2n$.

Algorithms in General Synchronous Graphs

- We consider an arbitrary connected graph G = (V,E) having n nodes. Sometimes, we will assume that the graph is a strongly-connected digraph.
- The number n of nodes and the diameter, diam, of the network can be either known or unknown to the processes, or an upper bound on these quantities might be known.
- Processes have unique identifiers. The identifier of process p_i is denoted by id_i.
- The indices 1, … ,n have been assigned to the processes (nodes) in order to name them.
- Unlike what happens in rings, these indices have now no connection to their position in the graph.
- The processes do not know their indices (each process knows only its id).

Leader Election in General Synchronous Graphs

Brief Description

- Every process maintains a record of the maximum pid it has seen so far (initially its own).
- At each round, each process propagates this maximum on all of its outgoing edges.
- After diam rounds, if the maximum value seen is the process's own pid, the process elects itself the leader.
- Otherwise, it is a non-leader.

State of p_i

- id_i: identifier of p_i
- max-id_i: maximum pid that p_i has seen so far, initially equal to id_i
- $status_i \in$ {UNKNOWN, LEADER, NON-LEADER}, initially UNKNOWN
- $rounds_i$: an integer, initially 0

Leader Election in General Synchronous Graphs

- Initially, id_i is contained in all outbuf tables of process p_i, $\forall$ i.

Actions of p_i in each round

$rounds_i$ = $rounds_i$ + 1;
let U be the set of UIDs that arrive from neighboring processes;
$max\text{-}uid_i = \max(\{max\text{-}uid_i\} \cup U)$
if ($rounds_i$ == diam) then
    if ($max\text{-}uid_i = id_i$) then $status_i$ = LEADER;
    else $status_i$ = NON-LEADER;
if ($rounds_i$ < diam) then
send $max\text{-}uid_i$ to all neighbors;

Leader Election in General Synchronous Graphs

- Let i_{max} be the index of the process with the maximum identifier and let id_{max} be that pid.

Theorem

- In each execution of the FloodMax algorithm, process i_{max} outputs leader and each other process outputs non-leader, within diam rounds.

Proof

- For each $0 \le k \le$ diam and for each process j, after k rounds, if the distance from i_{max} to j is at most k, then $max\text{-}id_j = id_{max}$.
- To prove the claim, we should first prove the following:
  - For every k and j, after k rounds, $rounds_j = k$.
  - For every k and j, after k rounds, $max\text{-}id_j \le id_{max}$.

Leader Election in General Synchronous Graphs

Complexity

- Time Complexity? O(diam) rounds
- Communication Complexity? O(diam*|E|) messages

Reducing the Communication Complexity - Algorithm OptFloodMax

- How can we decrease the communication complexity in many cases (without necessarily decreasing the order of magnitude in the worst case)?

Leader Election in General Synchronous Graphs

- The state of p_i includes an additional variable, called new-info$_i$, initially TRUE.
- Initially, id_i is contained in all outbuf tables of process p_i, $\forall$ i.

Actions of process p_i in each round

rounds$_i$ = rounds$_i$ + 1;
let U be the set of pids that arrive from neighboring processes

if (max(U) > max-id$_i$) then new-info$_i$ = TRUE;

else new-info$_i$ = FALSE;

max-uid$_i$ = max({max-uid$_i$} $\cup$ U)

if (rounds$_i$ == diam) then

    if (max-uid$_i$ = id$_i$) then status$_i$ = LEADER;

    else status$_i$ = NON-LEADER;

if (rounds$_i$ < diam AND new-info$_i$ == TRUE) then

    send max-uid$_i$ to all neighbors

Leader Election in General Synchronous Graphs

Theorem

- In each execution of the OptFloodMax algorithm, process i_{max} outputs leader and each other process outputs non-leader, within diam rounds.

Proof – Main Ideas

- **Lemma 1**: For any k, $0 \le k \le$ diam, and any i,j, where $j \in nbrs_i$, the following holds: after k rounds, if $\text{max-id}_j < \text{max-id}_i$ then new-info_i = TRUE.
- **Proof**: By induction on k.
- Base case: The claim holds trivially since all new-info variables are initialized to TRUE.
- Induction Step: Consider any particular processes i and j, where $j \in nbrs_i$.
- If max-id_i increases in round k, by the code, new-info_i gets set to TRUE (which suffices).
- If max-id_i does not increase in round k, the induction hypothesis implies that either max-id_j was already sufficiently large (i.e., as large as max-id_i) or else new-info_i == TRUE just before round k.
- In the former case, max-id_j remains sufficiently large because the value never decreases. In the latter case, the new information is sent from i to j at round k, which causes max-uid_j to become sufficiently large.

Leader Election in General Synchronous Graphs

- **Lemma 2:** For each k, $0 \le k \le$ diam, after k rounds, the values of variables: id, max-id, status, and rounds, are the same in the states of both algorithms.
- **Proof:** By induction on k.
- Consider any particular processes i and j, where $j \in nbrs_i$.
- If $new\text{-}info_i$ == TRUE before round k, then i sends the same information to j in round k in OptFloodMax as it does in FloodMax.
- If $new\text{-}info_i$ == FALSE before round k, then i sends nothing to j in round k in OptFloodMax, but sends $max\text{-}id_i$ to j in round k in FloodMax. However, Lemma 1 implies that, in this case, $max\text{-}id_j \ge max\text{-}id_i$ before round k. So, the message has no effect in FloodMax.
- Thus, i has the same effect on $max\text{-}uid_j$ in both algorithms.
- Since this is true for all i and j, it follows that the max-id values remain identical in both algorithms.

Shortest Paths

- We consider a strongly connected directed graph, with the possibility of unidirectional communication between some pairs of neighbors. We assume that each directed edge e = <i,j> has an associated non-negative real-valued weight, which we denote by weight(e) or $weight_{i,j}$.
- The weight of a path is defined to be the sum of the weights on its edges.
- A shortest path from some node i to some node j is a path with minimum weight (among all paths that connect i and j).

Problem

- Find a shortest path from a distinguished source node p_r in the digraph to each other node in the digraph.
- We assume that every process initially knows the weight of all its incident edges.
  - The weight of an edge appears in special weight variables at both endpoint processes.
- We assume that each process knows n.

Shortest Paths

- We require that each process should determine:
  - its parent in a particular shortest paths tree, and
  - the total weight of its shortest path from p_r.
- If all edges are of equal weight, then a BFS tree is also a shortest paths tree.
- We assume that the weights on the edges can be unequal.

Shortest Paths

The SynchBellmanFord Algorithm – Process i

- Each process p_i maintains a variable $dist_i$ where it stores the shortest distance from p_r it knows so far. Initially, $dist_r = 0$ and $dist_i = \infty$ for each $i \neq r$.
- Another variable $parent_i$, stores the incoming neighbor p_j that precedes p_i in a path whose weight is $dist_i$. Initially, $parent_i$ = nill, for each i.
- At each round, process p_i sends $dist_i$ to all its outgoing edges.
- Then, each process p_i updates its $dist_i$ by a "relaxation step", in which it takes the minimum of its previous dist value and all the values $dist_j + weight_{j,i}$, where j is an incoming neighbor.
- If $dist_i$ is changed, the $parent_i$ variable is also updated accordingly.
- After n-1 rounds, $dist_i$ contains the shortest distance, and $parent_i$ the parent of p_i in the shortest path tree.

Shortest Paths

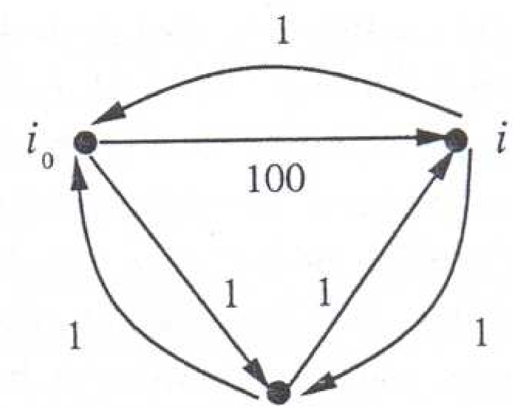
Correctness

It is not hard to see that, the following is true after k rounds:

- Every process p_i has its $dist_i$ and $parent_i$ variables corresponding to a shortest path among the paths from p_r to p_i consisting of at most k edges.
  - If there is no such paths, then $dist_i = \infty$ and $parent_i$ is undefined.

Complexity

- Number of messages? (n-1)*|E|
- Time Complexity? (n-1) rounds



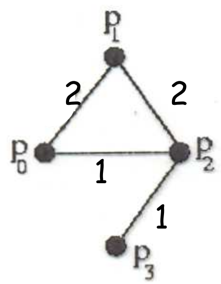
Minimum Spanning Tree

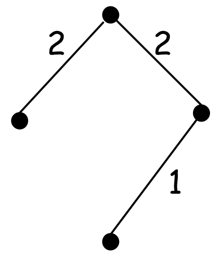
- A spanning forest of an undirected graph G = (V,E) is a forest (i.e., a graph that is acyclic but not necessarily connected) that consists entirely of undirected edges in E and that contains every vertex of G .
- A spanning tree of an undirected graph G is a spanning forest of G that is connected.
- If there are weights associated with the edges in E, then the weight of any subgraph of G (such as a spanning tree or spanning forest of G) is defined to be the sum of the weights of its edges.

Problem

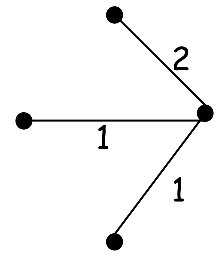
- Find a minimum weight spanning tree for the entire network.
  - Each process is required to decide which of its incident edges are and which are not part of the minimum spanning tree.

Minimum Spanning Tree versus Shortest-Path Trees





shortest path tree for p_1



minimum spanning tree

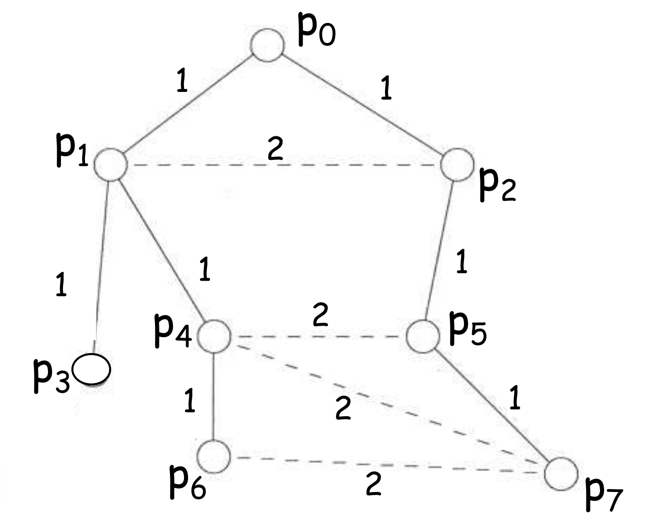
Minimum Spanning Tree – Basic Theory

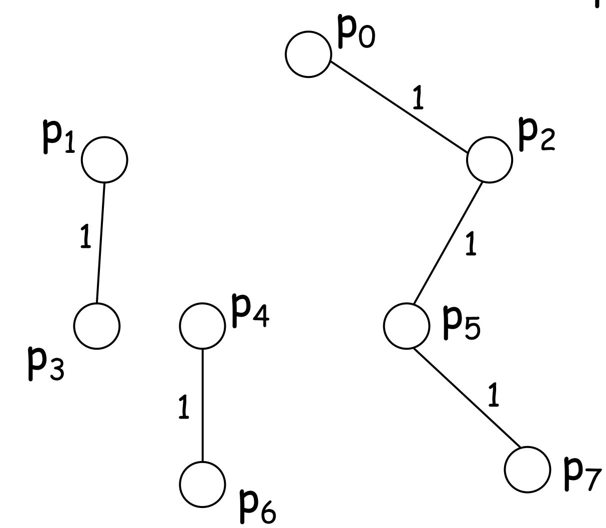
Main Ideas

- Start with the trivial spanning forest that consists of n individual nodes and no edges.
- Repeatedly merge components by connecting edges until a spanning tree is produced.
- In order to end up with a minimum spanning tree, the merging should occur with care.
- **Lemma 1**: Let $G = (V,E)$ be a weighted undirected graph, and let $\{(V_i, E_i): 1 \leq i \leq k\}$ be any spanning forest for G, where $k > 1$. Fix any i, $1 \leq i \leq k$. Let e be an edge of smallest weight in the set

  $\{e': e'$ has exactly one endpoint in $V_i\}$.

  Then, there is a spanning tree for G that includes $\cup_j E_j$ and e, and this tree is of minimum weight among all spanning trees for G that include $\cup_j E_j$.

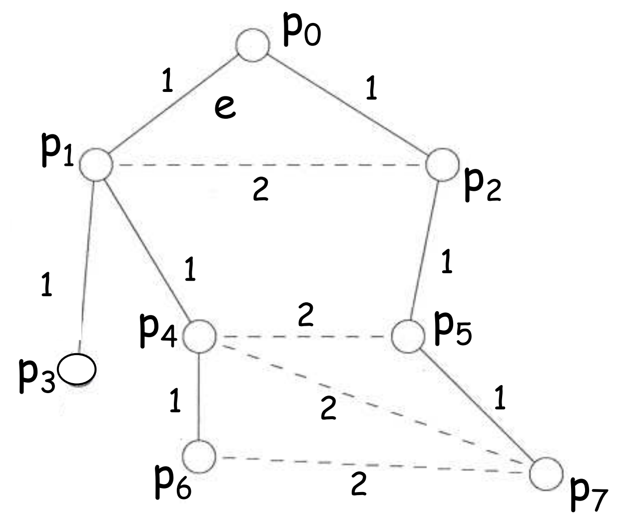


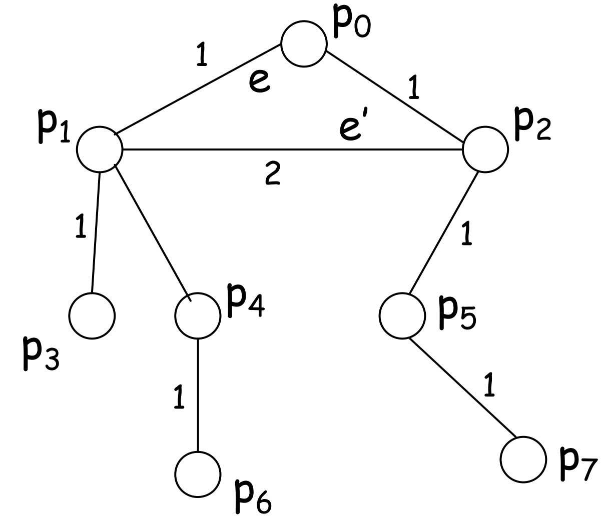


Minimum Spanning Tree

Proof of Lemma 1

- By contradiction. Suppose that there exists a spanning tree T that contains $\cup_j E_j$, does not contain e, and is of strictly smaller weight than any other spanning tree that contains $\cup_j E_j$ and e.
- Consider the graph T' obtained by adding e to T. Clearly, T' contains a cycle which has another edge $e' \neq e$ that is outgoing from V_i.
- By the choice of e, weight(e) $\leq$ weight(e').
- Now, consider the graph T'' constructed by deleting e' fromT'.
- Then T'' is a spanning tree for G, it contains $\cup_j E_j$ and e and its weight is no greater than that of T.
- This contradicts the claimed property of T.



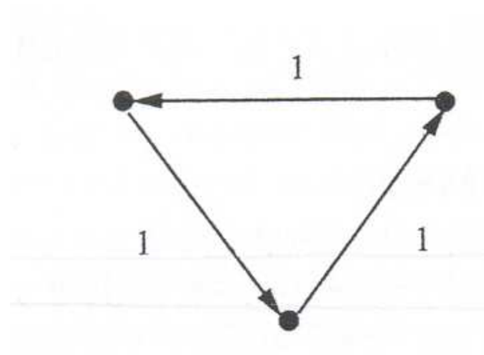


Minimum Spanning Tree

General Strategy for MST

- Start with the trivial spanning forest that consists of n individual nodes and no edges.
- Repeatedly do the following:
  - Select an arbitrary component C in the forest and an arbitrary outgoing edge e of C having minimum weight among the outgoing edges of C.
  - Combine C with the component at the other end of e, including edge e in the new combined component.
- Stop when the forest has a single component.

- What is the parallel version of this algorithm?
  - Extend the forest with several edges determined concurrently.
- Why does the algorithm fail in its parallel version?
  - If weights of edges are not distinct, a cycle can be created.

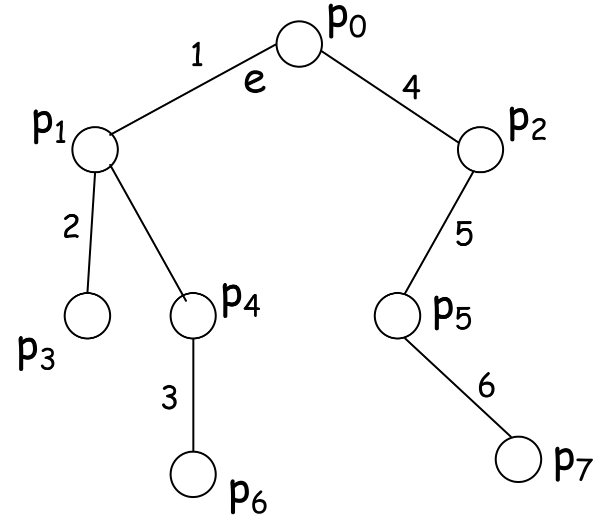


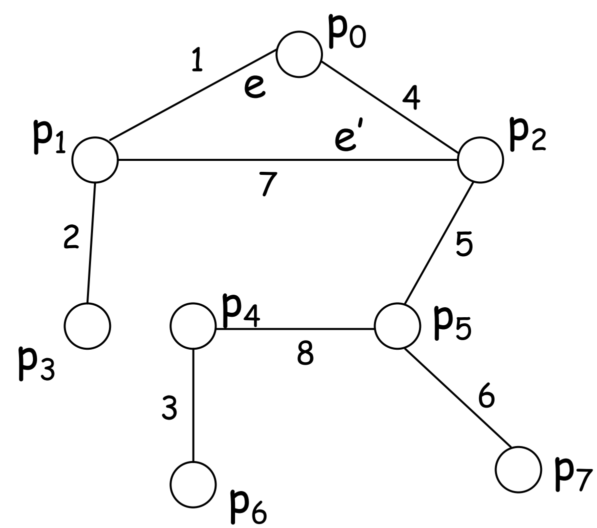
Minimum Spanning Tree

Lemma 2: If al edges of a graph G have distinct weights, then there is exactly one MST for G.

Proof: Similar to that of Lemma 1.

- Suppose there are two distinct minimum-weight spanning trees, T and T′, and let e be the minimum-weight edge that appears in only one of the two trees. Suppose wlog that e $\in$ T.
- Then the graph T″ obtained by adding e to T′ contains a cycle, and at least one other edge in that cycle, e′, is not in T.
- Since the edge weights are all distinct and since e′ is in only one of the two trees, we must have weight(e′) > weight(e), by our choice of e.
- Then, removing e′ from T″ yields a spanning tree with a smaller weight that T′, which is a contradiction.





Minimum Spanning Tree

- The algorithm builds the components in levels.
- For each k, the components of level k constitute a spanning forest, where:
  - Each level k component consists of a tree that is a subgraph of the MST.
  - Each level k component has at least 2^k nodes.
- Every component, at every level, has a distinguished leader node.
- The processes allow a fixed number of rounds, which is O(n), to complete each level.
- The n components of level 0 consist of one node each and no edges.
- Assume inductively that the level k components have been determined (along with their leaders), $k \geq 0$. Suppose that each process knows the id of the leader of its component. This id is used as an identifier of the entire component.
- Each process also knows which of its incident edges are in the component's tree.

Minimum Spanning Tree

To get the level k+1 components:

- Each level k component C conducts a search (along its spanning tree edges) for an edge e such that e is an outgoing edge of C and has the minimum weight among all outgoing edges of C (e is called MWOE). How can we implement this?
- When all level k components have found their MWOEs, the components are combined along all these MWOEs to form the level k+1 components.
- This involves the leader of each level k component communicating with the component process adjacent to the MWOE, to tell it to mark the edge as being in the new tree; the process at the other end of the edge is also told to do the same thing.
- Then a new leader is chosen for each level k+1 component.

Minimum Spanning Tree

<u>It can be proved that:</u>

- For each group of level k components that get combined into a single level k+1 component, there is a unique edge e that is the common MWOE of two of the level k components in the group.
- We let the new leader be the endpoint of e having the larger pid.
- The pid of the new leader is propagated throughout the new component, using broadcast.

Termination

- After some number of levels, the spanning forest consists of only a single component containing all the nodes in the network.
- Then, a new attempt to find a MWOE will fail, because no process will find an outgoing edge.
- When the leader learns this, it broadcasts a message saying that the algorithm is completed.

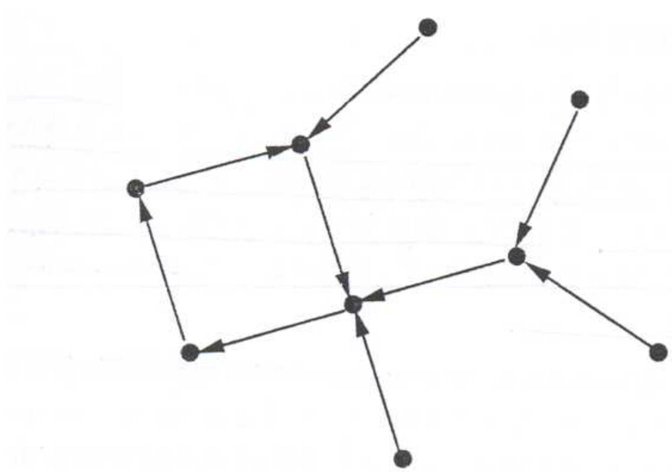
Minimum Spanning Tree

Claim

- Among each group of level k components that get combined, there is a unique edge that is the common MWOE of both endpoint components.

Proof

- Consider the component digraph *G'*:
  - The nodes of *G'* are the level k components that combine to form one level k+1 component.
  - The edges of *G'* are the MWOEs.
  - *G'* is a weakly connected digraph in which each node has exactly one outgoing edge. (A digraph is weakly connected if its undirected version is connected.)
- It can be proved that every weakly connected digraph in which each node has exactly one outgoing edge contains exactly one cycle.



Minimum Spanning Tree

- Because of the way *G'* is constructed, successive edges in the cycle must have non-increasing weights.
- $\Rightarrow$ the length of this cycle cannot be > 2
- $\Rightarrow$ the length of the cycle = 2
- $\Rightarrow$ this corresponds to an edge that is the common MWOE of both adjacent components.

- Why is it important that the system is synchronous?
  - To ensure that when a process p_i tries to determine whether or not the other endpoint p_j of a candidate edge is in the same component, both p_i and p_j have up-to-date component ids.

Minimum Spanning Tree

Complexity

- How many levels do we have until termination?
  O(logn). Why?
- How many rounds are executed in each level?
  O(n). Why?
- What is the time complexity of the algorithm?
  O(nlogn)
- How many messages are sent at each level?
  O(n+|E|). Why?
- What is the communication complexity of the algorithm?
  O((n+|E|)logn)

Minimum Spanning Tree

- The algorithm assumes that the weights of the edges are all distinct.
- How can we solve the problem without making this assumption?
- Is there any way to distinguish different edges that have the same weight?

Asynchronous Systems: Leader Election – General Undirected Graphs

- The FloodMax algorithm does not extend directly to the asynchronous setting, because there are no rounds in the asynchronous model.
- How can we simulate the rounds asynchronously?
- Each process that sends a round k message must tag that message with its round number k.
- The recipient waits to receive round k messages from all its neighbors before performing its round k transition.
- By simulating diam rounds in this way, the algorithm can terminate correctly.
- Can we simulate OptFloodMax (the optimized version of FloodMax) in an asynchronous network? What is the problem encountered?

Asynchronous Systems: Leader Election – General Undirected Graphs

- Whenever a process obtains a new maximum pid, it sends that pid to its neighbors at some later time.
- This strategy will indeed eventually propagate the maximum to all processes.

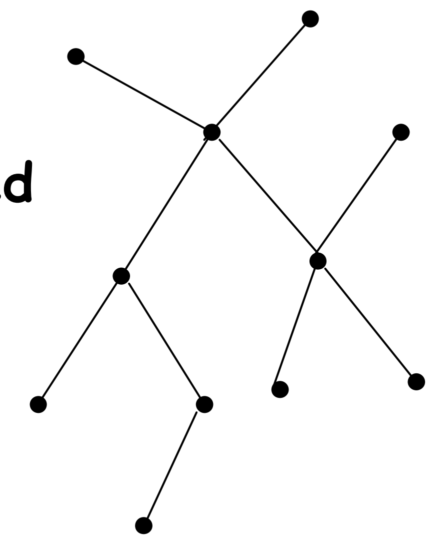
Problem

- Now the processes have no way of knowing when to stop.

Solutions to the asynchronous leader election problem

- Asynchronous broadcast and convergecast
- Convergecast using a spanning tree
- Using a synchronizer to simulate a synchronous algorithm
- Using a consistent global snapshot to detect termination of an asynchronous algorithm.

Leader Election in Asynchronous Systems given an Unrooted Spanning Tree



STtoLeader Algorithm

- A convergecast of <elect> messages is initiated starting from the leaves of the tree.
  - Each leaf node is initially enabled to send an <elect> message to its unique neighbor.
  - Any node that receives <elect> messages from all but one of its neighbors is enabled to send an <elect> message to its remaining neighbor.
- In the end,
  1. Some particular process receives <elect> messages along all of its channels before it has sent out an <elect> message
     - the process at which the <elect> messages converge elects itself as the leader.
  2. <Elect> messages are sent on some particular edge in both directions.
     - the process with the largest pid among the processes that are adjacent to this edge elects itself as the leader.

Breadth-First Search Tree

- We assume an undirected, connected graph with a distinguished node p_r.
- Each edge e = (i,j) has been assigned a weight, denoted by weight(e) or weight(i,j), which is a non-negative real number known to both processes that are incident to e.
- How can we modify the Flooding algorithm in order to construct a BFS spanning tree?

Breadth-First Search Tree

1st Solution: The AsynchBFS Algorithm

Code for process p_i

Initially, $parent_i$ = nill, $dist_i = 0$ if $p_i = p_r$ and $dist_i = \infty$ if $p_i \neq p_r$;

```
upon receiving no message:
    if (p_i == p_r) and (parent_i == nill) then
        send <0> to all neighbors;
        parent_i = p_i;

upon receiving <m> from neighbor p_j:
    if (m+1 < dist_i) then
        dist_i = m+1;
        parent_i = p_j;
        send <dist_i> to all neighbors except p_j;
```

Breadth-First Search Tree

1st Solution: The AsynchBFS Algorithm

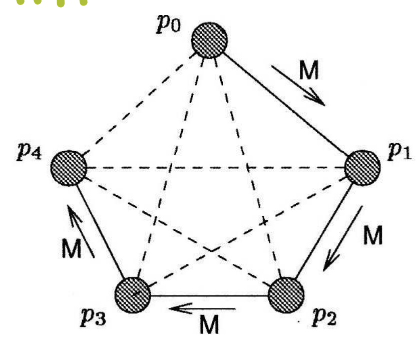
Theorem: In any execution of the AsynchBFS algorithm, the system eventually stabilizes to a state in which the parent variables represent a breadth-first tree.

Proof (brief):

- It can be proved that in any reachable configuration the following is true:
  - For each process $p_i \neq p_r$, $dist_i$ is the length of some path π from p_r to p_i in G in which the predecessor of p_i is $parent_i$.
  - For each message m in any of the inbuf tables of a process p_i, (m+1) is the length of some path π from p_r to p_i. A similar statement is true for the messages that are in the outbuf tables of p_i.
- It can also be proved that, in each reachable configuration, for each pair of neighboring processes i,j, either $dist_j \leq dist_i+1$, or the message <$dist_i$> is in one of the outbuf tables of p_i or in one of the inbuf tables of p_j.

Breadth-First Search Tree

1st Solution: The AsynchBFS Algorithm



- Complexities?
  Number of messages: O(n*m),
  Time Complexity: O(diam)

Termination

- How can I use an acknowledgement mechanism to get termination?
- For each message an acknowledgement is sent.
- Each time process p_i receives a message from some neighboring process p_j which causes an update on variable dist (and therefore results in sending messages with the new value to the neighboring processes), p_i waits for acknowledgments from all its neighboring processes before it sends its own acknowledgement to p_j.
- Bookkeeping is needed to keep the different sets of acknowledgments by the same process separate.

Breadth-First Search Tree

1st Solution: The LayeredBFS Algorithm

- The BFS spanning tree is constructed in layers.
- Each layer k consists of the nodes at depth k in the tree.
- The layers are constructed in a series of phases, one for each layer, all coordinated by process p_r.

1st Phase

- Process p_r sends <search> messages to all of its neighbors and waits to receive acknowledgements.
- A process that receives a search message at phase 1 sends a positive ack.
- This enables all processes at depth 1 to determine their parent, namely p_r, and of course, p_r knows its children.
- Inductively, we assume that k phases have been completed and that the first k layers have been constructed: each process at depth at most k knows its parent and each process at depth at most k-1 knows its children; p_r knows that phase k has been completed.

Breadth-First Search Tree

1st Solution: The LayeredBFS Algorithm

- **Phase (k+1): Construction of the (k+1)st level**
- Process p_r broadcasts a <newphase> message along all the edges of the spanning tree constructed so far. These messages are intended for the depth k processes.
- Upon receiving a <newphase> message, each depth k process sends out a <search> message to all its neighbors except its parent and waits to receive acks.
- When a process $p_j \neq p_r$ receives its first <search> message in an execution, it designates p_i as its parent and returns a positive ack. If p_j receives a subsequent <search> message, it returns a negative ack.
- Each time p_r receives a message of type <search>, it returns a negative ack.
- When a depth k process has received acks for all its <search> messages, it designates the processes that have sent positive acks as its children.
- The depth k processes convergecast the information that they have completed the determination of their children back to p_r, along the edges of the edges of the depth k spanning tree.
- They also convergecast a bit, saying whether any depth (k+1) nodes have been found. Process p_r terminates the algorithm after a phase at which no new nodes are discovered.

Breadth-First Search Tree
1st Solution: The LayeredBFS Algorithm

Theorem

- The LayeredBFS algorithm calculates a BFS spanning tree.

Complexities?

	Communication Complexity	Time Complexity
AsynchBFS	$O(m*n)$	$O(diam)$
LayeredBFS	$O(m + n*diam)$	$O(diam^2)$

-

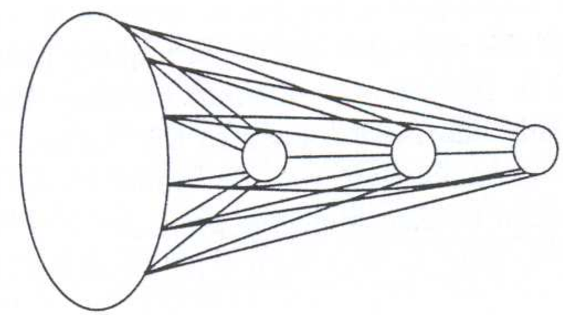
Asynchronous Systems: Minimum Spanning Tree

Assumptions

- The edge weights are unique.
- Processes do not know n or diam.
- The processes are initially quiscent and Each process receives a wakeup signal that makes it starting the execution of the algorithm.
- The output of the algorithm is the set of edges comprising an MST; every process is required to output the set of edges adjacent to it that are in the MST.

Asynchronous Systems: Minimum Spanning Tree

- Difficulties that arise if we try to run SynchGHS in an asynchronous network:
  - **Difficulty 1:** When a process p_i queries a neighbor process p_j to see if p_j is in the same component of the current spanning forest, a situation could arise whereby p_j is actually in the same component as p_i but has not yet learned this (because a message containing the latest component id has not yet reached it).
  - **Difficulty 2:** The SynchGHS achieves a message cost of O(nlogn + |E|), based on the fact that levels are kept synchronized. Each level k component has at least 2k nodes -> # of levels = O(logn).

    In the asynchronous setting, there is a danger of constructing the components in an unbalanced way, leading to many more messages, i.e., the number of messages sent by a component to find its MWOE can be at least proportional to the number of nodes in the component.

Asynchronous Systems: Minimum Spanning Tree

- **Difficulty 3:** In SynchGHS, the levels remain synchronized, whereas in the asynchronous setting, some components could advance to higher levels than others. It is not clear what type of interference might occur as a result of concurrent searches for MWOEs by adjacent components at different levels.

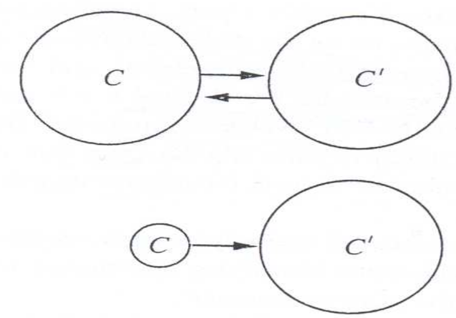
Asynchronous Systems: Minimum Spanning Tree

- The initial components are just the individual nodes. Each component has a distinguished leader node and a spanning tree that is a subgraph of the MST.
- Within any component, the processes cooperate in an algorithm to find the MWOE for the entire component:
  - the leader initiate a broadcast
  - each node finds its own mwoe
  - information about all these edges is convergecast back to the leader, who can determine the MWOE for the entire component. This MWOE will be included in the MST.
- The leader sends a message to the processes that are incident to the chosen MWOE and the two components may then combine into a new larger component.
- This procedure is repeated until all the nodes in the graph are included in a single component.

Asynchronous Systems: Minimum Spanning Tree

1. How does a process p_i know which of its edges lead outside its current component?
   - Some sort of synchronization is needed to, to ensure that process p_j does not respond that it is in a different component unless it has current information about its component name.
2. How is it possible to have just O(logn) phases?
   - We will associate a level with each component, as we do in SynchGHS. As in SychGHS, all the initial single-node components will have level = 0, and the number of nodes in a level k component will be at least 2^k.
   - A level k+1 component will only be formed by combining exactly two level k components.
3. How can the 3rd difficulty be solved?
   - Some synchronization will be required to avoid interference between concurrent searches for MWOEs by adjacent components at different levels.

Asynchronous Systems: Minimum Spanning Tree



- The AsynchGHS algorithm combines components in two different ways:
- **merge**: This combining operation is applied only to two components C and C′ where level(C) = level(C′), and C and C′ have the same MWOE.
  - The result of a merge is a new component of level = k+1.
- **absorb**: It is applied to two components C and C′ s.t. level(C) < level(C′) and the MWOE of C leads to a node in C′.
  - This enhances C′ by adding C to it; this enhanced version of C′ is at the same level as C′ was before the absorption.

Asynchronous Systems: Minimum Spanning Tree

Lemma

- Suppose that we start from an initial situation in which each component consists of a single node with level = 0, and apply any allowable finite sequence of merge and absorb operations. Then after this sequence of operations, either tjere is only one component, or else some merge or absorb operation is enabled.

Proof

- Suppose there is more than one components after a sequence of merge or absorb operations. We show that there is some applicable operation.
- We consider the "component digraph" G', whose nodes are the current components and whose directed edges correspond to MWOEs.
- In G' there is a cycle of length 2 $\Rightarrow$ there are two components C and C', whose MWOEs point to each other $\Rightarrow$ the two MWOEs must be the same edge in G
- If level(C) = level(C') $\Rightarrow$ merge. Otherwise $\Rightarrow$ absorb

Asynchronous Systems: Minimum Spanning Tree

- For every component of level 1 or greater, we identify a specific edge which we call its core edge. This edge is defined in terms of the series of merge and absorb operations that are used to construct the component:
  - after a merge operation, the core is the common MWOE of the two original components,
  - after an absorb operation, the core is the original component with the larger level number.
- For each component, the pair <weight of core edge, component level> is used as a component identifier.
- The endpoint of the core edge with the highest pid is designated to be the leader node of the component.

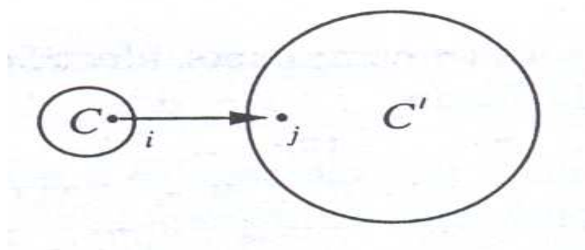
Asynchronous Systems: Minimum Spanning Tree

- How does a process p_i determines if a neighboring process p_j is outgoing from p_i's component?
- If process p_j's current component identifier is the same as that of p_i, then process p_i is certain that p_j is in the same component as itself.
- If these ids are different:
  - If p_j's latest known level is at least as high as that of p_i, then p_j cannot be at the same component as that of p_i.
    - A node can only have one component identifier for each level, and when p_i is actively searching for its outgoing edges, it is certain that p_i's component identifier is up-to-date.
  - If the level of p_j is strictly less than that of p_i, p_j simply delays answering p_i until its own level raises to become at least as great as that of p_i.

Asynchronous Systems: Minimum Spanning Tree

- Could this new delay conceivably cause progress to be blocked?
  - We repeat the same argument as previously (for proving progress), but with the nodes of G' to be only those components with the current lowest level, let it be k.
  - If some MWOE of such a component leads to a higher level component $\Rightarrow$ absorb is possible
  - Otherwise, there is a cycle of length 2 in G'. Thus, two of these components have the same MWOE $\Rightarrow$ merge is possible

Asynchronous Systems: Minimum Spanning Tree



- How shall we overcome the 3rd difficulty?
- What happens if a lower level component C gets absorbed into a higher level component C′ while C′ is involved in determining its own MWOE?
  1. Process p_j has not yet determined its MWOE from the component at the time the absorb occurs. Then C participates in the search of the MWOE.
  2. Process p_j has already determined its mwoe (let it be e). Then, e ≠ (i,j) (since e leads to a component with a level at least as large as that of C′) ⇒ weight(e) < weight(i,j).
  3. Then e cannot be incident to a node of C. Why is this so?
  4. No edge of C can have smaller weight ⇒ merge is correct!!!!

Asynchronous Systems: Minimum Spanning Tree

- **<initiate>**: it is broadcast throughout a component, starting at the leader, along the edges of the component's spanning tree; it triggers processes to start trying to find their mwoes, and it carriers the component id
- **<report>**: it convergecasts information about MWOEs back toward the leader
- **<test>**: a proces p_i sends a <test> message to a process p_j to try to ascertain whether or not p_j is in the same component as p_i; this is part of the procedure by which process p_i searchers for its own mwoe.
- **<accept>** and **<reject>**: these are sent in response to <test> messages (<accept> is responding node is in a different component, <reject> otherwise)
- **<changeroot>**: it is sent from the leader of a component toward the component process that is adjacent to the component's MWOE, after the MWOE has been determined; it is used to tell that process to attempt to combine with the component at the other end of the MWOE.
- **<connect>**: it is sent across the MWOE of a component C when that component attempts to combine with another component.
  - merge occurs when connect messages have been sent both ways along the same edge
  - absorb occurs when a connect message has been sent one way along an edge that leads to a process at a higher level than the sender.

Asynchronous Systems: Minimum Spanning Tree

- Each process p_i classifies its incident edges into three categories:
  - **branch**: edges that have already been determined to be part of the MST
  - **rejected**: edges that have already been determined not to be part of the MST (because the lead to other nodes within the same component)
  - **basic**: all other edges.
- Messages of type test are sent by a process p_i only across basic edges.
- Process p_i tests its basic edges sequentially, lowest weight to highest .

- When two <connect> messages cross a single edge, a merge operation occurs ⇒ new core edge, new level, new leader.
- The new leader then broadcast <initiate> messages to begin looking for the MWOE of the new component. This message informs all processes about the id of the new component.
- During an absorb (through edge (i,j)), process p_j knows whether it has already found its MWOE. In either case, process p_j will broadcast an <initiate> message to its previous component to tell the processes in that component the latest component identifier.

Asynchronous Systems: Minimum Spanning Tree

Theorem

- The GHS algorithm solves the MST problem in an arbitrary connected undirected graph network.
- **Proof**
- 4 different proofs of correctness for the algorithm have been proposed.
- All of them are very complicated. None of them is sufficiently nicely organized to be presented in class (or even in books)!
- The presentation of a simple, modular proof for the algorithm is still an open problem!

Asynchronous Systems: Minimum Spanning Tree

Communication Complexity: O(m + nlogn)

- O(m): number of test-reject messages
- All other messages are charged to the task of finding the MWOE for a specific component.
  - For each level, and for each component C:
    - For each node of C there is only one test-accept pair of messages.
    - O(|C|) messages of type initiate-report are sent.
    - The number of messages of type <changeroot> and <connect> is also O(|C|).
    - Thus, the total number of messages is bounded as follows: Sum_{C} |C| = Sum_{k: 0 ≤ k ≤logn} (Sum_{C: level(C) = k} |C|) = Sum_{0}^{logn} n = n logn

Time Complexity: O(nlogn)