Advanced Graph Algorithms

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.
- 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-uid_i = max({max-uid_i} ∪ 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) then
   send max-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 $0 \leq k \leq$ diam and for each process j, after k rounds, if the distance from i_{max} to j is at most k, then max-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-$id_j \leq id_{max}$.

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} ∪ 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, $0 \leq k \leq$ diam, and any i,j, where $j \in nbrs_i$, the following holds: after k rounds, if max-id$_j$ < max-id$_i$ then new-info$_i$ = TRUE.
- **Proof**: By induction on k.

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.

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.

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
- 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.

CS-486: Principles of Distributed Computing Panagiota Fatourou

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.

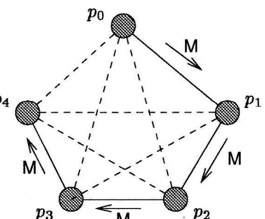
Proof (brief):

- 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)



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($diam^2$)

Shortest Paths in Synchronous Systems

- 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 in Synchronous Systems

- 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 in Synchronous Systems

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 in Synchronous Systems

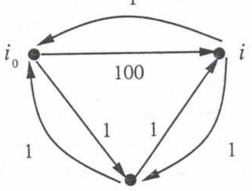
Correctness

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



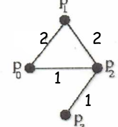
Minimum SpanningTree – Synchronous Systems

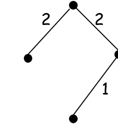
- A spanning forest of an undirected graph G = (V,E) is a forest (i.e., a graph that is acyclic but not necessarily connected) that consists of undirected edges in E and that contains every vertex of G .
- A spanning tree of an undirected graph G is a spanning forest of G that is connected.
- If there 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

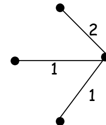
- Find a minimum weight spanning tree for the entire network.
  - Each process is required to decide which of its incident edges are and which are not part of the minimum spanning tree.

Minimum SpanningTree versus Shortest-Path Trees





shortest path tree for p_1



minimum spanning tree

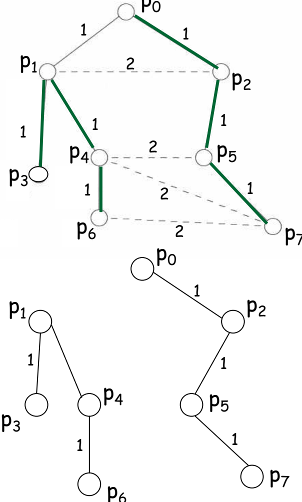
Minimum SpanningTree – Basic Theory

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

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

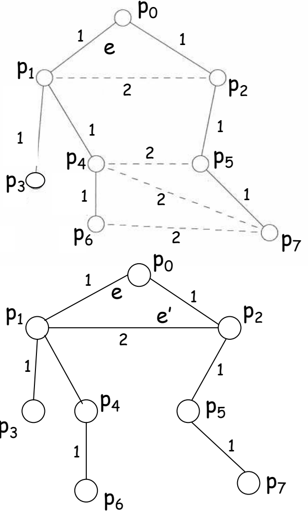
  Then, there is a spanning tree for G that includes $\cup_j E_j$ and e, and this tree is of minimum weight among all spanning trees for G that include $\cup_j E_j$.



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'≠e that is outgoing from V_i.
- By the choice of e, weight(e) ≤ weight(e').
- Now, consider the graph T'' constructed by deleting e' from T'.
- 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.
- A contradiction.

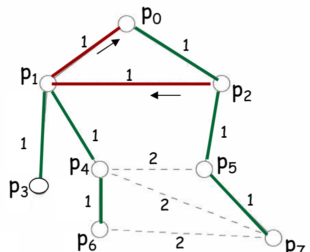


Minimum Spanning Tree

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 enc 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 cre

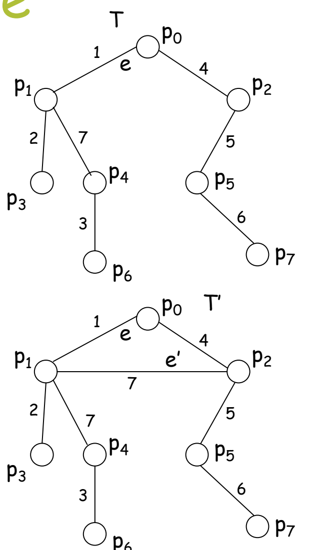


Minimum Spanning Tree

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

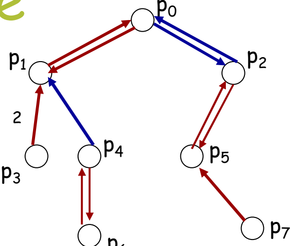
Proof: Similar to that of Lemma 1.

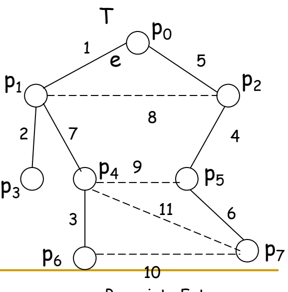
- Suppose there are two distinct minimum-weight spanning trees, T and T', and let e be the minimum-weight edge that appears in only one of the two trees. Suppose wlog that $e \in T$.
- Then the graph T'' obtained by adding e to T' contains a cycle, and at least one other edge in that cycle, e', is not 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

It can be proved that:

- 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.

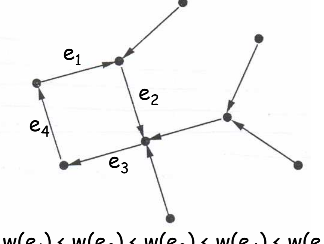
Minimum Spanning Tree

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'* represent 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.



$w(e_1) < w(e_2) < w(e_3) < w(e_4) < w(e_1)$

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 should 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: Minimum Spanning Tree

Assumptions

- The edge weights are unique.
- Processes do not know n or diam.
- The processes are initially quiescent 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 2^k 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

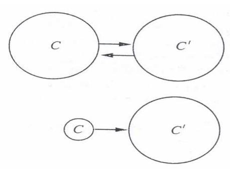
- The initial components are just the individual nodes. Each component has a distinguished leader node and a spanning tree that is a subgraph of the MST.
- Within any component, the processes cooperate in an algorithm to find the MWOE for the entire component:
  - the leader initiate a broadcast
  - 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 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. 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 there 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

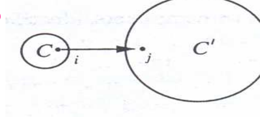
- How does a process p_i determines if a neighboring process p_j is outgoing from p_i's component?
- If process p_j's current component identifier is the same as that of p_i, then process p_i is certain that p_j is in the same component as itself.
- If these ids are different:
  - If p_j's latest known 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 ⇒ absorb is possible
  - Otherwise, there is a cycle of length 2 in G'. Thus, two of these components have the same MWOE ⇒ 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, $e \neq (i,j)$ (since e leads to a component with a level at least as large as that of C′) $\Rightarrow$ 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 $\Rightarrow$ 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: $O(|E| + n\log n)$

- $O(|E|)$: 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: $\sum_{C} |C| = \sum_{k: 0 \leq k \leq \log n} (\sum_{C: level(C) = k} |C|) = \sum_{0}^{\log n} n = n \log n$

Time Complexity: $O(n\log n)$