6.852: Distributed Algorithms
Fall, 2009

Class 9

Today’s plan

- Basic asynchronous network algorithms
  - Constructing a spanning tree
  - Breadth-first search
  - Shortest paths
  - Minimum spanning tree
- Reading: Sections 15.3-15.5, [Gallager, Humblet, Spira]

- Next lecture:
  - Synchronizers
  - Reading: Chapter 16.

Last time

- Formal model for asynchronous networks.
- Leader election algorithms for asynchronous ring networks (LCR, HS, Peterson).
- Lower bound for leader election in an asynchronous ring.
- Leader election in general asynchronous networks (didn't quite get there).

Leader election in general networks

- Undirected graphs.
- Can get asynchronous version of synchronous FloodMax algorithm:
  - Simulate rounds with counters.
  - Need to know diameter for termination.
- We’ll see better asynchronous algorithms later:
  - Don’t need to know diameter.
  - Lower message complexity.
- Depend on techniques such as:
  - Breadth-first search
  - Convergecast using a spanning tree
  - Synchronizers to simulate synchronous algorithms
  - Consistent global snapshots to detect termination

Spanning trees and searching

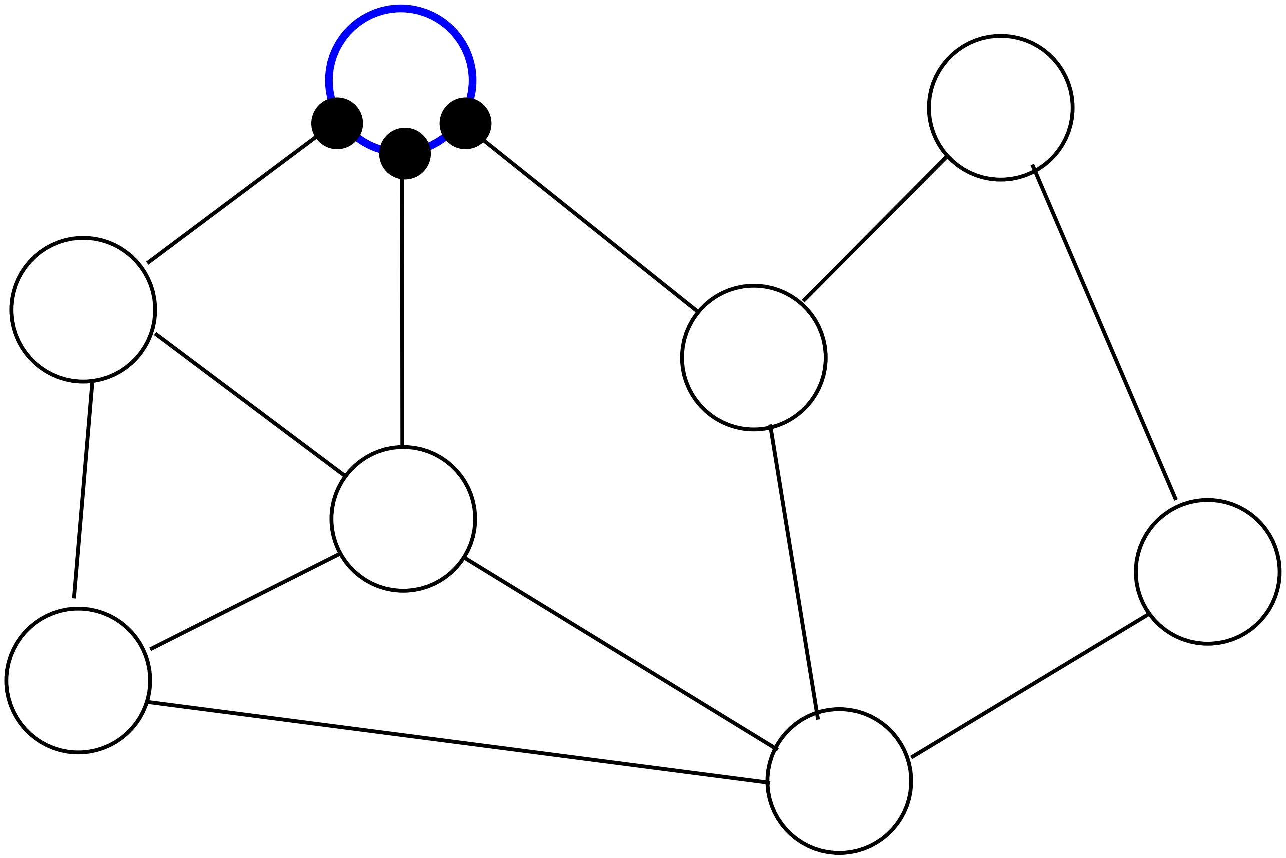
- Spanning trees are used for communication, e.g., broadcast/convergecast
- Start with the simple task of setting up some (arbitrary) spanning tree with a (given) root i_0.
- Assume:
  - Undirected, connected graph (i.e., bidirectional communication).
  - Root i_0
  - Size and diameter unknown.
  - UIDs, with comparisons.
  - Can identify in- and out-edges to same neighbor.
- Require: Each process should output its parent in tree, with a parent output action.
- Starting point: SynchBFS algorithm:
  - i_0 floods search message; parent of a node is the first node from which it receives a search message.
  - Try running the same algorithm in asynchronous network.
  - Still yields spanning tree, but not necessarily breadth-first tree.

AsynchSpanningTree, Process i

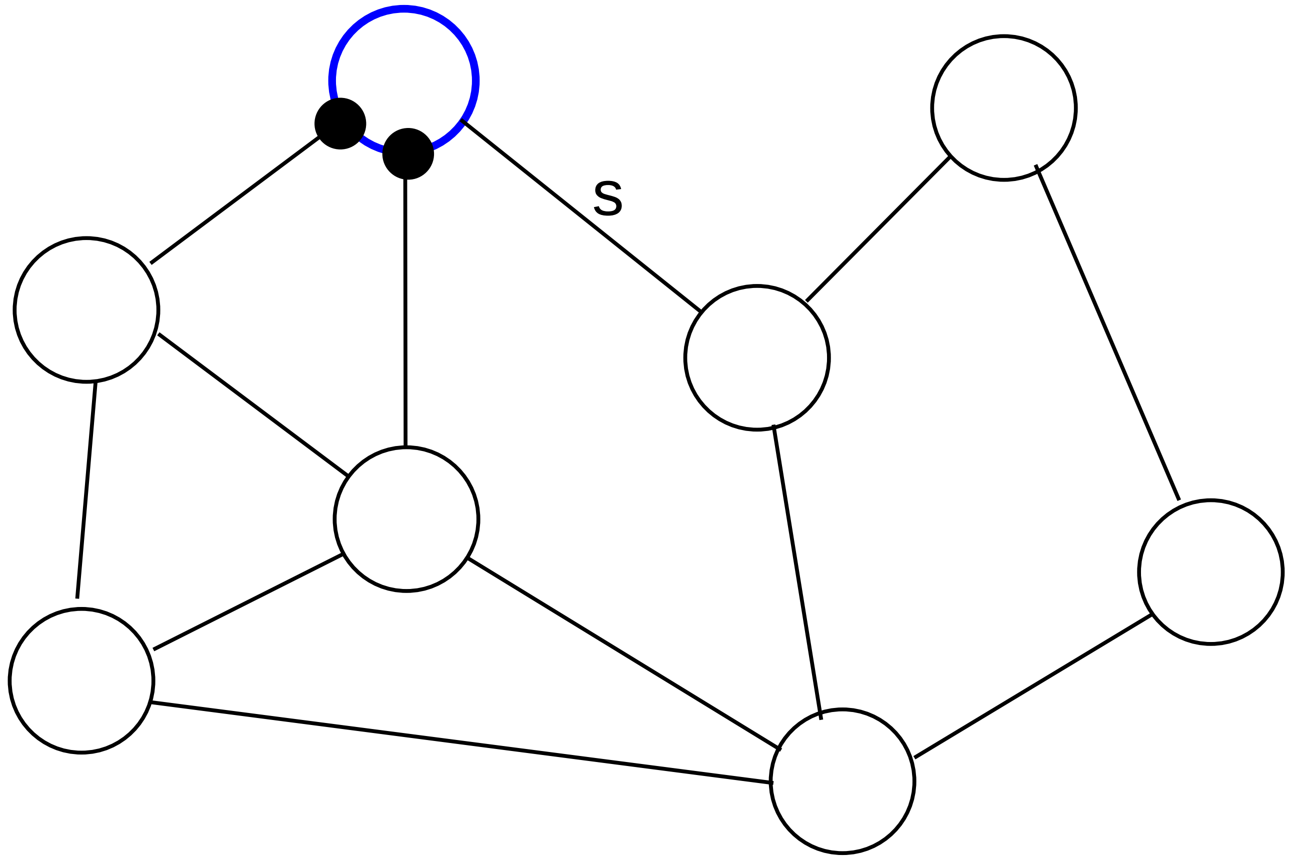
- Signature
  - ***in*** receive(“search”)$_{j,i}$, j ∈ nbrs
  - ***out*** send(“search”)$_{i,j}$, j ∈ nbrs
  - ***out*** parent(j)$_i$, j ∈ nbrs
- State
  - **parent**: nbrs U { null }, init null
  - **reported**: Boolean, init false
  - for each j ∈ nbrs:
    - **send**(j) ∈ { search, null }, init search if i = i_0, else null

- send(“search”)$_{i,j}$
  pre: **send**(j) = search
  eff: **send**(j) := null
- receive(“search”)$_{j,i}$
  eff: if i ≠ i_0 and **parent** = null then
  parent := j
  for k ∈ nbrs - { j } do
  send(k) := search
- parent(j)$_i$
  pre: **parent** = j
  reported = false
  eff: **reported** := true

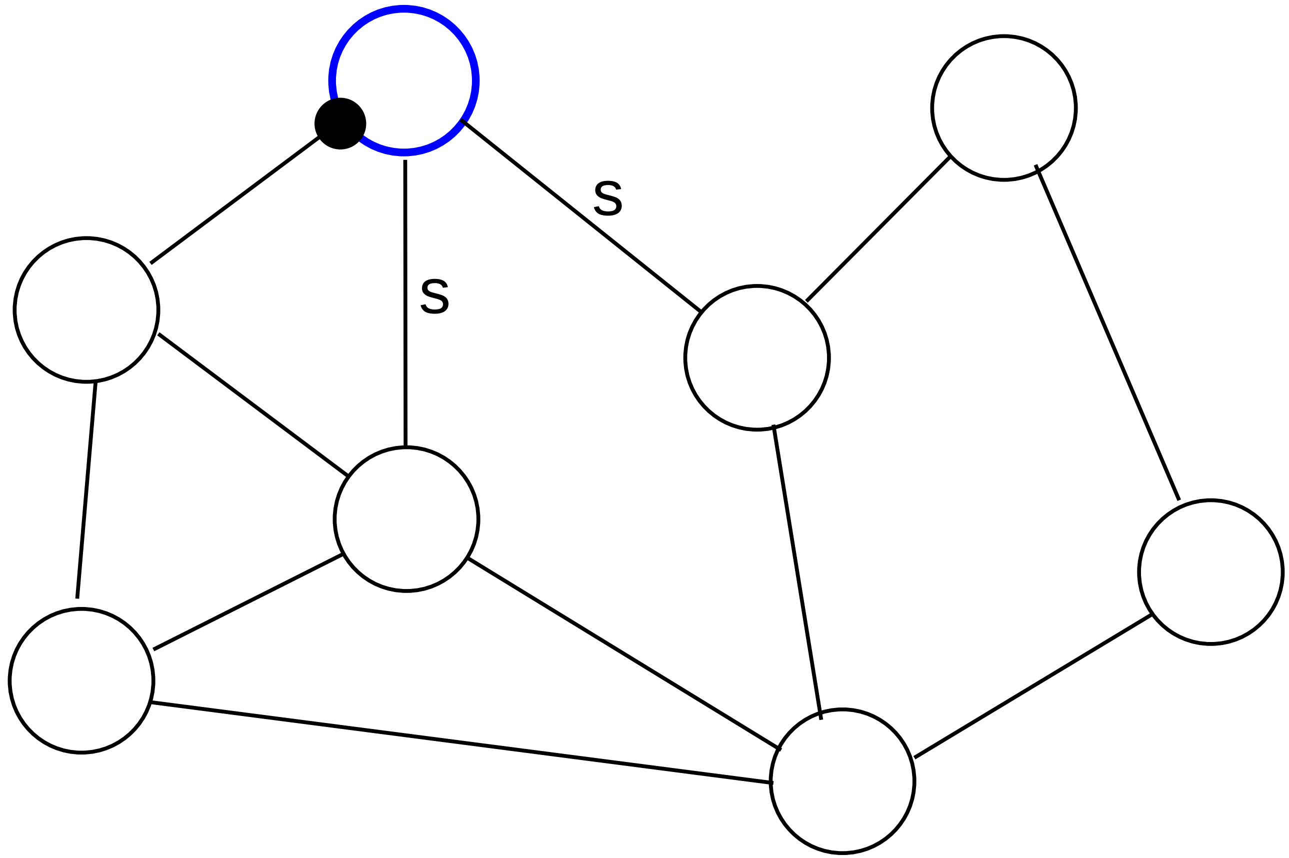
AsynchSpanningTree



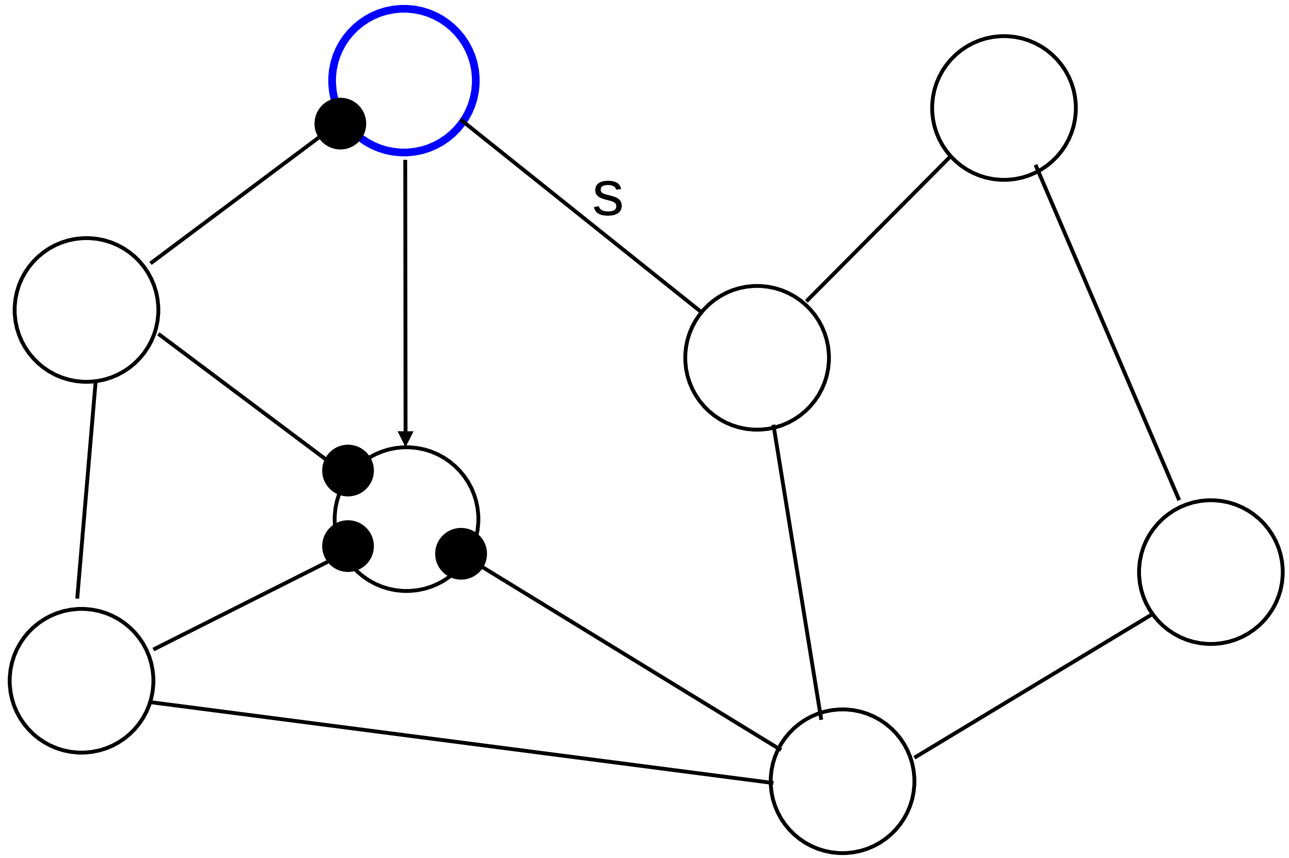
AsynchSpanningTree



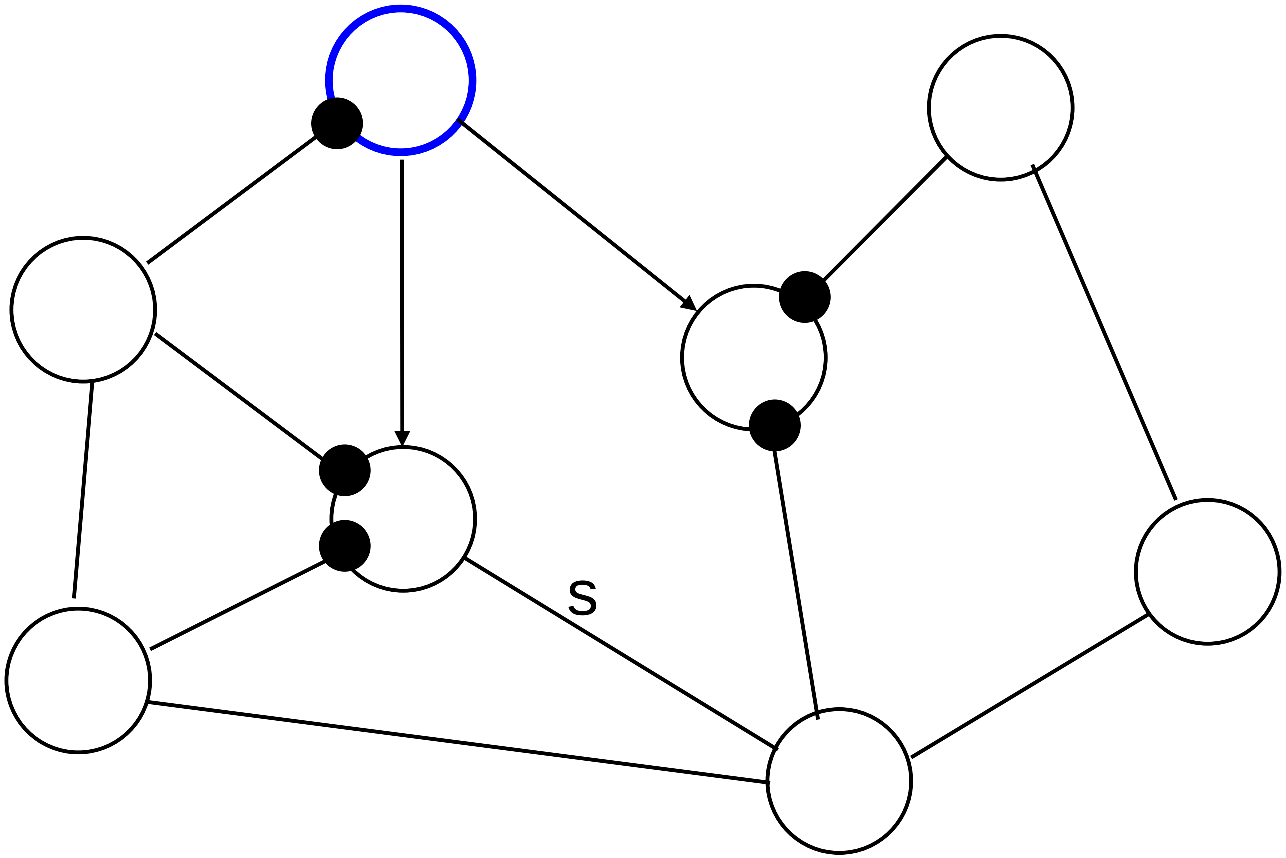
AsynchSpanningTree



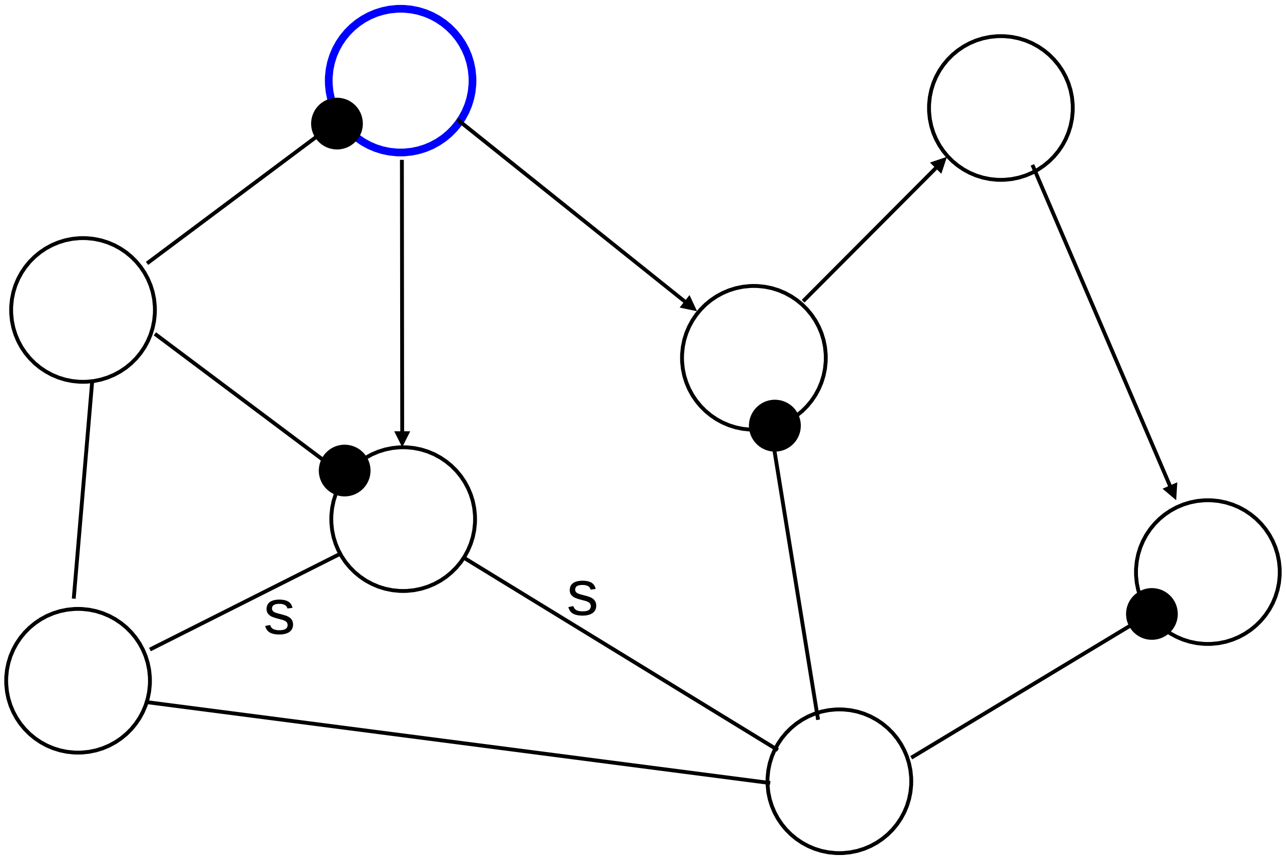
AsynchSpanningTree



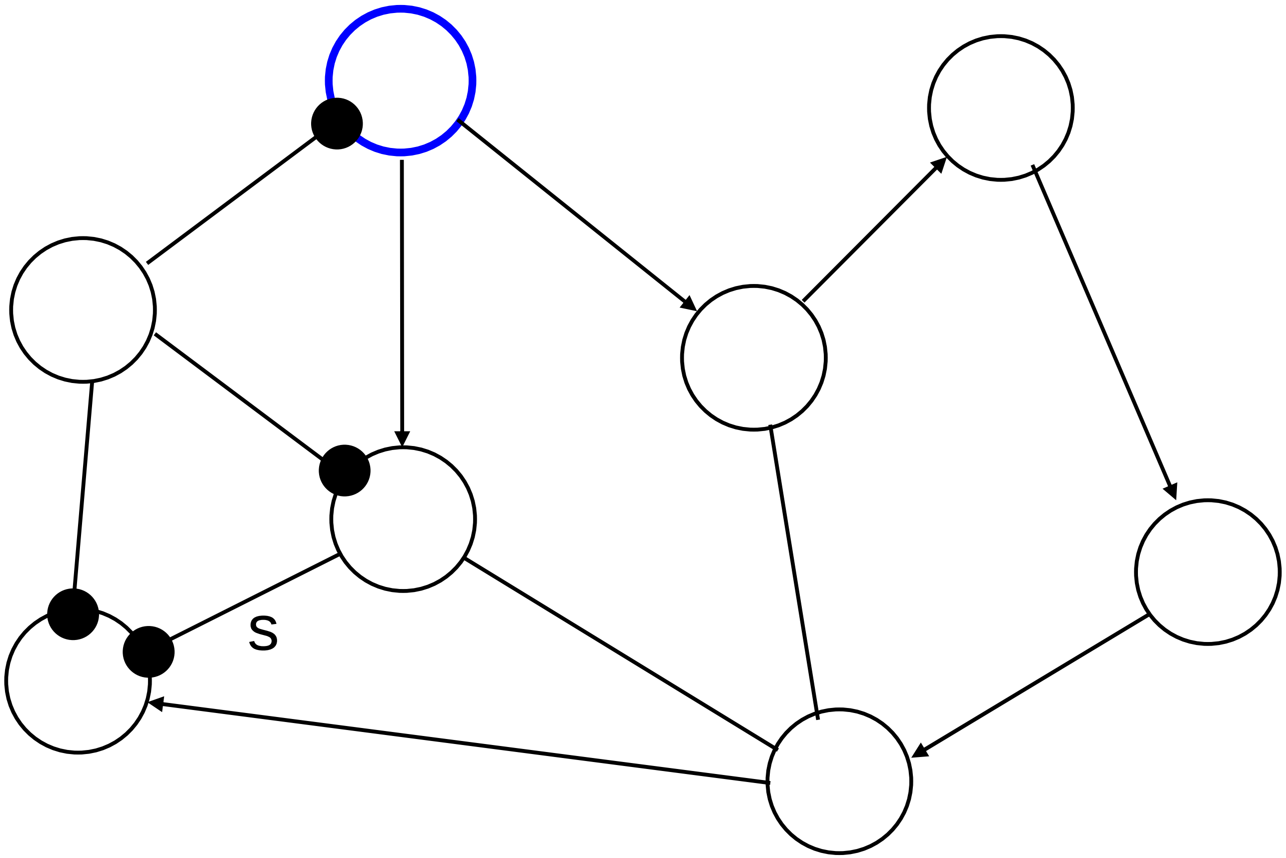
AsynchSpanningTree



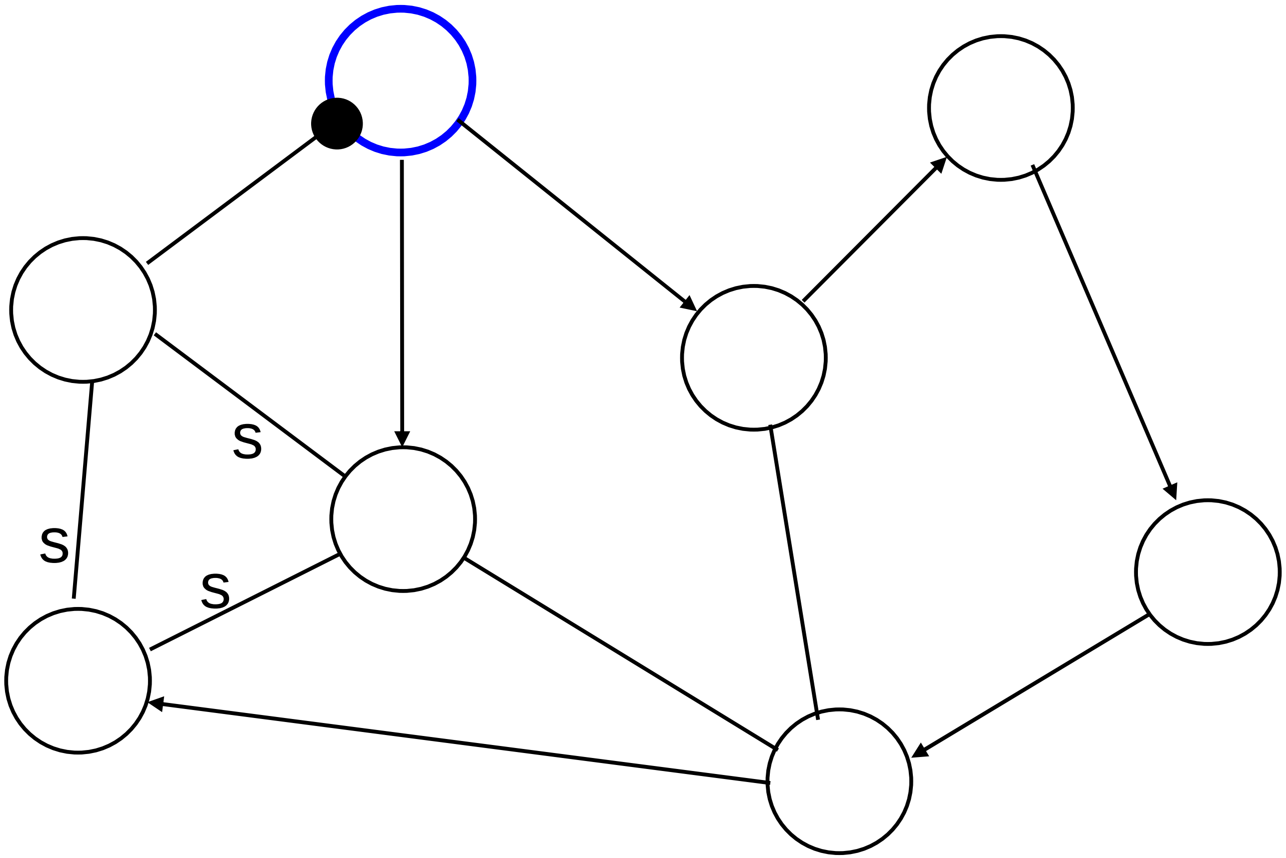
AsynchSpanningTree



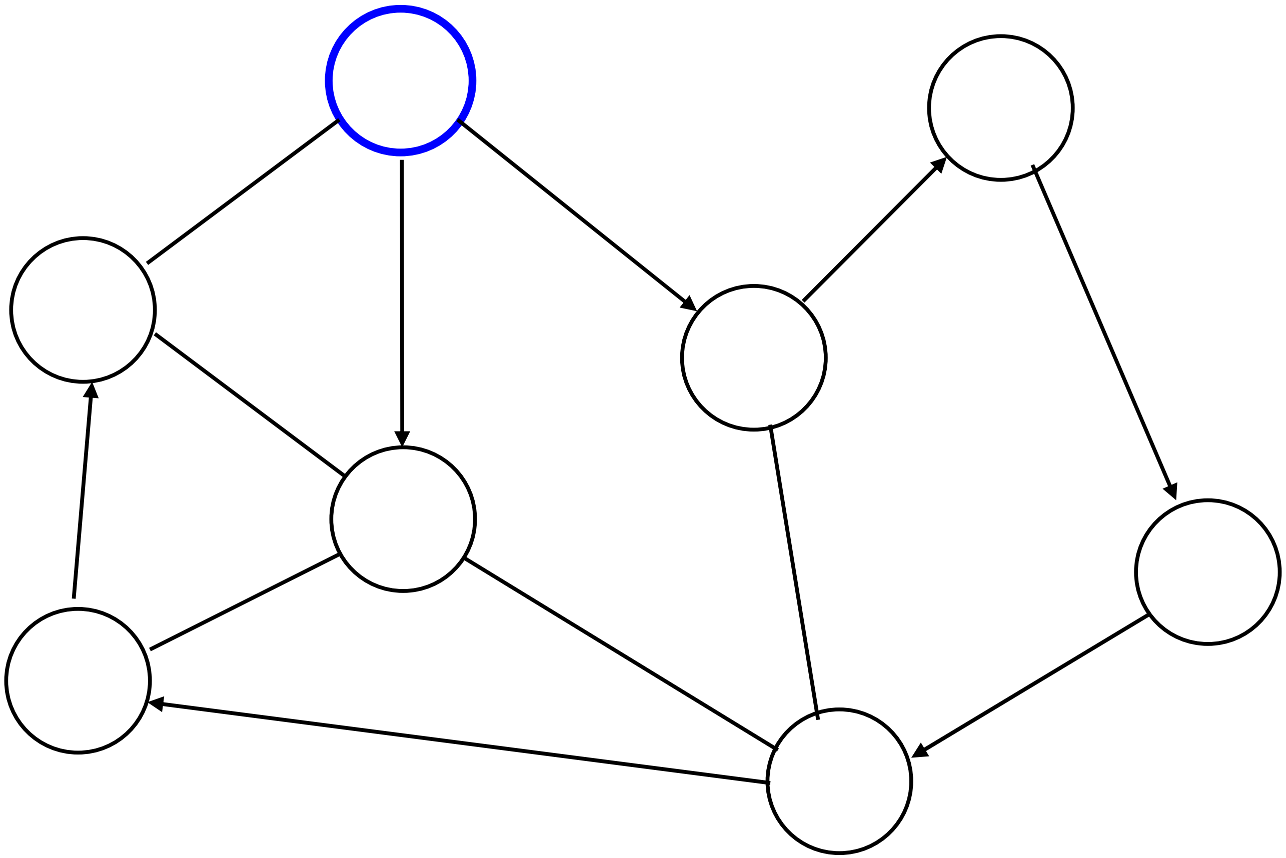
AsynchSpanningTree



AsynchSpanningTree

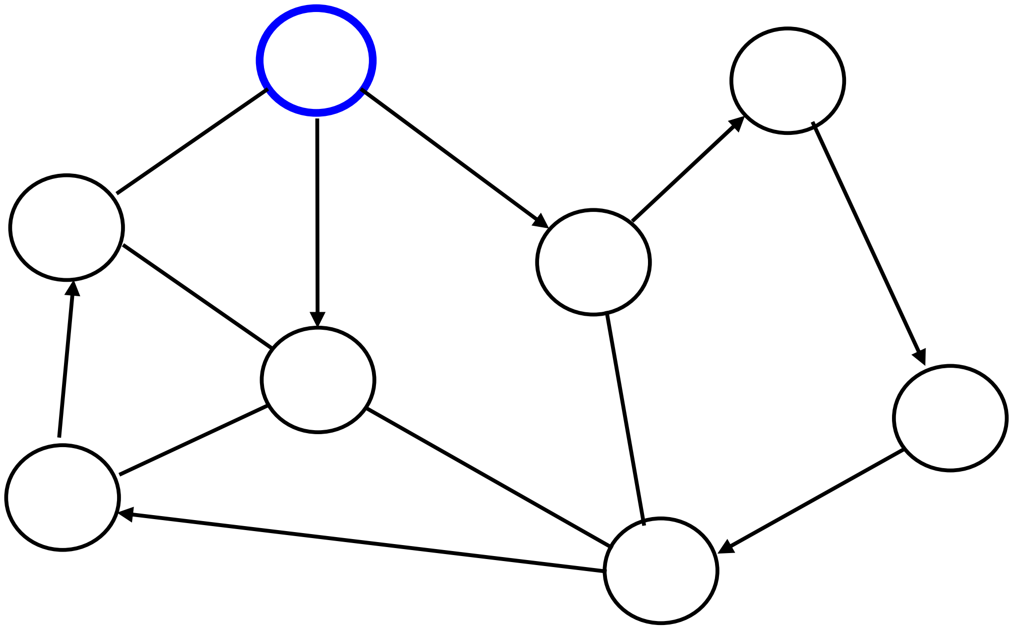


AsynchSpanningTree



AsynchSpanningTree

- Complexity
  - Messages: O(|E|)
  - Time: diam (l+d) + l
- Anomaly: Paths may be longer than diameter!
  - Messages may travel faster along longer paths, in asynchronous networks.



Applications of AsynchSpanningTree

- Similar to synchronous BFS
- Message broadcast: Piggyback on search message.
- Child pointers: Add responses to search messages, easy because of bidirectional communication.
- Use precomputed tree for bcast/convergecast
  - Now the timing anomaly arises.
  - O(h(l+d)) time complexity.
  - O(|E|) message complexity.
  - See book for details.

h = height of tree; may be n

More applications

- Asynchronous broadcast/convergecast:
  - Can also construct spanning tree while using it to broadcast message and also to collect responses.
  - E.g., to tell the root when the bcast is done, or to collect aggregated data.
  - See book, p. 499-500.
  - Complexity:
    - O(|E|) message complexity.
    - O(n (l+d)) time complexity, timing anomaly.
    - See book for details.
- Elect leader when nodes have no info about the network (no knowledge of n, diam, etc.; no root, no spanning tree):
  - All independently initiate AsynchBcastAck, use it to determine max, max elects itself.

Breadth-first spanning tree

- Assume (same as above):
  - Undirected, connected graph (i.e., bidirectional communication).
  - Root i_0.
  - Size and diameter unknown.
  - UIDs, with comparisons.
- Require: Each process should output its parent in a breadth-first spanning tree.
- In asynchronous networks, modified SynchBFS does not guarantee that the spanning tree constructed is breadth-first.
  - Long paths may be traversed faster than short ones.
- Can modify each process to keep track of distance, change parent when it hears of shorter path.
  - Relaxation algorithm (like Bellman-Ford).
  - Must inform neighbors of changes.
  - Eventually, tree stabilizes to a breadth-first spanning tree.

AsynchBFS

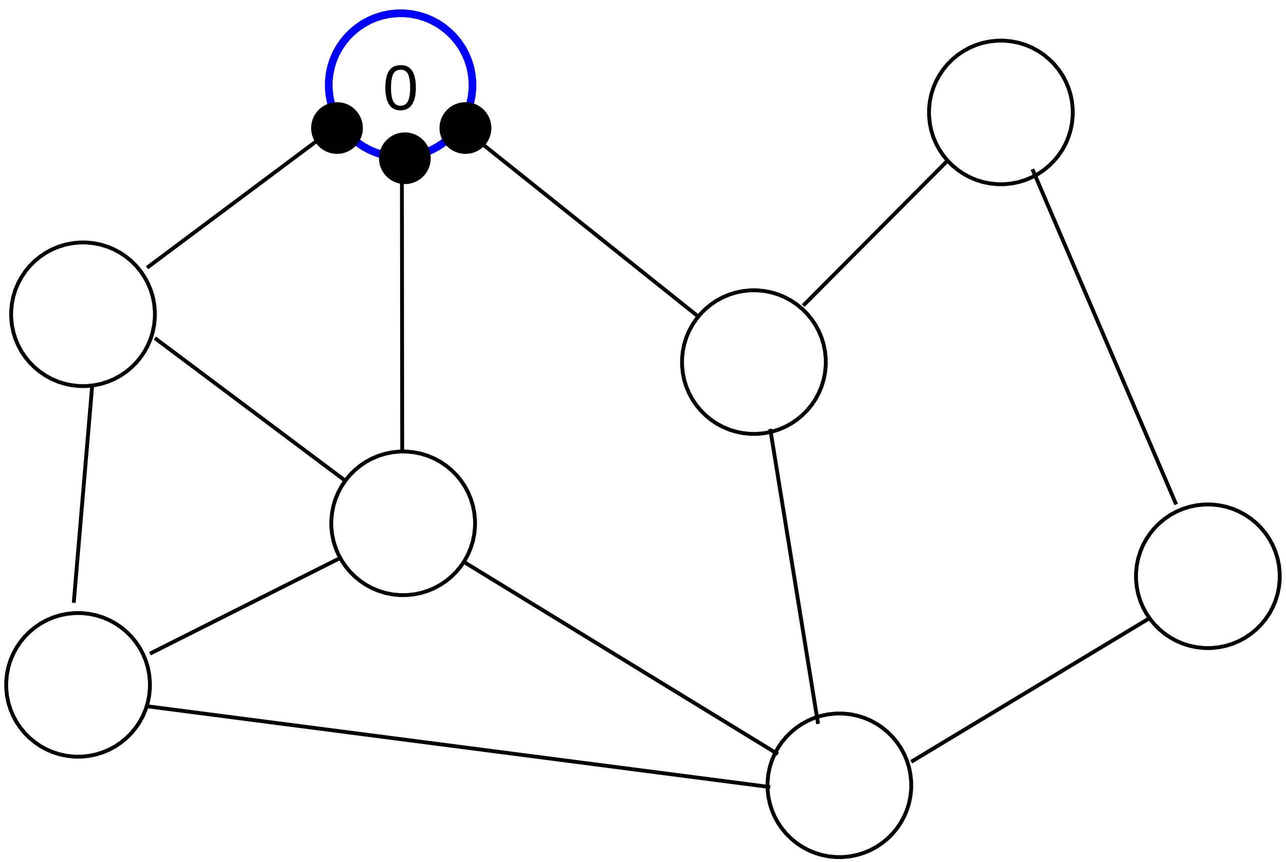
- Signature
  - ***in*** receive(m)$_{j,i}$, m ∈ **N,** j ∈ nbrs
  - ***out*** send(m)$_{i,j}$, m ∈ **N**, j ∈ nbrs
- State
  - **dist**: **N** U { ∞ }, init 0 if i = i_0, else ∞
  - **parent**: nbrs U { null }, init null
  - for each j ∈ nbrs:
    - **send**(j): FIFO queue of **N**, init (0) if i = i_0, else ∅

- send(m)$_{i,j}$
  pre: m = head(**send**(j))
  eff: remove head of **send**(j)

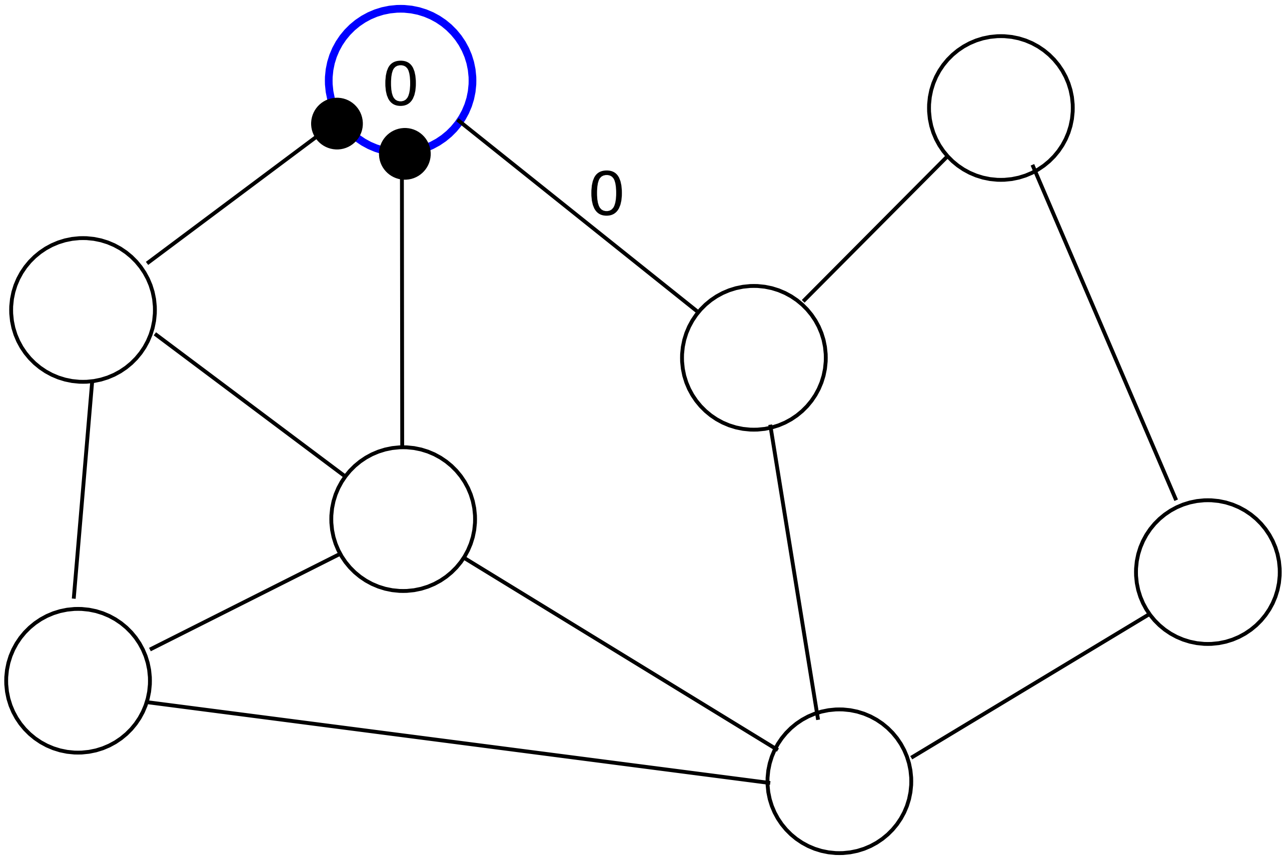
- receive(m)$_{j,i}$
  eff: if m+1 < **dist** then
  dist := m+1
  parent := j
  for k ∈ nbrs - { j } do
  add **dist** to **send**(k)

Note: No parent actions---no one knows when the algorithm is done

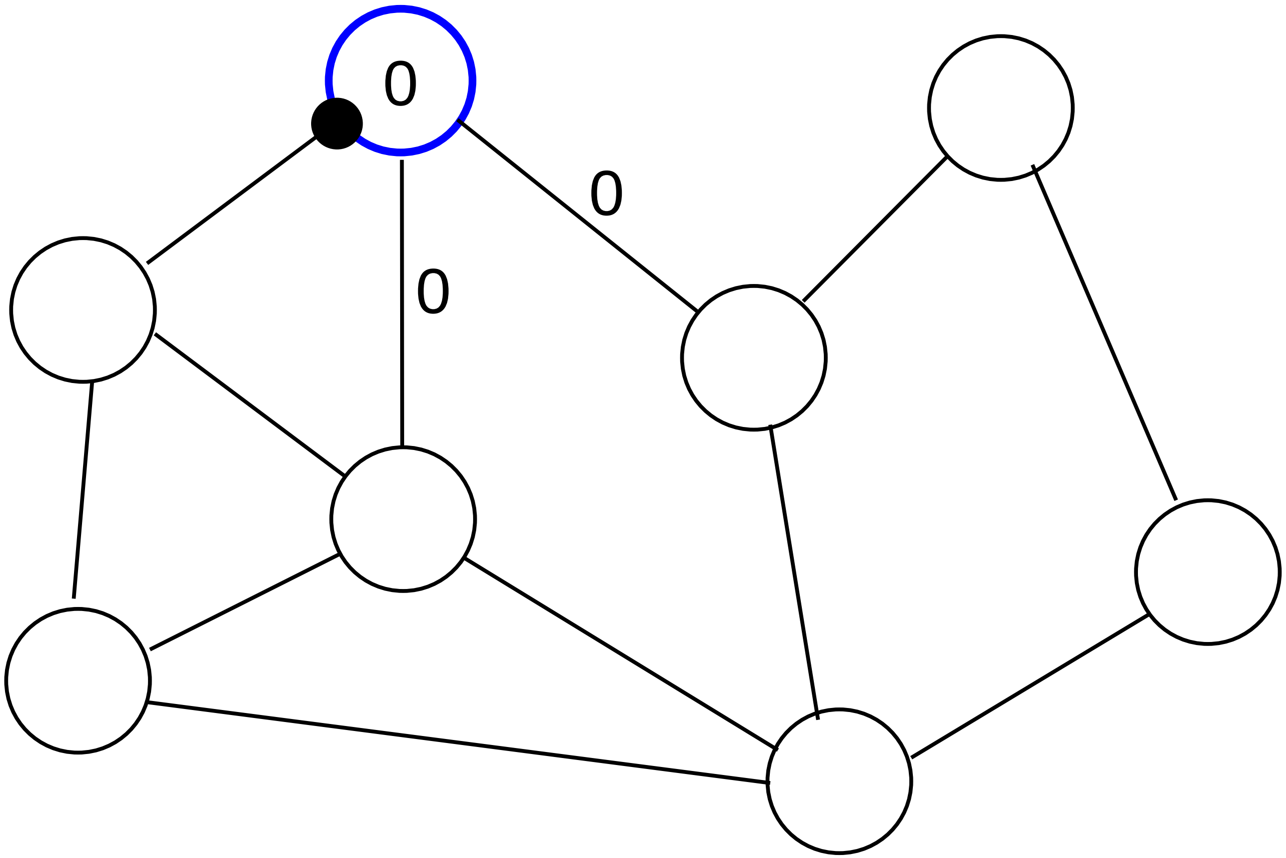
AsynchBFS



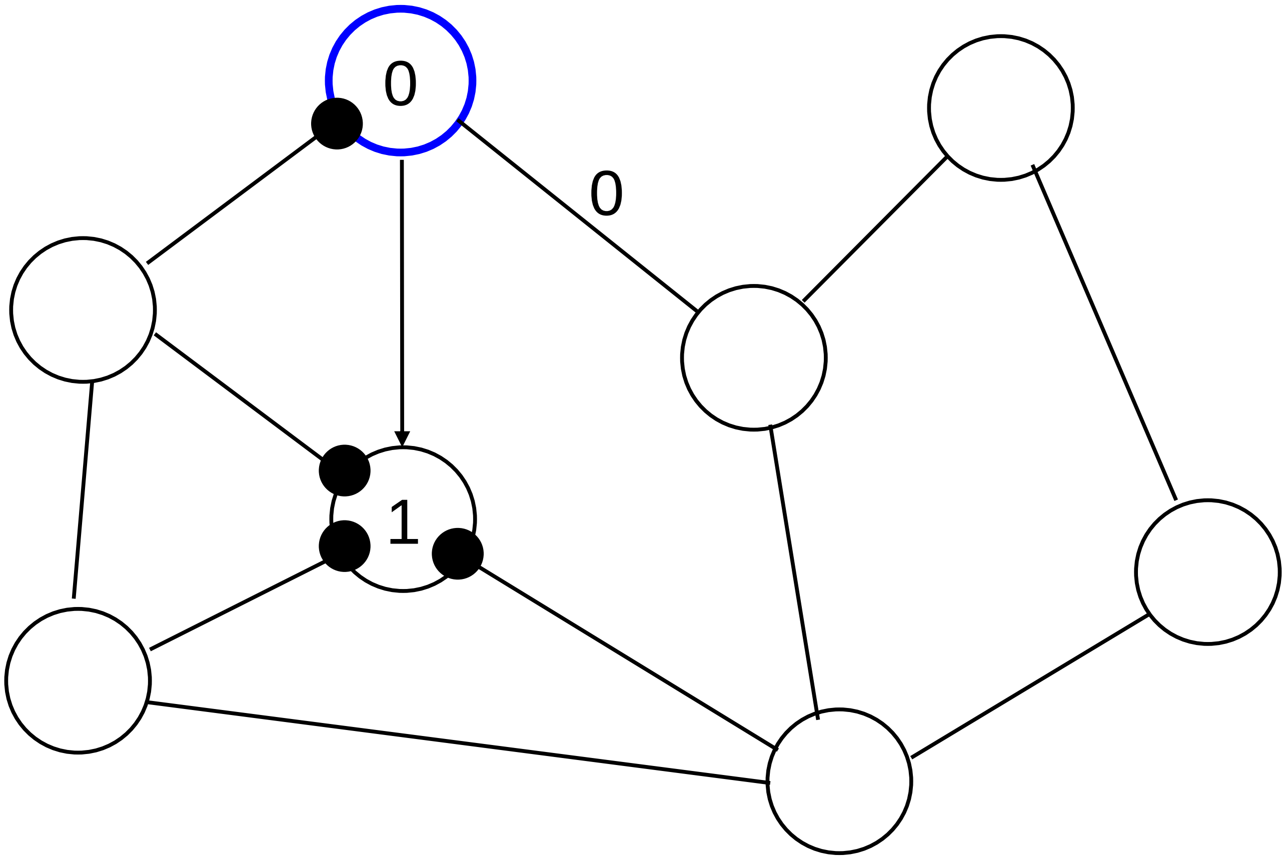
AsynchBFS



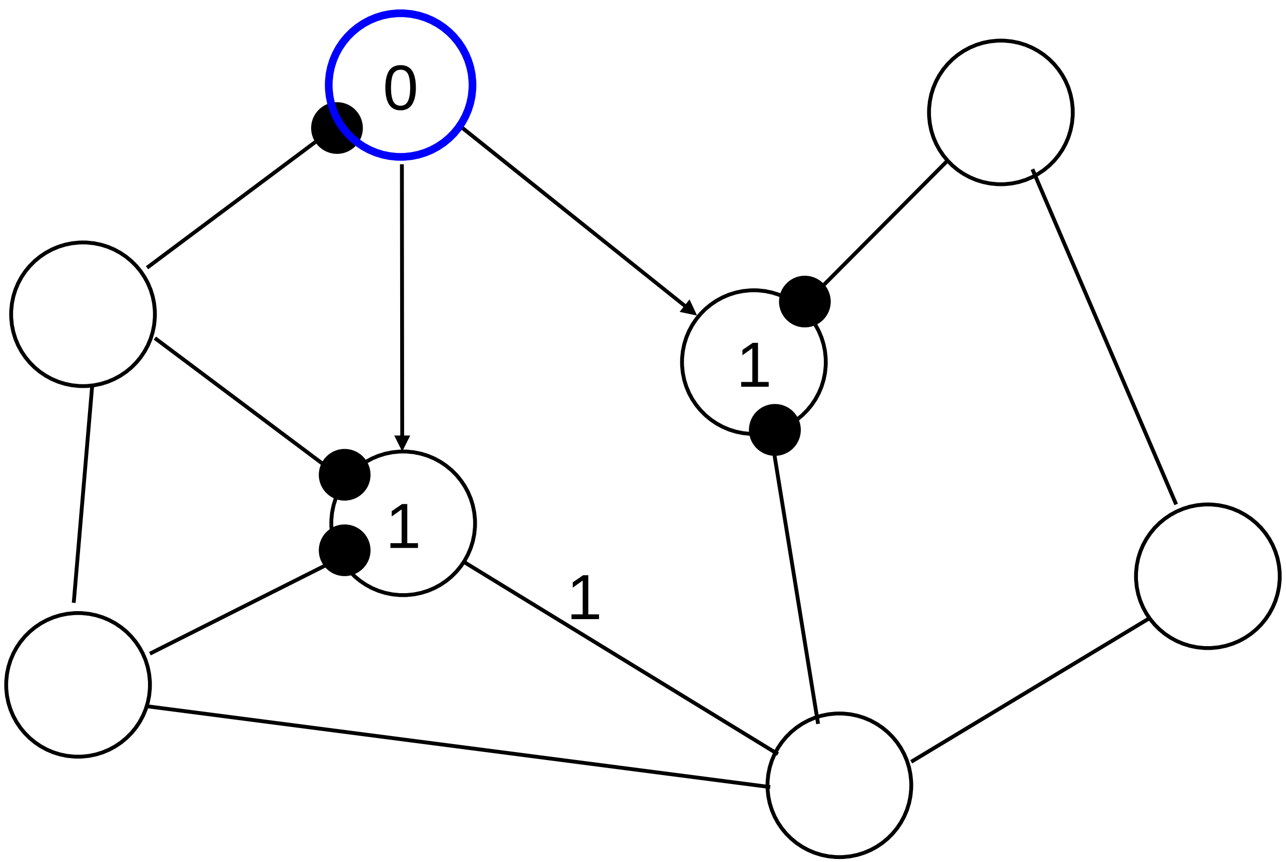
AsynchBFS



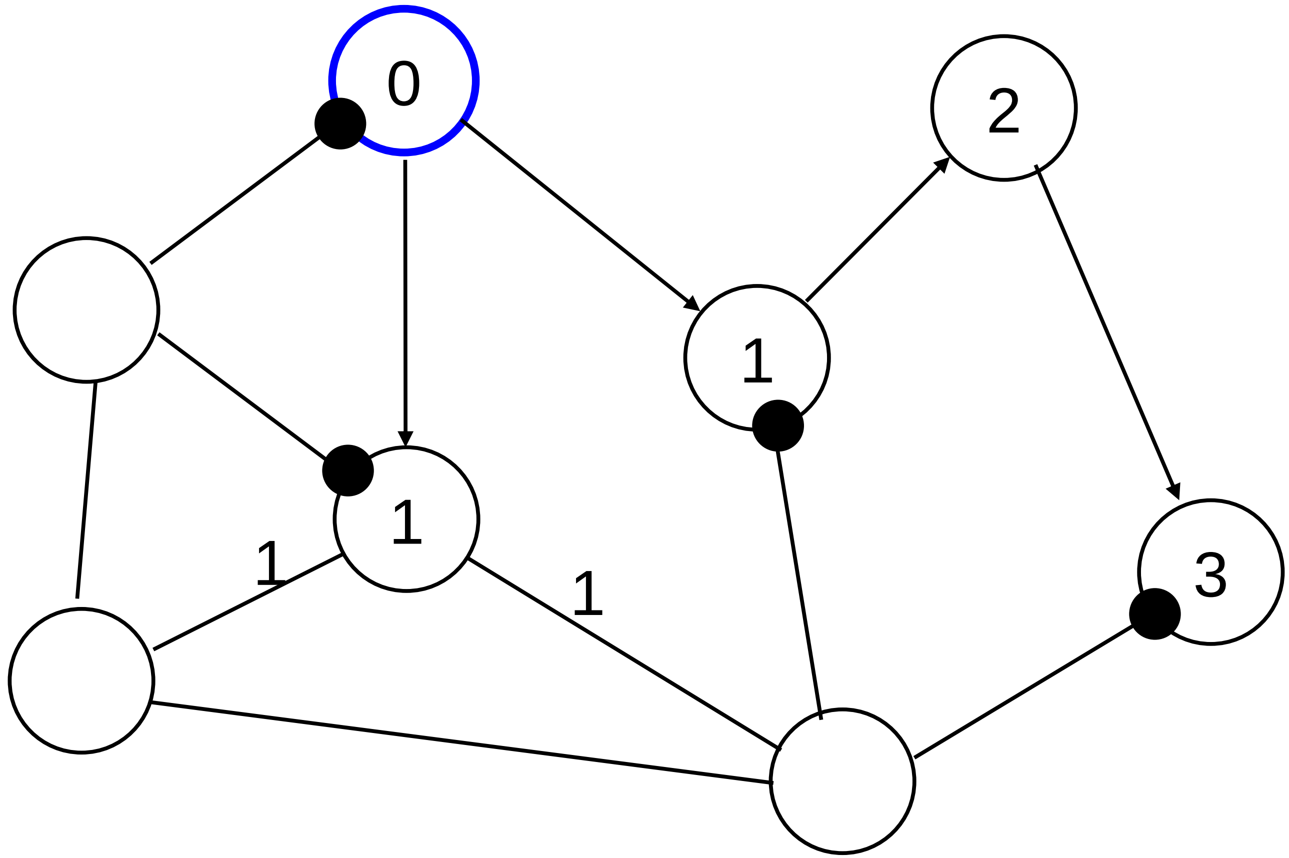
AsynchBFS



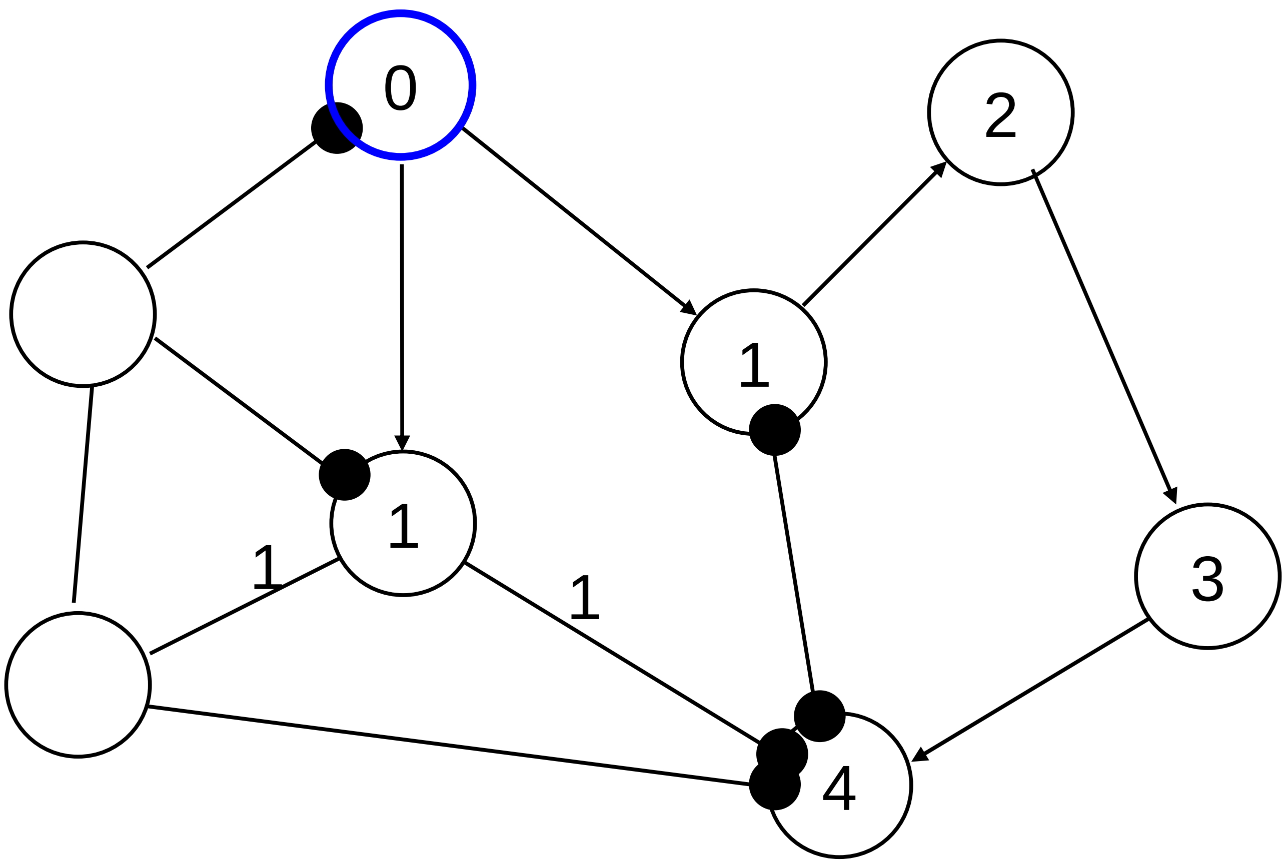
AsynchBFS



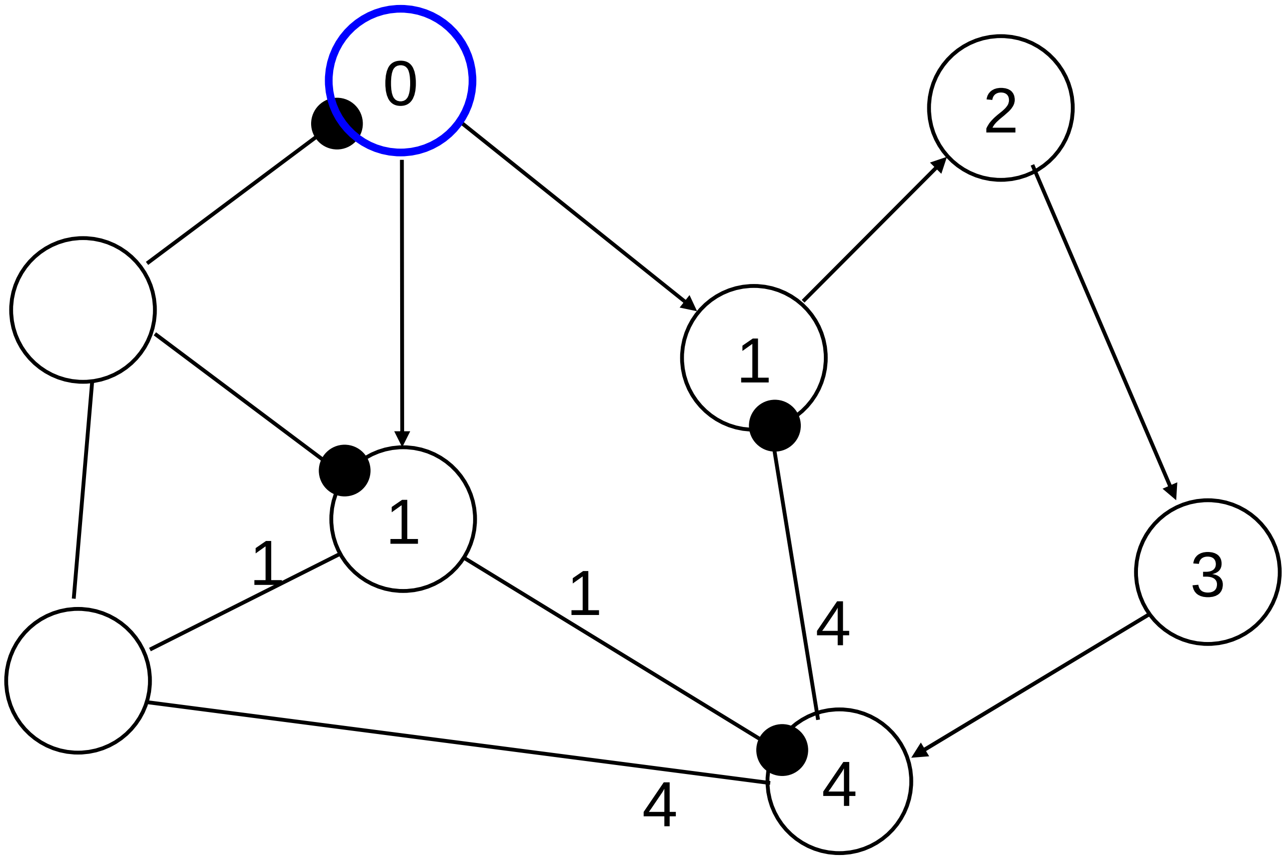
AsynchBFS



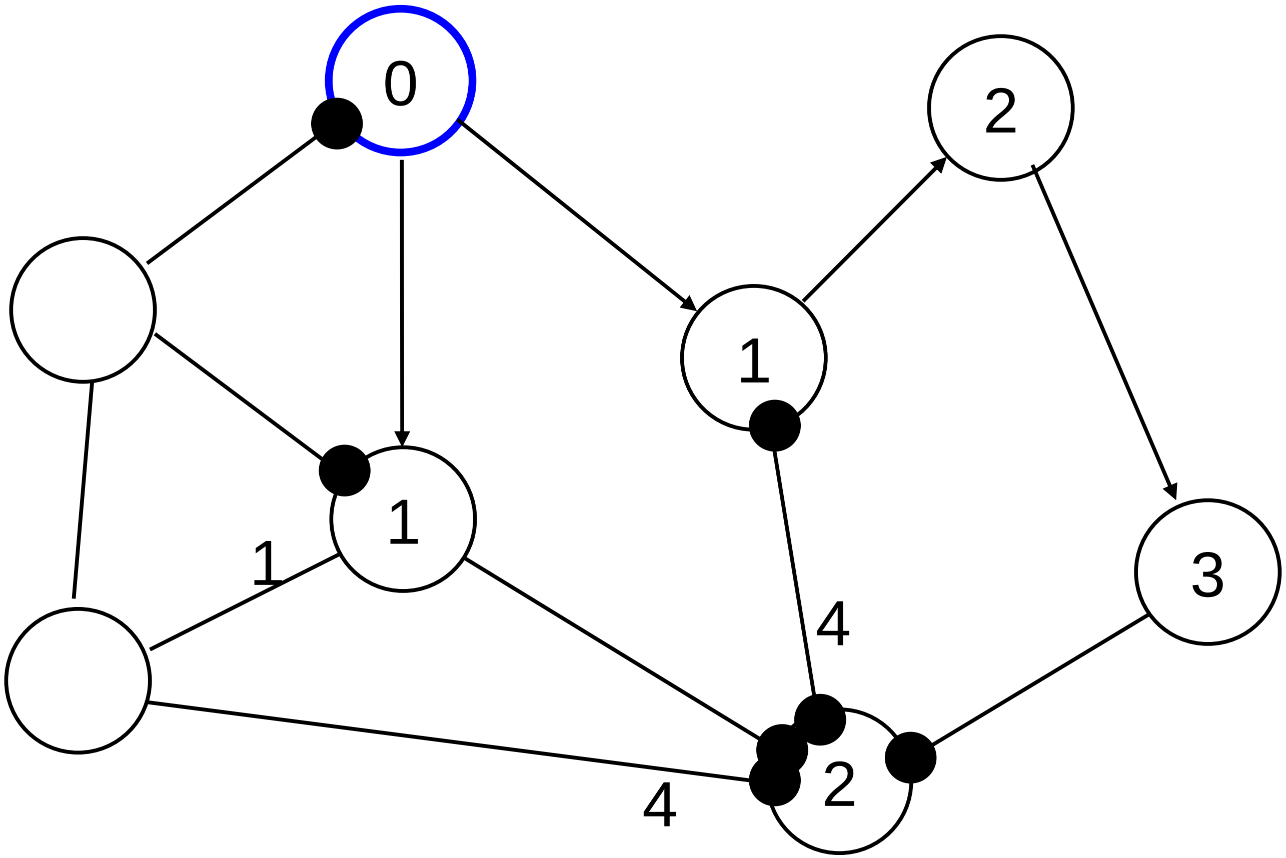
AsynchBFS



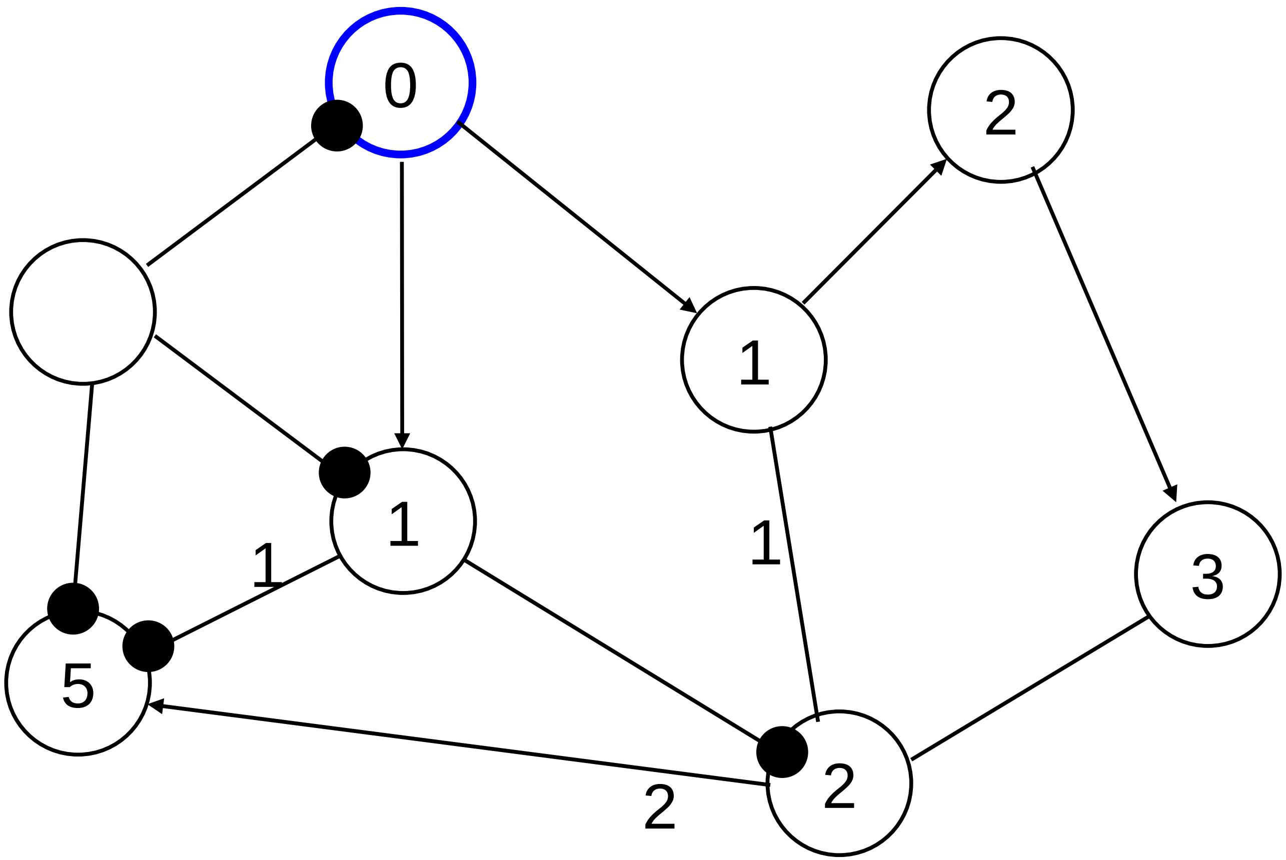
AsynchBFS



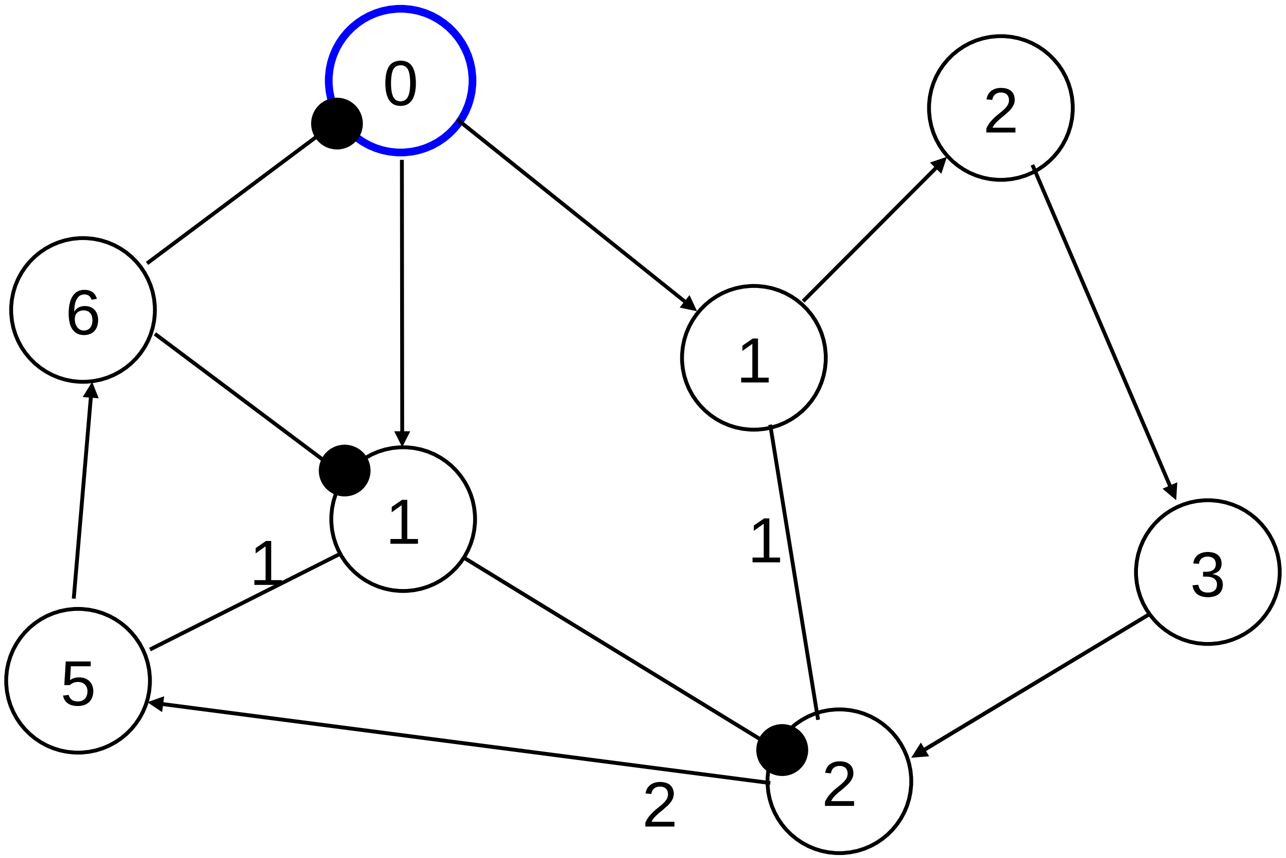
AsynchBFS



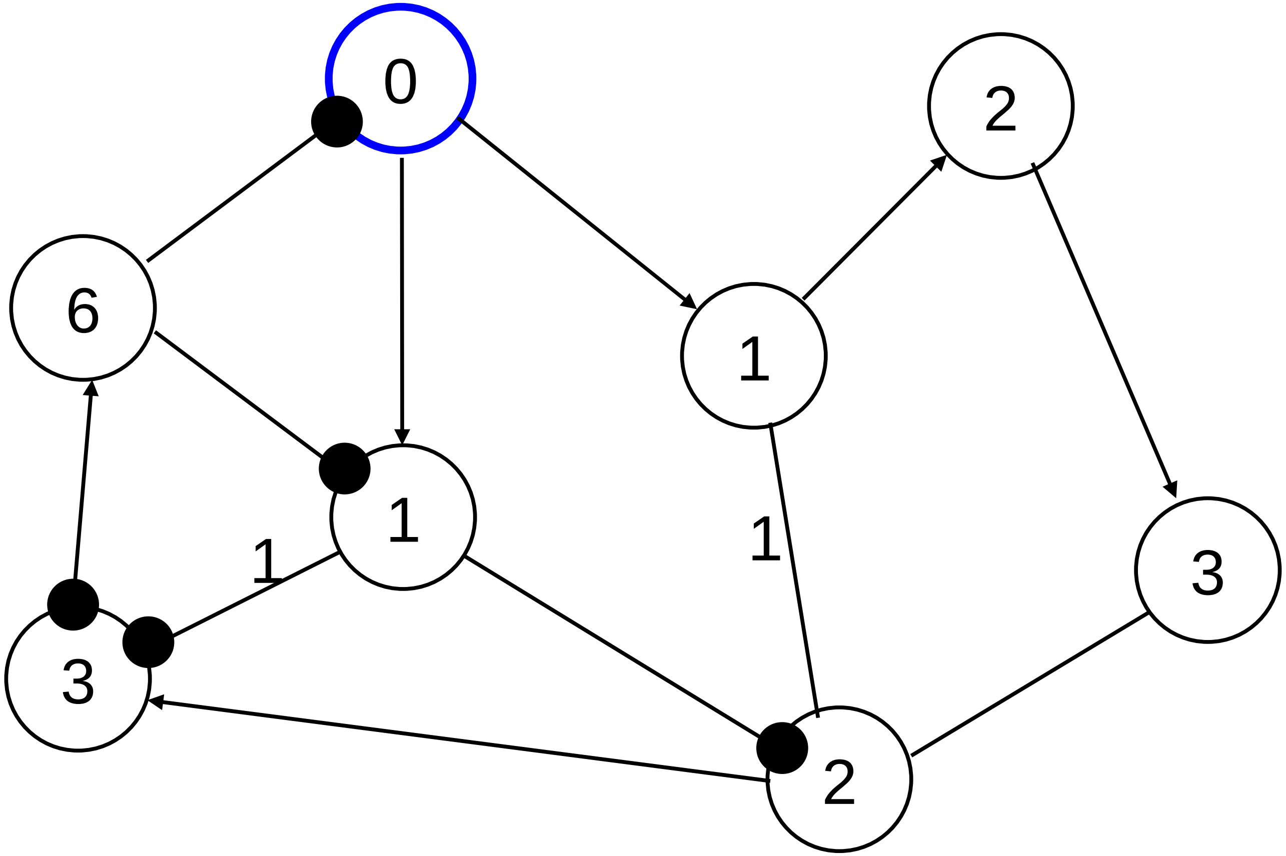
AsynchBFS



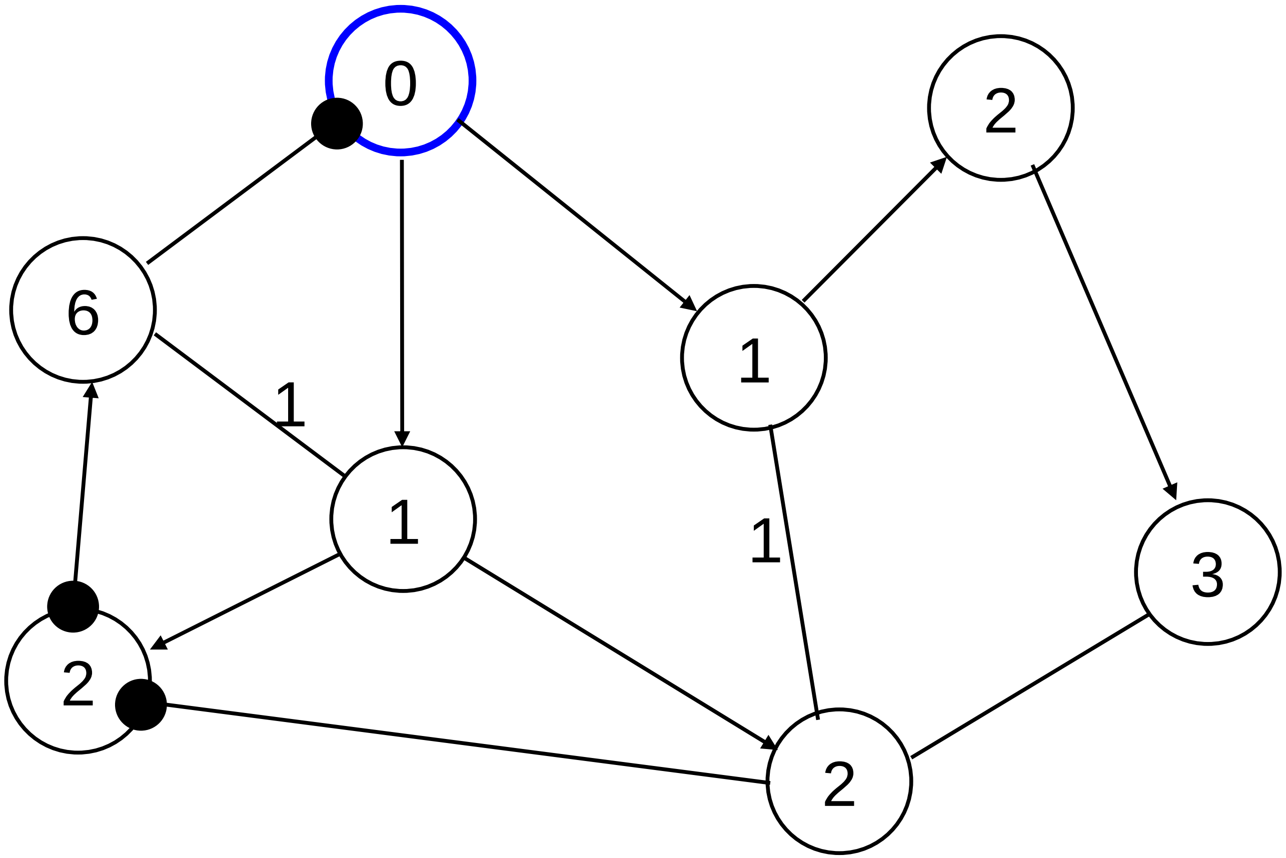
AsynchBFS



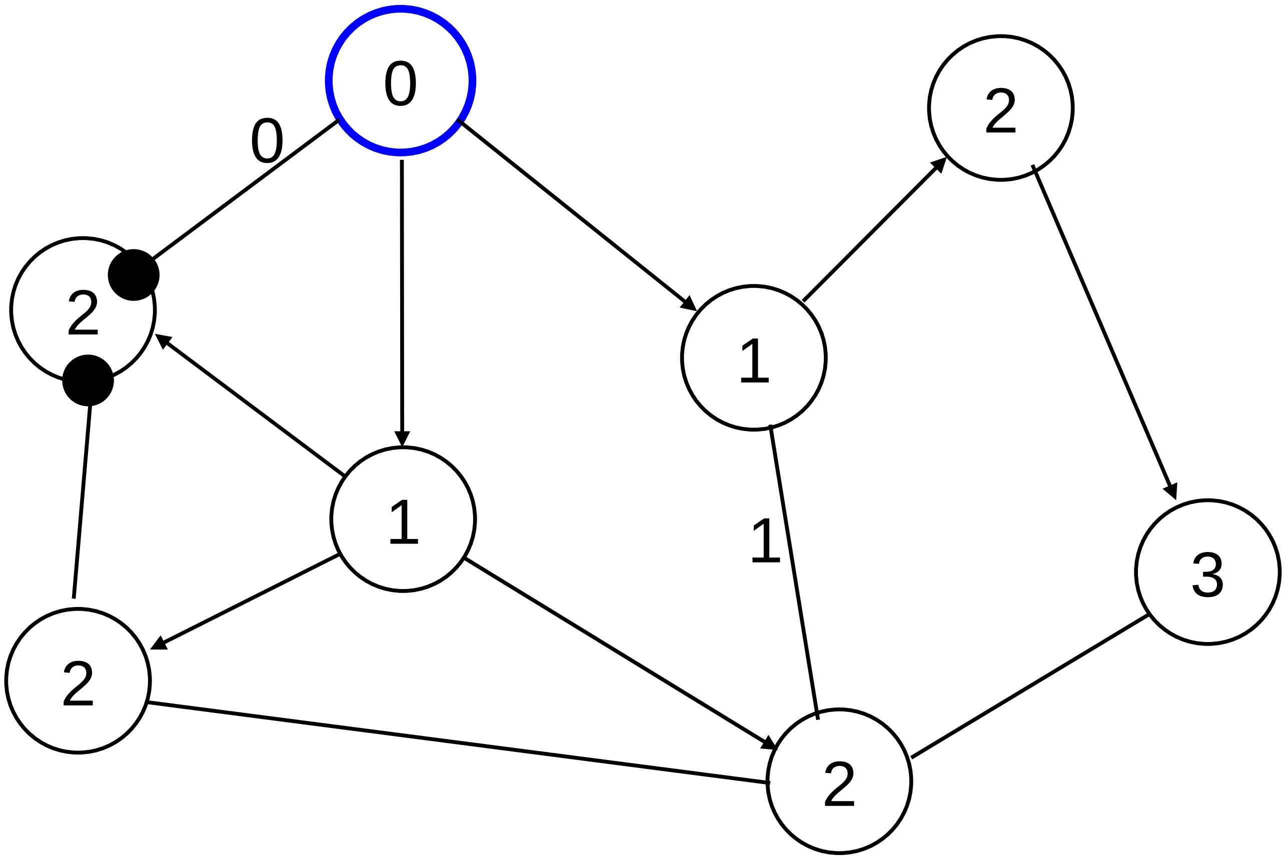
AsynchBFS



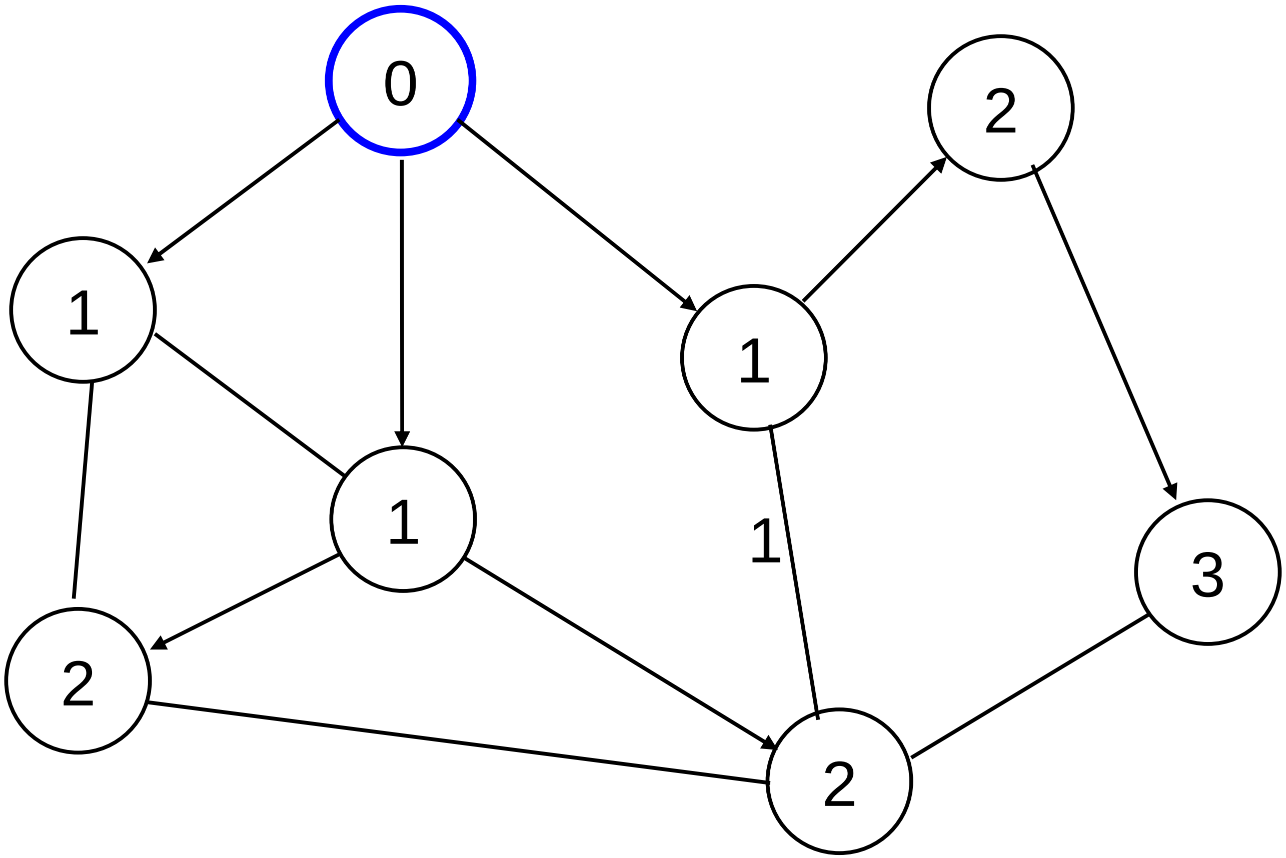
AsynchBFS



AsynchBFS



AsynchBFS



AsynchBFS

- Complexity:
  - Messages: O(n |E|)
    - May send O(n) messages on each link (one for each distance estimate).
  - Time: O(diam n (l+d)) (taking pileups into account).
  - Can reduce complexity if know bound D on diameter:
    - Allow only distance estimates ≤ D.
    - Messages: O(D |E|); Time: O(diam D (l+d))

- Termination:
  - No one knows when this is done, so can't produce parent outputs.
  - Can augment with acks for search messages, convergecast back to i_0.
  - i_0 learns when the tree has stabilized, tells everyone else.
  - A bit tricky:
    - Tree grows and shrinks.
    - Some processes may participate many times, as they learn improvements.
    - Bookkeeping needed.
    - Complexity?

Layered BFS

- Asynchrony leads to many corrections, which lead to lots of communication.
- Idea: Slow down communication, grow the tree in synchronized phases.
  - In phase k, incorporate all nodes at distance k from i_0.
  - i_0 synchronizes between incorporating nodes at distance k and k+1.
- Phase 1:
  - i_0 sends search messages to neighbors.
  - Neighbors set dist := 1, send acks to i_0.
- Phase k+1:
  - Assume phases 1,…,k are completed: each node at distance ≤ k knows its parent, and each node at distance ≤ k-1 also knows its children.
  - i_0 broadcasts newphase message along tree edges, to distance k processes.
  - Each of these sends search message to all neighbors except its parent.
  - When any non- i_0 process receives first search message, sets parent := sender and sends a positive ack; sends nacks for subsequent search msgs.
  - When distance k process receives acks/nacks for all its search messages, designates nodes that sent postive acks as its children.
  - Then distance k processes convergecast back to i_0 along depth k tree to say that they're done; include a bit saying whether new nodes were found.

Layered BFS

- Terminates: When i_0 learns, in some phase, that no new nodes were found.
- Obviously produces BFS tree.
- Complexity:
  - Messages: O(|E| + n diam)

Each edge explored at most once in each direction by search/ack.

Each tree edge traversed at most once in each phase by newphase/convergecast.

  - Time:
    - Use simplified analysis:
      - Neglecting local computation time l
      - Assuming that every message in a channel is delivered in time d (ignoring congestion delays).
    - $O(diam^2\ d)$

LayeredBFS vs AsynchBFS

- Message complexity:
  - AsynchBFS: O(diam |E|), assuming diam is known, O(n |E|) if not
  - LayeredBFS: O(|E| + n diam)
- Time complexity:
  - AsynchBFS: O(diam d)
  - LayeredBFS: O($diam^2$ d)

- Can also define “hybrid” algorithm (in book)
  - Add m layers in each phase.
  - Within each phase, layers constructed asynchronously.
  - Intermediate performance.

Shortest paths

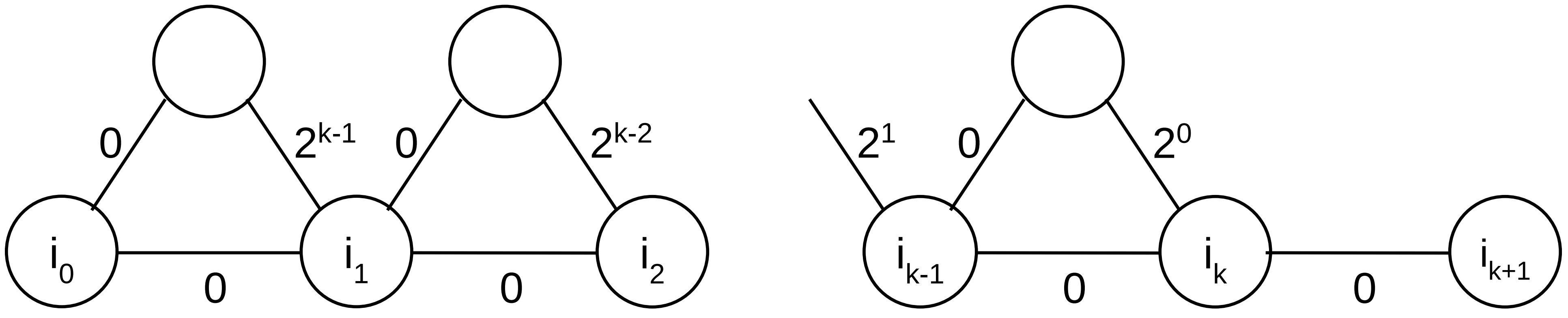
- Assumptions:
  - Same as for BFS, plus edge weights.
  - weight(i,j), nonnegative real, same in both directions.
- Require:
  - Output shortest distance and parent in shortest-paths tree.
- Use Bellman-Ford asynchronously
  - Used to establish routes in ARPANET 1969-1980.
  - Can augment with convergecast as for BFS, for termination.
  - But worst-case complexity is very bad…

AsynchBellmanFord

- Signature
  - ***in*** receive(w)$_{j,i}$, m $\in$ $\mathbf{R}^{\geq 0}$**,** j $\in$ nbrs
  - ***out*** send(w)$_{i,j}$, m $\in$ $\mathbf{R}^{\geq 0}$**,** j $\in$ nbrs
- State
  - **dist**: $\mathbf{R}^{\geq 0}$ U { ∞ }, init 0 if i = i_0, else ∞
  - **parent**: nbrs U { null }, init null
  - for each j $\in$ nbrs:
    - **send**(j): FIFO queue of $\mathbf{R}^{\geq 0}$; init (0) if i = i_0, else empty
- Transitions
  - send(w)$_{i,j}$
    pre: m = head(**send**(j))
    eff: remove head of **send**(j)
  - receive(w)$_{j,i}$
    eff: if w + weight(j,i) < **dist** then
    dist := w + weight(j,i)
    parent := j
    for k $\in$ nbrs - { j } do
    add **dist** to **send**(k)

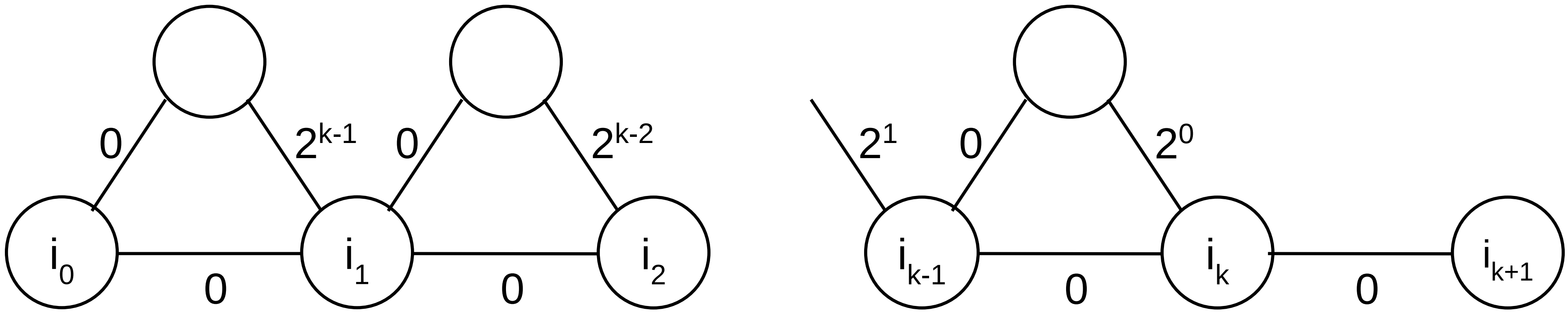
AsynchBellmanFord

- Termination:
  - Use convergecast (as for AsynchBFS).
- Complexity:
  - O(n!) simple paths from i_0 to any other node, which is $O(n^n)$.
  - So the number of messages sent on any channel is $O(n^n)$.
  - So message complexity = $O(n^n |E|)$, time complexity = $O(n^n\, n\, (l+d))$.
  - Q: Are the message and time complexity really exponential in n?
  - A: Yes: In some execution of network below, i_k sends 2^k messages to i_{k+1}, so message complexity is $\Omega(2^{n/2})$ and time complexity is $\Omega(2^{n/2}\, d)$.



Exponential time/message complexity

- i_k sends 2^k messages to i_{k+1}, so message complexity is $\Omega(2^{n/2})$ and time complexity is $\Omega(2^{n/2}\ d)$.
- Possible distance estimates for i_k are $2^k - 1$, $2^k - 2$,…,0.
- Moreover, i_k can take on all these estimates in sequence:
  - First, messages traverse upper links, $2^k - 1$.
  - Then last lower message arrives at i^k, $2^k - 2$.
  - Then lower message i_k-2 $\rightarrow$ i_k-1 arrives, reduces i_k-1's estimate by 2, message i_k-1 $\rightarrow$ i_k arrives on upper links, $2^k - 3$.
  - Etc. Count down in binary.
  - If this happens quickly, get pileup of 2^k search messages in $C_{k,k+1}$.



Shortest Paths

- Moral: Unrestrained asynchrony can cause problems.
- Return to this problem after we have better synchronization methods.

- Now, another good illustration of the problems introduced by asynchrony:

Minimum spanning tree

- Assumptions:
  - G = (V,E) connected, undirected.
  - Weighted edges, weights known to endpoint processes, weights distinct.
  - UIDs
  - Processes don't know n, diam.
  - Can identify in- and out-edges to same neighbor.
  - Input: wakeup actions, occurring at any time at one or more nodes.
  - Process wakes up when it first receives either a wakeup input or a protocol message.
- Requires:
  - Produce MST, where each process knows which of its incident edges belong to the tree.
  - Guaranteed to be unique, because of unique weights.

- Gallager-Humblet-Spira algorithm: Read this paper!

Recall synchronous algorithm

- Proceeds in phases (levels).
- After each phase, we have a spanning forest, in which each component tree has a leader.
- In each phase, each component finds min weight outgoing edge (MWOE), then components merge using all MWOEs to get components for next phase.
- In more detail:
  - Each node is initially in component by itself (level 0 components).
  - Phase 1 (produces level 1 components):
    - Each node uses its min weight edge as the component MWOE.
    - Send connect message across MWOE.
    - There is a unique edge that is the MWOE of two components.
    - Leader of new component is higher-id endpoint of this unique edge.
  - Phase k+1 (produces level k+1 components):

Synchronous algorithm

- Phase 1 (produces level 1 components):
  - Each node uses its min weight edge as the component MWOE.
  - Send connect across MWOE.
  - There is a unique edge that is the MWOE of two components.
  - Leader of new component is higher-id endpoint of this unique edge.

- Phase k+1 (produces level k+1 components):
  - Leader of each component initiates search for MWOE (broadcast initiate on tree edges).
  - Each node finds its mwoe:
    - Send test on potential edges, wait for accept (different component) or reject (same component).
    - Test edges one at a time in order of weight.
  - Report to leader (convergecast report); remember direction of best edge.
  - Leader picks MWOE for fragment.
  - Send change-root to MWOE's endpoint, using remembered best edges.
  - Send connect across MWOE.
  - There is a unique edge that is the MWOE of two components.
  - Leader of new component is higher-id endpoint of this unique edge.
  - Wait sufficient time for phase to end.

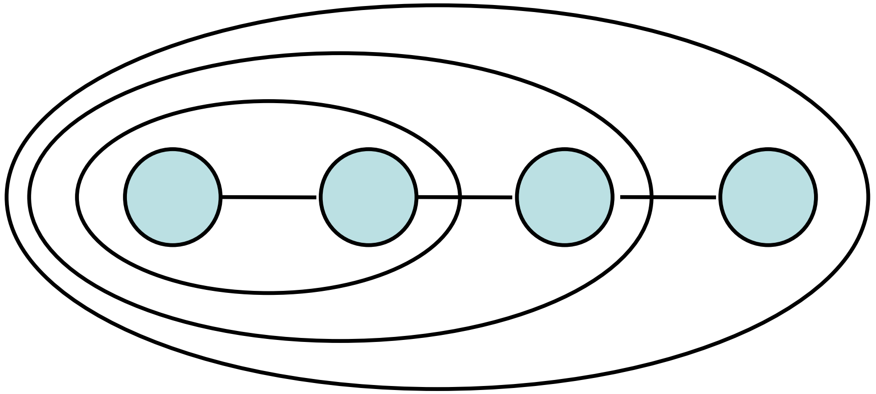
Synchronous algorithm

- Complexity is good:
  - Messages: $O(n \log n + |E|)$
  - Time (rounds): $O(n \log n)$
- Low message complexity depends on the way nodes test their incident edges, in order of weight, not retesting same edge once it's rejected.
- Q: How to run this algorithm asynchronously?

Running the algorithm asynchronously

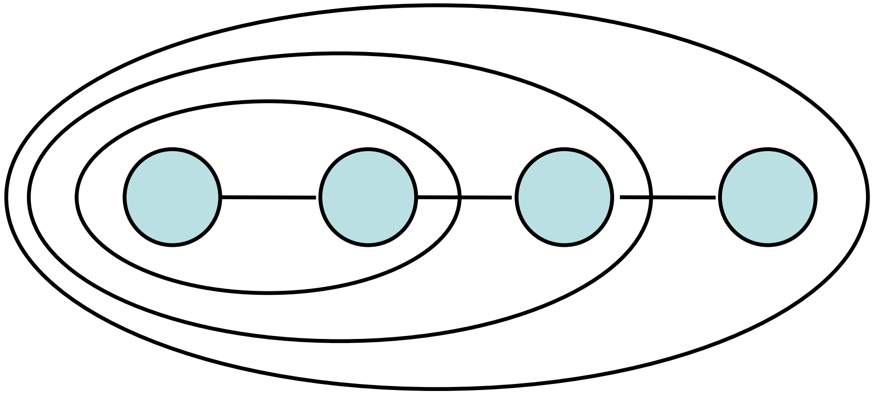
- Problems arise:
  - Inaccurate information about outgoing edges:
    - In synchronous algorithm, when a node tests its edges, it knows that its neighbors are already up to the same level, and have up-to-date information about their component.
    - In asynchronous version, neighbors could lag behind; they might be in same component but not yet know this.
  - Less "balanced" combination of components:
    - In synchronous algorithm, level k components have $\geq 2^k$ nodes, and level k+1 components are constructed from at least two level k components.
    - In asynchronous version, components at different levels could be combined.
    - Can lead to more messages overall.

    - Example: One component could keep merging with level 0 single-node components. After each merge, the number of messages sent in the tree is proportional to the component's size. Leads to $\Omega(n^2)$ messages overall.



Running the algorithm asynchronously

- Problems arise:
  - Inaccurate information about outgoing edges.
  - Less “balanced” combination of components:



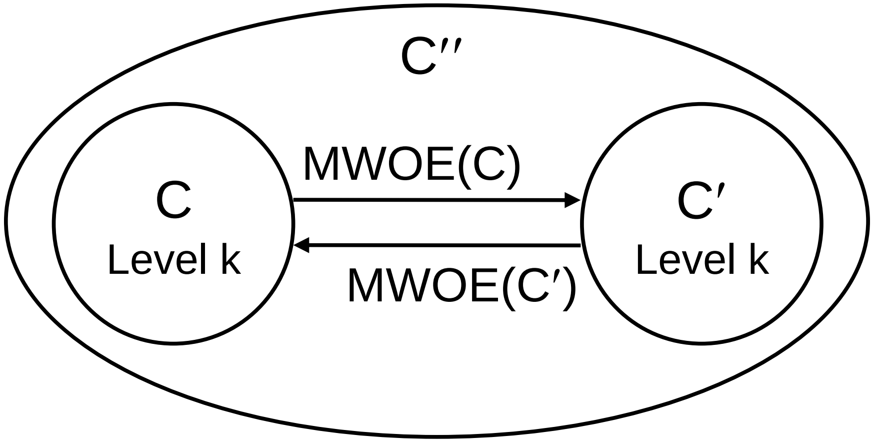
  - Concurrent overlapping searches/convergecasts:
    - When nodes are out of synch, concurrent searches for MWOEs could interfere with each other (we’ll see this).
  - Time bound:
    - These problems result from nodes being out-of-synch, at different levels.
    - We could try to synchronize levels, but this must be done carefully, so as not to hurt the time complexity too much.

GHS algorithm

- Same basic ideas as before:
  - Form components, combine along MWOEs.
  - Within any component, processes cooperate to find component MWOE.
  - Broadcast from leader, convergecast, etc.
- Introduce synchronization to prevent nodes from getting too far ahead of their neighbors.
  - Associate a “level” with each component, as before.
  - Number of nodes in a level k component $\geq 2^k$.
  - Now, each level k+1 component will be (initially) formed from exactly two level k components.
  - Level numbers are used for synchronization, and in determining who is in the same component.
- Complexity:
  - Messages: $O(|E| + n \log n)$
  - Time: $O(n \log n (d + l))$

GHS algorithm

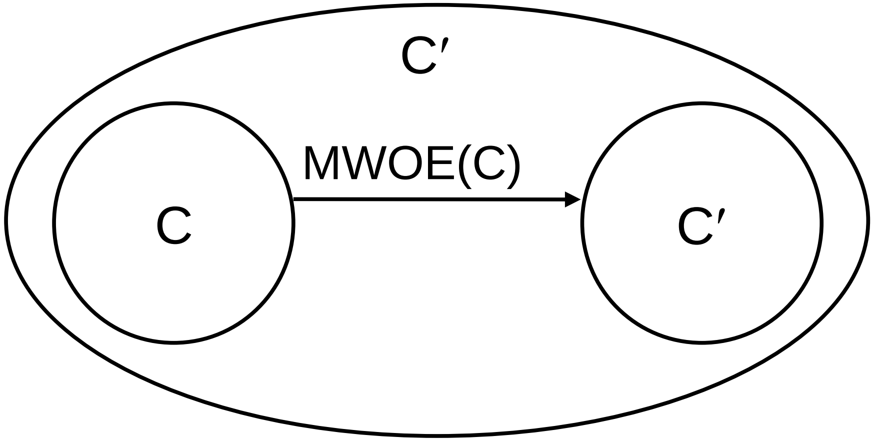
- Combine pairs of components in two ways, merging and absorbing.
- Merging:



  - C and C′ have same level k, and have a common MWOE.
  - Result is a new merged component C′′, with level k+1.

GHS algorithm

- Absorbing:



  – level(C) < level(C′), and C's MWOE leads to C′.
  – Result is to absorb C into C′.
  – Not creating a new component---just adding C to existing C′.
  – C "catches up" with the more advanced C′.
  – Absorbing is cheap, local.

- Merging and absorbing ensure that the number of nodes in any level k component $\geq 2^k$.
- Merging and absorbing are both allowable operations in finding MST, because they are allowed by the general theory for MSTs.

Liveness

- Q: Why are merging and absorbing sufficient to ensure that the construction is eventually completed?
- Lemma: After any allowable finite sequence of merges and absorbs, either the forest consists of one tree (so we're done), or some merge or absorb is enabled.
- Proof:
  - Consider the current "component digraph":
  - Nodes = components
  - Directed edges correspond to MWOEs
  - Then there must be some pair C, C′ whose MWOEs point to each other. (Why?)
  - These MWOEs must be the same edge. (Why?)
  - Can combine, using either merge or absorb:
    - If same level, merge, else absorb.
- So, merging and absorbing are enough.
- Now, how to implement them with a distributed algorithm?

Component names and leaders

- For every component with level ≥ 1, define the core edge of the component's tree.
- Defined in terms of the merge and absorb operations used to construct the component:
  - After merge: Use the common MWOE.
  - After absorb: Keep the old core edge of the higher-level component.
- "The edge along which the most recent merge occurred."

- Component name: (core, level)
- Leader: Endpoint of core edge with higher id.

Determining if an edge is outgoing

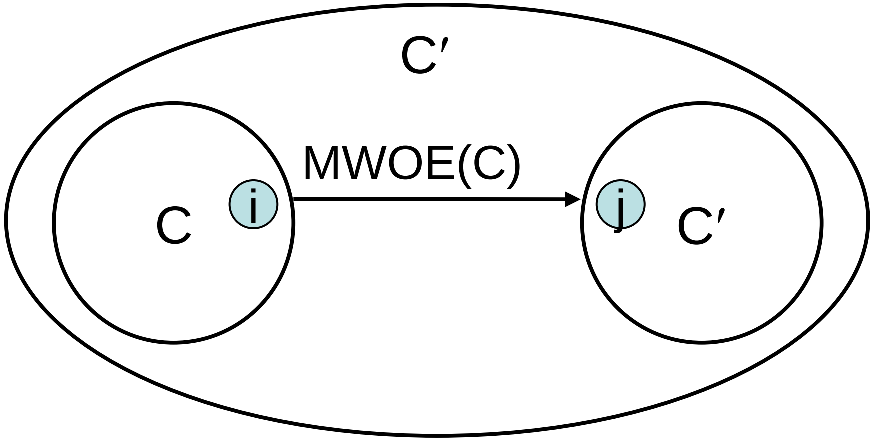
- Suppose i wants to know if the edge (i,j) is outgoing from i's current component.
- At that point, i's component name info is up-to-date:
  - Component is in "search mode".
  - i has received initiate message from the leader, which carried component name.
- So i sends j a test message.
- Three cases:
  - If j's current (core, level) is the same as i's, then j knows that j is in the same component as i.
  - If j's (core, level) is different from i's and j's level is $\geq$ i's, then j knows that j is in a different component from i.
    - Component has only one core per level.
    - No one in the same component currently has a higher level than i does, since the component is still searching for its MWOE.
  - If j's level is < i's, then j doesn't know if it is in the same or a different component. So it doesn't yet respond---waits to catch up to i's level.

Liveness, again

- Q: Can the extra delays imposed here affect the progress argument?
- No:
  - We can redo the progress argument, this time considering only those components with the lowest current level k.
  - All processes in these components must succeed in determining their mwoes, so these components succeed in determining the component MWOE.
  - If any of these level k components' MWOEs leads to a higher level, can absorb.
  - If not then all lead to other level k components, so as before, we must have two components that point to each other; so can merge.

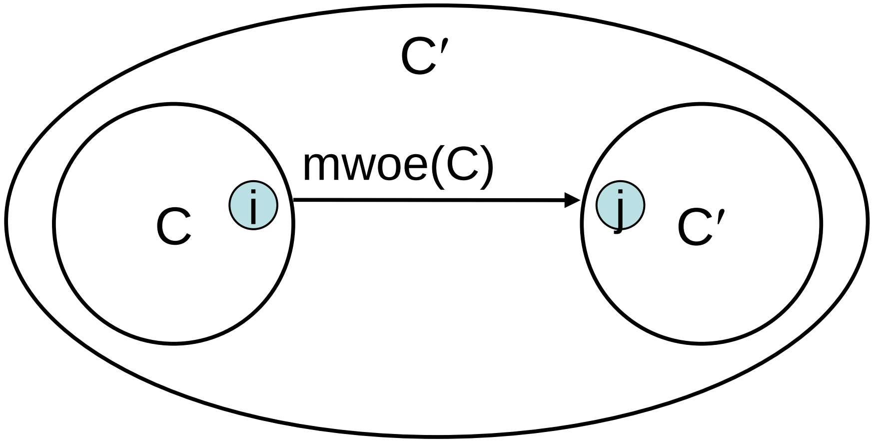
Interference among concurrent MWOE searches

- Suppose C gets absorbed into C′ via an edge from i to j, while C′ is working on determining its MWOE.



- Two cases:
  - j has not yet reported its local mwoe when the absorb occurs.
    - Then it's not too late to include C in the search for the MWOE of C′. So j forwards the initiate message into C.
  - j has already reported its local mwoe.
    - Then it's too late to include C in the search.
    - But it doesn't matter: the MWOE for the combined component can't be outgoing from a node in C anyhow!

Interference among concurrent MWOE searches



- Suppose j has already reported its local mwoe.
- Show that the MWOE for the combined component can't be outgoing from a node in C.

- Claim 1: Reported mwoe(j) cannot be the edge (j,i).
- Proof:
  - Since mwoe(j) has already been reported, it must lead to a node with level $\geq$ level(C′).
  - But the level of i is still < level(C′), when the absorb occurs.
  - So mwoe(j) is a different edge, one whose weight < weight(i,j).

- Claim 2: MWOE for combined component is not outgoing from a node in C.
- Proof:
  - (i,j) is the MWOE of C, so there are no edges outgoing from C with weight < weight(i,j).
  - So no edges outgoing from C with weight < already-reported mwoe(j).
  - So MWOE of combined component isn't outgoing from C.

A few details

- Specific messages:
  - initiate: Broadcast from leader to find MWOE; piggybacks component name.
  - report: Convergecast MWOE responses back to leader.
  - test: Asks whether an edge is outgoing from the component.
  - accept/reject: Answers.
  - changeroot: Sent from leader to endpoint of MWOE.
  - connect: Sent across the MWOE, to connect components.
    - We say merge occurs when connect message has been sent both ways on the edge (2 nodes must have same level).
    - We say absorb occurs when connect message has been sent on the edge from a lower-level to a higher-level node.

Test-Accept-Reject Protocol

- Bookkeeping: Each process i keeps a list of incident edges in order of weight, classified as:
  - branch (in the MST),
  - rejected (leads to same component), or
  - unknown (not yet classified).
- Process i tests only unknown edges, sequentially in order of weight:
  - Sends test message, with (core, level); recipient j compares.
  - If same (core, level), j sends reject (same component), and i reclassifies edge as rejected.
  - If (core, level) pairs are unequal and level(j) ≥ level(i) then j sends accept (different component). i does not reclassify the edge.
  - If level(j) < level(i) then j delays responding, until level(j) ≥ level(i).
- Retesting is possible, for accepted edges.
- Reclassify edge as branch as a result of changeroot message.

Complexity

- As for synchronous version.
- Messages: O(|E| + n log n)
  - 4|E| for test-reject msgs (one pair for each direction of every edge)
  - n initiate messages per level (broadcast: only sent on tree edges)
  - n report messages per level (convergecast)
  - 2n test-accept messages per level (one pair per node)
  - n change-root/connect messages per level (core to MWOE path)
  - log n levels
  - Total: 4|E| + 5n log n
- Time: O(n log n (l + d))

Proving Correctness

- GHS MST is hard to prove, because it's complex.
- GHS paper includes informal arguments.
  - Pretty convincing, but not formal.
  - Also simulated the algorithm extensively.
- Many successful attempts to formalize, all complicated
  - Many invariants because many variables and actions.
  - Some use simulation relations.
  - Recent proof by Moses and Shimony.

Minimum spanning tree

- Application to leader election:
  - Convergecast from leaves until messages meet at node or edge.
  - Works with any spanning tree, not just MST.
  - E.g., in asynchronous ring, this yields $O(n \log n)$ messages for leader election.
- Lower bounds on message complexity:
  - $\Omega(n \log n)$, from leader election lower bound and the reduction above.

Next time

- Synchronizers
- Reading: Chapter 16