

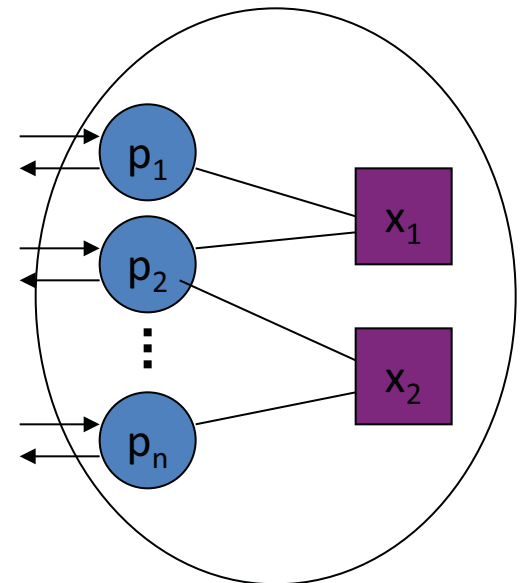
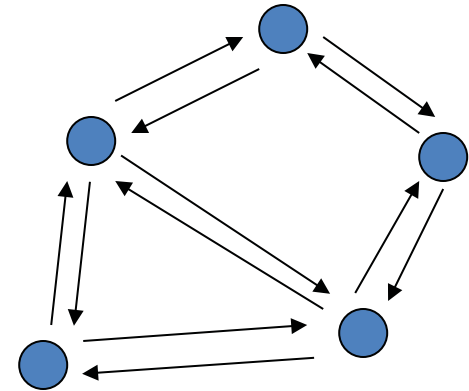
# Distributed Algorithms

## 6.046J, Spring, 2015

Nancy Lynch

# What are Distributed Algorithms?

- Algorithms that run on networked processors, or on multiprocessors that share memory.
- They solve many kinds of problems:
  - Communication
  - Data management
  - Resource management
  - Synchronization
  - Reaching consensus
  - ...
- They work in difficult settings:
  - Concurrent activity at many processing locations
  - Uncertainty of timing, order of events, inputs
  - Failure and recovery of processors, of channels.
- So they can be complicated:
  - Hard to design, prove correct, and analyze.

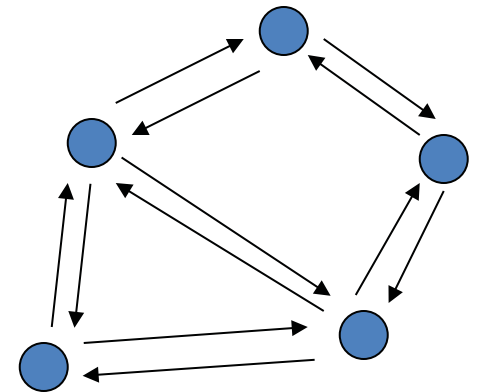


# Distributed Algorithms

- Studied since 1967, starting with Dijkstra and Lamport.
  - Edsger Dijkstra: 1972 Turing award winner
  - Leslie Lamport: 2013 Turing award winner
- Some sources:
  - Lynch, Distributed Algorithms
  - Attiya and Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics
  - Morgan Claypool series of monographs on Distributed Computing Theory
  - Conferences:
    - Principles of Distributed Computing (PODC)
    - Distributed Computing (DISC)

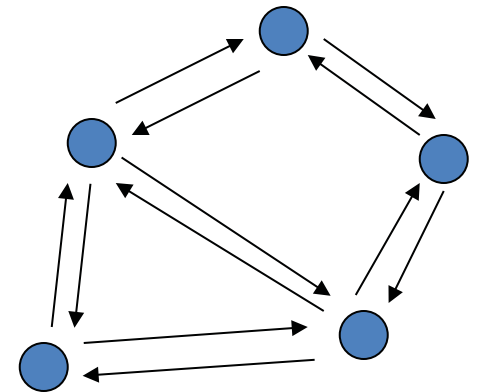
# This Week

- A quick introduction:
  - Two common distributed computing models,
  - A few fundamental algorithms, and
  - Techniques for analyzing algorithms.
- Synchronous distributed algorithms:
  - Leader Election
  - Maximal Independent Set
  - Breadth-First Spanning Trees
  - Shortest Paths Trees
- Asynchronous distributed algorithms:
  - Breadth-First Spanning Trees
  - Shortest Paths Trees



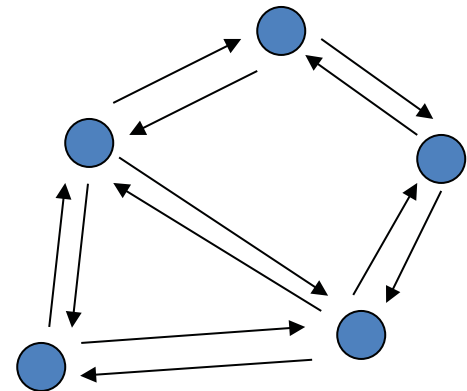
# Modeling, Proofs, Analysis

- Important for distributed algorithms, because they are complicated and error-prone.
- Infinite-state interacting state machines.
- Proofs use invariants, proved by induction.
- Abstraction relations.
- New complexity measures:
  - Time complexity: Rounds, or real time.
  - Communication complexity: Messages, or bits.

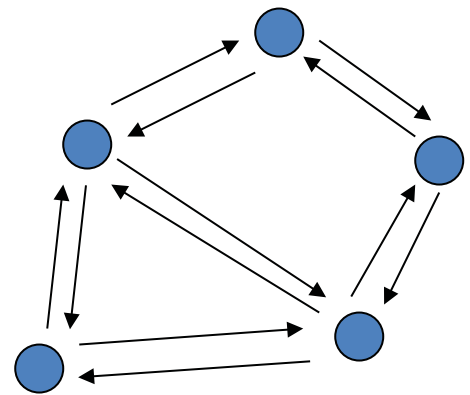


# Distributed Networks

- Based on an undirected graph  $G = (V, E)$ .
  - $n = |V|$
  - $\Gamma(u)$ , set of neighbors of vertex  $u$ .
  - $\deg(u) = |\Gamma(u)|$ , number of neighbors of vertex  $u$ .
- Associate a **process** with each graph vertex.
  - An infinite-state automaton.
  - Sometimes refer to processes, or vertices, as **nodes**.
- Associate two directed **communication channels** with each edge.
- **Note:** We won't consider failures.

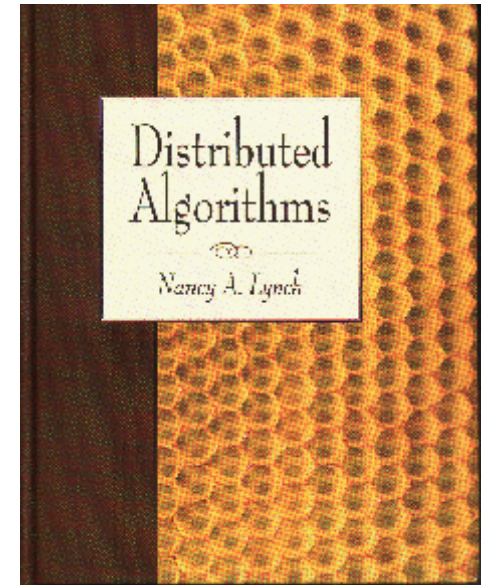


# Synchronous Distributed Algorithms



# Synchronous Network Model

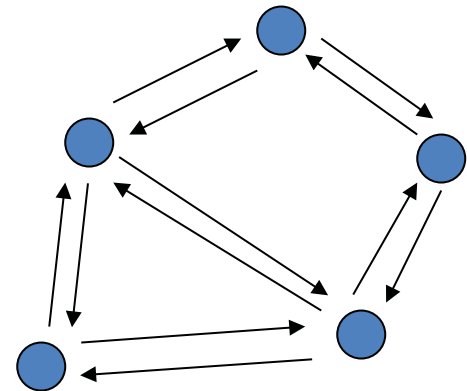
- Lynch, Distributed Algorithms, Chapter 2.
- Processes at nodes of an undirected graph, communicate using messages.
- Each process has **output ports, input ports** that connect to communication channels.
  - Process at vertex  $u$  doesn't know who its ports' channels connect to.
  - Knows the ports only by local names, such as  $1, 2, \dots, k$ , where  $k = \deg(u)$ .
- Processes need not be distinguishable.
  - E.g., they needn't have **Unique Identifiers (UIDs)**.
  - They just know how many ports they have, and their local names.



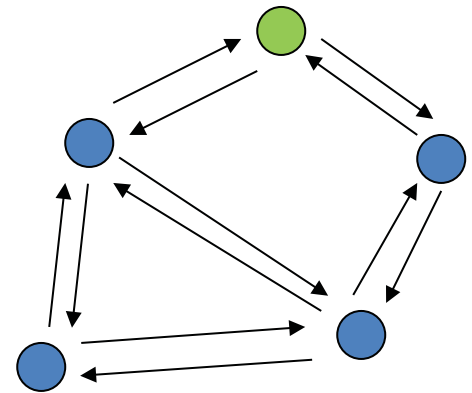


# Execution

- An algorithm executes in **synchronous rounds**.
  - In each round, each process determines, based on its state, the messages to send on all its ports.
    - At most one message per port per round.
  - Each message gets put into its channel, and delivered to the process at the other end.
  - Then each process computes a new state based on its old state and the arriving messages.
- 
- **Note on costs:**
    - Generally ignore costs of local computation (time and space).
    - Focus on time (number of rounds), and communication (number of messages or total number of bits).

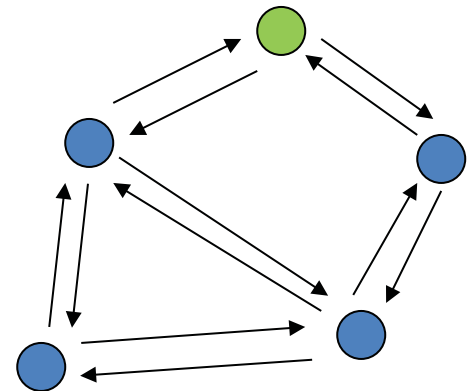


# Leader Election



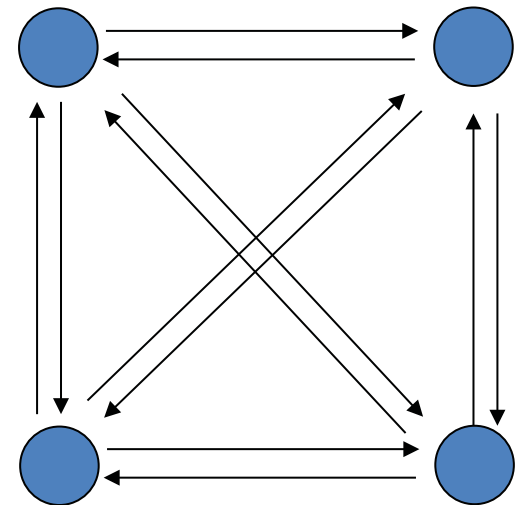
# Leader Election

- $G = (V, E)$  is an arbitrary connected (undirected) graph.
- Goal is for exactly one process to output a special **leader** signal.
- **Motivation:** Leader can take charge of:
  - Communication
  - Coordinating data processing
  - Allocating resources
  - Scheduling tasks
  - Reaching consensus
  - ...

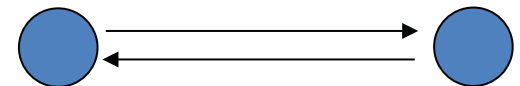


# Simple case: Clique Network

- All vertices are directly connected to all others.
- **Theorem 1:** Let  $G = (V, E)$  be an  $n$ -vertex clique. Then there is no algorithm consisting of deterministic, indistinguishable processes that is guaranteed to elect a leader in  $G$ .



- **Proof:**
  - For example, consider  $n = 2$ .
  - Two identical, deterministic processes.
  - Show by induction that the processes remain in the same state forever...

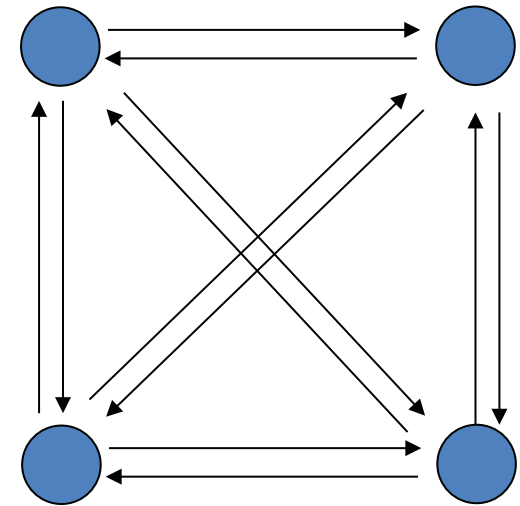


# Proof

- By contradiction. Assume an algorithm  $A$  that solves the problem.
- Both processes begin in the same start state.
- Prove by induction on the number  $r$  of completed rounds that both processes are in identical states after  $r$  rounds.
  - Generate the same message.
  - Receive the same message.
  - Make the same state change.
- Since the algorithm solves the leader election problem, eventually one of them gets elected.
- Then they both get elected, contradiction.

# Clique Network

- **Theorem 1:** Let  $G = (V, E)$  be an  $n$ -vertex clique. Then there is no algorithm consisting of deterministic, indistinguishable processes that is guaranteed to elect a leader in  $G$ .



- **Proof:**
  - Now consider  $n > 2$ .
  - Processes have  $n - 1$  output ports and  $n - 1$  input ports, each set numbered  $1, 2, \dots, n - 1$ .
  - Assume ports are connected “consistently”: output port  $k$  at a process is connected to input port  $k$  at the other end.
  - Show by induction that all the processes remain in the same state forever...

# Proof

- Assume an algorithm  $A$  that solves the leader election problem.
  - All processes start in the same start state.
  - Prove by induction on the number  $r$  of completed rounds that all processes are in identical states after  $r$  rounds.
    - For each  $k$ , they generate the same message on port  $k$ .
    - For each  $k$ , they receive the same message on port  $k$ .
    - Make the same state changes.
  - Since the algorithm solves the leader election problem, eventually some process gets elected.
  - Then everyone gets elected, contradiction.
- 
- A basic problem for distributed algorithms: **Breaking symmetry**
  - Deterministic, indistinguishable processes can't do it.
  - So we need something more...

# So we need something more...

- **Unique Identifiers (UIDs)**
  - Assume processes have unique identifiers (UIDs), which they “know”, e.g., each process starts with its own UID in a special state variable.
  - UIDs are elements of some totally-ordered set, e.g., the natural numbers.
  - Different UIDs can appear anywhere in the graph, but each can appear only once.
- **Randomness**



# Algorithm Using UIDs

- **Theorem 2:** Let  $G = (V, E)$  be an  $n$ -vertex clique. Then there is an algorithm consisting of deterministic processes with UIDs that is guaranteed to elect a leader in  $G$ .
- The algorithm takes only 1 round and uses only  $n^2$  point-to-point messages.

- **Algorithm:**

- Everyone sends its UID on all its output ports, and collects UIDs received on all its input ports.
- The process with the maximum UID elects itself the leader.

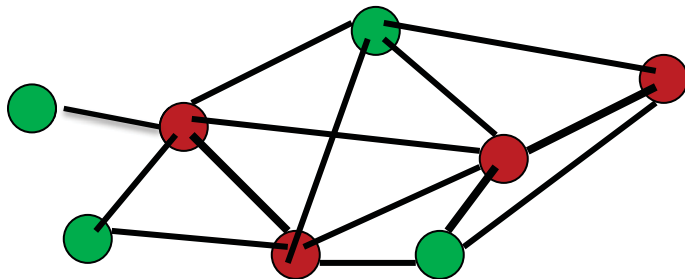
# Algorithm Using Randomness

- **Idea:** Processes choose IDs randomly, from a sufficiently large set so that it is likely that all are different. Then use them like UUIDs.
- **Q:** What is a “sufficiently large” set?
- **Lemma 3:** Let  $\epsilon$  be a real,  $0 < \epsilon < 1$ . Suppose that  $n$  processes choose ids uniformly at random, independently, from  $\{1, \dots, r\}$ , where  $r = \lceil n^2 / 2\epsilon \rceil$ . Then with probability at least  $1 - \epsilon$ , all the chosen numbers are different.
- **Proof:** The probability of any two particular processes choosing the same number is  $\frac{1}{r}$ . Taking a union bound for all  $\frac{n^2}{2}$  pairs, the probability is still  $\leq \epsilon$ .

# Algorithm Using Randomness

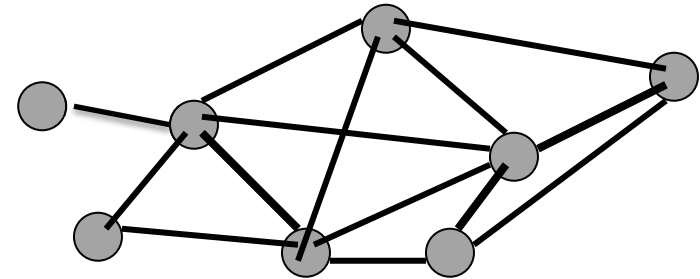
- **Theorem 4:** Let  $G = (V, E)$  be an  $n$ -vertex clique. Then there is an algorithm consisting of randomized, indistinguishable processes that eventually elects a leader in  $G$ , with probability 1.
- The algorithm takes expected time  $\leq \frac{1}{1-\epsilon}$ .
- Also, with probability  $\geq 1 - \epsilon$ , the algorithm finishes in only one round.
- **Algorithm:**
  - Processes choose random ids from a sufficiently large space.
  - Exchange ids; if the maximum is unique, the maximum wins.
  - Otherwise repeat, as many times as necessary.

# Maximal Independent Set

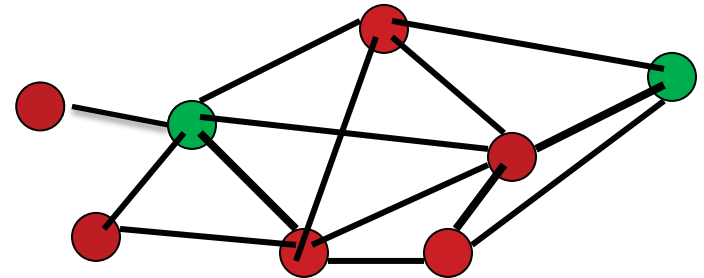
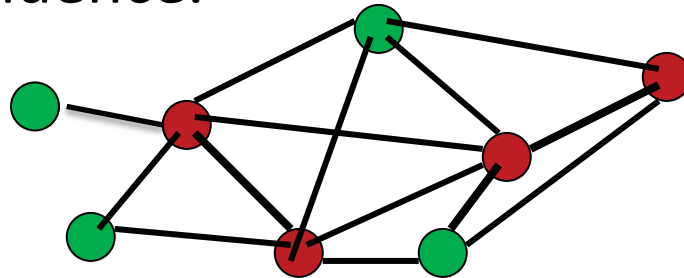


# Maximal Independent Set (MIS)

- General undirected graph network:
- **Problem:** Select a subset  $S$  of the nodes, so that they form a Maximal Independent Set.

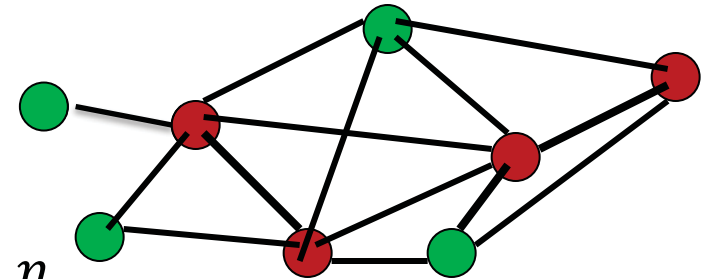


- **Independent:** No two neighbors are both in the set.
- **Maximal:** We can't add any more nodes without violating independence.



- Every node is either in  $S$  or has a neighbor in  $S$ .

# Distributed MIS

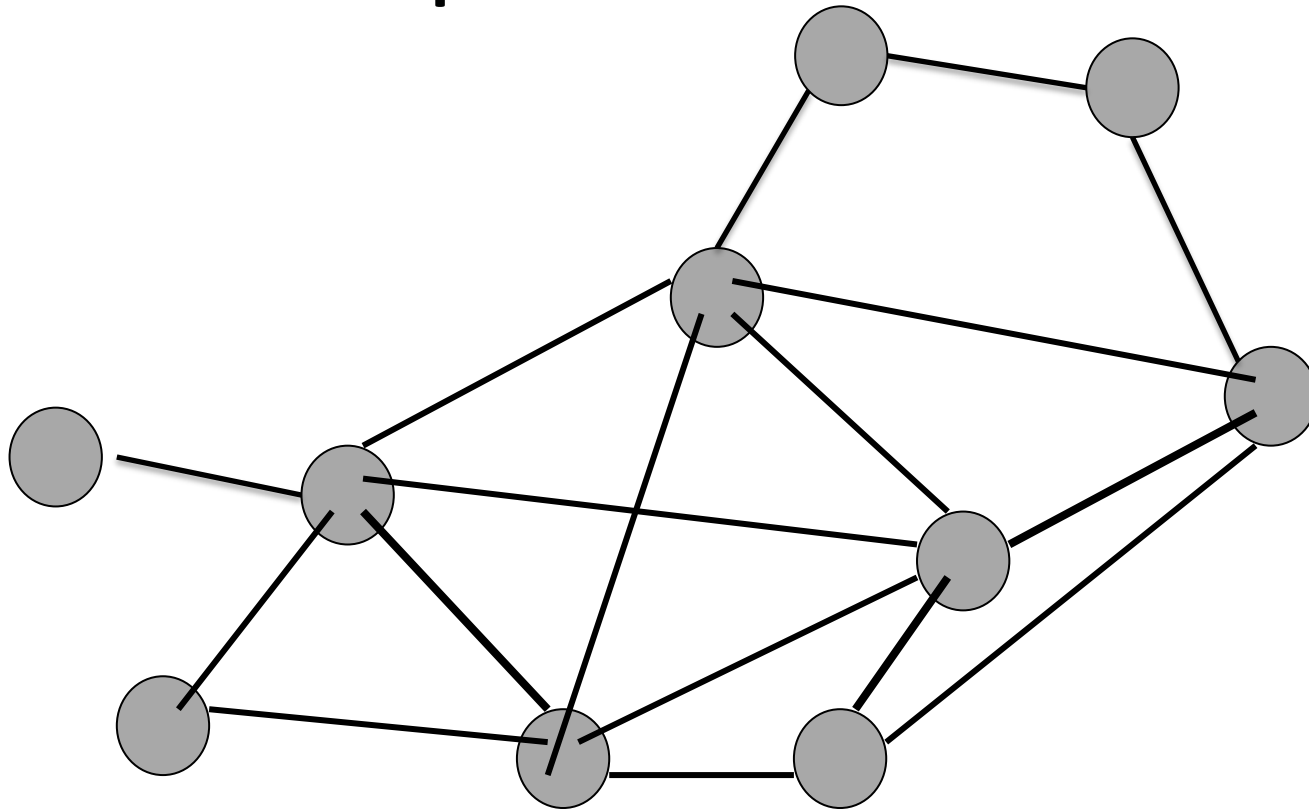


- **Assume:**
  - No UUIDs
  - Processes know a good upper bound on  $n$ .
- **Require:**
  - Compute an MIS  $S$  of the entire network graph.
  - Each process in  $S$  should output **in**, others output **out**.
- Unsolvable by deterministic algorithms, in some graphs.
- So consider randomized algorithms.
- **Applications of distributed MIS:**
  - **Communication networks:** Selected processes can take charge of communication, convey information to their neighbor processes.
  - **Developmental biology:** Distinguish cells in fruit fly's nervous system to become "Sensory Organ Precursor" cells [Afek, Alon, et al., Science].

# Luby's MIS Algorithm

- Executes in 2-round **phases**.
- Initially all nodes are **active**.
- At each phase, some active nodes decide to be **in**, others decide to be **out**, algorithm continues to the next phase with a smaller graph.
- Repeat until all nodes have decided.
  
- Behavior of active node  $u$  at phase  $ph$ :
- Round 1:
  - Choose a random value  $r$  in  $\{1, 2, \dots, n^5\}$ , send it to all neighbors.
  - Receive values from all active neighbors.
  - If  $r$  is strictly greater than all received values, then join the MIS, output **in**.
- Round 2:
  - If you joined the MIS, announce it in messages to all (active) neighbors.
  - If you receive such an announcement, decide not to join the MIS, output **out**.
  - If you decided one way or the other at this phase, become **inactive**.

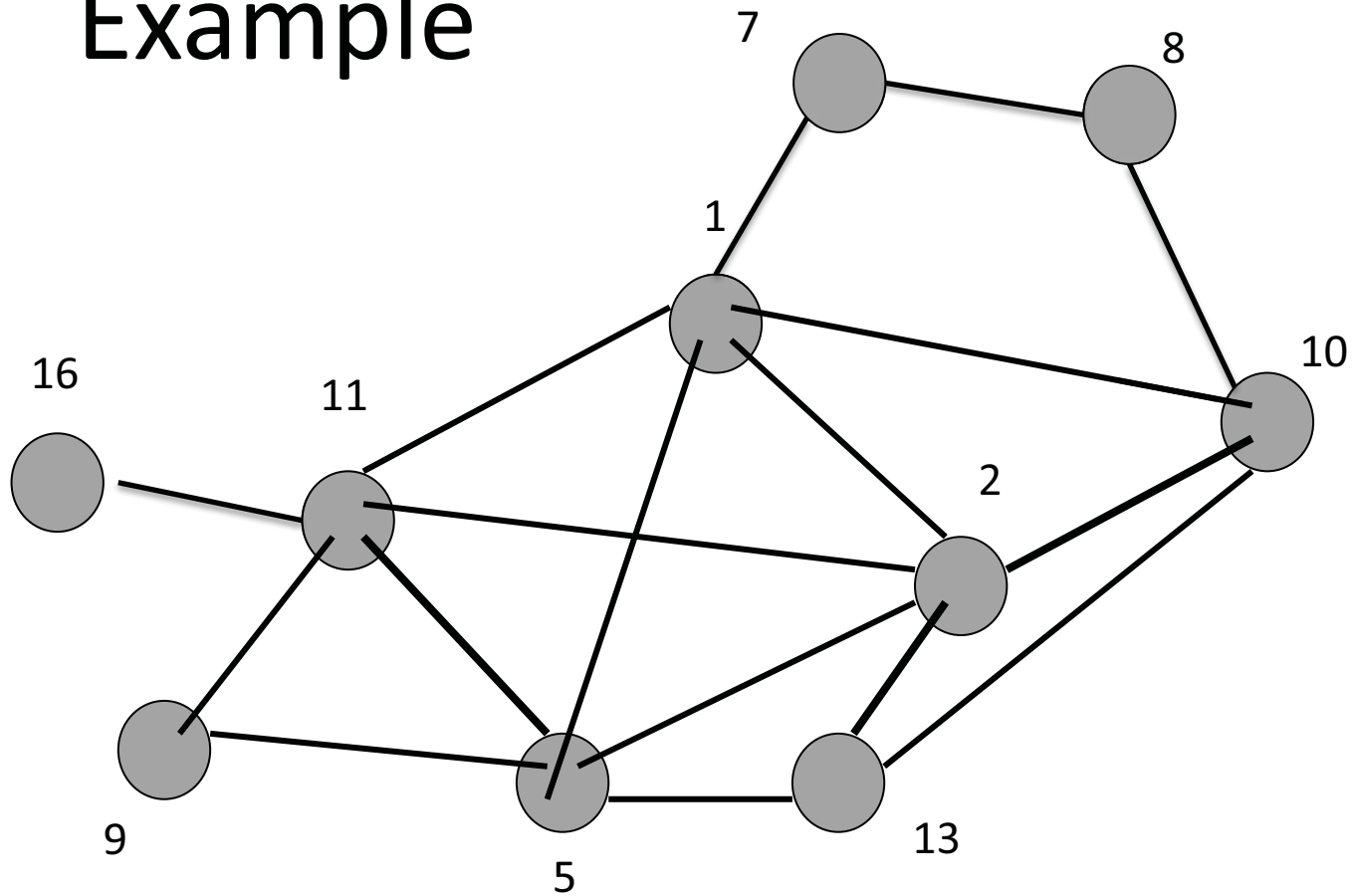
# Example



- All nodes start out identical.

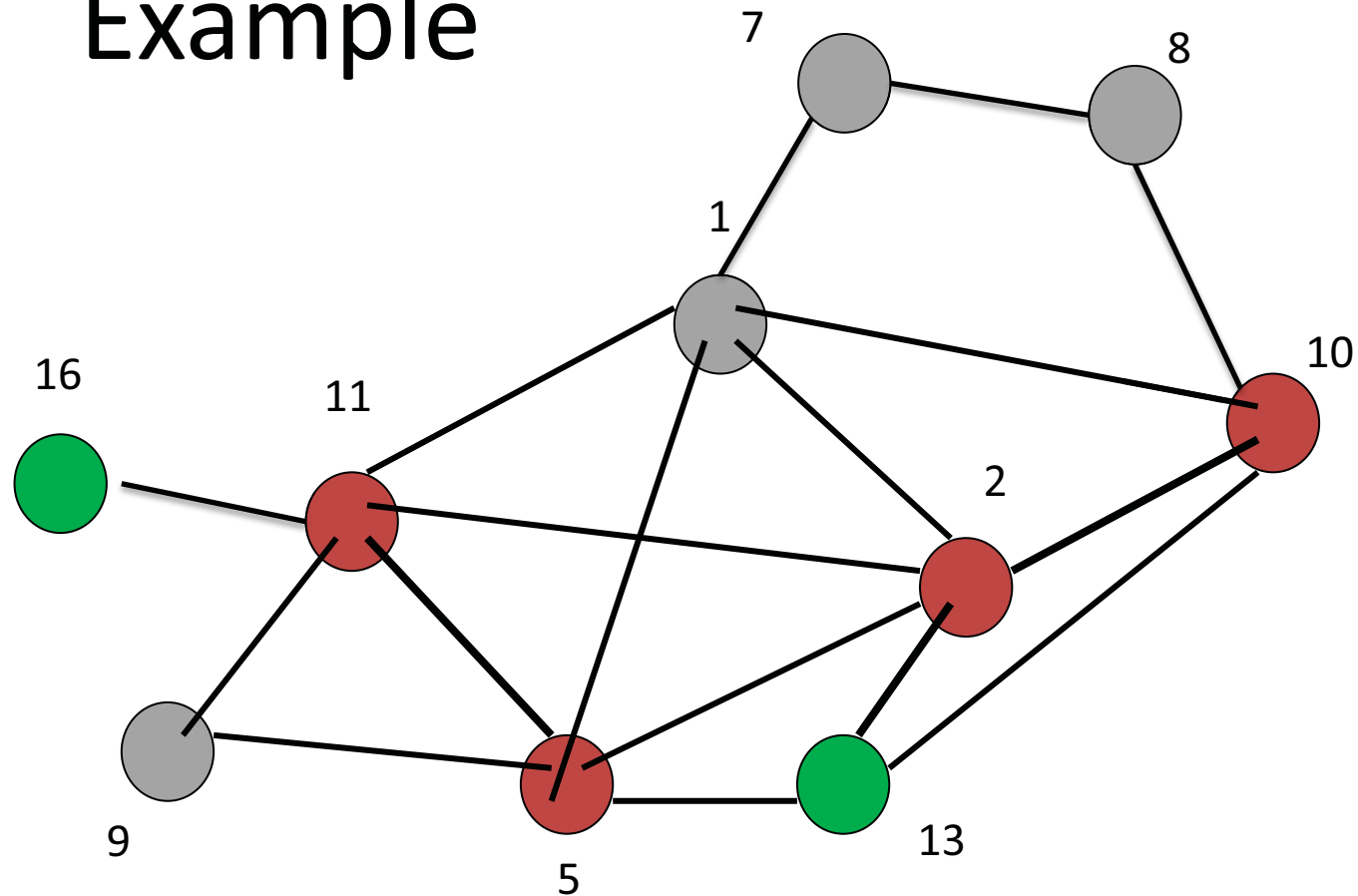


# Example

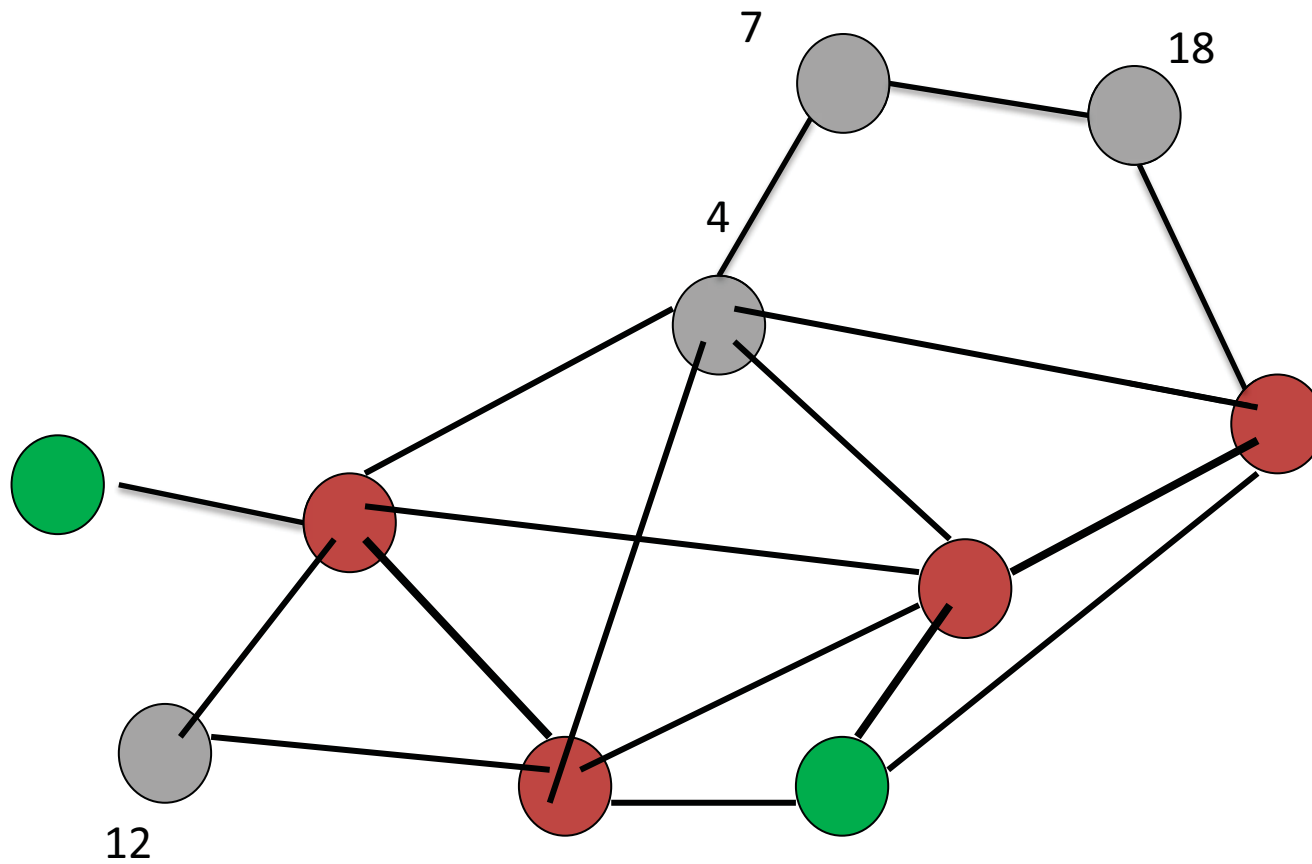


- Everyone chooses an ID.

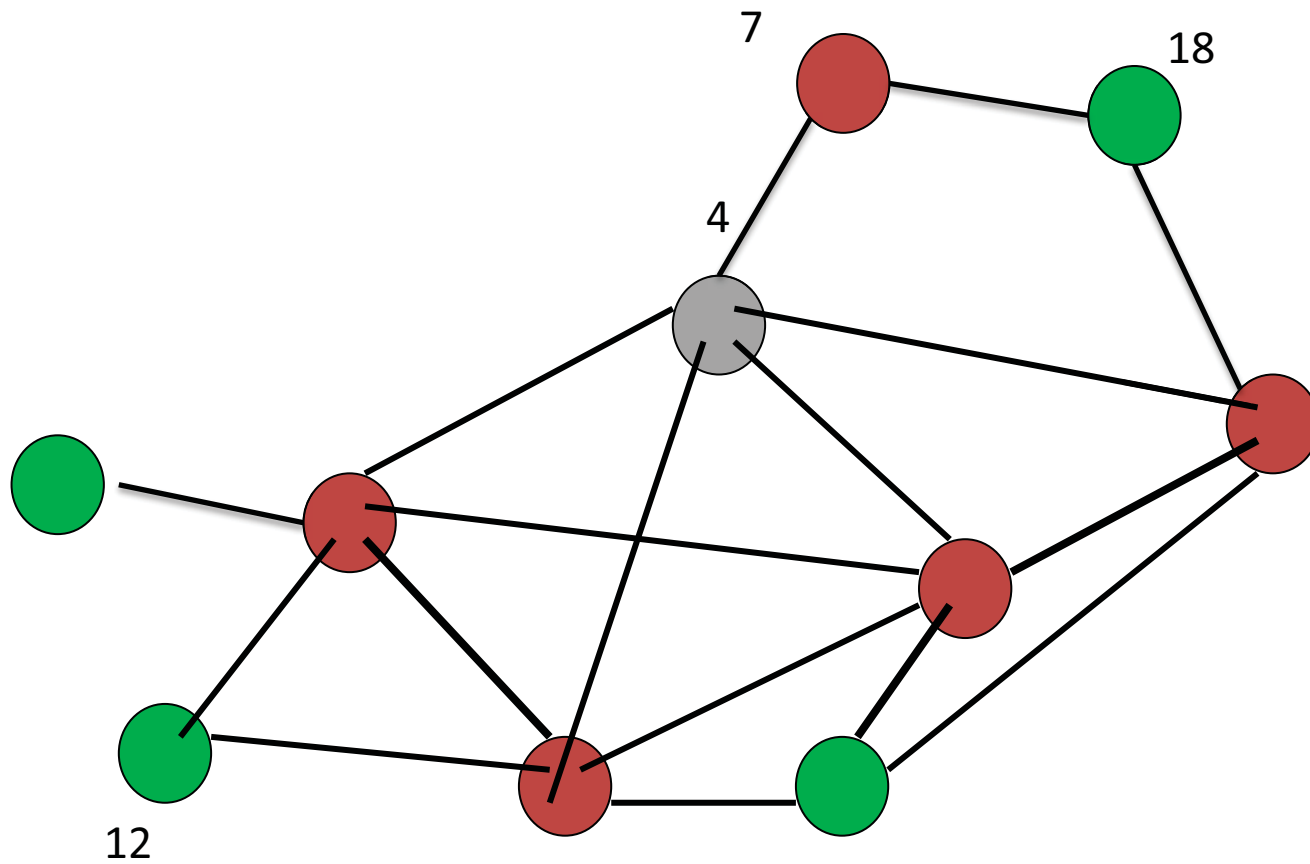
# Example



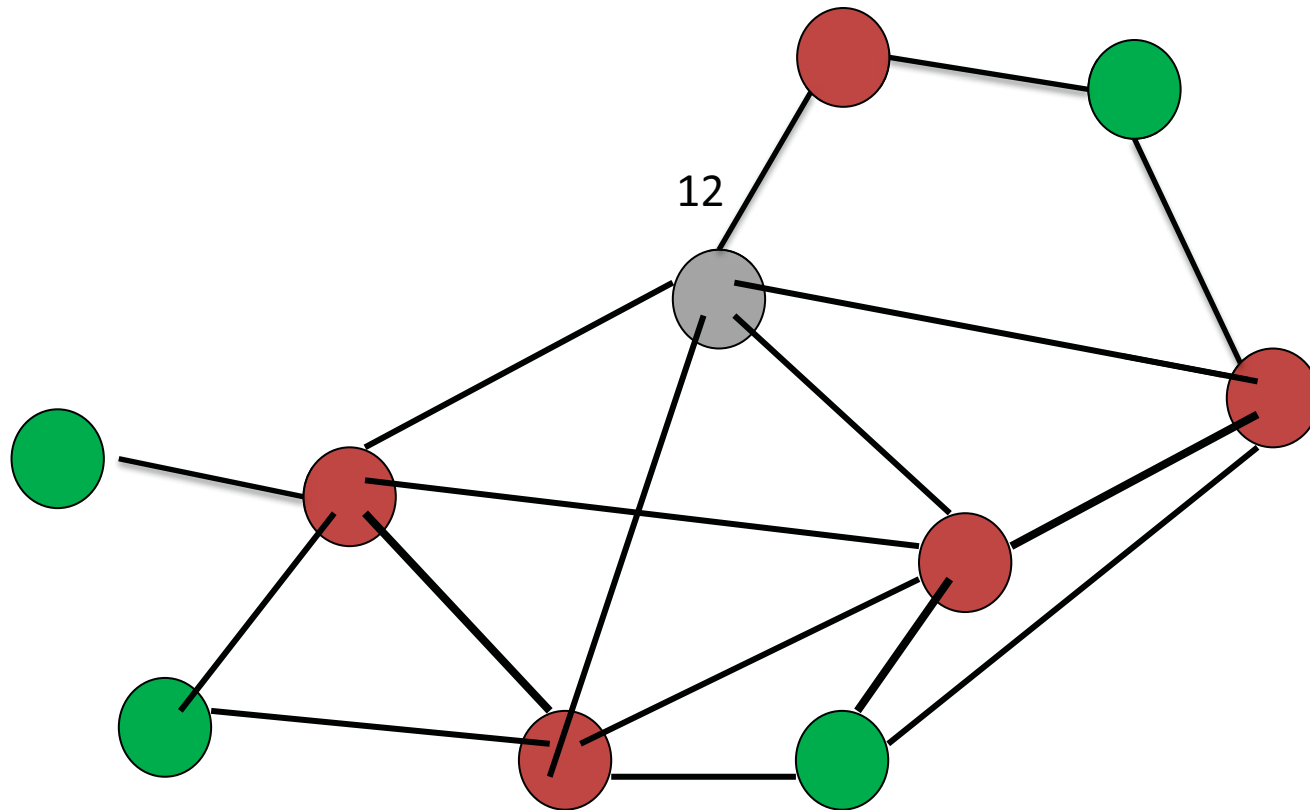
- Processes that chose 16 and 13 are in.
- Processes that chose 11, 5, 2, and 10 are out.



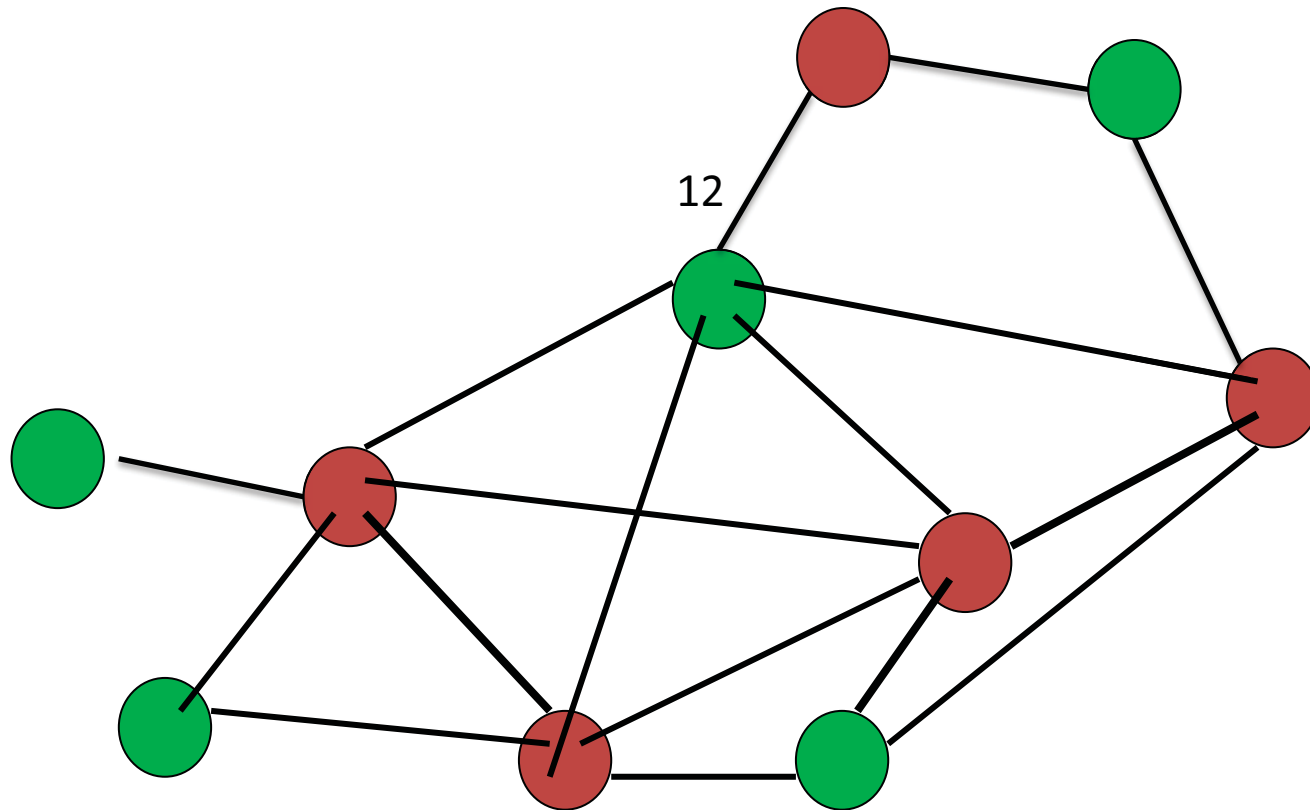
- Undecided (gray) processes choose new IDs.



- Processes that chose 12 and 18 are in.
- Process that chose 7 is out.



- Undecided (gray) process chooses a new ID.



- It's in.

# Independence

- **Theorem 5:** If Luby's algorithm ever terminates, then the final set  $S$  satisfies the independence property.
- **Proof:**
  - Each node joins  $S$  only if it has the unique maximum value in its neighborhood, at some phase.
  - When it does, all its neighbors become inactive.

# Maximality


- **Theorem 6:** If Luby's algorithm ever terminates, then the final set  $S$  satisfies the maximality property.
- **Proof:**
  - A node becomes inactive only if it joins  $S$  or a neighbor joins  $S$ .
  - We continue until all nodes are inactive.

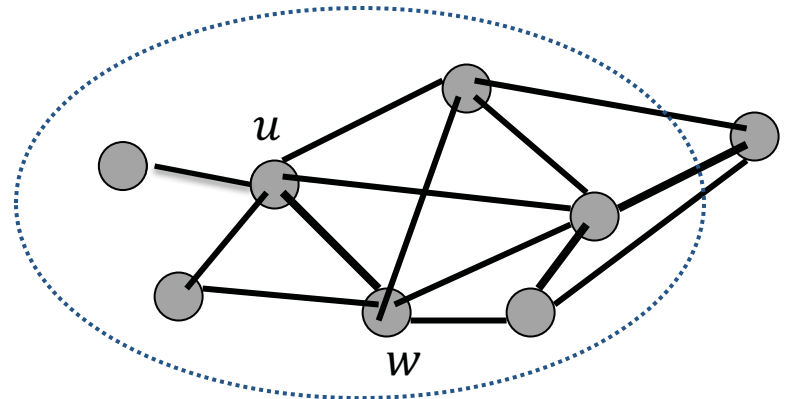


# Termination

- With probability 1, Luby's MIS algorithm eventually terminates.
- **Theorem 7:** With probability at least  $1 - \frac{1}{n}$ , all nodes decide within  $4 \log n$  phases.
- Proof uses a lemma similar to before:
- **Lemma 8:** With probability at least  $1 - \frac{1}{n^2}$ , in each phase  $1, \dots, 4 \log n$ , all nodes choose different random values.
- So we can essentially pretend that, in each phase, all the random numbers chosen are different.
- **Key idea:** Show the graph gets sufficiently “smaller” in each phase.
- **Lemma 9:** For each phase  $ph$ , the expected number of **edges** that are live (connect two active nodes) at the end of the phase is at most half the number that were live at the beginning of the phase.


# Termination

- **Lemma 9:** For each phase  $ph$ , the expected number of edges that are live (connect two active nodes) after the phase is at most half the number that were live before the phase.
- **Proof:**
  - If node  $u$  has some neighbor  $w$  whose chosen value is greater than all of  $w$ 's neighbors and all of  $u$ 's other neighbors, then  $u$  must decide **out** in phase  $ph$ . 
  - Probability that  $w$  chooses such a value is at **least**  $\frac{1}{\deg(u) + \deg(w)}$ .
  - Then the probability node  $u$  is “killed” by some neighbor in this way is at **least**  $\sum_{w \in \Gamma(u)} \frac{1}{\deg(u) + \deg(w)}$ .
  - Now consider an edge  $\{u, v\}$ .



# Termination, cont'd

- Proof:

- Probability  $u$  killed  $\geq \sum_{w \in \Gamma(u)} \frac{1}{\deg(u) + \deg(w)}$ .
- Probability that edge  $\{u, v\}$  “dies” 

$$\geq \frac{1}{2} (\text{probability } u \text{ killed} + \text{probability } v \text{ killed}).$$
- So the expected number of edges that die
 
$$\geq \frac{1}{2} \sum_{\{u,v\}} (\text{probability } u \text{ killed} + \text{probability } v \text{ killed}).$$
- The sum includes the “kill probability” for each node  $u$  exactly  $\deg(u)$  times.
- So rewrite the sum as:
 
$$\frac{1}{2} \sum_u \deg(u) (\text{probability } u \text{ killed}).$$
- Plug in the probability lower bound:
 
$$\geq \frac{1}{2} \sum_u \deg(u) \sum_{w \in \Gamma(u)} \frac{1}{\deg(u) + \deg(w)}.$$

$$= \frac{1}{2} \sum_u \sum_{w \in \Gamma(u)} \frac{\deg(u)}{\deg(u) + \deg(w)}.$$

# Termination, cont'd

- **Proof:**

- Expected number of edges that die  $\geq \frac{1}{2} \sum_u \sum_{w \in \Gamma(u)} \frac{\deg(u)}{\deg(u) + \deg(w)}$ .

- Write this expression equivalently as a sum over **directed edges  $(u, v)$** :

$$\frac{1}{2} \sum_{(u,v)} \frac{\deg(u)}{\deg(u) + \deg(v)}.$$

- Here each undirected edge is counted twice, once for each direction, so this is the same as the following sum over **undirected edges  $\{u, v\}$** .

$$\frac{1}{2} \sum_{\{u,v\}} \frac{\deg(u) + \deg(v)}{\deg(u) + \deg(v)}.$$

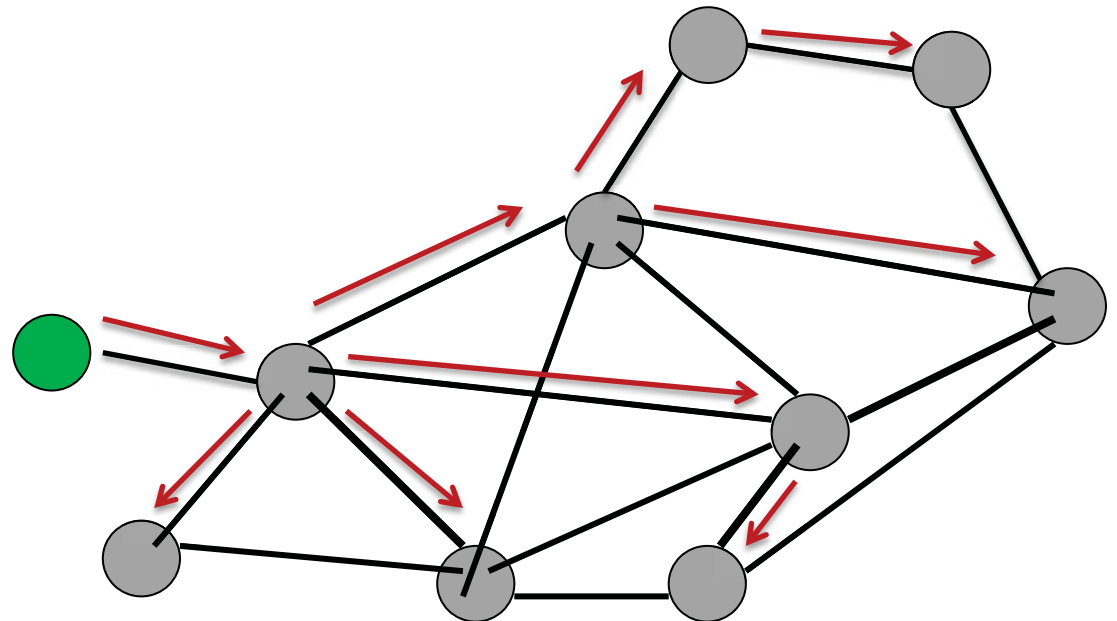
- This is half the total number of undirected edges, as needed!

- **Lemma 9:** For each phase  $ph$ , the expected number of edges that are live (connect two active nodes) at the end of the phase is at most half the number that were live at the beginning of the phase.

# Termination, cont'd

- **Lemma 9:** For each phase  $ph$ , the expected number of edges that are live (connect two active nodes) at the end of the phase is at most half the number that were live before the phase.
- **Theorem 7:** With probability at least  $1 - \frac{1}{n}$ , all nodes decide within  $4 \log n$  phases.
- **Proof sketch:**
  - Lemma 9 implies that the expected number of edges still live after  $4 \log n$  phases is at most  $\frac{n^2}{2} \div 2^{4 \log n} = \frac{1}{2n^2}$ .
  - Then the probability that *any* edges remain live is  $\leq \frac{1}{2n^2}$  (by Markov).
  - The probability that the algorithm doesn't terminate within  $4 \log n$  phases  $\leq \frac{1}{2n^2} + \frac{1}{n^2} < \frac{1}{n}$ .

# Breadth-First Spanning Trees



# Breadth-First Spanning Trees

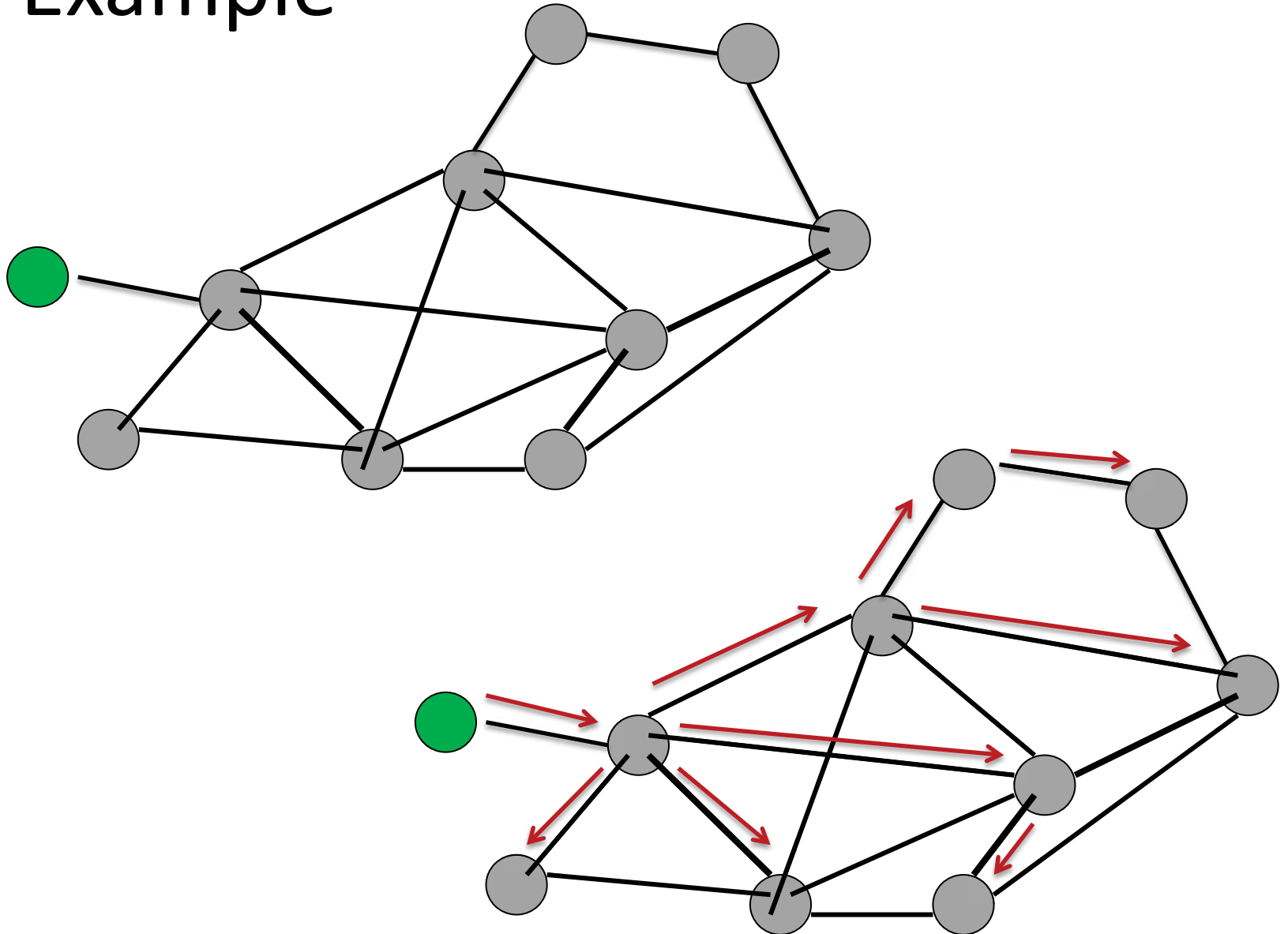
- New problem, new setting.
- Assume graph  $G = (V, E)$  is connected.
- $V$  includes a distinguished vertex  $v_0$ , which will be the origin (root) of the BFS tree.
- Generally, processes have no knowledge about the graph.
- Processes have UIDs.
  - Each process knows its own UID.
  - $i_0$  is the UID of the root  $v_0$ .
  - Process with UID  $i_0$  knows it is located at the root.
- We may assume (WLOG) that processes know the UIDs of their neighbors, and know which input and output ports are connected to each neighbor.
- Algorithms will be deterministic (or nondeterministic), but not randomized.

# Breadth-First Spanning Trees

- Processes must produce a Breadth-First Spanning Tree rooted at vertex  $v_0$ .
- Branches are directed paths from  $v_0$ .
  - **Spanning:** Branches reach all vertices.
  - **Breadth-first:** Vertex at distance  $d$  from  $v_0$  appears at depth exactly  $d$  in the tree.
- **Output:** Each process  $i \neq i_0$  should output *parent(j)*, meaning that  $j$ 's vertex is the parent of  $i$ 's vertex in the BFS tree.



# Example



# Simple BFS Algorithm

- Processes **mark** themselves as they get incorporated into the tree.
- Initially, only  $i_0$  is marked.
- Algorithm for process  $i$ :
  - Round 1:
    - If  $i = i_0$  then process  $i$  sends a *search* message to its neighbors.
    - If process  $i$  receives a message, then it:
      - Marks itself.
      - Selects  $i_0$  as its parent, outputs *parent*( $i_0$ ).
      - Plans to send at the next round.
  - Round  $r > 1$ :
    - If process  $i$  planned to send, then it sends a *search* message to its neighbors.
    - If process  $i$  is not marked and receives a message, then it:
      - Marks itself.
      - Selects one sending neighbor,  $j$ , as its parent, outputs *parent*( $j$ ).
      - Plans to send at the next round.

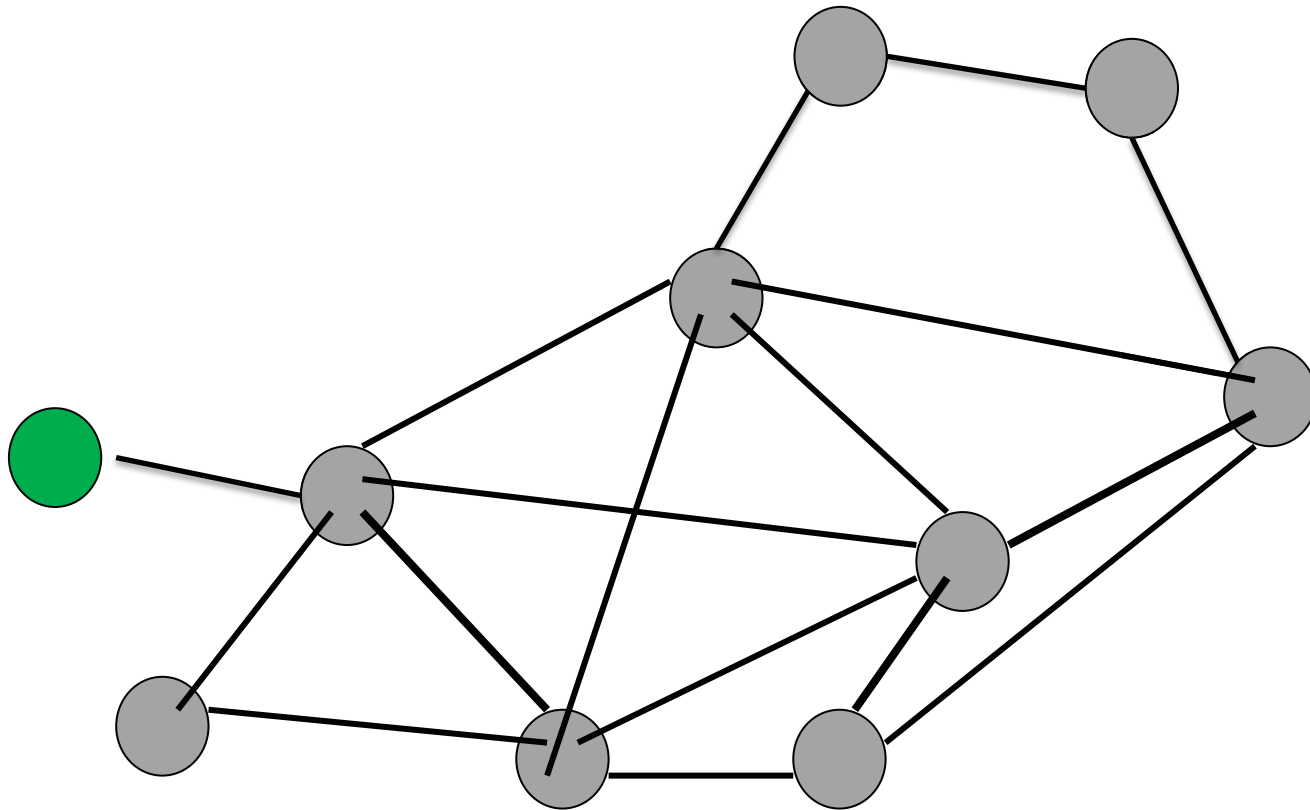
# Nondeterminism

- The algorithm is **slightly nondeterministic**, in that a process can choose arbitrarily among several possible parents.
- We could make this deterministic, by using a default, like “always choose the sender with the smallest UID”.
- But it’s also OK to leave it nondeterministic.
- For distributed algorithms, nondeterminism is regarded differently from the way it is for sequential algorithms.
- **A distributed algorithm should work correctly for all ways of resolving the nondeterministic choices.**

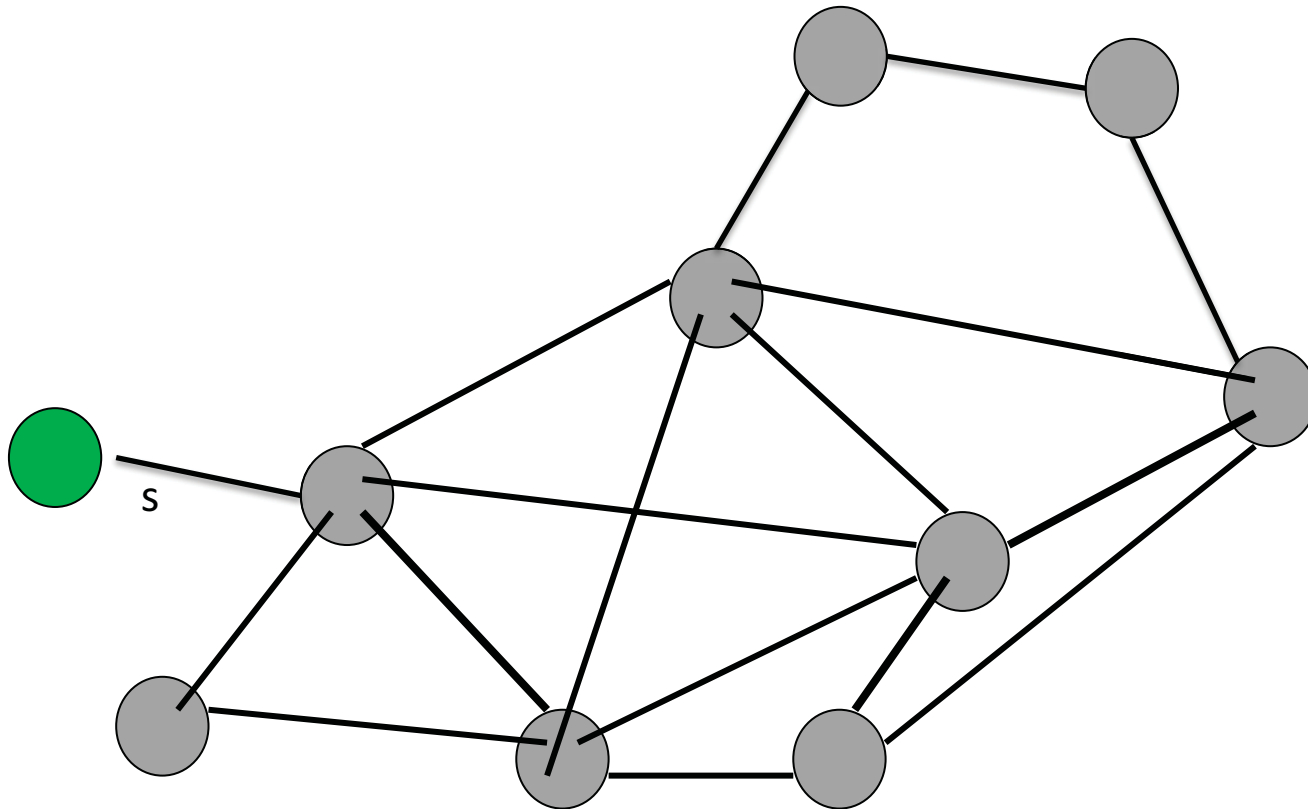
# Simple BFS Algorithm

- Q: Why does this algorithm yield a BFS tree?
- Because all branches are created synchronously, growing one hop at each round.
- Q: Why does it eventually span all the nodes?
- Because the graph is connected, and any marked node sends messages to its neighbors.

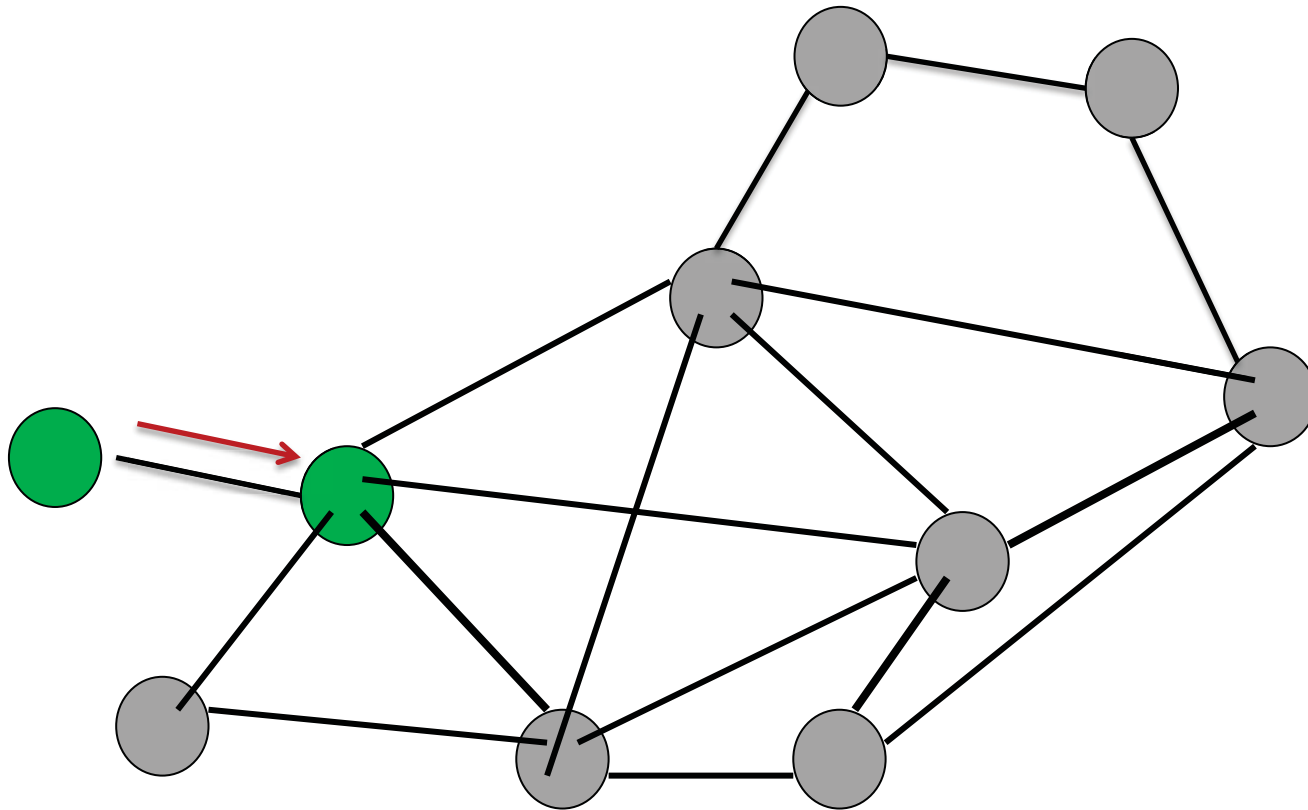
# Example: Simple BFS



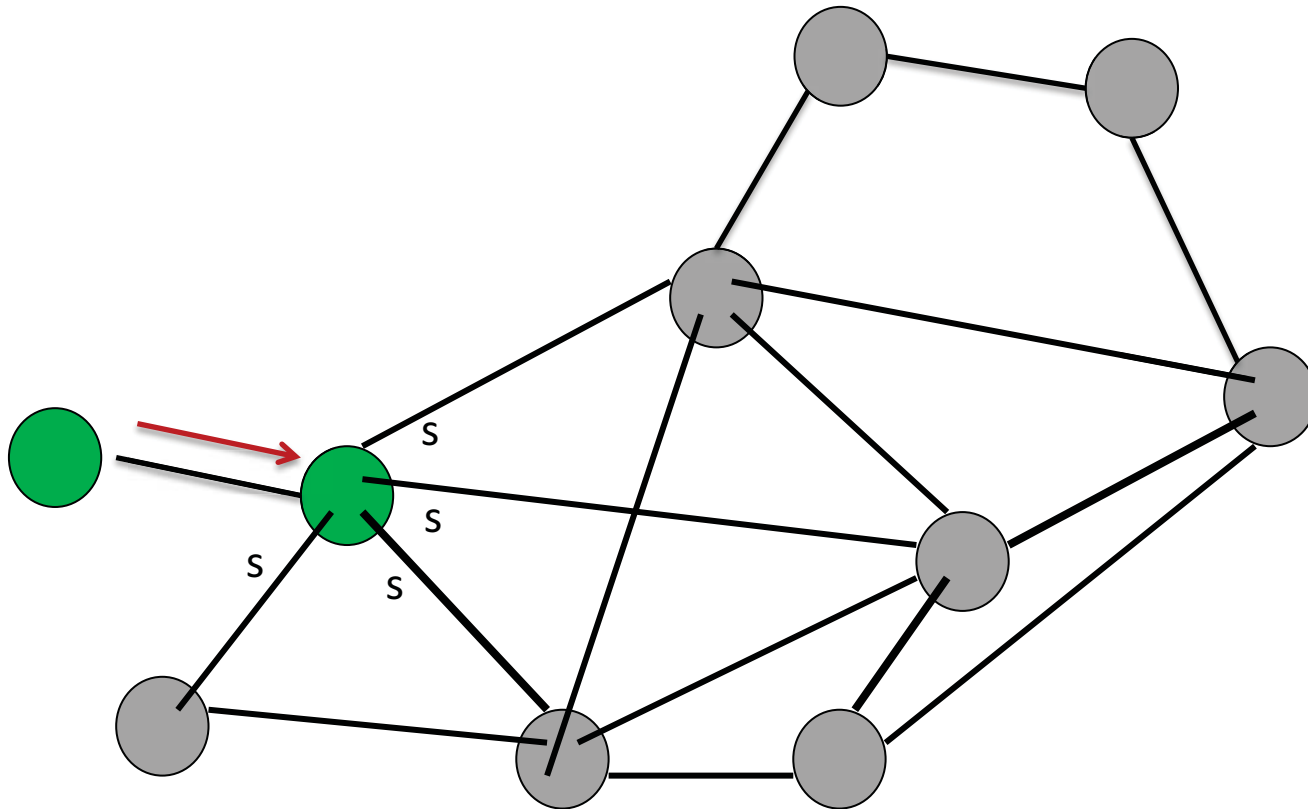
# Round 1



# Round 1

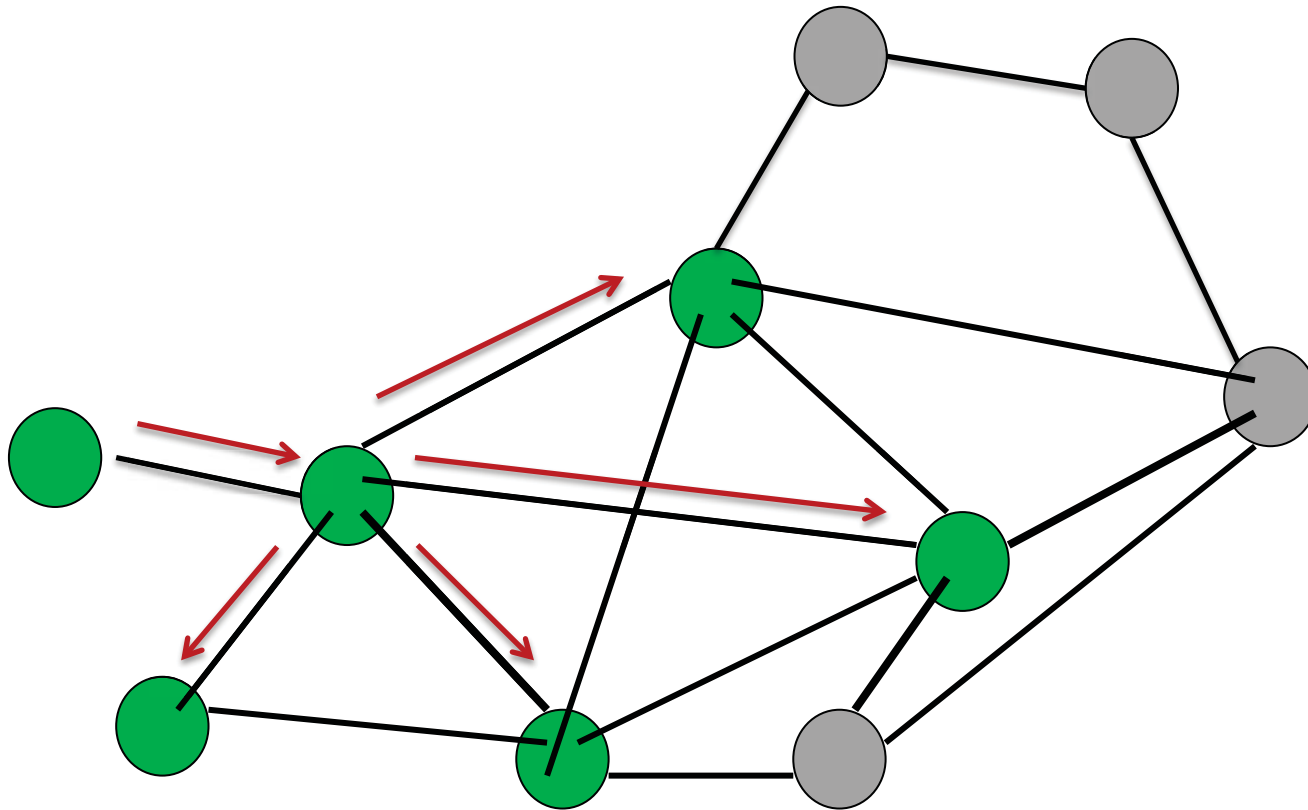


# Round 2

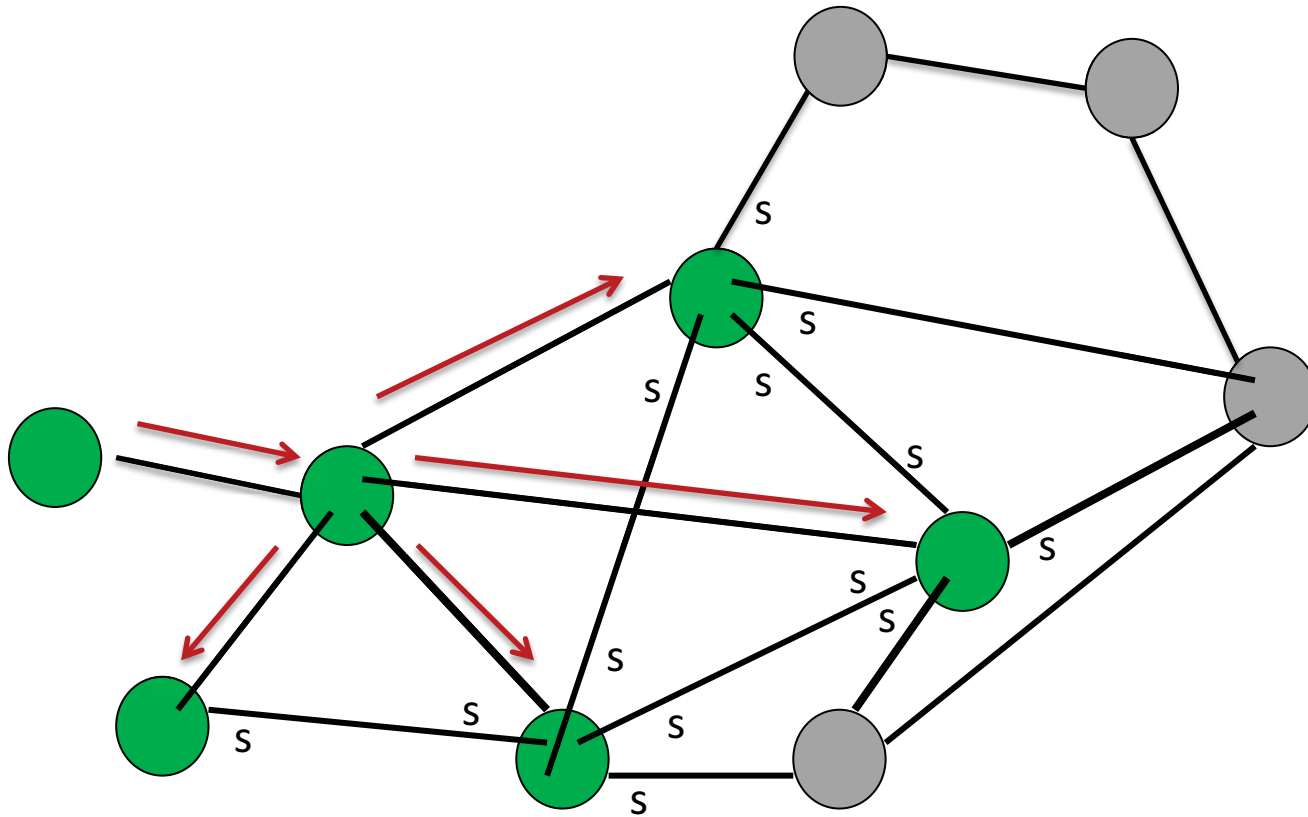




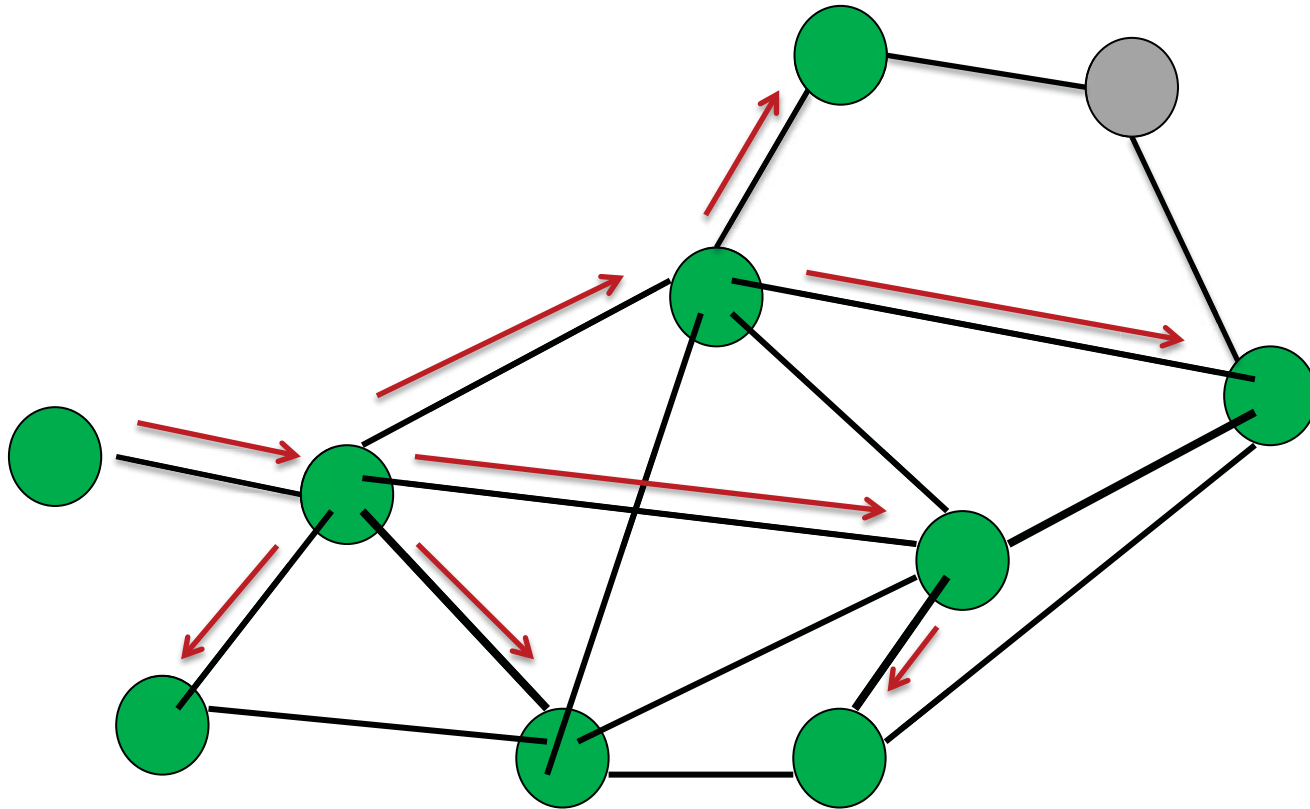
# Round 2



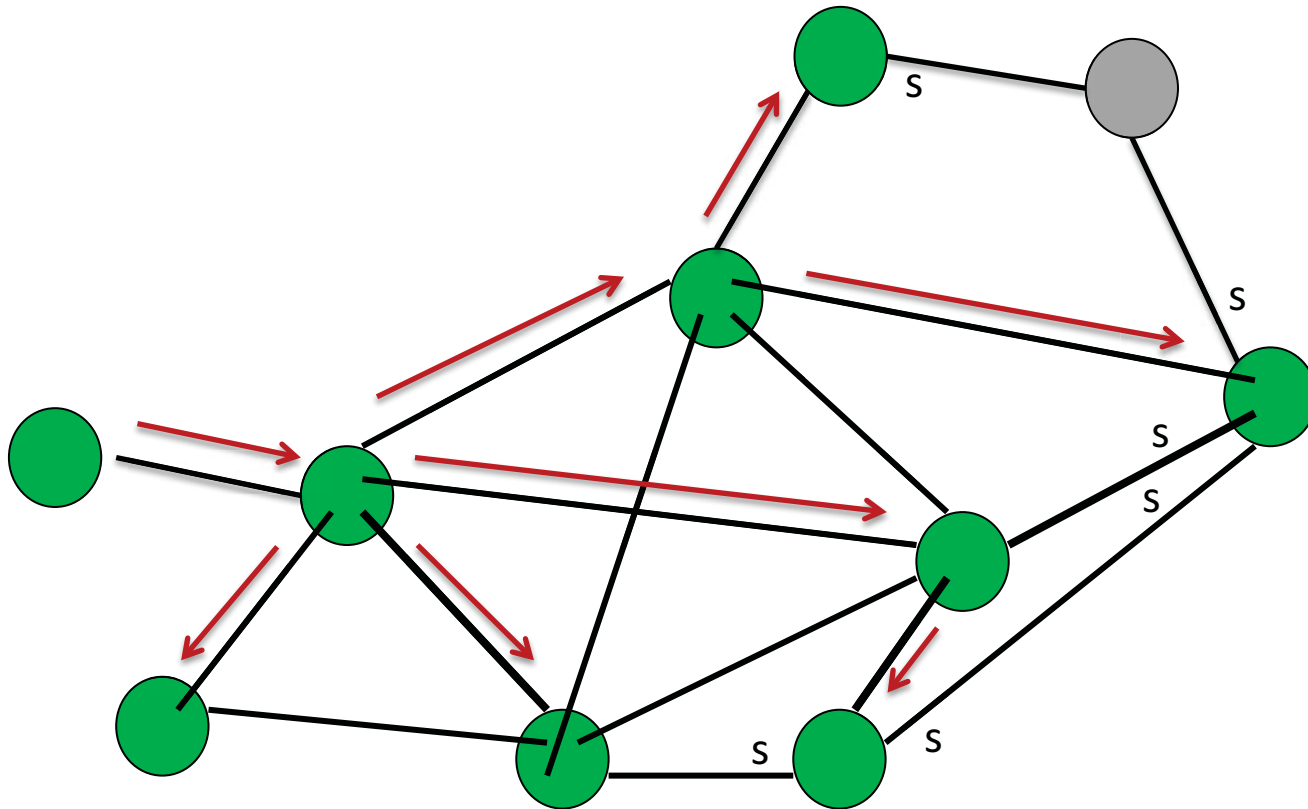
# Round 3



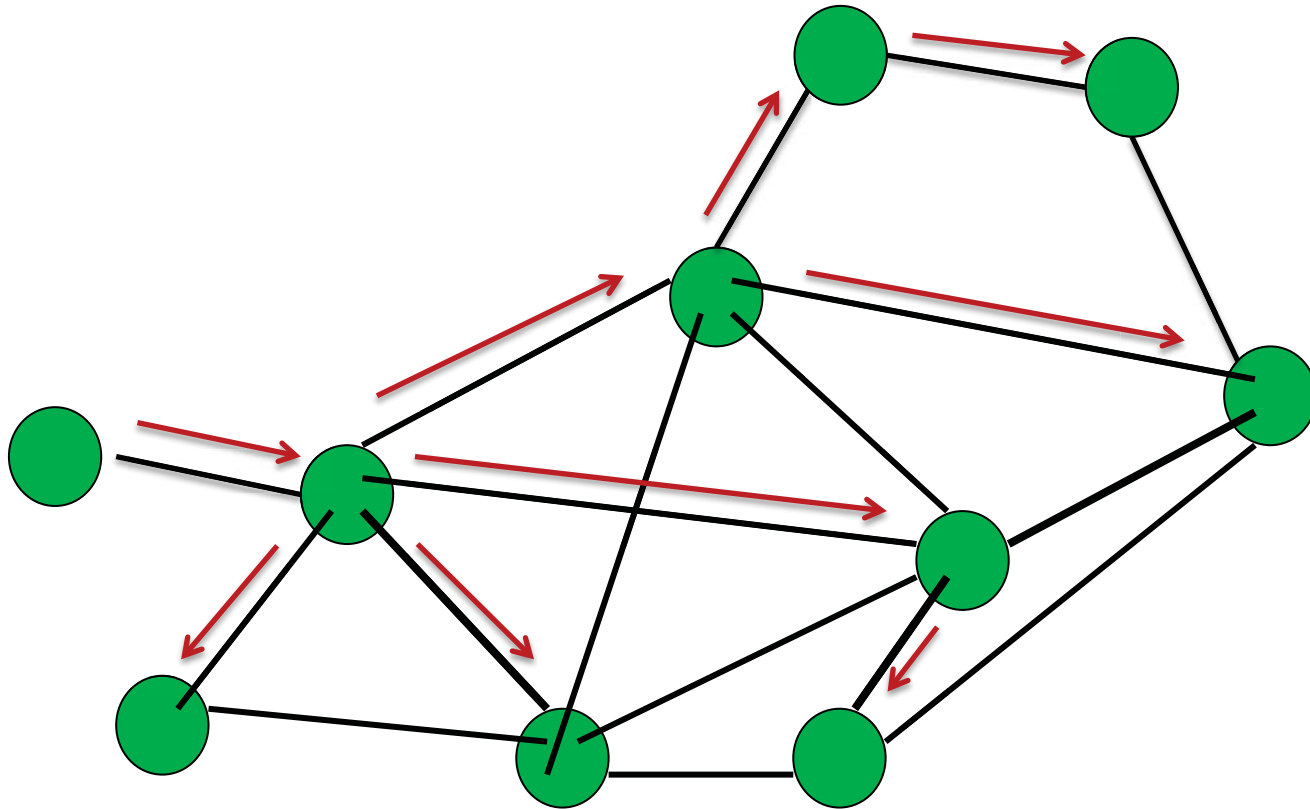
# Round 3



# Round 4



# Round 4



# Correctness

- **Q:** Key invariants describing the state after  $r$  rounds?
- **State variables, per process:**
  - *marked*, a Boolean, initially true for  $i_0$ , false for others
  - *parent*, a UID or undefined
  - *send*, a Boolean, initially true for  $i_0$ , false for others
  - *uid*
- **Invariants:**
  - At the end of  $r$  rounds, exactly the processes at distance  $\leq r$  from  $v_0$  are marked.
  - A process  $\neq i_0$  has its *parent* defined iff it is marked.
  - For any process at distance  $d$  from  $v_0$ , if its *parent* is defined, then it is the UID of a process at distance  $d - 1$  from  $v_0$ .
- Could use these in a formal correctness proof.

# Complexity


- Time complexity:
  - Number of rounds until all nodes outputs their parent information.
  - Maximum distance of any node from  $v_0$ , which is  $\leq diam$
- Message complexity:
  - Number of messages sent by all processes during the entire execution.
  - $O(|E|)$

# Bells and Whistles

- **Child pointers:**
  - So far, each process learns its parent, but not its children.
  - To add child pointers, everyone who receives a *search* message sends back a *parent* or *nonparent* response, at the next round.
- **Distances:**
  - Augment the algorithm so everyone also learns and outputs its distance from  $v_0$ .
  - Each process records its distance from  $v_0$  in a new *dist* variable, initially 0 for process  $i_0$ , and  $\infty$  for everyone else.
  - Include the *dist* value in every *search* message; when an unmarked process receives a *search*( $d$ ) message, it sets its own *dist* to  $d + 1$ .



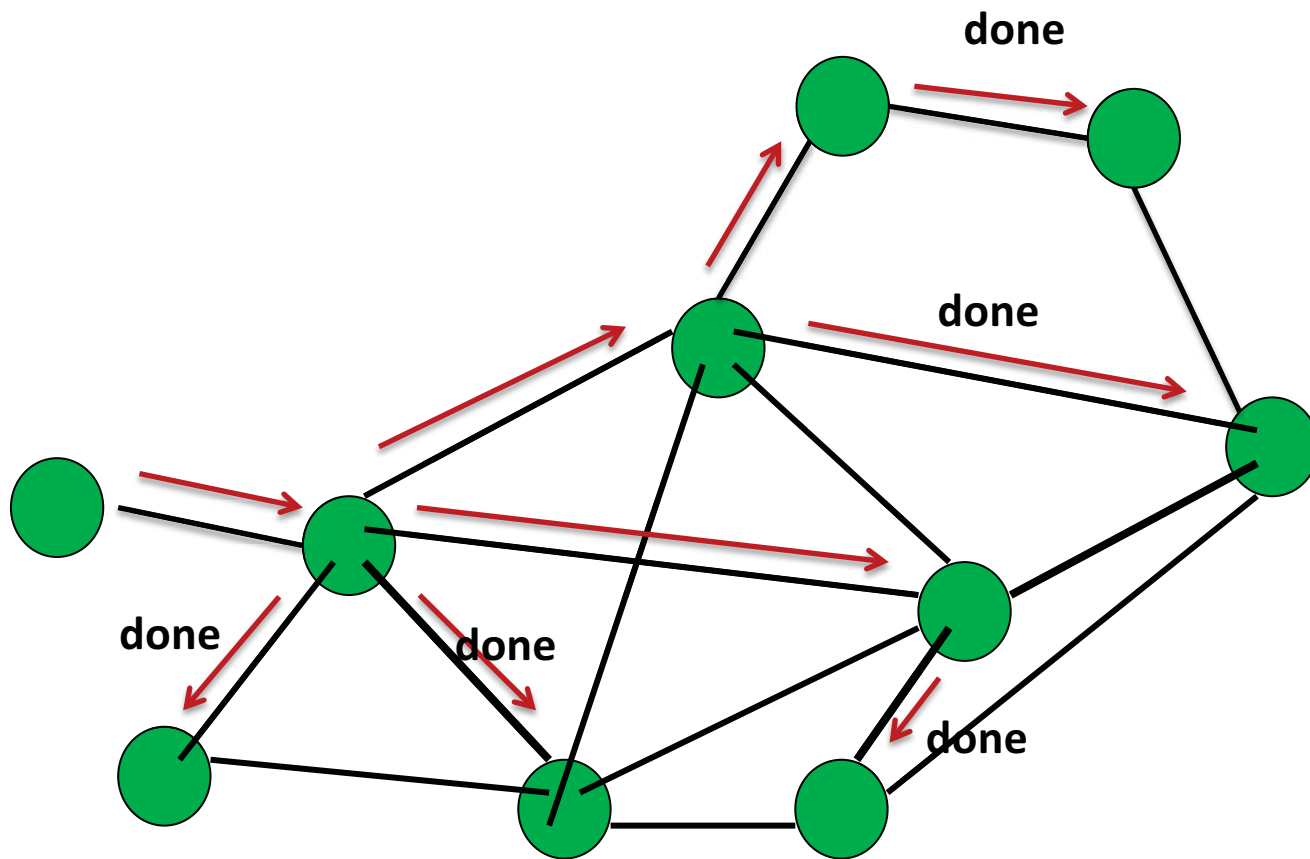
# Termination

- Q: How can processes learn when the BFS tree is completed?
- If they knew an upper bound on  $diam$ , then they could simply wait until that number of rounds have passed.
- Q: What if they don't know anything about the graph?
- Slightly simpler problem: Process  $i_0$  should learn when the tree is done.
- Termination algorithm: 
  - Assume each *search* message receives a response, *parent* or *nonparent*.
  - After a node has received responses to all its outgoing *search* messages, it knows who its children are, and knows they are all marked.
  - The leaves of the tree learn who they are (they receive only *nonparent* responses).
  - Now use a *convergecast* strategy:

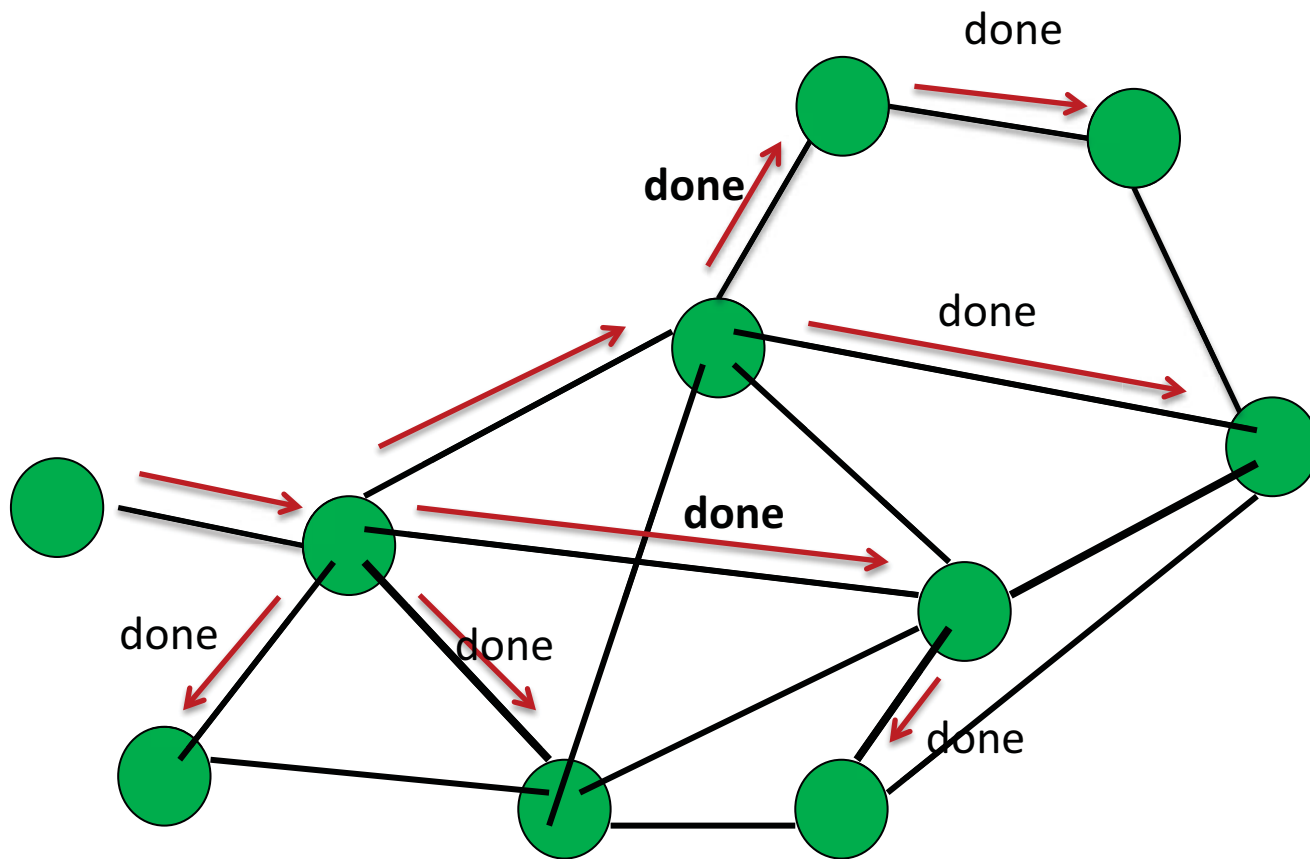
# Termination

- Termination algorithm:
  - After a node has received responses to all its *search* messages, it knows who its children are, and knows they are marked.
  - The leaves of the tree learn who they are.
  - Use a *convergecast* strategy:
    - Starting from the leaves, the processes fan in *done* messages up the BFS tree, toward  $i_0$ .
    - A process  $\neq i_0$  can send a *done* message to its parent after:
      - It has received responses to all its *search* messages (so it knows who its children are), and
      - It has received *done* messages from all its children.
    - $i_0$  knows that the BFS tree is completed, when:
      - It has received responses to all its *search* messages (so it knows who its children are), and
      - It has received *done* messages from all its children.

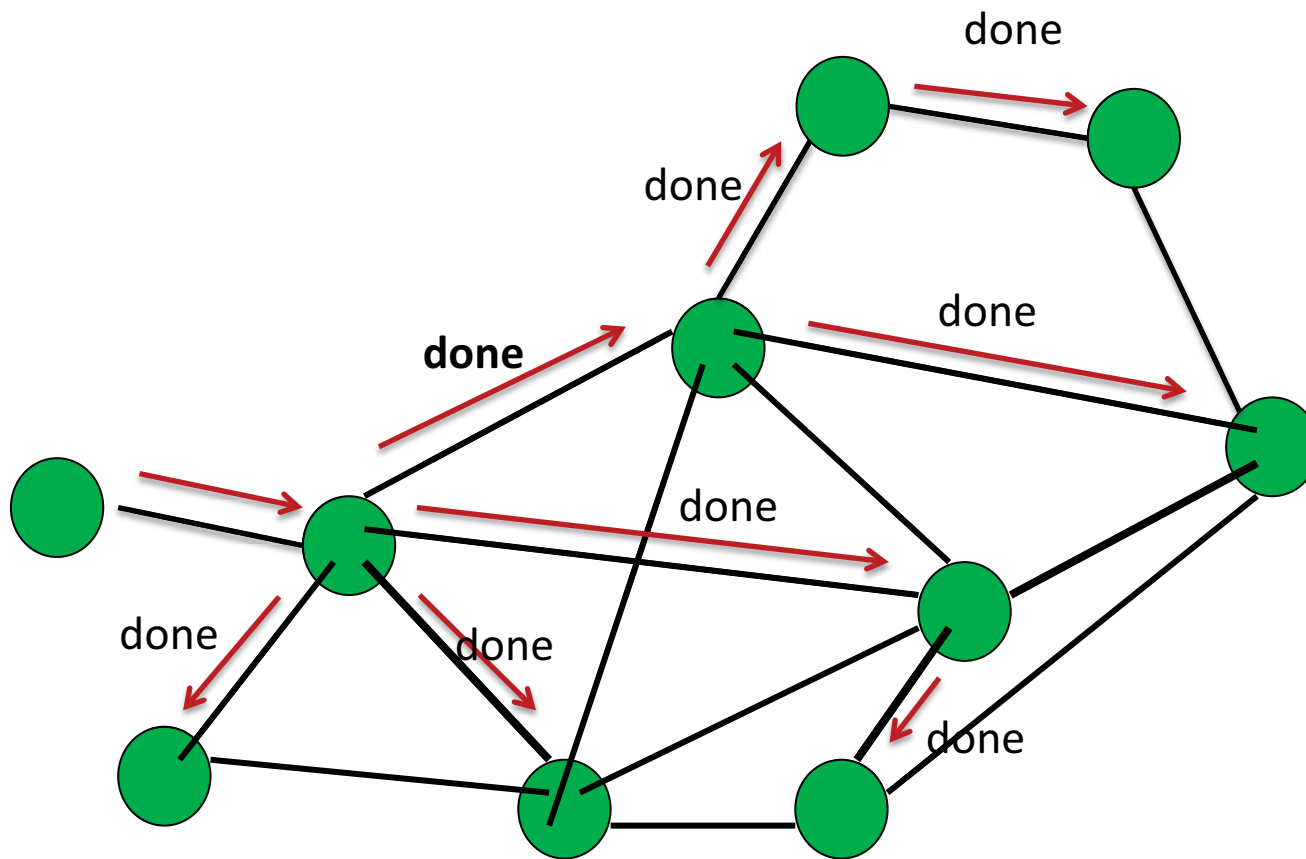
# Termination Using Convergecast



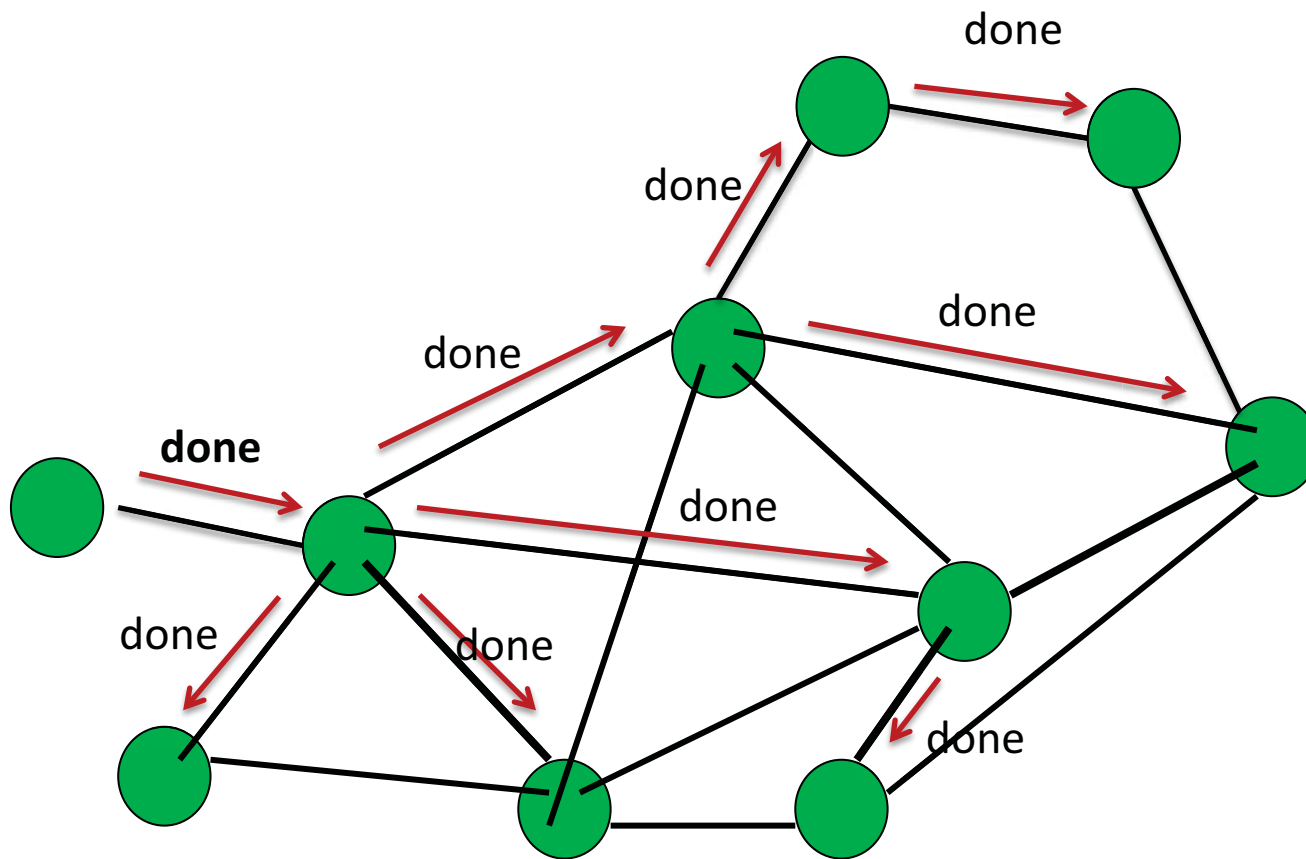
# Termination Using Convergecast



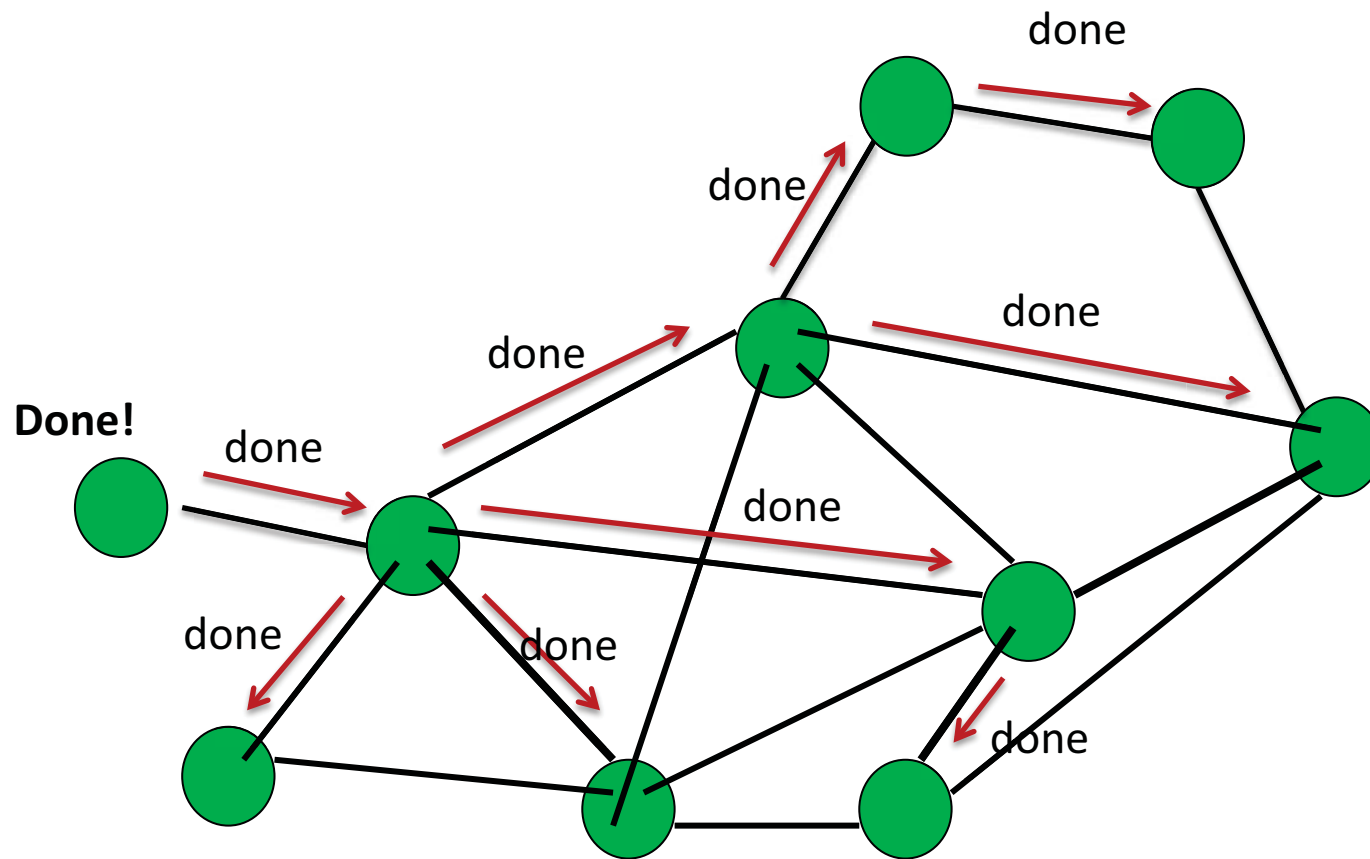
# Termination Using Convergecast



# Termination Using Convergecast



# Termination Using Convergecast



# Complexity for Termination

- After the tree is done, it takes  $\leq diam$  rounds and  $n$  messages for the *done* information to reach  $i_0$ .
- **Q:** How can all the processes learn that the BFS tree is completed?
- Process  $i_0$  can broadcast a message to everyone along the edges of the tree.
- Takes an additional  $diam$  rounds and  $n$  messages.



# Applications of Breadth-First Spanning Trees

- **Message broadcast:**

- Process  $i_0$  can use a BFS tree with child pointers to broadcast a sequence of messages to all processes in the network.
- Each takes  $diam$  rounds and  $n$  messages.
- Could pipeline them, so  $k$  messages take only  $diam + k$  rounds, not  $diam \times k$ .

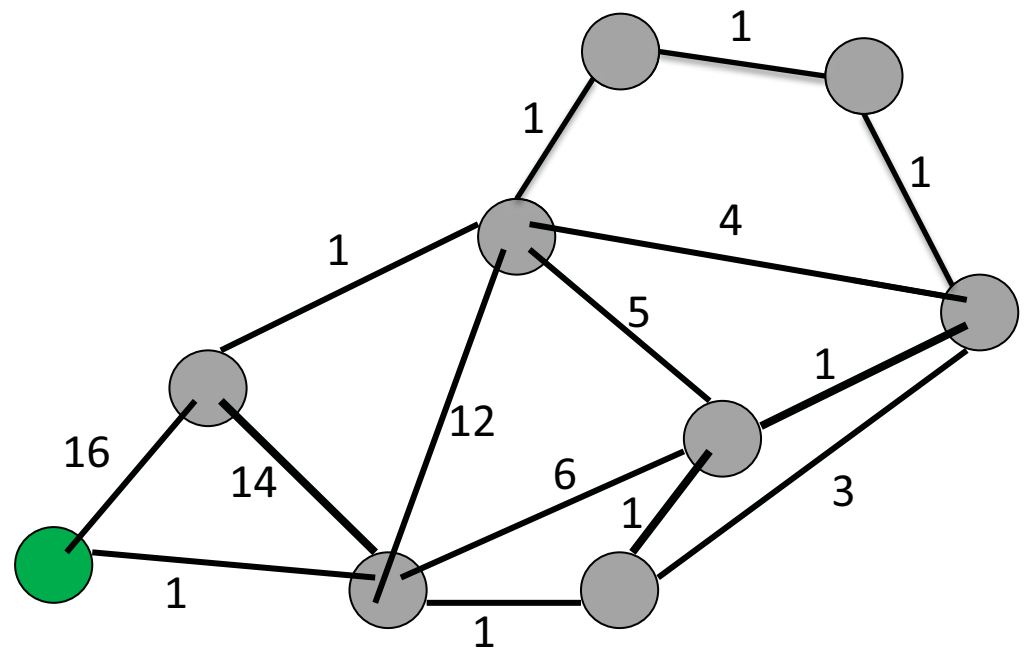
- **Global computation:**

- Suppose every process starts with some initial value, and process  $i_0$  should determine the value of some function of the set of all processes' values.
  - min, max, sum, average,...
- Can use convergecast on a BFS tree.
- Cost is  $diam$  rounds and  $n$  messages.
- In general, messages may be large, but for many functions, some data aggregation is possible along the way.

# Application of the BFS construction

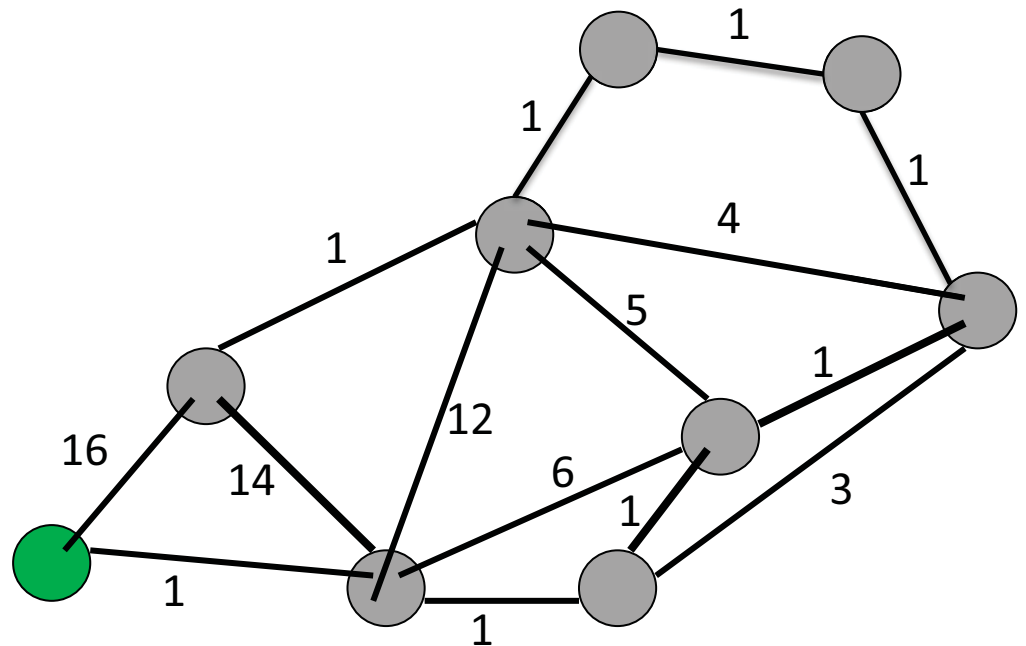
- Leader election in a general graph:
  - No distinguished process  $i_0$ .
  - Processes don't know anything about the graph.
  - Everyone can start its own BFS, acting as the root.
  - Use this to determine the maximum UID; process with the max UID is the leader.
  - Cost is  $O(diam)$  rounds,  $O(n |E|)$  messages.

# Shortest Paths Trees



# Shortest Paths

- Generalize the BFS problem to allow weights on the graph edges, *weight*<sub>{u,v}</sub> for edge {u, v}
- Connected graph  $G = (V, E)$ , root vertex  $v_0$ , process  $i_0$ .
- Processes have UIDs.
- Processes know their neighbors and the weights of their incident edges, but otherwise have no knowledge about the graph.



# Shortest Paths

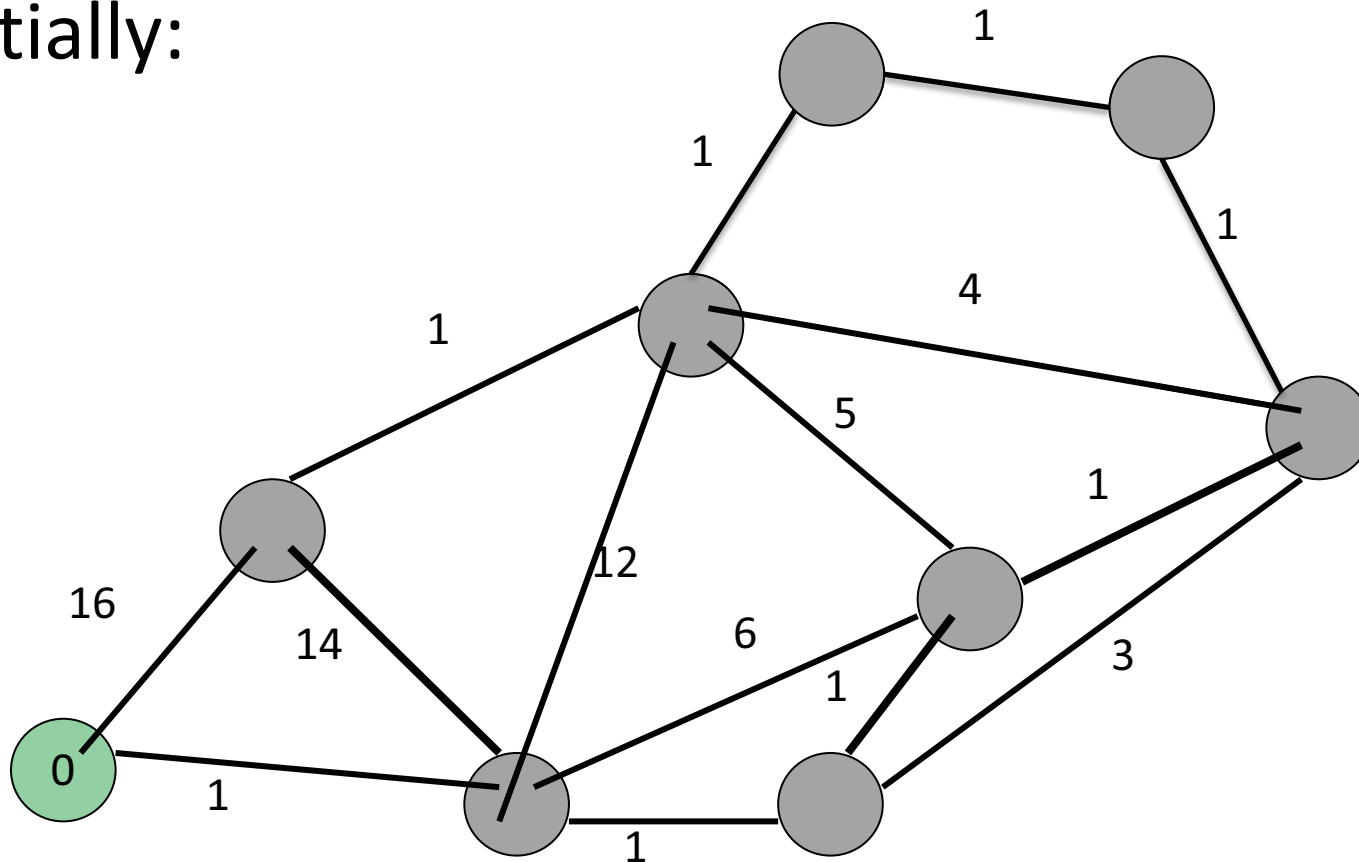
- Processes must produce a Shortest-Paths Spanning Tree rooted at vertex  $v_0$ .
- Branches are directed paths from  $v_0$ .
  - **Spanning:** Branches reach all vertices.
  - **Shortest paths:** The total weight of the tree branch to each node is the minimum total weight for any path from  $v_0$  in  $G$ .
- **Output:** Each process  $i \neq i_0$  should output  $parent(j), distance(d)$ , meaning that:
  - $j$ 's vertex is the parent of  $i$ 's vertex on a shortest path from  $v_0$ ,
  - $d$  is the total weight of a shortest path from  $v_0$  to  $j$ .

# Bellman-Ford Shortest Paths Algorithm

- **State variables:**
  - *dist*, a nonnegative real or  $\infty$ , representing the shortest known distance from  $v_0$ . Initially 0 for process  $i_0$ ,  $\infty$  for the others.
  - *parent*, a UID or undefined, initially undefined.
  - *uid*
- **Algorithm for process  $i$ :**
  - At each round:
    - Send a *distance*(*dist*) message to all neighbors.
    - Receive messages from neighbors; let  $d_j$  be the distance received from neighbor  $j$ .
    - Perform a **relaxation step**:  
$$\textit{dist} := \min(\textit{dist}, \min_j (d_j + \textit{weight}_{\{i,j\}})).$$
    - If *dist* decreases then set *parent*  $:= j$ , where  $j$  is any neighbor that produced the new *dist*.

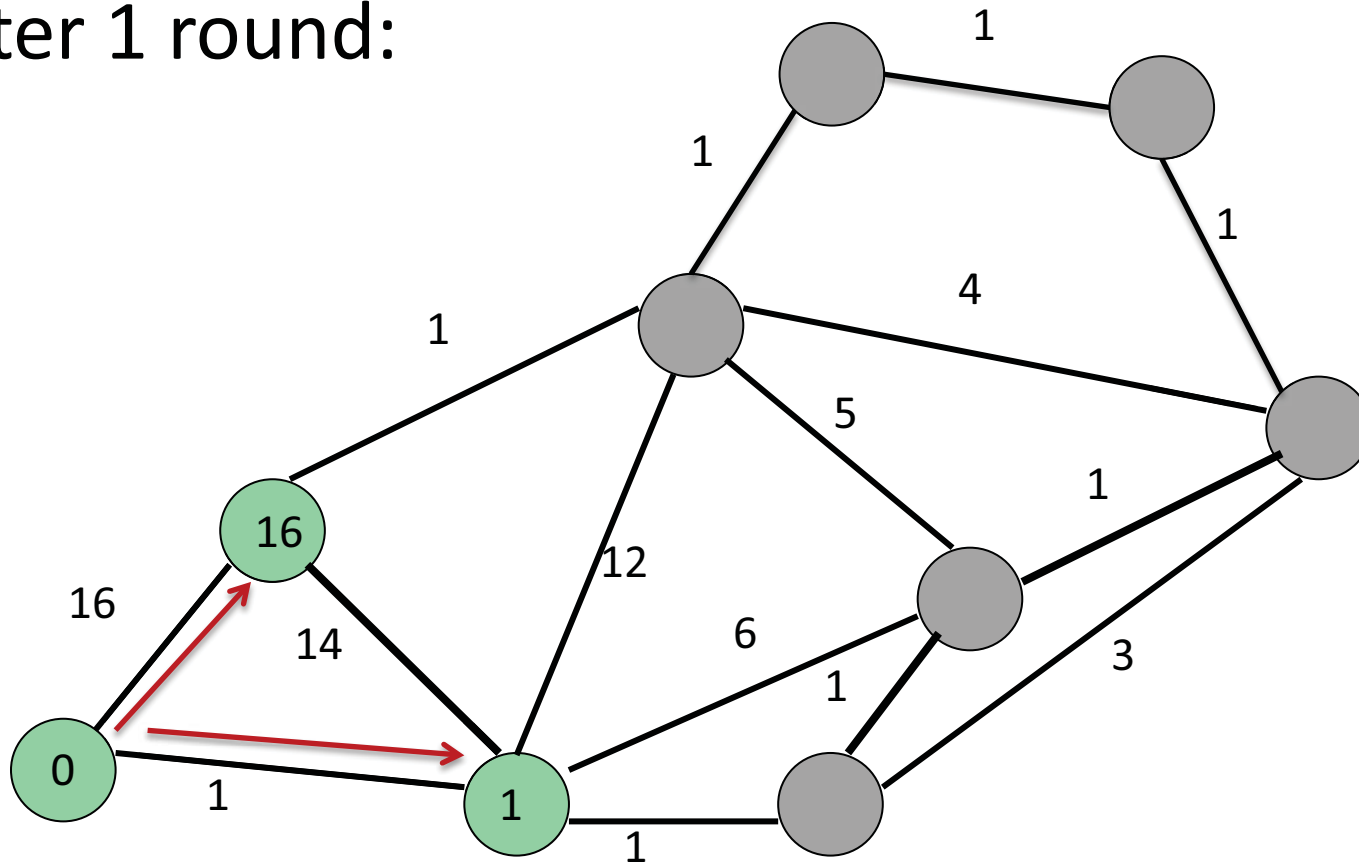
# Example: Bellman-Ford

- Initially:



# Bellman-Ford

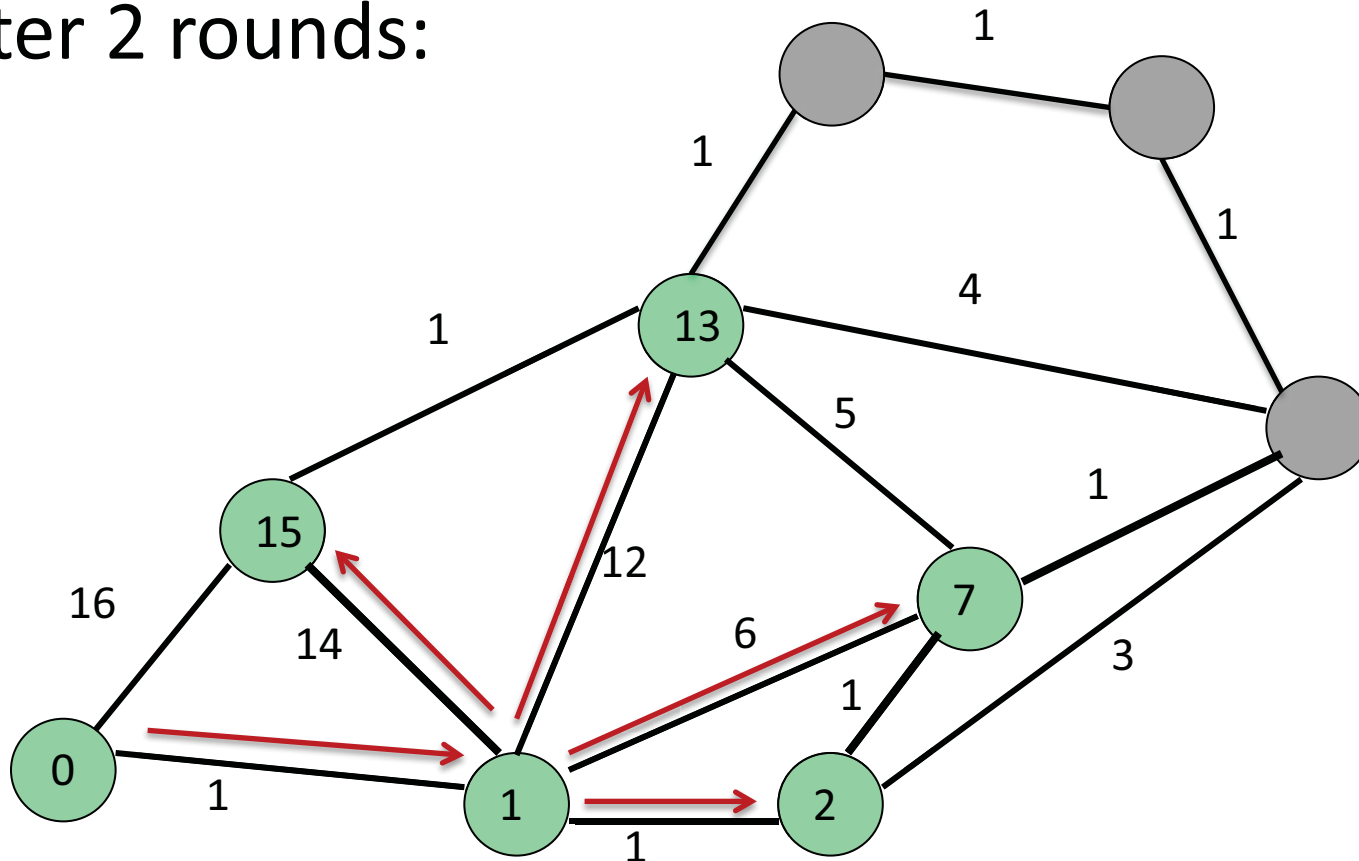
- After 1 round:





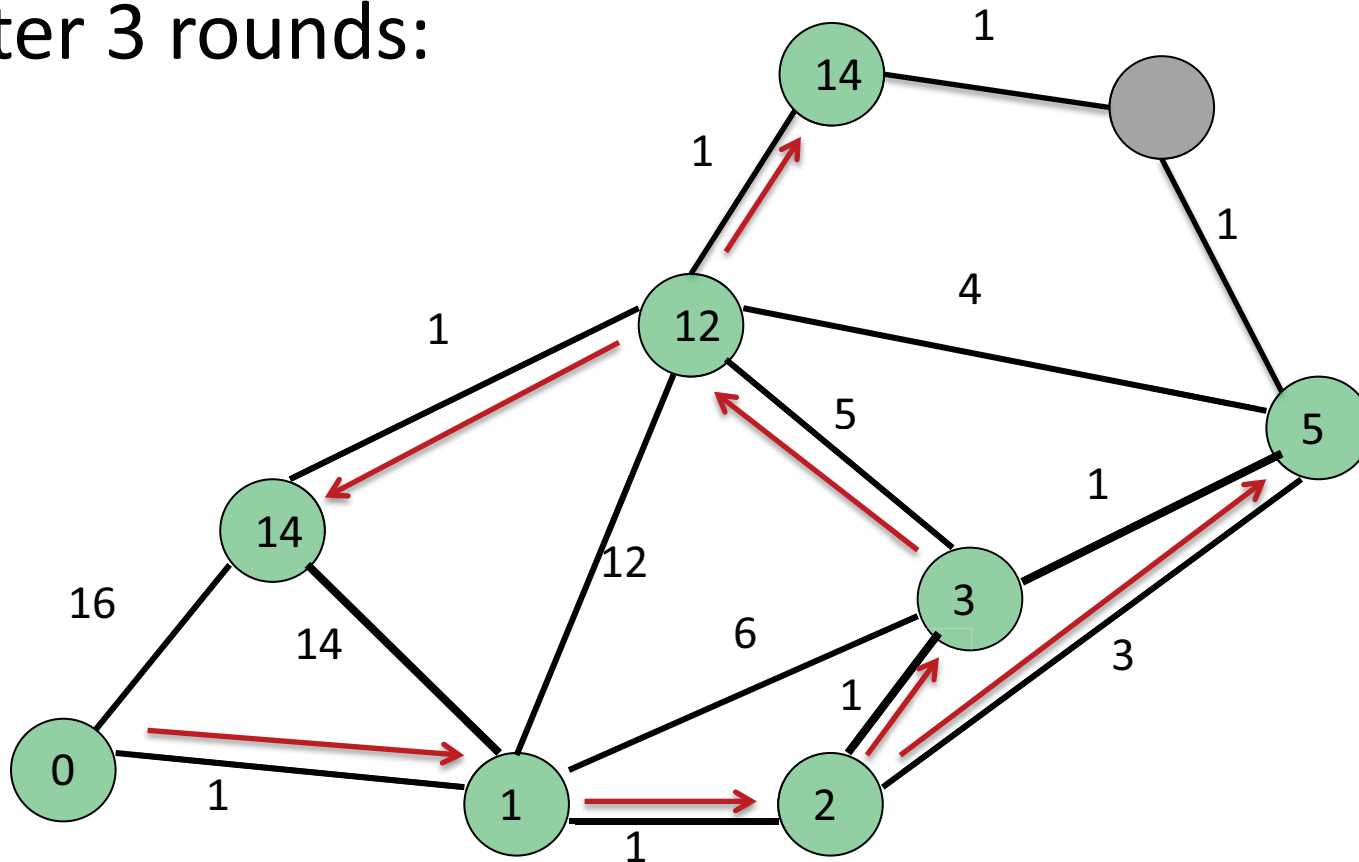
# Bellman-Ford

- After 2 rounds:



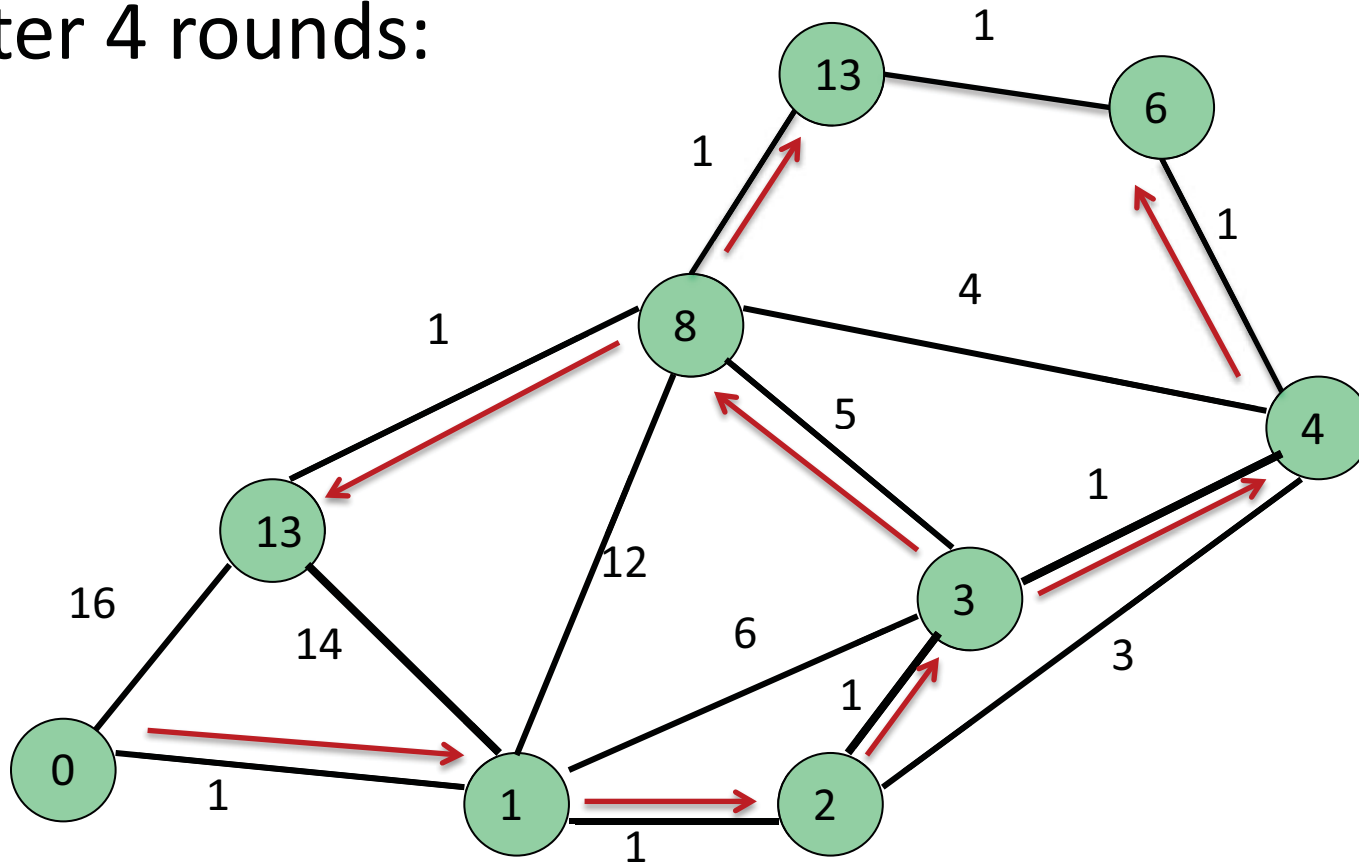
# Bellman-Ford

- After 3 rounds:



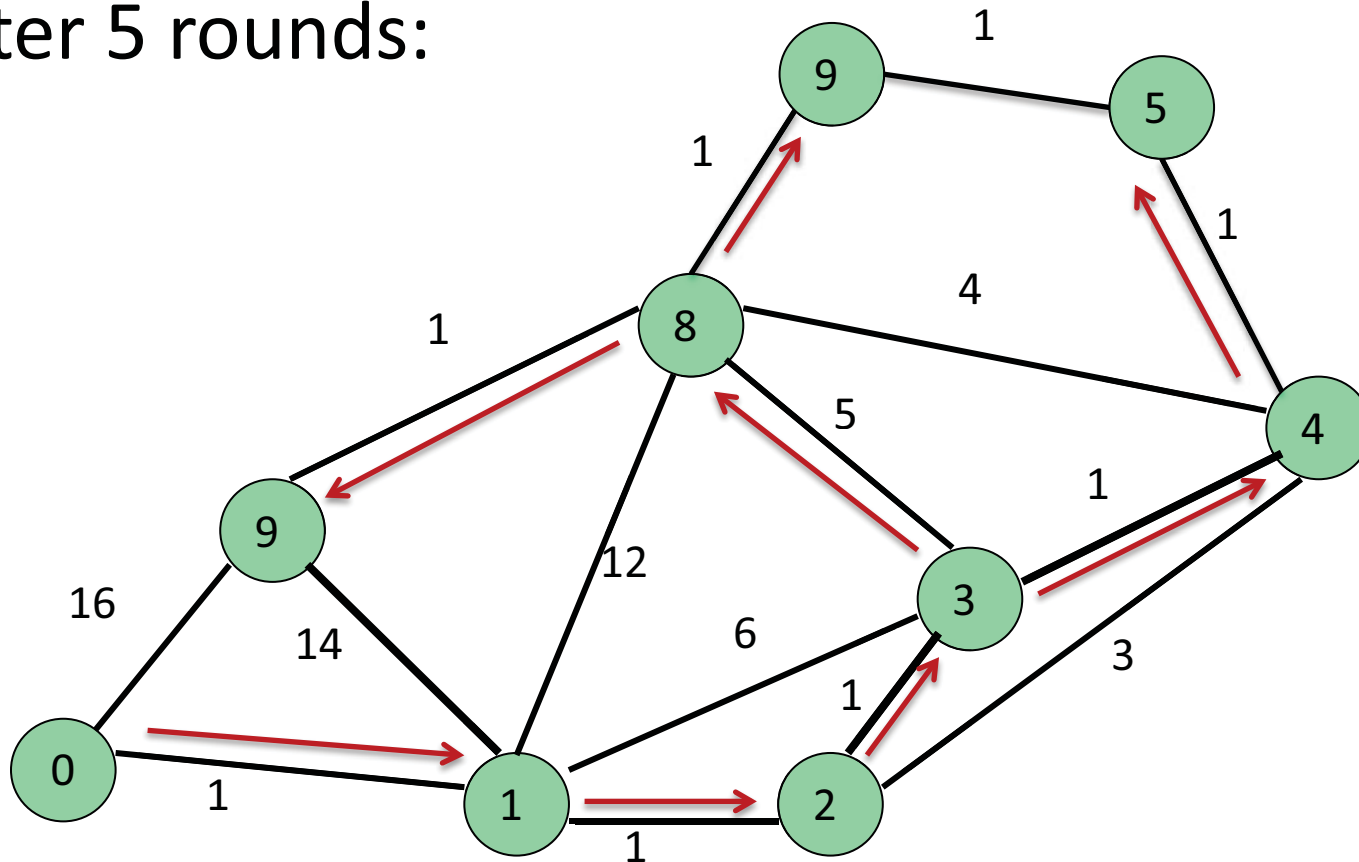
# Bellman-Ford

- After 4 rounds:



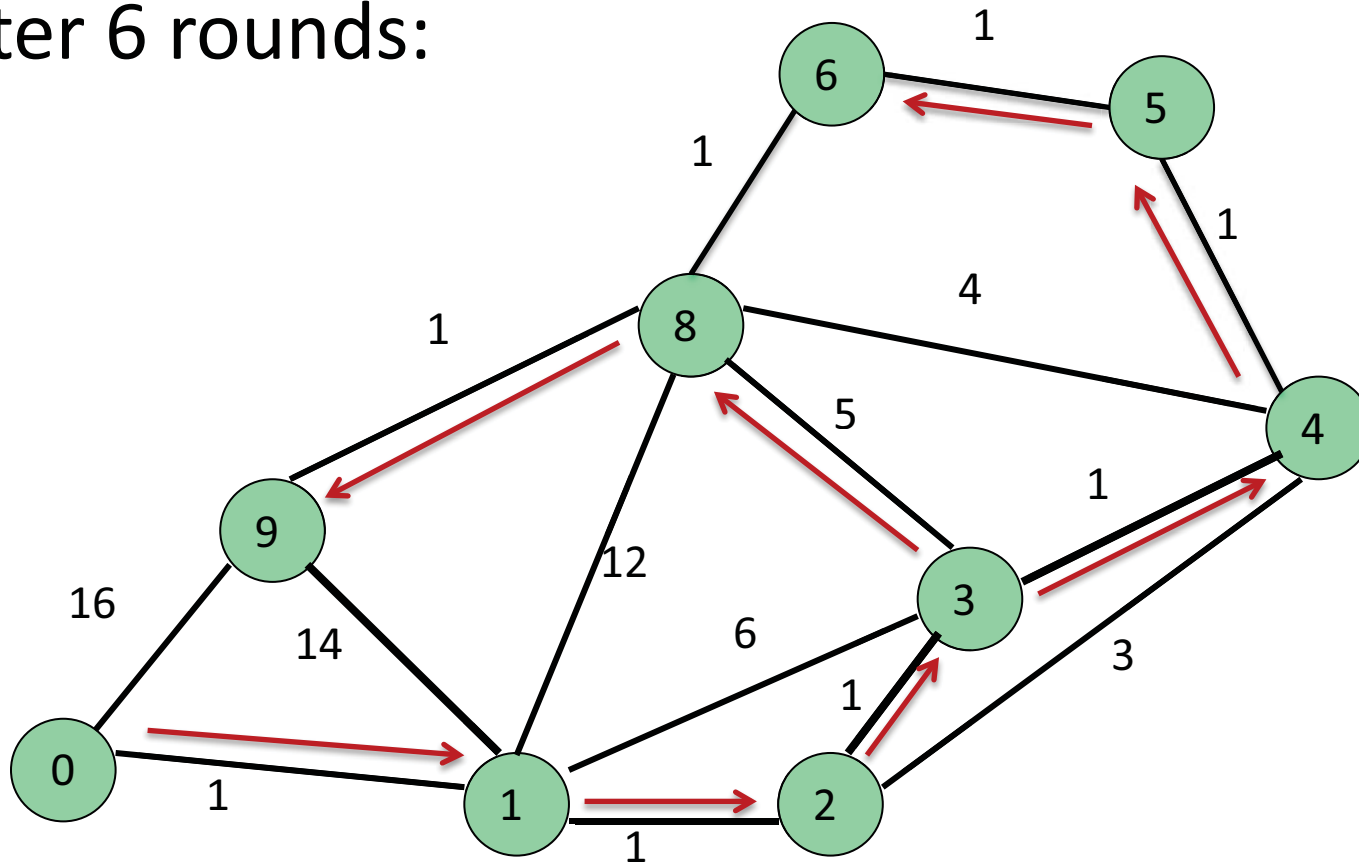
# Bellman-Ford

- After 5 rounds:



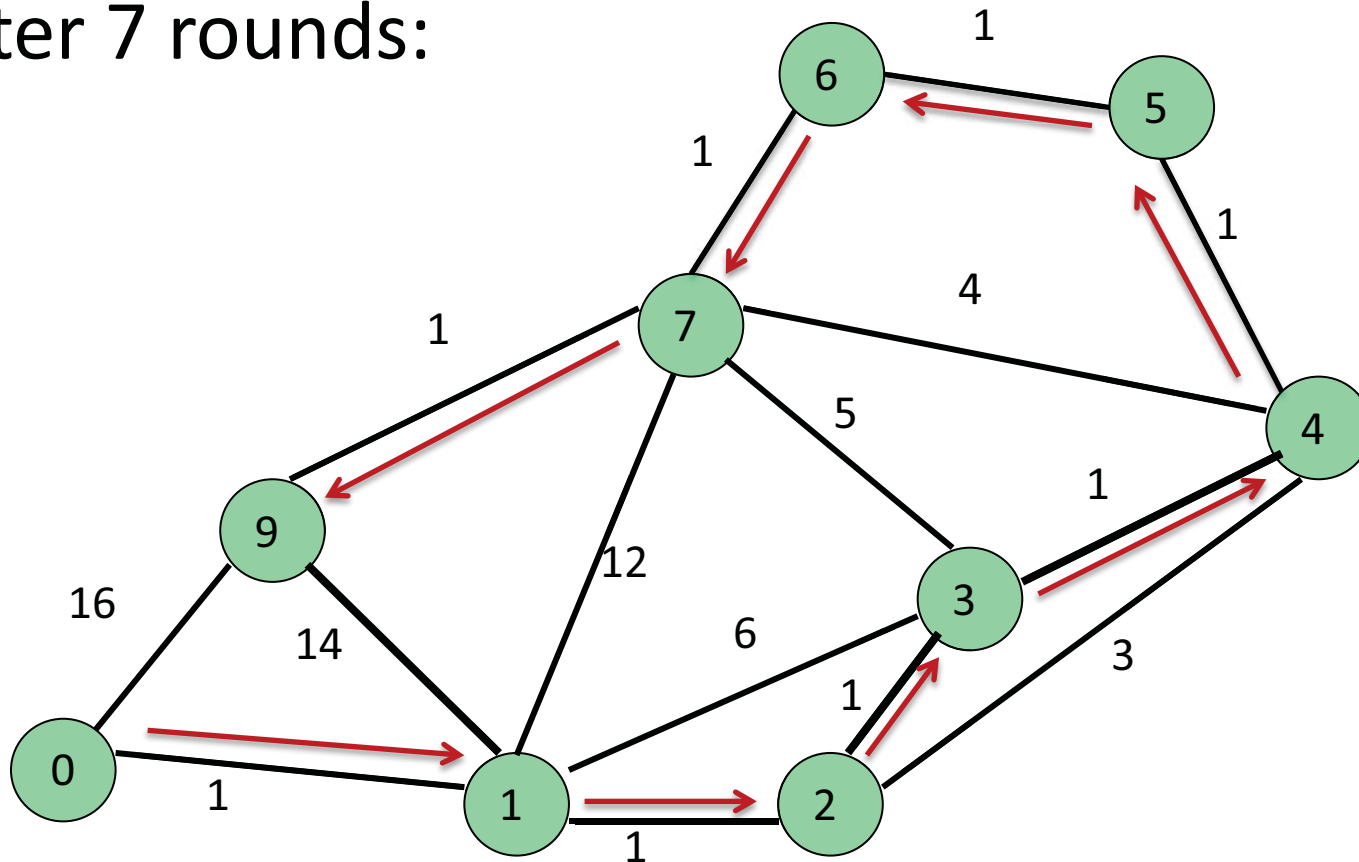
# Bellman-Ford

- After 6 rounds:



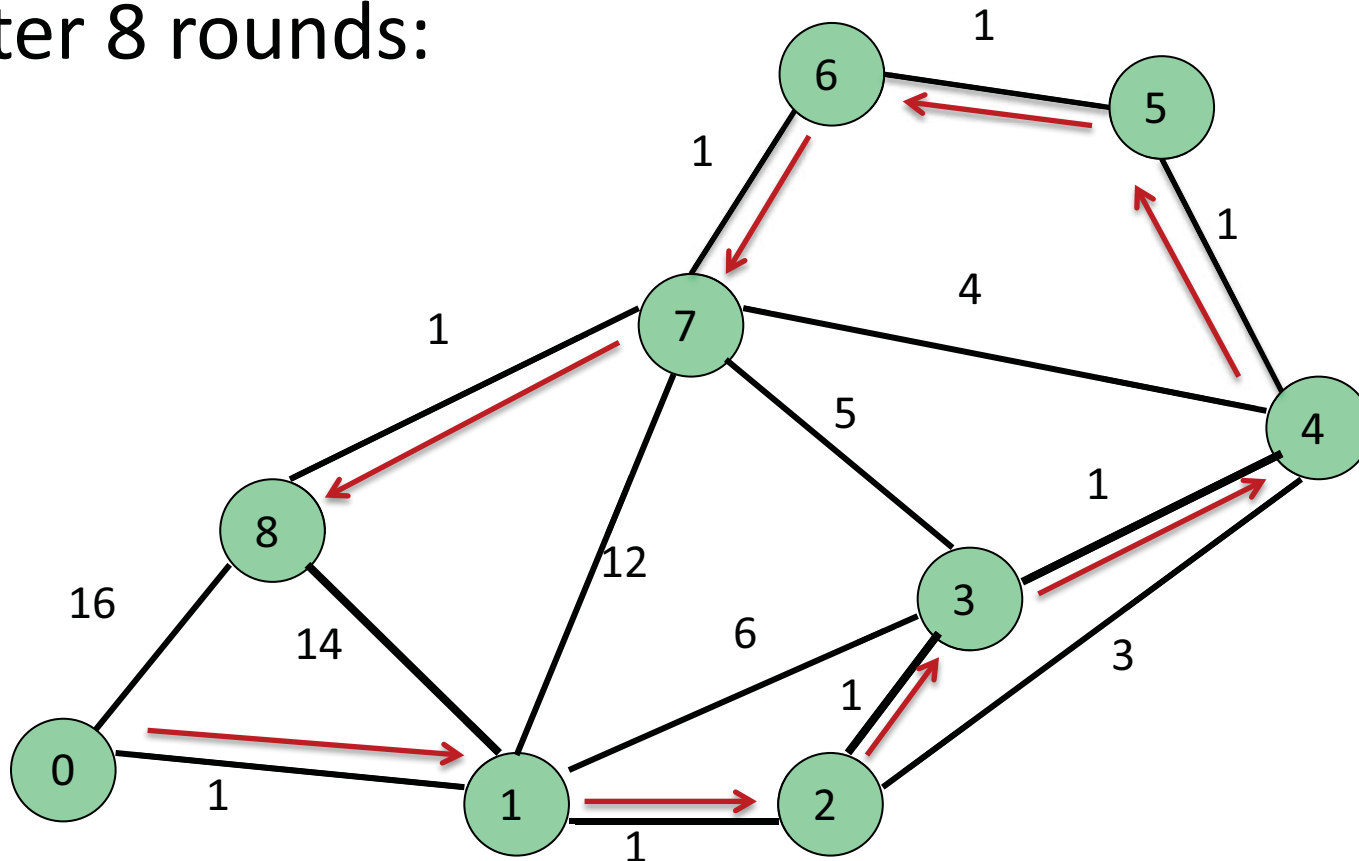
# Bellman-Ford

- After 7 rounds:



# Bellman-Ford

- After 8 rounds:



# Correctness

- **Claim:** Eventually, every process  $i$  has:
  - $dist$  = minimum weight of a path from  $i_0$  to  $i$ , and
  - if  $i \neq i_0$ ,  $parent$  = the previous node on some shortest path from  $i_0$  to  $i$ .
- **Key invariant:**
  - For every  $r$ , at the end of  $r$  rounds, every process  $i \neq i_0$  has its  $dist$  and  $parent$  corresponding to a shortest path from  $i_0$  to  $i$  among those paths that consist of at most  $r$  edges; if there is no such path, then  $dist = \infty$  and  $parent$  is undefined.



# Complexity

- **Time complexity:**
  - Number of rounds until all the variables stabilize to their final values.
  - $n - 1$  rounds
- **Message complexity:**
  - Number of messages sent by all processes during the entire execution.
  - $O(n \cdot |E|)$
- More expensive than BFS:
  - *diam* rounds,
  - $O(|E|)$  messages
- **Q:** Does the time bound really depend on  $n$ ?

# Child Pointers

- Ignore repeated messages.
- When process  $i$  receives a message that it does not use to improve *dist*, it responds with a *nonparent* message.
- When process  $i$  receives a message that it uses to improve *dist*, it responds with a *parent* message, and also responds to any previous parent with a *nonparent* message.
- Process  $i$  records nodes from which it receives *parent* messages in a set *children*.
- When process  $i$  receives a *nonparent* message from a current child, it removes the sender from its *children*.
- When process  $i$  improves *dist*, it empties *children*.

# Termination

- Q: How can the processes learn when the shortest-paths tree is completed?
- Q: How can a process know when it can output its own *parent* and *distance*?
- If processes knew an upper bound on  $n$ , then they could simply wait until that number of rounds have passed.
- But what if they don't know anything about the graph?
- Recall termination for BFS: Used **convergecast**.
- Q: Does that work here?

# Termination

- **Q:** How can the processes learn when the shortest-paths tree is completed?
- **Q:** Does convergecast work here?
- Yes, but it's trickier, since the tree structure changes.
- **Key ideas:**
  - A process  $\neq i_0$  can send a *done* message to its current parent after:
    - It has received responses to all its *distance* messages, so it believes it knows who its children are, and
    - It has received *done* messages from all of those children.
  - The same process may be involved several times in the convergecast, based on improved estimates.

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.046J / 18.410J Design and Analysis of Algorithms  
Spring 2015

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.