An Optimal Distributed Edge-Biconnectivity Algorithm

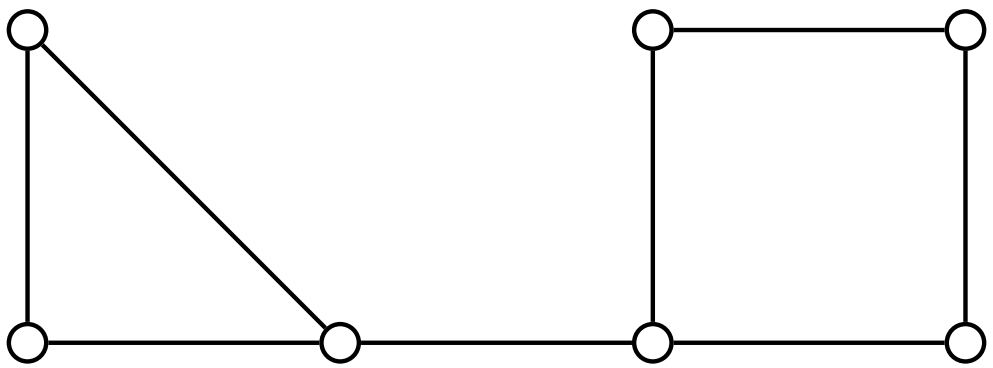
David Pritchard

Algorithms and Complexity, 2006

Outline

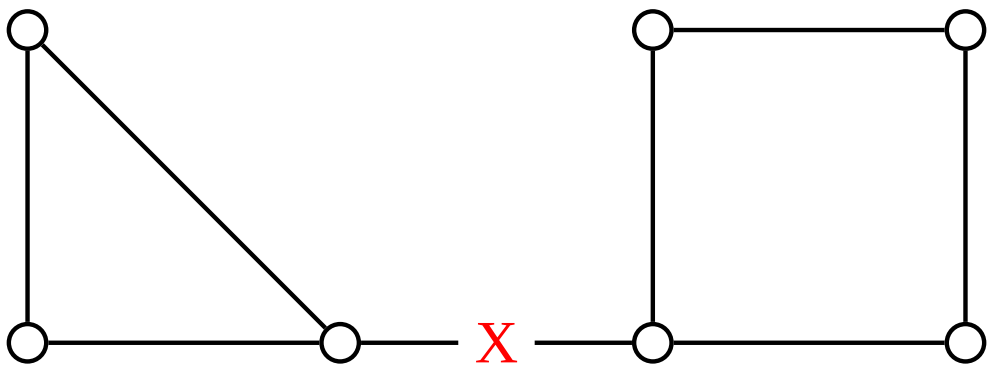
Reliable Networks

- Model a network by a graph $G = (V, E)$: nodes are computers and edges are two-way communication channels.
- A graph (network) is *connected* if every pair of nodes is connected by some path of edges. Otherwise we say the network is *disconnected*.
- Connectedness is very desirable, and necessary if we want global information about our distributed network.
- Sometimes the removal (or equivalently, failure) of a *single* edge or node can cause a connected network to become disconnected.



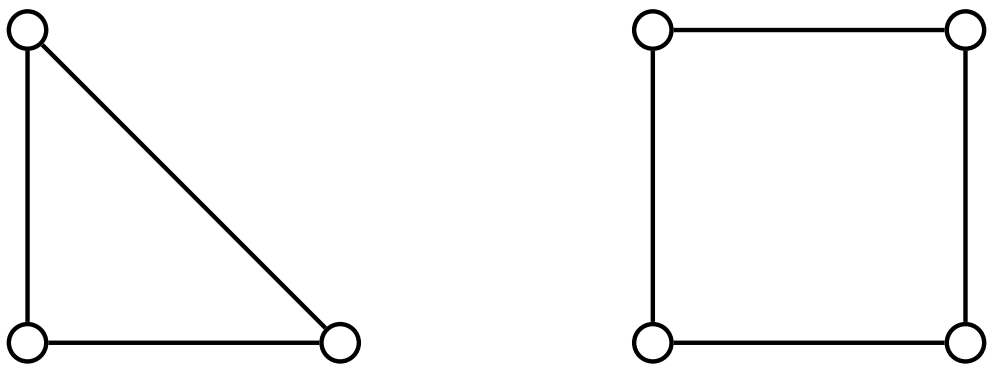
Reliable Networks

- Model a network by a graph $G = (V, E)$: nodes are computers and edges are two-way communication channels.
- A graph (network) is *connected* if every pair of nodes is connected by some path of edges. Otherwise we say the network is *disconnected*.
- Connectedness is very desirable, and necessary if we want global information about our distributed network.
- Sometimes the removal (or equivalently, failure) of a *single* edge or node can cause a connected network to become disconnected.



Reliable Networks

- Model a network by a graph $G = (V, E)$: nodes are computers and edges are two-way communication channels.
- A graph (network) is *connected* if every pair of nodes is connected by some path of edges. Otherwise we say the network is *disconnected*.
- Connectedness is very desirable, and necessary if we want global information about our distributed network.
- Sometimes the removal (or equivalently, failure) of a *single* edge or node can cause a connected network to become disconnected.



The Connectivity of a Graph

- We would like to characterize, for a given graph, how many failures can it tolerate and still remain connected?
- Definition: a graph is *k-edge-connected* if, despite the failure of any $(k-1)$ edges, the graph still remains connected.
- Replacing "edge" by "node" gives definition of *k-node-connected*.
- For both cases, a graph is 1-connected iff it is connected.
- The (edge or node) *connectivity* of a graph is the largest k for which it is k-(edge or node)-connected. (An exception, by convention, is that K_n has node connectivity $n-1$, not ∞.)

Our Distributed Model

- All nodes begin with distinct identifiers.
- Initially nodes do not know the network size, or the identities of their neighbours, but have a fixed list of neighbouring edges.
- Nodes have unbounded computational power.
- Communication takes place in synchronous rounds: on round i, each node reads the messages sent to it by its neighbours in round $i-1$, performs some computation, and then sends up to one message to each neighbour.
- Initially a single *leader* node is "jumpstarted" to initiate the algorithm.
- We restrict our attention to algorithms where all messages are $O(\log n)$ bits in size.
- Peleg's 2000 book is a good reference — this model is called $\mathcal{CONGEST}$.

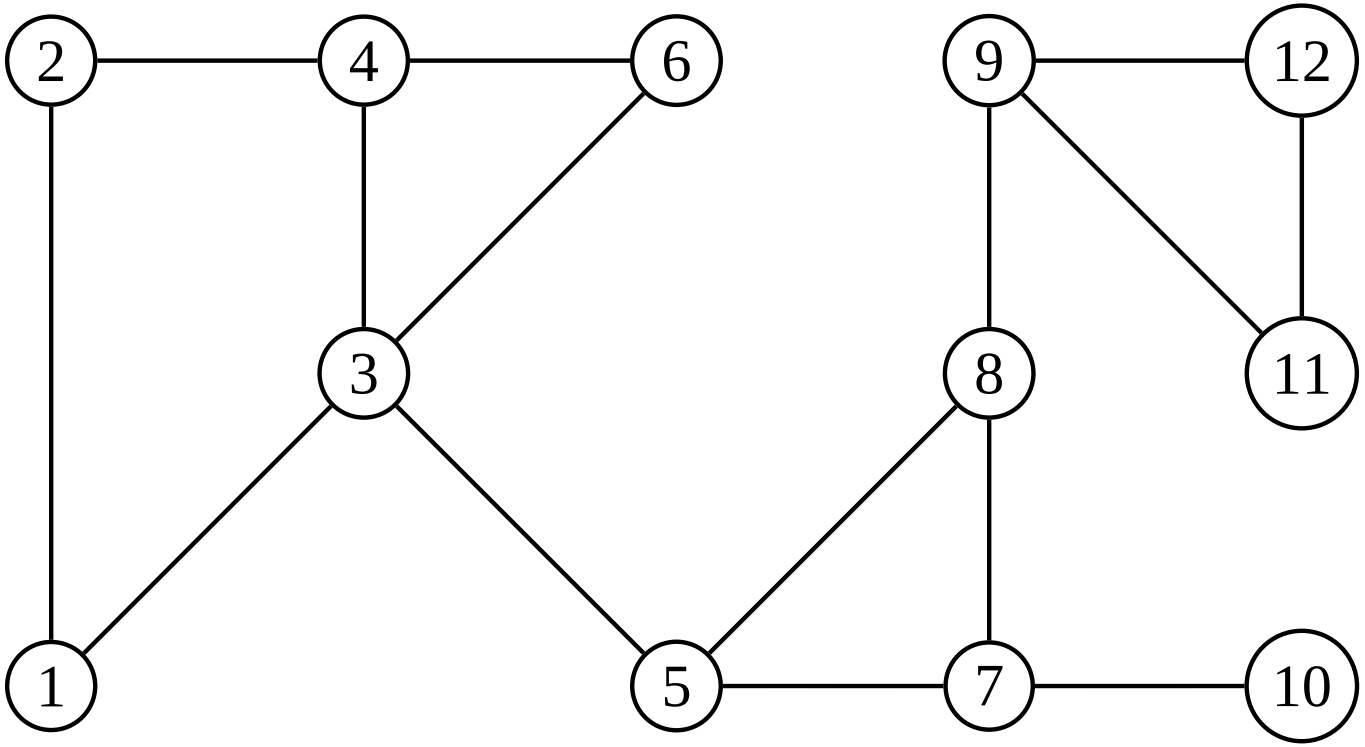
Our Distributed Model

- You may think of each node v as a Turing machine with $\deg(v)$ input tapes, $\deg(v)$ output tapes, and a read-only tape containing its ID.
- Two interesting measures of complexity.
- *Time complexity:* number of rounds elapsed before the algorithm completes.
- *Communication complexity:* number of messages sent before the algorithm completes.

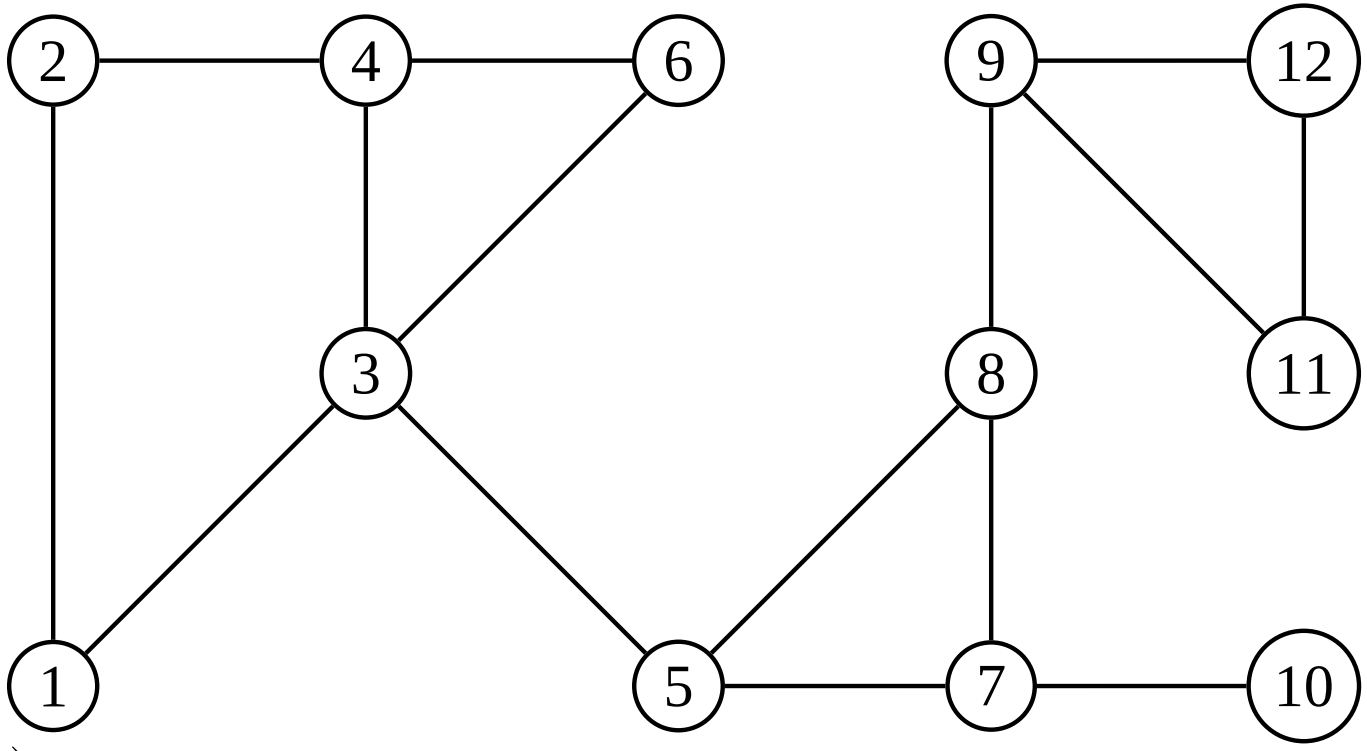
Example Algorithm: BFS

- We represent a BFS tree in the network by each node storing the identifier of its parent. Initially $parent(v) = nil$ for each node v.
- The leader joins the tree in the first round. Non-leader node v joins the tree when its parent is set to a non-null value.
- When node v joins the tree it sends "node v joined" to each neighbour.
- When a non-leader node v that is not in the tree receives one or more "node w_i joined" messages it picks one w_i arbitrarily, joins the tree, and sets $parent(v) := w_i$.

Example Algorithm: BFS

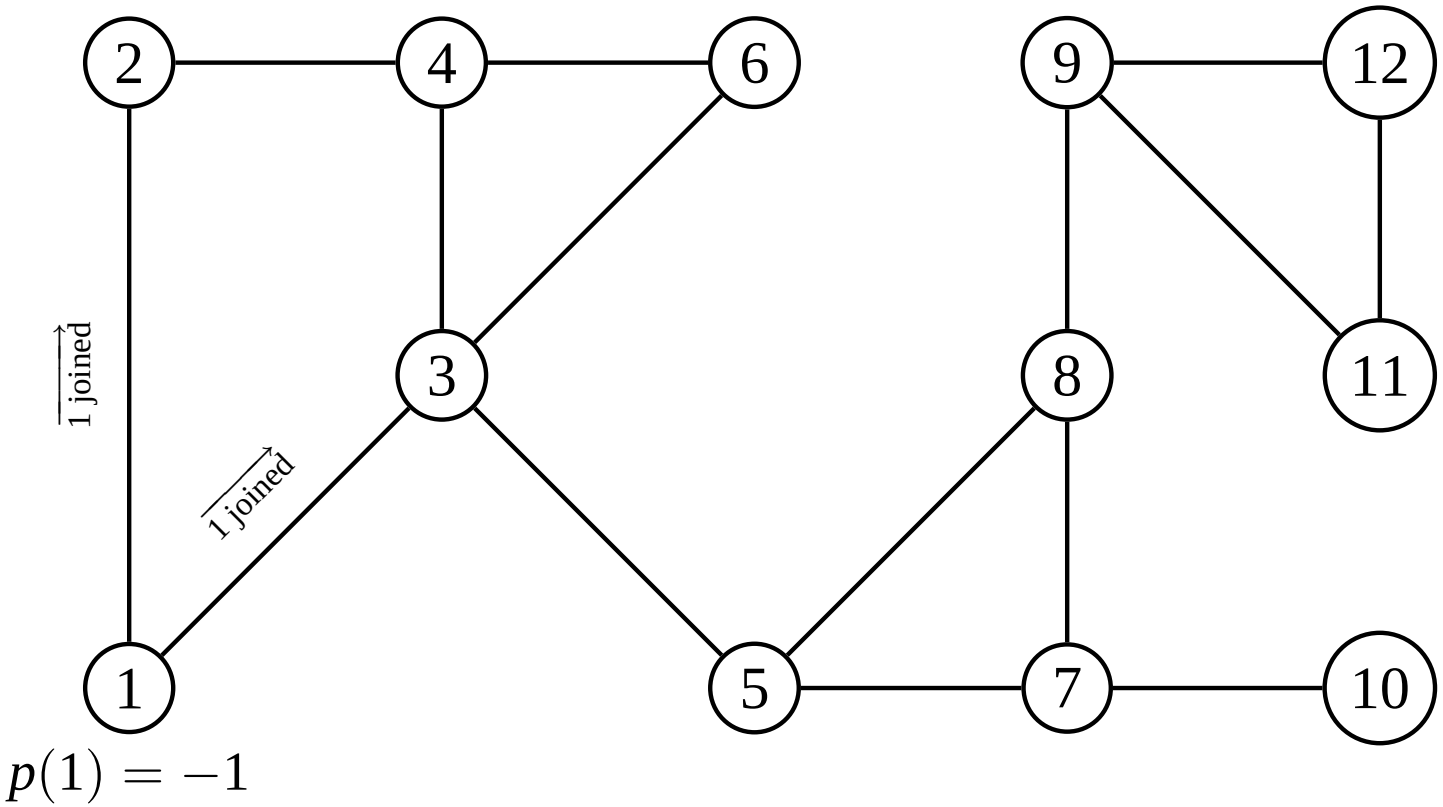


Example Algorithm: BFS

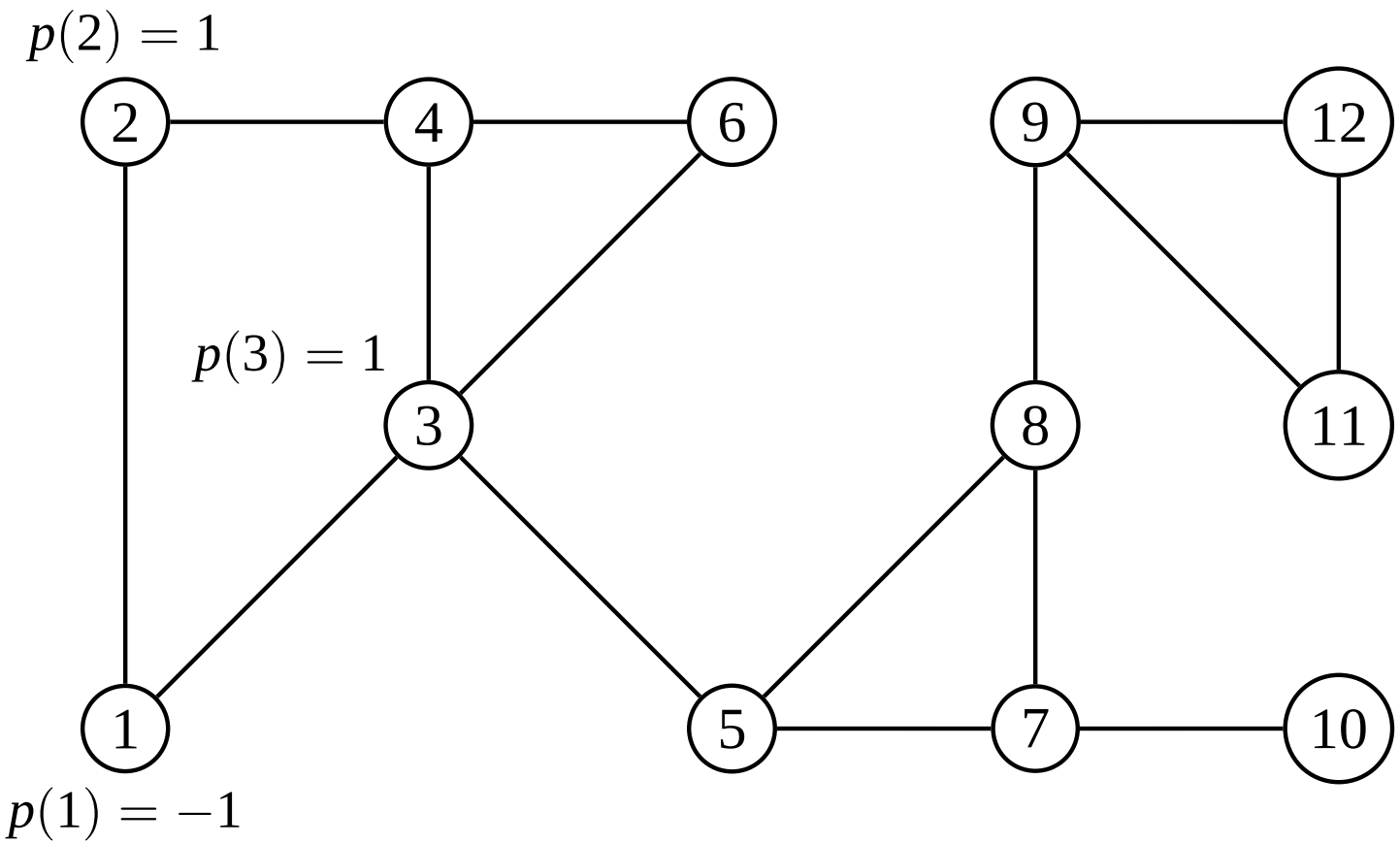


$p(1) = -1$

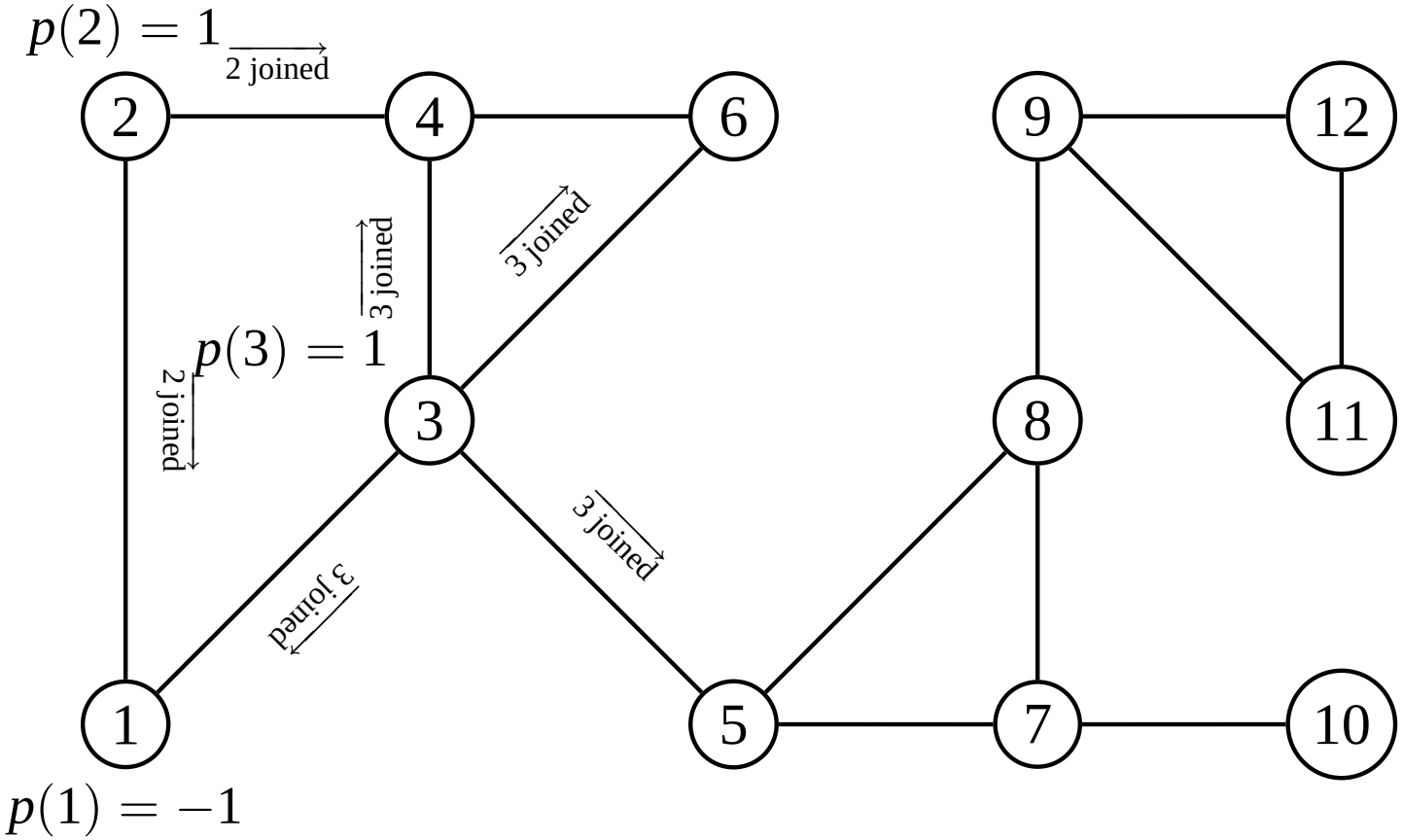
Example Algorithm: BFS



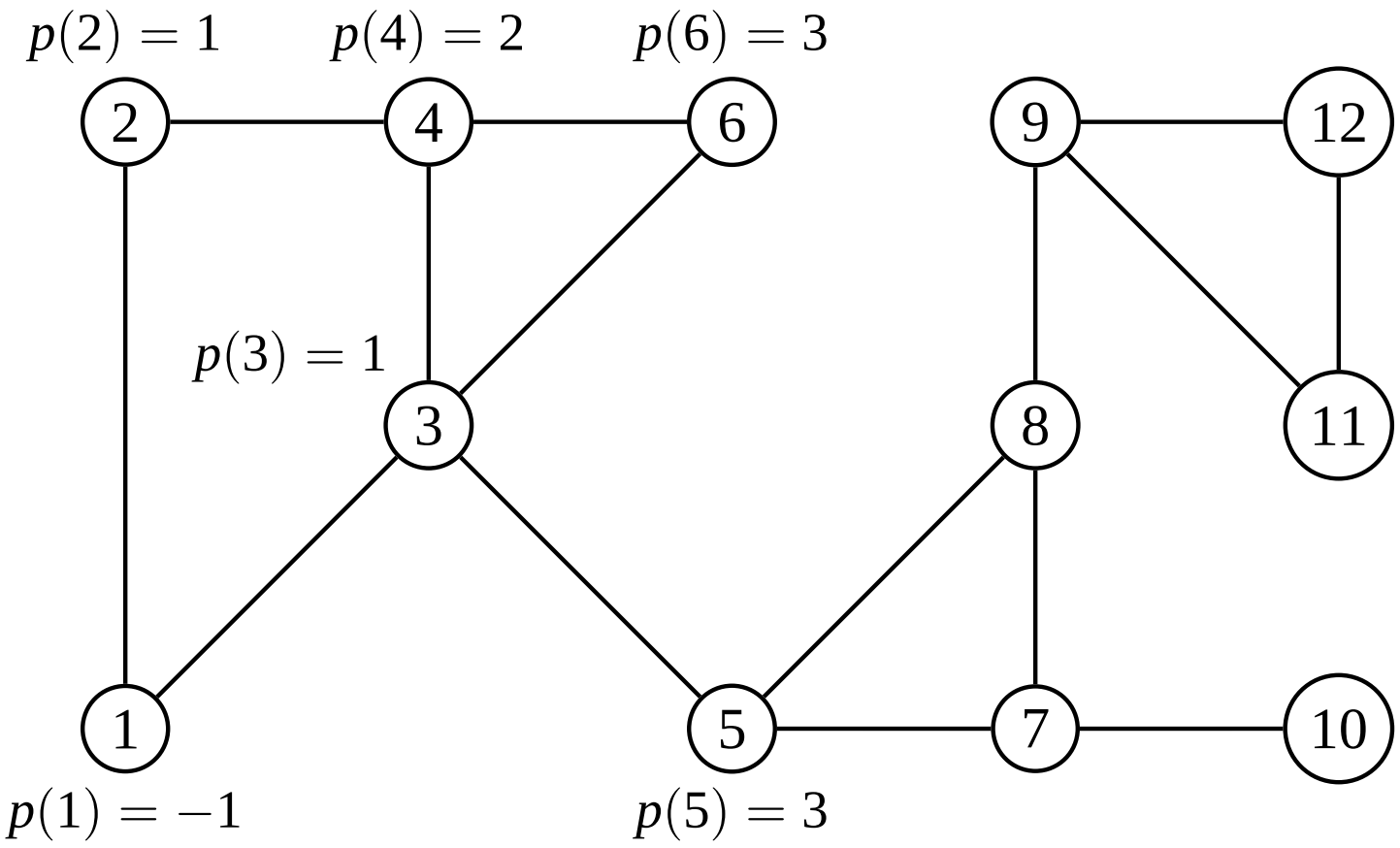
Example Algorithm: BFS



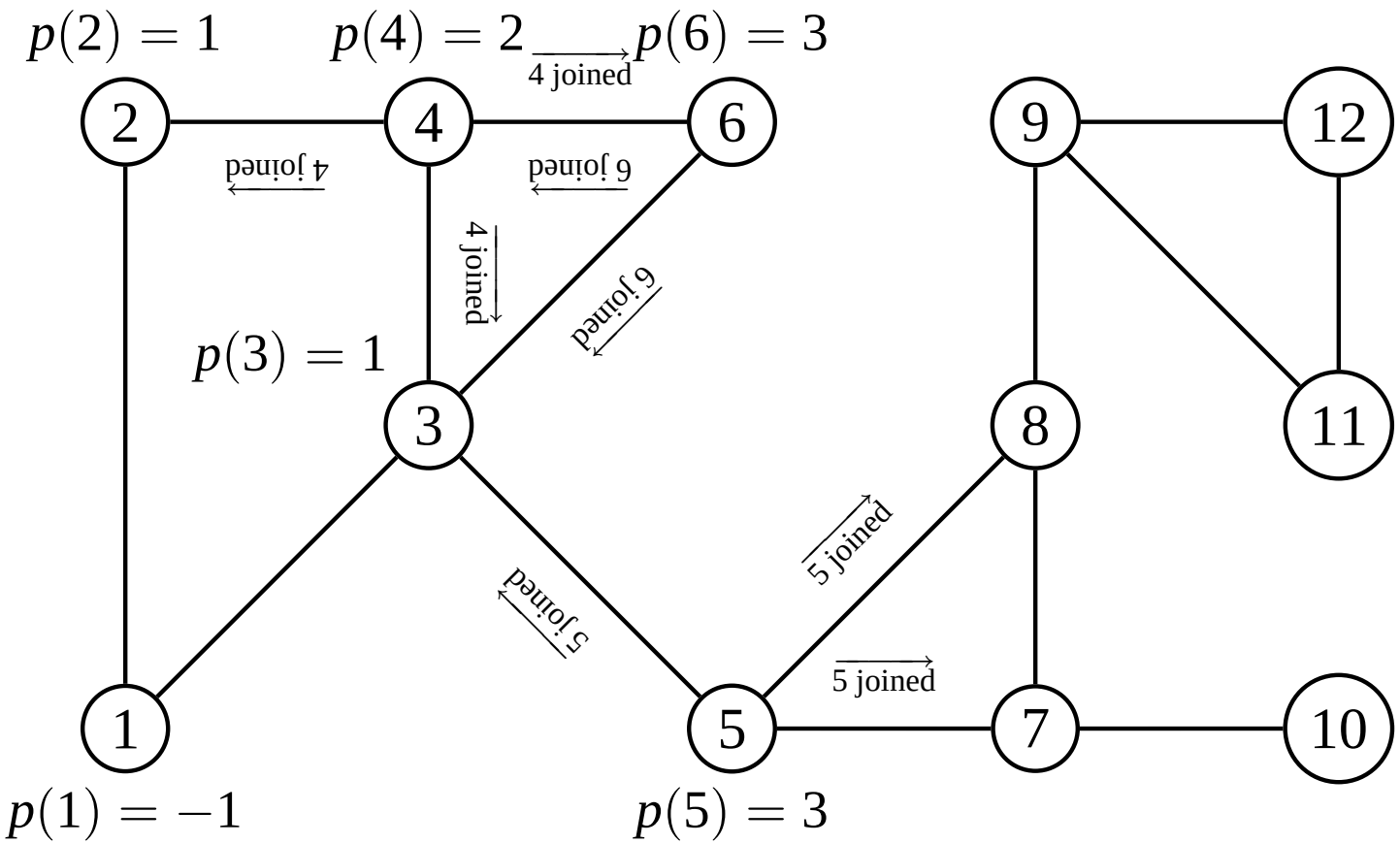
Example Algorithm: BFS



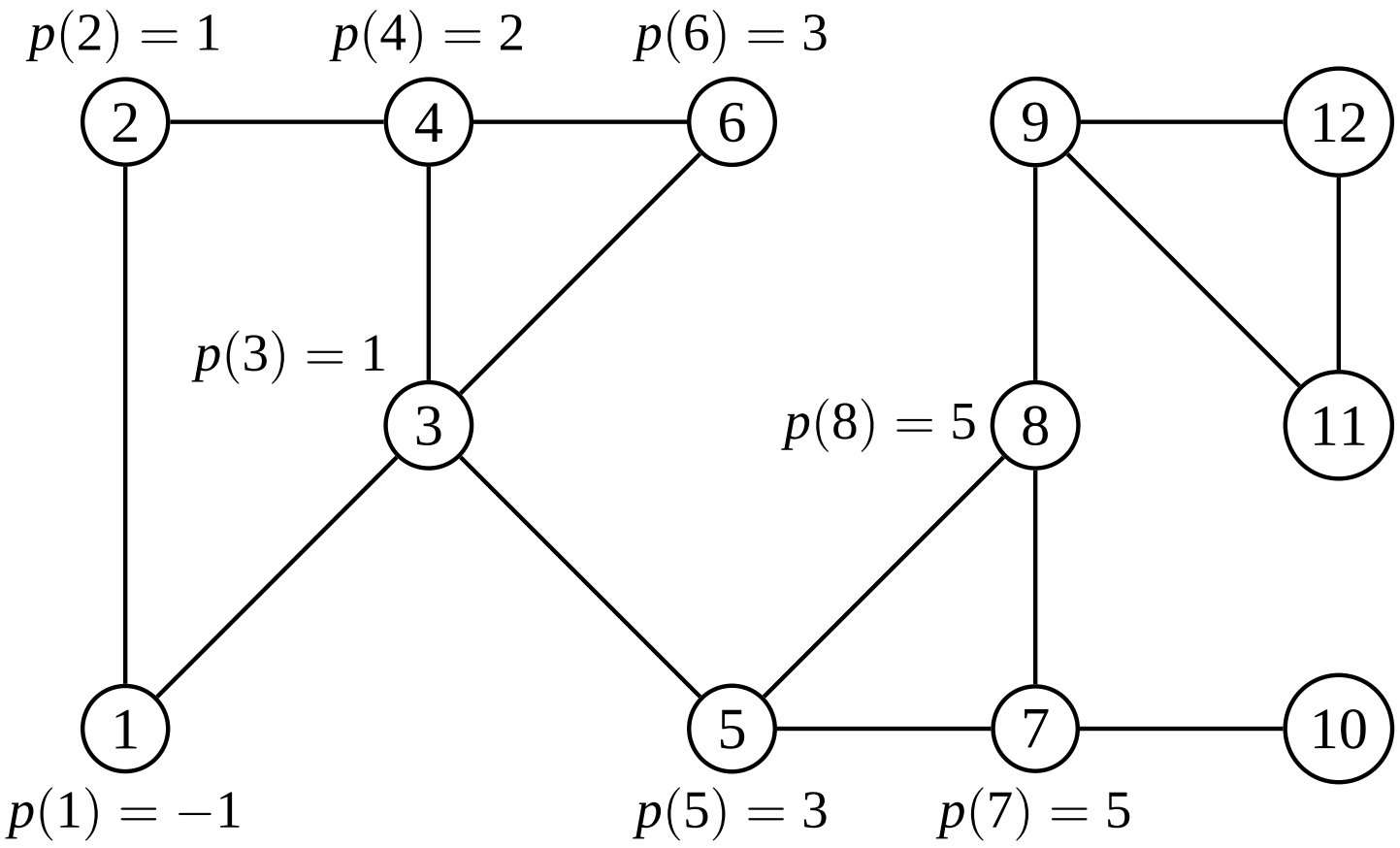
Example Algorithm: BFS



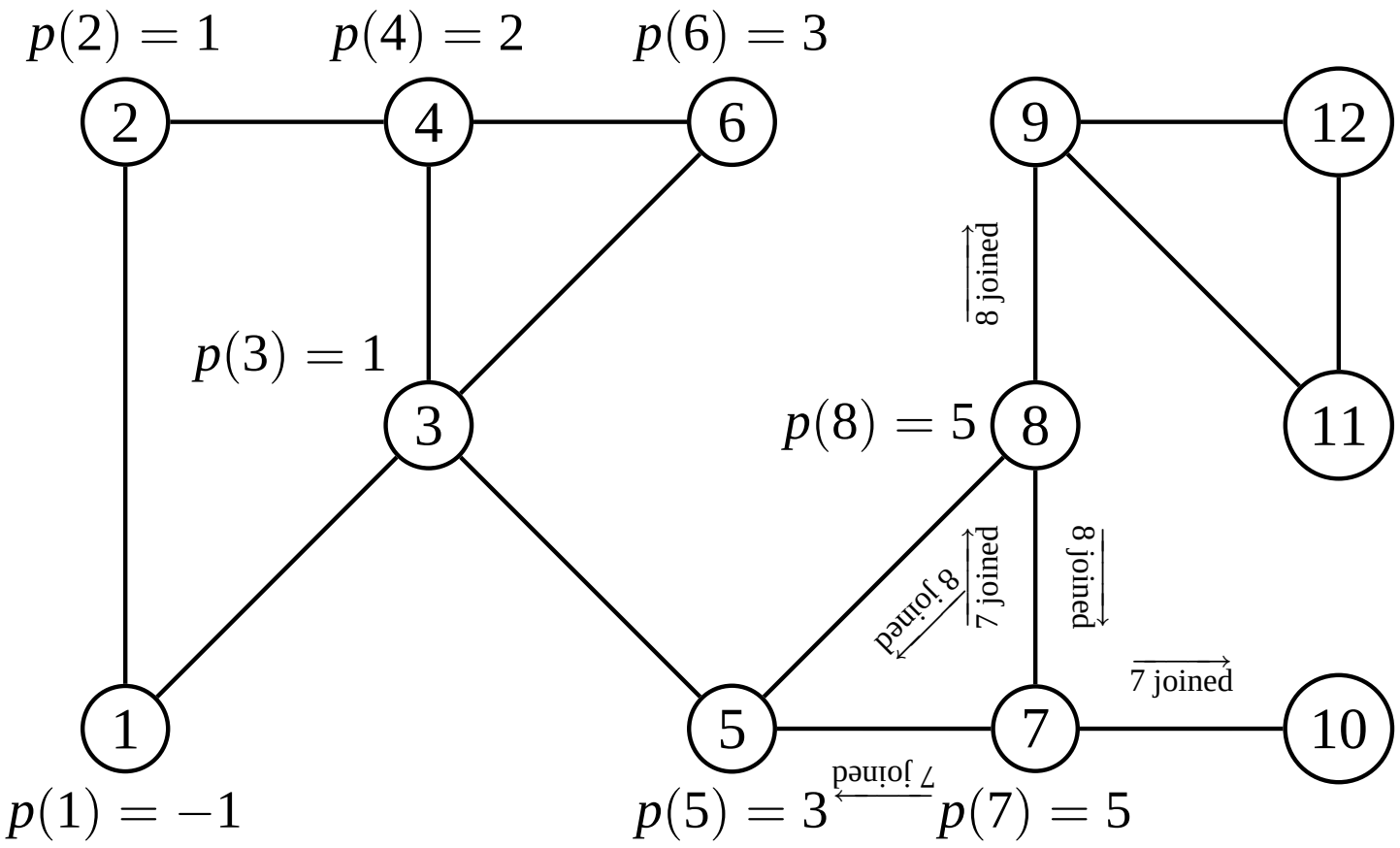
Example Algorithm: BFS



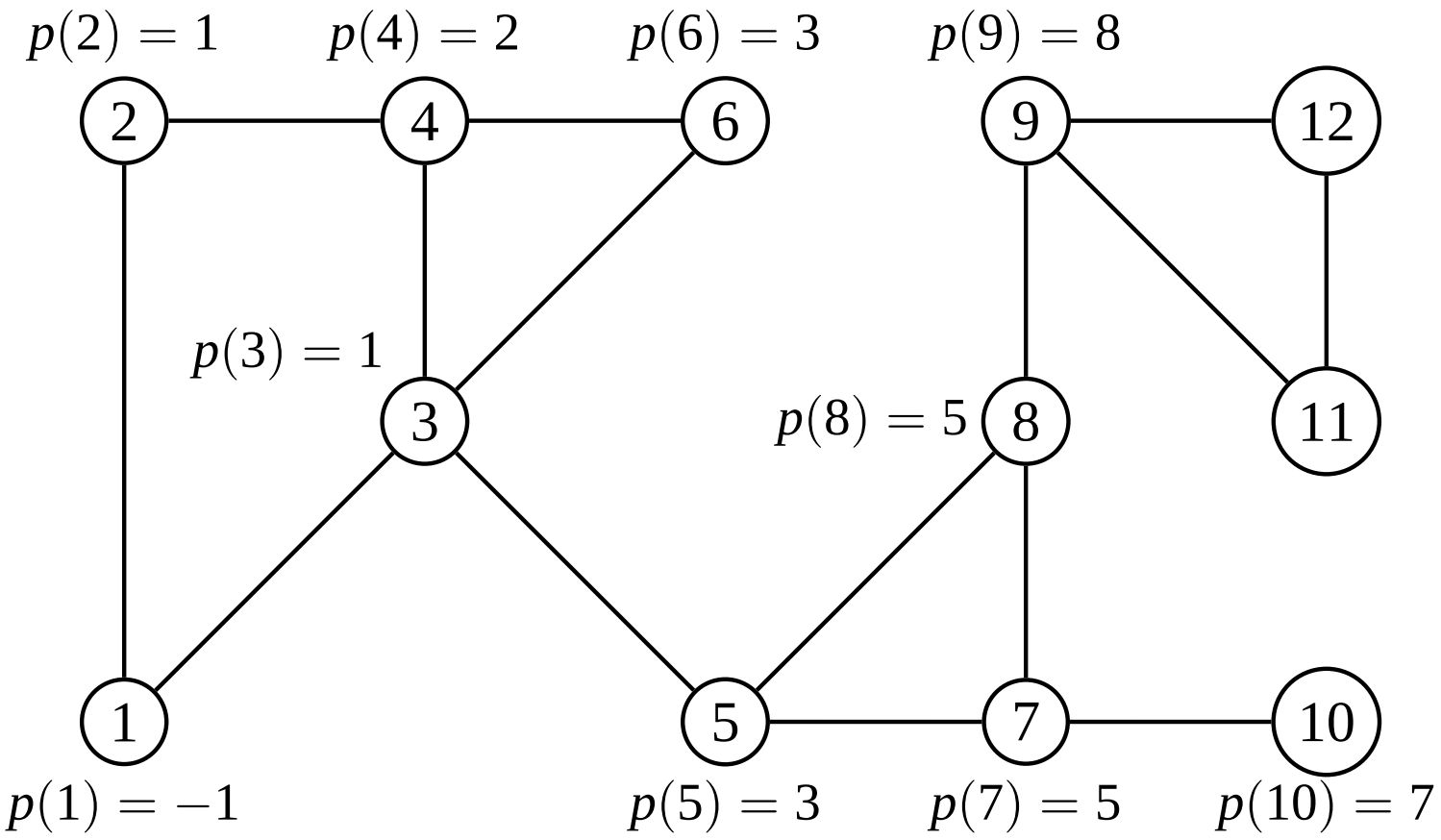
Example Algorithm: BFS



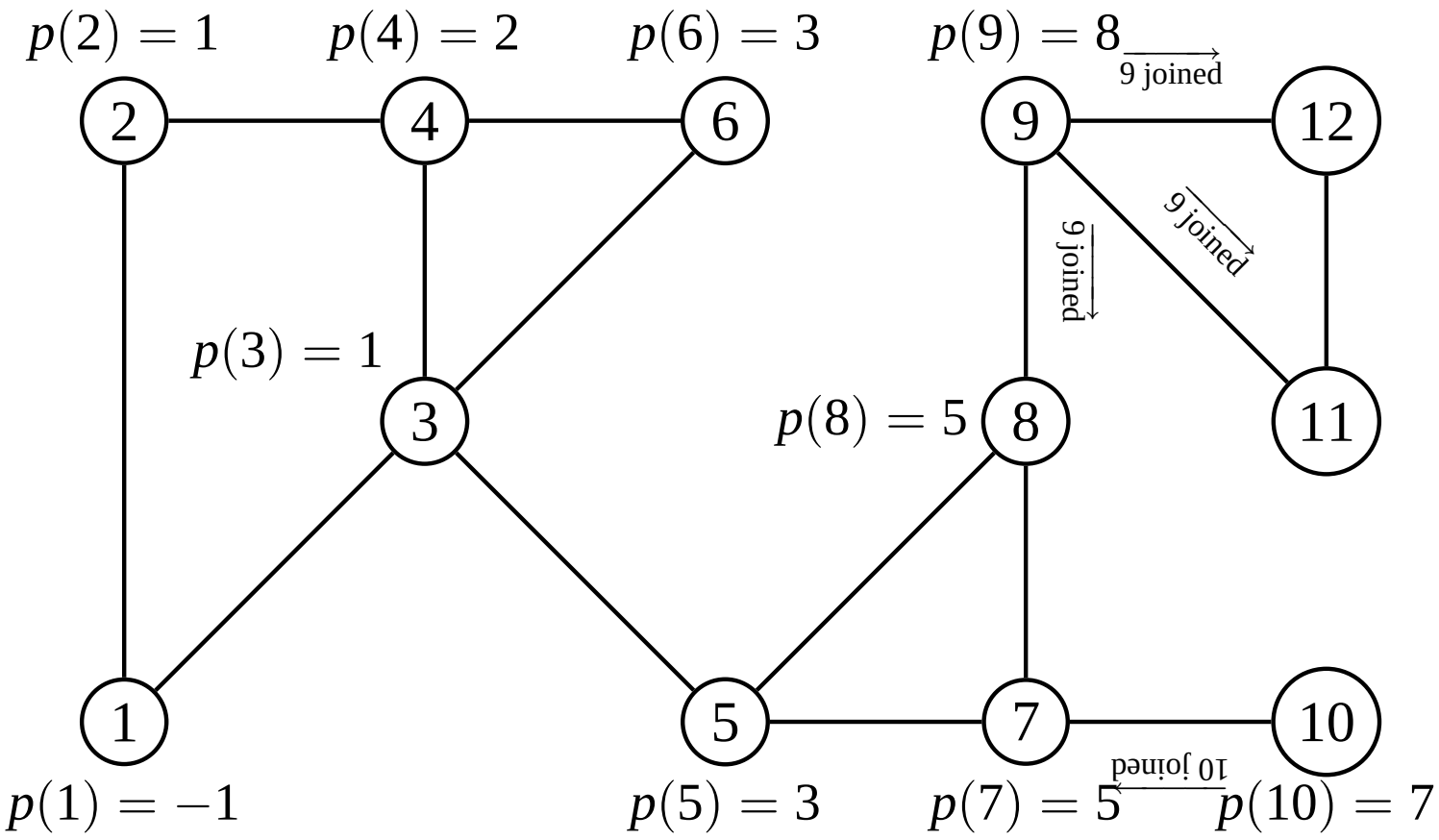
Example Algorithm: BFS



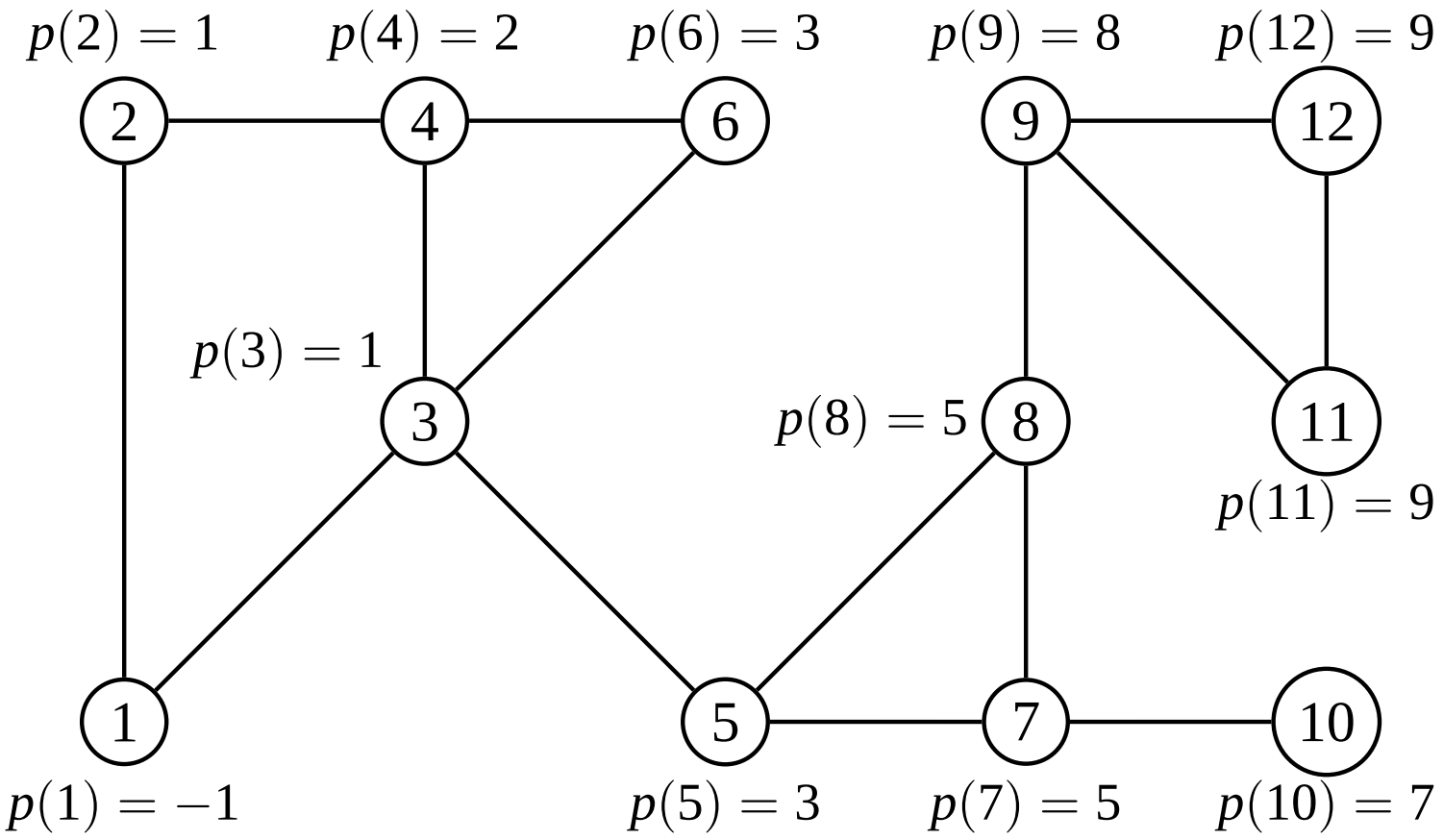
Example Algorithm: BFS



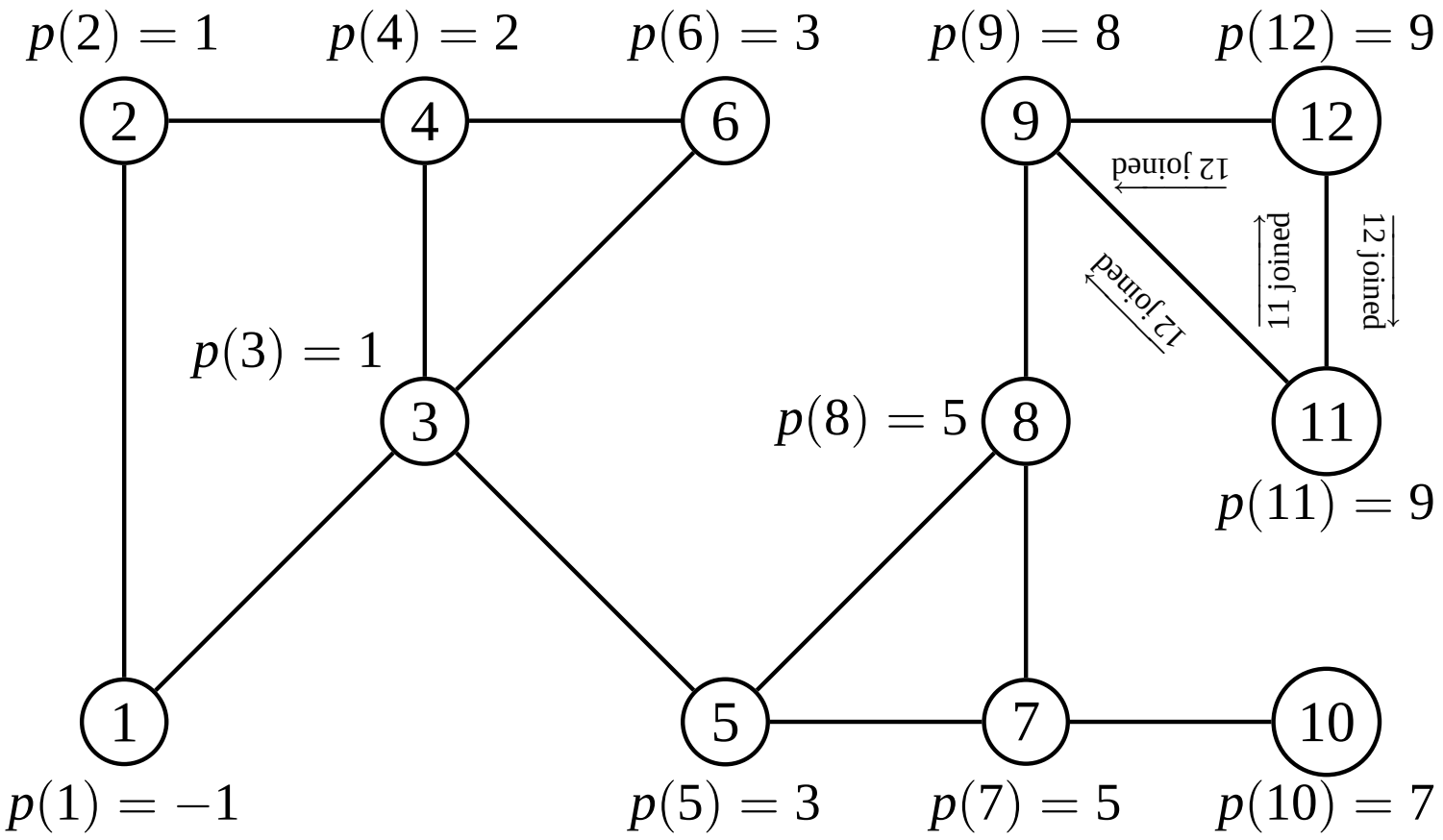
Example Algorithm: BFS



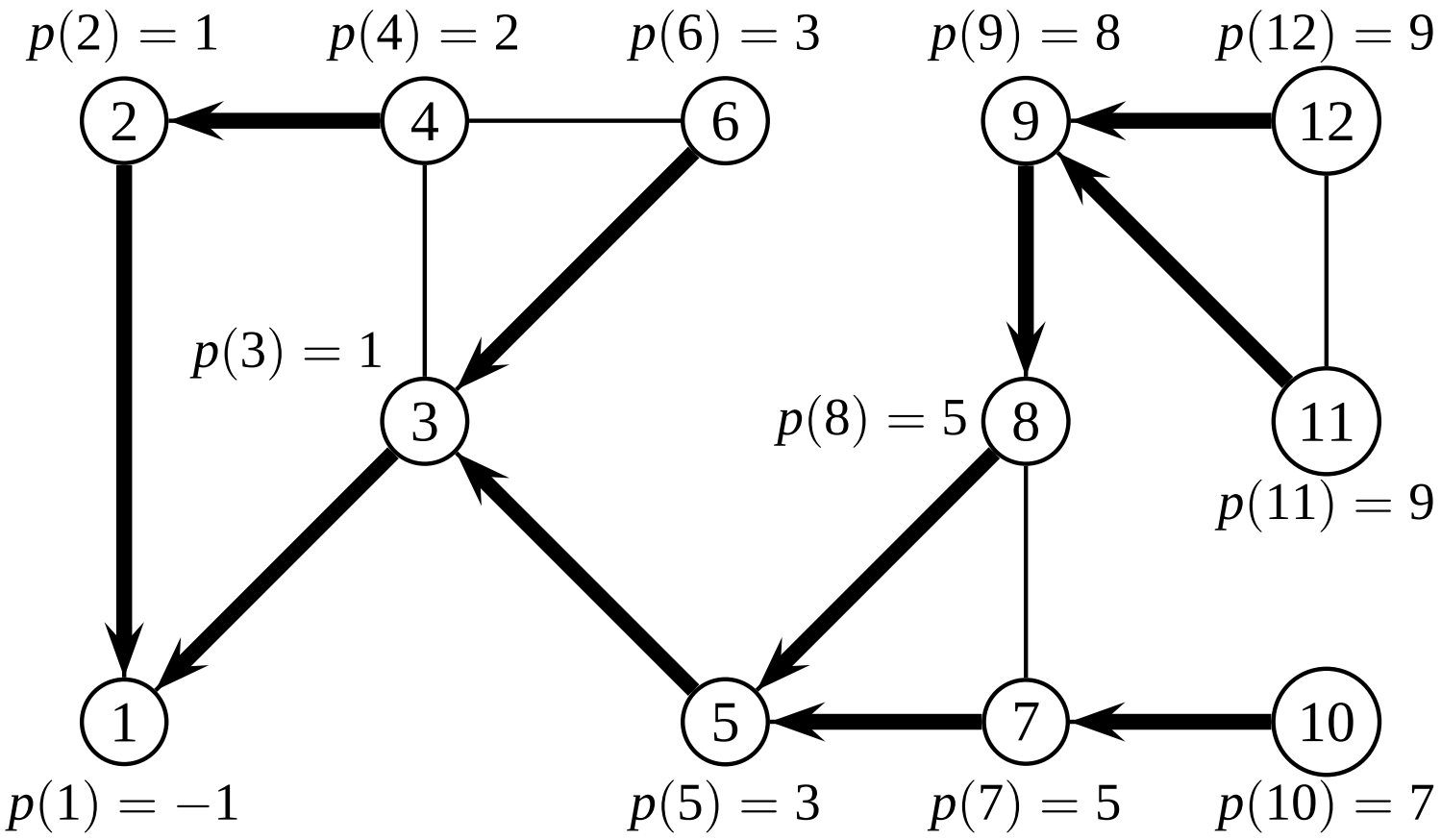
Example Algorithm: BFS



Example Algorithm: BFS



Example Algorithm: BFS



Analysis of BFS

- Each message is a single number between 1 and $|V|$, hence messages are at most $\log_2 |V| + 1$ bits long.
- Require that graph is connected.
- Each node v sends out $deg(v)$ messages.
- Thus message complexity is $2|E|$.
- Let r be the leader node. Node v joins the tree in $d(r, v)$ rounds.
- Thus time complexity is

$$\max_{v \in V} d(r, v) := radius(r, G).$$

- As

$$\mathsf{Diam}(G)/2 \leq radius(r, G) \leq \mathsf{Diam}(G)$$

  it is more customary to write running time as $\Theta(\mathsf{Diam})$, which is independent of the choice of leader.
- Straightforward to make each node aware of its children. To add notification of termination: when a node's subtree is complete, it informs its parent; this doubles the running time.

Justifying the Model

- **Why synchronous?** By applying a *synchronizer* any asynchronous network (i.e., where not all nodes run in lock-step, or messages are subject to unequal delays) can simulate a synchronous one. Mind you, this incurs an increase in complexity.
- **Why short messages?** Short messages make the algorithms practical.
- Furthermore, if we allow arbitrarily large messages, then any graph theoretic problem can be trivially solved in $O(\mathsf{Diam})$ rounds and with $O(m)$ messages as follows: in round i, each node broadcasts the nodes and edges of its radius-$(i-1)$ neighbourhood to all neighbours, and in the next round determines its radius-i neighbourhood based on the messages received.
- After Diam rounds each node knows the entire network topology and can solve any (decidable) graph-theoretic problem locally.

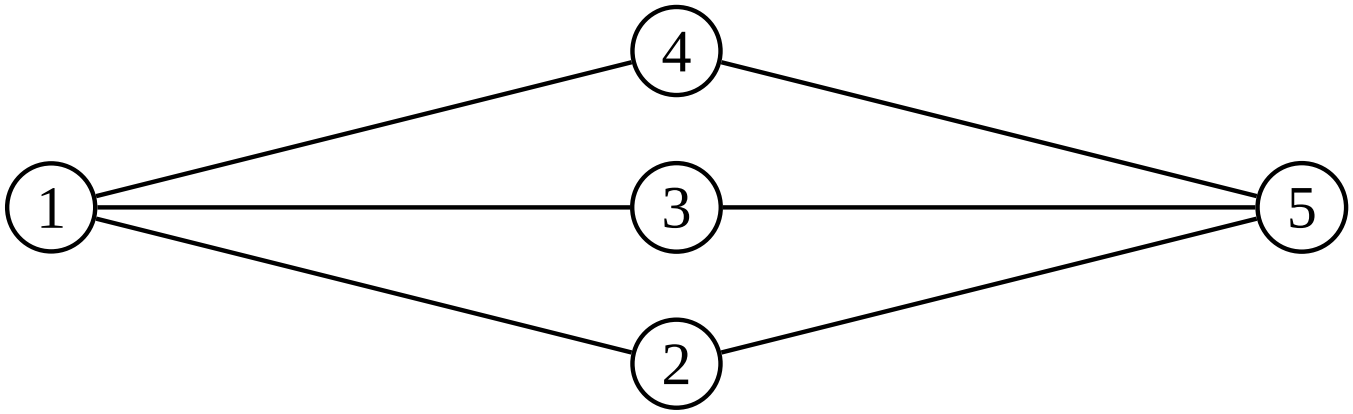
Outline

Back to Connectivity

- Perhaps our graph is not k-connected, but we would still like to know how well it can tolerate failures.
- Specifically, is there a succinct way to determine the pairs of points which would remain connected despite any $(k-1)$ failures?
- The edge-k-connected case is easier to describe. Write $u \approx_k^e v$ if u and v remain connected despite the deletion of any $(k-1)$ edges.
- This is an equivalence relation: if u remains connected to v despite any $k-1$ edge failures and v remains connected to w despite any $k-1$ edge failures then u remains connected to w despite any $k-1$ edge failures.

Back to Connectivity

- In general the equivalence relation $\approx_k^e$ which we just defined is hard to work with.
- For example, in the graph shown below with $k = 3$, the equivalence classes of $\approx_3^e$ are $\{\{1, 5\}, \{2\}, \{3\}, \{4\}\}$.



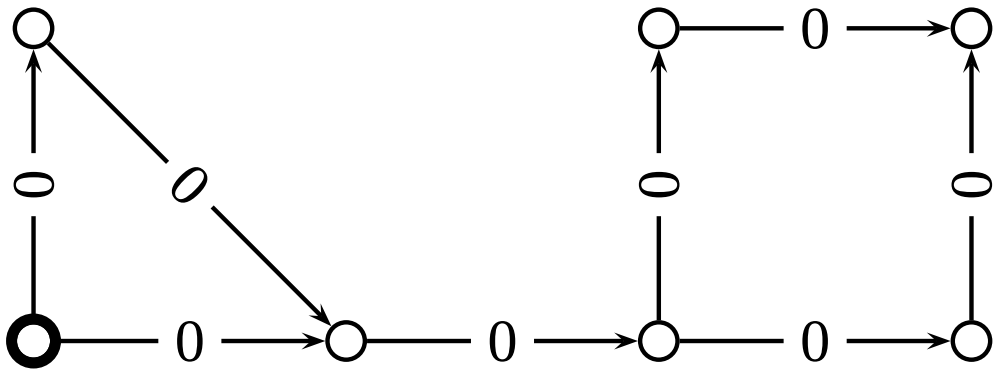
- Classes like $\{1, 5\}$ are hard to deal with since they do not induce connected subgraphs of G.
- So hereafter we focus on the somewhat "special" case $k = 2$, where we can show that the equivalence classes are connected.

Edge-Biconnectivity

- A *bridge* is an edge whose deletion causes the graph to become disconnected.
- In other words, a graph is 2-edge-connected if it has no bridge.
- An edge e is a bridge if and only if it does not lie in some simple cycle of G. Follows from the fact that simple cycles containing (u, v) correspond bijectively to simple u, v paths in $G - (u, v)$.
- Further, from this we can deduce that an edge (u, v) is a bridge if and only if $u \not\approx_2^e v$.
- So the equivalence classes of $\approx_2^e$ are just the connected components of $G - bridges(G)$.
- Call the equivalence classes of $\approx_2^e$ the *biconnected components*.

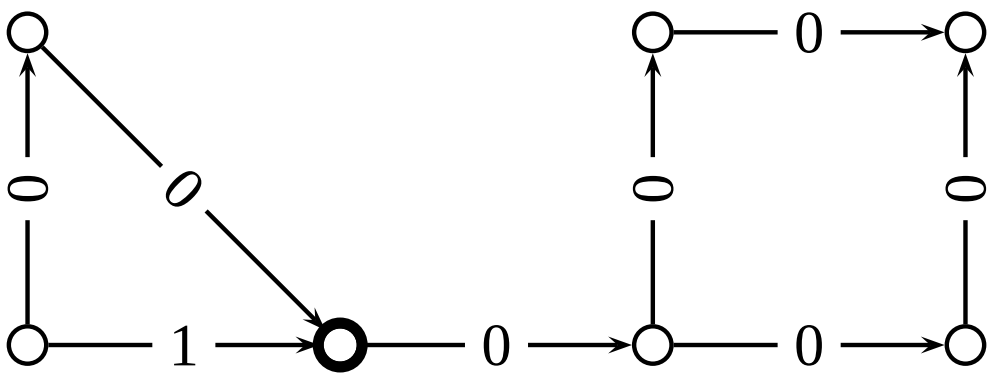
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



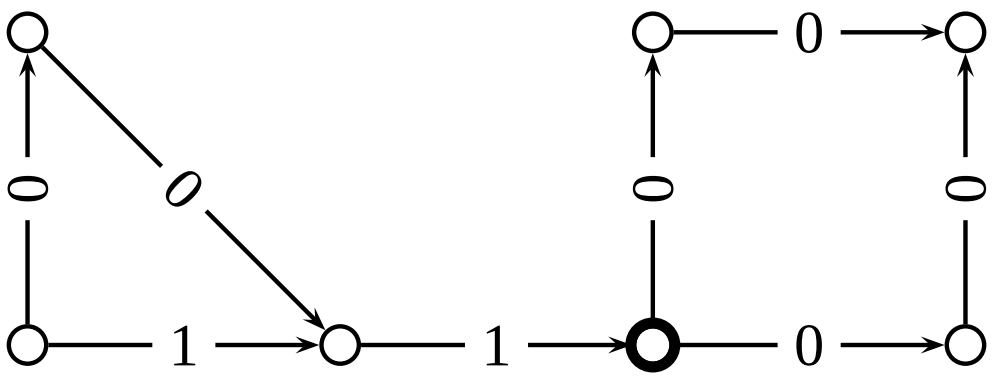
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



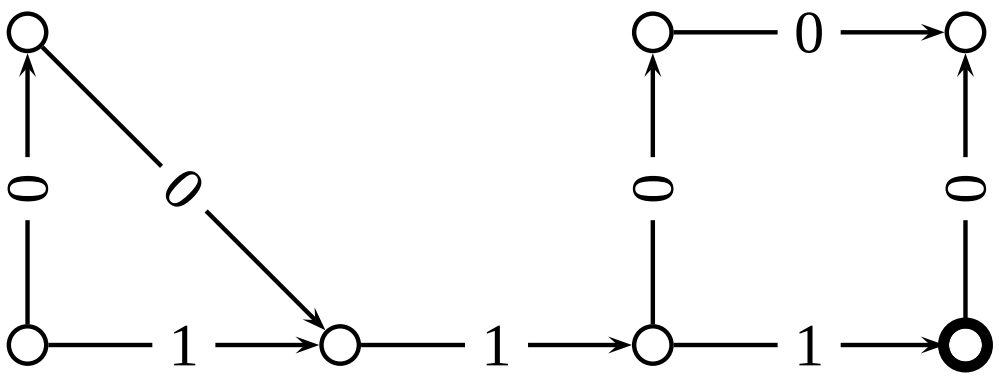
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



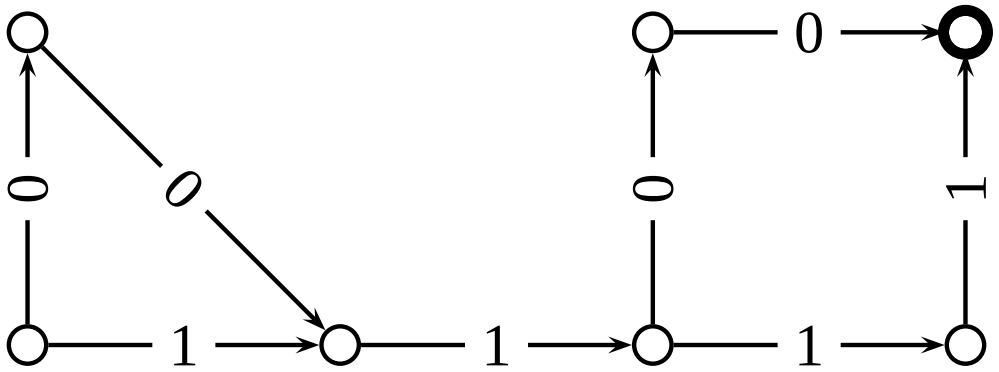
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



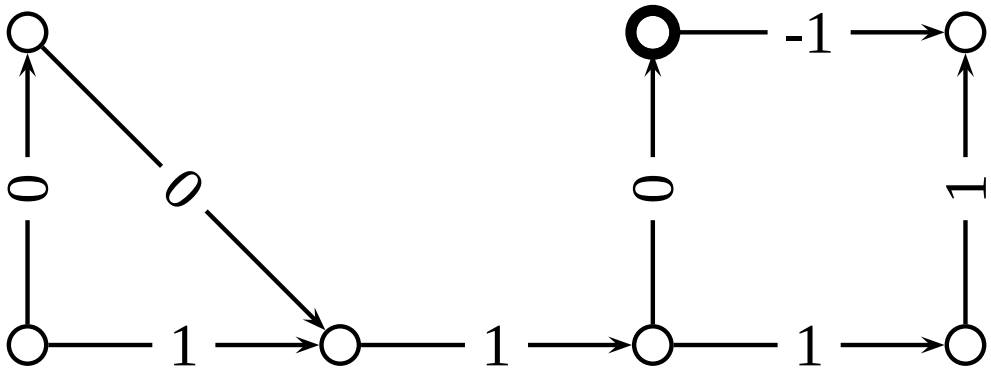
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



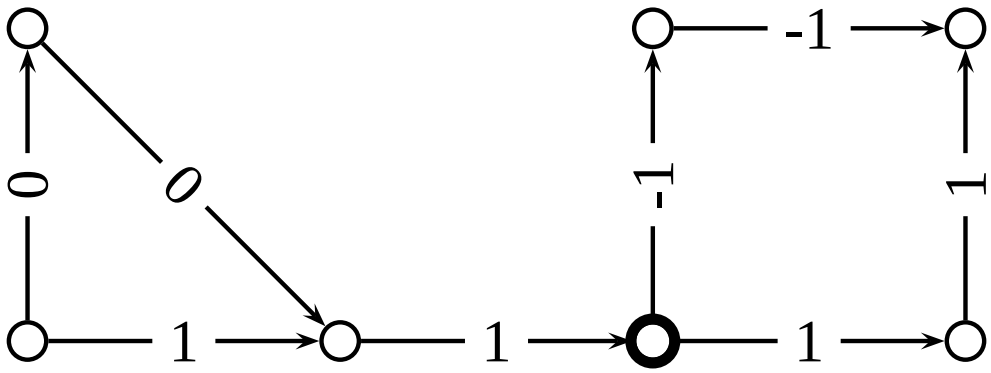
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



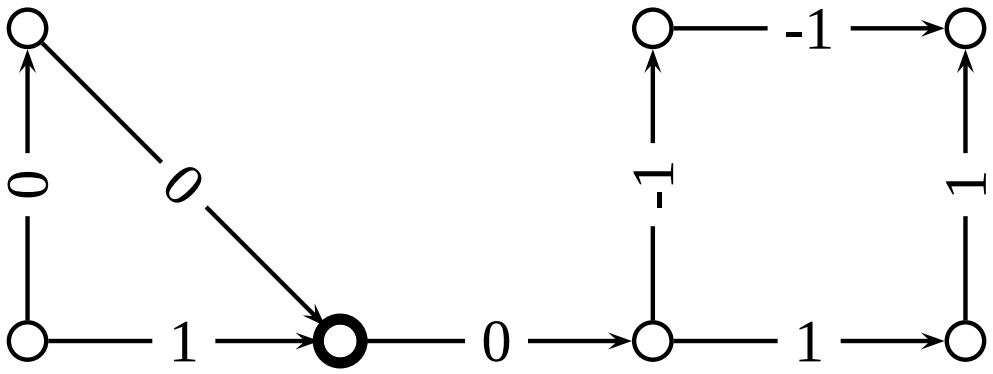
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



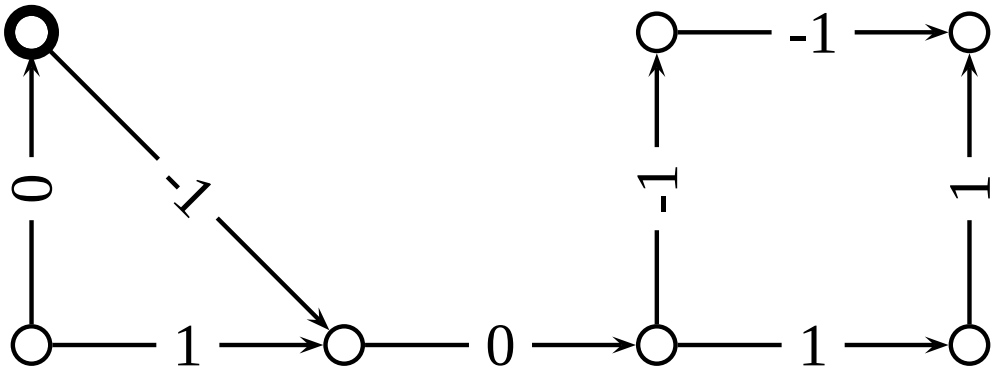
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



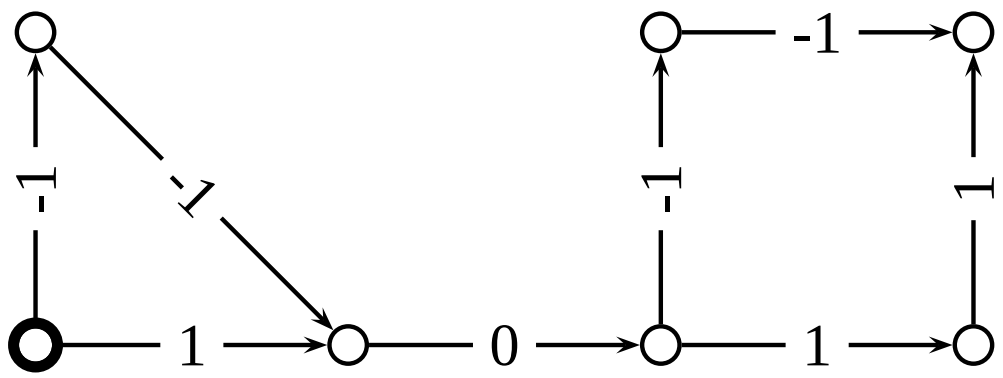
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



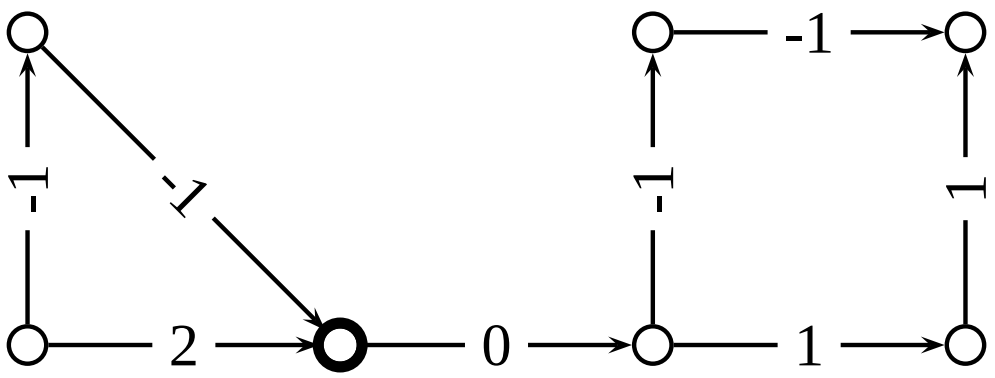
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



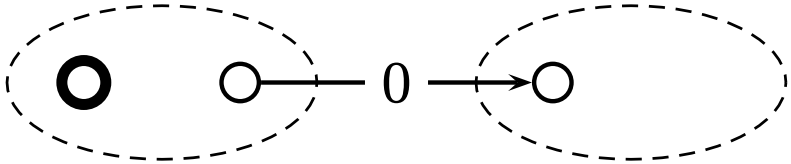
An “Alternative” Biconnectivity Algorithm (S. Vempala)

- We know that once we identify the bridges of the graph, it is straightforward to determine the biconnected components.
- Consider putting a “walker” on the graph. Orient each edge arbitrarily and give it a counter initialized to 0.
- When the walker traverses an edge, change the counter by +1 if the step was in agreement with its orientation, and change the counter by -1 otherwise.



An "Alternative" Biconnectivity Algorithm

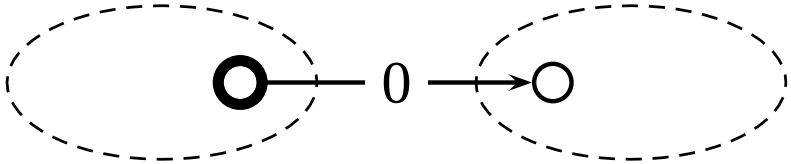
- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker's position and also the counter's value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

An "Alternative" Biconnectivity Algorithm

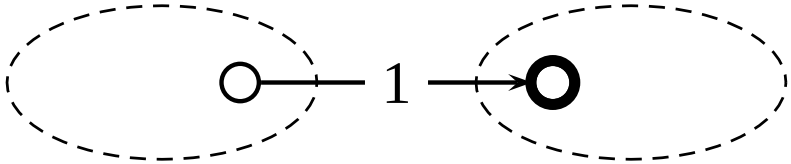
- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker's position and also the counter's value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

An "Alternative" Biconnectivity Algorithm

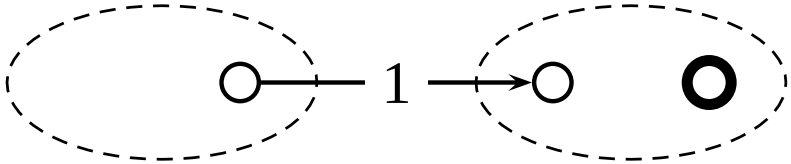
- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker's position and also the counter's value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

An “Alternative” Biconnectivity Algorithm

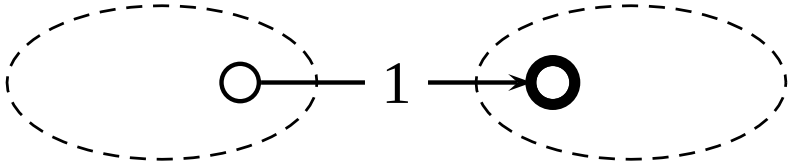
- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker’s position and also the counter’s value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

An "Alternative" Biconnectivity Algorithm

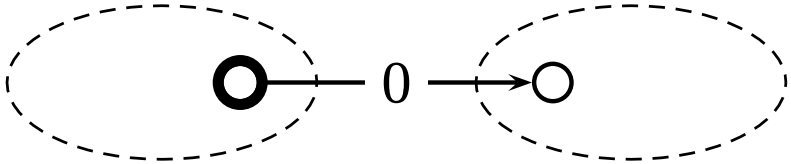
- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker's position and also the counter's value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

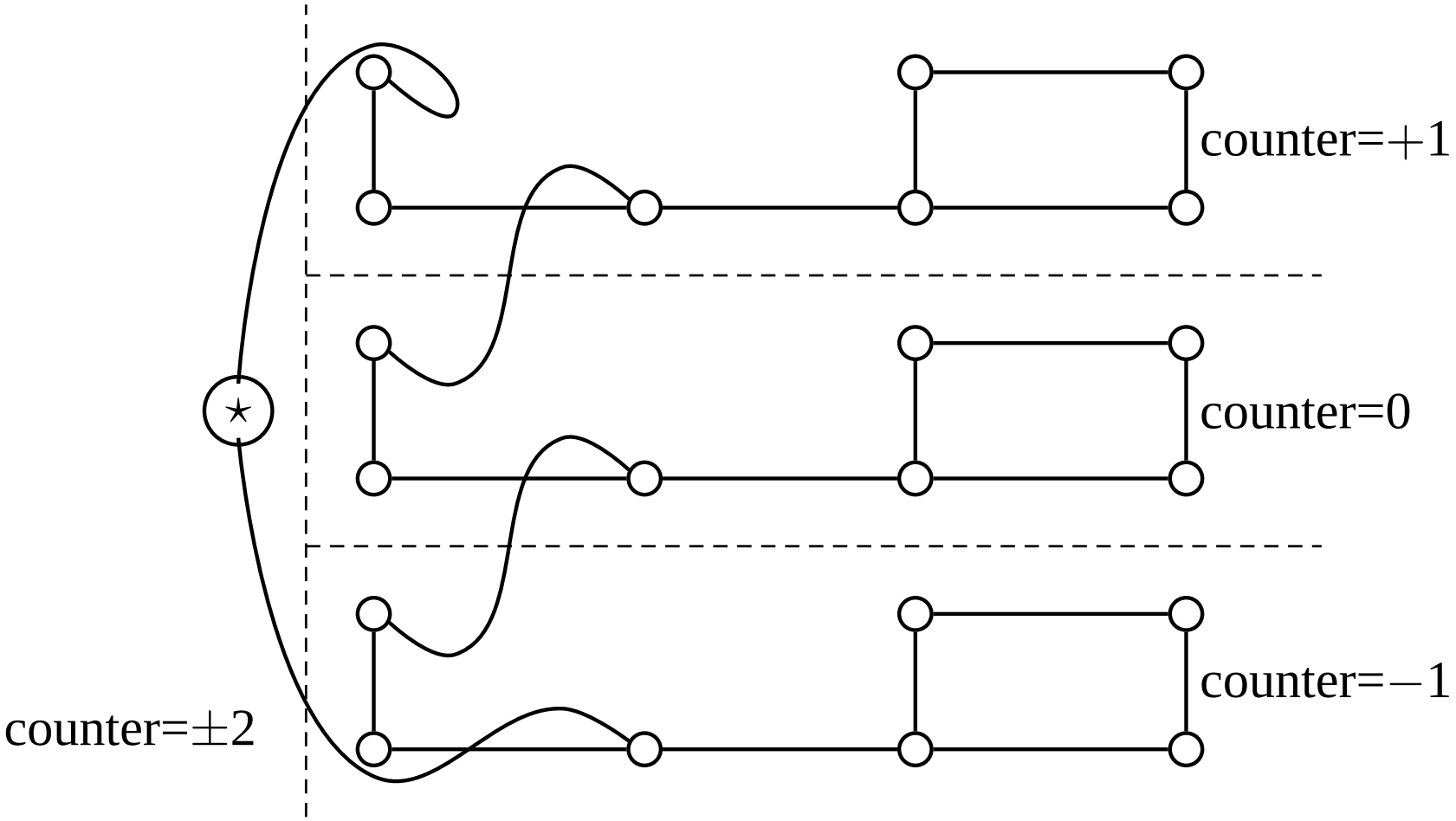
An "Alternative" Biconnectivity Algorithm

- First, no matter how the walker moves, the counter of a bridge will always remain in $\{-1, 0, 1\}$.



- Second, if the walker performs a *random walk*, it can be shown a given non-bridge edge is expected to exceed ± 1 in $O(|V||E|)$ steps.
- Proof idea: make a new graph that represents the walker's position and also the counter's value. A special node $\star$ indicates that the value on the counter exceeds 1 in absolute value.
- A random walk on the old graph corresponds to a random walk on the new graph. So, by classical results about random walks, we expect to hit $\star$ in $O(|V'||E'|)$ steps, where the modified graph is (V', E'). Observe $|V'| = O(|V|)$ and $|E'| = O(|E|)$.

An “Alternative” Biconnectivity Algorithm



An “Alternative” Biconnectivity Algorithm

- The walk-based algorithm is nice for a couple of reasons.
- First, the algorithm never misclassifies a bridge as a non-bridge.
- Further, assuming the walker never dies due to being caught in a failure, all non-bridges in the connected component ultimately containing the walker are correctly identified, in polynomial time, with high probability.
- But in a reliable dense network the running time of $\Omega(n^3)$ is much too high.

Outline

Distributing Tarjan's Node-Biconnectivity Algorithm

Tarjan 1972 "DFS and Linear Graph Algorithms"

- The first distributed biconnectivity algorithms were based off of an older *sequential* biconnectivity algorithm of Tarjan.
- An *articulation point* is a vertex whose deletion causes the graph to become disconnected.
- The *blocks* of a graph are the classes of an equivalence relation on the edges, such that two edges are equivalent iff they are contained in a simple cycle together (No proof here but easy).
- Bridges are singleton blocks and articulation points are those vertices which are incident on more than one block.
- Thus edge-biconnectivity is easy, given node-biconnectivity.
- Tarjan's 1972 algorithm, using DFS, computes the blocks, bridges, etc.
- Uses a preordering of the DFS spanning tree. Leads to distributed time complexity of $O(n)$ — bottleneck is DFS.

Improving Performance, I

- In a series of papers the basic algorithm was reformulated and streamlined.
- In what follows $n = |V|$ and $m = |E|$.
- In [Tarjan and Vishkin 1984; Huang 1989] it was realized that any spanning tree could be used, not just DFS. The algorithm, given a graph G, determines an auxiliary graph H such that each node of H is an edge of G, and the connected components of H are the blocks of G.
- The distributed complexity as described in these papers is $O(\mathsf{Diam})$ time and $O(mn)$ messages, but uses messages of size $\Omega(n)$.
- The algorithm I will give later may be viewed as a refinement and streamlining of these ideas.

Improving Performance, II

- In [Thurimella 1995; Thurimella 1997] the algorithm is further optimized to use a subgraph K of G that can be computed distributively.
- Can't use an arbitrary tree, rather a generalization of BFS called *scan-first search*.
- The algorithm uses MST and other subroutines that dominate the time complexity. Plugging in the best known subroutines the total complexity is $O^{\sim}(\sqrt{n} + \mathsf{Diam})$, where $\sim$ indicates that factors of $\log n$ are ignored.
- There are polynomially many messages, each of size $O(\log n)$.
- MST has a known lower bound of $\Omega^{\sim}(\sqrt{n} + \mathsf{Diam})$ time, so this is about as good as this technique can achieve.

Use of Minimum Weight Spanning Tree

- Why is MST a useful technique?
- The algorithm determines a subgraph K of G and wants to determine the connected components of K quickly.
- The connected components of K might have much larger diameter than G.
- Set the weight of each edge of K to 0 and each edge of $G - K$ to 1.
- Then a minimum-weight spanning tree of G will contain a spanning tree for each connected component of K.
- MST is also used in the fastest-known distributed leader election algorithm.

Improving Performance, III

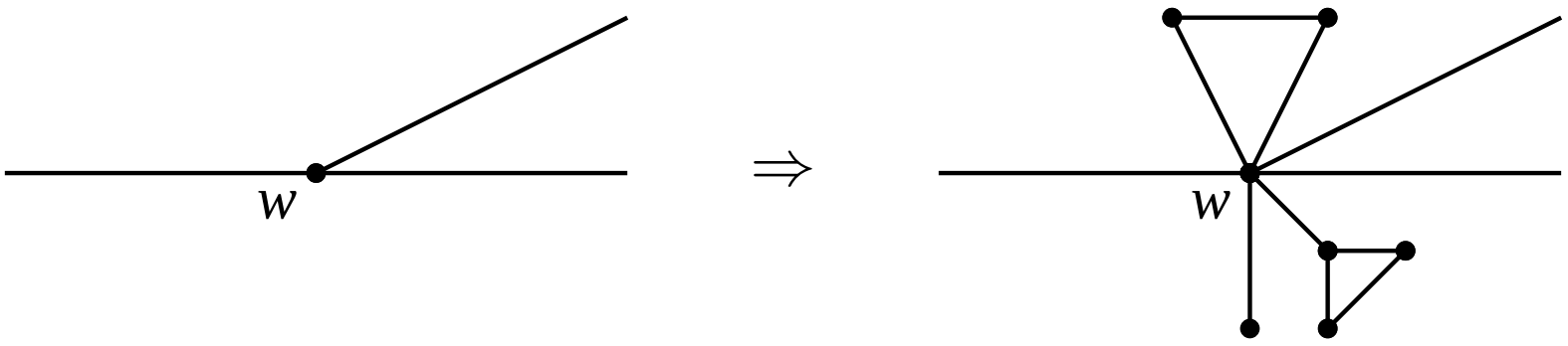
- The algorithm I am presenting improves on the previous ones in time and message complexity, and is fairly simple.
- The algorithm has $O(\mathsf{Diam})$ time complexity and $O(m)$ communication complexity, using messages of size $O(\log n)$.
- Computes the bridges and biconnected components.
- It doesn't seem possible to extend the method to computing articulation points and blocks.
- Uses preordering like the older algorithms.

A Universal Lower Bound

- Let us restrict our attention to *event-driven* algorithms: a node can only send a message in a given round if it received a message in the previous round, plus the leader can send messages in the first round.
- For every graph, there is a $\mathsf{Diam}/2$ lower bound on the time complexity of a correct edge-biconnectivity algorithm, assuming the algorithm is event-driven.
- Why? Only the leader can send messages in the 1st round, and in the ith round, only nodes within distance $(i-1)$ of the leader can send messages.

A Universal Lower Bound

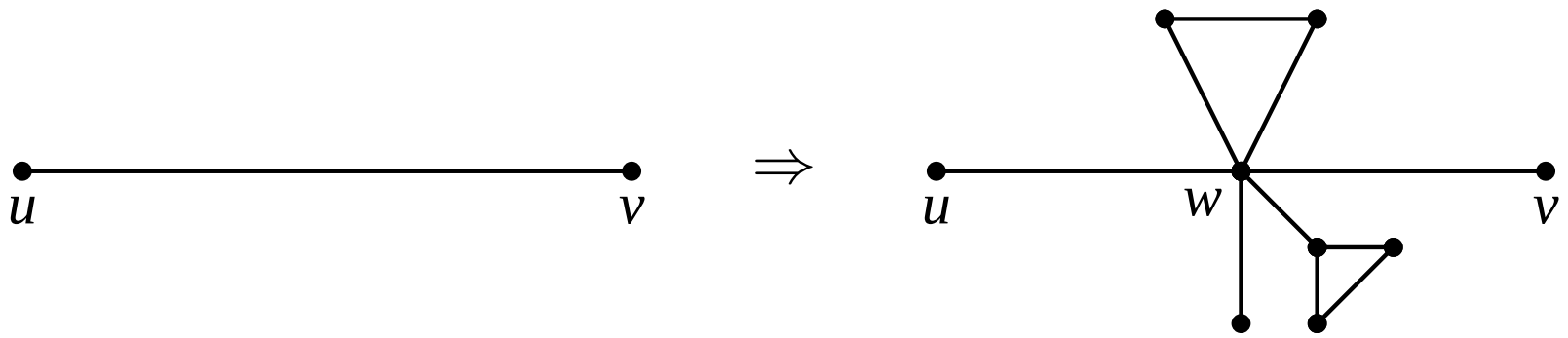
- So if the algorithm terminates in less than $\mathsf{Diam}/2 < radius(r)$ rounds, some node w never receives or sends a message.
- Modify G into G' by attaching some bridges and cycles to w.



- When running on G', the algorithm will never send any messages to the new nodes and edges, so it cannot be correct.

A Universal Lower Bound

- Thus, the older algorithms are time-optimal for *some* graphs, i.e., those with diameter larger than $\sqrt{n}$.
- Under the assumptions made above the new algorithm is time-optimal for *all* graphs.
- The proof that m messages are necessary is similar: each edge (u, v) must communicate some message, or else we can modify G into G' such that the algorithm does not reach all parts of G'.



Outline

Overview

- After some squinting this algorithm can be seen as a distributed version of a 1984 paper by Tarjan.
- As mentioned before, the main task is bridge-finding, and then the biconnected components are just the connected components of $G - Bridges(G)$.
- In order to find bridges it suffices to mark every edge that is lies in a simple cycle, then the bridges will just be the unmarked edges.
- The algorithm begins with a spanning tree of G. It works fastest when it is a BFS tree but any tree will do.

Overview

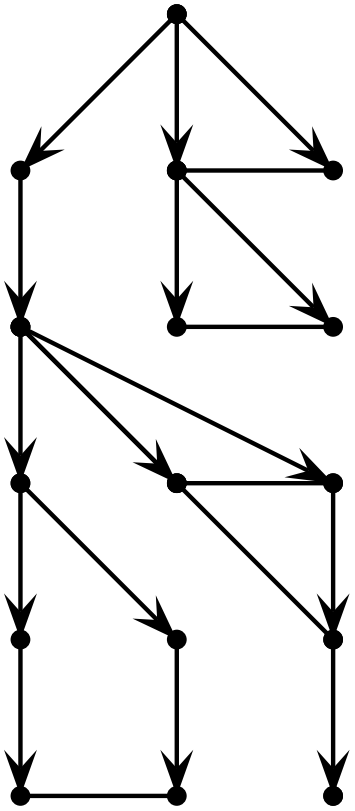
- Input: A rooted tree $\mathcal{T}$ is given.

1. Each node computes its number of descendants.
2. We *preorder* the nodes.
3. Mark each edge in every cycle of a cycle basis by upcasting.
4. Label each node according to its biconnected component.

- Output: Each node gets a label such that two nodes share the same label if and only if they remain connected despite any single edge deletion.

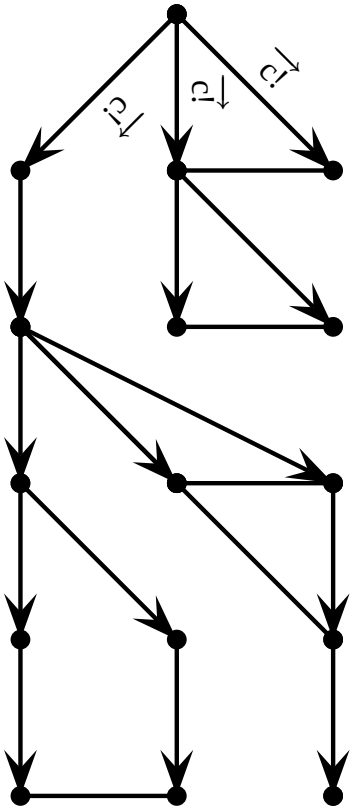
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



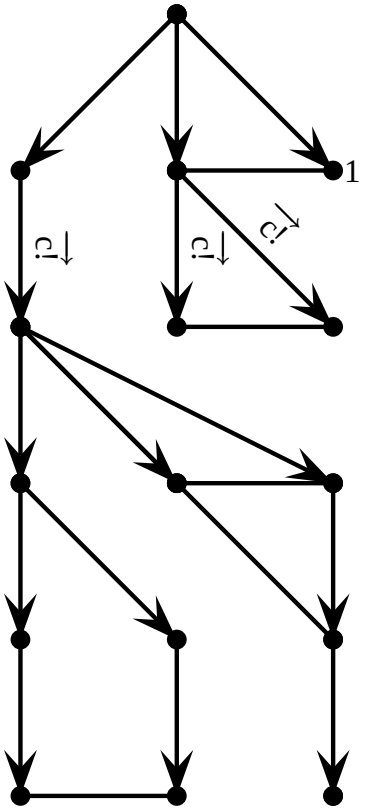
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



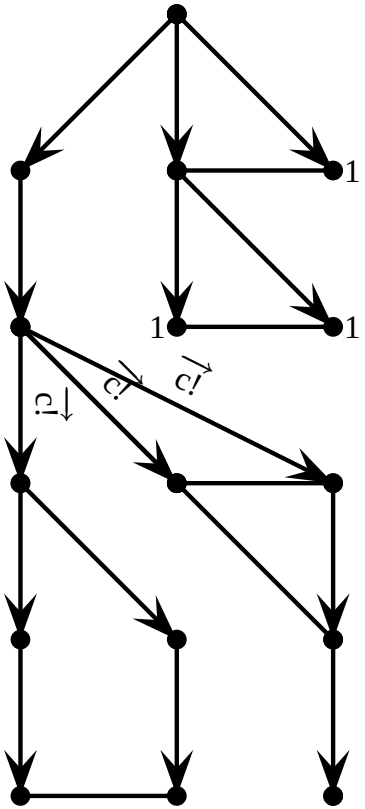
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



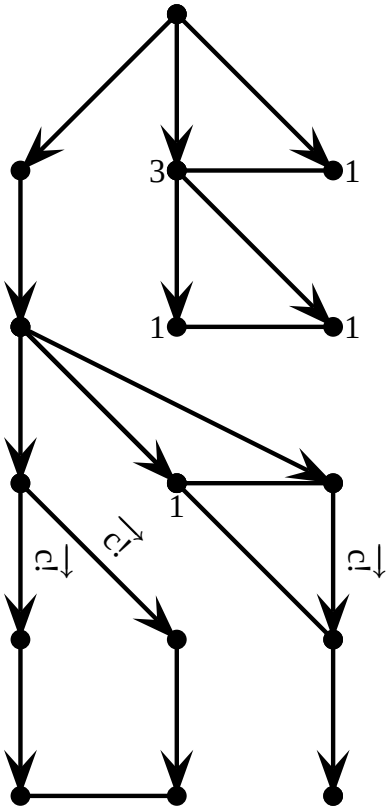
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



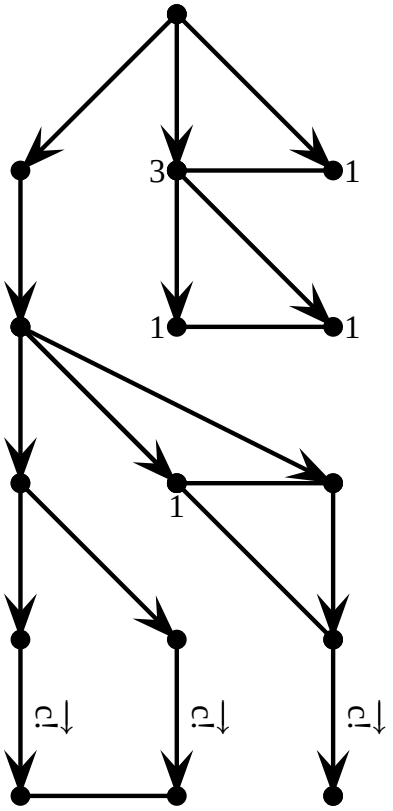
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



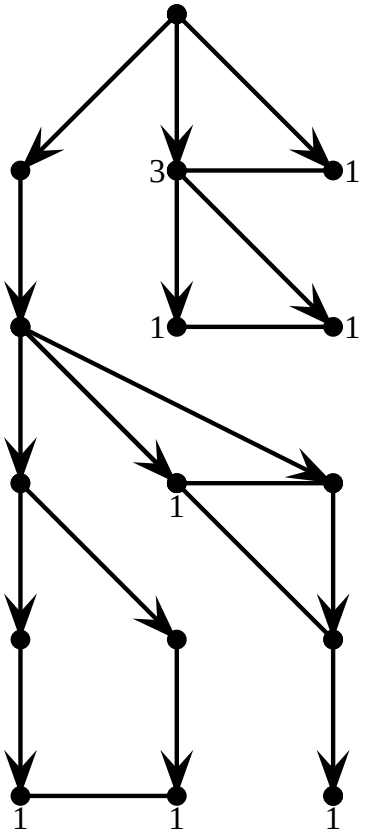
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



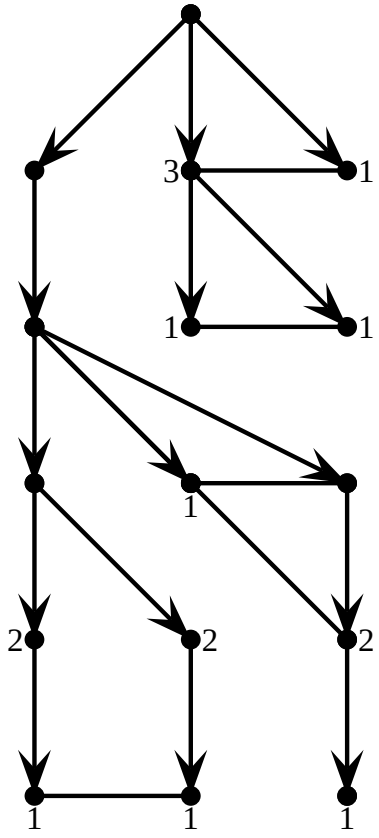
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



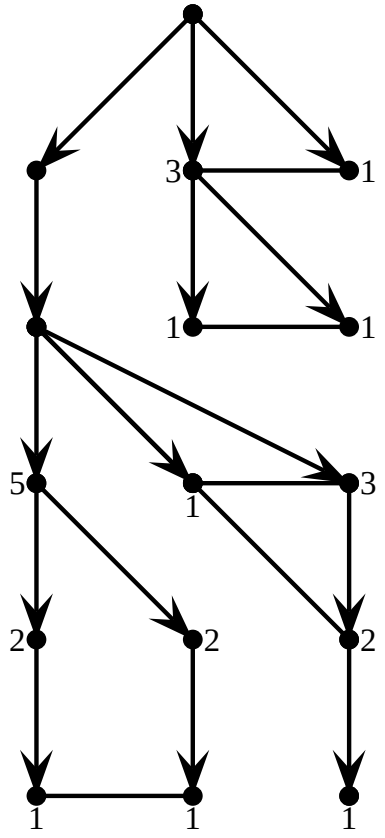
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



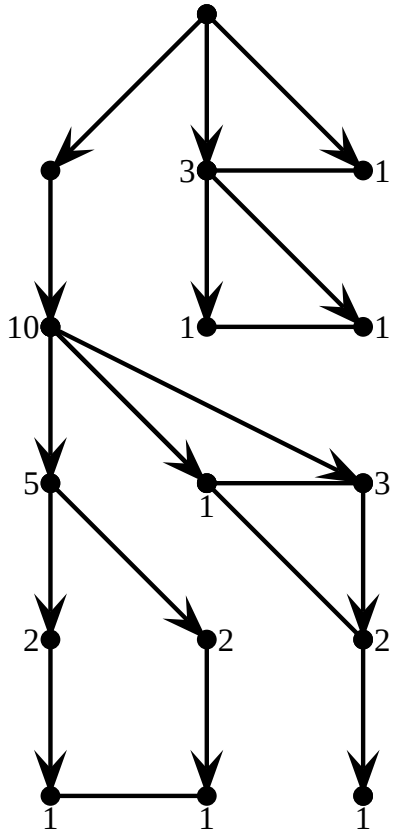
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



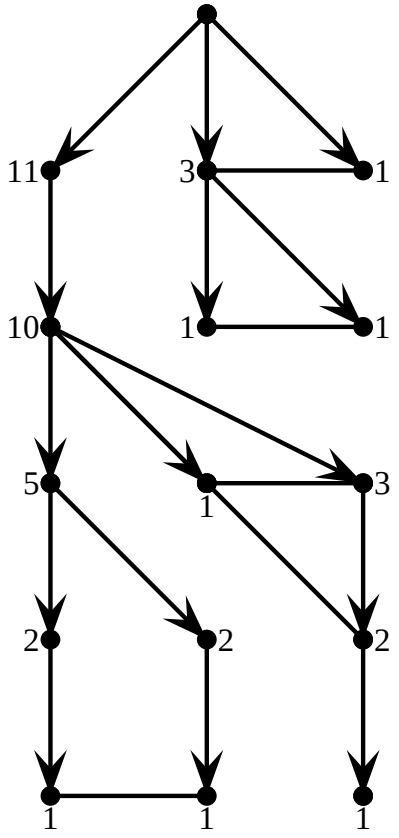
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



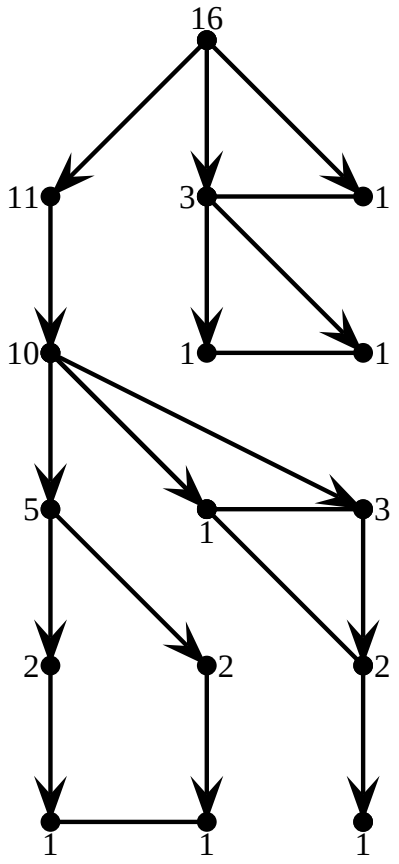
1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.



1. Each node computes its number of descendants

- Fix any rooted spanning tree with root r.
- Convention: every node is a descendant/ancestor of itself.
- First, each node v needs to compute the size $\#desc(v)$ of its subtree (i.e., the number of descendants it has).
- Straightforward if we use a *down/convergecast*, as follows.
- The root sends "compute $\#desc$ of yourself" to each child and all nodes pass this message down the tree.
- Each leaf immediately determines that their size is 1 and reports this to their parent; each other node waits to hear from all its children, sums their values, adds 1, and reports to its parent.

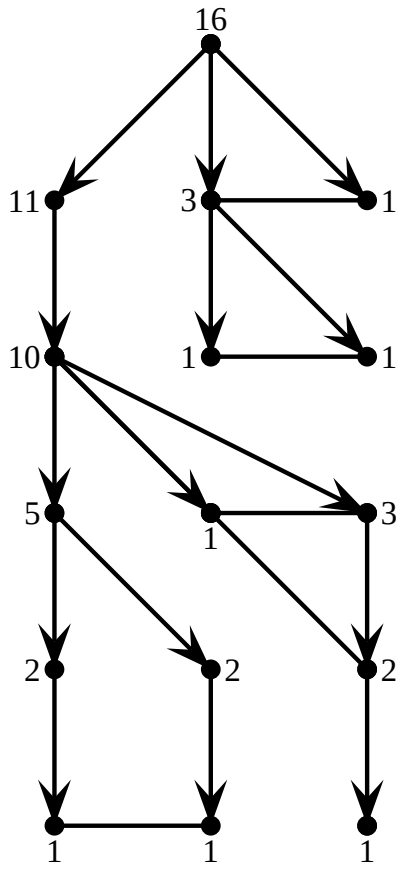


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

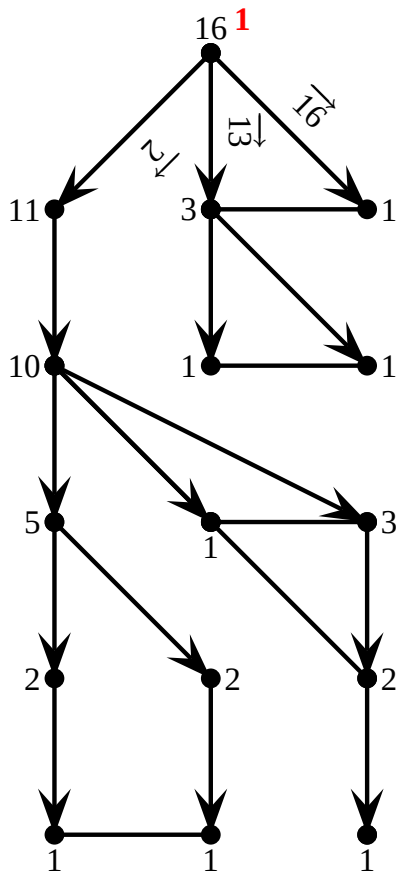


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

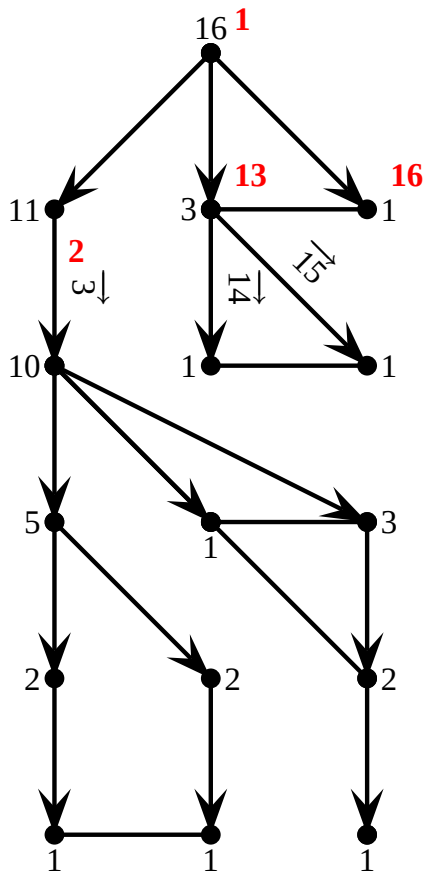


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

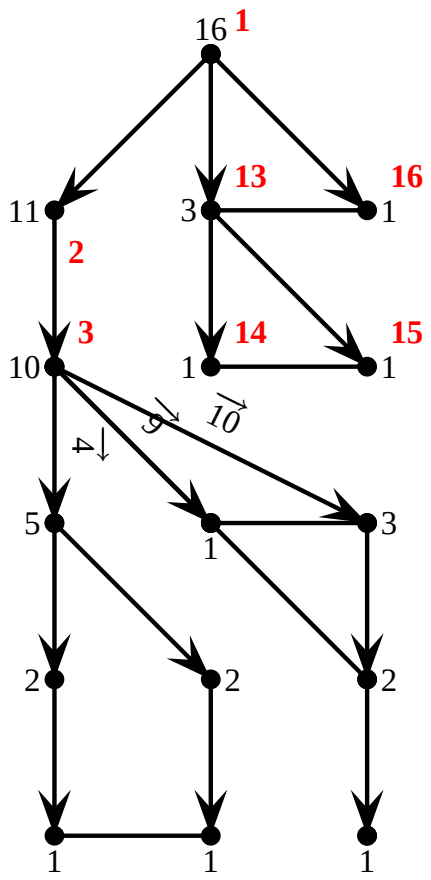


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

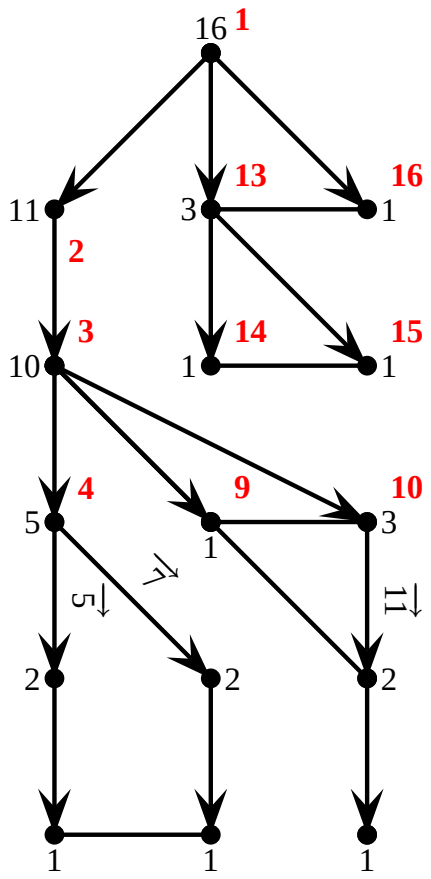


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

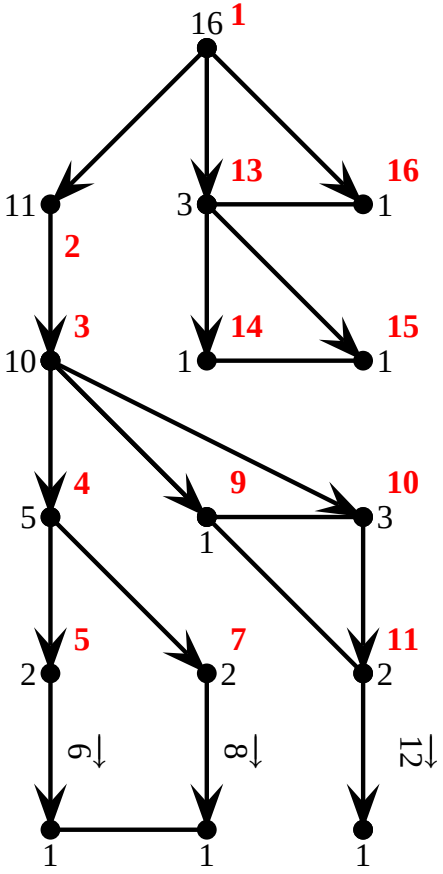


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.

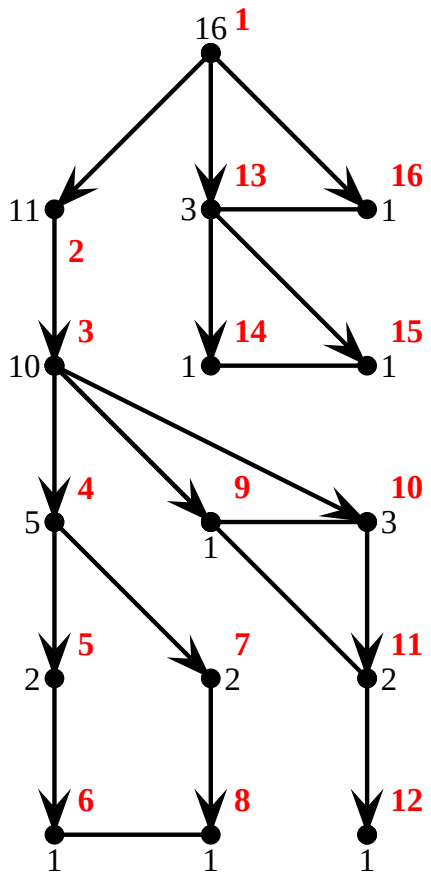


2. Preordering the Vertices

- In a *preorder*, the label of a vertex is smaller than the label of each of its children.
- We can compute a preorder of the vertices with respect to $\mathcal{T}$ distributively.
- The root gives itself label 1.
- When node v labels itself x, it orders its children arbitrarily as $c_1, c_2, \ldots$. Then it sends "Label yourself ℓ_i" to each c_i, where ℓ_i is computed by v as

$$\ell_i = x + 1 + \sum_{j<i} \#desc(c_j).$$

- Takes $height(\mathcal{T})$ time.



3. Marking Cycles

- Hereafter we refer to every node by its preorder label.
- In order to mark cycles we take advantage of some nice properties of "lowest common ancestors" in preorder. Let $\mathrm{LCA}(u, \ldots, v)$ denote the lowest node of $\mathcal{T}$ that is an ancestor of all of $u, \ldots, v$.
- Note that the descendants of node v are precisely

$$\{u \mid v \leq u < v + \#desc(v)\}.$$

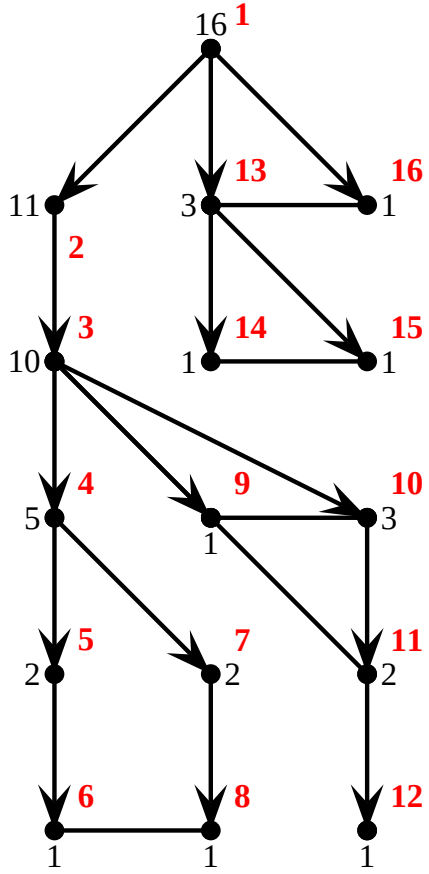
Corollary

1. *If* $v_1 \leq v_2 \leq v_3$, *then* $\mathrm{LCA}(v_1, v_3)$ *is an ancestor of* v_2.
2. $\mathrm{LCA}(u_1, u_2, \ldots u_k) = \mathrm{LCA}(\min_i(u_i), \max_i(u_i))$.
3. *If* $u_i \leq v_i$ *for all* i, *then*

$$\mathrm{LCA}(\mathrm{LCA}(u_1, v_1), \ldots \mathrm{LCA}(u_k, v_k)) = \mathrm{LCA}(\min_i(u_i), \max_i(v_i)).$$

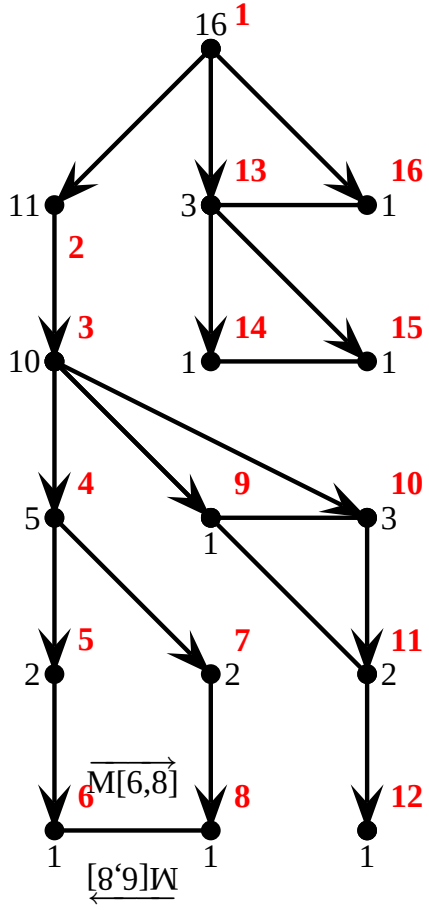
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



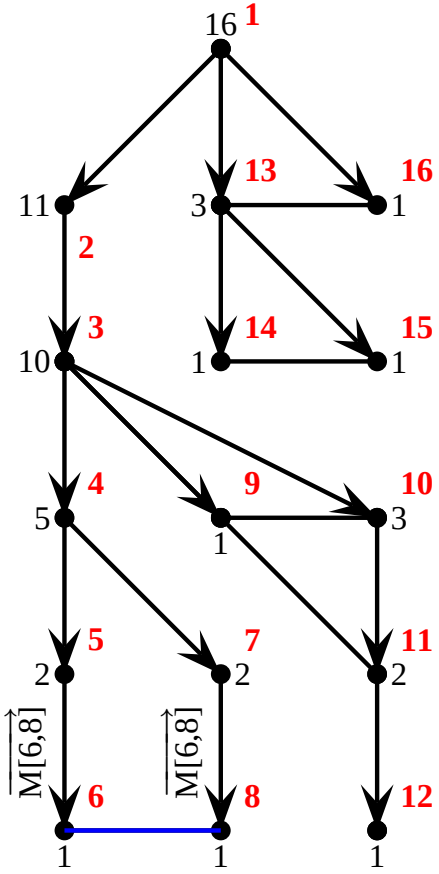
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



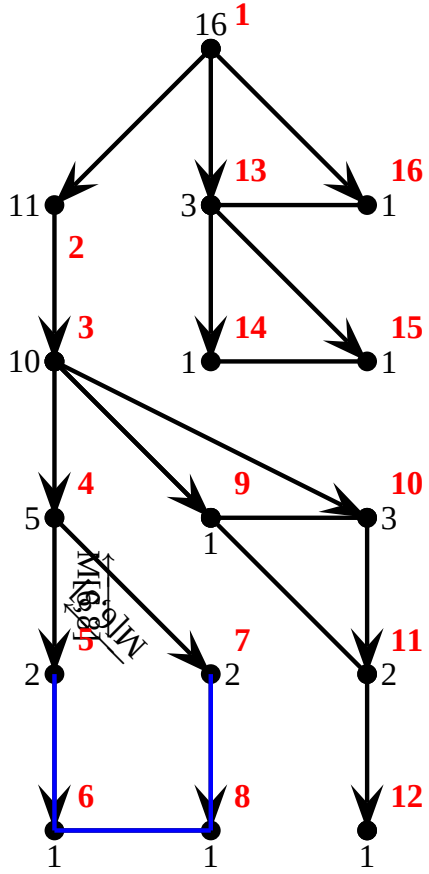
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



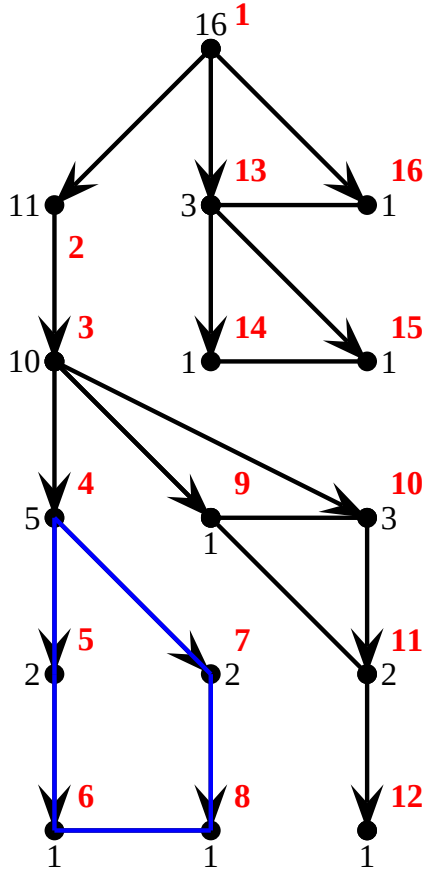
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



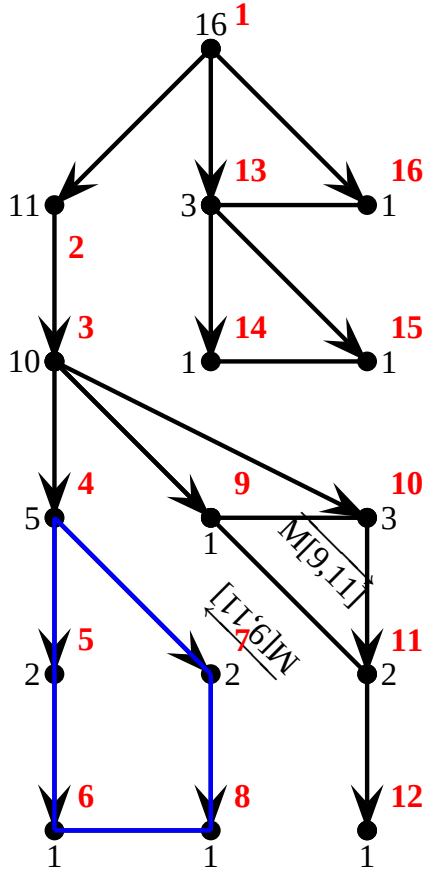
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



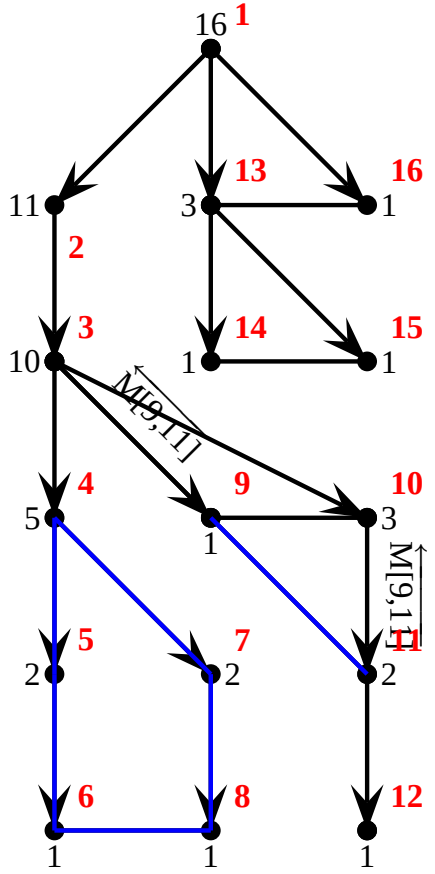
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



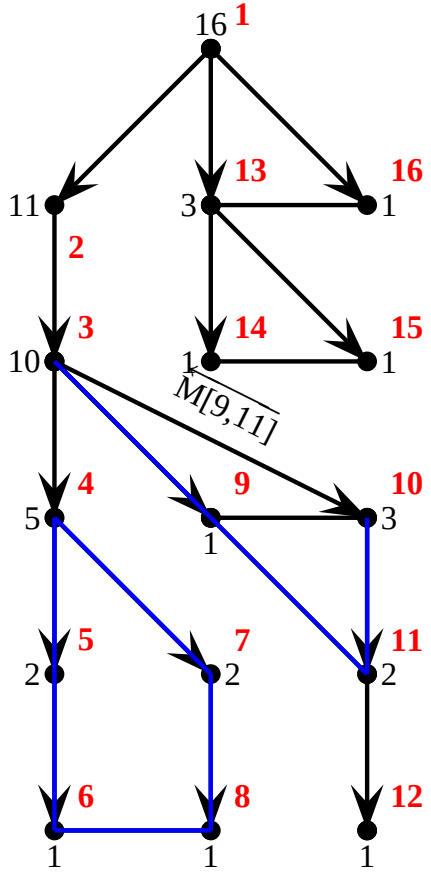
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message M$[u, v]$ along the edge in both directions:
- M$[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



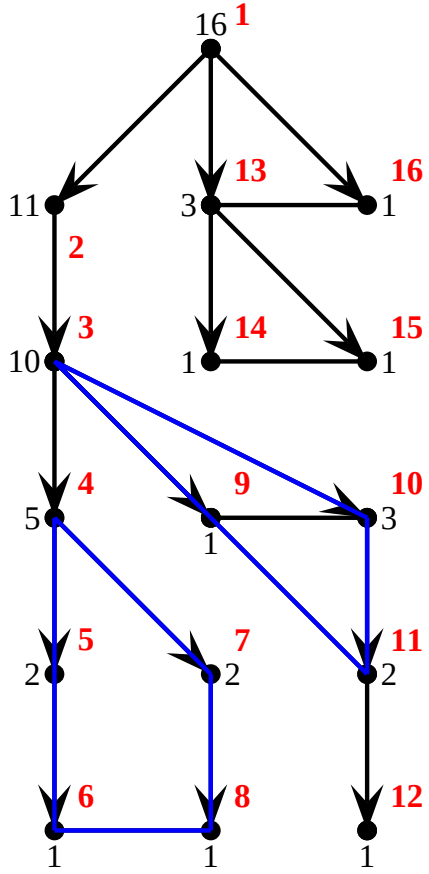
3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message M$[u, v]$ along the edge in both directions:
- M$[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



3. Marking Cycles

- Each non-tree edge e determines a single *fundamental cycle* of $\mathcal{T} \cup \{e\}$. It can be shown that each non-bridge edge lies in some fundamental cycle and so once we mark these cycles, only bridges are unmarked.
- Non-tree edges are never bridges.
- For a given non-tree edge (u, v), to mark its fundamental cycle, send the message $\mathrm{M}[u, v]$ along the edge in both directions:
- $\mathrm{M}[u, v]$: "If you are an ancestor of both u and v, then ignore this message. Otherwise, pass this message up to your parent, and mark the edge joining you to your parent."
- When node w checks ancestry condition it just checks if $\{u, v\} \subseteq \{w, \ldots, w + \#desc(w) - 1\}$.



3. Marking Cycles

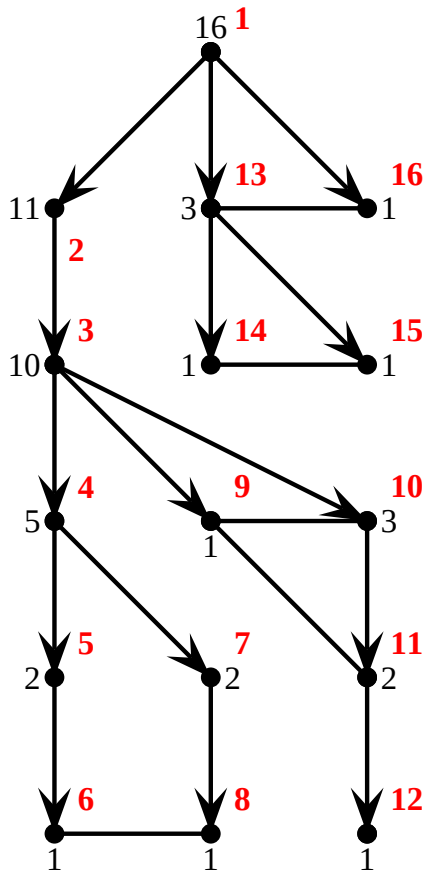
- We are almost done, the protocol described will correctly mark all non-bridges.
- The problem is, a vertex can receive several $\mathrm{M}[u, v]$ messages at once, whereas (due to the limit on message size) only $O(1)$ can be forwarded to its parent in any round.
- Without loss of generality each $\mathrm{M}[u, v]$ is sent with $u \leq v$.
- The fix relies on the earlier corollary, *If* $u_i \leq v_i$ *for all* i, *then*

$$\mathrm{LCA}(\mathrm{LCA}(u_1, v_1), \ldots \mathrm{LCA}(u_k, v_k)) = \mathrm{LCA}(\min_i(u_i), \max_i(v_i)).$$

- Namely, when a node receives messages $\mathrm{M}[u_i, v_i]$ it should act as if it received the single message $\mathrm{M}[\min_i u_i, \max_i v_i]$.
- Why? The goal of v sending messages to its parent is to mark a certain chain to some ancestor of v, and this formula will reach the oldest of all ancestors specified by any message. (Formal proof omitted).

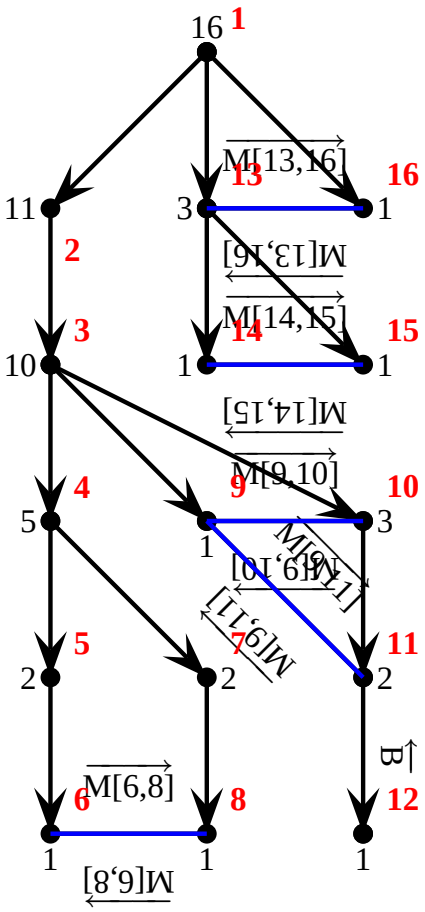
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



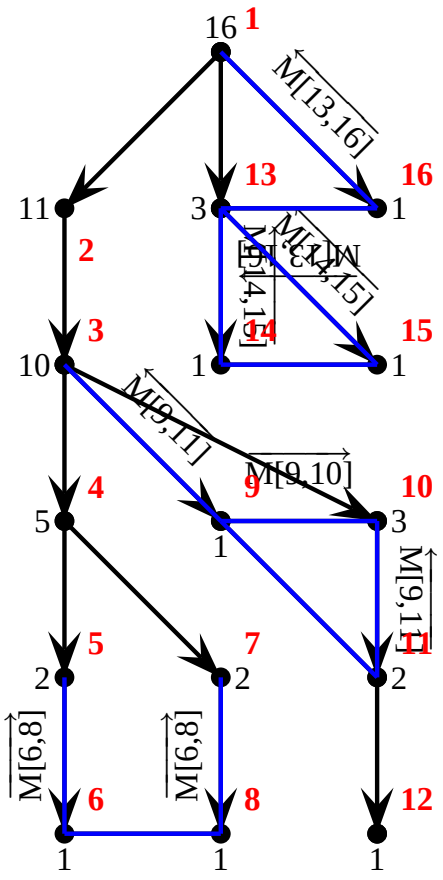
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



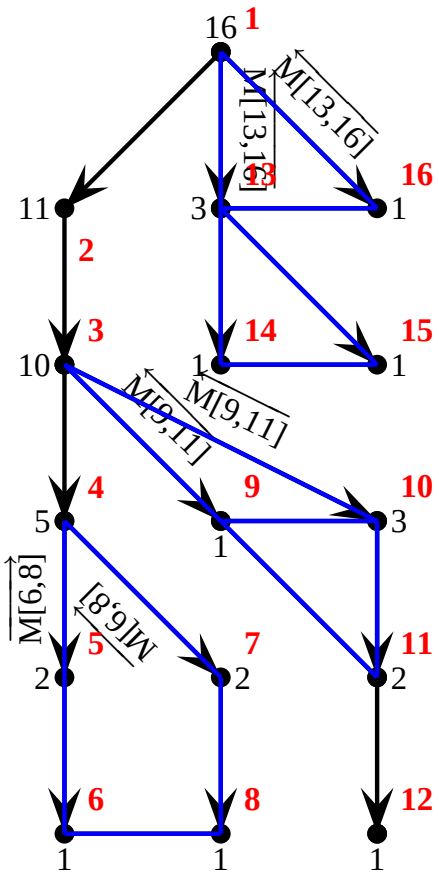
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



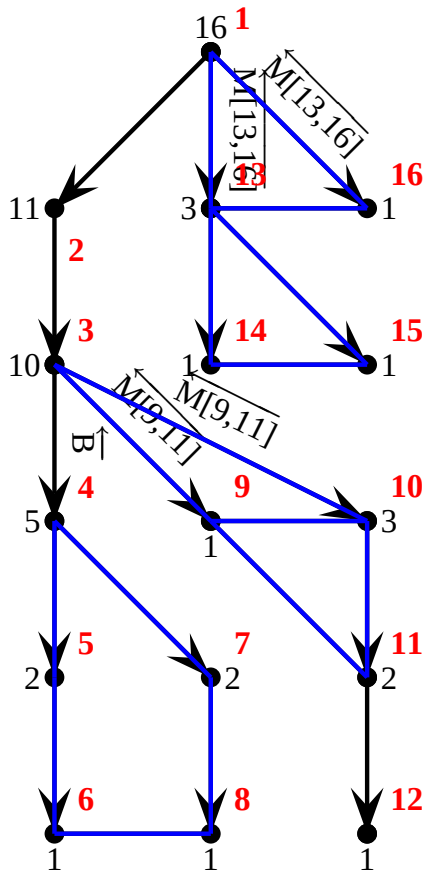
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



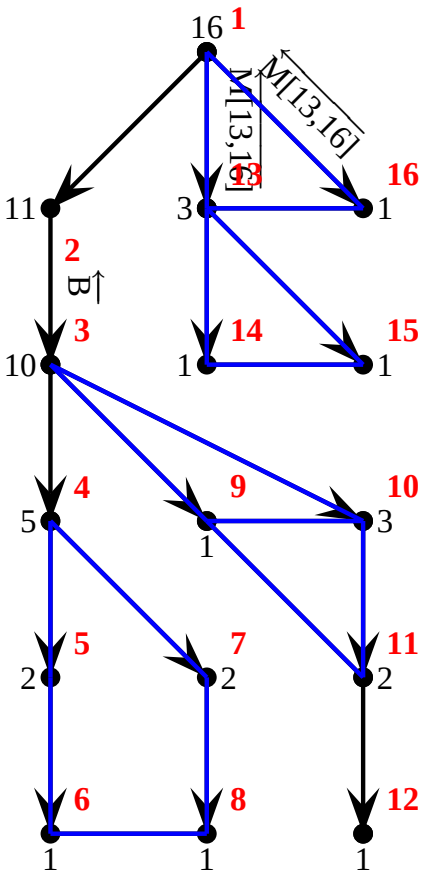
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



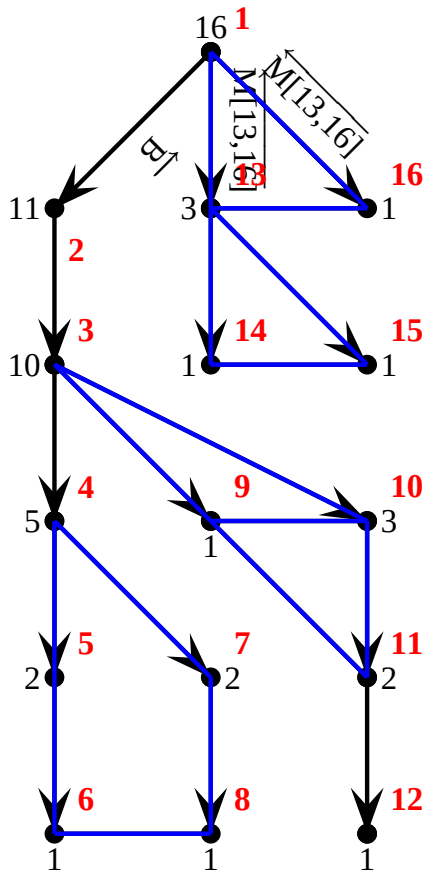
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



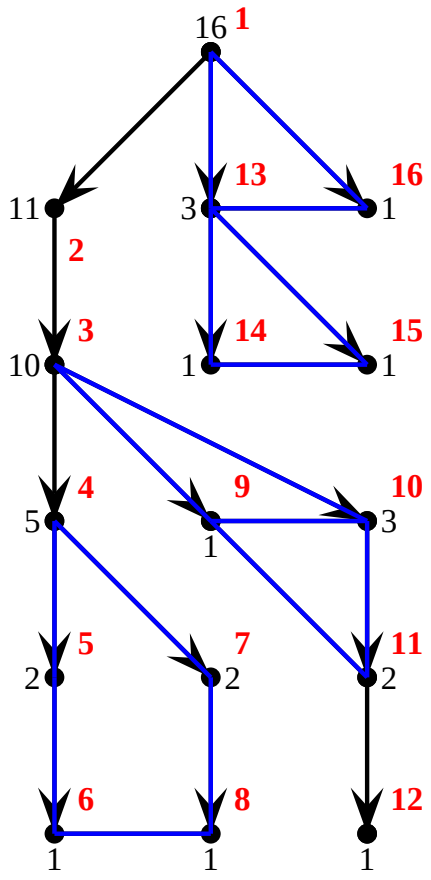
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



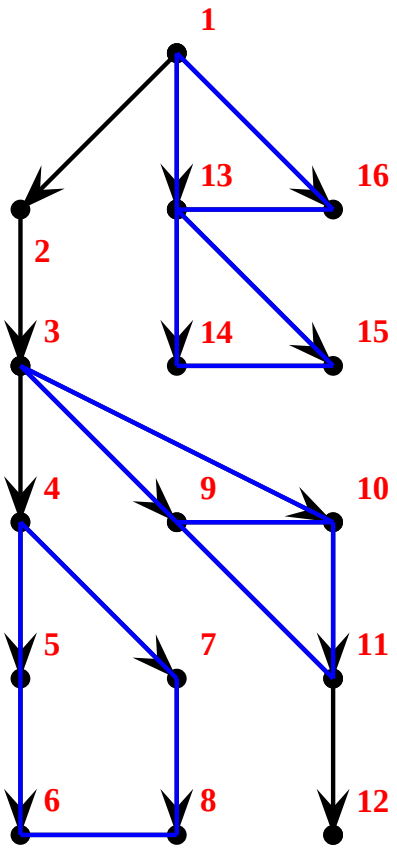
3. Marking Cycles

- Finally, to reduce the total number of messages sent, we want each node to send *exactly* one message to its parent during the marking phase.
- Think of each node storing all received messages in a buffer until it hears from each non-parent neighbour.
- Rather than an explicit buffer, which could grow very large, each node just tracks a cumulative min u_i and max v_i of all the $\mathrm{M}[u_i, v_i]$ messages it has received.
- Even if v determines that its edge to its parent should not be marked, it sends a token message to its parent.
- When v has received all not-to-parent edges incident on v have sent a message, the node sends a message to its parent.



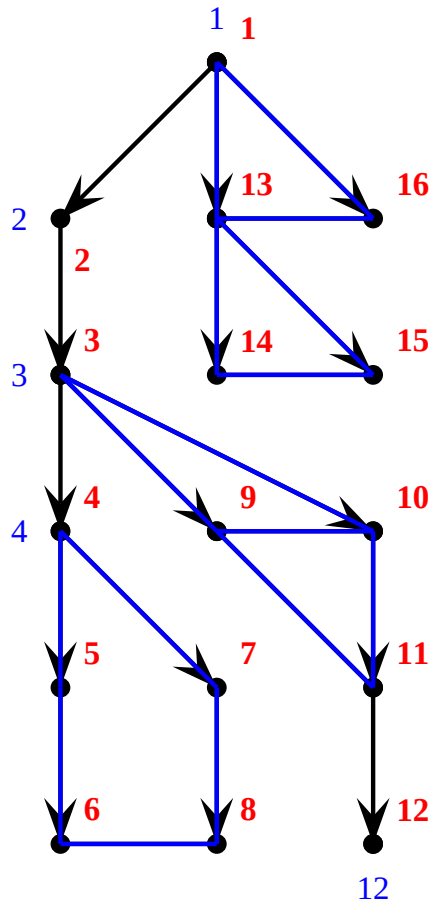
4. Label each node according to its biconnected component.

- As argued (much) earlier, the biconnected components (bccs) are connected, so the spanning tree $\mathcal{T}$ spans the subgraph induced by each bcc.
- In short: labeling is easy.
- The root, and each edge that is just below a bridge, use its own preorder label as its bcc label.
- Each node passes its bcc label downwards once it has been computed, and any edge not just below a bridge uses the label that it receives from its parent.



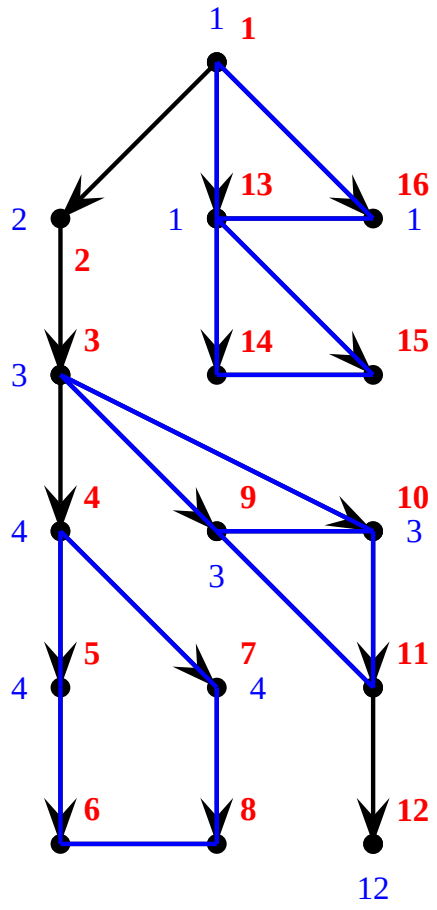
4. Label each node according to its biconnected component.

- As argued (much) earlier, the biconnected components (bccs) are connected, so the spanning tree $\mathcal{T}$ spans the subgraph induced by each bcc.
- In short: labeling is easy.
- The root, and each edge that is just below a bridge, use its own preorder label as its bcc label.
- Each node passes its bcc label downwards once it has been computed, and any edge not just below a bridge uses the label that it receives from its parent.



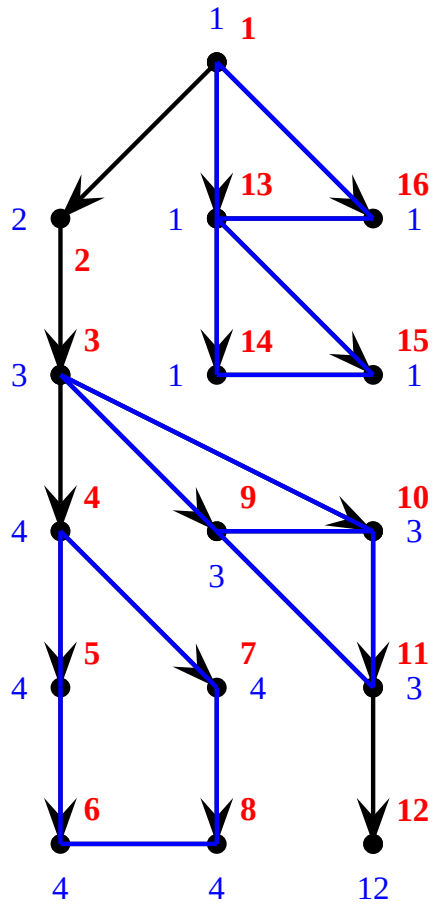
4. Label each node according to its biconnected component.

- As argued (much) earlier, the biconnected components (bccs) are connected, so the spanning tree $\mathcal{T}$ spans the subgraph induced by each bcc.
- In short: labeling is easy.
- The root, and each edge that is just below a bridge, use its own preorder label as its bcc label.
- Each node passes its bcc label downwards once it has been computed, and any edge not just below a bridge uses the label that it receives from its parent.



4. Label each node according to its biconnected component.

- As argued (much) earlier, the biconnected components (bccs) are connected, so the spanning tree $\mathcal{T}$ spans the subgraph induced by each bcc.
- In short: labeling is easy.
- The root, and each edge that is just below a bridge, use its own preorder label as its bcc label.
- Each node passes its bcc label downwards once it has been computed, and any edge not just below a bridge uses the label that it receives from its parent.



Complexity Analysis

1. Each node computes its number of descendants.
2. We *preorder* the nodes.
3. Mark each edge in every cycle of a cycle basis by upcasting.
4. Label each node according to its biconnected component.

- Each edge is used only a constant number of times. Twice to compute $\#desc$, once to preorder, at most twice for $\mathrm{M}[\cdot,\cdot]$ messages, and once to distribute bcc labels. So $O(m)$ communication complexity.
- Each of the 4 steps takes $O(height(\mathcal{T}))$ time to complete.
- All messages have 1 or 2 positive integers less than n, so each message is of length $O(\log n)$.
- The BFS protocol shown earlier gives a spanning tree of height less than Diam, in $O(\mathsf{Diam})$ rounds and using $O(m)$ messages.
- So, as claimed the total complexity is $O(m)$ communication and $O(\mathsf{Diam})$ time.

Outline

Tweaking the Algorithm

- Suppose we remove the restriction on messages sizes, and we start all nodes simultaneously.
- Use the "repeatedly broadcast known topology" approach.
- As soon as any node notices that a given edge is not a bridge, it tells that node.
- The time before all non-bridges are identified is $\Upsilon(G)$, where

$$\Upsilon(G) := \max_{e \text{ not a bridge}} \min_{\substack{K \text{ a cycle} \\ K \ni e}} \min_{v \in V(G)} \max_{u \in K} dist_G(u, v).$$

- Υ is the least value t such that each non-bridge is contained in a cycle belonging to a t-neighbourhood of some node.
- Furthermore this is also a lower bound on the time to identify all non-bridges (up to a constant), no matter what algorithm is used.

A Fast Local Algorithm

- A $(\log n, \Upsilon)$-neighbourhood cover of G is a collection of connected vertex sets called *clusters* such that
  1. For each vertex v, the Υ-neighborhood of v is entirely contained in some cluster.
  2. The subgraph of G induced by each cluster has diameter $O(\Upsilon \log n)$.
  3. Each node belongs to $O(\log n)$ clusters.
- Let us run our edge-biconnectivity algorithm on each cluster, one at a time.
- By definition of Υ, each non-bridge will be identified as such in one of these runs. Each cluster's run completes in time proportional to its diameter $O(\Upsilon \log n)$.
- In fact we can process all clusters in parallel at the cost of a multiplicative $\log n$ time factor due to congestion (e.g., message buffering.)

A Fast Local Algorithm

- A recent result [Elkin, 2004] gives a randomized algorithm for computing sparse neighborhood covers which, with high probability, runs in $O(\Upsilon \log^3 n)$ time and uses $O(m \log^2 n)$ messages on a synchronous network.
- Other wacky graph parameters discussed there.
- Stress: we require that all nodes are started at the same time.
- So the total time complexity of this *local* algorithm is $O(\Upsilon \log^3 n)$, provided that we know Υ.
- Seems hopeless to compute Υ quickly but by guessing $\Upsilon = 1, 2, 4, 8, \ldots$ we quickly guess a large-enough value. Results in an algorithm that is optimal up to log factors.

Open Questions

- Is there any way to compute the articulation points and blocks using a version of the proposed algorithm?
- If we could quickly (i.e., in $O(\mathsf{Diam}) + o(n)$ time) compute a DFS of any given graph, under the $\mathcal{CONGEST}$ model, then we could straightforwardly use Tarjan's algorithm to compute the articulation points. Seems to be very hard but as far as I know there is no impossibility result.

Your Questions?

- **Thank you for attending!**

- M. Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proc. 15th Symp. Discrete Algorithms*, pages 359–368, 2004. Full version at `http://www.cs.yale.edu/~elkin/mst.jour.ps`.
- S. T. Huang. A new distributed algorithm for the biconnectivity problem. In *Proc. 1989 International Conf. Parallel Processing*, pages 106–113, 1989.
- D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- R. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- R. E. Tarjan. A note on finding the bridges of a graph. *Inform. Process. Lett.*, 2:160– 161, 1974.
- R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.
- R. Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. In *Proc. 14th Symp. Principles of Distributed Computing*, pages 28–37, 1995. Journal: *J. Algorithms*, 23(1):160–179, 1997.