Self-stabilization

All configurations are initial configurations.

An algorithm is self-stabilizing if every computation eventually reaches a correct configuration.

Advantages:

- fault tolerance
- straightforward initialization

Self-stabilizing *operating systems* and *databases* have been developed.

Self-stabilization - Shared memory

In a *message-passing* setting, processes might all be initialized in a state where they are waiting for a message.

Then the self-stabilizing algorithm wouldn't exhibit any behavior.

Therefore, in self-stabilizing algorithms, processes communicate via variables in *shared memory*.

We assume that a process can read the variables of its neighbors.

Dijkstra's self-stabilizing token ring

Processes $p_0, \ldots, p_{N-1}$ form a directed ring.

Each p_i holds a value $x_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i for each $i = 1, \ldots, N-1$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $x_i \leftarrow x_{i-1}$ when $x_i \neq x_{i-1}$, for each $i = 1, \ldots, N-1$
- $x_0 \leftarrow (x_0 + 1) \bmod K$ when $x_0 = x_{N-1}$

If $K \geq N$, then Dijkstra's token ring *self-stabilizes*.

That is, each computation eventually satisfies *mutual exclusion*.

Moreover, Dijkstra's token ring is *starvation-free*.

Dijkstra's self-stabilizing token ring

Processes $p_0, \ldots, p_{N-1}$ form a directed ring.

Each p_i holds a value $x_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i for each $i = 1, \ldots, N-1$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $x_i \leftarrow x_{i-1}$ when $x_i \neq x_{i-1}$, for each $i = 1, \ldots, N-1$
- $x_0 \leftarrow (x_0 + 1) \bmod K$ when $x_0 = x_{N-1}$

If $K \geq N$, then Dijkstra's token ring *self-stabilizes*.

That is, each computation eventually satisfies *mutual exclusion*.

Moreover, Dijkstra's token ring is *starvation-free*.

Dijkstra's self-stabilizing token ring

Processes $p_0, \ldots, p_{N-1}$ form a directed ring.

Each p_i holds a value $x_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i for each $i = 1, \ldots, N-1$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $x_i \leftarrow x_{i-1}$ when $x_i \neq x_{i-1}$, for each $i = 1, \ldots, N-1$
- $x_0 \leftarrow (x_0 + 1) \bmod K$ when $x_0 = x_{N-1}$

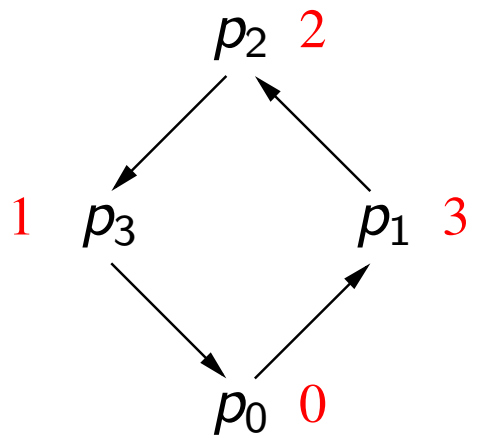
If $K \geq N$, then Dijkstra's token ring *self-stabilizes*.

That is, each computation eventually satisfies *mutual exclusion*.

Moreover, Dijkstra's token ring is *starvation-free*.

Dijkstra's token ring - Example

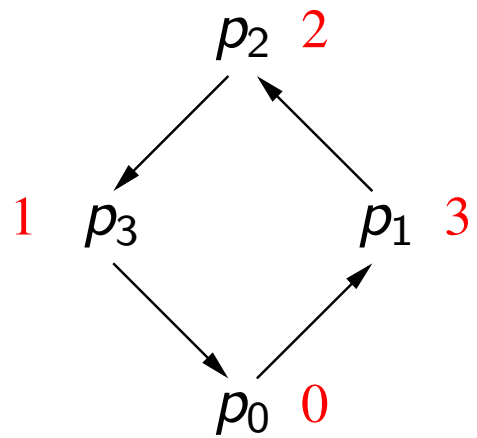
Let $N = K = 4$. Consider the initial configuration



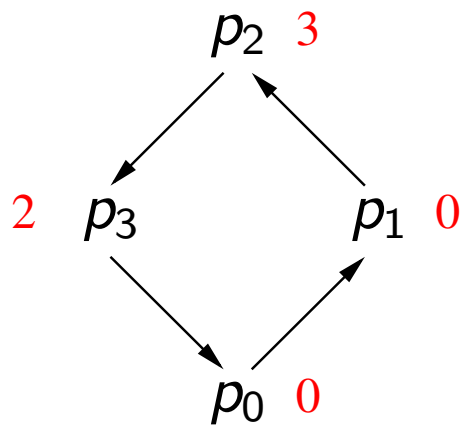
It isn't hard to see that the ring self-stabilizes. For instance,

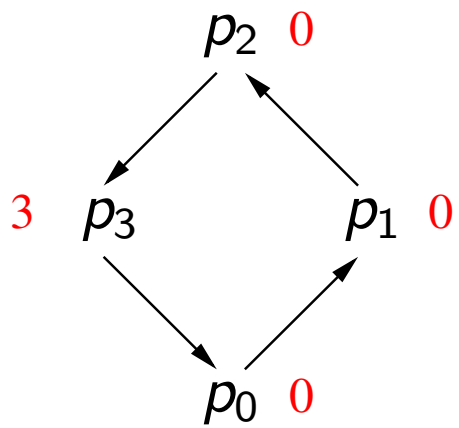
Dijkstra's token ring - Example

Let $N = K = 4$. Consider the initial configuration



It isn't hard to see that the ring self-stabilizes. For instance,





Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Question

Let $N \geq 3$. Argue that Dijkstra's token ring self-stabilizes if $K = N - 1$.

This lower bound for K is sharp! (See the next slide.)

Question

Let $N \geq 3$. Argue that Dijkstra's token ring self-stabilizes if $K = N - 1$.

This lower bound for K is sharp! (See the next slide.)

Answer: Consider any computation.

At some moment, p_{N-1} copies the value from p_{N-2}.

Then $p_1, \ldots, p_{N-1}$ hold $\leq N - 2$ different values (because $N \geq 3$).

Since $p_1, \ldots, p_{N-1}$ only copy values, they hold these $\leq N - 2$ values as long as x_0 equals x_i for some $i = 1, \ldots, N - 1$.

Since $K \geq N - 1$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N - 1$.

Question

Let $N \geq 3$. Argue that Dijkstra's token ring self-stabilizes if $K = N - 1$.

This lower bound for K is sharp! (See the next slide.)

Answer: Consider any computation.

At some moment, p_{N-1} copies the value from p_{N-2}.

Then $p_1, \ldots, p_{N-1}$ hold $\leq N - 2$ different values (because $N \geq 3$).

Since $p_1, \ldots, p_{N-1}$ only copy values, they hold these $\leq N - 2$ values as long as x_0 equals x_i for some $i = 1, \ldots, N - 1$.

Since $K \geq N - 1$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N - 1$.

Question

Let $N \geq 3$. Argue that Dijkstra's token ring self-stabilizes if $K = N - 1$.

This lower bound for K is sharp ! (See the next slide.)

Answer: Consider any computation.

At some moment, p_{N-1} copies the value from p_{N-2}.

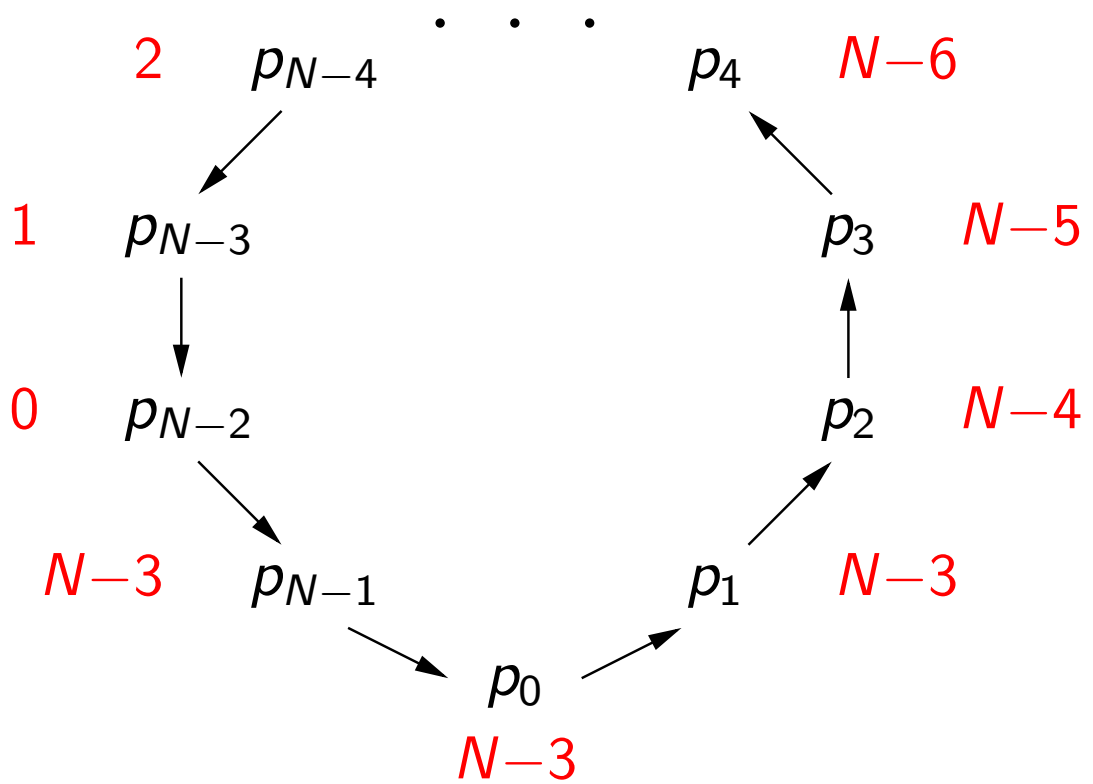
Then $p_1, \ldots, p_{N-1}$ hold $\leq N - 2$ different values (because $N \geq 3$).

Since $p_1, \ldots, p_{N-1}$ only copy values, they hold these $\leq N - 2$ values as long as x_0 equals x_i for some $i = 1, \ldots, N - 1$.

Since $K \geq N - 1$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N - 1$.

Dijkstra's token ring - Non-stabilization if $K = N - 2$

Example: Let $N \geq 4$ and $K = N - 2$, and consider the following initial configuration.



It doesn't always self-stabilize.

Dijkstra's token ring - Complexity

Worst-case message complexity: Mutual exclusion is achieved after at most $O(N^2)$ events.

p_i for $0 < i < N$ can copy the initial values of $p_0, \ldots, p_{i-1}$.

(Total: $\leq \frac{1}{2}(N-1)N$ events.)

p_0 takes on at most N new values to attain a *fresh* value.

These values can be copied by $p_1, \ldots, p_{N-1}$. (Total: $\leq N^2$ events.)

Arora-Gouda self-stabilizing spanning tree algorithm

Given an undirected network.

An upper bound K on network size is known to all processes.

The process with the largest id becomes the *root*.

Each process p maintains the following variables:

$parent_p$: its parent in the spanning tree

$root_p$: the root of the spanning tree

$dist_p$: its distance from the root via the spanning tree

Arora-Gouda self-stabilizing spanning tree algorithm

Given an undirected network.

An upper bound K on network size is known to all processes.

The process with the largest id becomes the *root*.

Each process p maintains the following variables:

$parent_p$:	its parent in the spanning tree
$root_p$:	the root of the spanning tree
$dist_p$:	its distance from the root via the spanning tree

Arora-Gouda spanning tree algorithm - Complications

Due to arbitrary initialization, there are three complications.

Complication 1: Multiple processes may consider themselves root.

Complication 2: There may be a cycle in the spanning tree.

Complication 3: $root_p$ may not be the id of any process in the network.

Arora-Gouda spanning tree algorithm

A non-root p declares itself *root*, i.e.

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0$$

if it detects an inconsistency in its *local* variables:

- $root_p < p$, or
- $parent_p = \perp$, and $root_p \neq p$ or $dist_p > 0$, or
- $parent_p \neq \perp$ and $parent_p$ isn't a neighbor of p, or
- $dist_p \geq K$.

Arora-Gouda spanning tree algorithm

Let there be no inconsistency in the local variables of p.

Let q be a neighbor of p with $dist_q < K$.

Suppose $q = parent_p$.

If $root_p \neq root_q$, then $root_p \leftarrow root_q$.

If $dist_p \neq dist_q + 1$, then $dist_p \leftarrow dist_q + 1$.

Suppose $q \neq parent_p$.

If $root_p < root_q$, then

$parent_p \leftarrow q$ $root_p \leftarrow root_q$ $dist_p \leftarrow dist_q + 1$

Arora-Gouda spanning tree algorithm

Let there be no inconsistency in the local variables of p.

Let q be a neighbor of p with $dist_q < K$.

Suppose $q = parent_p$.

If $root_p \neq root_q$, then $root_p \leftarrow root_q$.

If $dist_p \neq dist_q + 1$, then $dist_p \leftarrow dist_q + 1$.

Suppose $q \neq parent_p$.

If $root_p < root_q$, then

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Question

Suppose that during an application of the Arora-Gouda algorithm, the created directed network contains a cycle with a "false" root.

Why is such a cycle always eventually broken ?

Question

Suppose that during an application of the Arora-Gouda algorithm, the created directed network contains a cycle with a "false" root.

Why is such a cycle always eventually broken ?

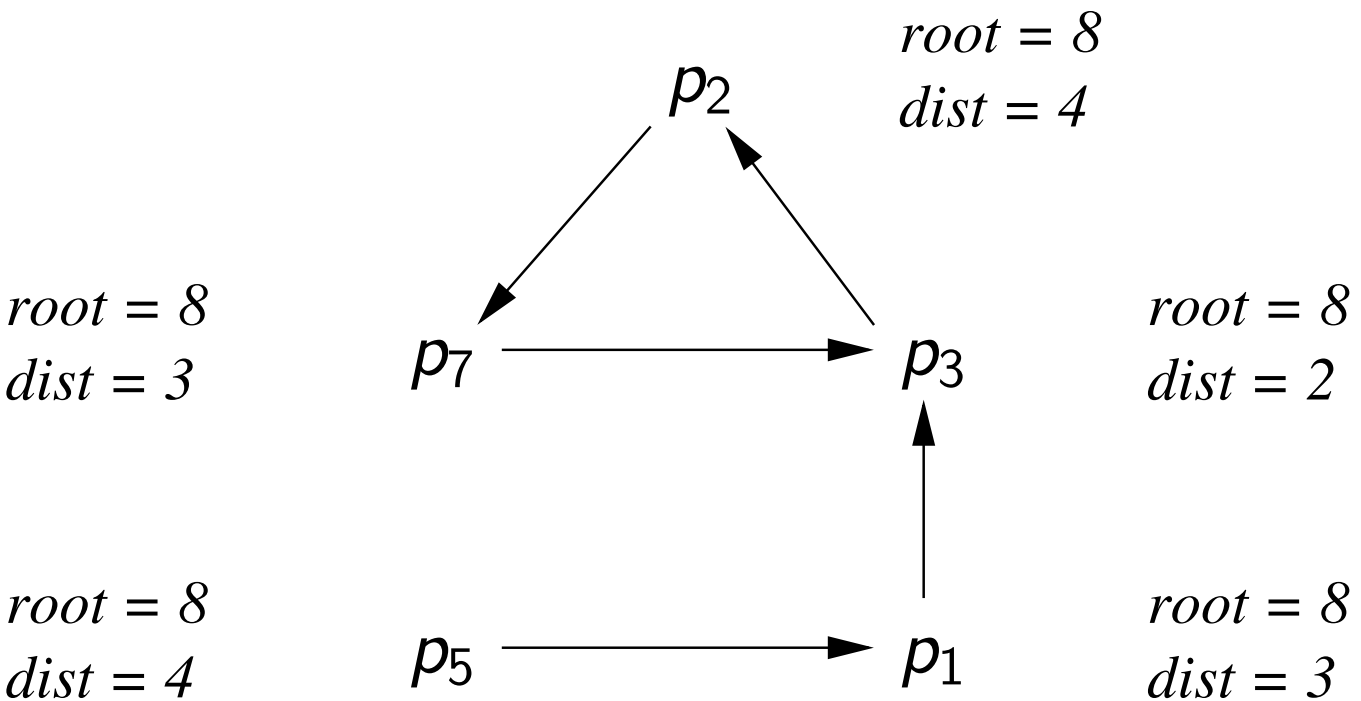
Answer: Always at some p on this cycle, $dist_p \neq dist_{parent_p} + 1$.

So distance values will increase, until at some p on this cycle, $dist_p = K$.

Then p declares itself root.

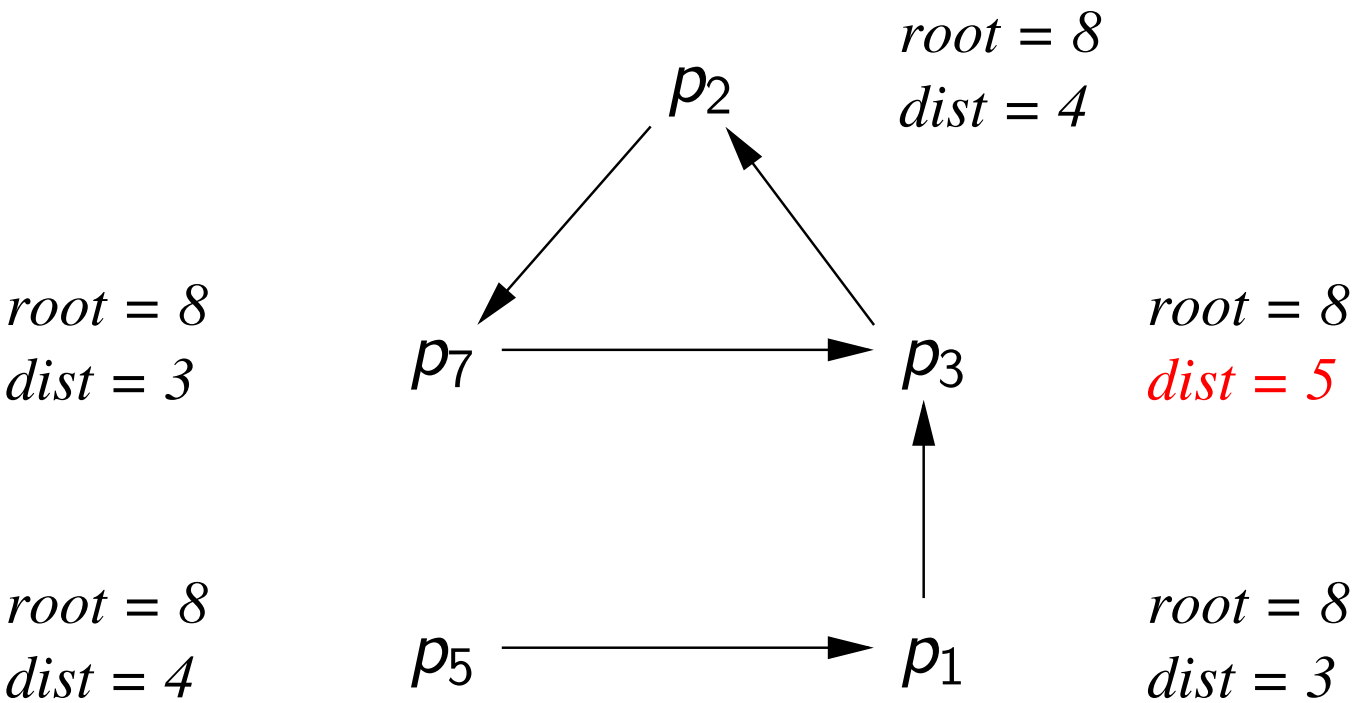
Arora-Gouda spanning tree algorithm - Example

$K = 5$



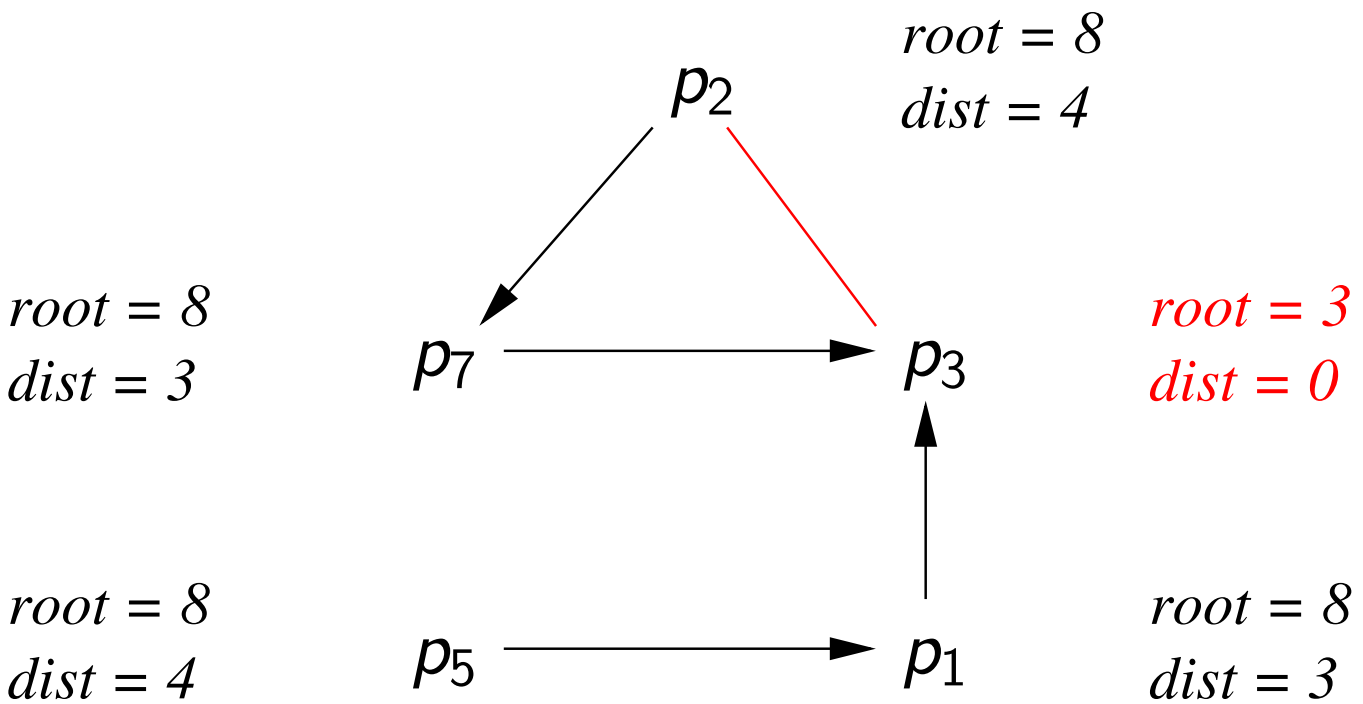
Arora-Gouda spanning tree algorithm - Example

$K = 5$



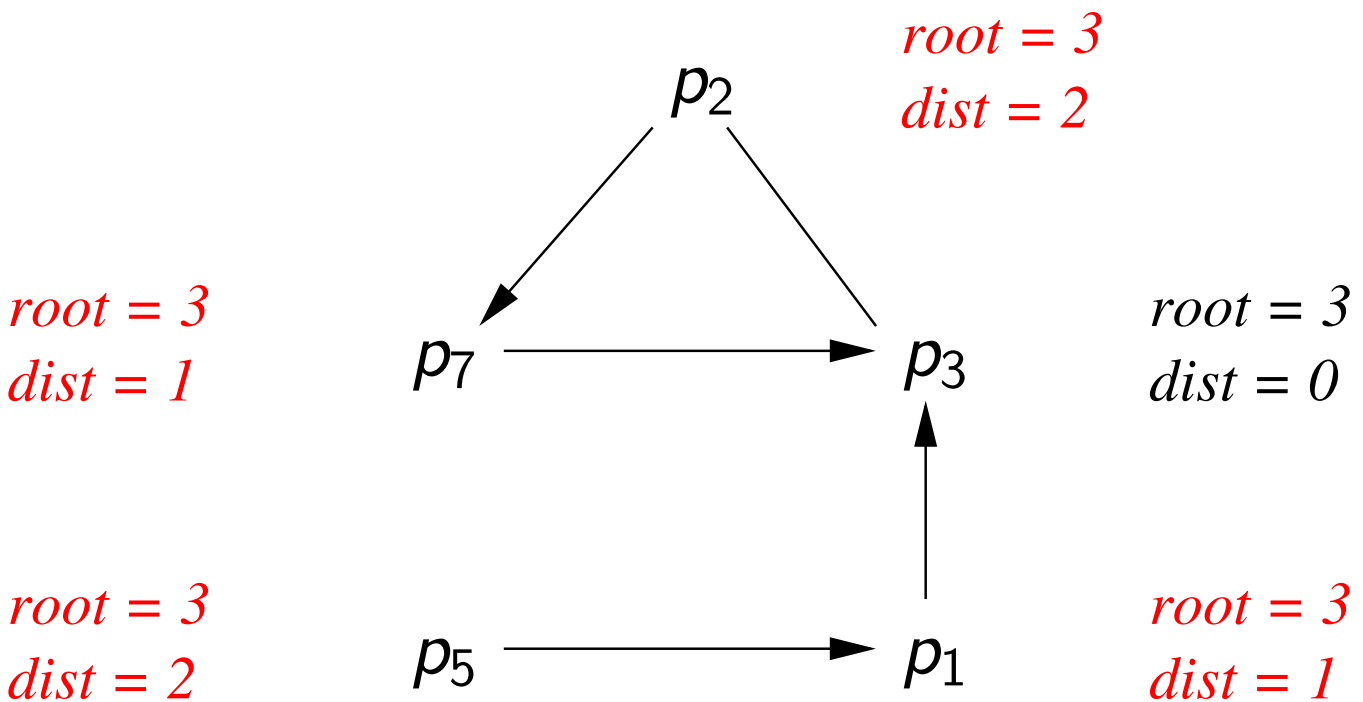
Arora-Gouda spanning tree algorithm - Example

$K = 5$



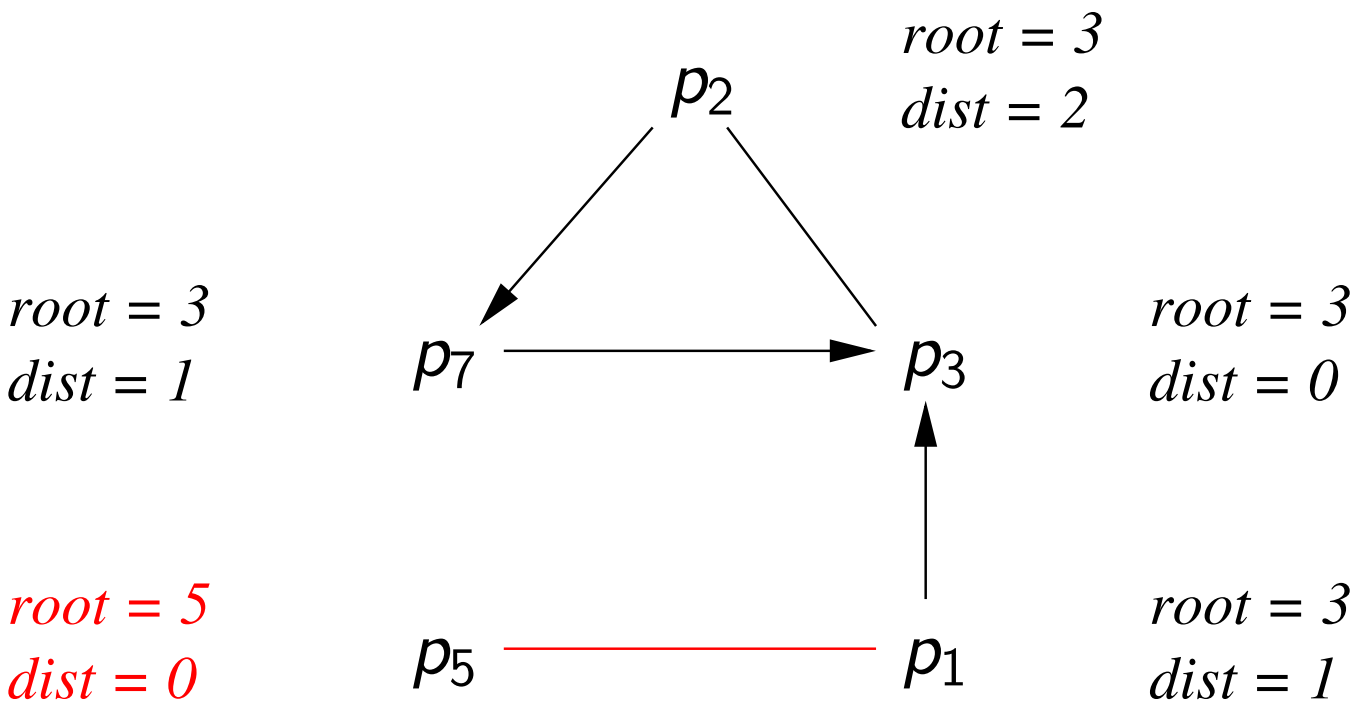
Arora-Gouda spanning tree algorithm - Example

$K = 5$



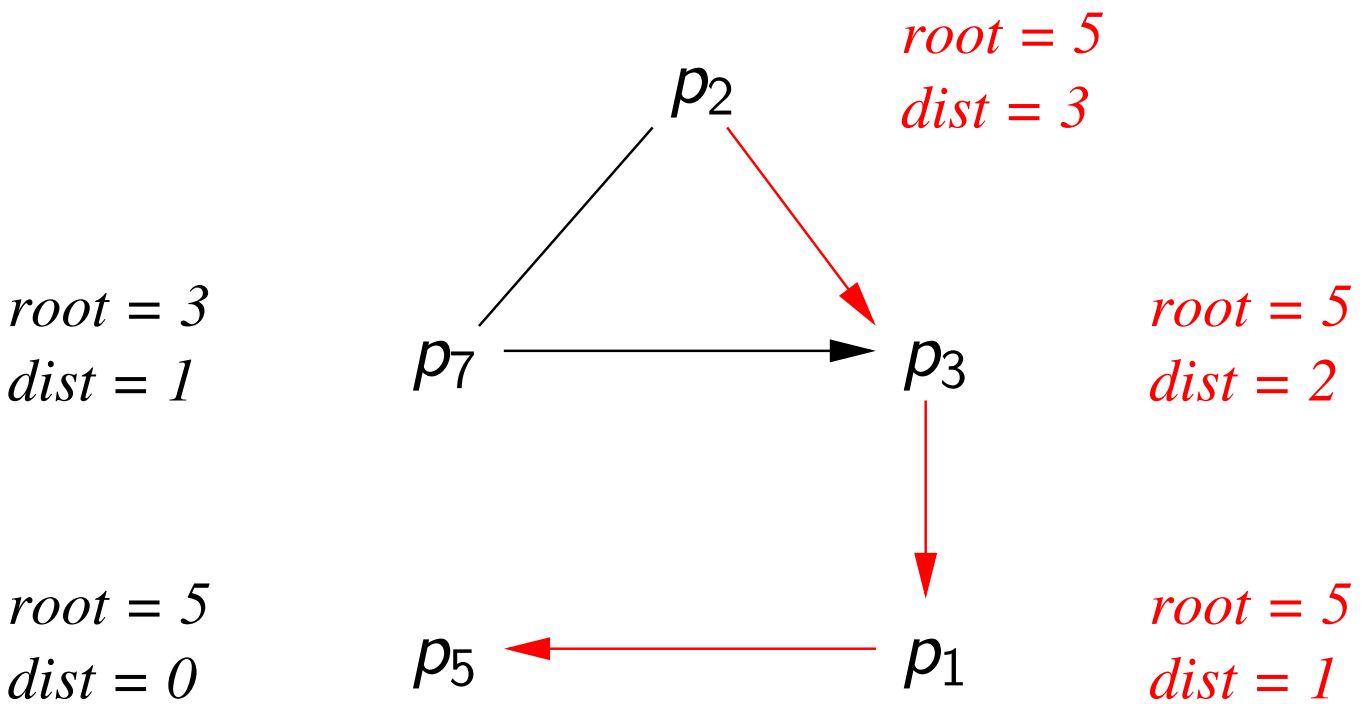
Arora-Gouda spanning tree algorithm - Example

$K = 5$



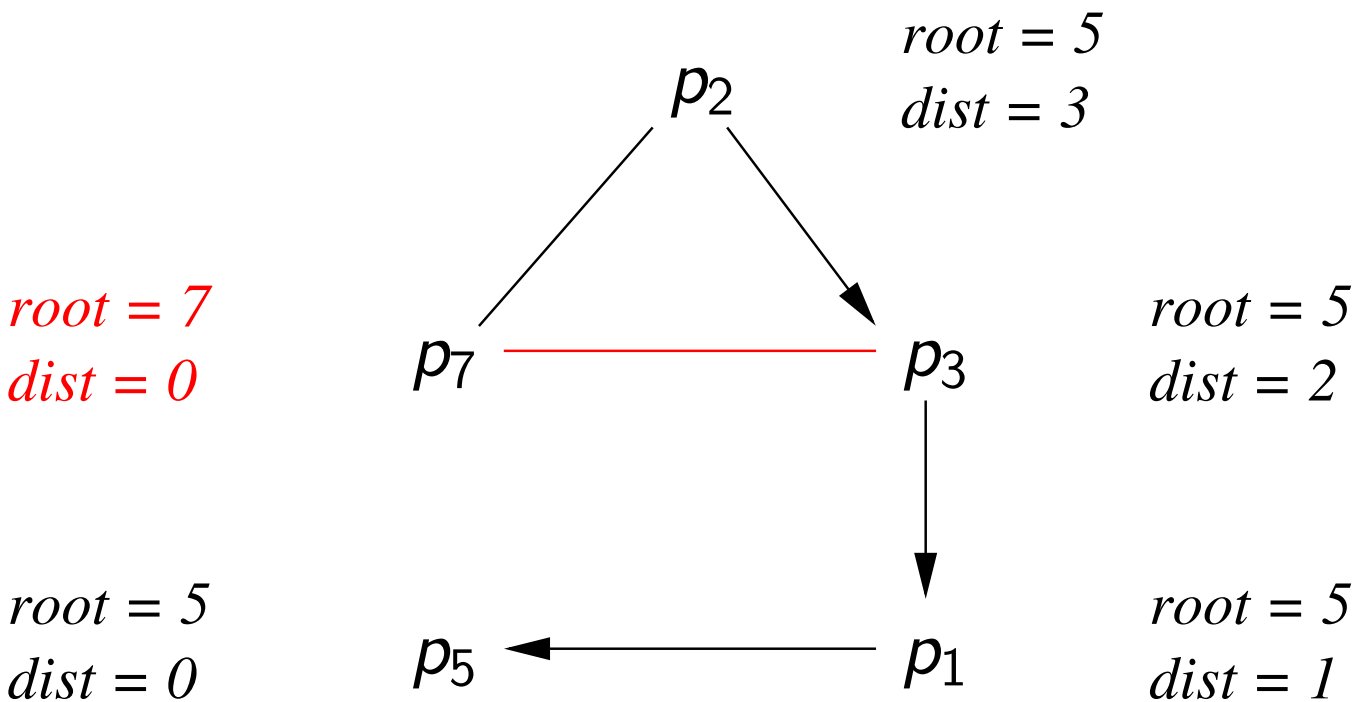
Arora-Gouda spanning tree algorithm - Example

$K = 5$



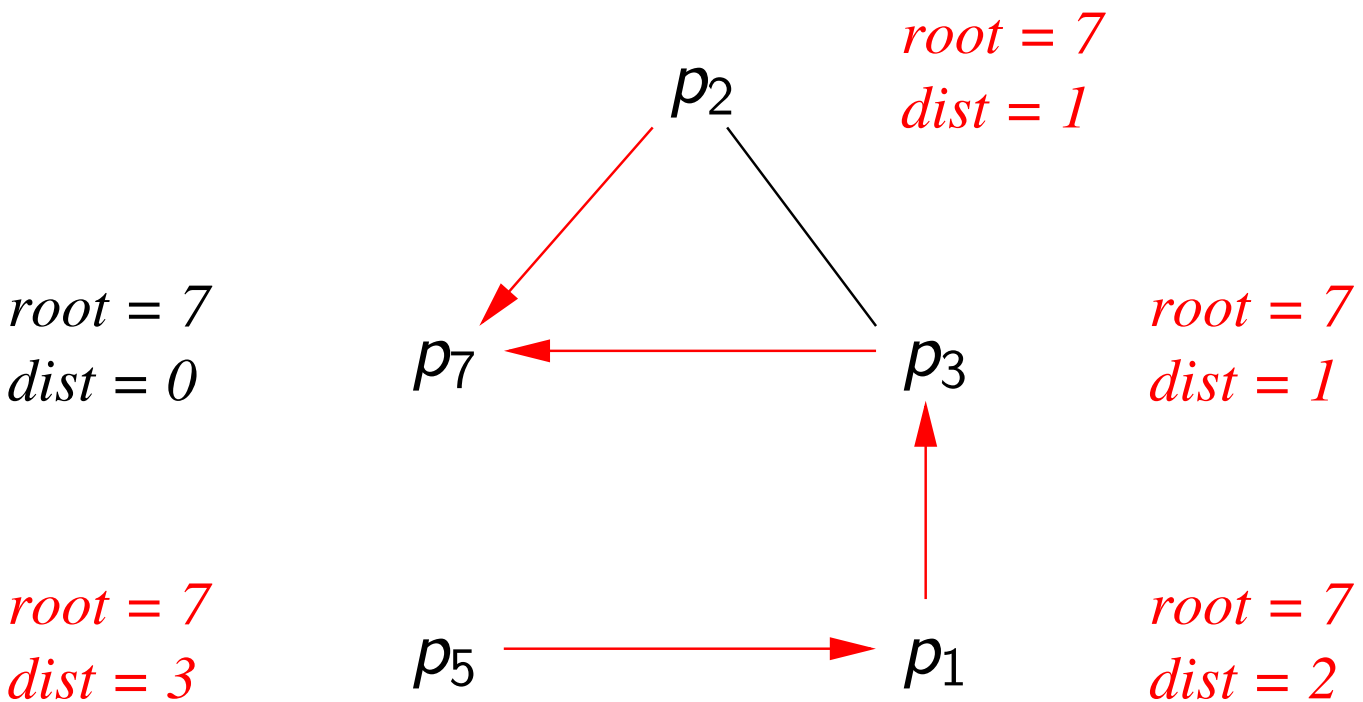
Arora-Gouda spanning tree algorithm - Example

$K = 5$



Arora-Gouda spanning tree algorithm - Example

$K = 5$



Arora-Gouda spanning tree algorithm - Correctness

"False" root values, which aren't an id of any process in the network, will eventually disappear in each fair computation.

Namely, such false roots can only "survive" if there is a cycle of processes that all have this root value.

Distance values of processes on such a cycle will keep on increasing, until one of them gets distance K and declares itself root.

By fairness the cycle can only be reestablished finitely many times.

Therefore the process with the largest id will eventually declare itself root.

Then the network will converge to a spanning tree with this process as root.

Arora-Gouda spanning tree algorithm - Correctness

"False" root values, which aren't an id of any process in the network, will eventually disappear in each fair computation.

Namely, such false roots can only "survive" if there is a cycle of processes that all have this root value.

Distance values of processes on such a cycle will keep on increasing, until one of them gets distance K and declares itself root.

By fairness the cycle can only be reestablished finitely many times.

Therefore the process with the largest id will eventually declare itself root.

Then the network will converge to a spanning tree with this process as root.

Afek-Kutten-Yung self-stabilizing spanning tree algorithm

We compute a spanning tree in an undirected network.

As always, each process is supposed to have a unique id.

The process with the largest id becomes the *root*.

Each process p maintains the following variables:

$parent_p$: its parent in the spanning tree

$root_p$: the root of the spanning tree

$dist_p$: its distance from the root via the spanning tree

Afek-Kutten-Yung self-stabilizing spanning tree algorithm

We compute a spanning tree in an undirected network.

As always, each process is supposed to have a unique id.

The process with the largest id becomes the *root*.

Each process p maintains the following variables :

$parent_p$: its parent in the spanning tree

$root_p$: the root of the spanning tree

$dist_p$: its distance from the root via the spanning tree

Afek-Kutten-Yung spanning tree algorithm - Complications

Due to arbitrary initialization, there are three complications.

Complication 1: Multiple processes may consider themselves root.

Complication 2: There may be a cycle in the spanning tree.

Complication 3: $root_p$ may not be the id of any process in the network.

Afek-Kutten-Yung spanning tree algorithm

A *non-root* p declares itself *root*, i.e.

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0$$

if it detects an inconsistency in its *root* or *parent* value, or with the *root* or *dist* value of its parent :

- $root_p \leq p$, or
- $parent_p = \perp$, or
- $parent_p \neq \perp$, and $parent_p$ isn't a neighbor of p or $root_p \neq root_{parent_p}$ or $dist_p \neq dist_{parent_p} + 1$.

Afek-Kutten-Yung spanning tree algorithm

A *non-root* p declares itself *root*, i.e.

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0$$

if it detects an inconsistency in its *root* or *parent* value, or with the *root* or *dist* value of its parent :

- $root_p \leq p$, or
- $parent_p = \perp$, or
- $parent_p \neq \perp$, and $parent_p$ isn't a neighbor of p or $root_p \neq root_{parent_p}$ or $dist_p \neq dist_{parent_p} + 1$.

Question

Suppose that during an application of the Afek-Kutten-Yung algorithm, the created directed network contains a cycle with a "false" root.

Why is such a cycle always broken ?

Question

Suppose that during an application of the Afek-Kutten-Yung algorithm, the created directed network contains a cycle with a "false" root.

Why is such a cycle always broken ?

Answer: At some p on this cycle, $dist_p \neq dist_{parent_p} + 1$.

So p declares itself root.

Afek-Kutten-Yung spanning tree algorithm

A *root* p makes a neighbor q its parent if $p < root_q$:

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Complication: Processes can infinitely often rejoin a component with a false root.

Afek-Kutten-Yung spanning tree algorithm

A *root* p makes a neighbor q its parent if $p < root_q$:

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Complication: Processes can infinitely often rejoin a component with a false root.

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Join Requests

Before p makes q its parent, it must wait until q's component has a proper root. Therefore p first sends a *join request* to q.

This request is forwarded through q's component, toward the root of this component.

The root sends back an *ack* toward p, which retraces the path of the request.

Only when p receives this ack, it makes q its parent :

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Join requests are only forwarded between "consistent" processes.

Afek-Kutten-Yung spanning tree alg. - Join Requests

Before p makes q its parent, it must wait until q's component has a proper root. Therefore p first sends a *join request* to q.

This request is forwarded through q's component, toward the root of this component.

The root sends back an *ack* toward p, which retraces the path of the request.

Only when p receives this ack, it makes q its parent :

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Join requests are only forwarded between "consistent" processes.

Afek-Kutten-Yung spanning tree alg. - Join Requests

Before p makes q its parent, it must wait until q's component has a proper root. Therefore p first sends a *join request* to q.

This request is forwarded through q's component, toward the root of this component.

The root sends back an *ack* toward p, which retraces the path of the request.

Only when p receives this ack, it makes q its parent :

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Join requests are only forwarded between "consistent" processes.

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Shared memory

A process can only be forwarding and awaiting an ack for at most one join request at a time.

(That's why in the previous example 1 can't send 0's join request on to 0.)

Communication is performed using *shared memory*, so join requests and ack's are encoded in shared variables.

The path of a join request is remembered in local variables.

For simplicity, join requests are here presented in a message passing framework with synchronous communication.

Afek-Kutten-Yung spanning tree alg. - Shared memory

A process can only be forwarding and awaiting an ack for at most one join request at a time.

(That's why in the previous example 1 can't send 0's join request on to 0.)

Communication is performed using *shared memory*, so join requests and ack's are encoded in shared variables.

The path of a join request is remembered in local variables.

For simplicity, join requests are here presented in a message passing framework with synchronous communication.

Afek-Kutten-Yung spanning tree alg. - Consistency check

Given a ring with processes p, q, r, and $s > p, q, r$.

Initially, p and q consider themselves root; r has p as parent and considers s the root.

Since $root_r > q$, q sends a join request to r.

Without the consistency check, r would forward this join request to p.

Since p considers itself root, it would send back an ack to q (via r), and q would make r its parent and consider s the root.

Since $root_r \neq root_p$, r makes itself root.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung spanning tree alg. - Consistency check

Given a ring with processes p, q, r, and $s > p, q, r$.

Initially, p and q consider themselves root; r has p as parent and considers s the root.

Since $root_r > q$, q sends a join request to r.

Without the consistency check, r would forward this join request to p.

Since p considers itself root, it would send back an ack to q (via r), and q would make r its parent and consider s the root.

Since $root_r \neq root_p$, r makes itself root.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung spanning tree alg. - Consistency check

Given a ring with processes p, q, r, and $s > p, q, r$.

Initially, p and q consider themselves root; r has p as parent and considers s the root.

Since $root_r > q$, q sends a join request to r.

Without the consistency check, r would forward this join request to p.

Since p considers itself root, it would send back an ack to q (via r), and q would make r its parent and consider s the root.

Since $root_r \neq root_p$, r makes itself root.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.

Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.

Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.

Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.