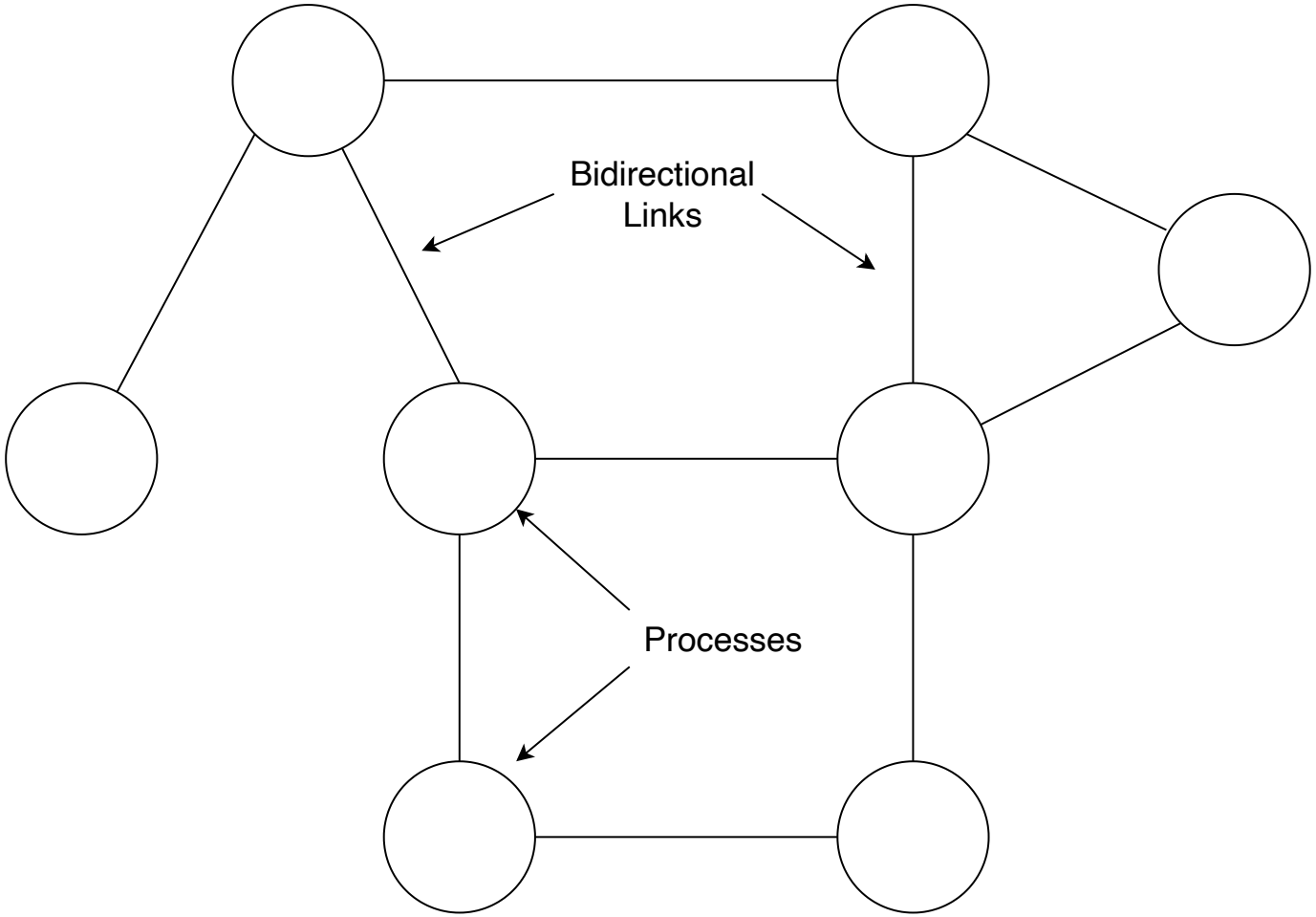
Asynchronous Self-stabilization Made Fast, Simple, and Energy-efficient

S. Devismes, D. Ilcinkas, C. Johnen, F. Mazoit

Université de Picardie Jules Verne, CNRS et Université de Bordeaux

Workshop COA 2024
27 novembre 2024

Distributed Systems



Communication Model

Classical model in the Self-Stab. community [Dijkstra, Commun. ACM, 1974]

Locally shared memory model...

Each node u has **local variables** that can be

- read by u and its neighbors,
- written only by u.

Communication Model

Classical model in the Self-Stab. community [Dijkstra, Commun. ACM, 1974]

Locally shared memory model...

Each node u has **local variables** that can be

- read by u and its neighbors,
- written only by u.

... with composite atomicity

The state of a node is **updated** based on the local neighborhood in an **atomic step**.

Terminating Synchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

Terminating Synchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

Terminating Synchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Terminating Synchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.

Terminating Synchronous Algorithms

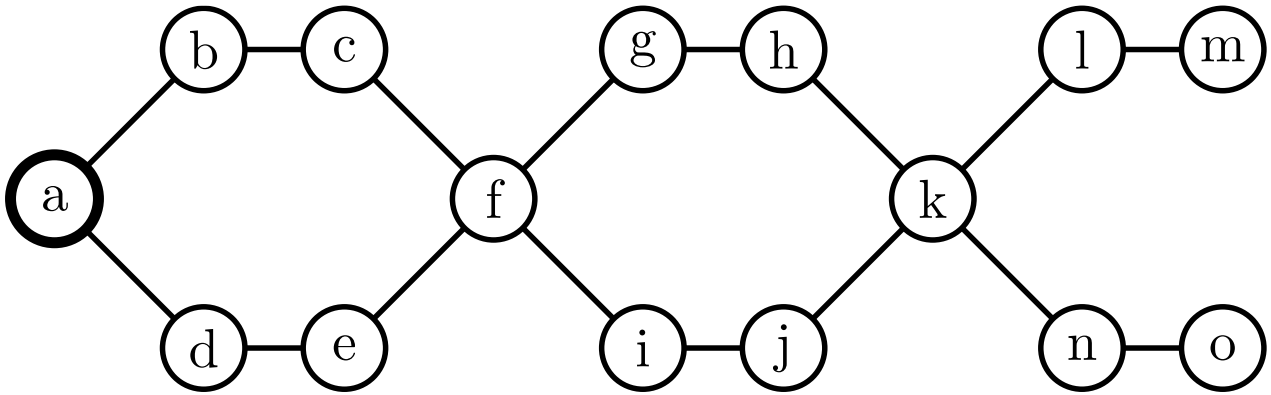
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

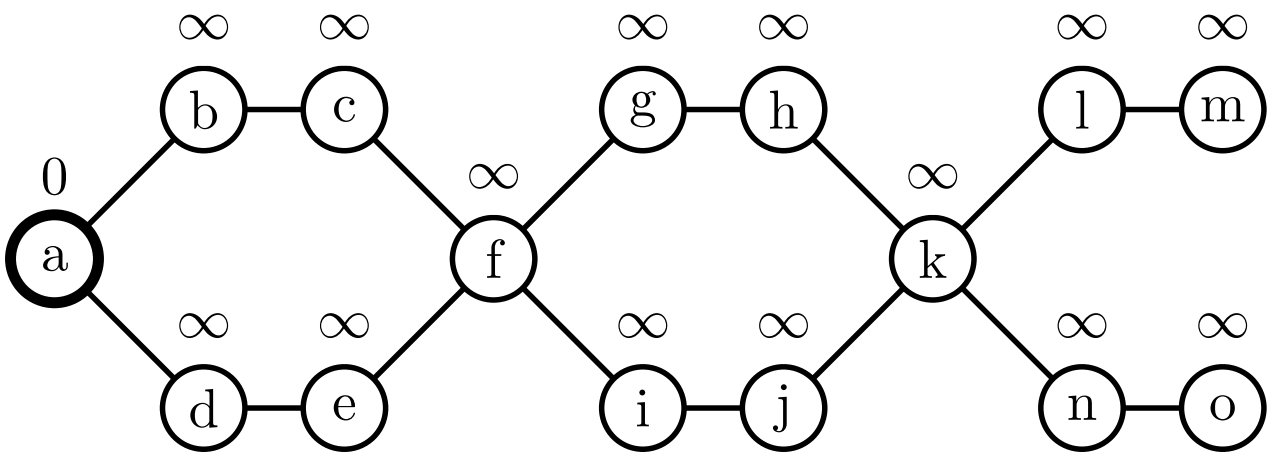
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

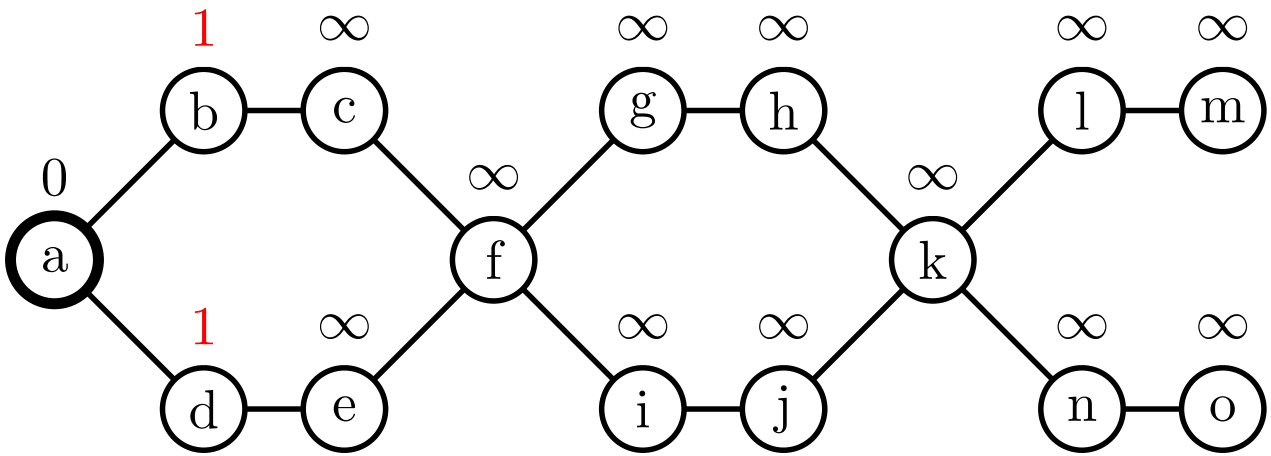
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

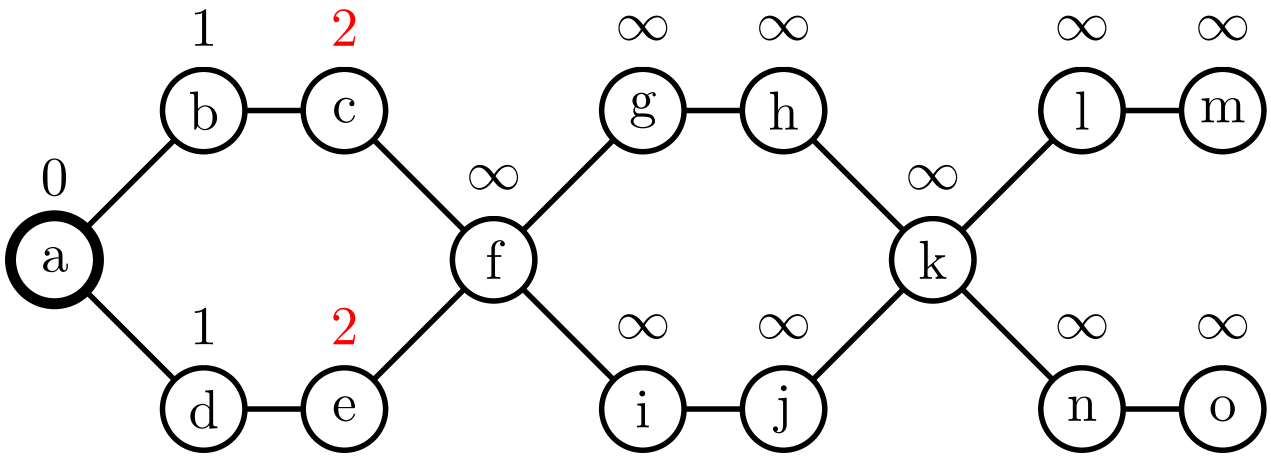
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

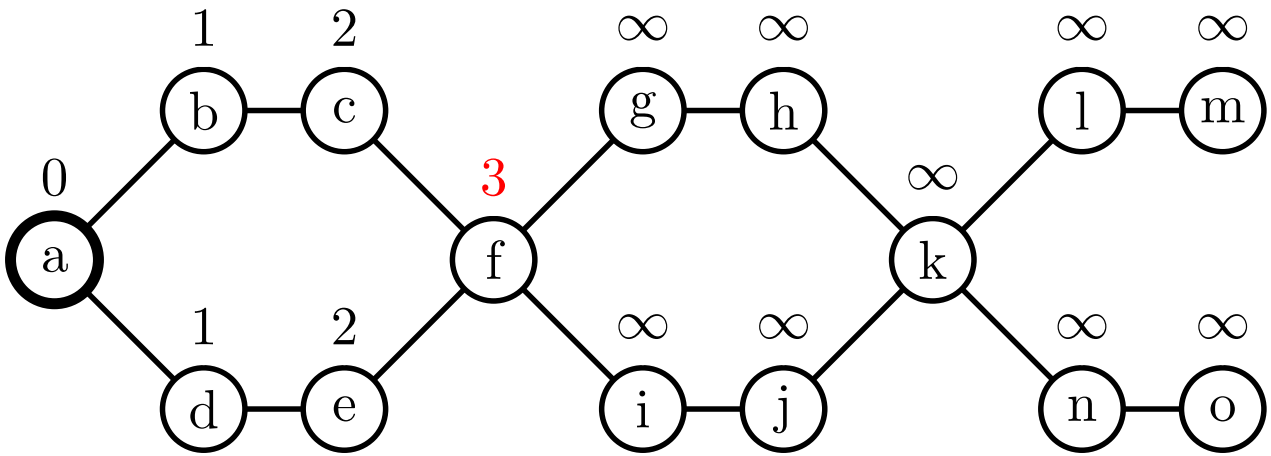
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

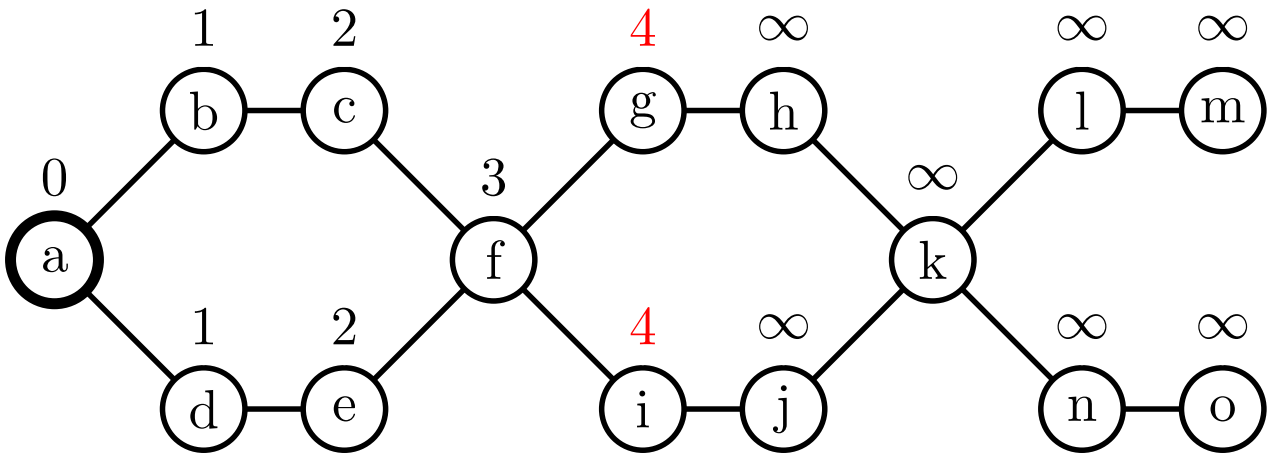
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

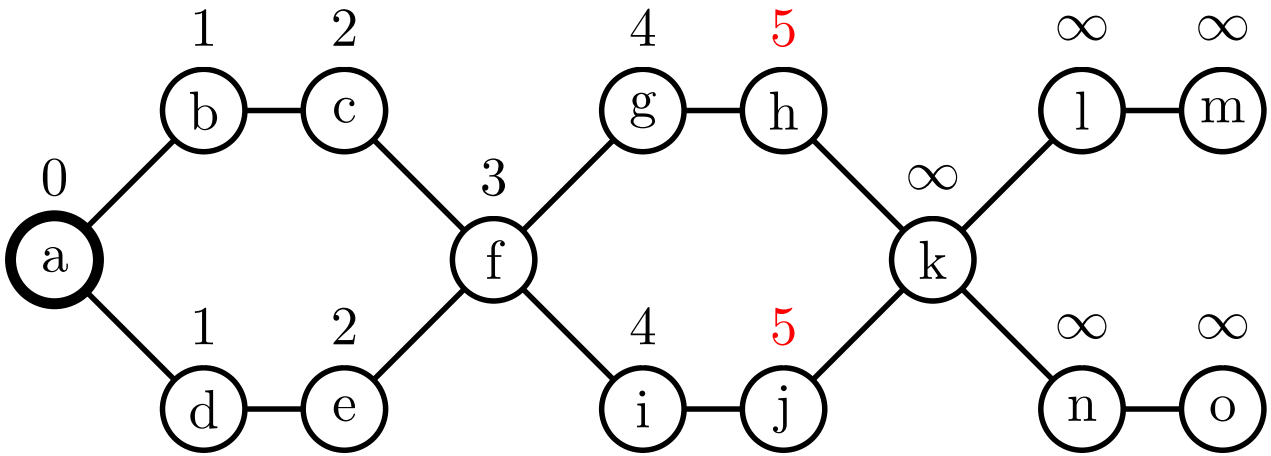
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

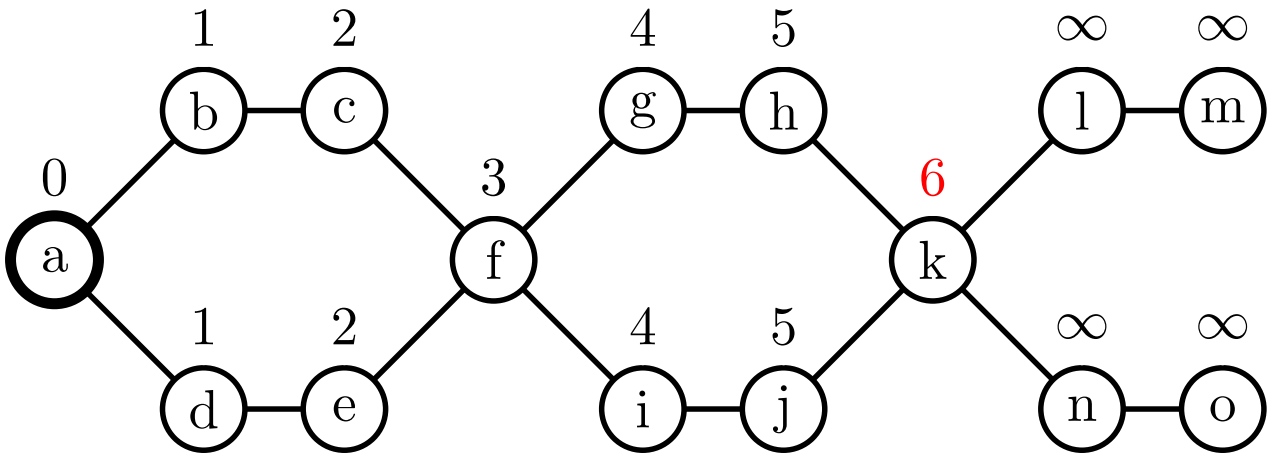
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

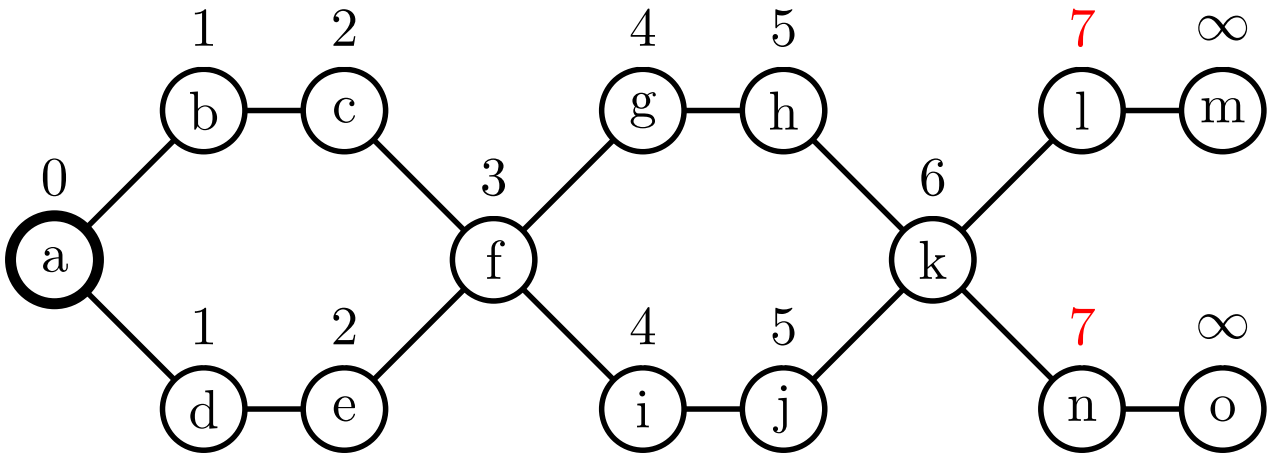
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

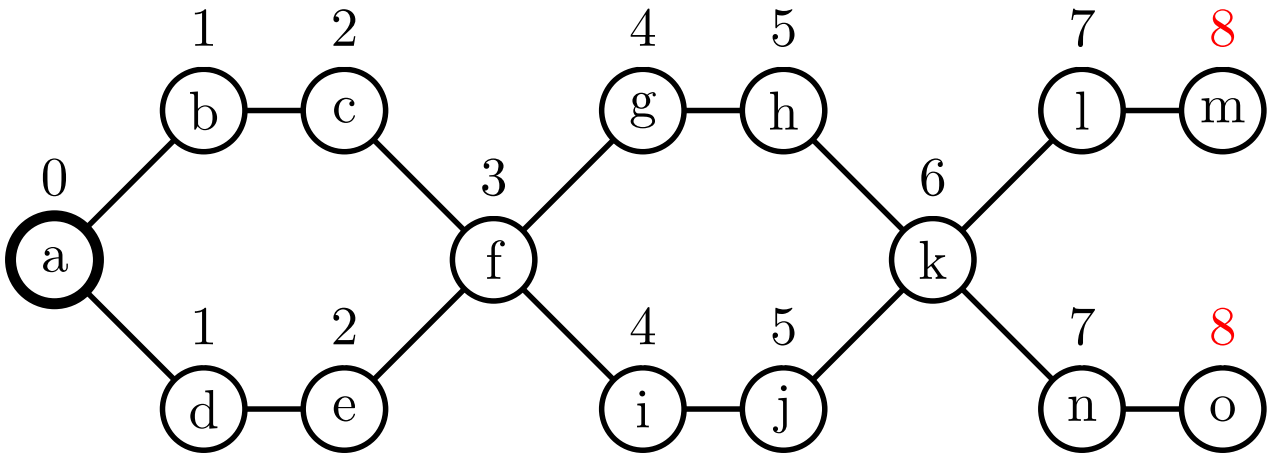
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Terminating Synchronous Algorithms

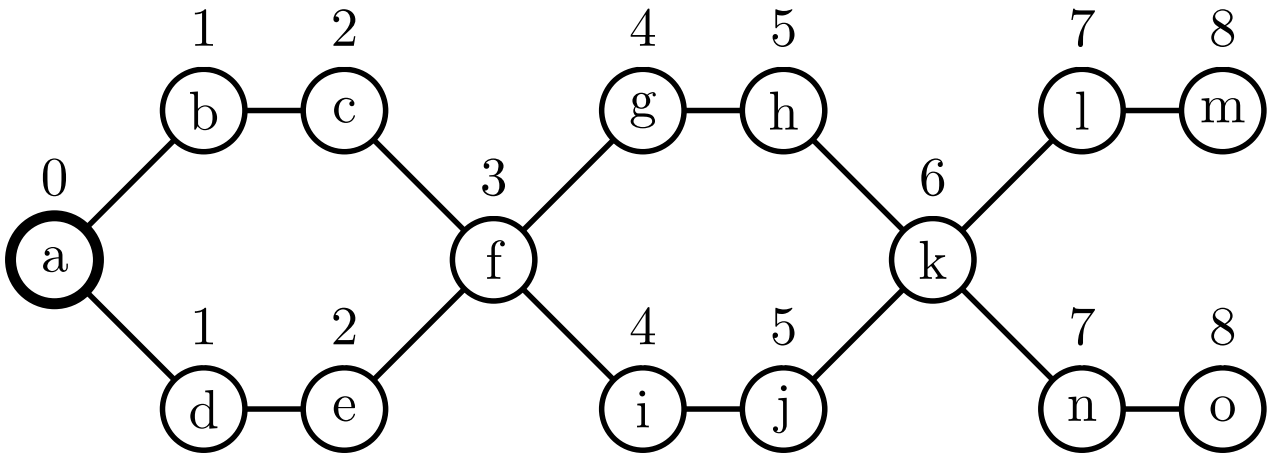
Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

At each step, **all nodes** are **synchronously** activated.

The initial configuration is **controlled**.

Starting from the **pre-defined** initial configuration, the execution eventually terminates.



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

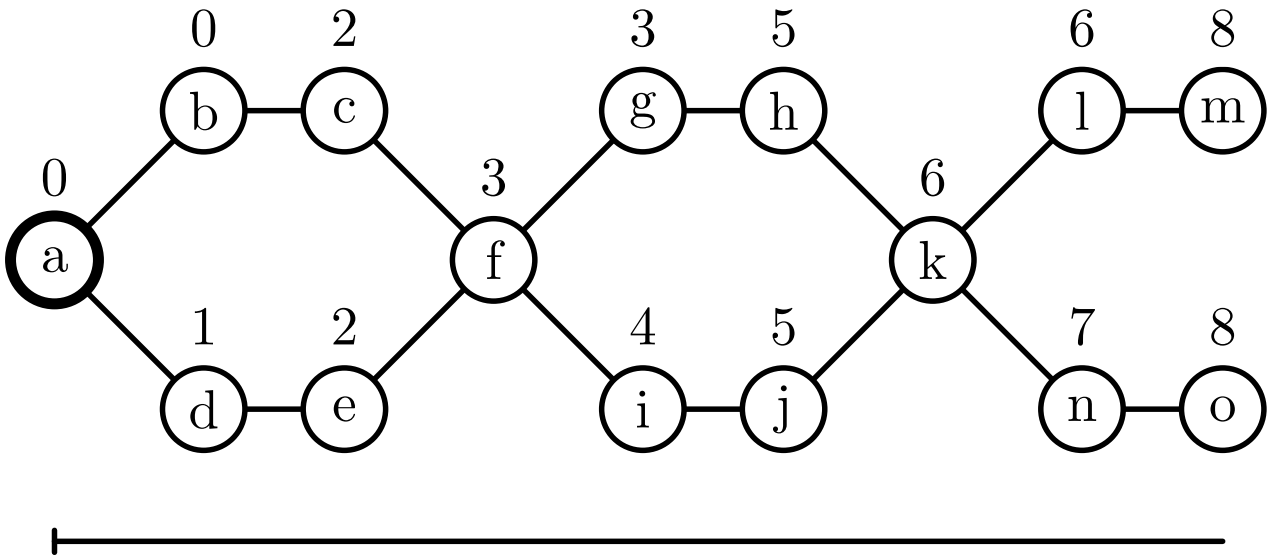
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

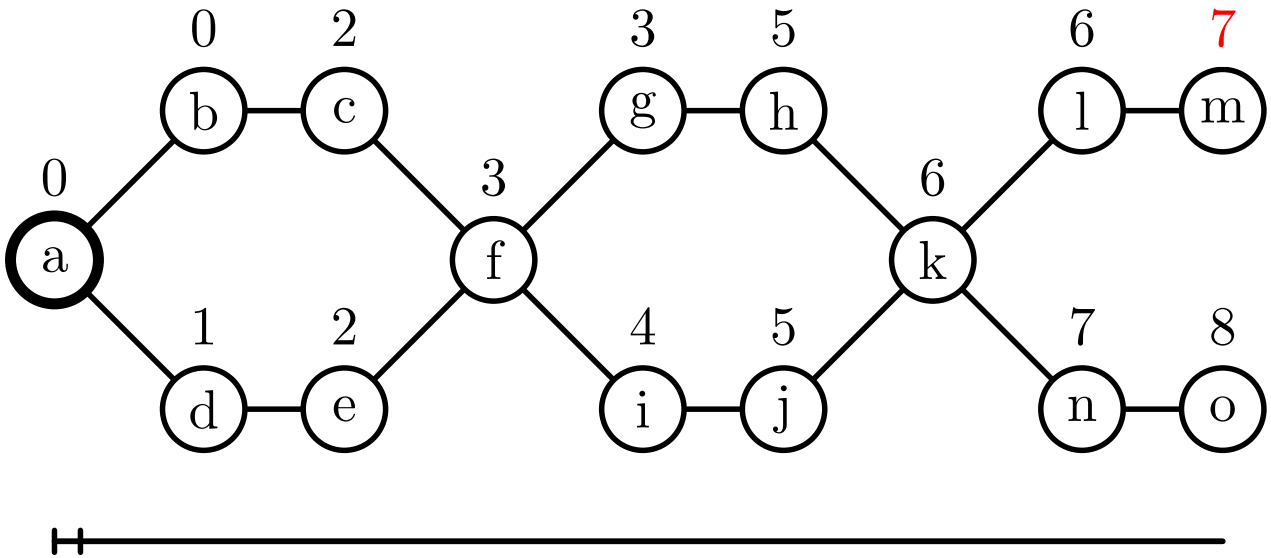
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

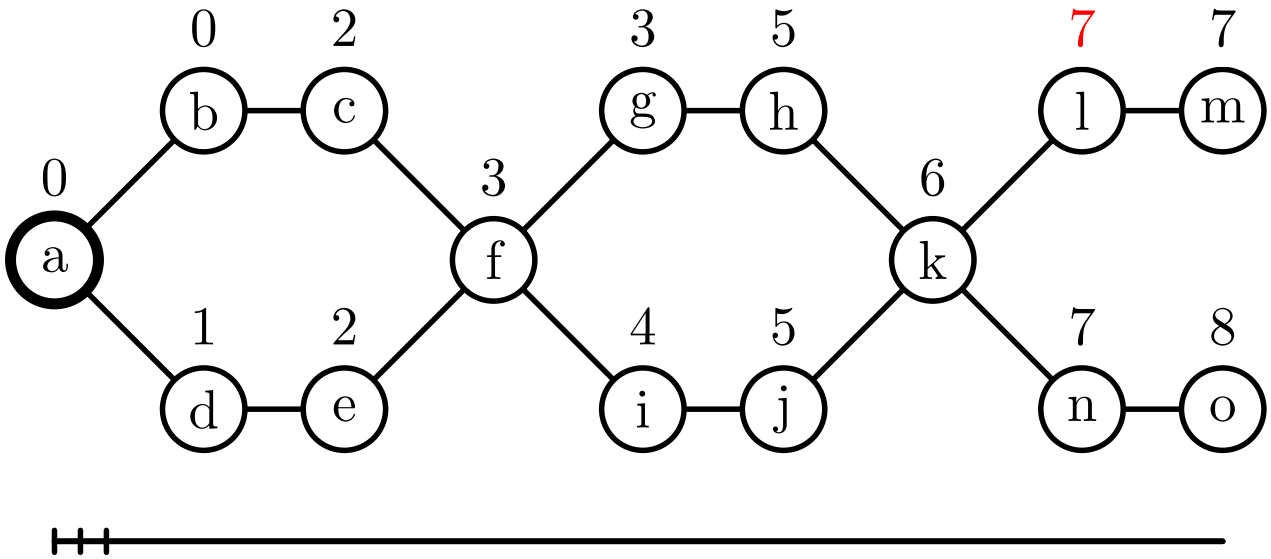
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

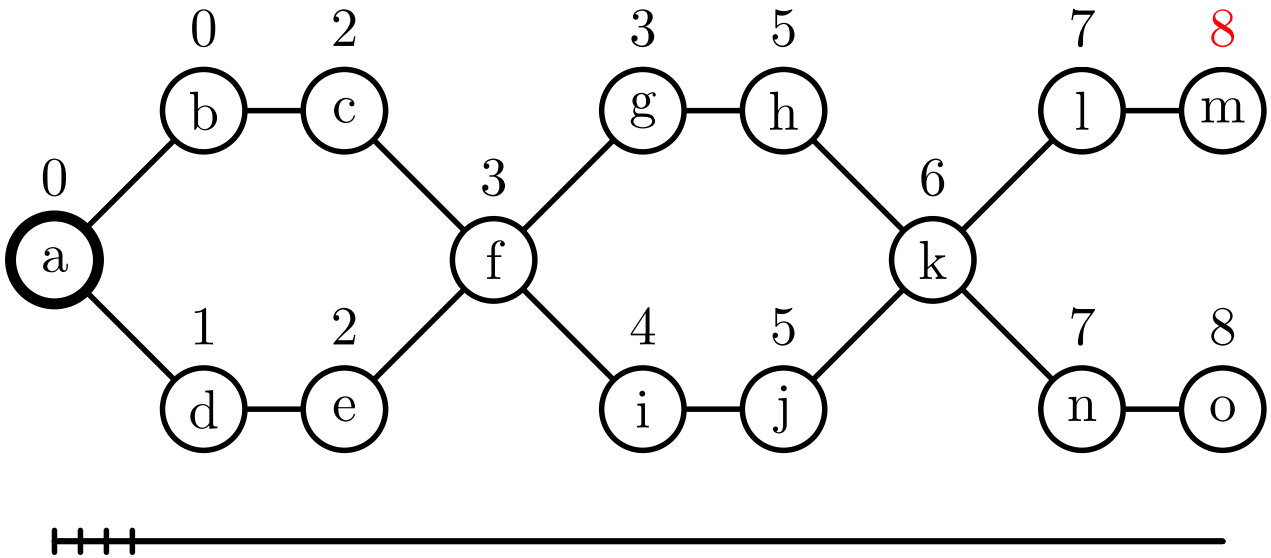
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

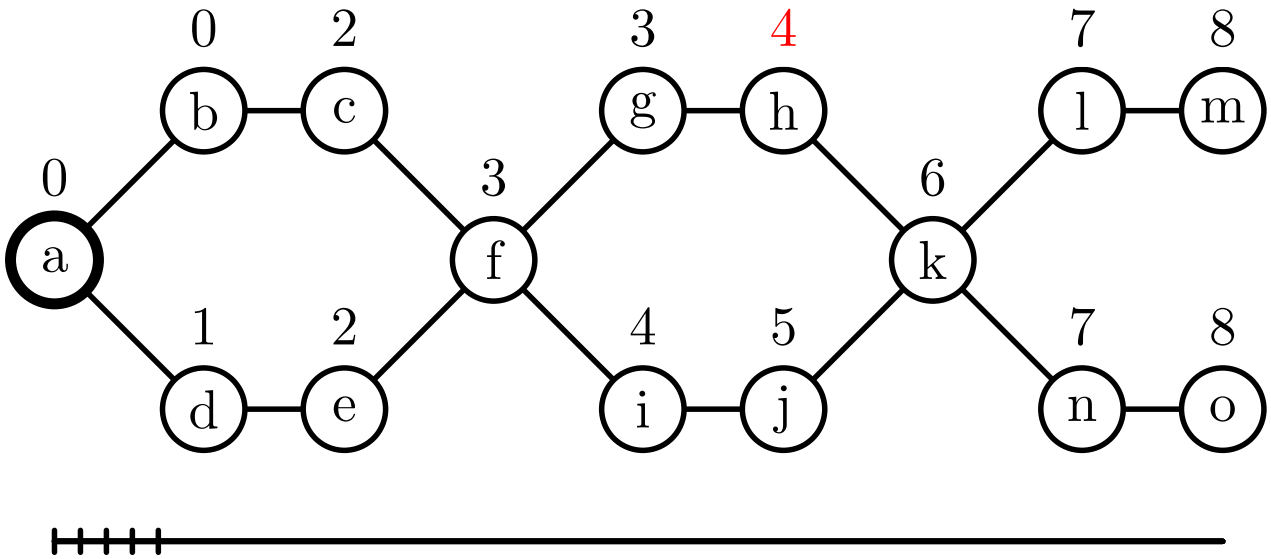
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

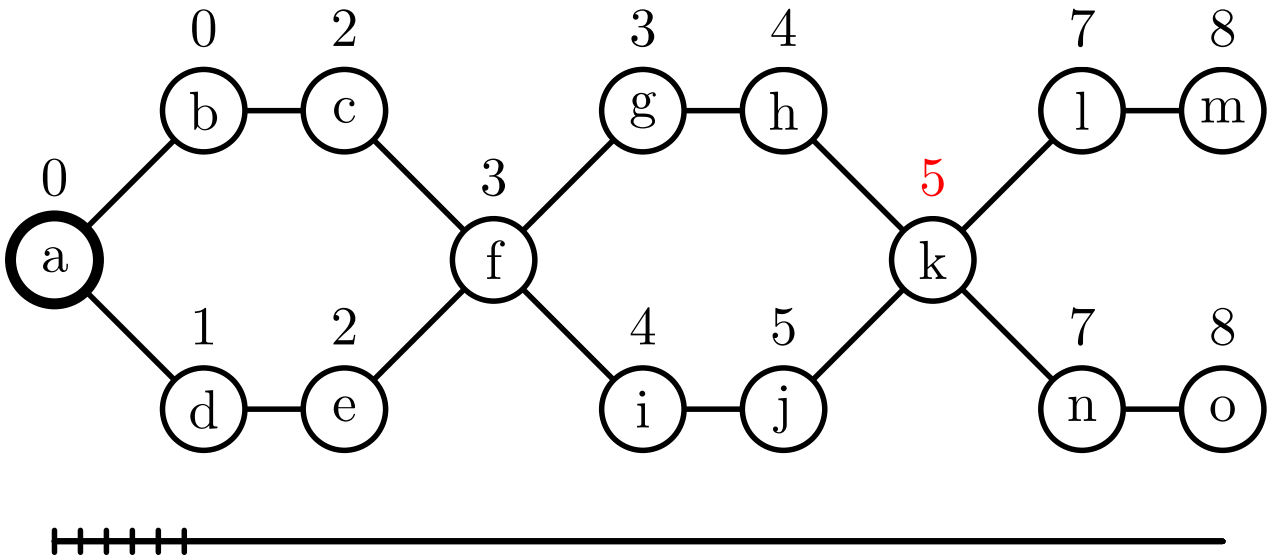
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

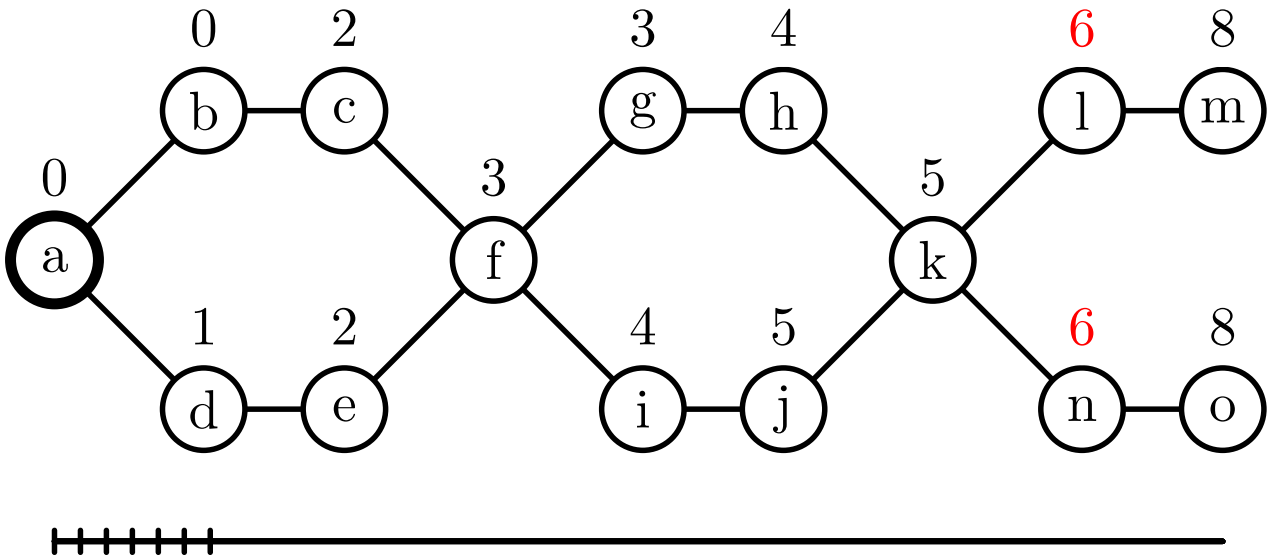
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

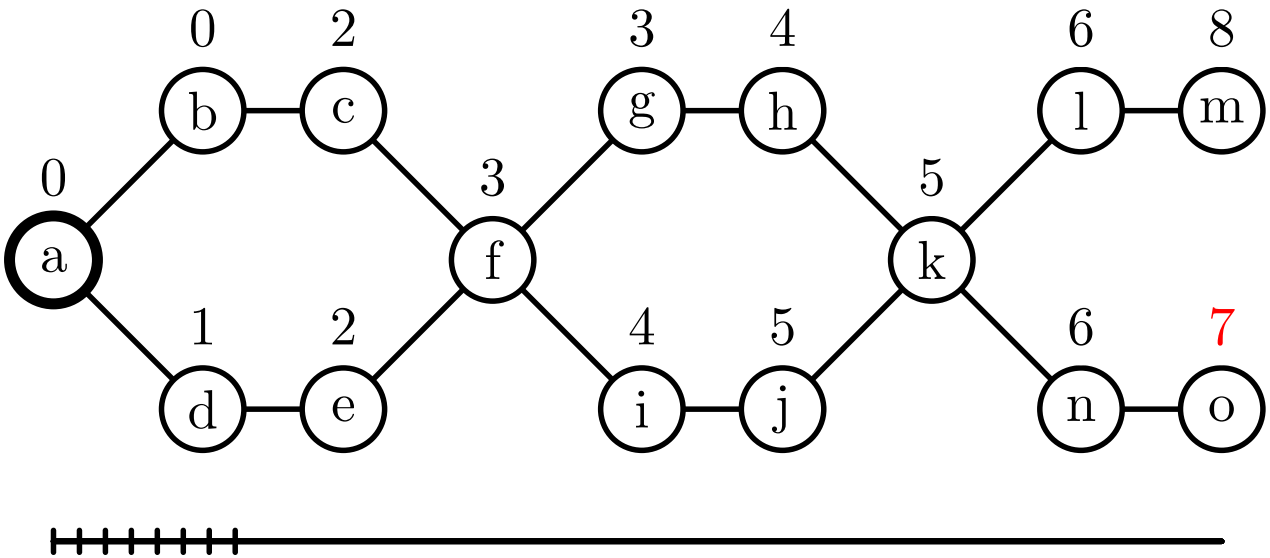
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

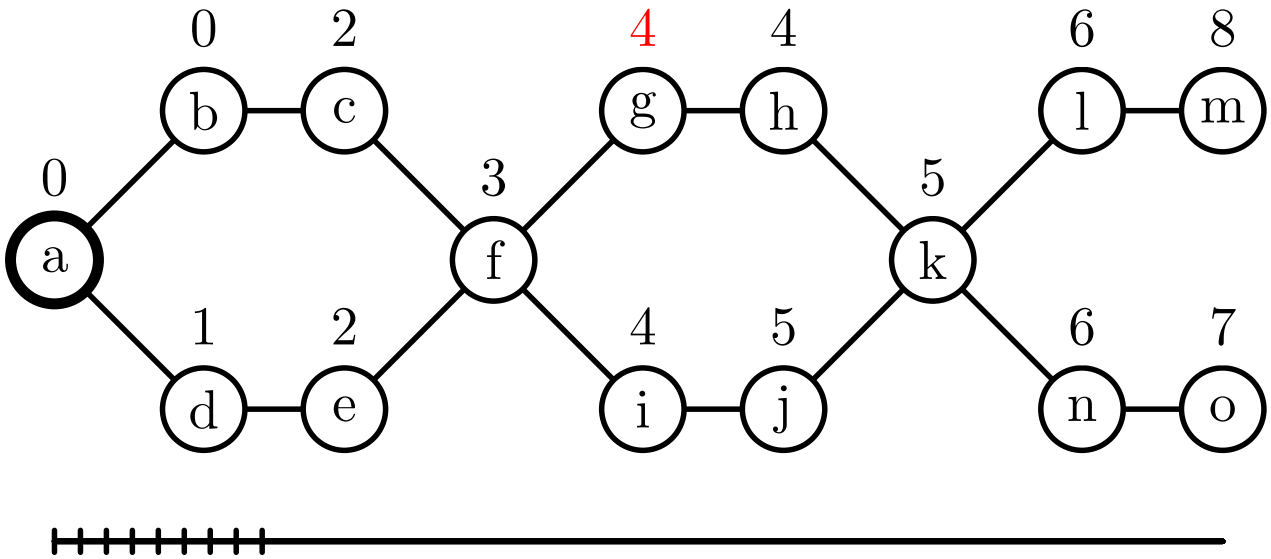
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

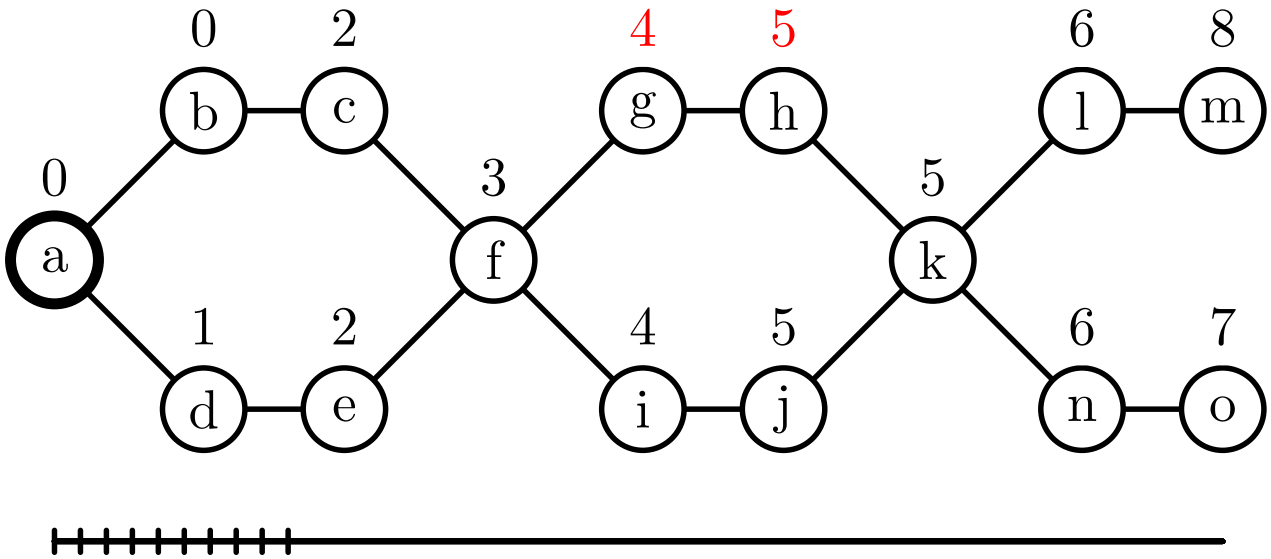
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

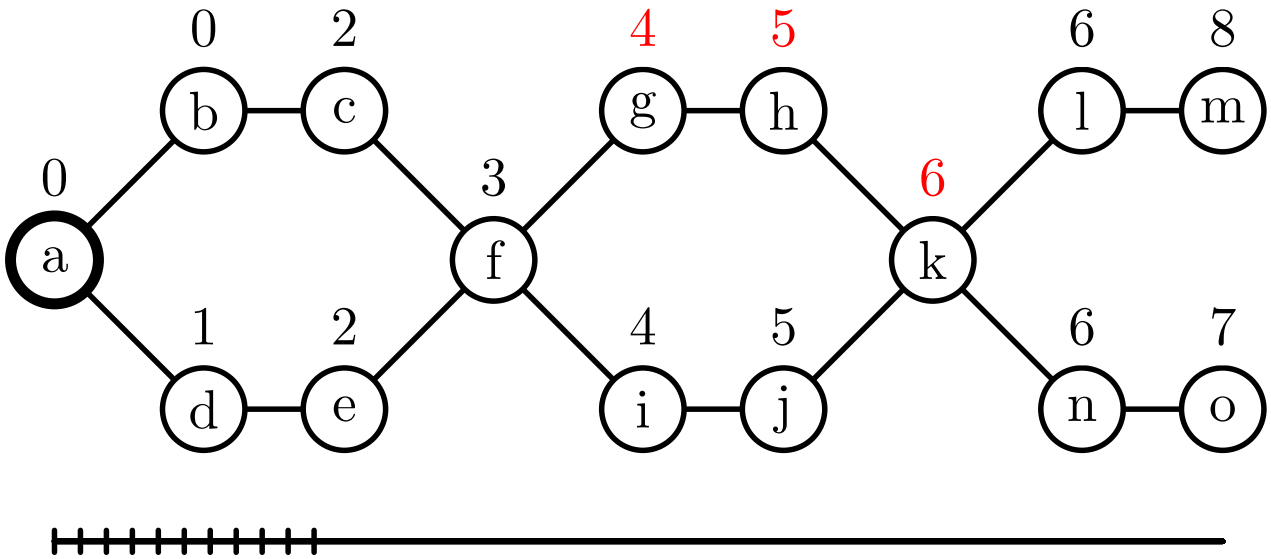
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

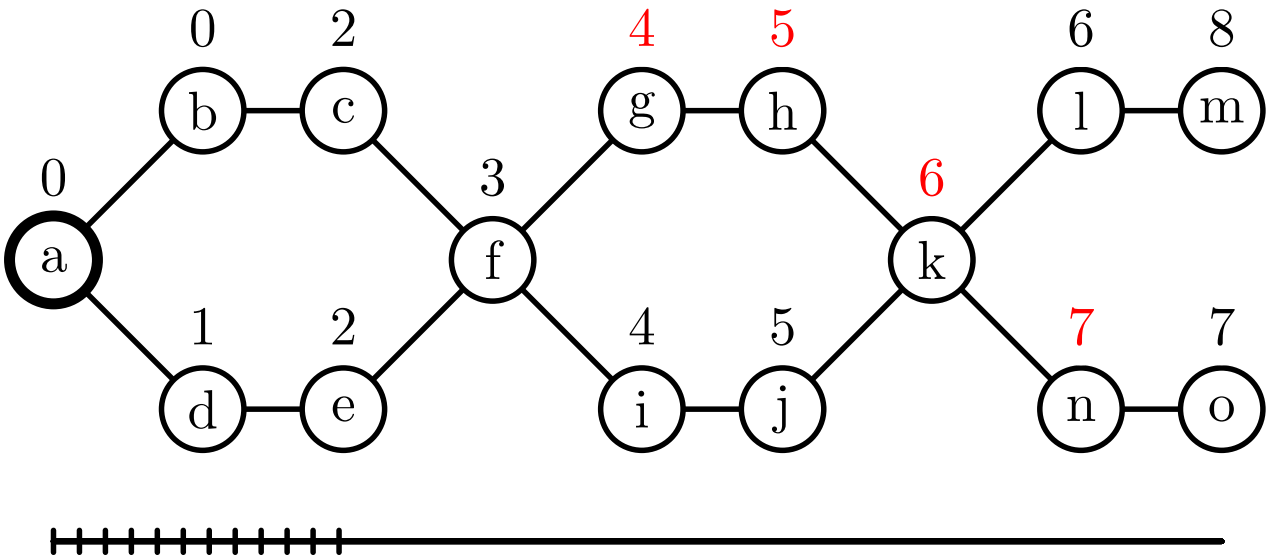
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

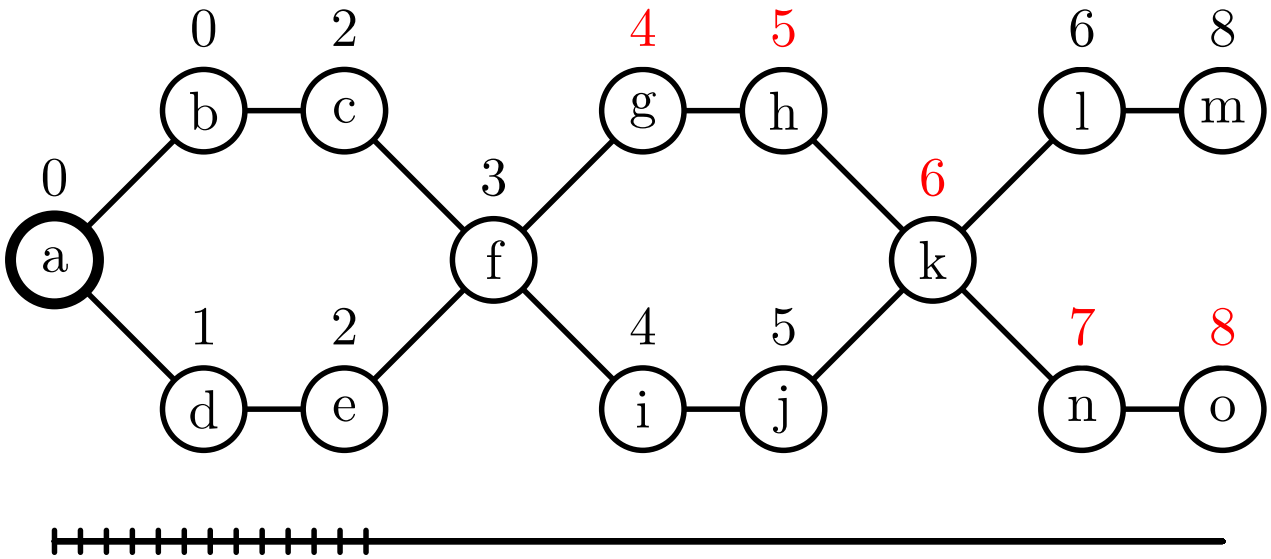
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

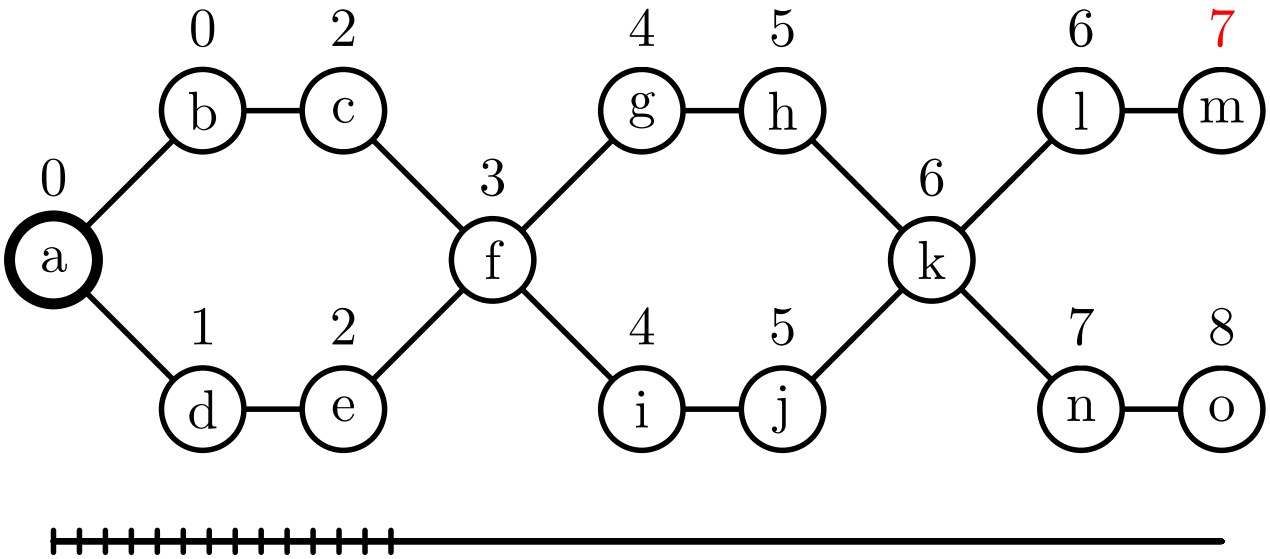
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

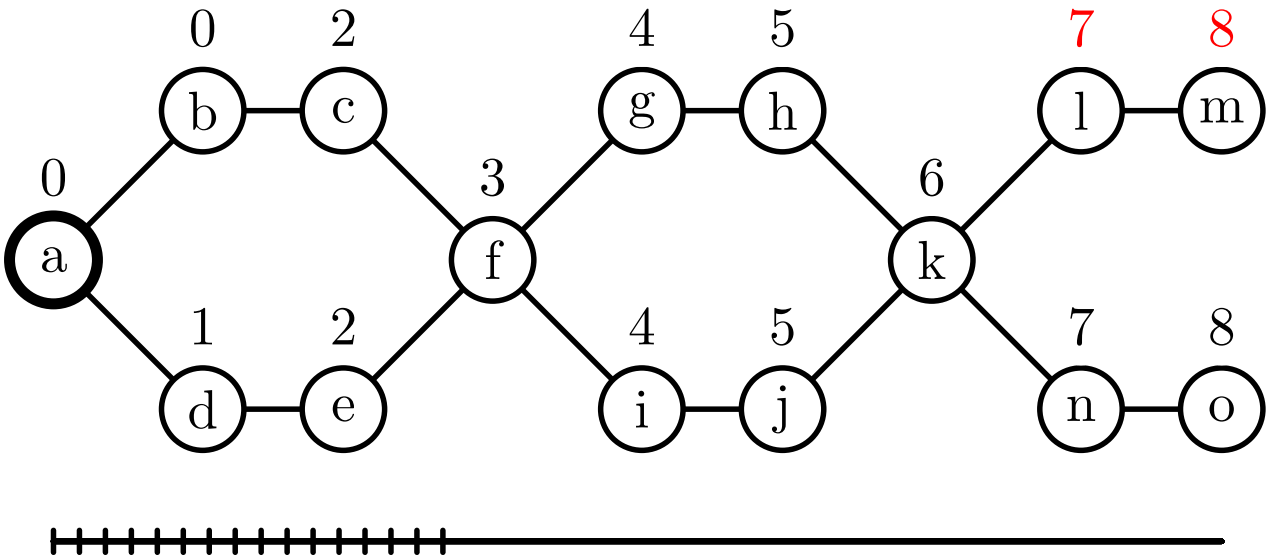
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

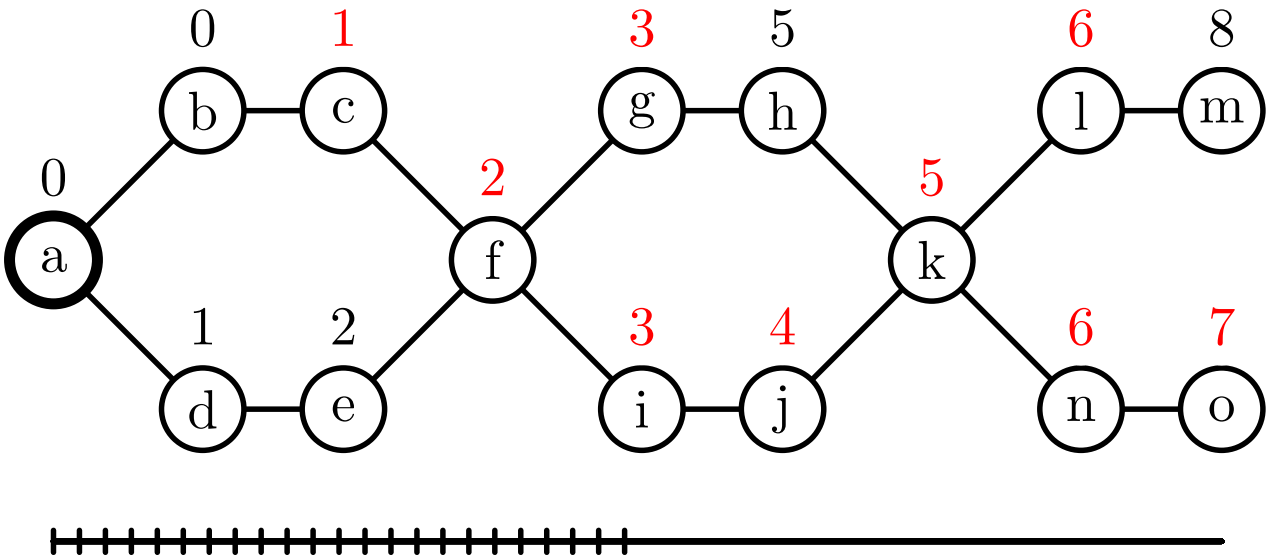
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

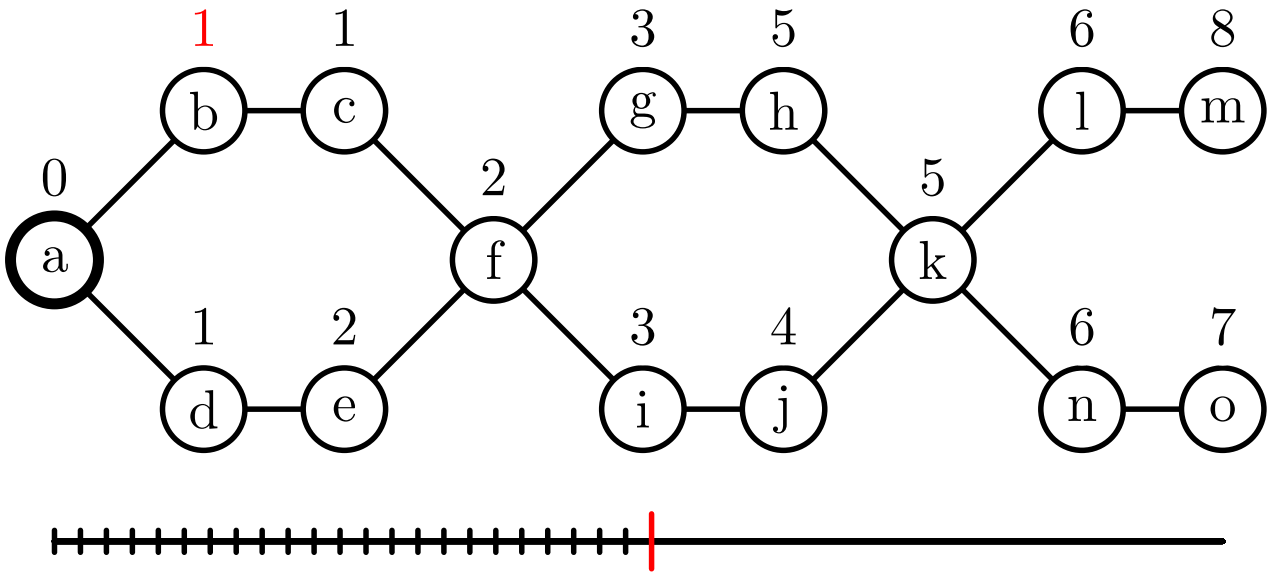
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

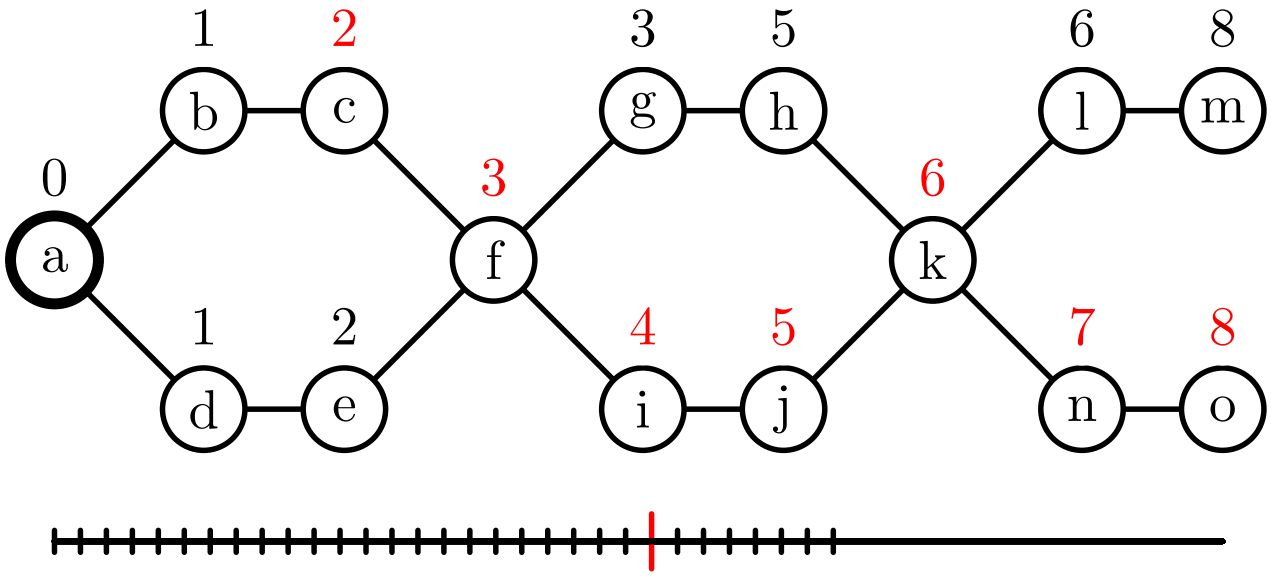
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

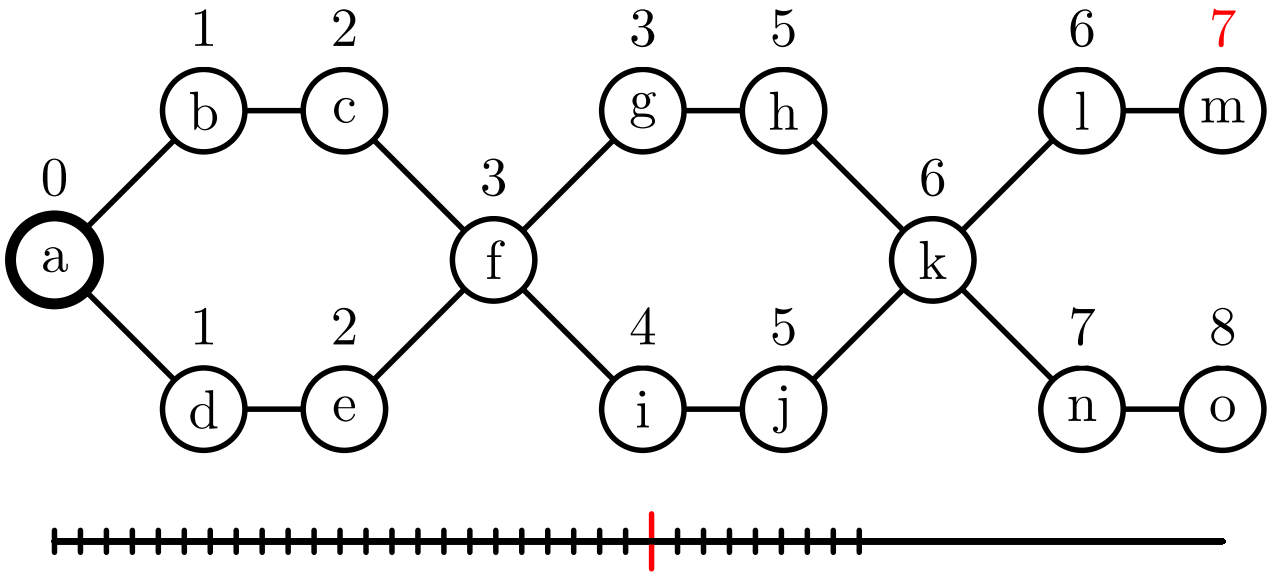
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

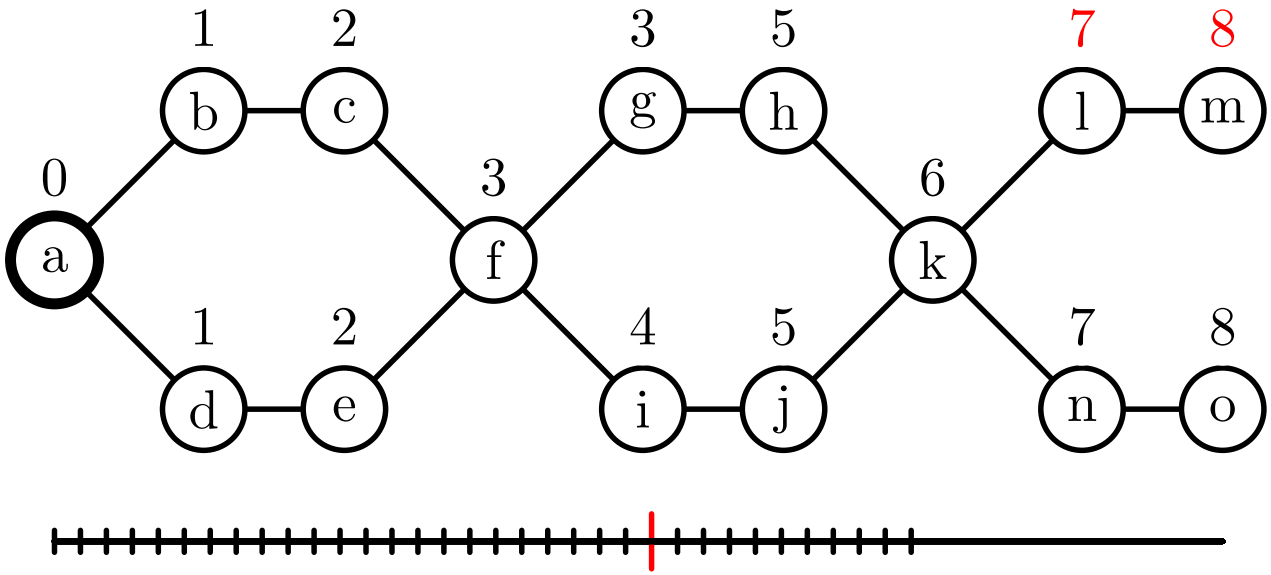
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

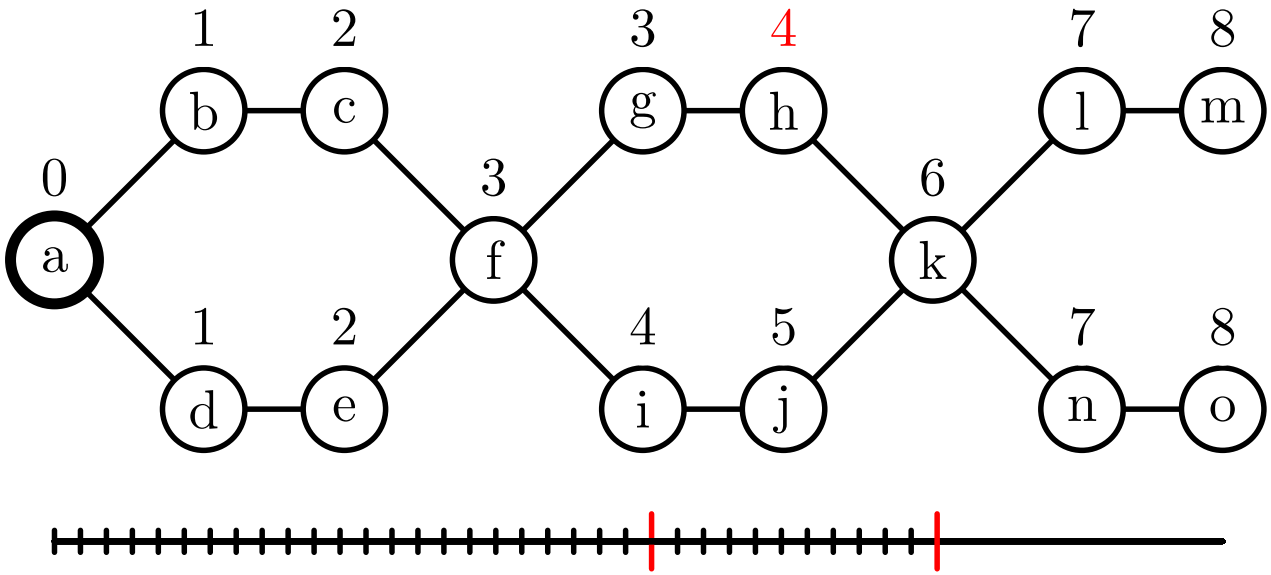
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "$\min +1$" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

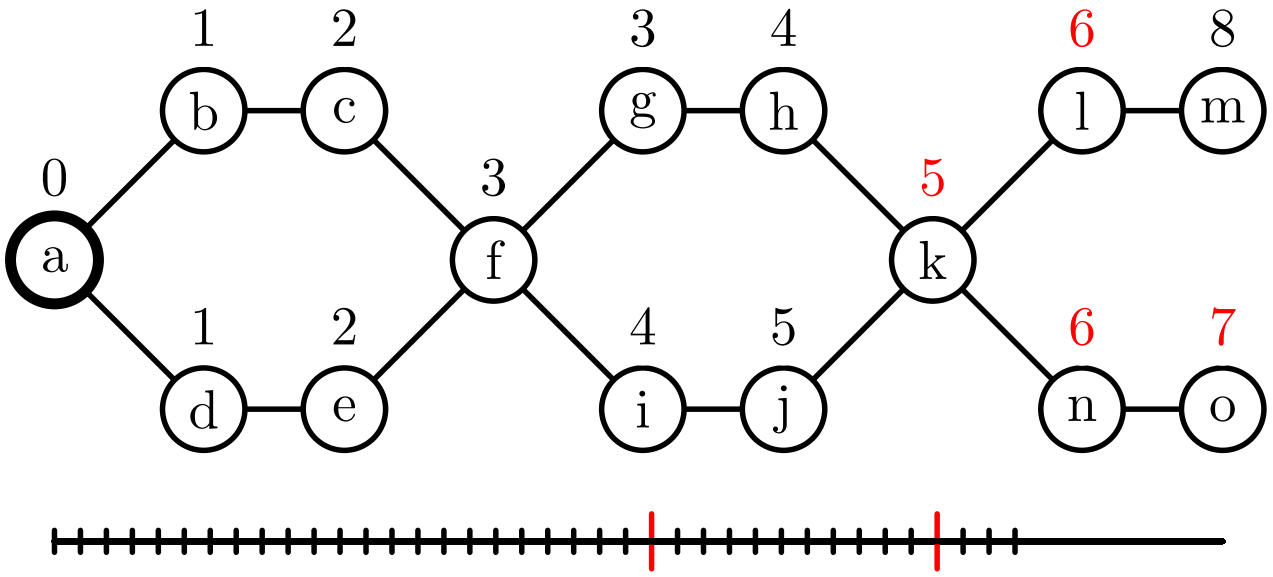
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

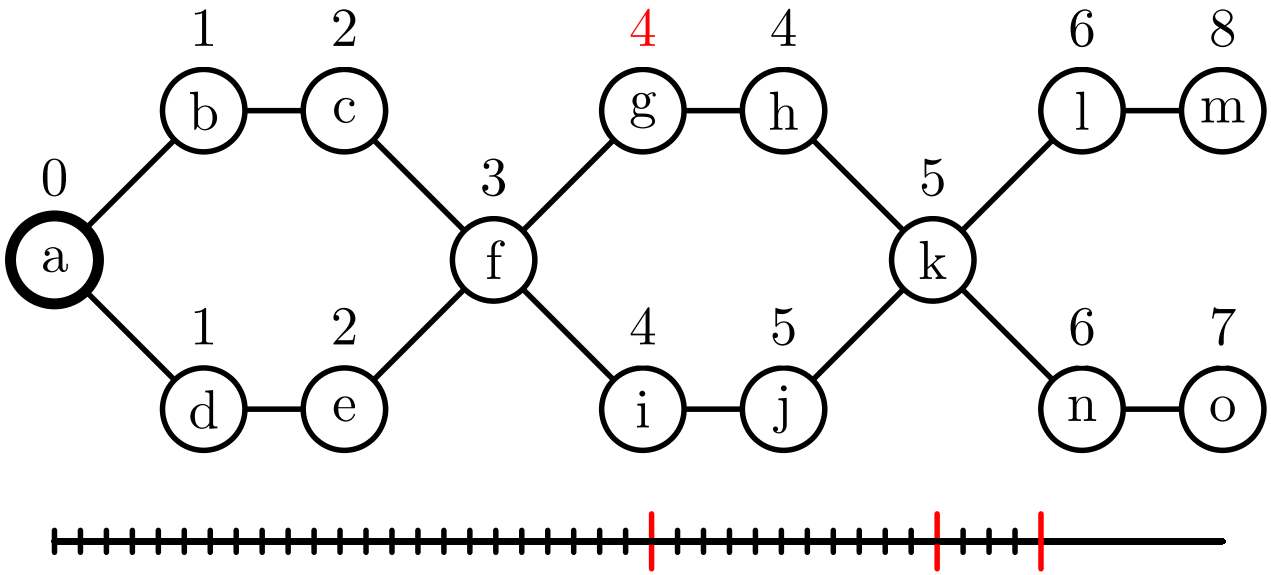
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

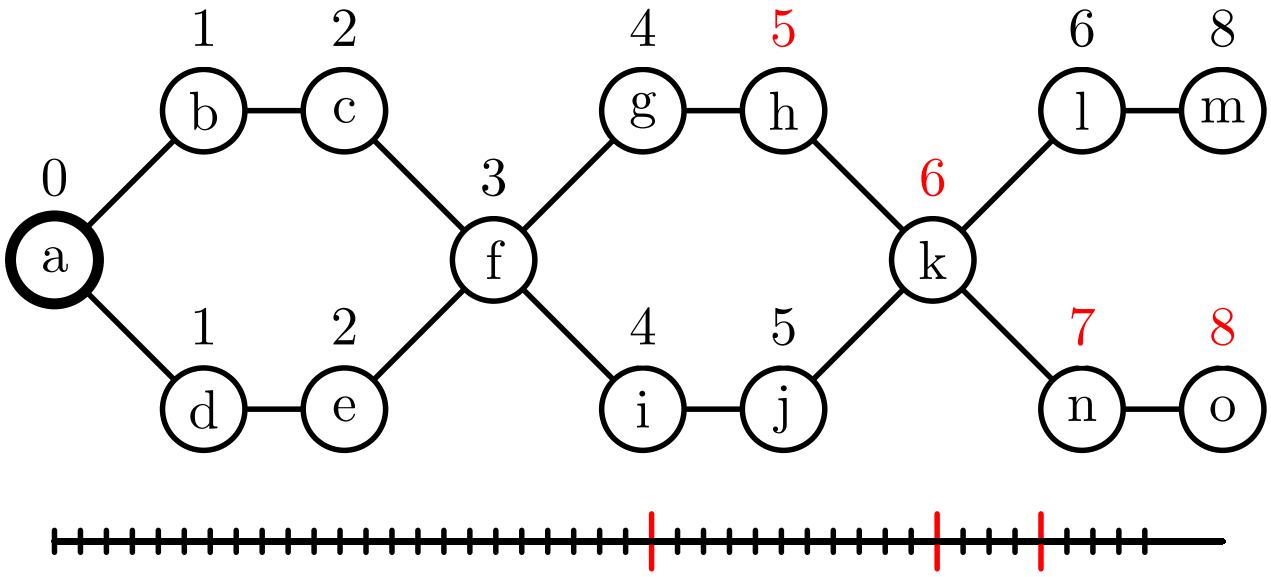
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

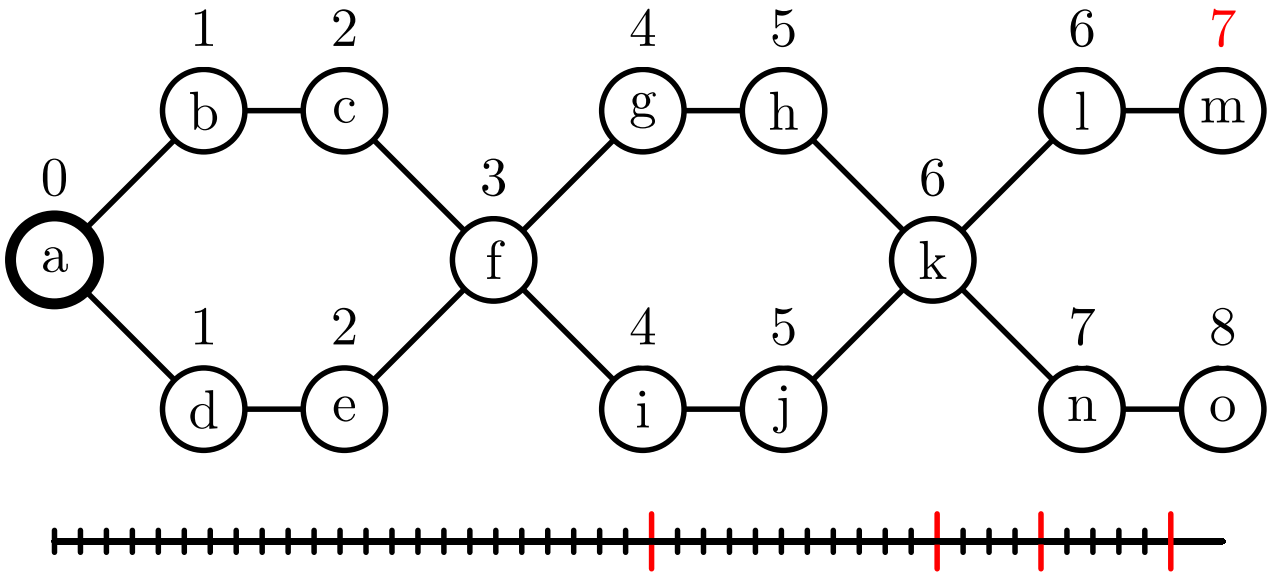
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

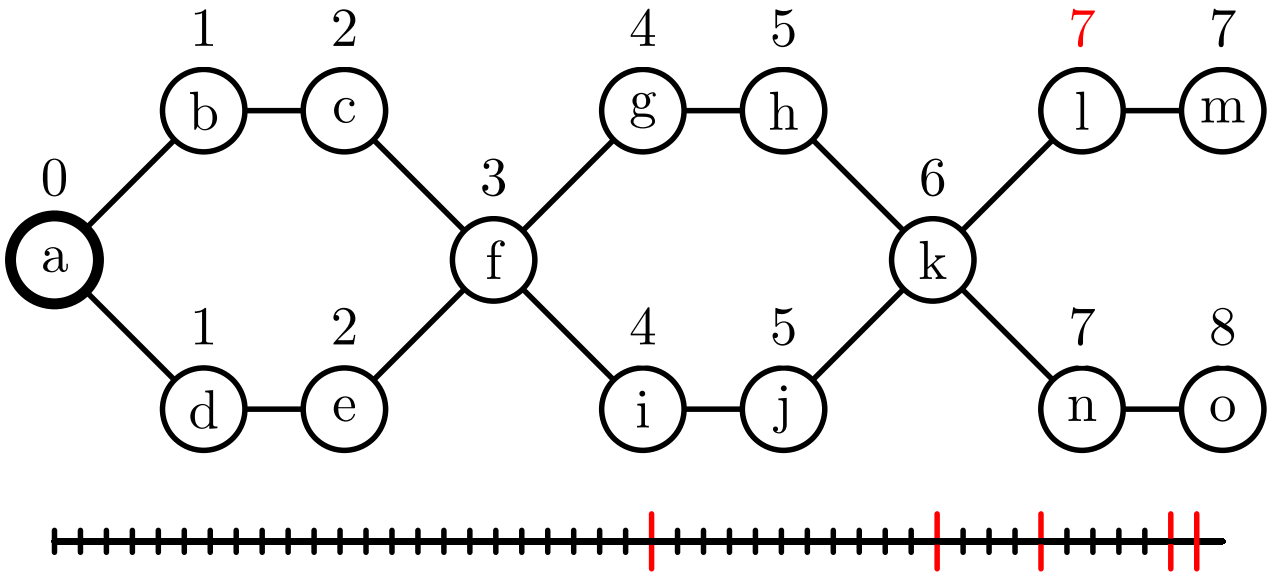
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Silent Self-Stabilizing Asynchronous Algorithms

Each node atomically

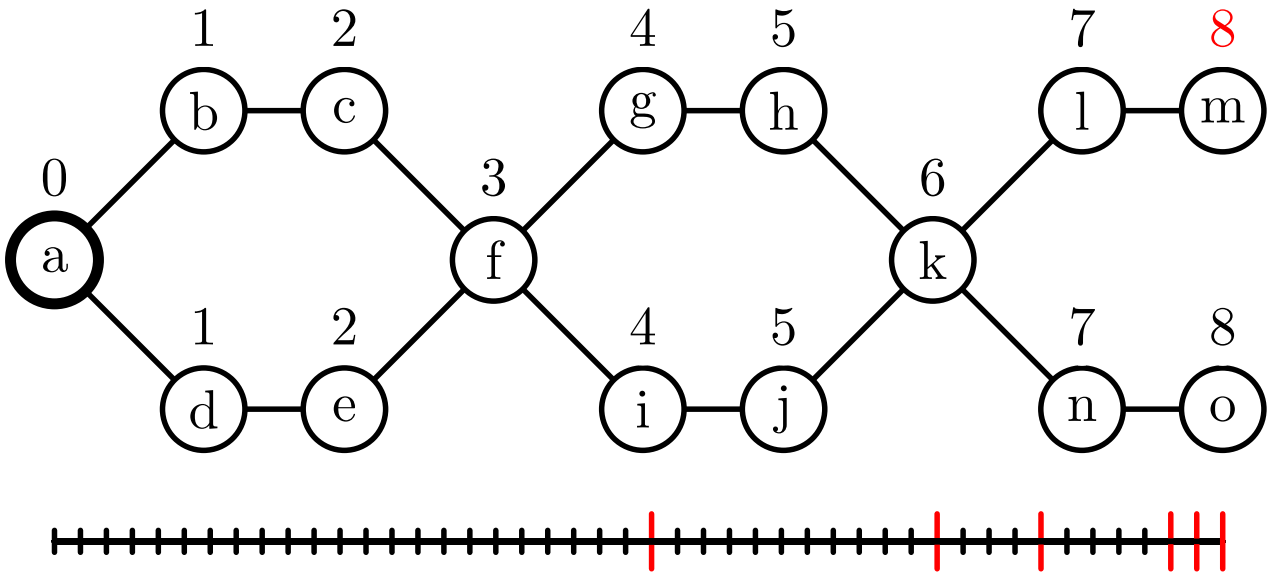
- reads its state and the state of its neighbors,
- changes its state.

Asynchronous: at each step, some nodes are activated.

(no fairness assumption: **unfair daemon**)

Self-stabilizing: the initial configuration is arbitrary.

Silent: every execution terminates + terminal $\Rightarrow$ legitimate



Example: Distance to a Root, "min +1" algorithm

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time" (according to the speed of the slowest node).

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time"
  (according to the speed of the slowest node).
  relevant parameter: D the diameter of G.
  often $O(\log n)$ in "real graphs"

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time" (according to the speed of the slowest node).

  relevant parameter: D the diameter of G.

  often $O(\log n)$ in "real graphs"

- **Move** complexity captures the "total work done".

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time" (according to the speed of the slowest node).

  relevant parameter: D the diameter of G.

  often $O(\log n)$ in "real graphs"

- **Move** complexity captures the "total work done".

  relevant parameter: n the number of nodes in G.

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time" (according to the speed of the slowest node).
  relevant parameter: D the diameter of G.
  often $O(\log n)$ in "real graphs"
- **Move** complexity captures the "total work done".
  relevant parameter: n the number of nodes in G.

Definition (Fully Polynomial)

[Cournier, Rovedakis, Villain. Inf. and Comp. 2019]
*A self-stabilizing algorithm is **fully polynomial** if*

- *round complexity:* $Poly(D)$,
- *move complexity:* $Poly(n)$.

Self-Stabilizing Asynchronous Algorithms

Round and Move Complexity

- **Round** complexity captures the "execution time" (according to the speed of the slowest node).
  relevant parameter: D the diameter of G.
  often $O(\log n)$ in "real graphs"
- **Move** complexity captures the "total work done".
  relevant parameter: n the number of nodes in G.

Definition (Fully Polynomial)

[Cournier, Rovedakis, Villain. Inf. and Comp. 2019]
*A self-stabilizing algorithm is **fully polynomial** if*

- *round complexity:* $Poly(D)$,
- *move complexity:* $Poly(n)$.

Note: A node may "**starve**" in asynchronous **unfair** executions.

Our Contribution 1

- Transformer
  - 1991. Awerbuch et al. Proceedings FOCS. $O(D)$ rounds, $Exp(n)$ moves

Our Contribution 1

- Transformer
  - 1991. Awerbuch et al. Proceedings FOCS.

$O(D)$ rounds, $O(n^3)$ moves

$O(D)$ rounds, $Exp(n)$ moves

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves

- Leader election
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves
- Leader election — **$O(D)$ rounds, $O(n^3)$ moves**
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves
- Leader election — **$O(D)$ rounds, $O(n^3)$ moves**
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.
- BFS
  - 1993: Dolev et al. Acta Inform. — $O(D)$ rounds, $Exp(n)$ moves
  - 1997: Johnen. Proceedings PODC. — $D^2 \cdot n$ rounds
  - 2009: Cournier et al. ACM Trans. Auton. Adapt. Syst. — $D^2 + n$ rounds, $O(\Delta \cdot n^3)$ moves
  - 2019: Cournier et al. Inf. Comput. — $O(D^2)$ rounds, $O(n^6)$ moves

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves
- Leader election — **$O(D)$ rounds, $O(n^3)$ moves**
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.
- BFS — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1993: Dolev et al. Acta Inform. — $O(D)$ rounds, $Exp(n)$ moves
  - 1997: Johnen. Proceedings PODC. — $D^2 \cdot n$ rounds
  - 2009: Cournier et al. ACM Trans. Auton. Adapt. Syst. — $D^2 + n$ rounds, $O(\Delta \cdot n^3)$ moves
  - 2019: Cournier et al. Inf. Comput. — $O(D^2)$ rounds, $O(n^6)$ moves

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves
- Leader election — **$O(D)$ rounds, $O(n^3)$ moves**
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.
- BFS — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1993: Dolev et al. Acta Inform. — $O(D)$ rounds, $Exp(n)$ moves
  - 1997: Johnen. Proceedings PODC. — $D^2 \cdot n$ rounds
  - 2009: Cournier et al. ACM Trans. Auton. Adapt. Syst. — $D^2 + n$ rounds, $O(\Delta \cdot n^3)$ moves
  - 2019: Cournier et al. Inf. Comput. — $O(D^2)$ rounds, $O(n^6)$ moves
- Cycle coloration in 3 colors
  - 1986: Cole et al. Proceedings FOCS — $O(log^* n)$ steps, LOCAL model
  - 2009: Lenzen et al. Proceedings SSS. — $O(log^* n)$ rounds, $> Poly(n)$ moves
  - 2018: Baremboim et al. Proceedings PODC. — $O(log^* n)$ rounds, synchronous

Our Contribution 1

- Transformer — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1991. Awerbuch et al. Proceedings FOCS. — $O(D)$ rounds, $Exp(n)$ moves
- Leader election — **$O(D)$ rounds, $O(n^3)$ moves**
  - 2011: Datta et al. J. Parallel Distrib. Comput. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2011: Datta et al. TCS. — $\Theta(n)$ rounds, $Exp(n)$ moves
  - 2017: Altisen et al. Inf. Comput. — $\Theta(n)$ rounds, $Poly(n)$ moves.
- BFS — **$O(D)$ rounds, $O(n^3)$ moves**
  - 1993: Dolev et al. Acta Inform. — $O(D)$ rounds, $Exp(n)$ moves
  - 1997: Johnen. Proceedings PODC. — $D^2 \cdot n$ rounds
  - 2009: Cournier et al. ACM Trans. Auton. Adapt. Syst. — $D^2 + n$ rounds, $O(\Delta \cdot n^3)$ moves
  - 2019: Cournier et al. Inf. Comput. — $O(D^2)$ rounds, $O(n^6)$ moves
- Cycle coloration in 3 colors — **$O(log^*n)$ rounds, $O(n^2 log^*n)$ moves**
  - 1986: Cole et al. Proceedings FOCS — $O(log^*n)$ steps, LOCAL model
  - 2009: Lenzen et al. Proceedings SSS. — $O(log^*n)$ rounds, $> Poly(n)$ moves
  - 2018: Baremboim et al. Proceedings PODC. — $O(log^*n)$ rounds, synchronous

Contribution

*A **tranformer**:*

Input: *a synchronous terminating algorithm A.(time and space complexities: T, S)*

a bound B such that $T \leq B \leq +\infty$

Output: *an asynchronous algorithm which simulates A.*

- *silent, self-stabilizing* *(even in the unfair case)*

Contribution

*A **tranformer**:*

Input: a synchronous terminating algorithm A.(time and space complexities: T, S)

a bound B such that $T \leq B \leq +\infty$

Output: an asynchronous algorithm which simulates A.

- *silent, self-stabilizing* *(even in the unfair case)*

	***Round** complexity*	***Move** complexity*
Error recovery	$O(\mathbf{min}(D, B))$	$O(\mathbf{min}(n^3, n^2B))$
***Greedy** mode*	$O(\boldsymbol{B})$	$O(\min(n^3 + n\boldsymbol{B}, n^2\boldsymbol{B}))$
***Lazy** mode*	$O(D + \boldsymbol{T})$	$O(\min(n^3 + n\boldsymbol{T}, n^2B))$
Space complexity		$S \cdot B$

Contribution

*A **tranformer**:*

Input: a synchronous terminating algorithm A.(time and space complexities: T, S)

a bound B such that $T \leq B \leq +\infty$

Output: an asynchronous algorithm which simulates A.

- *silent, self-stabilizing* *(even in the unfair case)*

	***Round** complexity*	***Move** complexity*
Error recovery	$O(\mathbf{min}(D, B))$	$O(\mathbf{min}(n^3, n^2B))$
***Greedy** mode*	$O(\boldsymbol{B})$	$O(\min(n^3 + n\boldsymbol{B}, n^2\boldsymbol{B}))$
***Lazy** mode*	$O(D + \boldsymbol{T})$	$O(\min(n^3 + n\boldsymbol{T}, n^2B))$
Space complexity		$S \cdot B$

Powerful tool to design efficient asynchronous self-stabilizing algorithms.

Overview of the algorithm

Ideas:

- storage of the whole synchronous execution;

Overview of the algorithm

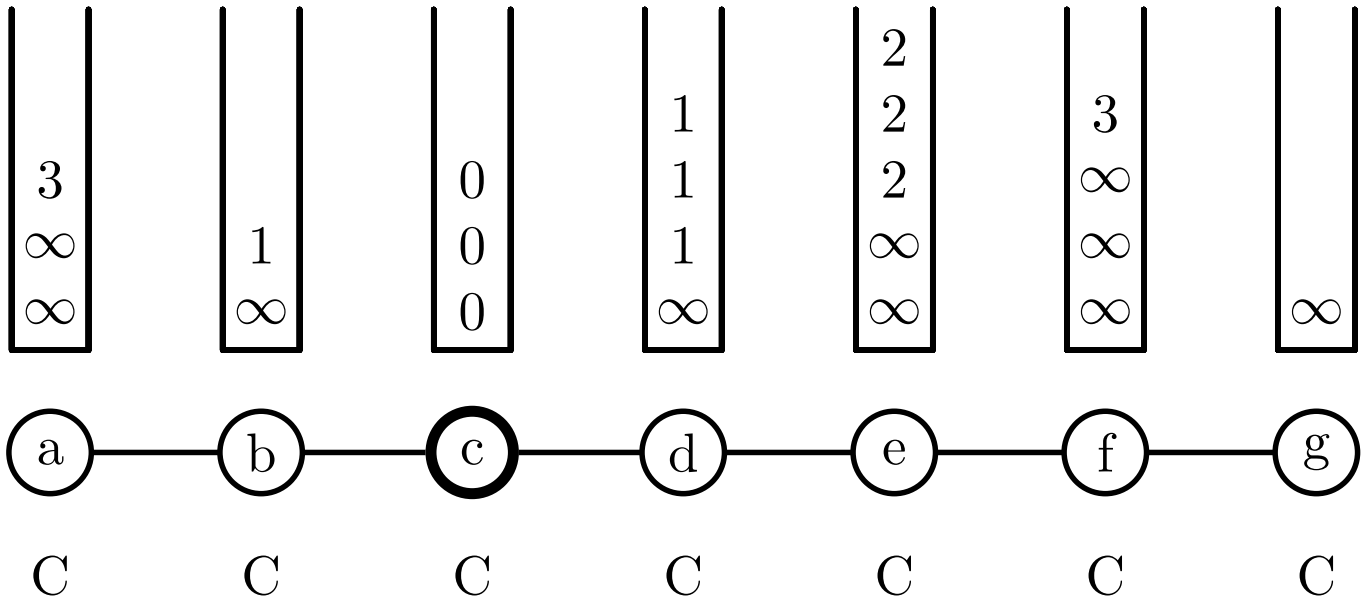
Ideas:

- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.

Overview of the algorithm

Ideas:

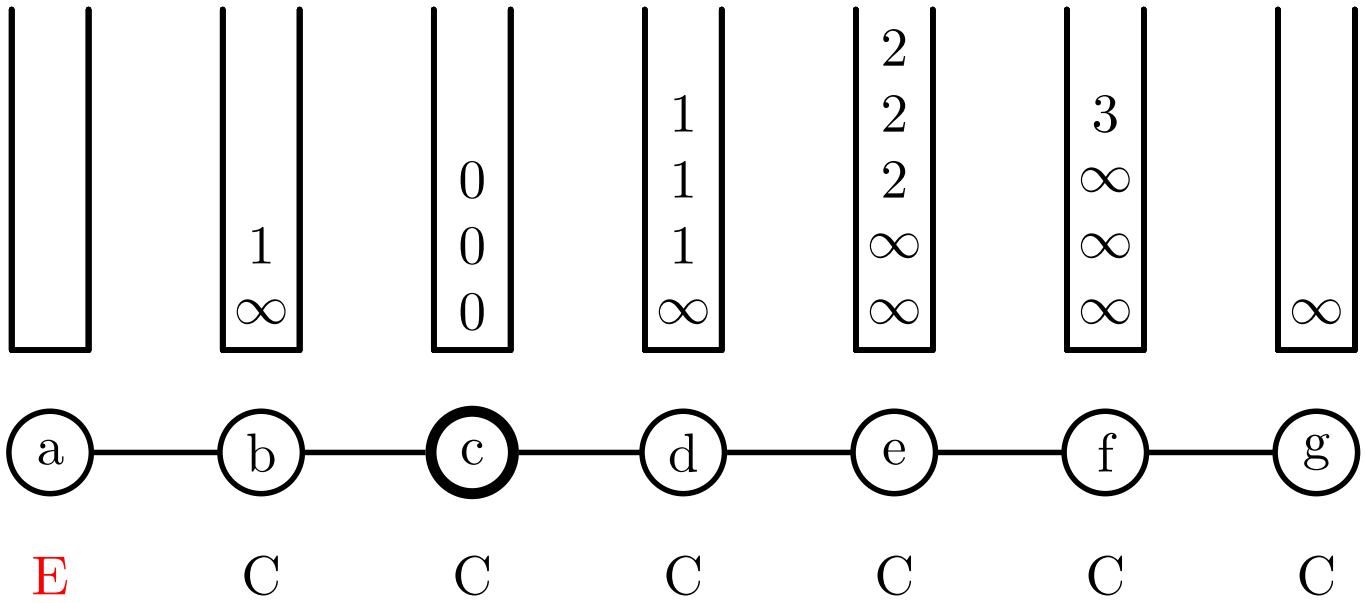
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

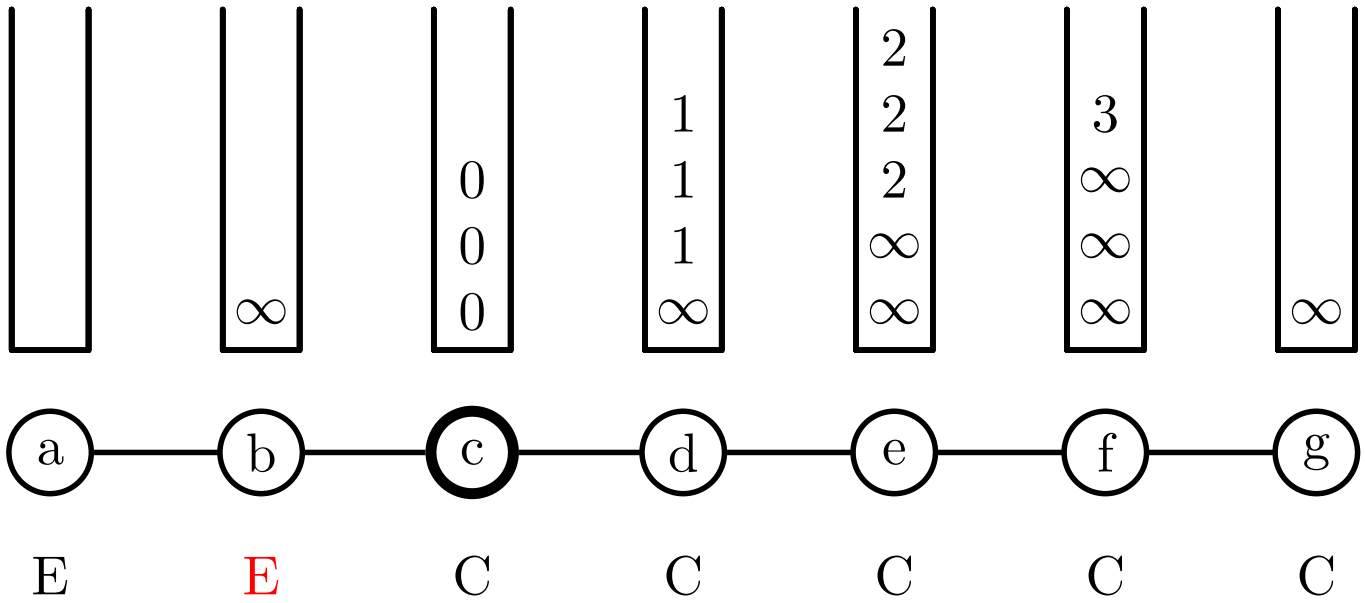
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

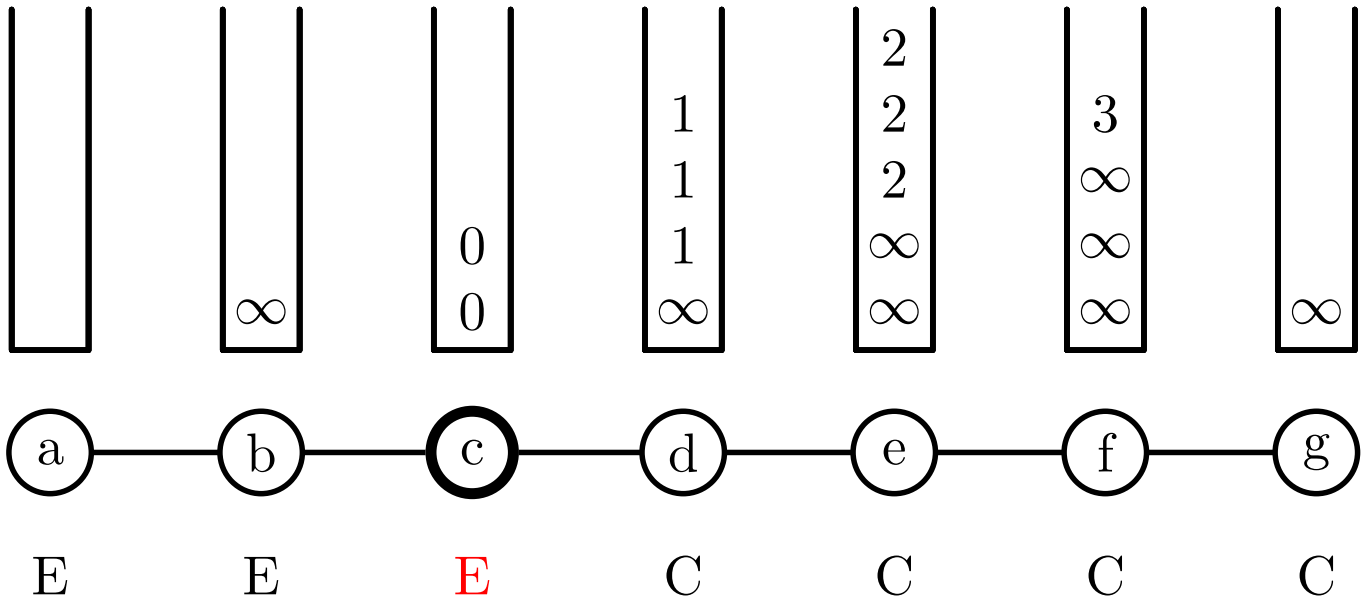
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

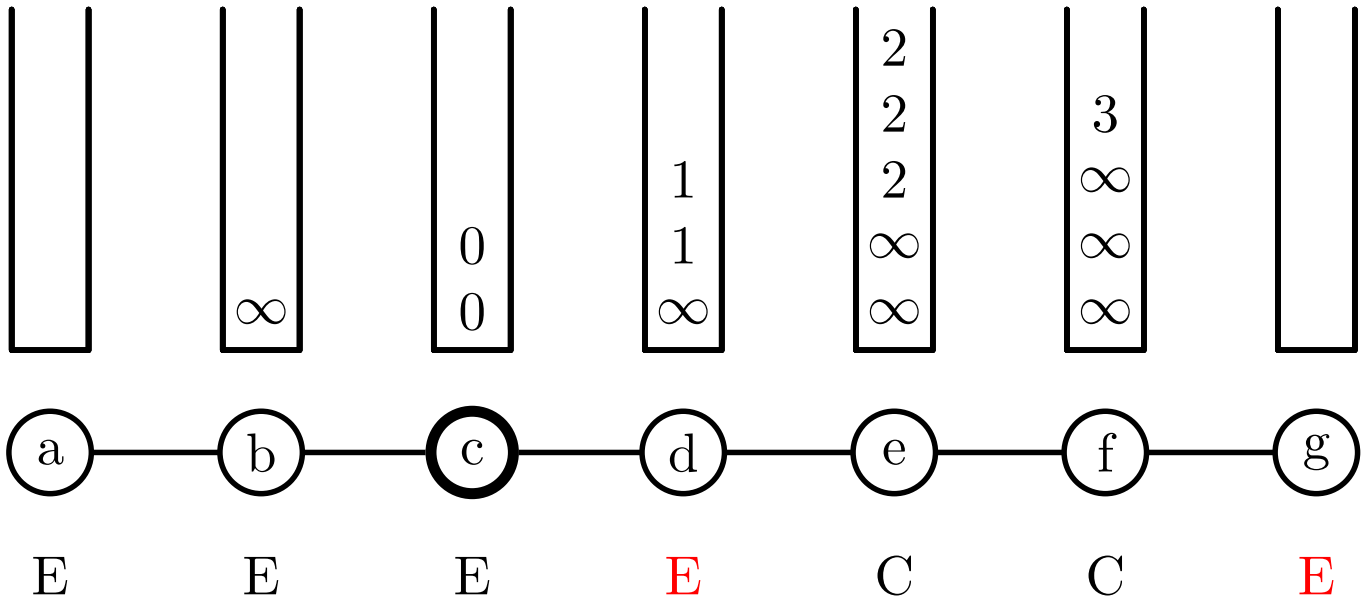
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

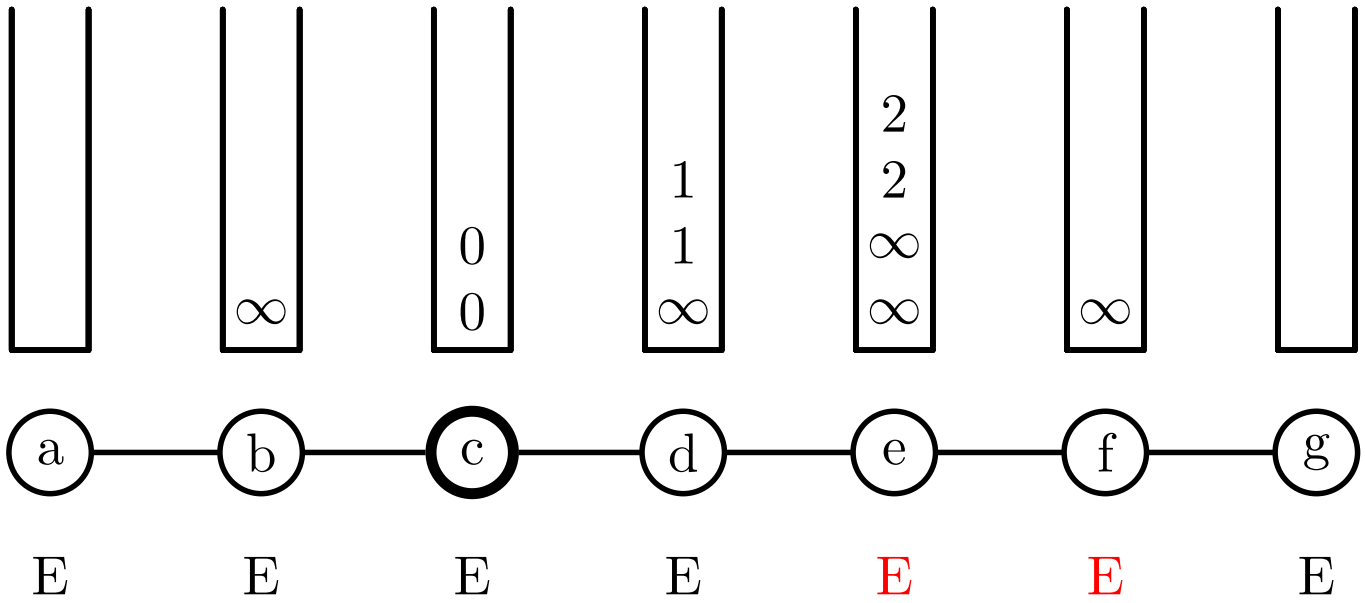
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

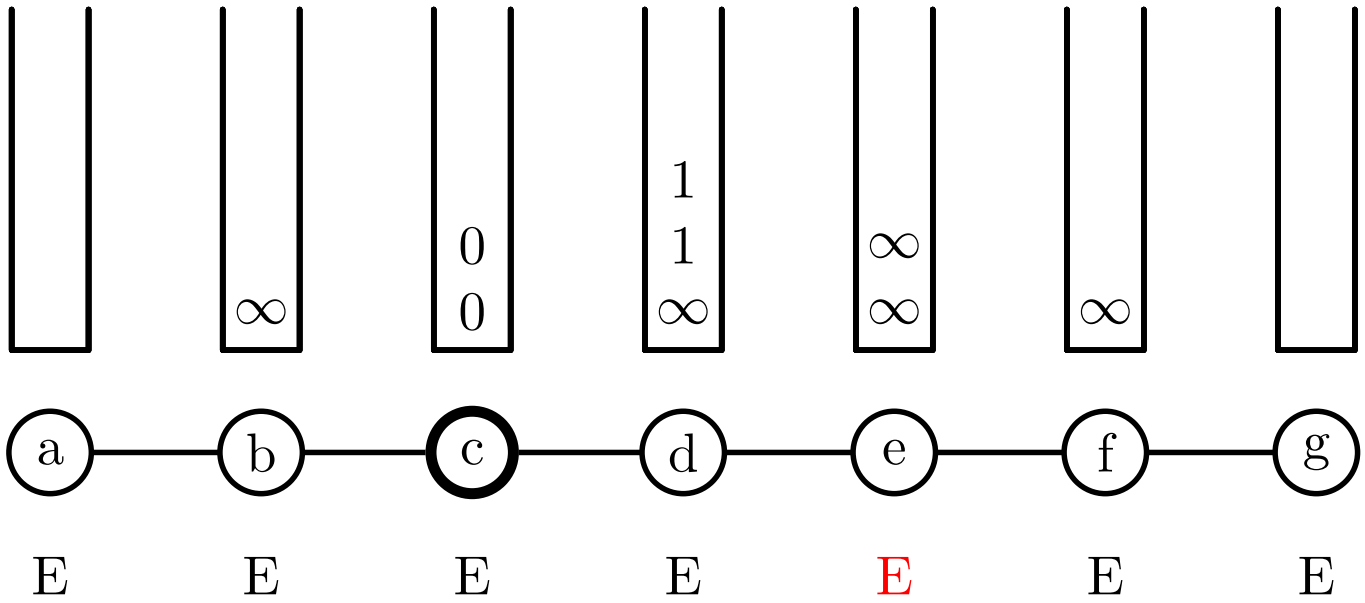
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

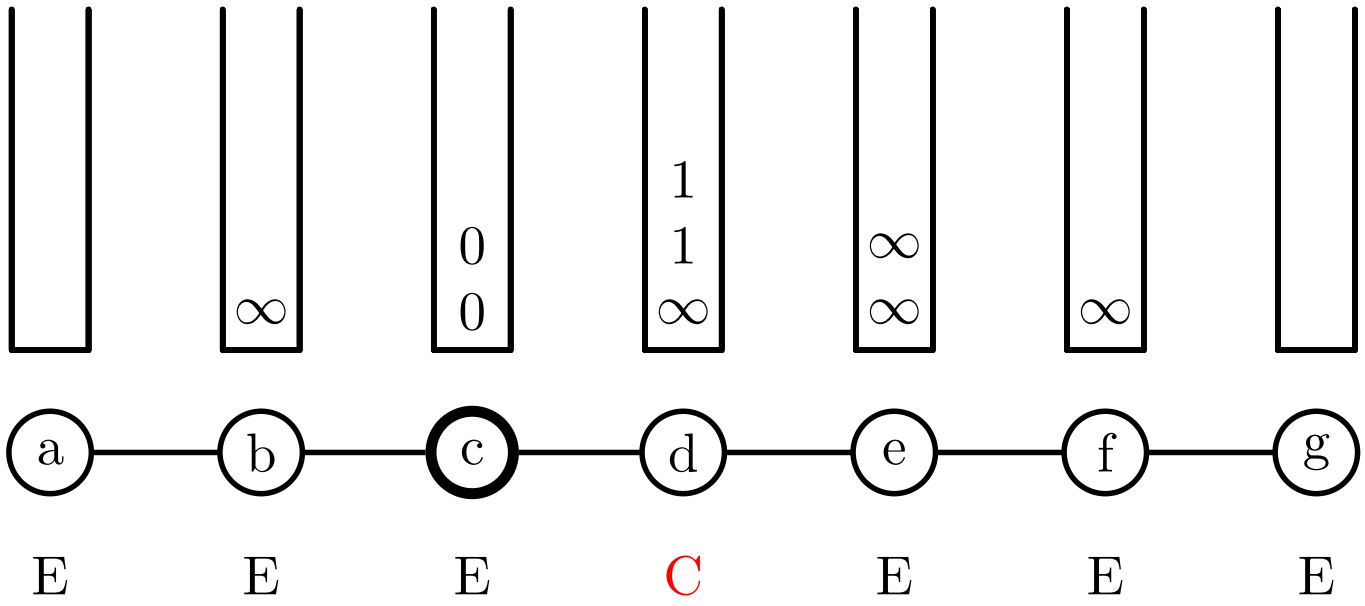
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

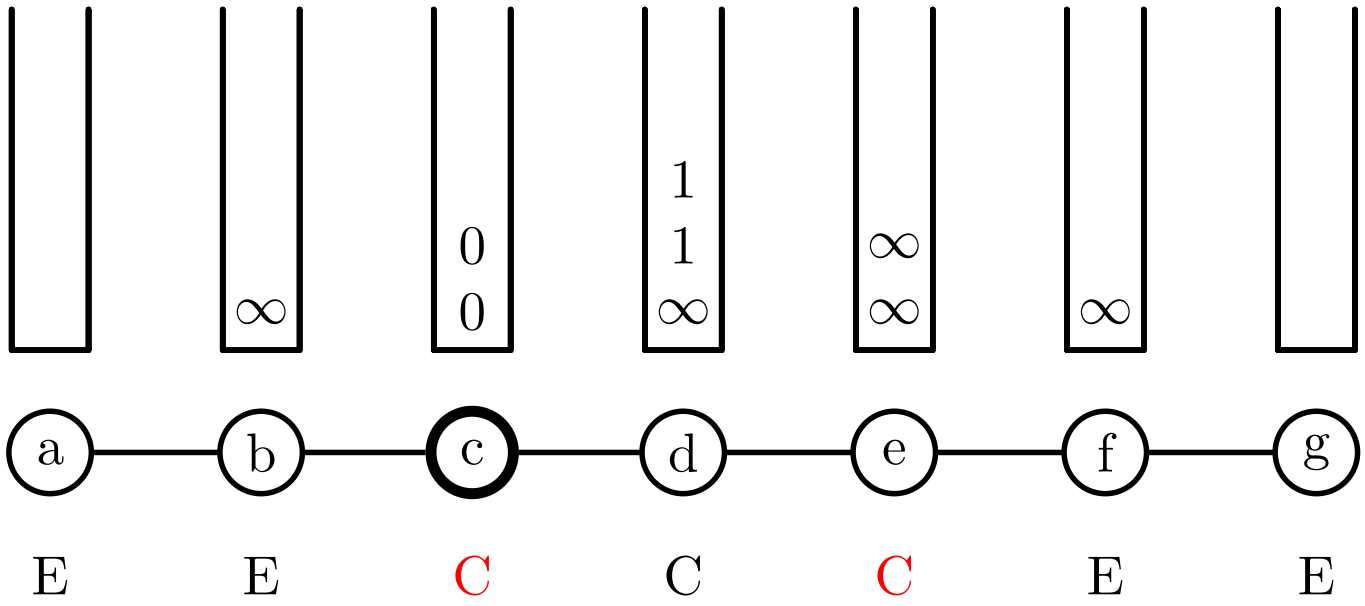
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

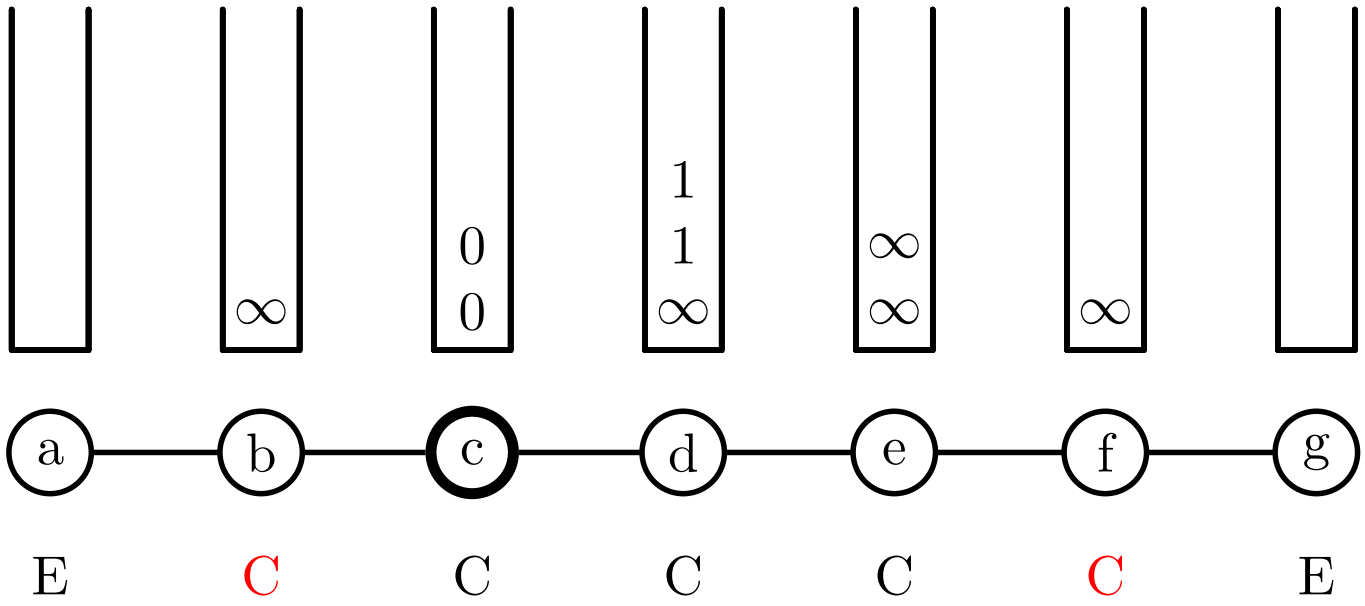
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

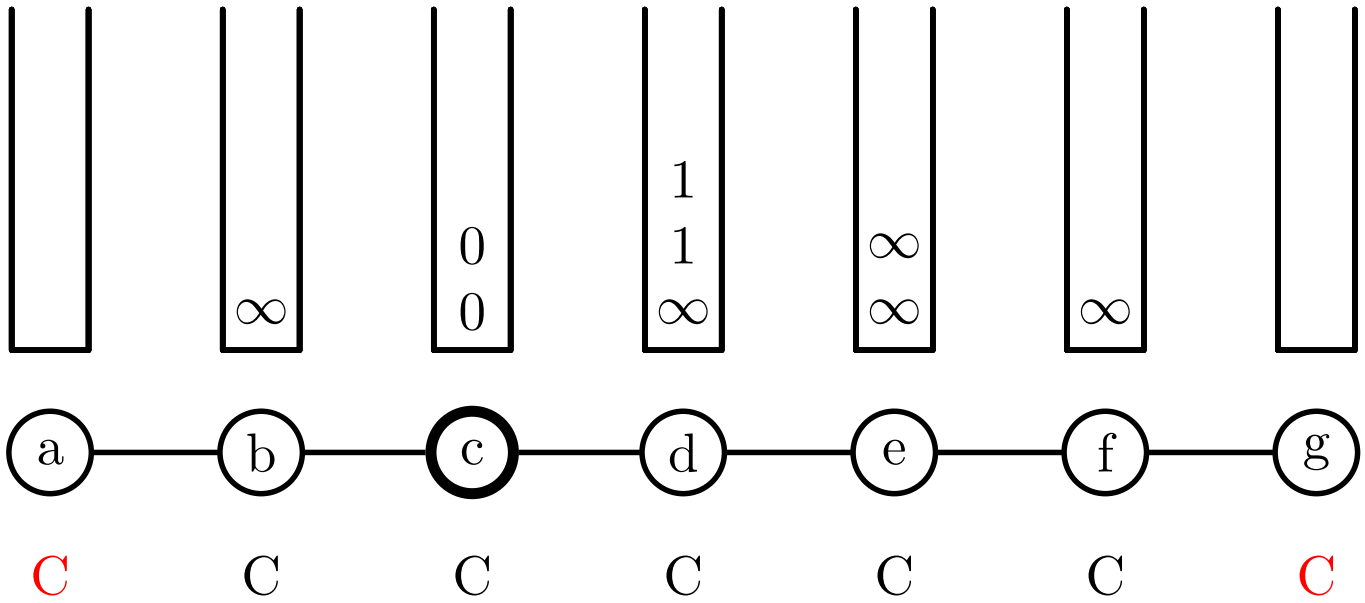
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

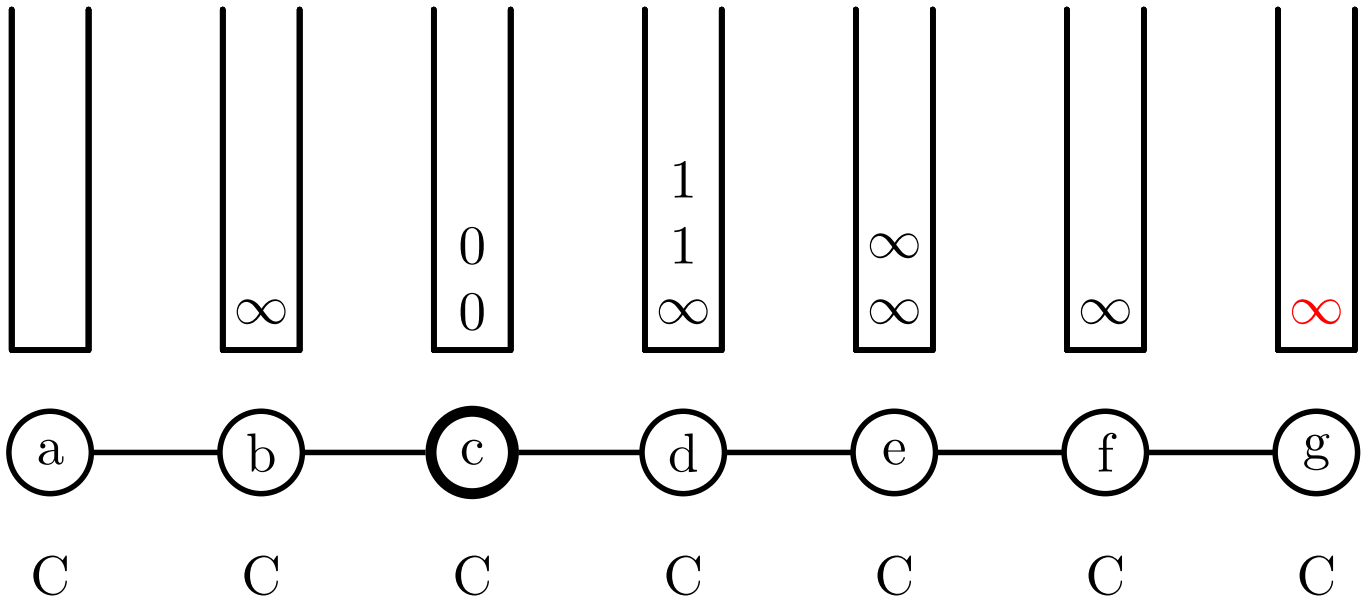
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

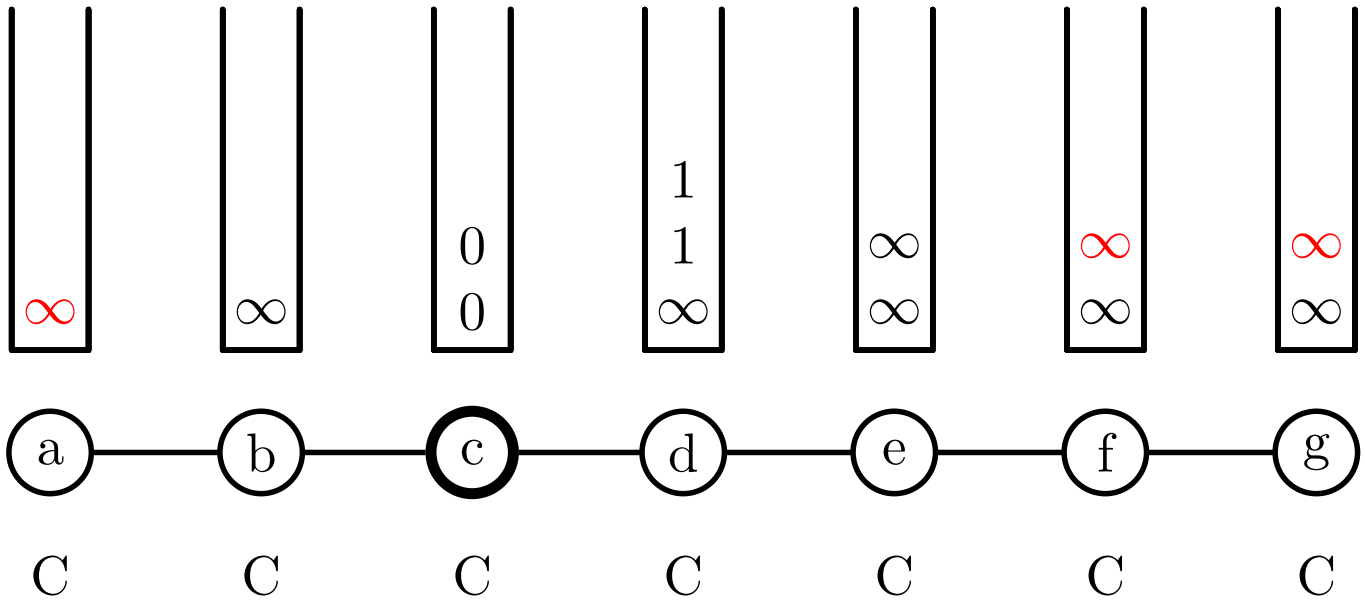
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

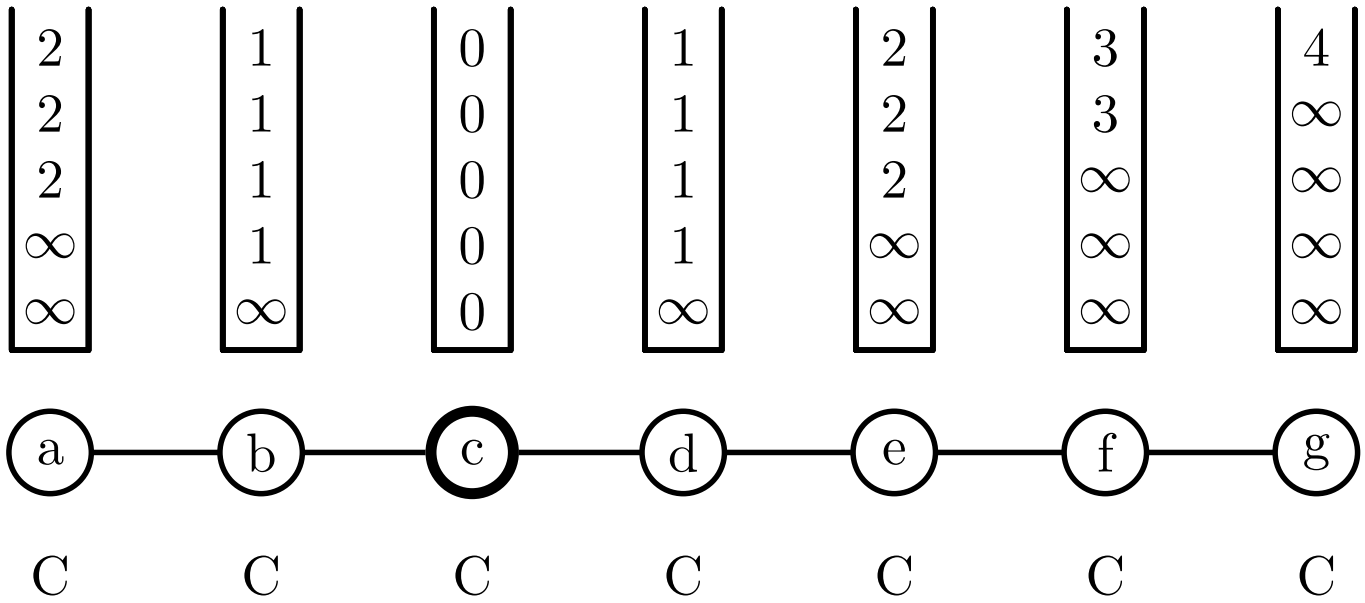
- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Overview of the algorithm

Ideas:

- storage of the whole synchronous execution;
- a node must correct all its dependency errors before it simulates the algorithm.



Dependences $\Rightarrow$ Reset Dags

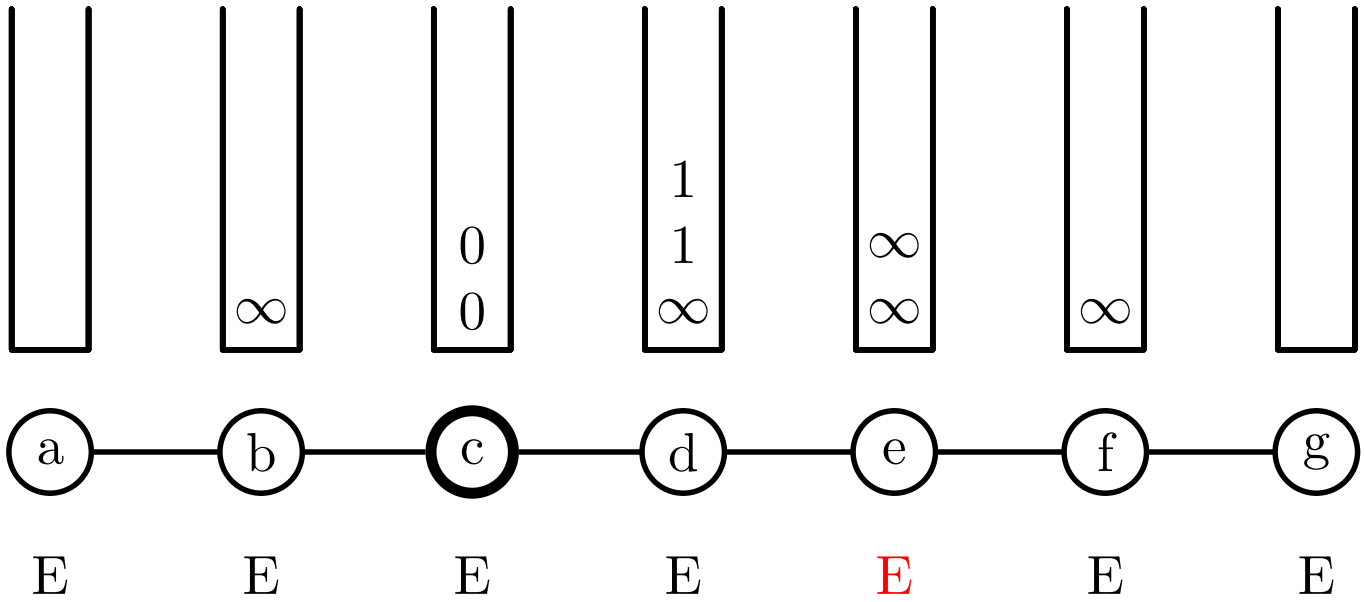
- Nodes with status E
- p is a parent of its neighbor q if p also has a status E and its list is smaller

Two key principles:

- **"Controlled" resets:** efficiency in moves
- **On-the-fly reset dag shrinking:** efficiency in rounds

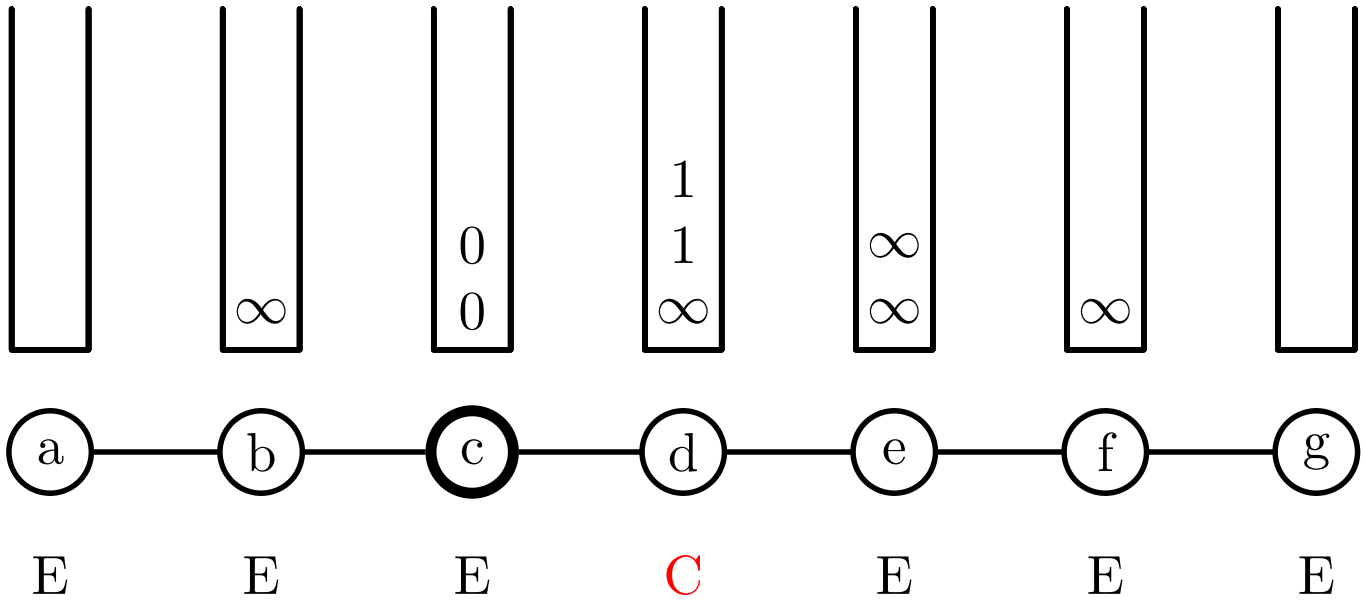
Efficiency in moves

"Controlled" resets: a node in error always resumes the "normal" execution after roots of its reset dag



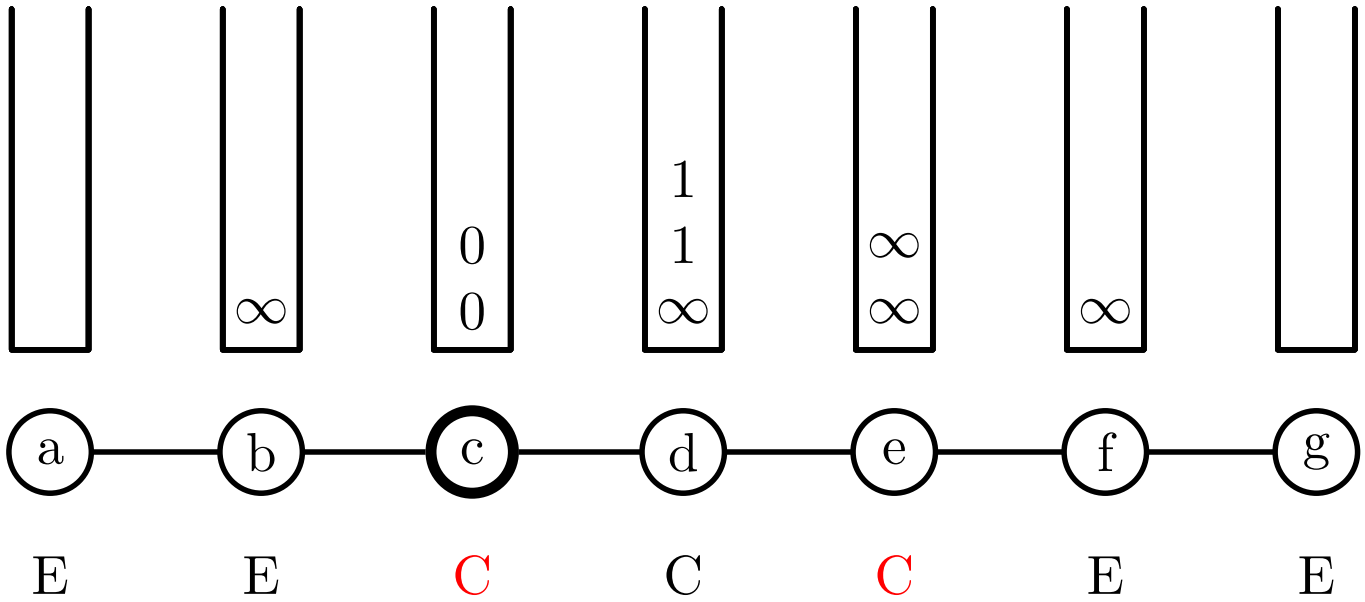
Efficiency in moves

"Controlled" resets: a node in error always resumes the "normal" execution after roots of its reset dag



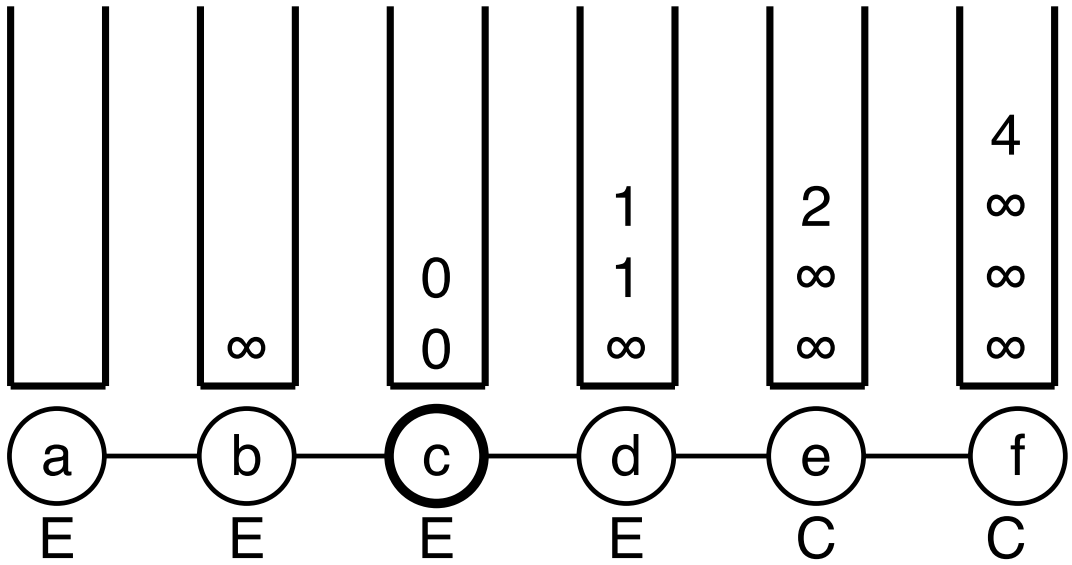
Efficiency in moves

"Controlled" resets: a node in error always resumes the "normal" execution after roots of its reset dag



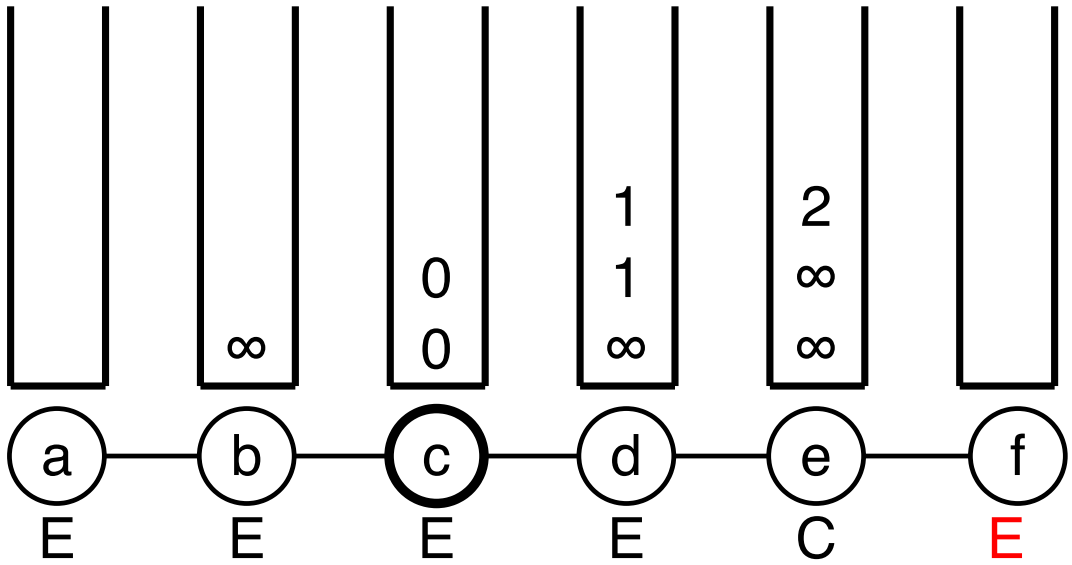
Efficiency in rounds

On-the-fly reset dag shrinking: an error node must be an error root, or its list must have one more element than the smallest list of an error neighbor.



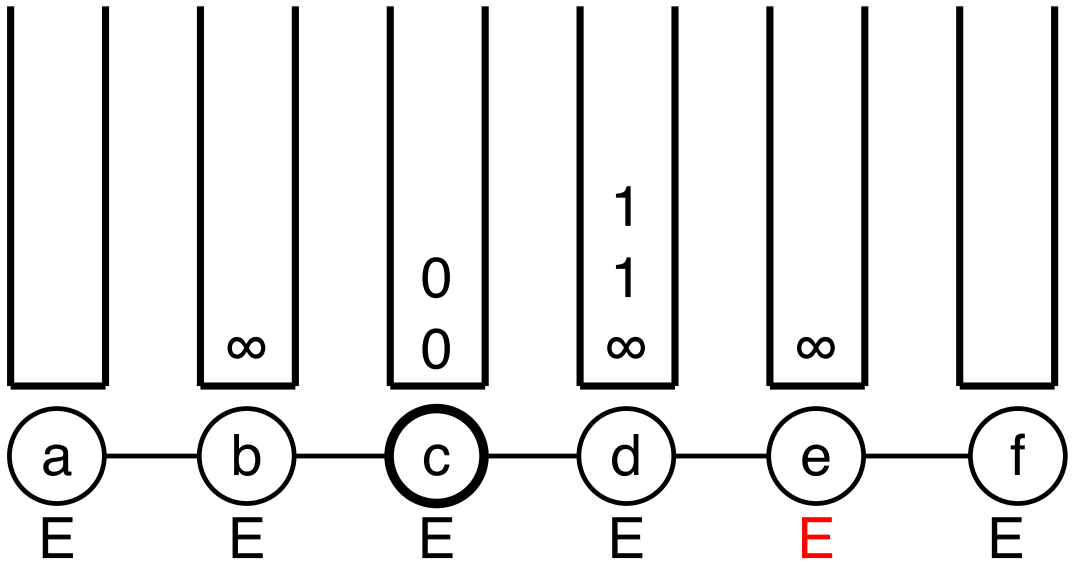
Efficiency in rounds

On-the-fly reset dag shrinking: an error node must be an error root, or its list must have one more element than the smallest list of an error neighbor.



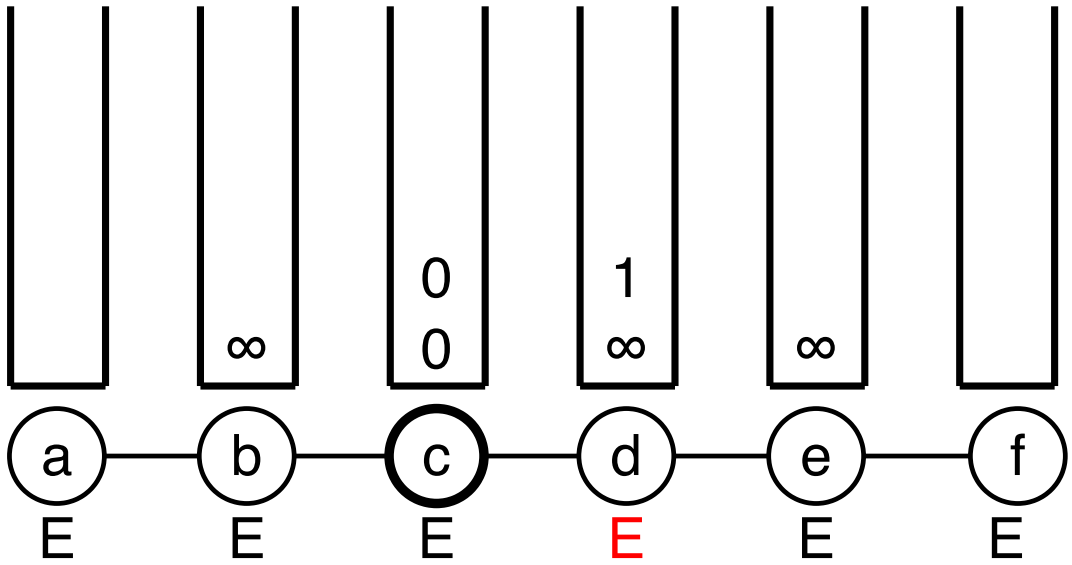
Efficiency in rounds

On-the-fly reset dag shrinking: an error node must be an error root, or its list must have one more element than the smallest list of an error neighbor.



Efficiency in rounds

On-the-fly reset dag shrinking: an error node must be an error root, or its list must have one more element than the smallest list of an error neighbor.



The algorithm

$$
\begin{aligned}
algoErr(p) &:= \exists i, 1 \leq i \leq p.h,\ (\forall q \in N(p), q.h \geq i-1) \wedge \\
&\qquad p.L[i] \neq \widehat{algo}(p, i-1) \\
depErr(p) &:= \Big(p.s = E \wedge \neg(\exists q \in N(p),\ q.s = E \wedge q.h < p.h)\Big) \\
&\vee \Big(p.s = C \wedge \exists q \in N(p),\ (q.h \geq p.h+2)\Big) \\
errPropag(p,i) &:= \exists q \in N(p),\ q.s = E \wedge q.h < i < p.h \\
canClearE(p) &:= p.s = E \wedge \forall q \in N(p),\ \Big(|q.h - p.h| \leq 1 \wedge (q.h \leq p.h \vee q.s = C)\Big) \\
updatable(p) &:= p.s = C \quad \wedge \quad p.h < B \quad \wedge \\
&\qquad \Big(\forall q \in N(p),\ q.h \in \{p.h, p.h+1\}\Big) \wedge \\
&\qquad \Big(\texttt{Greedy} \vee (p.L[p.h] \neq \widehat{algo}(p, p.h) \vee \exists q \in N(p),\ q.h > p.h)\Big)
\end{aligned}
$$

- $R_R : (p.h > 0 \vee p.s = C) \wedge (algoErr(p) \vee depErr(p)) \longrightarrow p.h := 0\ ;\ p.s := E$
- $R_P(i) : errorPropag(p,i) \longrightarrow p.h := i\ ;\ p.s := E$
- $R_C : canClearE(p) \longrightarrow p.s := C$
- $R_U : updatable(p) \longrightarrow push(\widehat{algo}(p, p.h))$

Priorities: $R_R > R_P(i)$ and $R_P(i) > R_P(i+l)$.

Perspectives

- Reduce Memory Requirement
  (a less general transformer, submitted to STACS)
- Weaker Models (e.g., link-register model)
- Non-terminating tasks (e.g., token circulation)

Thank You