Stably Computable Properties of Network Graphs

Dana Angluin James Aspnes Melody Chan
Michael J. Fischer Hong Jiang René Peralta

Yale University

June 30th, 2005

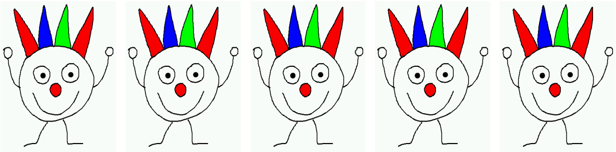
Topology of sensor networks

- Consider a **sensor network** with radio communication between nearby sensors.
- For some applications, we need to compute sensor positions (**localization**).
- But for other applications, computing the **topology** of the network may be enough.

Our question is: how much can we learn about the topology with extremely limited memory at each node and no built-in identities?

Why no identities is a problem

Every day, a neighbor rings my doorbell.



Is it one neighbor who keeps coming back or many neighbors who take turns?

For a finite-state machine, it's hard to remember.

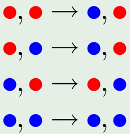
Population protocols

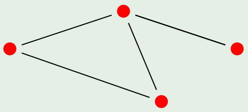
- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

Example: Leader Election

● = leader

● = non-leader

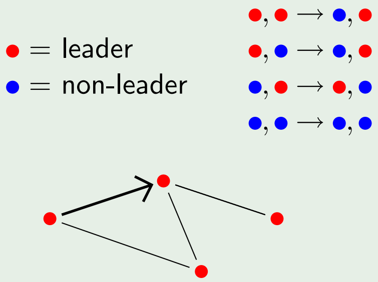




Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

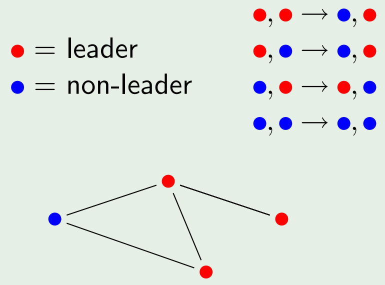
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

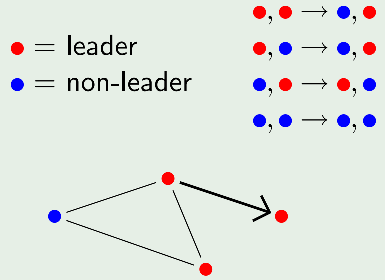
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

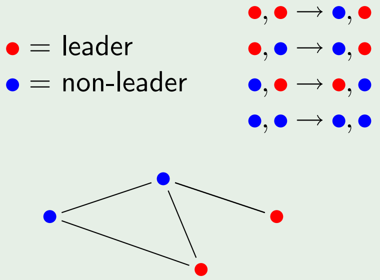
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

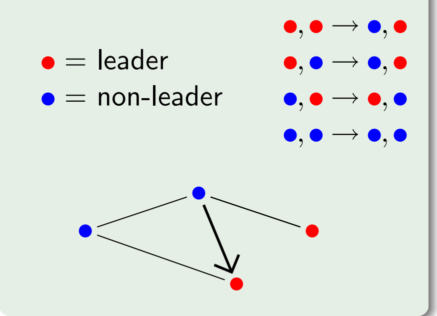
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

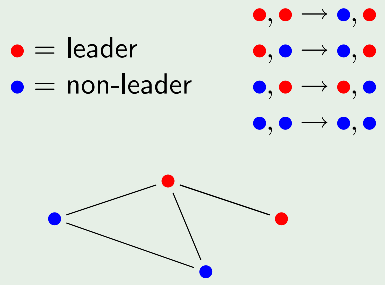
Example: Leader Election



Population protocols

- A **population protocol** [AAD[+]04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

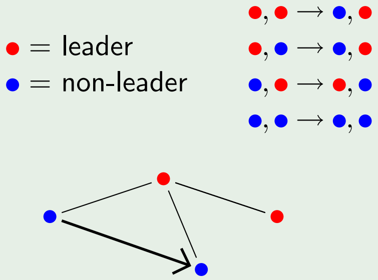
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

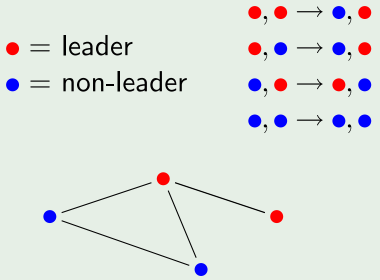
Example: Leader Election



Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

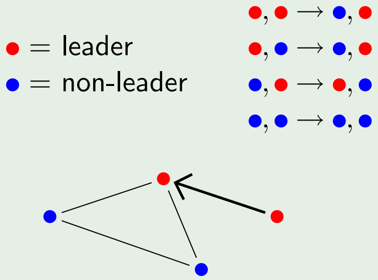
Example: Leader Election



Population protocols

- A **population protocol** [AAD$^+$04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

Example: Leader Election

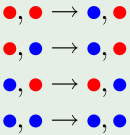


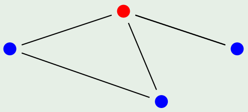
Population protocols

- A **population protocol** [AAD+04] consists of a collection of **finite-state nodes** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both nodes* according to a joint **transition function**.
- Interactions are *asymmetric*: one node is the **initiator** and one the **responder**.

Example: Leader Election

● = leader
● = non-leader





Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

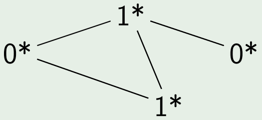
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

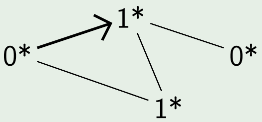
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

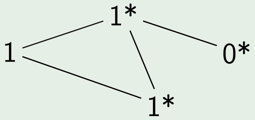
$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

$$0*, 0* \rightarrow 0, 0*$$
$$0*, 1* \rightarrow 1, 1*$$
$$1*, 0* \rightarrow 1, 1*$$
$$1*, 1* \rightarrow 0, 0*$$
$$x, y* \rightarrow y*, y$$
$$x*, y \rightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

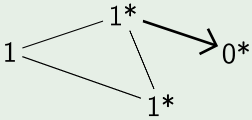
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

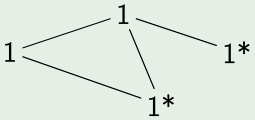
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

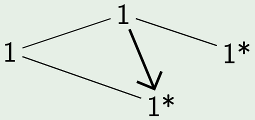
$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

$$0*, 0* \rightarrow 0, 0*$$
$$0*, 1* \rightarrow 1, 1*$$
$$1*, 0* \rightarrow 1, 1*$$
$$1*, 1* \rightarrow 0, 0*$$
$$x, y* \rightarrow y*, y$$
$$x*, y \rightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

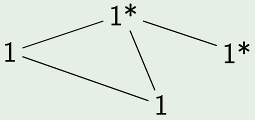
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

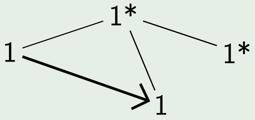
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

$$x \longrightarrow x*$$

Out:

$$x \longrightarrow x$$

$$x* \longrightarrow x$$

$$0*, 0* \longrightarrow 0, 0*$$

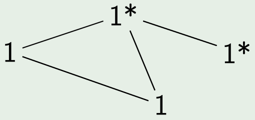
$$0*, 1* \longrightarrow 1, 1*$$

$$1*, 0* \longrightarrow 1, 1*$$

$$1*, 1* \longrightarrow 0, 0*$$

$$x, y* \longrightarrow y*, y$$

$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

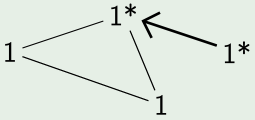
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

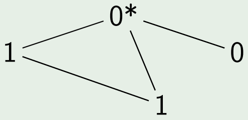
$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

$$0*, 0* \rightarrow 0, 0*$$
$$0*, 1* \rightarrow 1, 1*$$
$$1*, 0* \rightarrow 1, 1*$$
$$1*, 1* \rightarrow 0, 0*$$
$$x, y* \rightarrow y*, y$$
$$x*, y \rightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

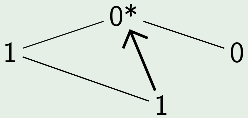
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

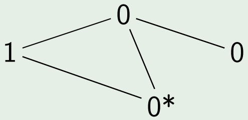
$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

$$0*, 0* \rightarrow 0, 0*$$
$$0*, 1* \rightarrow 1, 1*$$
$$1*, 0* \rightarrow 1, 1*$$
$$1*, 1* \rightarrow 0, 0*$$
$$x, y* \rightarrow y*, y$$
$$x*, y \rightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

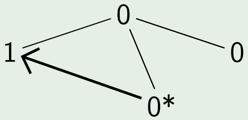
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all nodes.

Example: Parity

In:

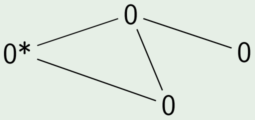
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$$0*, 0* \longrightarrow 0, 0*$$
$$0*, 1* \longrightarrow 1, 1*$$
$$1*, 0* \longrightarrow 1, 1*$$
$$1*, 1* \longrightarrow 0, 0*$$
$$x, y* \longrightarrow y*, y$$
$$x*, y \longrightarrow x, x*$$



Presburger predicates

Population protocols on connected graphs can **stably compute** all of **first-order Presburger arithmetic** on counts of input letters, including

- Addition
- Subtraction
- Multiplication by a constant k
- Remainder mod k
- $> k$, $< k$, $= k$
- $\wedge$, $\vee$, $\neg$, $\forall x$, $\exists x$, applied to above.

Shown for fixed inputs in [AAD$^+$04]. Still true even if inputs are not fixed, but converge after some finite time [current paper].
Conjecture: In a complete interaction graph, this is it.

Protocols on graphs

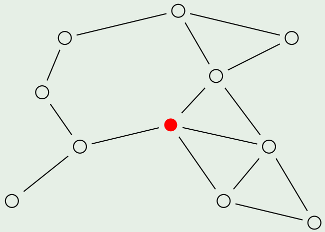
Counting input tokens tells us nothing about the graph.
Should we care?

- Detecting graph structure may tell us something about node locations.
- Some graphs may give more computational power. For example, nodes in a line can be organized into a cellular automaton (equivalent in power to a Turing machine!)

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

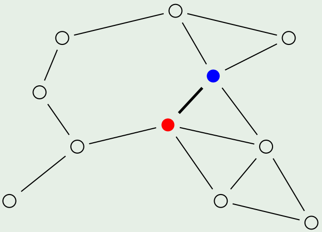


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

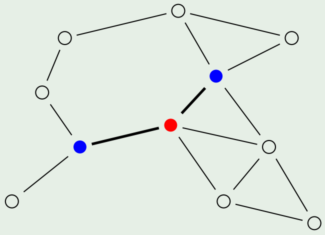


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

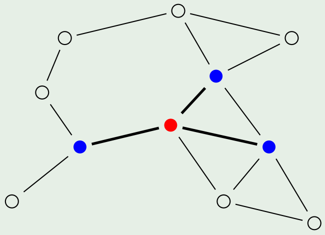


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

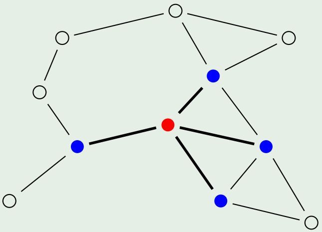


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

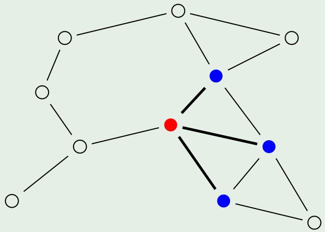


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

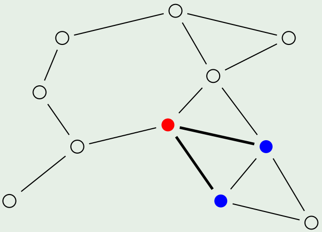


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

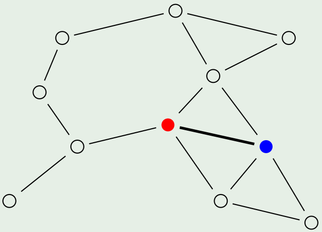


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

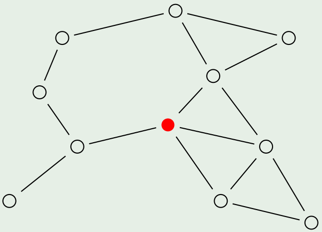


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

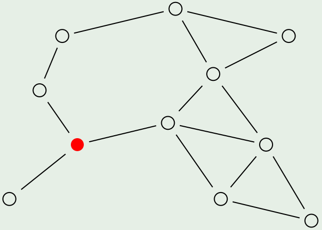


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

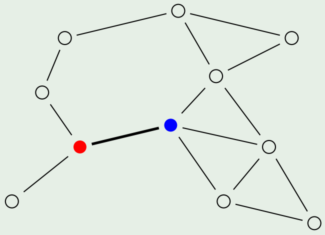


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

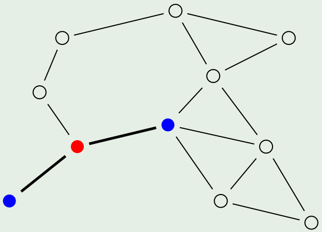


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

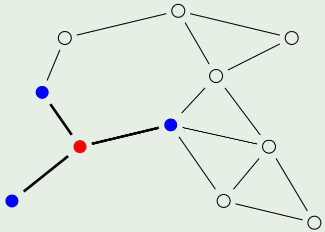


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

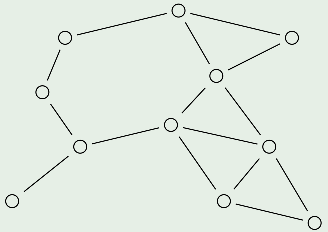


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

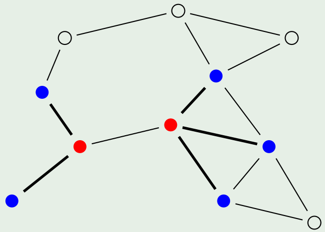
Example: Computing degrees



Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

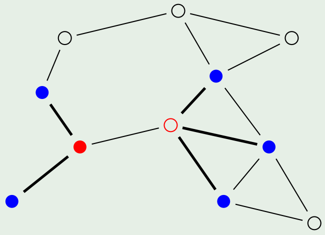


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

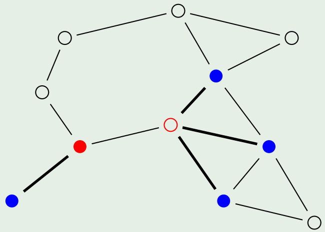


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

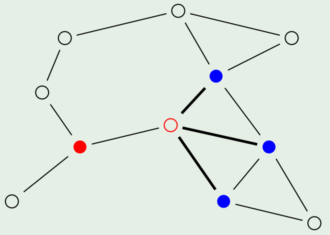


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

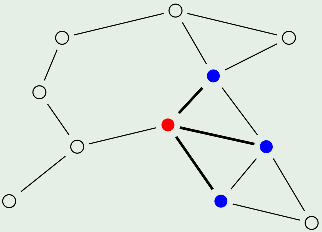


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

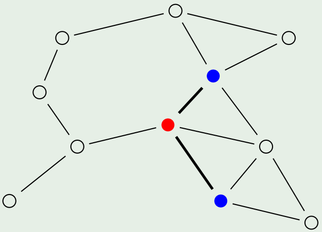


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

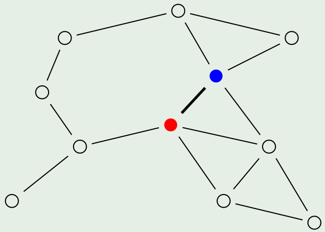


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

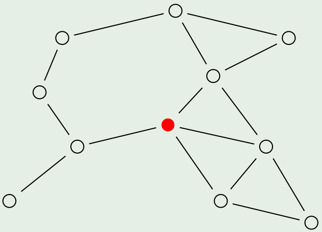


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Example: Computing degrees

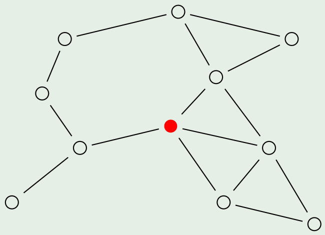


With two leaders, one consumes the other and then cleans up all followers.

Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

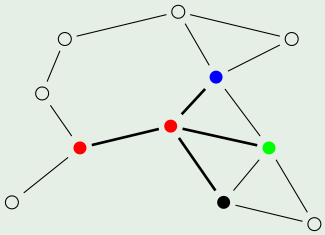
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

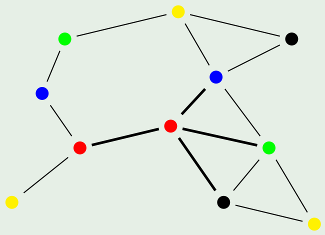
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

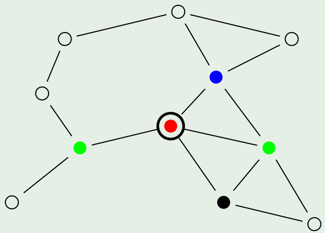
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

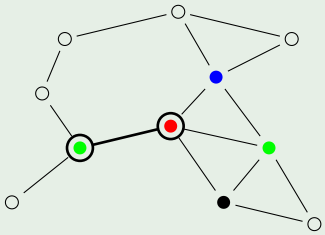
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

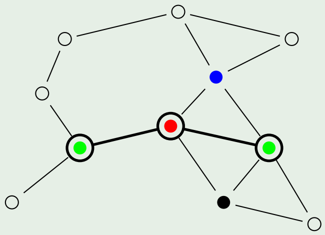
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

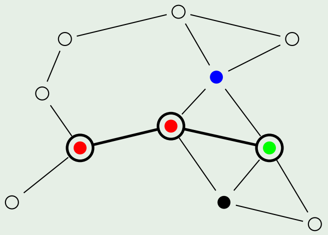
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

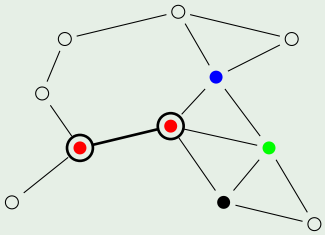
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

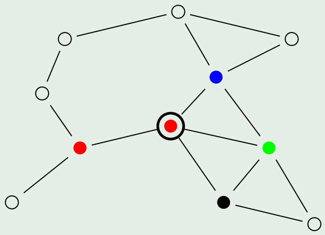
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

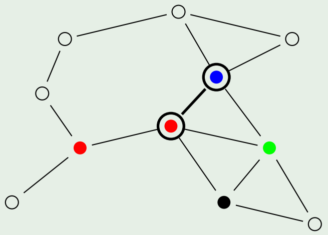
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

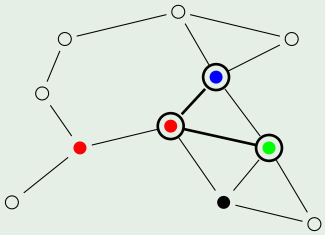
Example: Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.
- **Colorizer** agent walks around replacing forbidden colors.

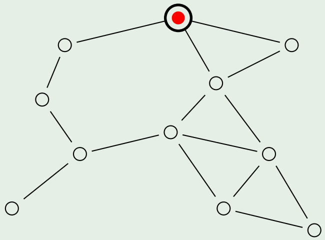
Example: Distance-2 coloring



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

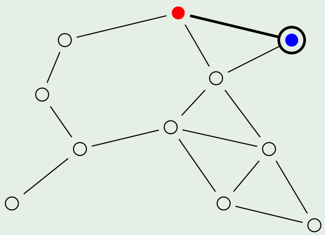
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

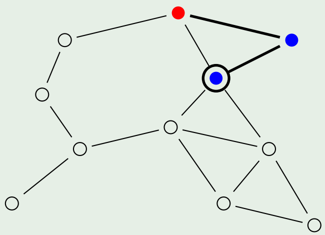
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

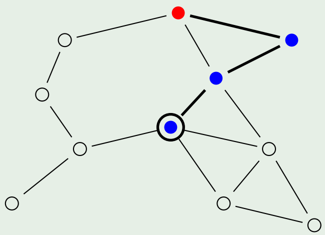
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

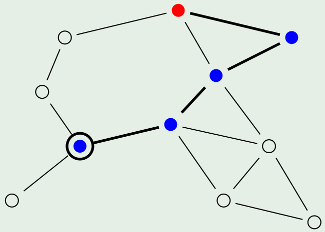
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

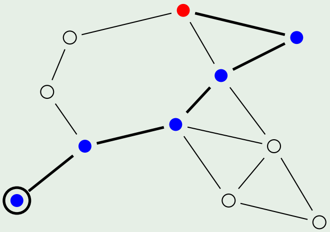
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

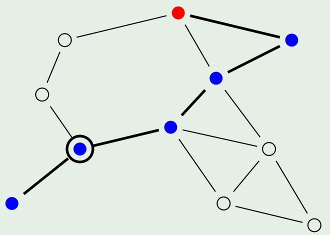
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

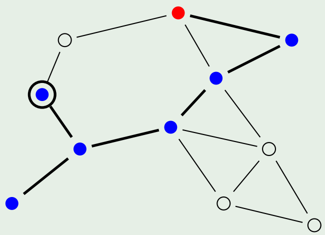
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

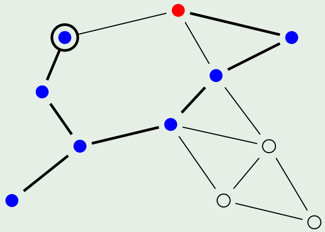
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

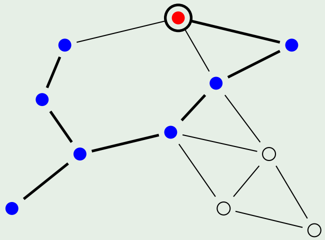
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

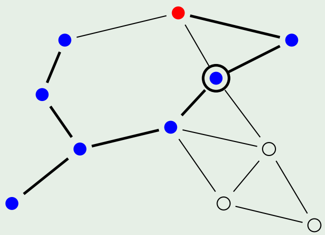
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

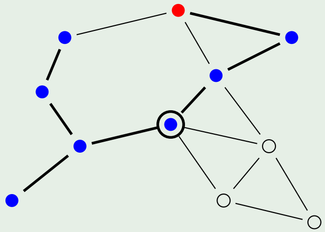
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

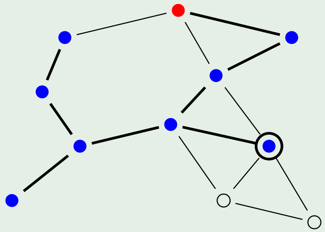
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

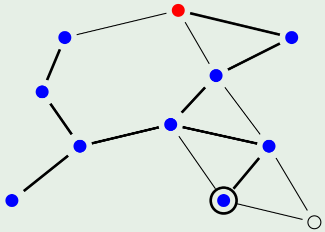
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

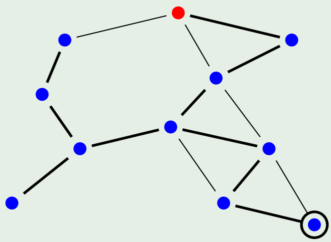
Example: Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

Example: Spanning tree



Distributed computation

- Unroll DFS traversal of spanning tree to get a linear Turing machine tape. [IL94]
- $\Rightarrow$ bounded-degree graph can compute all of LINSPACE.

Is it practical?

Population protocol model simplifies away many details of real systems.

- Two-way interactions may be unrealistic.
- Node failures are assumed not to occur.
- Running forever will run down batteries.

Problem is to incorporate more realism without losing simplicity.

Is it practical?

Algorithms have poor performance even if we assume non-adversarial interaction pattern.

- No performance analysis (yet).
- Wandering leaders may require $O(n^3)$ cover time to visit all nodes.
- Unique leader/colorizer/walker agents are bottlenecks.

Perhaps we can do better using a dominating-set approach as in ad-hoc networks.

Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta.
Computation in networks of passively mobile finite-state sensors.
In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 290–299. ACM Press, 2004.

G. Itkis and L. A. Levin.
Fast and lean self-stabilizing asynchronous protocols.
In *Proceeding of 35th Annual Symposium on Foundations of Computer Science*, pages 226–239. IEEE Press, 1994.