Symmetric Network Computation

and the Finite-State Symmetric Graph Automaton (FSSGA) Model

David Pritchard, with Santosh Vempala

2006 ACM Symposium on Parallelism in Architectures and Algorithms

Overview

- The “standard” message-passing model of distributed algorithms puts only weak restrictions on the computational power of each node.
- Some modern networks are very large and quite faulty (the internet, sensor networks).
- Motivation: can a much simpler model automatically ensure fault-tolerance? E.g., ‘smart dust’ networks with billions of identical microscopic finite-state nodes, unbounded degree.
- Using some qualitative principles that are common to many fault-tolerant algorithms, we propose a specific new model of finite-state computation, Finite-State Symmetric Graph Automata (FSSGA).
- Extends web automata, cellular automata.
- We don’t get what we *really* wanted (automatic fault tolerance). E.g., can break symmetry and elect a leader.
- Interesting features: multiple equivalent formulations, simple (simulatable by “most” models), interplay between parallel and sequential computing.

Overview

- The “standard” message-passing model of distributed algorithms puts only weak restrictions on the computational power of each node.
- Some modern networks are very large and quite faulty (the internet, sensor networks).
- Motivation: can a much simpler model automatically ensure fault-tolerance? E.g., ‘smart dust’ networks with billions of identical microscopic finite-state nodes, unbounded degree.
- Using some qualitative principles that are common to many fault-tolerant algorithms, we propose a specific new model of finite-state computation, Finite-State Symmetric Graph Automata (FSSGA).
- Extends web automata, cellular automata.
- We don’t get what we *really* wanted (automatic fault tolerance). E.g., can break symmetry and elect a leader.
- Interesting features: multiple equivalent formulations, simple (simulatable by “most” models), interplay between parallel and sequential computing.

Overview

- The “standard” message-passing model of distributed algorithms puts only weak restrictions on the computational power of each node.
- Some modern networks are very large and quite faulty (the internet, sensor networks).
- Motivation: can a much simpler model automatically ensure fault-tolerance? E.g., ‘smart dust’ networks with billions of identical microscopic finite-state nodes, unbounded degree.
- Using some qualitative principles that are common to many fault-tolerant algorithms, we propose a specific new model of finite-state computation, Finite-State Symmetric Graph Automata (FSSGA).
- Extends web automata, cellular automata.
- We don’t get what we *really* wanted (automatic fault tolerance). E.g., can break symmetry and elect a leader.
- Interesting features: multiple equivalent formulations, simple (simulatable by “most” models), interplay between parallel and sequential computing.

Outline

Problem Statement — Computing the Network Size

- Here's a "toy problem" to introduce three different computing paradigms: tree-based, agent-based, and decentralized.
- Problem. Given a network of unknown size and topology, compute the number of nodes in the network. You may assume a leader exists.
- Simple problem, but typical for sensor and ad-hoc networks. Also it generalizes to other forms of aggregation such as sums and averages.

Problem Statement — Computing the Network Size

- Here's a "toy problem" to introduce three different computing paradigms: tree-based, agent-based, and decentralized.
- Problem. Given a network of unknown size and topology, compute the number of nodes in the network. You may assume a leader exists.
- Simple problem, but typical for sensor and ad-hoc networks. Also it generalizes to other forms of aggregation such as sums and averages.

Trees and Agents

- In this talk, $n := |V|$, the number of nodes.
- First idea for computing network size: construct a spanning tree, and each node reports its subtree size to its parent.
- Easy enough to implement, but if any tree edge dies during the algorithm, then it fails. And there can be $\Theta(n)$ tree edges.
- Second idea: have an *agent* traverse the network.
- Each node stores a "visited" flag, and by this the agent can determine the number of unique nodes visited.
- Using, for example, a greedy routing strategy, we can traverse the network in $O(n \log n)$ steps.
- This improves the situation: even if some nodes/edges die, as long as the agent is not in the midst of a failure, and the graph remains connected, the algorithm works. But this one critical node remains, can we do better?
- Yes, if all we really need is an *estimate* of n.

Trees and Agents

- In this talk, $n := |V|$, the number of nodes.
- First idea for computing network size: construct a spanning tree, and each node reports its subtree size to its parent.
- Easy enough to implement, but if any tree edge dies during the algorithm, then it fails. And there can be $\Theta(n)$ tree edges.
- Second idea: have an *agent* traverse the network.
- Each node stores a "visited" flag, and by this the agent can determine the number of unique nodes visited.
- Using, for example, a greedy routing strategy, we can traverse the network in $O(n \log n)$ steps.
- This improves the situation: even if some nodes/edges die, as long as the agent is not in the midst of a failure, and the graph remains connected, the algorithm works. But this one critical node remains, can we do better?
- Yes, if all we really need is an *estimate* of n.

Trees and Agents

- In this talk, $n := |V|$, the number of nodes.
- First idea for computing network size: construct a spanning tree, and each node reports its subtree size to its parent.
- Easy enough to implement, but if any tree edge dies during the algorithm, then it fails. And there can be $\Theta(n)$ tree edges.
- Second idea: have an *agent* traverse the network.
- Each node stores a "visited" flag, and by this the agent can determine the number of unique nodes visited.
- Using, for example, a greedy routing strategy, we can traverse the network in $O(n \log n)$ steps.
- This improves the situation: even if some nodes/edges die, as long as the agent is not in the midst of a failure, and the graph remains connected, the algorithm works. But this one critical node remains, can we do better?
- Yes, if all we really need is an *estimate* of n.

Trees and Agents

- In this talk, $n := |V|$, the number of nodes.
- First idea for computing network size: construct a spanning tree, and each node reports its subtree size to its parent.
- Easy enough to implement, but if any tree edge dies during the algorithm, then it fails. And there can be $\Theta(n)$ tree edges.
- Second idea: have an *agent* traverse the network.
- Each node stores a "visited" flag, and by this the agent can determine the number of unique nodes visited.
- Using, for example, a greedy routing strategy, we can traverse the network in $O(n \log n)$ steps.
- This improves the situation: even if some nodes/edges die, as long as the agent is not in the midst of a failure, and the graph remains connected, the algorithm works. But this one critical node remains, can we do better?
- Yes, if all we really need is an *estimate* of n.

A Decentralized Census Algorithm

[Flajolet and Martin, 1985 / Shah and Mosk-Aoyama, PODC 2006]

- The agent algorithm still has one critical node whose failure is disastrous.
- Some algorithms completely avoid having any fragile structure, with no node more important than any other. Often called *decentralized*.
- Here's an algorithm to *approximately* count the number of nodes in a faulty network.
- Suppose each node has $k \geq \log_2 \log_2 n$ bits of memory, initially all 0.
- Each node assigns its memory a value by this distribution: for each $0 \leq i < 2^k$, set memory value to i with probability 2^{-1-i}.
- All nodes repeatedly broadcast their value and replace memory contents with maximum known value. (Information propagation)
- Let v be final value. With prob. 2/3, the number 2^v is a constant-factor approximation to the network size. (Can improve somewhat.)
- This algorithm can tolerate any "reasonable" failure pattern.

A Decentralized Census Algorithm

[Flajolet and Martin, 1985 / Shah and Mosk-Aoyama, PODC 2006]

- The agent algorithm still has one critical node whose failure is disastrous.
- Some algorithms completely avoid having any fragile structure, with no node more important than any other. Often called *decentralized*.
- Here's an algorithm to *approximately* count the number of nodes in a faulty network.
- Suppose each node has $k \geq \log_2 \log_2 n$ bits of memory, initially all 0.
- Each node assigns its memory a value by this distribution: for each $0 \leq i < 2^k$, set memory value to i with probability 2^{-1-i}.
- All nodes repeatedly broadcast their value and replace memory contents with maximum known value. (Information propagation)
- Let v be final value. With prob. 2/3, the number 2^v is a constant-factor approximation to the network size. (Can improve somewhat.)
- This algorithm can tolerate any "reasonable" failure pattern.

A Decentralized Census Algorithm

[Flajolet and Martin, 1985 / Shah and Mosk-Aoyama, PODC 2006]

- The agent algorithm still has one critical node whose failure is disastrous.
- Some algorithms completely avoid having any fragile structure, with no node more important than any other. Often called *decentralized*.
- Here's an algorithm to *approximately* count the number of nodes in a faulty network.
- Suppose each node has $k \geq \log_2 \log_2 n$ bits of memory, initially all 0.
- Each node assigns its memory a value by this distribution: for each $0 \leq i < 2^k$, set memory value to i with probability 2^{-1-i}.
- All nodes repeatedly broadcast their value and replace memory contents with maximum known value. (Information propagation)
- Let v be final value. With prob. 2/3, the number 2^v is a constant-factor approximation to the network size. (Can improve somewhat.)
- This algorithm can tolerate any "reasonable" failure pattern.

What Properties are Essential?

- Rank by fault-tolerance: tree-based $\preceq$ agent-based $\preceq$ decentralized. (See *sensitivity* in proceedings).
- Do the "best" decentralized algorithms have some similarities? We isolated three properties.

(P1) *Global Symmetry:* the computation proceeds via a single operation that is performed repeatedly by every node.

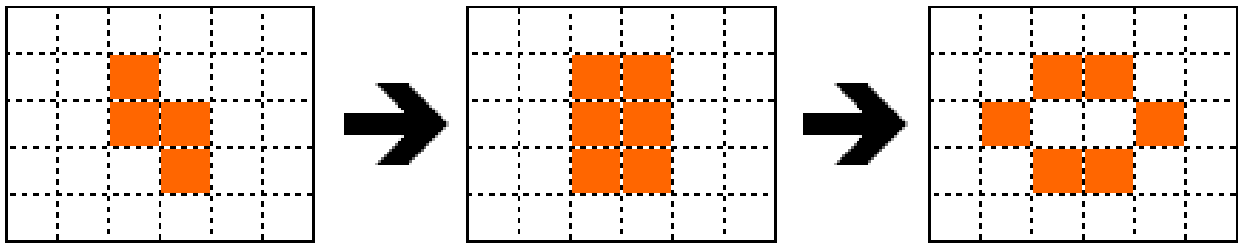
(P2) *Local Symmetry:* every node acts symmetrically on its neighbours.

(P3) *Steady State Convergence:* the network is brought to a steady state when all nodes perform their operation repeatedly.

- (Examples: Flajolet-Martin, preflow-push, α synchronizer, harmonic functions, self-stabilizing model)
- We'd like to construct a distributed model with these three properties.
- Let us review some older FSA-based models with these symmetry principles in mind. **Properties 1 and 2 motivate our model the most.**
- FSA = finite-state automaton

Cellular Automata, Lattices

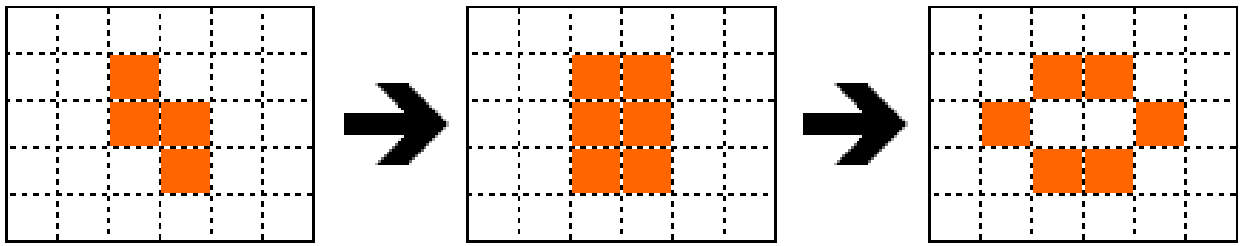
- "Life" (Conway) is a highly popularized example of a cellular automaton in 2D. State space $Q = \{alive, dead\}$; all nodes compute synchronously, with the same rule, based on own state and number of of live neighbours.



- Cellular automata: network topology is an arbitrary lattice. Uses a state space and a set of transition rules telling each node how to evolve based on its neighbourhood.

Cellular Automata, Lattices

- "Life" (Conway) is a highly popularized example of a cellular automaton in 2D. State space $Q = \{alive, dead\}$; all nodes compute synchronously, with the same rule, based on own state and number of of live neighbours.



- Cellular automata: network topology is an arbitrary lattice. Uses a state space and a set of transition rules telling each node how to evolve based on its neighbourhood.

Regular and Bounded-Degree Symmetric Models

- Even if graph doesn't have automorphisms, in the case of Δ-regular graphs we can still make all nodes run identically and symmetrically.
- Let Q be the set of states. Given an arbitrary Δ-regular graph, we may think of the transition function as

$$f : Q \times Q^{\Delta} \to Q$$

  where we may insist that f is symmetric in its second argument.
- Generalizes to Δ-bounded degree graphs; add a special "deficient" symbol ϵ and have

$$f : Q \times (Q \cup \{\epsilon\})^{\Delta} \to Q$$

  and symmetric as before. [Martin; Rémila; Rosenstiehl, Fiksel, Holliger]
- But the restriction to bounded degrees seems artificial. We want our new model to be simple, and to work in graphs of unbounded degree.

Outline

Building Block: Symmetric Multi-Input Automata

- We want each node to act symmetrically on its neighbours (local symmetry). We also want all nodes to be the same (global symmetry), regardless of differing degree. Further, we'd like to be able to process arbitrarily many neighbours using only finite space.
- The transition function operates as

$$Q \times Q^* \to Q.$$

- For now focus on functions

$$Q^* \to Q.$$

Our distributed model will be: with each state q associate a symmetric function $f[q] : Q^* \to Q$; when a node in state q activates, its state is replaced by the output of $f[q]$ using its neighbours' states as input.

- We came up with two reasonable FSA-based models with these properties: *sequential* and *parallel* symmetric multi-input automata.
- "Low" level of model: symmetric multi-input automata. "High" level: connect many identical automata together as a graph.

Building Block: Symmetric Multi-Input Automata

- We want each node to act symmetrically on its neighbours (local symmetry). We also want all nodes to be the same (global symmetry), regardless of differing degree. Further, we'd like to be able to process arbitrarily many neighbours using only finite space.
- The transition function operates as

$$Q \times Q^* \to Q.$$

- For now focus on functions

$$Q^* \to Q.$$

  Our distributed model will be: with each state q associate a symmetric function $f[q] : Q^* \to Q$; when a node in state q activates, its state is replaced by the output of $f[q]$ using its neighbours' states as input.
- We came up with two reasonable FSA-based models with these properties: *sequential* and *parallel* symmetric multi-input automata.
- "Low" level of model: symmetric multi-input automata. "High" level: connect many identical automata together as a graph.

Building Block: Symmetric Multi-Input Automata

- We want each node to act symmetrically on its neighbours (local symmetry). We also want all nodes to be the same (global symmetry), regardless of differing degree. Further, we'd like to be able to process arbitrarily many neighbours using only finite space.
- The transition function operates as

$$Q \times Q^* \to Q.$$

- For now focus on functions

$$Q^* \to Q.$$

  Our distributed model will be: with each state q associate a symmetric function $f[q] : Q^* \to Q$; when a node in state q activates, its state is replaced by the output of $f[q]$ using its neighbours' states as input.
- We came up with two reasonable FSA-based models with these properties: *sequential* and *parallel* symmetric multi-input automata.
- "Low" level of model: symmetric multi-input automata. "High" level: connect many identical automata together as a graph.

Sequential Multi-Input Automata

- Idea behind the sequential multi-input automaton: when a node activates, it should read each of its neighbours in one-at-a-time; each neighbour causes a state transition of a "working state" visible only to that node.
- Formal definition, take
  - A finite set W of (inner) "working states"
    [disjoint from (outer) node states Q]
  - An initial working state w_0
  - A processing function $p : W \times Q \to W$
    [current working state $\times$ neighbour's input $\mapsto$ new working state]
  - An output function $\beta : W \to R$ ($R =$ output set)
    [R will be the same as Q when we glue these automata together]

  If for all $\vec{q} \in Q^*$, where $|\vec{q}| = k$, for all $\pi \in S_k$, the expression

$$\beta(p(p \cdots p(p(w_0, q_{\pi(1)}), q_{\pi(2)}), \cdots, q_{\pi(k)}))$$

  is independent of π, then (W, w_0, p, β) defines a sequential multi-input automaton. We call (W, w_0, p, β) a *sequential program*, maps $Q^* \to R$.
- Computational power is precisely the following: can distinguish between a finite number of symmetric regular languages.

Sequential Multi-Input Automata

- Idea behind the sequential multi-input automaton: when a node activates, it should read each of its neighbours in one-at-a-time; each neighbour causes a state transition of a "working state" visible only to that node.
- Formal definition, take
  - A finite set W of (inner) "working states"
    [disjoint from (outer) node states Q]
  - An initial working state w_0
  - A processing function $p : W \times Q \to W$
    [current working state $\times$ neighbour's input $\mapsto$ new working state]
  - An output function $\beta : W \to R$ ($R =$ output set)
    [R will be the same as Q when we glue these automata together]

  If for all $\vec{q} \in Q^*$, where $|\vec{q}| = k$, for all $\pi \in S_k$, the expression

$$\beta(p(p \cdots p(p(w_0, q_{\pi(1)}), q_{\pi(2)}), \cdots, q_{\pi(k)}))$$

  is independent of π, then (W, w_0, p, β) defines a sequential multi-input automaton. We call (W, w_0, p, β) a *sequential program*, maps $Q^* \to R$.
- Computational power is precisely the following: can distinguish between a finite number of symmetric regular languages.

Sequential Multi-Input Automata

- Idea behind the sequential multi-input automaton: when a node activates, it should read each of its neighbours in one-at-a-time; each neighbour causes a state transition of a "working state" visible only to that node.
- Formal definition, take
  - A finite set W of (inner) "working states"
    [disjoint from (outer) node states Q]
  - An initial working state w_0
  - A processing function $p : W \times Q \to W$
    [current working state $\times$ neighbour's input $\mapsto$ new working state]
  - An output function $\beta : W \to R$ ($R =$ output set)
    [R will be the same as Q when we glue these automata together]

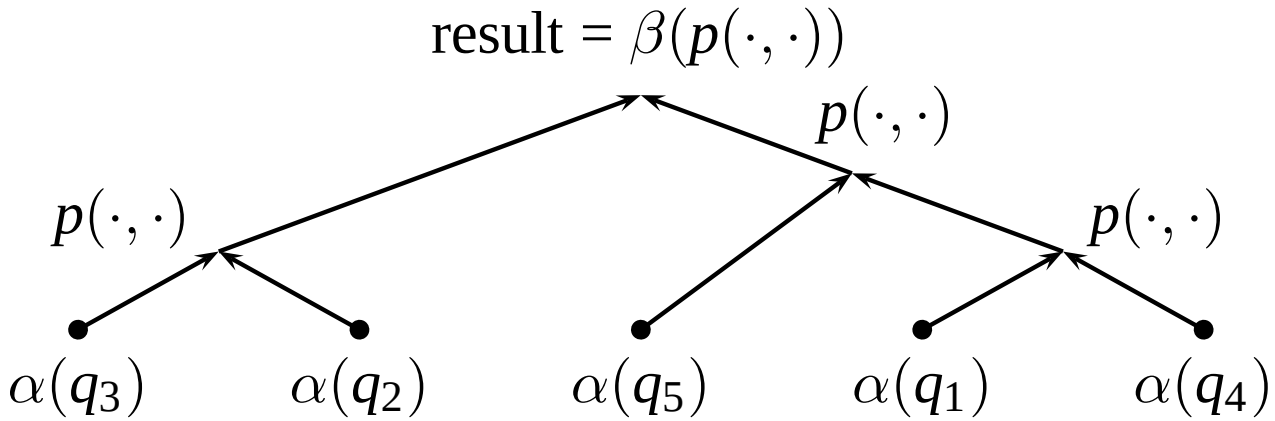
  If for all $\vec{q} \in Q^*$, where $|\vec{q}| = k$, for all $\pi \in S_k$, the expression

$$\beta(p(p \cdots p(p(w_0, q_{\pi(1)}), q_{\pi(2)}), \cdots, q_{\pi(k)}))$$

  is independent of π, then (W, w_0, p, β) defines a sequential multi-input automaton. We call (W, w_0, p, β) a *sequential program*, maps $Q^* \to R$.
- Computational power is precisely the following: can distinguish between a finite number of symmetric regular languages.

Parallel Multi-Input Automata

- We thought of another reasonable model by which a symmetric multi-input function could be realized: parallel instead of sequential.
- Basic idea: each neighbour contributes a single datum and they are reduced pairwise.



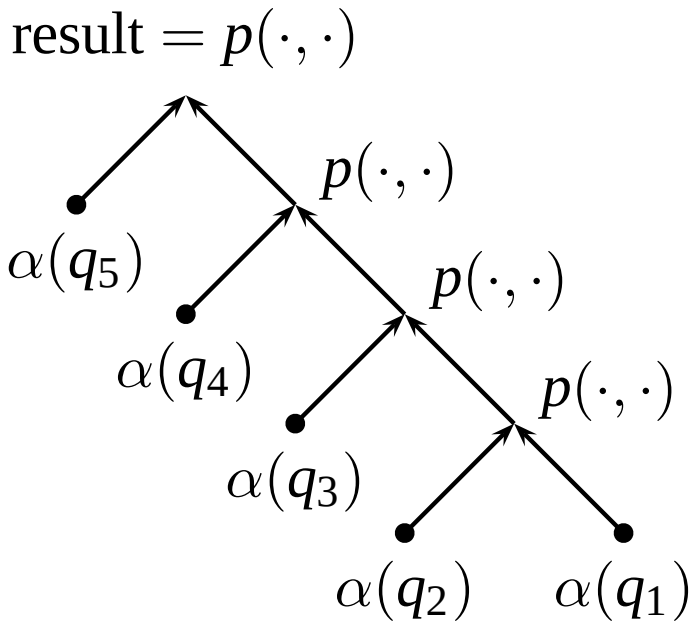
- A *parallel* symmetric multi-automaton is uses a "combining" rule but it must be the case that the same result occurs regardless of the specific way in which the inputs are combined.
- Formalized in proceedings.

Sequential Can Simulate Parallel

Lemma

Each sequential symmetric multi-input automaton can be simulated by a parallel one.

- Idea: if we can divide and conquer, then conquer just one input at a time.



Sequential-Parallel Equivalence

- Main technical contribution: we show the sequential and parallel models are equivalent. Proof uses the *mod-thresh* model, defined as follows:
- Denote by $\mu_i(\vec{q})$ the multiplicity of state i in $\vec{q}$.
- *Atoms* are logical statements in the unqualified variable $\vec{q}$.
  - *mod atom:* "$\mu_i(\vec{q}) \equiv r \pmod{m}$"
  - *thresh atom:* "$\mu_i(\vec{q}) < t$"
- A function $f(\vec{q}) : Q^* \to R$ is *mod-thresh* if it can be expressed using these atoms, `and`, `or`, `not`, `if-then-else`.
- E.g., "if there are an even number of A inputs and at least 4 B inputs, then return 1, else return 2."
- A mod-thresh function is automatically symmetric, since it depends on the input $\vec{q}$ only via the symmetric multiplicity functions μ_i.
- In proceedings we show that Sequential $\subseteq$ Mod-Thresh $\subseteq$ Parallel $\subseteq$ Sequential, essentially via simulation arguments.
- Recent work (on website, not in paper): a Sequential automaton can be simulated by a Parallel one *without increasing the memory requirements.*

Sequential-Parallel Equivalence

- Main technical contribution: we show the sequential and parallel models are equivalent. Proof uses the *mod-thresh* model, defined as follows:
- Denote by $\mu_i(\vec{q})$ the multiplicity of state i in $\vec{q}$.
- *Atoms* are logical statements in the unqualified variable $\vec{q}$.
  - *mod atom:* "$\mu_i(\vec{q}) \equiv r \pmod{m}$"
  - *thresh atom:* "$\mu_i(\vec{q}) < t$"
- A function $f(\vec{q}) : Q^* \to R$ is *mod-thresh* if it can be expressed using these atoms, `and`, `or`, `not`, `if-then-else`.
- E.g., "if there are an even number of A inputs and at least 4 B inputs, then return 1, else return 2."
- A mod-thresh function is automatically symmetric, since it depends on the input $\vec{q}$ only via the symmetric multiplicity functions μ_i.
- In proceedings we show that Sequential $\subseteq$ Mod-Thresh $\subseteq$ Parallel $\subseteq$ Sequential, essentially via simulation arguments.
- Recent work (on website, not in paper): a Sequential automaton can be simulated by a Parallel one *without increasing the memory requirements.*

Sequential-Parallel Equivalence

- Main technical contribution: we show the sequential and parallel models are equivalent. Proof uses the *mod-thresh* model, defined as follows:
- Denote by $\mu_i(\vec{q})$ the multiplicity of state i in $\vec{q}$.
- *Atoms* are logical statements in the unqualified variable $\vec{q}$.
  - *mod atom:* "$\mu_i(\vec{q}) \equiv r \pmod{m}$"
  - *thresh atom:* "$\mu_i(\vec{q}) < t$"
- A function $f(\vec{q}) : Q^* \to R$ is *mod-thresh* if it can be expressed using these atoms, `and`, `or`, `not`, `if-then-else`.
- E.g., "if there are an even number of A inputs and at least 4 B inputs, then return 1, else return 2."
- A mod-thresh function is automatically symmetric, since it depends on the input $\vec{q}$ only via the symmetric multiplicity functions μ_i.
- In proceedings we show that Sequential $\subseteq$ Mod-Thresh $\subseteq$ Parallel $\subseteq$ Sequential, essentially via simulation arguments.
- Recent work (on website, not in paper): a Sequential automaton can be simulated by a Parallel one *without increasing the memory requirements*.

Joining Multi-Input Automata Together

(Up to the "high" level)

- Let an *FSM function* (finite-state, symmetric, multi-input) mean a sequential/parallel/mod-thresh function.
- Definition of FSSGA (finite-state symmetric graph automata) model: let Q be a finite set of states and for each $q \in Q$ let $f[q] : Q^* \to Q$ be an FSM function.
- When a node activates, let q be its state and let $\vec{q}$ be a list of its neighbours' states; that node's new state is $f[q](\vec{q})$.
- Gives local and global symmetry as desired without need to bound degree.
- We can define both a synchronous model and an asynchronous model.
- Note, message complexity is not meaningful for our model as every neighbouring pair of nodes exchange information (their states) every round.

Joining Multi-Input Automata Together

(Up to the "high" level)

- Let an *FSM function* (finite-state, symmetric, multi-input) mean a sequential/parallel/mod-thresh function.
- Definition of FSSGA (finite-state symmetric graph automata) model: let Q be a finite set of states and for each $q \in Q$ let $f[q] : Q^* \to Q$ be an FSM function.
- When a node activates, let q be its state and let $\vec{q}$ be a list of its neighbours' states; that node's new state is $f[q](\vec{q})$.
- Gives local and global symmetry as desired without need to bound degree.
- We can define both a synchronous model and an asynchronous model.
- Note, message complexity is not meaningful for our model as every neighbouring pair of nodes exchange information (their states) every round.

Simple Example FSSGA Algorithm: 2-Coloring

- Take $Q = \{\mathcal{BLANK}, \mathcal{RED}, \mathcal{BLUE}, \mathcal{FAILED}\}$. Either 2-color a graph if possible, or report $\mathcal{FAILED}$ at every node if impossible.
- Initially, one node is in state $\mathcal{RED}$, all others are in state $\mathcal{BLANK}$. Each $f[q]$ is as follows:

if $\neg(\mu_{\mathcal{FAILED}}(\vec{q}) < 1)$ **then** return $\mathcal{FAILED}$
else if $\neg(\mu_{\mathcal{RED}}(\vec{q}) < 1) \wedge \neg(\mu_{\mathcal{BLUE}}(\vec{q}) < 1)$ **then** return $\mathcal{FAILED}$
else if $\neg(\mu_{\mathcal{RED}}(\vec{q}) < 1)$ **then** return $\mathcal{BLUE}$
else if $\neg(\mu_{\mathcal{BLUE}}(\vec{q}) < 1)$ **then** return $\mathcal{RED}$
else return $\mathcal{BLANK}$

Random Walk

- In proceedings: straightforward probabilistic extension of the FSSGA model.
- "Random walk" a useful local symmetry-breaking primitive (e.g., in leader election).
- Usual random walk description, "send the agent to a random neighbour," does not apply since a node can't directly affect its neighbours' states, nor can it pick from an arbitrarily large set at random.
- Instead, we run a local election. To decide next location, agent asks neighbours to flip coins. Tails remain eligible, heads are eliminated.
- Keep running coin flip rounds until agent determines that exactly one neighbour remains. (If everyone's eliminated in a given round then you re-run the previous round).
- With high probability, takes $\Theta(\log degree(v))$ rounds to move agent away from node v.

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathrm{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathrm{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathrm{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathrm{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathrm{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathsf{Diam} \log n)$.
- Java demo!

More Algorithms

- We can also implement:
- α synchronizer. Neighbours keep clocks at most within ± 1 of each other. Implement with mod-3 clocks.
- Breadth-first search. We use mod-3 distance labels (like the mod-3 clocks of the synchronizer).
- Network traversal. A very nice (but sadly obscure) algorithm by [Milgram, 1975] allows an agent to traverse a scan-first-search tree of the network (DFS-BFS hybrid). [Note; we actually can't explicitly store any tree in the network, due to the symmetry and finiteness-of-state. In fact, not only is it impossible for node to identify a "parent" neighbour, a node can't count how many neighbours it has.]
- Greedy graph traversal (more fault-tolerant than Milgram's, but slower).
- Leader election. Uses a handful of known techniques, $O(n \log n)$ running time, probably can do $O(\mathsf{Diam} \log n)$.
- Java demo!

Outline

Passive Mobility ("Birds") Model

- Angluin, Aspnes, Diamadi, Fischer, and Peralta have a model of "passively mobile" sensors.
- Idea: the network is composed of n entities that interact pairwise.
- Repeatedly select a pair of entities, say with states $[q_1, q_2]$, and replace their states by $[p_1(q_1, q_2), p_2(q_1, q_2)]$.
- This models sensors attached to a "flock of birds:" there is a lot of interaction but the order of operations is largely unpredictable.
- Similarities to "low model:" exact power was recently characterized in PODC 2006 (semilinear vs. our mod-thresh).
- Similarities to "high model:" motivated by unpredictable networks, symmetry.

Open Algorithmic Problems

- *Firing Squad*: every node in a synchronous network must "fire" exactly once in the future, and all at the same time.
- To avoid trivial solutions, we demand that if there is no "admiral" present in the network at the beginning, then nobody should fire at all.
- Long history of solutions in path/grid graphs, but in general graphs, seemingly all solutions work by embedding a spanning path in the graph.
- Cannot embed a path in FSSGA model (impossible to identify any one neighbour). Is there any firing squad algorithm?
- *Self-stabilizing Leader Election:* an algorithm is self-stabilizing if it is eventually correct despite any finite number of initial failures.
- Self-stabilizing leader election would allow other FSSGA algorithms to be made self-stabilizing, but no existing algorithms seem to be adaptable to this situation.

Open Algorithmic Problems

- *Firing Squad*: every node in a synchronous network must "fire" exactly once in the future, and all at the same time.
- To avoid trivial solutions, we demand that if there is no "admiral" present in the network at the beginning, then nobody should fire at all.
- Long history of solutions in path/grid graphs, but in general graphs, seemingly all solutions work by embedding a spanning path in the graph.
- Cannot embed a path in FSSGA model (impossible to identify any one neighbour). Is there any firing squad algorithm?
- *Self-stabilizing Leader Election:* an algorithm is self-stabilizing if it is eventually correct despite any finite number of initial failures.
- Self-stabilizing leader election would allow other FSSGA algorithms to be made self-stabilizing, but no existing algorithms seem to be adaptable to this situation.

Non-Finite State Models

(Back to the "low" level)

- What if we allow the node states and working states to be binary tapes and not just elements of a finite state space?
- Sequential model bears resemblance to online and streaming algorithms. Parallelism: could have a network where multiple "synopses" circulate and combine.
- We have shown in finite-state: if we want to compute a function of an arbitrary number of inputs, then provided only that our desired function is symmetric, parallelism is just as powerful as the sequential model.
- Could this possibly extend to non-finite state models? Seems that general Turing machines are much harder to work with than FSAs. But we can't seem to find a counterexample yet.
- **Thanks for listening!**

Non-Finite State Models

(Back to the "low" level)

- What if we allow the node states and working states to be binary tapes and not just elements of a finite state space?
- Sequential model bears resemblance to online and streaming algorithms. Parallelism: could have a network where multiple "synopses" circulate and combine.
- We have shown in finite-state: if we want to compute a function of an arbitrary number of inputs, then provided only that our desired function is symmetric, parallelism is just as powerful as the sequential model.
- Could this possibly extend to non-finite state models? Seems that general Turing machines are much harder to work with than FSAs. But we can't seem to find a counterexample yet.
- **Thanks for listening!**

Non-Finite State Models

(Back to the "low" level)

- What if we allow the node states and working states to be binary tapes and not just elements of a finite state space?
- Sequential model bears resemblance to online and streaming algorithms. Parallelism: could have a network where multiple "synopses" circulate and combine.
- We have shown in finite-state: if we want to compute a function of an arbitrary number of inputs, then provided only that our desired function is symmetric, parallelism is just as powerful as the sequential model.
- Could this possibly extend to non-finite state models? Seems that general Turing machines are much harder to work with than FSAs. But we can't seem to find a counterexample yet.
- **Thanks for listening!**