Symmetric Network Computation

David Pritchard
Department of Combinatorics and Optimization
University of Waterloo
Waterloo, ON, Canada
dagpritc@math.uwaterloo.ca

Santosh Vempala
Department of Mathematics
MIT
Cambridge, MA, USA
vempala@math.mit.edu

ABSTRACT

We introduce a simple new model of distributed computation — finite-state symmetric graph automata (FSSGA) — which captures the qualitative properties common to fault-tolerant distributed algorithms. Roughly speaking, the computation evolves homogeneously in the entire network, with each node acting symmetrically and with limited resources. As a building block, we demonstrate the equivalence of two automaton models for computing symmetric multi-input functions. We give FSSGA algorithms for several well-known problems.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata, relations between models*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*

General Terms

Algorithms, Reliability, Theory

Keywords

Symmetry, fault-tolerance, agents, election

1. INTRODUCTION

Distributed algorithms play a fundamental role in computer science. In recent years, practical developments such as sensor networks further motivate such algorithms, while introducing restrictions on the resources of each node. For example, Angluin et al. [1] have modeled a sensor network by an interacting collection of identical finite-state agents. In this paper, we present a model of distributed computation whose goal is to foster fault-tolerant computation.

We consider *decreasing benign* faults: a node or edge may permanently be deleted from the graph because it malfunctions, but nodes and edges never join the network, and there is no malicious behaviour. Many simple distributed algorithms cannot tolerate even a single fault. For example, a spanning tree-based algorithm (like the β synchronizer [2]) fails if one of the tree edges dies, since then not all nodes can communicate along the remainder of the tree.

Our starting point is the observation that the following properties are common to many fault-tolerant algorithms:

(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.

We might call an algorithm that follows these three principles a *balancing* algorithm. Each node ensures that a local balancing rule is satisfied when it activates, and when the whole graph is in equilibrium the algorithm is complete. Faults may cause a temporary loss of balance, but as the nodes iterate their operation, balance is restored in the network.

Flajolet and Martin's census algorithm [6] provides a good illustration of these principles. The algorithm approximately computes the number of nodes in a network of unknown size. Hereafter let $n = |V|$, the number of nodes in the network. Each node v has a k-bit vector $v.m$ of memory; denote the ith bit by $v.m_i$, where $1 \leq i \leq k$. The algorithm requires $k \geq \log_2 n$. Initially all memory is set to 0. Next, each node v probabilistically performs one action: for $1 \leq i \leq k$ with probability 2^{-i}, it sets $v.m_i$ to 1, and with probability 2^{-k} it does nothing. In the remainder of the algorithm, each node repeatedly sends its memory contents to all of its neighbours. Whenever v receives message $w.m$ from its neighbour w, it sets $v.m := v.m \,\texttt{OR}\, w.m$. After stabilizing, each node estimates $n = 1.3 \cdot 2^{\ell}$ where ℓ is the minimum index of a 0 bit in its memory. It can be shown that when no failures occur, with high probability, the estimate is correct within a factor of 2. The correctness is clearly unaffected by edge faults that do not disconnect the network; this is essentially optimal, considering the fundamental impossibility of complete communication in any disconnected network. Furthermore, even if some small parts of the original network $\mathcal{G}$ become disconnected, for any connected component $\mathcal{G}'$ of the final network, with high probability, the nodes in $\mathcal{G}'$ obtain an estimate between $\frac{1}{2}|V(\mathcal{G}')|$ and $2|V(\mathcal{G})|$.

SPAA'06, July 30–August 2, 2006, Cambridge, Massachusetts, USA.

In Section 2, we define the notion of a *k-sensitive* algorithm, which generalizes the fault-tolerance of the above algorithm. Roughly speaking, for a k-sensitive algorithm, at any point in the computation there are at most k critical nodes, and failures at noncritical nodes are harmless. Usually, *decentralized* algorithms (e.g., [8] [10]) have sensitivity 0, *agent-based* algorithms (see Section 2.1) have sensitivity 1, and tree-based algorithms have sensitivity $\Theta(n)$. Thus, ranking algorithms by their sensitivity, the decentralized paradigm provides the most fault tolerance. This motivated our choices (P1–P3) of key properties. In Sections 2.1 and 2.2 we show two more fault-tolerant algorithms from our study: random walk-based biconnectivity and distributed shortest paths.

In Section 3, we present our main contribution, a precise model of symmetric computation with limited resources. In brief, we imagine that each node of a graph has a copy of the same finite-state automaton. Indeed, from (P1) above, one would like the transition function at each node to be the same and further, from (P2) it should be symmetric. Such symmetric models have been considered before, e.g., cellular automata (Conway's "Life" [7]) but usually assume that the graph is regular or has bounded degree. In our model, we retain the symmetry but allow unbounded degrees. Our model was thus designed to have the following properties:

(S0) An automaton with finite memory inhabits each node, using its neighbours' states as inputs.

(S1) All nodes, even those with different degrees, are inhabited by identical automata.

(S2) Each automaton acts symmetrically on its neighbours.

We thought of two models whereby each node would itself use a constant amount of working space no matter how many neighbours are to be processed. In the *sequential* model, when a node activates, it treats its neighbours as a sequence of inputs, and one by one they are fed into the automaton's transition function. In the *parallel* model, the neighbours are processed via a divide-and-conquer approach. Each neighbour contributes a single unit of data, and then the data are reduced pairwise. After all the data have been combined, a state transition occurs. Of the automata in these two classes, we are interested in those that also satisfy (S2). The main technical contribution of our paper is a proof that the sequential and parallel versions are in fact equivalent and can be characterized explicitly in terms of *mod* and *threshold* operations.

In Section 4 we give a number of algorithms for our model. As nodes have finite state but unbounded degree, a node cannot even count its neighbours, and yet in Section 4.7 we show that randomized leader election can be efficiently implemented. This leads us to believe that our simple model is both practical (there are limited resources per node and nontrivial problems can be solved) and interesting (it has multiple formulations). Despite these features, we are not completely satisfied. One of our initial hopes was that the model's local symmetry would imply decentralization and fault-tolerance for all algorithms meeting the model. Unfortunately, this is not the case, and indeed the leader election algorithm shows that global symmetry-breaking is still possible.

In Section 5 we discuss other issues related to our model. We show how the *isotonic web automaton* model [19][14] can simulate our model (with a $\Theta(m)$ factor slowdown) and vice-versa. We note three other relevant models here. First, the class of *semi-lattice* [16] (or *infimum* [23, §6.1.5]) functions essentially provide the automatic fault-tolerance we desire, but these functions are limited in their scope. One example of a semi-lattice function is the iterated `OR` of the Flajolet-Martin algorithm. Second, the *parallel web automaton* model [18] is close in spirit to our model. In that model every node and directed edge is an automaton; each node reads its incident edges symmetrically and each edge reads its two endpoints asymmetrically. However, that model is not completely formalized and so no direct comparison is possible. Third, we mentioned the "passive mobility" model of Angluin et al. [1]; in that model all interactions occur in asymmetric pairs, while in our model, nodes communicate symmetrically, and with all neighbours at once.

2. K-SENSITIVE ALGORITHMS

For a distributed algorithm, let χ be a deterministic function whose input is the instantaneous description σ of the state of a connected network, and whose output is a subset $\chi(\sigma)$ of its nodes, called the *critical nodes*. In an execution of the algorithm, when the network is in state σ, a *critical failure* is either the failure of a node in $\chi(\sigma)$, or a node/edge failure that causes two nodes of $\chi(\sigma)$ to lie in different connected components of the network. If we always have $|\chi(\sigma)| \leq k$, and if the algorithm is always "reasonably correct" provided that no critical failures occur, then we call the algorithm *k-sensitive*. Since a k-sensitive algorithm is automatically $(k+1)$-sensitive, define the *sensitivity* of an algorithm to be the least k for which it is k-sensitive.

Hereafter, we write $\mathcal{G}$ for a graph that models our distributed network, $V(\mathcal{G})$ for its nodes, and $E(\mathcal{G})$ for its edges; we simply write V and E when the meaning of $\mathcal{G}$ is clear. Our definition of "reasonably correct" is as follows. Consider a run of the algorithm where f failures occur, none of them critical. Let $\mathcal{G}_0$ be the initial network topology. When the ith failure occurs, let σ_i be the current state of $\mathcal{G}_{i-1}$, and let $\mathcal{G}_i$ be a connected component of $\mathcal{G}_{i-1}$ that contains all of $\chi(\sigma_i)$. Let the final answer computed in the nodes of $\mathcal{G}_f$ be A. We say that the algorithm was *reasonably correct* in this execution if there is some graph $\mathcal{G}'$ with $\mathcal{G}_0 \supseteq \mathcal{G}' \supseteq \mathcal{G}_f$ such that executing the algorithm on $\mathcal{G}'$ in a fault-free environment gives the same answer A.

The algorithms from the introduction exhibit two typical sensitivity values. The tree-based β synchronizer has sensitivity $\Theta(n)$, as a spanning tree may have $n/2$ internal nodes, and the failure of any one disconnects the tree. In contrast, the Flajolet-Martin algorithm is 0-sensitive, as it will work on whatever portion of the network remains connected. We describe two more low-sensitivity algorithms in the remainder of this section.

2.1 Biconnectivity via a Random Walk

An *agent* is an entity that inhabits one node of the network at a time. An agent at v can move to w in one step if and only if v and w are adjacent in $\mathcal{G}$. Agent algorithms often have small sensitivity; in this section and in Section 4.6 we give agent algorithms with sensitivity $\Theta(1)$.

A *bridge* of a connected graph is an edge whose deletion separates the graph. We will describe a simple agent-based algorithm for determining the bridges of a graph. First, fix an arbitrary orientation on each edge. Each edge stores an

integral counter, initialized to zero. Whenever the agent traverses an edge in agreement with that edge's orientation, increment its counter by 1; whenever the agent traverses that edge the other way, decrement its counter by one.

It is easy to show that the counter for a bridge will always remain in $\{-1, 0, 1\}$. On the other hand, the counter of any non-bridge may exceed ± 1 if the agent takes a suitable walk. In fact, if the agent takes a random walk — at each step, it picks its next position uniformly at random from the neighbours of its current position — then we can show that all non-bridges will be quickly identified. Let $n = |V|$ and $m = |E|$. The following complexity analysis ignores failures.

CLAIM 2.1. *If an edge is not a bridge, then the expected number of steps before its counter exceeds 1 in absolute value is $O(mn)$.*

PROOF. Write $V = \{v_1, \ldots, v_n\}$ and let the edge be $e = (v_1, v_2)$, oriented towards v_2. Write c for the value of e's counter. We construct a new graph. It has $3n + 1$ nodes: three labeled v_i^{-1}, v_i^0, v_i^1 for each $v_i \in V$, plus the special node EXCEEDED. The idea is that v_i^r corresponds to a state where $c = r$ and the agent is at node v_i, while the node EXCEEDED corresponds to a state where $c = \pm 2$. Specifically, this new graph has $3m + 1$ undirected edges in total:

$(v_i^r, v_{i'}^r)$ for each $r \in \{-1, 0, 1\}$ and $(v_i, v_{i'}) \in E - \{(v_1, v_2)\}$

as well as

$$(v_1^{-1}, v_2^0), (v_1^0, v_2^1), (v_1^1, \text{EXCEEDED}), (\text{EXCEEDED}, v_2^{-1}).$$

It is straightforward to show that a random walk on the new graph corresponds to the original process on the old graph.

Since (v_1, v_2) is not a bridge, we can reach any v_i^r from v_1^0: first, if $r \neq 0$, then traverse a cycle containing (v_1, v_2) to set c correctly; second, walk to v_i without using (v_1, v_2). Thus, the new graph is connected. By applying the hitting time bound for an undirected graph [15, p. 137], we expect to reach EXCEEDED in at most $2(3m + 1)(3n) = O(mn)$ steps. □

To make a bridge-finding algorithm, we make each edge remember if its counter has ever hit ± 2. If the agent walks for $O(cmn \log n)$ steps, then with probability $1 - n^{1-c}$, all non-bridges of the graph will have been identified. In terms of sensitivity, failures at non-agent nodes are unimportant, so we may define $\chi(\sigma)$ to output just the agent's position in σ. Hence this algorithm is 1-sensitive.

2.2 Shortest Paths and Clustering

Fix a set of nodes T in the network. There is a decentralized algorithm by which each node can determine its distance to T. Each node v stores a single integer variable $\ell(v)$ which will, at termination, hold the distance from that node to the nearest node in T. Each node in T fixes its label at 0. When any other node v activates, it sets its label to 1 more than the minimum of its neighbours' labels:

$$\ell(v) := 1 + \min_{(v,u) \in E(\mathcal{G})} \ell(u).$$

It is straightforward to show that a node v at distance d from T will have its label stabilize at d, within d rounds. Practically, we should also cap each label at n in case it happens that some connected component contains no node of T. This algorithm can be shown to be 0-sensitive.

These labels implicitly define shortest paths to T. As an application, consider a sensor network where most nodes have no permanent storage and T represents "data sinks." If each node routes packets to a minimum-label neighbour, then every packet traverses a shortest path to the nearest sink.

3. A FORMAL MODEL BASED ON FINITE-STATE AUTOMATA

The starting point for our model is what Tel [23, p. 524] calls *read-all state* communication. Each node *activates* at certain times. When a node activates, it atomically reads its own state and the states of its neighbours, and its new state is determined by those inputs. We note that this model can simulate the ubiquitous message-passing model, by using message buffers.

3.1 Symmetric Multi-Input Finite-State Automata

In this section, we define a class of symmetric functions that take in any number of arguments. These functions have three equivalent descriptions, two of which are automaton-based. Our new model of distributed computing will be introduced in Section 3.4 but is essentially as follows. Given an undirected, connected graph, we replace each node with a copy of the same automaton, and the inputs for a given node are the neighbours of that node.

To keep it simple, for each algorithm in our model, all nodes' states will be drawn from a finite set Q. A *network state* (or *instantaneous description* [14]) is a function from V to Q. We denote by σ the current network state, so $\sigma(v)$ represents the current state of node v. Let Q^+ denote the set of sequences, of any positive length, with elements drawn from Q. We use $|\vec{q}|$ to denote the number of elements in the sequence $\vec{q}$, and write $\vec{q} = (q_1, \ldots, q_{|\vec{q}|})$ for an arbitrary element of Q^+.

Motivated by (P1), we would like every node to use the same transition function. Now, if our graph is regular of degree Δ, then the transition function could be described as a function of the form $f : Q \times Q^\Delta \to Q$. Namely, when a node v with neighbours $u_1, \ldots, u_\Delta$ operates, set

$$\sigma(v) := f(\sigma(v), (\sigma(u_1), \ldots, \sigma(u_\Delta))). \tag{1}$$

From (P2), the transition function should be symmetric. Thus we would require $f(q_0, \vec{q}) = f(q_0, \pi(\vec{q}))$ for all permutations $\pi \in S_\Delta$.

When the graph is not regular, we need to modify the transition function to take in a variable number of neighbours. If we only wish to use network topologies where each node has degree at *most* Δ, then we can generalize the automaton described by Equation (1) as follows. Introduce a special "null" symbol ϵ. In Equation (1), when a node v of degree $d < \Delta$ activates, we take $\sigma(u_{d+1}) = \cdots = \sigma(u_\Delta) = \epsilon$. Thus $f : Q \times (Q \cup \{\epsilon\})^\Delta \to Q$. See [17][12][21] for similar bounded-degree models.

For our new model, we did not want to restrict our attention to bounded-degree graphs. Note that, if there are a finite number of states, and unbounded degrees, then generally a node cannot even count its neighbours. Some "web automaton" models [19] [14] [18] similarly allow unbounded degrees but enforce symmetry restrictions.

The transition function for the graph automata which we

are describing operates as $Q \times Q^+ \to Q$, that is, the first argument is the current state of the activating node, the second argument is the collection of its neighbours' states, and the output is the new state of that node. The essential feature of our model (recall S0-S2) is that the nodes act symmetrically, and are all the same, but take in differing numbers of arguments. To narrow our discussion, we ignore the first input for the time being.

DEFINITION 3.1. *Suppose* $|Q|, |R| < \infty$. *Let* $f : Q^+ \to R$ *be such that for all* $\vec{q}$, *where* $|\vec{q}| = k$, *and for all* $\pi \in S_k$,

$$f(q_1, \ldots, q_k) = f(q_{\pi(1)}, \ldots, q_{\pi(k)}).$$

Then f *is a* SM function.

Here SM stands for "symmetric, multi-input." In our application to graph automata we will have $R = Q$, and Q will be the set of node states.

3.2 Sequential and Parallel Automata for SM Functions

A sequential SM function from Q to R is defined by a tuple (W, w_0, p, β). Here W is a finite set of *working states,* $w_0 \in W$ is a distinguished starting state, $p : W \times Q \to W$ is a *processing* function, and $\beta : W \to R$ maps the final working state back to a result in R. Using this tuple we define a function from Q^+ to R as follows. Initialize a "working state" variable w to w_0. Then, for each input q_i, compute $w := p(w, q_i)$. Finally, output $\beta(w)$ after all inputs have been processed. Keeping in mind (S2) and the definition of an SM function, if the final value $\beta(w)$ is independent of the ordering of the inputs, then this process defines a sequential SM function. A formal definition follows.

DEFINITION 3.2. *Suppose that we have* $|W| < \infty, w_0 \in W, p : W \times Q \to W$, *and* $\beta : W \to R$. *Suppose further that for all* $\vec{q} \in Q^+$, *where* $|\vec{q}| = k$, *for all* $\pi \in S_k$, *the expression*

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

is independent of π. *Then the function* $f : Q^+ \to R$ *which maps* $\vec{q}$ *to Equation* (2) *is defined to be a* sequential SM function.

We call (W, w_0, p, β) a *sequential program* for f.

The second form of finite-state symmetric processing we consider uses the divide-and-conquer paradigm. Take a finite set of working states W and $\beta : W \to R$ as before but instead of the distinguished state w_0 we have a function $\alpha : Q \to W$. On input $\vec{q}$ of length k, we turn each input q_i into its own working state $\alpha(q_i)$. This defines a multiset $\mathcal{W}$ of working states. Then, as long as $\mathcal{W}$ contains at least two states, we remove two states w_1, w_2 from $\mathcal{W}$ and add $p(w_1, w_2)$ to $\mathcal{W}$. Thus, we now have $p : W \times W \to W$. Finally, when $\mathcal{W}$ contains only one state w, we return $\beta(w)$. One might visualize the combination process as a tree, as shown in Figure 1. In order that the function be well-defined and symmetric, we insist that the final result is independent of the order in which elements are combined.

As the tree formulation is somewhat more concise, we make the following definitions. For a rooted binary tree T on more than one node, we write $T.\ell$ for the left subtree of T, we write $T.r$ for the right subtree of T, and we write $T.root$ for the root node of T.

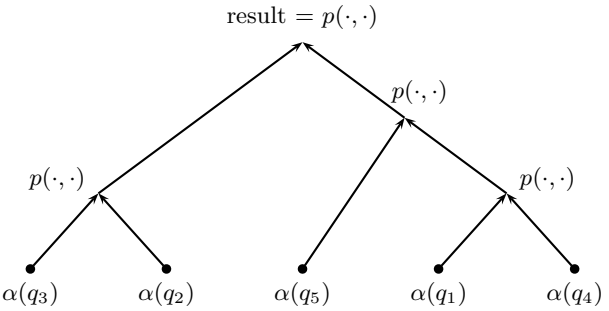

Figure 1: Visualizing a parallel SM automaton as a tree process.

DEFINITION 3.3. *Suppose that* T *is a rooted binary tree with* k *leaves. Label the leaves from leftmost to rightmost as* $t_1, \ldots, t_k$. *Let* $p : W \times W \to W$. *For each non-empty subtree* S *of* T, *recursively define the function* $c^S : W^k \to W$ *by*

$$c^S(\overrightarrow{w}) = \begin{cases} w_i, & \text{if } S.root = t_i; \\ p(c^{S.\ell}(\overrightarrow{w}), c^{S.r}(\overrightarrow{w})), & \text{otherwise.} \end{cases}$$

Then we define the tree-combination *of* p *on* T, *denoted* $TC^{(p,T)}$, *to be* c^T.

DEFINITION 3.4. *Suppose that we have* $|W| < \infty, \alpha : Q \to W, p : W \times W \to W$, *and* $\beta : W \to R$. *Suppose further that for all* $\vec{q} \in Q^+$, *where* $|\vec{q}| = k$, *for all* $\pi \in S_k$, *and for all rooted binary trees* T *with* k *leaves, the expression*

$$\beta(TC^{(p,T)}(\alpha(q_{\pi(1)}), \alpha(q_{\pi(2)}), \ldots, \alpha(q_{\pi(k)}))) \tag{3}$$

is independent of π *and* T. *Then the function* $f : Q^+ \to R$ *which maps* $\vec{q}$ *to Equation* (3) *is defined to be a* parallel SM function.

We call (W, α, p, β) a *parallel program* for f, similarly to before.

The following lemma essentially says that, if we know how to solve a problem by divide-and-conquer, we can simply conquer one input at a time and solve it sequentially.

LEMMA 3.5. *Every parallel SM function can be written as a sequential SM function.*

PROOF. Consider a parallel SM function with parallel program (W, α, p, β). Then there is a sequential program (W', w_0, p', β) that computes the same function, defined by

$$W' = W \cup \{NIL\};$$
$$w_0 = NIL;$$
$$p' : (w, q) \mapsto \begin{cases} \alpha(q), & \text{if } w = NIL, \\ p(\alpha(q), w), & \text{otherwise.} \end{cases}$$

□

3.3 Mod-Thresh Functions

Surprisingly, the converse of Lemma 3.5 is true: every sequential SM function can be written as a parallel SM function. Thus, regarded as computing devices, both models are equally powerful. Our proof proceeds by showing that both classes are equivalent to the set of *mod-thresh* functions, which we define below. The mod-thresh model is more in the style of a programming language, giving a more intuitive description of sequential/parallel SM functions.

Write $s = |Q|$, and without loss of generality let $Q = \{1, 2, \ldots, s\}$. Denote by $\mu_i(\vec{q})$ the multiplicity of i in $\vec{q}$. We need to define two kinds of boolean atoms. Each atom is a logical statement in the unqualified variable $\vec{q}$. A *mod atom* is of the form "$\mu_i(\vec{q}) \equiv r \pmod m$," where $0 \leq k < m$ are integers and $i \in Q$. A *thresh atom* is of the form "$\mu_i(\vec{q}) < t$," where t is a positive integer and $i \in Q$. The set of *mod-thresh propositions* is the closure, under (finite) logical conjunction, disjunction, and negation, of the union of all mod atoms and all thresh atoms.

DEFINITION 3.6. *Let $P_1, \ldots, P_{c-1}$ be mod-thresh propositions, and $r_1, \ldots, r_c$ be elements of R, not necessarily distinct. The function $f : Q^+ \to R$ described procedurally by*

***procedure** $f(\vec{q})$*
  ***if** P_1 is true **then** return r_1*
  ***else if** P_2 is true **then** return r_2*
    $\cdots$
  ***else** return r_c*
  end if
end procedure

is a mod-thresh SM function.

We call $(P_1, \ldots, P_{c-1}; r_1, \ldots, r_c)$ a *mod-thresh program* for f. Note that a mod-thresh function is automatically symmetric since it depends on $\vec{q}$ only via the symmetric functions μ_i. Also note that there is another, quite different, proposition-based model of distributed computing in [4].

THEOREM 3.7. *The classes of mod-thresh, parallel, and sequential SM functions are all the same.*

PROOF. Let Sequential denote the class of sequential SM functions, Parallel denote the class of parallel SM functions, and Mod-Thresh denote the class of mod-thresh SM functions. We will demonstrate that Mod-Thresh $\subseteq$ Parallel $\subseteq$ Sequential $\subseteq$ Mod-Thresh. The second inclusion follows from Lemma 3.5.

LEMMA 3.8. *Mod-Thresh $\subseteq$ Parallel*

PROOF. Let f be any mod-thresh SM function, with program $MT = (P_1, \ldots, P_{c-1}; r_1, \ldots, r_c)$. We demonstrate a parallel program for f. Essentially, the multiplicity counts needed to determine the outcome of MT are evaluated in a divide-and-conquer fashion.

For each state $i \in Q$ define the integers M_i and T_i by

$$M_i := \operatorname{lcm}\Big(\{1\} \cup \bigcup_{j=1}^{c-1} \bigcup_{r \geq 0} \{m : P_j \ni \text{“}\mu_i(\vec{q}) \equiv r \pmod m\text{”}\}\Big),$$

$$\text{and } T_i := \max\Big(\{1\} \cup \bigcup_{j=1}^{c-1} \{t : P_j \ni \text{“}\mu_i(\vec{q}) < t\text{”}\}\Big).$$

In order to evaluate $f(\vec{q})$ for a given $\vec{q}$, it suffices to know the value of each $\mu_i(\vec{q}) \pmod{M_i}$, and whether $\mu_i(\vec{q}) < n$ for each $0 \leq n \leq T_i$, since from this information each of the atoms can be evaluated. Thus, our working state will consist of two finite-state counters for each $i \in Q$.

With δ_x^y the Dirac delta, define

$$\begin{aligned}
W &= \bigotimes_{i \in Q} \{0, 1, \ldots, M_i - 1\} \times \{0, 1, \ldots, T_i - 1, \infty\}, \\
\alpha &: q \mapsto \bigotimes_{i \in Q} (\delta_q^i, \delta_q^i), \\
p &: \bigotimes_{i \in Q} (a_i, b_i), \bigotimes_{i \in Q} (a_i', b_i') \mapsto \bigotimes_{i \in Q} (a_i + a_i', b_i + b_i'),
\end{aligned}$$

where the addition $a_i + a_i'$ is performed modulo M_i, and the addition $b_i + b_i'$ produces ∞ if the result is greater than or equal to T_i.

Finally, we need to define β. For each $w = \bigotimes_{i \in Q}(a_i, b_i) \in W$, replace each atom "$\mu_i(\vec{q}) \equiv r \pmod m$" in MT with the boolean value of $(a_i \equiv r \pmod m)$, and replace each atom "$\mu_i(\vec{q}) < t$" in MT with the boolean value of $(b_i < t)$. Then the result of MT on w can be determined and so this defines $\beta(w)$. □

The final containment is the most involved. Here $g^{(a)}$ denotes the ath iterate of g.

LEMMA 3.9. *Sequential $\subseteq$ Mod-Thresh*

PROOF. Fix a sequential function f and denote its program by (W, w_0, p, β). We will show that for each state $j \in Q$, the value of $f(\vec{q})$ depends on $\mu_j(\vec{q})$ in a "mod-thresh way."

In the computation of $f(\vec{q})$ by the sequential program, suppose that we process those items of $\vec{q}$ which are equal to j first. This partial processing brings the working state w to

$$w = p(p(p(\cdots p(p(w_0, j), j) \cdots, j), j), j),$$

where p is applied $\mu_j(\vec{q})$ times. We could also write this as $w = g_j^{(\mu_j(\vec{q}))}(w_0)$, where $g_j : x \mapsto p(x, j)$. However, the fact that the space W of working sets is finite means that the iterated image of w_0 under g_j is "eventually periodic." To be precise, there are integers t_j and m_j such that for all z_1, z_2 such that $z_1 \geq t_j, z_2 \geq t_j$, and $z_1 \equiv z_2 \pmod{m_j}$, we have $g_j^{(z_1)}(w_0) = g_j^{(z_2)}(w_0)$.

For $j \in Q$, define $\sim_j$ to be the equivalence relation on $\{n \in \mathbb{Z} : n \geq 0\}$ with the $(t_j + m_j)$ equivalence classes

$$\{i\}, 0 \leq i < t_j \text{ and } \{n \geq t_j : n \equiv i \,(\operatorname{mod} m_j)\}, 0 \leq i < m_j.$$

Note that mod-thresh propositions can determine the equivalence class of $\sim_j$ that contains $\mu_j(\vec{q})$. Specifically we have

$$\mu_j(\vec{q}) \in \{i\} \Leftrightarrow \text{“}(\mu_j(\vec{q}) < (i+1)) \wedge \neg(\mu_j(\vec{q}) < i)\text{”} \tag{4}$$

and

$$\begin{aligned}
&\mu_j(\vec{q}) \in \{n \geq t_j : n \equiv i \,(\operatorname{mod} m_j)\} \\
&\Leftrightarrow \text{“}\neg(\mu_j(\vec{q}) < t_j) \wedge (\mu_i(\vec{q}) \equiv r \,(\operatorname{mod} m_j))\text{.”}
\end{aligned} \tag{5}$$

For any j, consider $\vec{q}$ and $\vec{q}\,'$ such that $\mu_i(\vec{q}) = \mu_i(\vec{q}\,')$ for all $i \neq j$, and $\mu_j(\vec{q}) \sim_j \mu_j(\vec{q}\,')$. We argue that $f(\vec{q}) = f(\vec{q}\,')$. Compute $f(\vec{q})$ and $f(\vec{q}\,')$ using Equation (2), choosing each π to put all occurrences of state j first and then the remaining elements of $\vec{q}$ and $\vec{q}\,'$ in the same order. Then the working states for the two computations are the same after processing all occurrences of j, by the definition of t_i and m_i; following that, the same states are processed in both computations, so they give the same result. Consequently $f(\vec{q}) = f(\vec{q}\,')$.

Using the above argument and stepping through all states $j \in Q$, we can show that if $\mu_j(\vec{q}) \sim_j \mu_j(\vec{q}\,')$ for all $j \in Q$, then $f(\vec{q}) = f(\vec{q}\,')$. It follows that we can write a mod-thresh program for f with $\Pi_{i=1}^{s}(t_i + m_i)$ clauses. Each clause is a conjunction of s terms, where each term is like the right-hand side of either Equation (4) or Equation (5). For each proposition P_i, to determine its corresponding result r_i, we pick a representative value of $\mu_j(\vec{q})$ for each j, thereby determining $\vec{q}$ up to order; then we set r_i equal to the sequential program's output on (any permutation of) $\vec{q}$. □

By Lemmas 3.5, 3.8, and 3.9, the proof of Theorem 3.7 is complete.

Henceforth let us call these three classes the *FSM functions* (where F stands for "finite.") We note briefly that the constructions of Lemmas 3.8 and 3.9 can entail an exponential increase in program complexity.

3.4 Finite-State Symmetric Graph Automata

Having found an automaton model (in fact, two) that satisfy (S0–S2), we now formally describe the associated model of distributed computing. When a node activates, it computes an FSM function of its neighbours' states, and changes its state to the output of that function. However, we also allow the node to read in its own state a priori, and this determines exactly which FSM function is used. So any node acts symmetrically on its neighbours but asymmetrically on itself.

DEFINITION 3.10. *Suppose that $|Q|$ is a finite set of* states. *For each $q \in Q$, let $f[q]$ be any FSM function from Q^+ to Q. Then (Q, f) describes a* finite-state symmetric graph automaton *(FSSGA).*

An FSSGA system can evolve either synchronously or asynchronously. Let $\sigma(\vec{\Gamma}(v))$ denote a list of the states of v's neighbours. In the asynchronous model, nodes activate one at a time, and when v_a activates, the network state σ is succeeded by the network state

$$\sigma' : v \mapsto \begin{cases} \sigma(v), & \text{if } v \neq v_a; \\ f[\sigma(v_a)](\sigma(\vec{\Gamma}(v_a))), & \text{if } v = v_a. \end{cases}$$

In the synchronous model, the network state σ is succeeded by the network state σ' defined by

$$\sigma' : v \mapsto f[\sigma(v)](\sigma(\vec{\Gamma}(v))).$$

In either model, by "running" an algorithm, we mean to iteratively replace the current network state with its successor. We assume the network is connected and has more than one node.

3.4.1 Randomness

So far, the model which we have described is deterministic. However, some tasks are well-known to be impossible unless some randomness is allowed, such as leader election [11]. Thus, we now state a probabilistic variant of the FSSGA model. In keeping with the minimalism of our finite-state model, each activating node is allowed a finite amount of randomness.

DEFINITION 3.11. *Suppose that $|Q|$ is a finite set of* states *and r is a finite positive integer. For each $q \in Q$ and $0 \leq i < r$, let $f[q, r]$ be any FSM function from Q^+ to Q. Then (Q, r, f) describes a* probabilistic FSSGA.

When a node v_a activates asynchronously, we uniformly select $i \in \{0, \ldots, r-1\}$ at random, and the new state of v_a is

$$f[\sigma(v_a), i](\sigma(\vec{\Gamma}(v_a))).$$

A synchronous step likewise incurs n independent random choices of i.

4. ALGORITHMS FOR THE MODEL

We now describe several algorithms that can be implemented in the FSSGA model. A Java applet demonstrating the algorithms of this section is currently available at `http://www.math.uwaterloo.ca/~dagpritc/fssga.html`. These algorithms culminate in a randomized leader election protocol that works in $O(n \log n)$ time with high probability.

4.1 2-colouring

Here is a very simple FSSGA algorithm that determines if a graph is bipartite, by attempting to 2-colour it. We take $Q = \{\mathcal{BLANK}, \mathcal{RED}, \mathcal{BLUE}, \mathcal{FAILED}\}$. Initially, one node is in the state $\mathcal{RED}$, and all others are in the 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 $\mathcal{BLANK}$
end if

4.2 Synchronizer

A *synchronizer* allows an asynchronous network to simulate a synchronous one. In the case of the FSSGA model we can adapt the α synchronizer of Awerbuch [2]. The basic idea behind the α synchronizer is that each node keeps a clock recording the number of rounds it has performed, and each pair of adjacent nodes keeps their clocks within ± 1 of each other. Each node remembers its "previous" state in order that its slower neighbours can catch up. As noted in [9] [3] [21] and elsewhere, adjacent nodes' clock values always differ by one of $\{-1, 0, 1\}$, so it suffices for nodes to keep track of their clocks modulo 3, i.e., using finite memory.

In the message-passing model the α synchronizer increases the communication complexity as a message is sent along every edge each round. However, in the FSSGA model, neighbour information is always available, and so the α synchronizer entails no increase in complexity. Precisely, assume for an asynchronous network that each node activates at least once per unit time; then we can show that in k units of time each node has advanced the clock of its synchronizer at least k times.

Given a FSSGA (Q, f) designed for a synchronous network, the synchronizer produces $(Q \times Q \times \{0, 1, 2\}, f_s)$, with f_s as follows. For each $q_c \in Q$, where the sequential program for $f[q_c]$ is (W, w_0, p, β), for each $q_p \in Q$ and $i \in \{0, 1, 2\}$, define the sequential program for $f_s[q_c, q_p, i]$ to be $(W \cup \{WAIT\}, w_0, p', \beta')$ where

$$p' \quad : \quad (w, (q_c', q_p', i')) \mapsto \begin{cases} WAIT, & \text{if } w = WAIT \text{ or } i' = (i-1) \bmod 3; \\ p(w, q_c'), & \text{if } w \neq WAIT \text{ and } i' = i; \\ p(w, q_p'), & \text{if } w \neq WAIT \text{ and } i' = (i+1) \bmod 3. \end{cases}$$

$$\beta' \quad : \quad w \mapsto \begin{cases} (q_c, q_p, i), & \text{if } w = WAIT; \\ (\beta(w), q_c, (i+1) \bmod 3), & \text{otherwise.} \end{cases}$$

Here q_c is the current state and q_p is the previous state. This is the last algorithm which we describe using formal FSM programs; hereafter we use informal descriptions in mod-thresh terms.

4.3 Breadth-First Search

In a synchronous setting, a breadth-first search (BFS) is like a broadcast in that both expand outwards in all directions as fast as possible. For this reason, we describe a BFS algorithm for the synchronous FSSGA model, and by using the result of Section 4.2 this can be transformed into an asynchronous algorithm.

In our implementation, each node labels itself according to its mod-3 distance from the (unique) originator of the search. If x is adjacent to y and the label of y is (modulo 3) one more than the label of x, then we call y a *successor* of x and x a *predecessor* of y. In this terminology, an algorithmic description of our BFS protocol is shown in Algorithm 4.1. In a formal mod-thresh definition, each of the clauses shown would be copied three times, once for each numeric value of *label*.

Algorithm 4.1 Breadth-first search in the FSSGA model.

```
let originator, target be fixed booleans
let label be a variable in {0, 1, 2, ⋆}
let status be a variable in {waiting, found, failed}
initialize label := ⋆ and status := waiting
if originator = true and label = ⋆ then
    label := 0
else if (label = ⋆) and (a neighbour has label x ≠ ⋆) then
    label := (x + 1) mod 3
    if target = true then
        status := found
    end if
else if status = waiting and any predecessor has status
    found then
    do nothing        ▷ avoid reporting non-shortest paths
else if status = waiting and any successor has status
    found then
    status := found
else if status = waiting and all successors have status
    failed then
    status := failed
end if
```

Note, to implement several "variables" as shown in the pseudocode, we make the set of states equal to a cartesian product of the variables' ranges. Specifically the set Q of node states is

$$\{true, false\}^2 \times \{0, 1, 2, NIL\} \times \{waiting, found, failed\}.$$

We will use this trick again implicitly in the algorithm descriptions to come.

4.4 Random Walk

The naive distributed description of a random walk, "if you contain the walker, then send the walker to a random neighbour," does not work for FSSGAs since a node cannot randomly pick from an arbitrarily large set of neighbours, nor can it directly modify any neighbour's state. Nonetheless there is a relatively simple randomized program which gives rise to a random walk.

We assume the existence of a single distinguished node in the network, which is the walker's initial position. We distinguish a subset Q_w of Q as *walker states.* In every time step, there will be exactly one node with state in Q_w, representing the walker's position.

The basic idea is that the node containing the walker asks its neighbours to flip coins, in order to determine who "wins" the walker next. On each round, those neighbours which flip *heads* are eliminated, until only one neighbour remains. One catch is that, if everybody flips *heads* on a given round, then the round must be re-run or else nobody would win. Finally, when all neighbours but exactly one are eliminated, the walker moves to that neighbour. It can be shown that, when the walker is at a node of degree d, the expected number of rounds before it moves is $\Theta(\log d)$.

We show pseudocode for a *synchronous* FSSGA random walk in Algorithm 4.2. The walker states are

$$Q_w := \{flip!, waiting\text{-}for\text{-}flips, notails, onetails\}.$$

The whole state space is

$$Q := Q_w \cup \{blank, heads, tails, eliminated\}. \qquad (6)$$

Algorithm 4.2 Random walk in the synchronous FSSGA model.

```
if any neighbour is in a walker state q_w ∈ Q_w then
    if q_w = flip! and I am heads then
        set my state to eliminated
    else if q_w = flip! and I am not eliminated then
        pick my state randomly from {heads, tails}
    else if q_w = notails and I am heads then
        pick my state randomly from {heads, tails}
    else if q_w = onetails and I am tails then
        set my state to flip!          ▷ receive the walker
    else if q_w = onetails then
        set my state to blank
    end if
else if I am waiting-for-flips then
    if no neighbours are in state tails then
        set my state to notails
    else if exactly one neighbour is in state tails then
        set my state to onetails       ▷ send the walker
    else
        set my state to flip!
    end if
else if I am notails or flip!, then
    set my state to waiting-for-flips  ▷ neighbours flip
else if I am onetails then
    set my state to blank              ▷ clear the walker's remains
end if
```

4.5 Graph Traversal

The *graph traversal* problem is to make a single agent visit every node of the network at least once. In [14], Milgram gives an algorithm for graph traversal in the IWA model. We can adapt this algorithm to the FSSGA model as follows.

Each node has a status drawn from the set

$$\{blank, arm, hand, by\text{-}arm, visited\}.$$

The set of nodes whose status lie in $\{arm, hand\}$ always form a sequence $\{v_0, \ldots, v_k\}$ such that

1. v_0 is the originator node,
2. nodes $v_0, \ldots, v_{k-1}$ have status arm, and
3. v_i is adjacent to v_j if and only if $i = j \pm 1$.

To paraphrase Milgram, the last property implies that the arm never touches or crosses itself. An unvisited non-arm node is supposed to have status *by-arm* or *blank* according to whether one of its neighbours has status *arm* or not, and this allows us to maintain property 3. In the implementation shown in Algorithm 4.3 we use the synchronizer's counter like a "clock" in order to alternate running the agent with updating the *by-arm* information.

The hand moves from node to adjacent node, like an agent. When possible, the hand moves onto a *blank* neighbour of its current position, thereby extending the arm. In order for the hand to choose a unique neighbour for extension, local symmetry breaking must be performed, and for this we "call" the random walk automaton as a subroutine. When the arm cannot extend, it instead retracts, and marks its previous endpoint (the hand) as visited. We refer to [14] for a full proof of correctness.

It can be shown that, in a given execution of Milgram's protocol, the arm traces out a tree. Specifically, the union of the paths $v_0, \ldots, v_k$ is a *scan-first search* spanning tree; and so the hand moves $2n-2$ times in total. Each step of symmetry breaking requires $O(\log n)$ time, so the total time complexity of this algorithm is $O(n \log n)$.

Algorithm 4.3 A synchronous traversal automaton.

```
if originator = true then
    initialize status := hand
else
    initialize status := blank
end if
if the current time is even then
    if status ∈ {blank, by-arm} then
        if any neighbour is arm then
            status := by-arm
        else
            status := blank
        end if
    end if
else                                   ▷ the current time is odd
    if status = arm then
        if (originator = false and at most one neighbour
            is arm or hand) or (originator = true and no
            neighbour is arm or hand) then
            status := hand                          ▷ retract arm
        end if
    else if status = hand then
        if no neighbour is blank then
            status := visited                       ▷ retract arm
        else
            update q_random-walk to elect a blank neighbour
            if the election is complete then
                status := arm                       ▷ extend arm
            end if
        end if
    else if status = blank and I've been elected then
        status := hand                              ▷ extend arm
    end if
end if
```

4.6 Greedy Traversal

Here we describe another graph traversal algorithm which we call the *greedy tourist*. It is slightly slower than Milgram's algorithm, but has better sensitivity. Let T denote a subset of $V(\mathcal{G})$, initially $T = V(\mathcal{G})$. Whenever a node in T is visited by the agent, remove it from T. Finally, make the agent always follow a shortest path to T. It is clear that the agent will eventually visit each node of the graph. It can be shown by [20] that the entire graph is traversed in $O(n \log n)$ steps. We may determine the shortest path to T by using the BFS of Section 4.3, obtaining (with slowdown due to local-symmetry breaking) a traversal in $O(n \log^2 n)$ time. But, whereas Milgram's algorithm has sensitivity $\Theta(n)$, the greedy tourist has sensitivity 1. Note, when adapting the greedy tourist to an asynchronous FSSGA network, the sensitivity becomes 2, as there are times where the tourist is "in transit" between two nodes. The same may be said of the biconnectivity algorithm from Section 2.1.

4.7 Leader Election

An election algorithm is an algorithmic form of global symmetry breaking; initially, all nodes are in the same state, but at the end, exactly one node must be in the state *leader*. We can implement an FSSGA leader election algorithm by combining some existing algorithmic ideas.

The basic idea can be found in [3]. Each node keeps a boolean flag *remain*, according to which we say that the node is either "remaining" or "eliminated." Each node is initially remaining, and once a node is eliminated, it never becomes remaining again. The algorithm proceeds in phases. In each phase, each remaining node picks a label uniformly at random from $\{0, 1\}$. Node v is eliminated in phase p if and only if v has label 0 in phase p *and* v detects that some other remaining node has label 1 in phase p. It follows that there is always at least one remaining node. We keep nodes synchronized in phases using a similar abstraction to that given in Section 4.2; in the psuedocode to follow, the phase counter p is a mod-3 variable. Our phases correspond to the "RESET" of [3].

At the start of phase p, each remaining node v builds a BFS cluster outwards from itself in all directions, hoping either to verify that it is the only remaining node or to discover other remaining nodes. We say that v is the *root* of this cluster. Each cluster consists of a root plus eliminated nodes, and each eliminated node joins the first cluster that grows to meet it. We make each BFS cluster propagate the label of its root. There are a few ways that a node w can discover that there are multiple clusters. For example, w may notice two neighbours propagating different labels (both 0 and 1), or it may be that two growing BFS clusters meet in the neighbourhood of w in such a way that the clusters' distance labels preclude the existence of just 1 root.

When a node determines that there are two or more remaining nodes (roots), it enters the state NP_i, which denotes that a new phase must occur, and that the largest label that it "knows about" is i. These NP_i messages propagate through the graph like a broadcast. Every node increments its phase counter immediately after being in state NP_i. Consistent with our description above, a remaining node in NP_1 becomes eliminated if its label was 0 in that phase.

The idea by which nodes verify their uniqueness comes from a self-stabilizing leader election algorithm of Dolev [5]. Recall that each remaining node is the root of a BFS cluster. When it appears that the BFS is complete, the root starts colouring itself randomly (say, red or blue) at each time

step. These colours propagate, using the successor relation, away from the root of each cluster. If there are more than 2 clusters, then some node v is in multiple clusters, and v is likely to eventually notice that two of its predecessors have different colours; this causes an NP message, hence a new phase and more chances for elimination.

Otherwise, after about n rounds, if no inconsistency is found, then the root elects itself as leader. A clever usage of Milgram's agent (Section 4.5) allows us to wait for about n rounds even though we can't explicitly count to n in our model. This decreases the probability of failure to $2^{-\Omega(n)}$. We give the pseudocode for this algorithm in Algorithm 4.4.

Algorithm 4.4 A synchronous election automaton.

```
initialize p := 0 and remain := true
at start of algorithm, pick a label and begin BFS
if any neighbor has phase p − 1 then
    do nothing
else if (any neighbor has phase p + 1) or (state = NP_x)
    then
    if (state = NP_1) and (remain) and (label = 0) then
        remain := false
    end if
    p := p+1
    if (remain) then pick a label and begin BFS end if
else if (I detect a BFS or tree-recolouring inconsistency)
    or (any neighbour is NP_x) then
    if (any neighbour is NP_1) or (label = 1) or (any neigh-
        bours' label is 1) then
        enter state NP_1
    else enter state NP_0
    end if
else if my BFS cluster is not complete then
    participate in BFS cluster construction
    propagate the label and colour of my cluster's root
else if (remain) then
    if my BFS cluster construction just finished then
        release a Milgram agent
    else if I have already released an agent then
        choose a new colour to propagate down cluster
    else if my agent just returned then
        enter state leader
    end if
end if
```

4.7.1 Correctness and Complexity

CLAIM 4.1. *In a given phase, if u remains and some other nodes remain, then u is eliminated with probability at least 1/4 in that phase.*

PROOF. For each node v let $t(v)$ denote the (synchronous) time that v entered this phase. Pick a remaining node $v \neq u$ so that $t(v)+dist_{\mathcal{G}}(v,u)$ is minimal. Then by considering the growth of v's BFS, and of the propagation of NP messages, u will be eliminated in this phase if its label is 0 and v's label is 1, which happens with probability 1/4. □

In the next claim, "steps" refer to synchronous time steps.

CLAIM 4.2. *If there is more than one remaining node in a given phase, then an inconsistency is detected during random recolouring, within $O(n)$ steps, with probability at least $1 - 2^{-n/2}$.*

PROOF. First, note that at least n recolourings have to occur in total, even if there are multiple clusters, since each step of an agent corresponds to a recolouring of its root, and the agents visit every vertex. It follows easily that there are at least n recolourings in the first n steps.

If there is more than one cluster, then each cluster is adjacent to at least one other cluster. So each randomly chosen colour is compared to at least one other randomly chosen colour. We now can show that at least $n/2$ colour pairs are compared whose consistencies are independent, so the probability that no inconsistency is detected is at most $2^{-n/2}$. □

It can be shown from Claim 4.1 that, with high probability, there will be $\Theta(\log n)$ phases, and from Claim 4.2 we can argue that with high probability every phase but the last will take $O(n)$ time. The last phase uses Milgram's agent and so takes $O(n \log n)$ time. Thus the total time complexity of our algorithm is

$$\Theta(\log n) \cdot O(n) + O(n \log n) = O(n \log n).$$

We note that in a long enough path graph, multiple nodes will likely enter the *leader* state prematurely. However, at termination, there is exactly one leader with high probability, and termination occurs in $O(n \log n)$ time with high probability.

5. DISCUSSION

A possible generalization of our model is to allow each node a binary tape of a certain size, instead of a finite choice of state. Let N be a positive integer parameter, $q, w : \mathbb{N} \to \mathbb{N}$, and define $Q_N := \{0,1\}^{q(N)}, W_N := \{0,1\}^{w(N)}$. Suppose that $w_{0N} \in W_N, \beta_N : W_N \to Q_N, p_N : W_N \times Q_N \to W_N$ are *uniformly* Turing-computable in N (so for example, $\beta_N(w,q)$ is computed by a three-input Turing machine whose inputs are N, w, q). Finally suppose that for each N, $(W_N, w_{0N}, p_N, \beta_N)$ is a sequential program for a SM function f_N. Then extending the techniques of this paper, we can get a uniformly Turing-computable parallel program for f_N with working states in $\{0,1\}^{w'(N)}$ for $w'(N) = O(2^{q(N)}w(N))$. However, we do not know of an example where we cannot take $w'(N) = O(w(N))$. Is it possible that the class of SM functions is so restrictive that sequential processing is never much more efficient than parallel processing?

We also note that it seems that the state of the activating node should be fed to α as a second input if tapes are used instead of finite state. For example, v can sequentially determine if any neighbour has the same tape-state as v, and so this should also be possible in parallel processing.

5.1 Equivalence with Isotonic Web Automata

The *isotonic web automaton* (IWA) distributed model [14] uses a finite-state agent and a finite set of node labels. It resembles our model in that the computation is symmetric and uses finitely many states. The main difference is that the IWA model has a single locus of action whereas our model has inherent parallelism. The agent has a finite set of transition rules. Each rule is conditional on the presence/absence of a particular label in the neighbourhood of the agent's position; the effect of each rule is to relabel the current position, for the agent to take a step to any neighbour having some specified label, and for the agent to enter a new state. A property that can be computed in the

IWA model can also be computed in the FSSGA model, and vice-versa; this is easily shown by simulating each model in the other, although we omit the details. An IWA can compute a single synchronous FSSGA round in $O(m)$ time, by using Milgram's traversal algorithm [14] and the neighbour-counting technique from Lemma 3.8. An FSSGA network can simulate an IWA with $O(\log \Delta)$ time delay; this delay is needed to break local symmetry and pick the agent's next destination, as in Sections 4.4–4.6.

5.2 Open FSSGA Problems

The *firing squad problem* for synchronous networks is, essentially, to make every node in the network enter a distinguished state *fire* at the same time. On path graphs there is a long history of solutions, some symmetric [22]. The usual solution to the firing squad problem in non-path graphs [21] is to find a spanning "virtual path graph" embedded in the graph, and then to run an algorithm like [22] on that path. The impossibility of permanent neighbour identification in our model makes this strategy inapplicable, and finding a non-path-based solution seems challenging.

An algorithm which is eventually correct despite any finite number of arbitrary faults is called *self-stabilizing* [5]. A self-stabilizing leader election algorithm for the FSSGA model would allow many other FSSGA algorithms to be made self-stabilizing. Of self-stabilizing election algorithms, there is a finite-state one for cycle graphs [13] and there are low-memory ones for general graphs [3][9], but none that we know of can be adapted to the FSSGA model for general graphs.

We have not yet found any practical use for mod atoms. Perhaps they can be cleverly applied to one of these problems, or else removed to yield a simpler model.

6. REFERENCES

[1] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In *Proc. 23rd Symp. Principles of Distributed Computing*, pages 290–299, 2004.

[2] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[3] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *Proc. 13th Symp. Principles of Distributed Computing*, pages 254–263, 1994.

[4] S. R. Buss, C. H. Papadimitriou, and J. N. Tsitsiklis. On the predictability of coupled automata: An allegory about chaos. In *Proc. 31st Symp. Foundations of Computer Science*, pages 788–293, 1990.

[5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[6] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[7] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[8] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, 1980.

[9] G. Itkis and L. Levin. Fast and lean self-stabilizing asynchronous protocols. In *Proc. 35th Symp. Foundations of Computer Science*, pages 226–239, 1994.

[10] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *Proc. 36th Symp. Theory of Computing*, pages 561–568, 2004.

[11] N. Lynch. A hundred impossibility proofs for distributed computing. In *Proc. 8th Symp. Principles of Distributed Computing*, pages 1–28, 1989.

[12] B. Martin. A geometrical hierarchy on graphs via cellular automata. *Fundamenta Informaticae*, 52(1–3):157–181, 2002.

[13] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant space. In *Proc. 24th Symp. Theory of Computing*, pages 667–678, 1992.

[14] D. L. Milgram. Web automata. *Information and Control*, 29(2):162–184, 1975.

[15] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 2000.

[16] S. Nath, P. B. Gibbons, Z. Anderson, and S. Seshan. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. 2nd Conf. Embedded Networked Sensor Systems*, pages 250–262, 2004.

[17] E. Rémila. Recognition of graphs by automata. *Theoret. Comput. Sci.*, 136(2):291–332, 1994.

[18] A. Rosenfeld. Networks of automata: some applications. *IEEE Trans. Systems, Man, and Cybernetics*, 5:380–383, 1975.

[19] A. Rosenfeld and D. L. Milgram. Web automata and web grammars. *Machine Intelligence*, 7:307–324, 1972.

[20] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(5):563–581, 1977.

[21] P. Rosenstiehl, J. Fiksel, and A. Holliger. Intelligent graphs: Networks of finite automata capable of solving graph problems. In R. C. Read, editor, *Graph Theory and Computing*, pages 219–265. Academic Press, 1972.

[22] H. Szwerinkski. Time-optimal solution of the firing-squad synchronization-problem for n-dimensional rectangles with the general at an arbitrary position. *Theoret. Comput. Sci.*, 19:305–320, 1982.

[23] G. Tel. *Introduction to Distibuted Algorithms*. Cambridge University Press, 2000.