ABELIAN NETWORKS I. FOUNDATIONS AND EXAMPLES

BENJAMIN BOND AND LIONEL LEVINE

ABSTRACT. In Deepak Dhar's model of abelian distributed processors, automata occupy the vertices of a graph and communicate via the edges. We show that two simple axioms ensure that the final output does not depend on the order in which the automata process their inputs. A collection of automata obeying these axioms is called an *abelian network*. We prove a least action principle for abelian networks. As an application, we show how abelian networks can solve certain linear and nonlinear integer programs asynchronously. In most previously studied abelian networks, the input alphabet of each automaton consists of a single letter; in contrast, we propose two non-unary examples of abelian networks: *oil and water* and *abelian mobile agents*.

1. INTRODUCTION

In recent years, it has become clear that certain interacting particle systems studied in combinatorics and statistical physics have a common underlying structure. These systems are characterized by an *abelian property* which says changing the order of certain interactions has no effect on the final state of the system. Up to this point, the tools used to study these systems – least action principle, local-to-global principles, burning algorithm, transition monoids and critical groups – have been developed piecemeal for each particular system. Following Dhar [Dha99a], we aim to identify explicitly what these various systems have in common and exhibit them as special cases of what we call an *abelian network*.

After giving the formal definition of an abelian network in §2, we survey a number of examples in §3. These include the well-studied sandpile and rotor networks as well as two *non-unary* examples: oil and water, and abelian mobile agents. In §4 we prove a least action principle for abelian networks and explore some of its consequences. One consequence is that "local abelianness implies global abelianness" (Lemma 4.7). Another is that abelian networks solve optimization problems of the following form: given a nondecreasing function $F : \mathbb{N}^k \to \mathbb{N}^k$, find the coordinatewise smallest vector $\mathbf{u} \in \mathbb{N}^k$ such that $F(\mathbf{u}) \leq \mathbf{u}$ (if it exists).

This paper is the first in a series of three. In the sequel [BL16a] we give conditions for a finite abelian network to halt on all inputs. Such a network has a

Date: February 12, 2016.

2010 *Mathematics Subject Classification.* 68Q10, 37B15, 90C10 .

Key words and phrases. abelian distributed processors, asynchronous computation, chip-firing, finite automata, least action principle, local-to-global principle, monotone integer program, rotor walk.

natural invariant attached to it, the *critical group*, whose structure we investigate in [BL16b].

2. Definition of an abelian network

This section begins with the formal definition of an abelian network, which is based on Deepak Dhar's model of *abelian distributed processors* [Dha99a, Dha99b, Dha06]. The term "abelian network" is convenient when one wants to refer to a collection of communicating processors as a single entity. Some readers may wish to look at the examples in §3 before reading this section in detail.

Let $G = (V, E)$ be a directed graph, which may have self-loops and multiple edges. Associated to each vertex $v \in V$ is a *processor* $\mathcal{P}_v$, which is an automaton with a single input port and multiple output ports, one for each edge $(v, u) \in E$. Each processor reads the letters in its input port in first-in-first-out order.

The processor $\mathcal{P}_v$ has an input alphabet A_v and state space Q_v. These sets will usually be finite (but see §3.8 for an example with infinite state space). We will always take the sets A_v for $v \in V$ to be disjoint, so that a given letter belongs to the input alphabet just one processor. No generality is lost by imposing this condition.

The behavior of the processor $\mathcal{P}_v$ is governed by a *transition function* T_v and *message passing functions* $T_{(v,u)}$ associated to each edge $(v, u) \in E$. Formally, these are maps

$$T_v : A_v \times Q_v \to Q_v \qquad \text{(new internal state)}$$

$$T_{(v,u)} : A_v \times Q_v \to A_u^* \qquad \text{(letters sent from } v \text{ to } u\text{)}$$

where A_u^* denotes the free monoid of all finite words in the alphabet A_u. We interpret these functions as follows. If the processor $\mathcal{P}_v$ is in state q and processes input a, then two things happen:

(1) Processor $\mathcal{P}_v$ transitions to state $T_v(a, q)$; and
(2) For each edge $(v, u) \in E$, processor $\mathcal{P}_u$ receives input $T_{(v,u)}(a, q)$.

If more than one $\mathcal{P}_v$ has inputs to process, then changing the order in which processors act may change the order of messages arriving at other processors. Concerning this issue, Dhar writes that

> "In many applications, especially in computer science, one considers such networks where the speed of the individual processors is unknown, and where the final state and outputs generated should not depend on these speeds. Then it is essential to construct protocols for processing such that the final result does not depend on the order at which messages arrive at a processor." [Dha06]

Therefore we ask that the following aspects of the computation *do not depend on the order in which individual processors act*:

(a) The **halting status** (i.e., whether or not processing eventually stops).
(b) The **final states** of the processors.
(c) The **run time** (total number of letters processed by all $\mathcal{P}_v$).
(d) The **local run times** (number of letters processed by a given $\mathcal{P}_v$).

(e) The **detailed local run times** (number of times a given $\mathcal{P}_v$ processes a given letter $a \in A_v$).

A priori it is not obvious that these goals are actually achievable by any nontrivial network. In §4 we will see, however, that a simple local commutativity condition ensures all five goals are achieved. To state this condition, we extend the domain of T_v and $T_{(v,u)}$ to $A_v^* \times Q_v$: if $w = aw'$ is a word in alphabet A_v beginning with a, then set $T_v(w,q) = T_v(w', T_v(a,q))$ and $T_{(v,u)}(w,q) = T_{(v,u)}(a,q)T_{(v,u)}(w', T_v(a,q))$, where the product denotes concatenation of words. For the empty word ϵ, we set $T_v(\epsilon, q) = q$ and $T_{(v,u)}(\epsilon, q) = \epsilon$.

Let $\mathbb{N}^A$ be the free commutative monoid generated by A, and write $w \mapsto |w|$ for the natural map $A^* \to \mathbb{N}^A$. So $|w|$ is a vector with coordinates indexed by A, and its coordinate $|w|_a$ is the number of letters a in the word w. In particular, words w, w' satisfy $|w| = |w'|$ if and only if w' is a permutation of w.

Definition 2.1. (Abelian Processor) The processor $\mathcal{P}_v$ is called *abelian* if for any words $w, w' \in A_v^*$ such that $|w| = |w'|$, we have for all $q \in Q_v$ and all edges $(v,u) \in E$

$$T_v(w,q) = T_v(w',q) \qquad \text{and} \qquad |T_{(v,u)}(w,q)| = |T_{(v,u)}(w',q)|.$$

That is, permuting the letters input to $\mathcal{P}_v$ does not change the resulting state of the processor $\mathcal{P}_v$, and may change each output word sent to $\mathcal{P}_u$ only by permuting its letters.

A simple induction shows that if Definition 2.1 holds for words w, w' of length 2, then it holds in general; see [HLW16, Lemma 2.1].

Definition 2.2. (Abelian Network) An *abelian network* on a directed graph $G = (V, E)$ is a collection of automata $\mathcal{N} = (\mathcal{P}_v)_{v \in V}$ indexed by the vertices of G, such that each $\mathcal{P}_v$ is abelian.

We make a few remarks about the definition:

1. The definition of an abelian network is *local* in the sense that it involves checking a condition on each processor individually. As we will see, these local conditions imply a "global" abelian property (Lemma 4.7).

2. A processor $\mathcal{P}_v$ is called *unary* if its alphabet A_v has cardinality 1. A unary processor is trivially abelian, and any network of unary processors is an abelian network. Most of the examples of abelian networks studied so far are actually unary networks (an exception is the block-renormalized sandpile defined in [Dha99a]). Non-unary networks represent an interesting realm for future study. The "oil and water model" defined in §3.8 is an example of an abelian network that is not a block-renormalized unary network.

2.1. **Comparison with cellular automata.**

Cellular automata are traditionally studied on the grid $\mathbb{Z}^d$ or on other lattices, but they may be defined on any directed graph G. Indeed, we would like to suggest (see §5.1) that the study of cellular automata on G could be a fruitful means of revealing interesting graph-theoretic properties of G.

Abelian networks may be viewed as cellular automata enjoying the following two properties.

1. **Abelian networks can update asynchronously.** Traditional cellular automata update in parallel: at each time step, all cells *simultaneously* update their states based on the states of their neighbors. Since perfect simultaneity is hard to achieve in practice, the physical significance of parallel updating cellular automata is open to debate. Abelian networks do not require the kind of central control over timing needed to enforce simultaneous parallel updates, because they reach the same final state no matter in what order the updates occur.

2. **Abelian networks do not rely on shared memory.** Implicit in the update rule of cellular automata is an unspecified mechanism by which each cell is kept informed of the states of its neighbors. The lower-level interactions needed to facilitate this exchange of information in a physical implementation are absent from the model. Abelian networks include these interactions by operating in a "message passing" framework instead of the "shared memory" framework of cellular automata: An individual processor in an abelian network cannot access the states of neighboring processors. It can only read the messages they send.

3. Examples

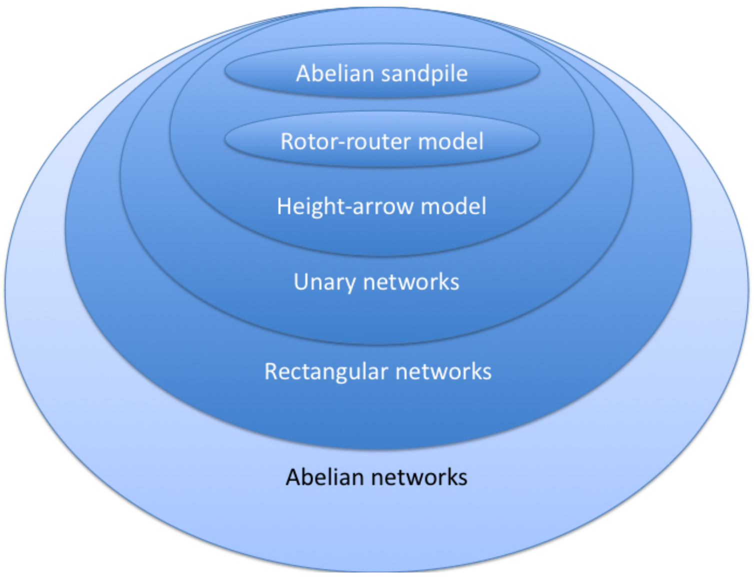

Figure 1. Venn diagram illustrating several classes of abelian networks.

3.1. **Sandpile networks.** Figure 1 shows increasingly general classes of abelian networks. The oldest and most studied is the *abelian sandpile model* [BTW87, Dha90], also called *chip-firing* [BLS91, Big99]. Given a directed graph $G = (V, E)$, the processor at each vertex $v \in V$ has a one-letter input alphabet $A_v = \{v\}$ (we call the letter v in order to keep the alphabets of different processors disjoint) and state space $Q_v = \{0, 1, \dots, r_v - 1\}$, where r_v is the outdegree of v. The transition function is

$$T_v(q) = q + 1 \pmod{r_v}.$$

(Formally we should write $T_v(v, q)$, but when $\#A_v = 1$ we omit the redundant first argument.) The message passing functions are

$$T_{(v,u)}(q) = \begin{cases} \epsilon, & q < r_v - 1 \\ u, & q = r_v - 1 \end{cases}$$

for each edge $(v, u) \in E$. Here $\epsilon \in A^*$ denotes the empty word (and passing the message ϵ is equivalent to passing nothing). Thus each time the processor at vertex v transitions from state $r_v - 1$ to state 0, it sends one letter to each of its out-neighbors (Figure 2). When this happens we say that vertex v *topples* (or "fires").

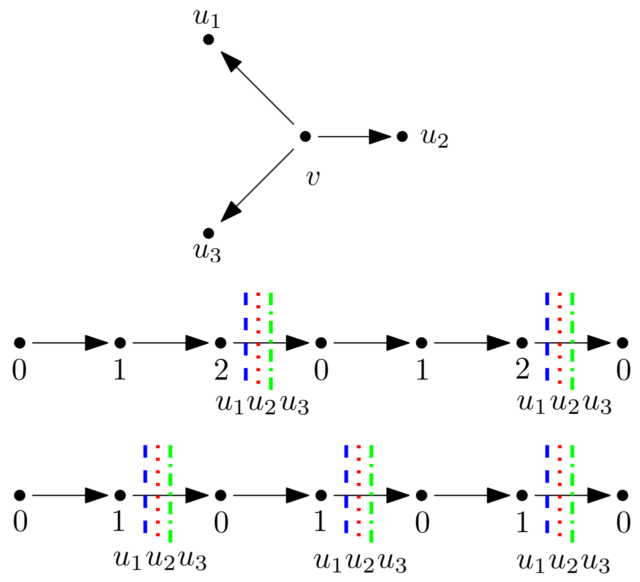

FIGURE 2. Top: portion of graph G showing a vertex v and its outneighbors u_1, u_2, u_3. Middle: State diagram for v in a *sandpile network*. Dots represent states, arrows represent transitions when a letter is processed, and dashed vertical lines indicate when letters are sent to the neighbors. Bottom: State diagram for the same v in a *toppling network* with $r_v = 2$.

Studies of pattern formation in sandpile networks include [Ost03, DSC09, ?]. The computational complexity of sandpile networks is investigated in [GM97] (where a parallel update rule is required) and in [MN99, MM11], where the focus is on comparing the computational power of sandpile networks with underlying graph $\mathbb{Z}^d$ for different dimensions d.

3.2. **Toppling networks.** These have the same transition and message passing functions as the sandpile networks above, but we allow the number of states r_v (called the *threshold* of vertex v) to be different from the outdegree of v. These networks can be concretely realized in terms of "chips": If a vertex in state q has k letters in its input port, then we say that there are $q+k$ chips at that vertex. When v has at least r_v chips, it can *topple*, losing r_v chips and sending one chip along each outgoing edge. In a sandpile network the total number of chips is conserved, but in a toppling network, chips may be created (if r_v is less than the outdegree of v, as in the last diagram of Figure 2) or destroyed (if r_v is larger than the outdegree of v).

Note that some chips are "latent" in the sense that they are encoded by the internal states of the processors. For example if a vertex v with $r_v = 2$ is in state 0, receives one chip and processes it, then the letter representing that chip is gone, but the internal state increases to 1 representing a latent chip at v. If v receives another chip and processes it, then its state returns to 0 and it topples by sending one letter to each out-neighbor.

It is convenient to specify a toppling network by its *Laplacian*, which is the $V \times V$ matrix L with diagonal entries $L_{vv} = r_v - d_{vv}$ and off-diagonal entries $L_{uv} = -d_{uv}$. Here d_{uv} is the number of edges from v to u in the graph G.

Sometimes it is useful to consider toppling networks where the number of chips at a vertex may become negative [Lev14]. We can model this by enlarging the state space of each processor to include $-\mathbb{N}$; these additional states have transition function $T_v(q) = q+1$ and send no messages. In §4.4 we will see that these enlarged toppling networks solve certain integer programs.

3.3. **Sinks and counters.** It is common to consider the sandpile network $\texttt{Sand}(G, s)$ with a *sink* s, a vertex whose processor has only one state and never sends any messages. If every vertex of G has a directed path to the sink, then any finite input to $\texttt{Sand}(G, s)$ will produce only finitely many topplings.

The set of *recurrent states* of a sandpile network with sink is in bijection with objects of interest in combinatorics such as oriented spanning trees and G-parking functions [PS04]. Recurrent states of more general abelian networks are defined and studied in the sequel paper [BL16b].

A *counter* is a unary processor with state space $\mathbb{N}$ and transition $T(q) = q+1$, which never sends any messages. It behaves like a sink, but keeps track of how many letters it has received.

3.4. **Bootstrap percolation.** In this simple model of crack formation, each vertex v has a threshold b_v. Vertex v becomes "infected" as soon as at least b_v of its in-neighbors are infected. Infected vertices remain infected forever. A question that has received a lot of attention [Ent87, Hol03] due to its subtle scaling behavior is: What is the probability the entire graph becomes infected, if each vertex independently starts infected with probability p? To realize bootstrap percolation as an abelian network, we take $A_v = \{v\}$ and $Q_v = \{0, 1, \dots, b_v\}$, with

$T_v(q) = \min(q+1, b_v)$ and

$$T_{(v,u)}(q) = \begin{cases} u, & q = b_v - 1 \\ \epsilon, & q \neq b_v - 1. \end{cases}$$

State b_v represents that v is infected. Starting from a blank slate $q_v = 0$ for all v, the user sets up the initial condition by inputing b_v letters v to each initially infected vertex v. The internal state q of an initially healthy processor $\mathcal{P}_v$ keeps track of how many in-neighbors of v are infected. When this count reaches b_v, the processor $\mathcal{P}_v$ sends a letter to each out-neighbor of v informing them that v is now infected.

3.5. **Rotor networks.** A *rotor* is a unary processor $\mathcal{P}_v$ that outputs exactly one letter for each letter input. That is, for all $q \in Q_v$

$$\sum_{(v,u)\in E} \sum_{a \in A_u} |T_{(v,u)}(q)|_a = 1. \tag{1}$$

Inputting a single letter into a network of rotors yields an infinite walk $(v_n)_{n\geq 0}$, where vertex v_n is the location of the single letter present after n processings. This walk has been termed *stack walk* [HP10] because of the following equivalent description (originating in [DF91]). Each vertex v has an infinite stack of cards, with each card labeled by a neighbor of v. The walker pulls the top card from the stack at her current location, steps to the indicated neighbor, throws away the card, and repeats. The stack perspective features prominently in Wilson's algorithm for sampling uniformly from the set of spanning trees of a finite graph [Wil96].

In the special case that each stack is periodic, the stack walk has been studied under various names: In computer science it was introduced as a model of autonomous agents exploring a territory ("ant walk," [WLB96]) and later studied as a means of broadcasting information through a network [DFS08]. In statistical physics it was proposed as a model of self-organized criticality ("Eulerian walkers," [PDDK96]). Propp called this case *rotor walk* and proposed it as a way of derandomizing certain features of random walk [Pro03, CS06, HP10, Pro10].

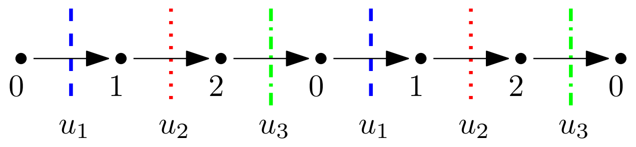

FIGURE 3. State diagram for a vertex v in a simple rotor network. The out-neighbors u_1, u_2, u_3 of v are served repeatedly in a fixed order.

Most commonly studied are the *simple* rotor networks on a directed graph G, in which the out-neighbors of vertex v are served repeatedly in a fixed order

$u_1, \ldots, u_{d_v}$ (Figure 3). Formally, we set $Q_v = \{0, 1, \ldots, d_v - 1\}$, with transition function $T_v(q) = q + 1 \pmod{d_v}$ and message passing functions

$$T_{(v,u_j)}(q) = \begin{cases} u_j, & q \equiv j - 1 \pmod{d_v} \\ \epsilon, & q \not\equiv j - 1 \pmod{d_v}. \end{cases}$$

Rotor aggregation. Enlarge each state space Q_v of a simple rotor network to include a transient state -1, which transitions to state 0 but passes no message. Starting with all processors in state -1, the effect is that each vertex "absorbs" the first letter it receives, and behaves like a rotor thereafter. If we input n letters to one vertex v_0, then each letter performs a rotor walk starting from v_0 until reaching a site that has not yet been visited by any previous walk, where it gets absorbed. Propp [Pro03] proposed this model as a way of derandomizing a certain random growth process (internal DLA). When the underlying graph is the square grid $\mathbb{Z}^2$, the resulting set of n visited sites is very close to circular [LP09], and the final states of the processors display intricate patterns that are still not at all understood.

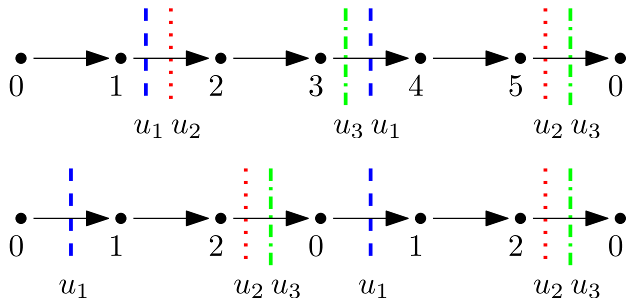

FIGURE 4. Example state diagrams for a vertex v in the height arrow model (top) and Eriksson's periodically mutating game (bottom).

3.6. **Unary networks.** As shown in Figure 4, various other abelian processor state diagrams can be obtained by changing the locations of the vertical lines in Figure 3. For example, Priezzhev, Dhar, Dhar and Krishnamurthy [PDDK96] proposed a common generalization of rotor and sandpile networks, later studied by Dartois and Rossin [DR04] under the name *height arrow model*. More generally, Diaconis and Fulton [DF91] and Eriksson [Eri96] studied generalizations of chip-firing in which each vertex has a stack of instructions: When a vertex accumulates enough chips to follow the top instruction in its stack, it pops that instruction off the stack and follows it. These and all preceding examples are *unary networks*, that is, abelian networks in which each alphabet A_v has cardinality 1. Informally, a unary network on a graph G is a system of local rules by which *indistinguishable* chips move around on the vertices of G.

Figures 2–4 are all one-dimensional because they diagram unary processors. In general, a processor with input alphabet $\{a_1, \ldots, a_d\}$ has a d-dimensional state diagram: states correspond to vectors in $\mathbb{N}^d$, and processing letter a_i results in a transition from state q to state $q + e_i$ where $e_1, \dots, e_d$ are the standard basis vectors. The vertical bars that indicate message passing in Figures 2–4 become $(d-1)$-dimensional plaquettes, each labeled by a letter to be passed. A visual manifestation of the abelian property is that these plaquettes join up into surfaces of "negative slope": For example, beginning at the left side of Figure 5 each red or blue message line $(d-1=1)$ consists of only downward and rightward steps, ensuring that for any two states q, q' any two paths of upward and rightward steps from q to q' cross the same set of message lines.

The next two sections discuss non-unary examples.

3.7. **Abelian mobile agents.** In the spirit of [WLB96], one could replace the messages in our definition of abelian networks by mobile agents each of which is an automaton. As a function of its own internal state a and the state q of the vertex v it currently occupies, an agent acts by doing three things:

(1) it changes its own state to $S_v(a,q)$; and
(2) it changes the state of v to $T_v(a,q)$; and
(3) it moves to a neighboring vertex $U_v(a,q)$.

Two or more agents may occupy the same vertex, in which case we require that the outcome of their actions is the same regardless of the order in which they act. For purposes of deciding whether two outcomes are the same, we regard agents with the same internal state and location as indistinguishable.

This model may appear to lie outside our framework of abelian networks, because the computation is located in the moving agents (who carry their internal states with them) instead of in the static processors. However, it has identical behavior to the following abelian network. Denoting by M the set of possible agent internal states (we could call them "moods" to distinguish them from the internal states of the vertices), let each vertex have input alphabet M (technically, we should take the input alphabet of v to be $\{v\} \times M$ to abide by our convention that the input alphabets are disjoint) with transition function $M \times Q_v \to Q_v$ sending $(a,q) \mapsto T_v(a,q)$, and message passing function $M \times Q_v \to M \cup \{\epsilon\}$ given by

$$T_{(v,u)}(a,q) = \begin{cases} S_v(a,q) & \text{if } u = U_v(a,q) \\ \epsilon & \text{else.} \end{cases}$$

Abelian mobile agents generalize the rotor networks (§3.5) by dropping the requirement that processors be unary.

The defining property of abelian mobile agents is that each processor sends exactly one letter for each letter received. In Figure 5 this property is apparent from the fact that each segment of the square grid lies on exactly one message line. The caption is written from the processor's point of view. From the agent's point of view, it could read as follows. When an agent arrives at vertex v, she updates the internal state q of $\mathcal{P}_v$ depending on her mood: if her mood is red then she increments q by $(0,1)$ and if blue then she increments q by $(1,0)$. The old and new states are adjacent black dots in the figure, separated by exactly one message line. The agent updates her mood to red or blue according to the color of this

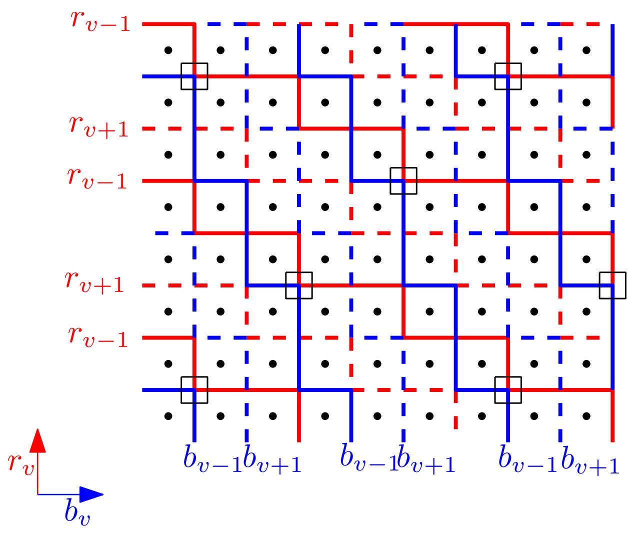

FIGURE 5. Abelian mobile agents: Example state diagram for a processor $\mathcal{P}_v$ in a network whose underlying graph is $\mathbb{Z}$. The two dimensions correspond to the two letters in the input alphabet $A_v = \{r_v, b_v\}$, representing a red or blue agent at vertex v. Each black dot represents a state $q \in Q_v$. When processor $\mathcal{P}_v$ in state q processes a letter, it transitions to state $q + (0,1)$ or $q + (1,0)$ depending on whether the letter was r_v or b_v. The solid and dashed colored lines indicate message passing: Each line is labeled by one of the letters $r_{v-1}, r_{v+1}, b_{v-1}, b_{v+1}$, representing that the agent may step either left or right from v and may change color. The small black boxes highlight the lattice of periodicity, generated by $(6,0)$ and $(2,2)$. The size of the state space $\#Q_v$ is the index of the lattice, which is 12 in this example.

line, and she moves to vertex $v-1$ or $v+1$ according to whether this line is solid or dashed.

For example, supposing the initial state of $\mathcal{P}_v$ is $(0,0)$ (the bottom left dot) and there is one red and one blue agent at v. This means $\mathcal{P}_v$ has two letters in its input port, r_v and b_v. If the blue agent acts first, then $\mathcal{P}_v$ transitions to state $(1,0)$ and outputs b_{v-1}, representing that the blue agent steps to $v-1$ and remains blue. If now the red agent at v acts, then $\mathcal{P}_v$ transitions from state $(1,0)$ to $(1,1)$ and outputs r_{v-1}, representing that the red agent steps to $v-1$ and remains red. Note that if the agents had acted in the opposite order, then both would have changed color, so the net result is the same: one red and one blue agent at $v-1$.

3.8. **Oil and water model.** This is a non-unary generalization of sandpiles, inspired by Paul Tseng's asynchronous algorithm for solving certain linear programs [Tse90]. Each edge of G is marked either as an oil edge or a water edge. When a vertex topples, it sends out one oil chip along each outgoing oil edge and also one water chip along each outgoing water edge. The interaction between oil and water is that a vertex is permitted to topple if and only if sufficiently many chips of *both* types are present at that vertex: that is, the number of oil chips present must be at least the number of outgoing oil edges, and the number of water chips present must be at least the number of outgoing water edges.

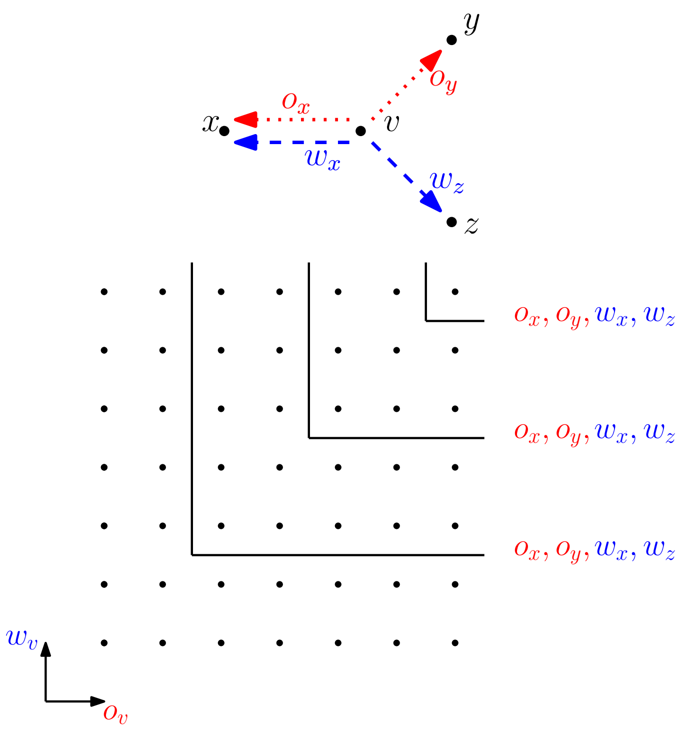

FIGURE 6. Example state diagram for the oil and water model. Top: Vertex v has outgoing oil edges to x and y, and water edges to x and z. Bottom: each dot represents a state in $Q_v = \mathbb{N} \times \mathbb{N}$, with the origin at lower left. A toppling occurs each time the state transition crosses one of the bent lines (for example, by processing an oil o_v in state $(1, 2)$, resulting in transition to state $(2, 2)$). Since v has outdegree 2 in both the oil graph and the water graph, the bent lines run to the left of columns whose x-coordinate is divisible by 2, and below rows whose y-coordinate is divisible by 2.

Unlike most of the preceding examples, oil and water can not be realized with a finite state space Q_v, because an arbitrary number of oil chips could accumulate at v and be unable to topple if no water chips are present. We set $Q_v = \mathbb{N} \times \mathbb{N}$ and $A_v = \{o_v, w_v\}$ representing an oil or water chip at vertex v, with transition function

$$T_v(o_v, q) = q + (0, 1), \qquad T_v(w_v, q) = q + (1, 0).$$

The internal state of the processor at v is a vector $q = (q_{oil}, q_{water})$ keeping track of the total number chips of each type it has received (Figure 6). Stochastic versions of the oil and water model are studied in [APR09, CGHL16].

3.9. **Stochastic abelian networks.** In a stochastic abelian network, we allow the transition functions to depend on a probability space Ω:

$$\begin{aligned} T_v : A_v \times Q_v \times \Omega &\to Q_v && \text{(new internal state)} \\ T_{(v,u)} : A_v \times Q_v \qquad &\to A_u^* && \text{(letters sent from } v \text{ to } u\text{)} \end{aligned}$$

A variety of models in statistical mechanics — including classical Markov chains and branching processes, branching random walk, certain directed edge-reinforced walks, internal DLA [DF91], the Oslo model [Fre93], the abelian Manna model [Dha99c], excited walk [BW03], the Kesten-Sidoravicius infection model [KS05, KS08], two-component sandpiles and related models derived from abelian algebras [AR08, APR09], activated random walkers [DRS10], stochastic sandpiles [RS12, CMS13], and low-discrepancy random stack [FL13] — can all be realized as stochastic abelian networks. In at least one case [RS12] the abelian nature of the model enabled a major breakthrough in proving the existence of a phase transition. Stochastic abelian networks are beyond the scope of the present paper and will be treated in a sequel.

4. Least action principle

Our first aim is to prove a least action principle for abelian networks, Lemma 4.3. This principle says — in a sense to be made precise — that each processor in an abelian network performs the minimum amount of work possible to remove all letters from the network. Various special cases of the least action principle to particular abelian networks have enabled a flurry of recent progress: bounds on the growth rate of sandpiles [FLP10], exact shape theorems for rotor aggregation [KL10, HS11], proof of a phase transition for activated random walkers [RS12], and a fast simulation algorithm for growth models [FL13]. The least action principle was also the starting point for the recent breakthrough by Pegden and Smart [PS13] showing existence of the abelian sandpile scaling limit.

The proof of the least action principle follows Diaconis and Fulton [DF91, Theorem 4.1]. Our observation is that their proof actually shows something more general: it applies to any abelian network. Moreover, as noted in [Gab94, FLP10, RS12], the proof applies even to executions that are complete but not legal. To explain the last point requires a few definitions.

Let $\mathcal{N}$ be an abelian network with underlying graph $G = (V, E)$, total state space $Q = \prod Q_v$ and total alphabet $A = \sqcup A_v$. In this section we do not place any finiteness restrictions on $\mathcal{N}$: the underlying graph may be finite or infinite, and the state space Q_v and alphabet A_v of each processor may be finite or infinite.

We may view the entire network $\mathcal{N}$ as a single automaton with alphabet A and state space $\mathbb{Z}^A \times Q$. For its states we will use the notation $\mathbf{x}.\mathbf{q}$, where $\mathbf{x} \in \mathbb{Z}^A$ and $\mathbf{q} \in Q$. If $\mathbf{x} \in \mathbb{N}^A$ the state $\mathbf{x}.\mathbf{q}$ corresponds to the configuration of the network $\mathcal{N}$ such that

- For each $a \in A$, there are $\mathbf{x}_a$ letters a waiting to be processed; and
- For each $v \in V$, the processor at vertex v is in state q_v.

Formally, $\mathbf{x}.\mathbf{q}$ is just an alternative notation for the ordered pair $(\mathbf{x}, \mathbf{q})$. The decimal point in $\mathbf{x}.\mathbf{q}$ is intended to evoke the intuition that the internal states $\mathbf{q}$ of the processors represent latent "fractional" messages. Note that $\mathbf{x}$ indicates only the *number* of letters present of each type. We may think of $\mathbf{x}$ as a collection of piles of letters, one pile for each vertex: Recalling that the alphabets A_v are disjoint, the $\mathbf{x}_a$ letters a are in the pile of the unique vertex v such that $a \in A_v$.

In what follows it may be helpful to imagine that some entity, the *executor*, chooses the order in which letters are processed. Formally, these choices are encoded by a word $w = w_1 \cdots w_r$ where each letter $w_i \in A$, instructing the network first to process letter w_1, then w_2, etc. We are going to allow for the possibility that the executor makes an "illegal" move by choosing to process some letter (say a) even if it is not in the pile, resulting in the coordinate $\mathbf{x}_a$ becoming negative.

For $v \in V$ and $a \in A_v$, denote by $t_a : Q \to Q$ the map

$$t_a(\mathbf{q})_u = \begin{cases} T_v(a, q_v), & u = v \\ q_u, & u \neq v \end{cases}$$

where T_v is the transition function of vertex v (defined in §2). The effect of processing one letter a on the pair $\mathbf{x}.\mathbf{q}$ is described by a map $\pi_a : \mathbb{Z}^A \times Q \to \mathbb{Z}^A \times Q$, namely

$$\pi_a(\mathbf{x}.\mathbf{q}) = (\mathbf{x} - 1_a + \mathbf{N}(a, q_v)).t_a(\mathbf{q}) \tag{2}$$

where $(1_a)_b$ is 1 if $a = b$ and 0 otherwise; and $\mathbf{N}(a, q_v)_b$ is the number of b's produced when processor $\mathcal{P}_v$ in state q_v processes the letter a. In other words,

$$\mathbf{N}(a, q_v) = \sum_e |T_e(a, q_v)|$$

where T_e is the message passing function of edge e, and the sum is over all outgoing edges e from v (both sides are vectors in $\mathbb{Z}^A$).

Having defined π_a for letters a, we define π_w for a word $w = w_1 \cdots w_r \in A^*$ as the composition $\pi_{w_r} \circ \cdots \circ \pi_{w_1}$. To generalize equation (2), we extend the domain of $\mathbf{N}$ to $A^* \times Q$ as follows. Let $\mathbf{q}^{i-1} = (t_{w_{i-1}} \circ \cdots \circ t_{w_1})\mathbf{q}$ and let

$$\mathbf{N}(w, \mathbf{q}) := \sum_{i=1}^{r} \mathbf{N}(w_i, \mathbf{q}^{i-1}_{v(i)})$$

where $v(i)$ is the unique vertex such that $w_i \in A_{v(i)}$. Note that if $a \in A_v$ and $b \in A_u$ for $v \neq u$, then

$$\mathbf{N}(ab, \mathbf{q}) = \mathbf{N}(a, \mathbf{q}) + \mathbf{N}(b, \mathbf{q}) \tag{3}$$

since t_a acts by identity on Q_u and t_b acts by identity on Q_v.

Recall that $|w| \in \mathbb{N}^A$ and $|w|_a$ is the number of occurrences of letter a in the word w. From the definition of π_a we have by induction on r

$$\pi_w(\mathbf{x}.\mathbf{q}) = (\mathbf{x} - |w| + \mathbf{N}(w, \mathbf{q})).t_w(\mathbf{q}) \tag{4}$$

where $t_w := t_{w_r} \circ \cdots \circ t_{w_1}$.

In the next lemma and throughout this paper, inequalities on vectors are coordinatewise.

Lemma 4.1. (Monotonicity) *For $w, w' \in A^*$ and $\mathbf{q} \in Q$, if $|w| \le |w'|$, then $\mathbf{N}(w,\mathbf{q}) \le \mathbf{N}(w',\mathbf{q})$.*

Proof. For a vertex $v \in V$ let $p_v : A^* \to A_v^*$ be the monoid homomorphism defined by $p_v(a) = a$ for $a \in A_v$ and $p_v(a) = \epsilon$ (the empty word) for $a \notin A_v$. Equation (3) implies that

$$\mathbf{N}(w,\mathbf{q}) = \sum_{v\in V} \mathbf{N}(p_v(w),\mathbf{q}),$$

so it suffices to prove the lemma for $w, w' \in A_v^*$ for each $v \in V$.

Fix $v \in V$ and $w, w' \in A_v^*$ with $|w| \le |w'|$. Then there is a word w'' such that $|ww''| = |w'|$. Given a letter $a \in A_u$, if $(v,u) \notin E$ then $\mathbf{N}(w,\mathbf{q})_a = \mathbf{N}(w',\mathbf{q})_a = 0$. If $(v,u) \in E$, then since $\mathcal{P}_v$ is an abelian processor,

$$\begin{aligned}\mathbf{N}(w',\mathbf{q})_a = |T_{(v,u)}(w',q_v)|_a &= |T_{(v,u)}(ww'',q_v)|_a \\ &= |T_{(v,u)}(w,q_v)|_a + |T_{(v,u)}(w'',T_v(w,q_v))|_a.\end{aligned}$$

The first term on the right side equals $\mathbf{N}(w,\mathbf{q})_a$, and the remaining term is nonnegative, completing the proof. □

Lemma 4.2. *For $w, w' \in A^*$, if $|w|_a = |w'|_a$ for all $a \in A$, then $\pi_w = \pi_{w'}$.*

Proof. Suppose $|w| = |w'|$. Then for any $\mathbf{q} \in Q$ we have $\mathbf{N}(w,\mathbf{q}) = \mathbf{N}(w',\mathbf{q})$ by Lemma 4.1. Since t_a and t_b commute for all $a, b \in A$, we have $t_w(\mathbf{q}) = t_{w'}(\mathbf{q})$. Hence the right side of (4) is unchanged by substituting w' for w. □

4.1. **Legal and complete executions.** An *execution* is a word $w \in A^*$. It prescribes an order in which letters in the network are to be processed. For simplicity, we consider only finite executions in the present paper, but we remark that infinite executions (and non-sequential execution procedures) are also of interest [FMR09].

Fix an initial state $\mathbf{x}.\mathbf{q}$. The letter $a \in A$ is called a *legal move* from $\mathbf{x}.\mathbf{q}$ if $\mathbf{x}_a \ge 1$. An execution $w = w_1 \cdots w_r$ is called *legal* for $\mathbf{x}.\mathbf{q}$ if w_i is a legal move from $\pi_{w_1\cdots w_{i-1}}(\mathbf{x}.\mathbf{q})$ for all $i = 1,\dots,r$. An execution w is called *complete* for $\mathbf{x}.\mathbf{q}$ if $\pi_w(\mathbf{x}.\mathbf{q}) = \mathbf{y}.\mathbf{q}'$ for some $\mathbf{q}' \in Q$ and $\mathbf{y} \in \mathbb{Z}^A$ with $\mathbf{y}_a \le 0$ for all $a \in A$. We emphasize that a complete execution need not be legal.

Lemma 4.3. (Least Action Principle) *If w is legal for $\mathbf{x}.\mathbf{q}$ and w' is complete for $\mathbf{x}.\mathbf{q}$, then $|w|_a \le |w'|_a$ for all $a \in A$.*

Proof. Let $w = w_1 \cdots w_r$. Supposing for a contradiction that $|w| \not\le |w'|$, let i be the smallest index such that $|w_1 \cdots w_i| \not\le |w'|$. Let $u = w_1 \cdots w_{i-1}$ and $a = w_i$. By the choice of i we have $|u|_a = |w'|_a$, and $|u|_b \le |w'|_b$ for all $b \ne a$. Since w is legal for $\mathbf{x}.\mathbf{q}$, at least one letter a is present in $\pi_{\mathbf{u}}(\mathbf{x}.\mathbf{q})$, so by (4) and Lemma 4.1

$$\begin{aligned}1 &\le \mathbf{x}_a - |u|_a + \mathbf{N}(u,\mathbf{q})_a \\ &\le \mathbf{x}_a - |w'|_a + \mathbf{N}(w',\mathbf{q})_a.\end{aligned}$$

Since w' is complete for $\mathbf{x}.\mathbf{q}$, the right side is ≤ 0 by (4), which yields the required contradiction. □

4.2. **Halting dichotomy.**

Lemma 4.4. (Halting Dichotomy) *For a given initial state* $\mathbf{q}$ *and input* $\mathbf{x}$ *to an abelian network* $\mathcal{N}$*, either*

(1) There does not exist a finite complete execution for $\mathbf{x}.\mathbf{q}$*; or*
(2) Every legal execution for $\mathbf{x}.\mathbf{q}$ *is finite, there exists a complete legal execution for* $\mathbf{x}.\mathbf{q}$*, and any two complete legal executions* w, w' *for* $\mathbf{x}.\mathbf{q}$ *satisfy* $|w| = |w'|$.

Proof. If there exists a finite complete execution, say of length s, then every legal execution has length $\leq s$ by Lemma 4.3. The empty word is a legal execution, and any legal execution of maximal length is complete (else it could be extended by a legal move). If w and w' are complete legal executions, then $|w| \leq |w'| \leq |w|$ by Lemma 4.3. □

Note that in case (1) any finite legal execution w can be extended by a legal move: since w is not complete, there is some letter $a \in A$ such that wa is legal. So in this case there is an infinite word $a_1 a_2 \cdots$ such that $a_1 \cdots a_n$ is a legal execution for all $n \geq 1$. The *halting problem* for abelian networks asks, given $\mathcal{N}$, $\mathbf{x}$ and $\mathbf{q}$, whether (1) or (2) of Lemma 4.4 is the case. In case (2) we say that $\mathcal{N}$ *halts* on input $\mathbf{x}.\mathbf{q}$. In the sequel [BL16a] we characterize the finite abelian networks that halt on all inputs.

4.3. **Global abelianness.**

Definition 4.5. (Odometer) If $\mathcal{N}$ halts on input $\mathbf{x}.\mathbf{q}$, we denote by $[\mathbf{x}.\mathbf{q}]_a = |w|_a$ the total number of letters a processed during a complete legal execution w of $\mathbf{x}.\mathbf{q}$. The vector $[\mathbf{x}.\mathbf{q}] \in \mathbb{N}^A$ is called the *odometer* of $\mathbf{x}.\mathbf{q}$. By Lemma 4.4, the odometer does not depend on the choice of complete legal execution w.

No messages remain at the end of a complete legal execution w, so the network ends in state $\pi_w(\mathbf{x}.\mathbf{q}) = \mathbf{0}.t_w(\mathbf{q})$. Hence by (4), the odometer can be written as

$$[\mathbf{x}.\mathbf{q}] = |w| = \mathbf{x} + \mathbf{N}(w, \mathbf{q})$$

which simply says that the total number of letters processed (of each type $a \in A$) is the sum of the letters input and the letters produced by message passing. The coordinates of the odometer are the "detailed local run times" from §2. We can summarize our progress so far in the following theorem.

Theorem 4.6. *Abelian networks have properties (a)–(e) from §2.*

Proof. By Lemma 4.4 the halting status does not depend on the execution, which verifies item (a). Moreover for a given $\mathcal{N}, \mathbf{x}, \mathbf{q}$ any two complete legal executions have the same odometer, which verifies items (c)–(e). The odometer and initial state $\mathbf{q}$ determine the final state $t_w(\mathbf{q})$, which verifies (b). □

The next lemma illustrates a general theme of *local-to-global principles* in abelian networks. Suppose we are given a partition $V = I \sqcup M \sqcup O$ of the vertex set into "input", "mediating" and "output" nodes, and that the output nodes never send messages (for example, the processor at each output node could be a counter, §3.3).

We allow the possibility that M and/or O is empty. If $\mathcal{N}$ halts on all inputs, then we can regard the induced subnetwork $(\mathcal{P}_v)_{v \in I \cup M}$ of non-output nodes as a single processor $\mathcal{P}_{I,M}$ with input alphabet $A_I := \sqcup_{v \in I} A_v$, state space $Q_{I \cup M} := \prod_{v \in I \cup M} Q_v$, and an output port for each edge $(v, u) \in (I \cup M) \times O$.

Lemma 4.7. (Local Abelianness Implies Global Abelianness) *If $\mathcal{N}$ halts on all inputs and $\mathcal{P}_v$ is an abelian processor for each $v \in I \cup M$, then $\mathcal{P}_{I,M}$ is an abelian processor.*

Proof. Given an input $\iota \in A_I^*$ and an initial state $\mathbf{q} \in Q_{I \cup M}$, we can process one letter at a time to obtain a complete legal execution for $|\iota|.\mathbf{q}$. Now suppose we are given inputs ι, ι' such that $|\iota| = |\iota'|$. By Lemma 4.4, any two complete legal executions w, w' for $|\iota|.\mathbf{q} = |\iota'|.\mathbf{q}$ satisfy $|w| = |w'|$. In particular, $t_w(\mathbf{q}) = t_{w'}(\mathbf{q})$, so the final state of $\mathcal{P}_{I,M}$ does not depend on the order of input.

Now given $v \in I \cup M$, let w_v and w'_v respectively be the words obtained from w and w' by deleting all letters not in A_v. Then $|w_v| = |w'_v|$. For each edge $(v, u) \in (I \cup M) \times O$, since $\mathcal{P}_v$ is an abelian processor,

$$|T_{(v,u)}(w_v, q_v)| = |T_{(v,u)}(w'_v, q_v)|$$

so for each $a \in A_u$ the number of letters a sent along (v, u) does not depend on the order of input. $\square$

For another example of a local-to-global principle, see [BL16b, Lemma 2.6] (note that $I = V$ in that example: input is permitted anywhere in the network). Further local-to-global principles in the case of rotor networks are explored in [GLPZ12].

Remark 4.8. In the preceding lemma, the input to an abelian network takes the form of letters sent by the user to nodes in I, and the output takes the form of letters received by the user from nodes in O. In particular, the user has no access to the internal states of any processors, nor to letters sent or received by the mediating nodes. In this setup, we can say that the network “computes” a function $\mathbb{N}^I \to \mathbb{N}^O$. This notion of computation is explored in [HLW16], which identifies a set of five *abelian logic gates* such that any function computable by a finite abelian processor can be computed by a finite network of abelian logic gates.

In cases when the user has access to the internal states of the processors in $I \cup O$, we can regard the input as a pair $\mathbf{x}.\mathbf{q}$ with $\mathbf{x} \in \mathbb{N}^I$ and $\mathbf{q} \in Q_I$, and the output as a pair $\mathbf{x}'.\mathbf{q}'$ with $\mathbf{x}' \in \mathbb{N}^O$ and $\mathbf{q}' \in Q_O$. For example, in the case of a sandpile or rotor network on $\mathbb{Z}^2$, one might want to think of the network's output as the intricate patterns displayed by the final states of the processors. The next section describes an example when the user benefits from the ability to set up the initial states of the processors as part of the input.

4.4. **Monotone integer programming.** In this section we describe a class of optimization problems that abelian networks can solve. Let A be a finite set and $F : \mathbb{N}^A \to \mathbb{N}^A$ a nondecreasing function: $F(\mathbf{u}) \leq F(\mathbf{v})$ whenever $\mathbf{u} \leq \mathbf{v}$ (inequalities are coordinatewise). Let $\mathbf{c} \in \mathbb{R}^A$ be a vector with all coordinates positive, and consider the following problem.

$$\begin{array}{lll} \text{Minimize} & \mathbf{c}^T\mathbf{u} & \\ \text{subject to} & \mathbf{u} \in \mathbb{N}^A \quad \text{and} & F(\mathbf{u}) \le \mathbf{u}. \end{array} \tag{5}$$

Let us call a vector $\mathbf{u} \in \mathbb{N}^A$ *feasible* if $F(\mathbf{u}) \le \mathbf{u}$. If $\mathbf{u}_1$ and $\mathbf{u}_2$ are both feasible, then their coordinatewise minimum is feasible:

$$F(\min(\mathbf{u}_1, \mathbf{u}_2)) \le \min(F(\mathbf{u}_1), F(\mathbf{u}_2)) \le \min(\mathbf{u}_1, \mathbf{u}_2).$$

Therefore if a feasible vector exists then the minimizer is unique and independent of the positive vector $\mathbf{c}$: it is simply the coordinatewise minimum of all feasible vectors.

Let $\mathcal{N}$ be an abelian network with finite alphabet A and finite or infinite state space Q. Fix $\mathbf{x} \in \mathbb{N}^A$ and $\mathbf{q} \in Q$, and let $F : \mathbb{N}^A \to \mathbb{N}^A$ be given by

$$F(\mathbf{u}) = \mathbf{x} + \mathbf{N}(\mathbf{u}, \mathbf{q})$$

where $\mathbf{N}(\mathbf{u}, \mathbf{q})$ is defined as $\mathbf{N}(w, \mathbf{q})$ for any word w such that $|w| = \mathbf{u}$. The function F is well-defined and nondecreasing by Lemma 4.1.

Recall the odometer $[\mathbf{x}.\mathbf{q}]$ is the vector of detailed local run times (Definition 4.5).

Theorem 4.9. (Abelian Networks Solve Monotone Integer Programs)

(i) *If $\mathcal{N}$ halts on input $\mathbf{x}.\mathbf{q}$, then $\mathbf{u} = [\mathbf{x}.\mathbf{q}]$ is the unique minimizer of* (5).
(ii) *If $\mathcal{N}$ does not halt on input $\mathbf{x}.\mathbf{q}$, then* (5) *has no feasible vector $\mathbf{u}$.*

Proof. By (4), any complete execution w for $\mathbf{x}.\mathbf{q}$ satisfies $F(|w|) \le |w|$; and conversely, if $F(\mathbf{u}) \le \mathbf{u}$ then any $w \in A^*$ such that $|w| = \mathbf{u}$ is a complete execution for $\mathbf{x}.\mathbf{q}$.

If $\mathcal{N}$ halts on input $\mathbf{x}.\mathbf{q}$ then the odometer $[\mathbf{x}.\mathbf{q}]$ is defined as $|w|$ for a complete *legal* execution w. By the least action principle (Lemma 4.3), for any complete execution w' we have $|w|_a \le |w'|_a$ for all $a \in A$. Thus

$$[\mathbf{x}.\mathbf{q}]_a = \min\{|w'|_a \,:\, w' \text{ is a complete execution for } \mathbf{x}.\mathbf{q}\}$$

so $[\mathbf{x}.\mathbf{q}]$ is the coordinatewise minimum of all feasible vectors.

If $\mathcal{N}$ does not halt on input $\mathbf{x}.\mathbf{q}$, then there does not exist a complete execution for $\mathbf{x}.\mathbf{q}$, so there is no feasible vector. □

For any nondecreasing $F : \mathbb{N}^A \to \mathbb{N}^A$, there is an abelian network $\mathcal{N}_F$ that solves the corresponding optimization problem (5). Its underlying graph is a single vertex v with a loop $e = (v, v)$. It has state space $Q = \mathbb{N}^A$, transition function $T_v(a, \mathbf{q}) = \mathbf{q} + 1_a$ and message passing function satisfying

$$|T_e(a, \mathbf{q})| = F(\mathbf{q} + 1_a) - F(\mathbf{q})$$

for all $a \in A$ and $\mathbf{q} \in Q$. For the input we take $\mathbf{x} = F(\mathbf{0})$ and $\mathbf{q} = \mathbf{0}$.

Remark 4.10. In general the problem (5) is nonlinear, but in the special case of a *toppling network* it is equivalent to a linear integer program of the following form.

$$\begin{aligned} &\text{Minimize} && \mathbf{c}^T\mathbf{v} \\ &\text{subject to} && \mathbf{v} \in \mathbb{N}^A \quad \text{and} \quad L\mathbf{v} \geq \mathbf{b}. \end{aligned} \tag{6}$$

Here $\mathbf{c} \in \mathbb{R}^A$ has all coordinates positive; L is the Laplacian matrix (§3.2); and $\mathbf{b} = \mathbf{x} - \mathbf{r} + \mathbf{1}$ where $\mathbf{x}_a$ is the number of chips input at a and $\mathbf{r}_a$ is the threshold of a. The coordinate $\mathbf{v}_a$ of the minimizer is the number of times a topples. To see the equivalence of (5) and (6) for toppling networks, note that F takes the following form for a toppling network:

$$F(\mathbf{u}) = \mathbf{x} + (D - L)\left\lfloor D^{-1}\mathbf{u}\right\rfloor$$

where D is the diagonal matrix with diagonal entries $\mathbf{r}_a$, and $\lfloor\cdot\rfloor$ denotes the coordinatewise greatest integer function. Using that $D-L$ is a nonnegative matrix, one checks that $\mathbf{u} = \mathbf{x} + (D - L)\mathbf{v}$ is feasible for (5) if and only if $\mathbf{v}$ is feasible for (6). We remark that for a *general* integer matrix the problem of whether (6) has a feasible vector $\mathbf{v}$ is NP-complete (see, for example, [Pap81]) but that the Laplacian L for an abelian network is constrained to have off-diagonal entries ≤ 0. See [FL16] for a discussion of the computational complexity of this problem when L is a directed graph Laplacian.

5. Concluding Remarks

We indicate here a few directions for further research on abelian networks. Other directions are indicated in the sequels [BL16a, BL16b].

5.1. **Asynchronous graph algorithms.** Chan, Church and Grochow [CCG14] have shown that a rotor network can detect whether its underlying graph is planar (with edge orderings respecting the planar embedding). Theorem 4.6 shows that abelian networks can compute asynchronously, and Theorem 4.9 gives an example of something they can compute. It would be interesting to explore whether abelian networks can perform computational tasks like shortest path, pagerank, image restoration and belief propagation. We note one practical deficiency of abelian networks: In the words of an anonymous referee, "determining when a network has finished computing requires some computational overhead" outside the network.

5.2. **Abelian networks with shared memory.** In §2.1 we have emphasized that abelian networks do not rely on shared memory. Yet there are quite a few examples of processes with a global abelian property that do. Perhaps the simplest is *sorting by adjacent transpositions*: suppose G is a finite path and each vertex v has state space $Q_v = \mathbb{Z}$. The processors now live on the edges: for each edge $e = (v, v+1)$ the processor $\mathcal{P}_e$ acts by swapping the states $q(v)$ and $q(v+1)$ if $q(v) > q(v+1)$. This example does not fit our definition of abelian network because the processors of edges $(v-1, v)$ and $(v, v+1)$ share access to the state $q(v)$. Indeed, from our list of five goals in §2 this example satisfies items (a)–(c) only: The final output is always sorted, and the run time does not depend on the execution, but the local run times do depend on the execution. For instance, when G is a path with three vertices and two edges s_1 and s_2, both $s_1 s_2 s_1$ and $s_2 s_1 s_2$ are complete legal executions for the initial state $(3, 2, 1)$. The edge s_1 performs two swaps in the first execution, but only one swap in the second execution.

What is the right definition of an abelian network with shared memory? Examples could include the numbers game of Mozes [Moz90], k-cores of graphs and hypergraphs, Wilson cycle popping [Wil96] and its extension by Gorodezky and Pak [GP14], source reversal [GP00] and cluster firing [H$^+$08, Bac12, CPS12].

5.3. **Nonabelian networks.** The work of Krohn and Rhodes [KR65, KR68] led to a detailed study of how the algebraic structure of monoids relates to the computational strength of corresponding classes of automata. It would be highly desirable to develop such a dictionary for classes of automata *networks*. Thus one would like to weaken the abelian property and study networks of solvable automata, nilpotent automata, etc. Such networks are nondeterministic — the output depends on the order of execution — so their theory promises to be rather different from that of abelian networks. It could be fruitful to look for networks that exhibit only limited nondeterminism. A concrete example is a sandpile network with annihilating particles and antiparticles, studied by Robert Cori (unpublished) and in [CPS12] under the term "inverse toppling."

Acknowledgments

The authors thank Spencer Backman, Olivier Bernardi, Deepak Dhar, Anne Fey, Sergey Fomin, Christopher Hillar, Michael Hochman, Alexander Holroyd, Benjamin Iriarte, Mia Minnes, Ryan O'Donnell, David Perkinson, James Propp, Leonardo Rolla, Farbod Shokrieh, Allan Sly and Peter Winkler for helpful discussions. We thank two anonymous referees for insightful comments that improved the paper.

This research was supported by an NSF postdoctoral fellowship and NSF grants DMS-1105960 and DMS-1243606, and by the UROP and SPUR programs at MIT.

References

[AR08] F. C. Alcaraz and V. Rittenberg, Directed abelian algebras and their applications to stochastic models *Phys. Rev. E* **78**:041126, 2008. arXiv:0806.1303

[APR09] F. C. Alcaraz, P. Pyatov and V. Rittenberg, Two-component abelian sandpile models *Phys. Rev. E* **79**:042102, 2009. arXiv:0810.4053

[Bac12] Spencer Backman, A bijection between the recurrent configurations of a hereditary chip-firing model and spanning trees. arXiv:1207.6175

[BTW87] Per Bak, Chao Tang and Kurt Wiesenfeld, Self-organized criticality: an explanation of the $1/f$ noise, *Phys. Rev. Lett.* **59**(4):381–384, 1987.

[BW03] Itai Benjamini and David B. Wilson, Excited random walk, *Elect. Comm. Probab.* **8**:86–92, 2003.

[Big99] Norman L. Biggs, Chip-firing and the critical group of a graph, *J. Algebraic Combin.* **9**(1):25–45, 1999.

[BW97] Norman L. Biggs and Peter Winkler, Chip-firing and the chromatic polynomial. Technical Report LSE-CDAM-97-03, London School of Economics, Center for Discrete and Applicable Mathematics, 1997.

[BLS91] Anders Björner, László Lovász and Peter Shor, Chip-firing games on graphs, *European J. Combin.* **12**(4):283–291, 1991.

[BL16a] Benjamin Bond and Lionel Levine, Abelian networks II. Halting on all inputs. *Selecta Math.*, to appear. arXiv:1409.0169

[BL16b] Benjamin Bond and Lionel Levine, Abelian networks III. The critical group. *J. Alg. Combin.*, to appear. arXiv:1409.0170

[CGHL16] Elisabetta Candellero, Shirshendu Ganguly, Christopher Hoffman and Lionel Levine, Oil and water: a two-type internal aggregation model. arXiv:1408.0776

[CPS12] Sergio Caracciolo, Guglielmo Paoletti and Andrea Sportiello, Multiple and inverse topplings in the abelian sandpile model. *The European Physical Journal-Special Topics* **212**(1)23–44, 2012. arXiv:1112.3491

[CCG14] Melody Chan, Thomas Church and Joshua A. Grochow, Rotor-routing and spanning trees on planar graphs. *Int. Math. Res. Not.* (2014): rnu025. arXiv:1308.2677

[CMS13] Yao-ban Chan, Jean-François Marckert, and Thomas Selig, A natural stochastic extension of the sandpile model on a graph, *J. Combin. Theory A* **120**(7):1913–1928, 2013. arXiv:1209.2038

[CP05] Denis Chebikin and Pavlo Pylyavskyy, A family of bijections between G-parking functions and spanning trees, *J. Combin. Theory A* **110**(1):31–41, 2005. arXiv:math/0307292

[CS06] Joshua Cooper and Joel Spencer, Simulating a random walk with constant error, *Combin. Probab. Comput.* **15**:815–822, 2006. arXiv:math/0402323

[CL03] Robert Cori and Yvan Le Borgne. The sand-pile model and Tutte polynomials. *Adv. in Appl. Math.* **30**(1-2):44–52, 2003.

[DD14] Rahul Dandekar and Deepak Dhar, Proportionate growth in patterns formed in the rotor-router model, *J. Stat. Mech.* **11**:P11030, 2014. arXiv:1312.6888

[DR04] Arnoud Dartois and Dominique Rossin, Height-arrow model, *Formal Power Series and Algebraic Combinatorics*, 2004.

[Dha90] Deepak Dhar, Self-organized critical state of sandpile automaton models, *Phys. Rev. Lett.* **64**:1613–1616, 1990.

[Dha99a] Deepak Dhar, The abelian sandpile and related models, *Physica A* **263**:4–25, 1999. arXiv:cond-mat/9808047

[Dha99b] Deepak Dhar, Studying self-organized criticality with exactly solved models, 1999. arXiv:cond-mat/9909009

[Dha99c] Deepak Dhar, Some results and a conjecture for Manna's stochastic sandpile model, *Physica A* **270**:69–81, 1999. arXiv:cond-mat/9902137

[Dha06] Deepak Dhar, Theoretical studies of self-organized criticality, *Physica A* **369**:29–70, 2006.

[DSC09] Deepak Dhar, Tridib Sadhu and Samarth Chandra, Pattern formation in growing sandpiles, *Europhysics Lett.* **85**:48002, 2009. arXiv:0808.1732

[DF91] Persi Diaconis and William Fulton, A growth model, a game, an algebra, Lagrange inversion, and characteristic classes, *Rend. Sem. Mat. Univ. Pol. Torino* **49**(1):95–119, 1991.

[DRS10] Ronald Dickman, Leonardo T. Rolla and Vladas Sidoravicius, Activated random walkers: facts, conjectures and challenges, *J. Stat. Phys.* **138**(1-3):126–142, 2010. arXiv:0910.2725

[DFS08] Benjamin Doerr, Tobias Friedrich and Thomas Sauerwald, Quasirandom rumor spreading, *Proceedings of the nineteenth annual ACM-SIAM symposium on discrete algorithms (SODA '08)*, pages 773–781, 2008. arXiv:1012.5351

[Ent87] Aernout C. D. van Enter, Proof of Straley's argument for bootstrap percolation. *J. Stat. Phys.* **48**(3-4):943–945, 1987.

[Eri96] Kimmo Eriksson, Chip-firing games on mutating graphs, *SIAM J. Discrete Math.* **9**(1):118–128, 1996.

[FLP10] Anne Fey, Lionel Levine and Yuval Peres, Growth rates and explosions in sandpiles, *J. Stat. Phys.* 138:143–159, 2010. arXiv:0901.3805

[FMR09] Anne Fey, Ronald Meester, and Frank Redig, Stabilizability and percolation in the infinite volume sandpile model, *Ann. Probab.* **37**(2):654-675, 2009. arXiv:0710.0939

[Fre93] Vidar Frette, Sandpile models with dynamically varying critical slopes, *Phys. Rev. Lett.* **70**:2762–2765, 1993.

[FL13] Tobias Friedrich and Lionel Levine, Fast simulation of large-scale growth models, *Random Struct. Alg.* **42**:185–213, 2013. arXiv:1006.1003.

[FL16] Matthew Farrell and Lionel Levine, CoEulerian graphs, *Proc. Amer. Math. Soc.*, to appear. arXiv:1502.04690

[Gab94] Andrei Gabrielov, Asymmetric abelian avalanches and sandpiles. Preprint, 1994. http://www.math.purdue.edu/~agabriel/asym.pdf

[GM97] Eric Goles and Maurice Margenstern, Universality of the chip-firing game, *Theoret. Comp. Sci.* **172**(1): 121–134, 1997.

[GP00] Eric Goles and Erich Prisner, Source reversal and chip firing on graphs, *Theoret. Comp. Sci.* **233**:287–295, 2000.

[GP14] Igor Gorodezky and Igor Pak, Generalized loop-erased random walks and approximate reachability, *Random Struct. Alg.* **44**(2):201–223, 2014.

[GLPZ12] Giuliano Pezzolo Giacaglia, Lionel Levine, James Propp and Linda Zayas-Palmer, Local-to-global principles for the hitting sequence of a rotor walk, *Electr. J. Combin.* **19**:P5, 2012. arXiv:1107.4442

[Hol03] Alexander E. Holroyd, Sharp metastability threshold for two-dimensional bootstrap percolation, *Probab. Theory Related Fields*, **125**(2):195–224, 2003. arXiv:math/0206132

[H$^+$08] Alexander E. Holroyd, Lionel Levine, Karola Mészáros, Yuval Peres, James Propp and David B. Wilson, Chip-firing and rotor-routing on directed graphs, in *In and out of equilibrium 2*, pages 331–364, Progress in Probability **60**, Birkhäuser, 2008. arXiv:0801.3306

[HLW16] Alexander E. Holroyd, Lionel Levine and Peter Winkler, Abelian logic gates. arXiv:1511.00422

[HP10] Alexander E. Holroyd and James G. Propp, Rotor walks and Markov chains, in *Algorithmic Probability and Combinatorics*, American Mathematical Society, 2010. arXiv:0904.4507

[HS11] Wilfried Huss and Ecaterina Sava, Rotor-router aggregation on the comb, *Electr. J. Combin.* **18**(1):P224, 2011. arXiv:1103.4797

[KL10] Wouter Kager and Lionel Levine, Rotor-router aggregation on the layered square lattice, *Electr. J. Combin.* **17**:R152, 2010. arXiv:1003.4017

[KS05] Harry Kesten and Vladas Sidoravicius, The spread of a rumor or infection in a moving population, *Ann. Probab.* **33**:2402–2462, 2005. arXiv:math/0312496

[KS08] Harry Kesten and Vladas Sidoravicius, A shape theorem for the spread of an infection, *Ann. Math.* **167**:701–766, 2008. arXiv:math/0312511

[KR65] Kenneth Krohn and John Rhodes, Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines, *Trans. Amer. Math. Soc.* **116**:450–464, 1965.

[KR68] Kenneth Krohn and John Rhodes, Complexity of finite semigroups, *Ann. Math.* **88**:128–160, 1968.

[Lev14] Lionel Levine, Threshold state and a conjecture of Poghosyan, Poghosyan, Priezzhev and Ruelle, *Comm. Math. Phys.*, to appear, 2014. arXiv:1402.3283

[LP09] Lionel Levine and Yuval Peres, Strong spherical asymptotics for rotor-router aggregation and the divisible sandpile, *Potential Anal.* **30**:1–27, 2009. arXiv:0704.0688

[Man91] S. S. Manna, Two-state model of self-organized criticality, *J. Phys. A: Math. Gen.* **24**:L363, 1991.

[MM11] Juan Andres Montoya and Carolina Mejia, The computational complexity of the abelian sandpile model, 2011. http://matematicas.uis.edu.co/jmontoya/sites/default/files/notas-ASM.pdf

[MN99] Cristopher Moore and Martin Nilsson. The computational complexity of sandpiles. *J. Stat. Phys.* **96**:205–224, 1999.

[Moz90] Shahar Mozes, Reflection processes on graphs and Weyl groups, *J. Comb. Theory A* **53**(1):128–142, 1990.

[Ost03] Srdjan Ostojic, Patterns formed by addition of grains to only one site of an abelian sandpile, *Physica A* **318**:187–199, 2003.

[Pap81] Christos H. Papadimitriou, On the complexity of integer programming, *Journal of the ACM* **28**(4):765–768, 1981.

[PDDK96] V. B. Priezzhev, Deepak Dhar, Abhishek Dhar and Supriya Krishnamurthy, Eulerian walkers as a model of self-organised criticality, *Phys. Rev. Lett.* **77**:5079–5082, 1996. arXiv:cond-mat/9611019

[PS04] Alexander Postnikov and Boris Shapiro, Trees, parking functions, syzygies, and deformations of monomial ideals. *Trans. Amer. Math. Soc.* **356**(8):3109–3142, 2004. arXiv:math.CO/0301110

[PS13] Wesley Pegden and Charles K. Smart, Convergence of the Abelian sandpile, Duke Mathematical Journal **162**(4):627–642, 2013. arXiv:1105.0111

[Pro03] James Propp, Random walk and random aggregation, derandomized, 2003. http://research.microsoft.com/apps/video/default.aspx?id=104906

[Pro10] James Propp, Discrete analog computing with rotor-routers. *Chaos* **20**:037110, 2010. arXiv:1007.2389

[RS12] Leonardo T. Rolla and Vladas Sidoravicius, Absorbing-state phase transition for driven-dissipative stochastic dynamics on $\mathbb{Z}$, *Inventiones Math.* **188**(1):127–150, 2012. arXiv:0908.1152

[Tse90] Paul Tseng, Distributed computation for linear programming problems satisfying a certain diagonal dominance condition, *Mathematics of Operations Research* **15**(1):33–48, 1990.

[WLB96] Israel A. Wagner, Michael Lindenbaum and Alfred M. Bruckstein, Smell as a computational resource — a lesson we can learn from the ant, *4th Israeli Symposium on Theory of Computing and Systems*, pages 219–230, 1996.

[Wil96] David B. Wilson, Generating random spanning trees more quickly than the cover time, *28th Annual ACM Symposium on the Theory of Computing (STOC '96)*, pages 296–303, 1996.

Benjamin Bond, Department of Mathematics, Stanford University, Stanford, California 94305. http://stanford.edu/~benbond

Lionel Levine, Department of Mathematics, Cornell University, Ithaca, NY 14853. http://www.math.cornell.edu/~levine