Computing by Programmable Particles

Joshua J. Daymude[1][0000−0001−7294−5626], Kristian Hinnenthal[2][0000−0001−9464−295X], Andréa W. Richa[1], and Christian Scheideler[2]

[1] Computer Science, CIDSE, Arizona State University, Tempe, AZ, USA, {jdaymude,aricha}@asu.edu

[2] Department of Computer Science, Paderborn University, Paderborn, Germany, {krijan,scheidel}@mail.upb.de

Abstract. The vision for *programmable matter* is to realize a physical substance that is scalable, versatile, instantly reconfigurable, safe to handle, and robust to failures. Programmable matter could be deployed in a variety of domain spaces to address a wide gamut of problems, including applications in construction, environmental science, synthetic biology, and space exploration. However, there are considerable engineering and computational challenges that must be overcome before such a system could be implemented. Towards developing efficient algorithms for novel programmable matter behaviors, the *amoebot model* for self-organizing particle systems and its variant, *hybrid programmable matter*, provide formal computational frameworks that facilitate rigorous algorithmic research. In this chapter, we discuss distributed algorithms under these models for shape formation, shape recognition, object coating, compression, shortcut bridging, and separation in addition to some underlying algorithmic primitives.

Keywords: programmable matter, self-organizing particle systems, distributed algorithms

1 Introduction

The idea of a robot that can transform into different shapes and sizes (e.g., Hasbro's *Transformers*) or multitudes of tiny mobile robots that collectively build structures, move objects, or even act as weapons or shields (e.g., Disney's *Big Hero 6* or Marvel's *Black Panther*) have become as ubiquitous and iconic in science fiction futurism as flying cars, holographic video conferencing, and teleportation. Yet this vision does not exist solely in fiction; since the 1990s, many researchers spanning across biology, chemistry, physics, mathematics, and computer science have contributed significant results towards realizing versatile, scalable robotic systems. In 1991, Toffoli and Margolus [54] defined *programmable matter* as a physical computing medium composed of simple, homogeneous nodes that can be (*i*) assembled into "lumps" of arbitrary size, (*ii*) dynamically reconfigured into any regular structure that grows at most polynomially, (*iii*) interactively controlled by user input or environmental stimuli, and (*iv*) accessed in real time for observation, analysis, or modification.

The vision for programmable matter is to realize a physical substance that is scalable, versatile, instantly reconfigurable, safe to handle, and robust to failures. Programmable matter could be deployed in numerous domain spaces to address a wide gamut of problems: in construction, it could be used as a self-repairing building material or as a dynamically reconfigurable support scaffolding; in environmental science, it could be used to locate and metabolize pollutants at the micro-scale; in biological processes, it could aid in the construction and maintenance of nanoscale structures or even boost healing by artificially transporting and applying medicine where it's most needed; in robotics, it could be used to sustain long-term missions in isolated or hazardous environments where it would be difficult for a human to intervene.

There are formidable challenges to realizing programmable matter both from the engineering and computational perspectives. In this chapter, we abstractly consider programmable matter as a collection of simple computational entities that must coordinate at the individual level to achieve useful behaviors at the system level. Using the amoebot model and hybrid programmable matter model as our computational frameworks, we will present a series of distributed algorithms for programmable matter and give rigorous theoretical results regarding their correctness and efficiency.

1.1 Related Work

When considering the various models and implementations of programmable matter, one can differentiate between *passive* and *active* systems. In passive systems, individual units of programmable matter cannot control their own movements, instead relying on their structural properties and interactions with their environment for locomotion. They may, in some cases, have limited computational abilities to make decisions and communicate. Prominent examples of passive systems include population protocols [2], molecular computing and tile self-assembly models [26, 40], and slime molds [7, 44].

Our focus is primarily on active systems, where the individual units can control their actions and movements to achieve some task. Examples of physical active systems include swarm robotics [47] as well as self-reconfigurable modular robotics [59]. These systems seek similar coordinated behaviors as those considered in the amoebot model, but use robots that often have significantly more powerful sensing and communication abilities. Among the theoretical models of active systems, the *nubot model* from molecular programming [58] and *metamorphic robots* [12, 56] have the most similarities to the amoebot model, including their representations of space and their emphasis on simple, local computational units. However, they include some capabilities (e.g., rigid body movements in the nubot model) that prohibit a direct translation between the models.

The amoebot model for self-organizing particle systems (fully described in Section 2.1) envisions programmable matter as a system of simple, homogeneous *particles* that have only local communication and vision, constant-size memory, and no global sense of direction. This model was introduced to facilitate rigorous algorithmic research on programmable matter systems, and has since served as the computational framework for many theoretical investigations and even one experimental study [48].

Hybrid programmable matter (Section 6) combines the active and passive approaches by considering a passive structure of connected tiles that can be reconfigured by a collection of active robots. When considering only the passive tiles, this model shares many similarities with the tile self-assembly models mentioned before, where tiles bond to each other based on predefined "glues" (see, e.g., [26,40]). On the other hand, the active robots in the hybrid setting are very similar to the particles of the amoebot model, with the added capability of lifting and moving tiles. When considering only the robots' movements on a static tile structure, hybrid programmable matter reduces to an instance of the *mobile agents on graphs* model, where problems such as gathering/rendezvous [41], intruder caption [6], and graph searching and exploration [13, 28] have been studied extensively. *DNA nanomachines* offer a promising realization of hybrid programmable matter, and are capable of walking on one- and two-dimensional surfaces [35,39,57], transporting cargo [51,53], and acting as the head of a finite automaton on an input tape [45].

1.2 Chapter Organization

In Section 2, we define the amoebot model for programmable matter, including the rationale behind its modeling choices and a list of common model extensions. Section 3 contains four algorithmic primitives under the amoebot model — direct read/write communication, leader election, the spanning forest primitive, and distributed binary counters — that are utilized by the algorithms of Sections 4 and 5. The shape formation and object coating algorithms of Section 4 are largely deterministic, while the algorithms for compression, shortcut bridging, and separation in Section 5 are fully stochastic. Hybrid programmable matter is defined in Section 6, and algorithms for shape formation and shape recognition in this hybrid setting are described in Section 7. A summary of the chapter and an outline of future research are given in Section 8.

For clarity and brevity, we do not give any proofs of the theoretical results in this chapter and occasionally omit algorithm details that detract from a clear understanding of an algorithm's main ideas. However, we cite all underlying publications in their respective sections and encourage the interested reader to read further.

2 The Amoebot Model

The *amoebot model* is an abstract computational model of programmable matter intended to enable rigorous algorithmic analysis of collective systems at the nano-scale. Originally proposed as "amoeba-inspired self-organizing particle systems" in [25], the model was polished and formally announced as the *amoebot model* in [18]. It has since undergone many updates and changes over the years to support new settings and considerations, but has kept to the same core principles throughout. Here, we give a complete description of the current model in Section 2.1, provide some intuition behind its details in Section 2.2, and describe its common extensions in Section 2.3.

2.1 Model Description

In the amoebot model, programmable matter consists of individual, homogeneous computational elements called *particles*. Any structure that a particle system $\mathcal{P}$ can form is represented as a subgraph of an infinite, undirected graph $G = (V, E)$ where V represents all positions a particle can occupy relative to its structure and E represents all atomic movements a particle can make. Each node in V can be occupied by at most one particle at a time. In the *geometric amoebot model*, it is further assumed that $G = G_\Delta$, where G_Δ is the triangular lattice[3] (see Fig. 1a). Fixing the position of some particle, G_Δ represents the discretization of space relative to this particle and the possible atomic movements between these discrete positions. This discretization can be conceptualized as a tiling of two-dimensional space; G_Δ corresponds to the hexagonal tiling (Fig. 1a).

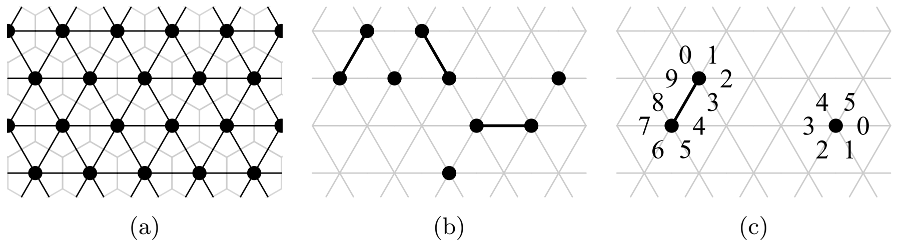


Fig. 1: (a) A section of the triangular lattice G_Δ (black) and its dual, the hexagonal tiling (gray). (b) Expanded and contracted particles (black dots) on G_Δ (gray lattice). Particles with a black line between their nodes are expanded. (c) Two particles with different offsets for their port labels.

Each particle occupies either a single node in V (i.e., it is *contracted*) or a pair of adjacent nodes in V (i.e., it is *expanded*), as in Fig. 1b. Particles move via a series of *expansions* and *contractions*: a contracted particle can expand into an unoccupied adjacent node to become expanded, and completes its movement by contracting to once again occupy a single node. An expanded particle's *head* is the node it last expanded into and the other node it occupies is its *tail*; a contracted particle's head and tail are both the single node it occupies.

Two particles occupying adjacent nodes are said to be *neighbors*. Neighboring particles can coordinate their movements in a *handover*, which can occur in one

[3] Some papers refer to G_Δ as the *equilateral triangular grid graph* G_{eqt} or the triangular lattice $\mathit{\Gamma}$.

of two ways. A contracted particle P can initiate a "push" handover with an expanded neighbor Q by expanding into a node occupied by Q, forcing it to contract. Alternatively, an expanded particle Q can initiate a "pull" handover with a contracted neighbor P by contracting, forcing P to expand into the node it is vacating.

Each particle keeps a collection of ports — one for each edge incident to the node(s) it occupies — that have unique labels from its own local perspective. Although each particle is *anonymous*, lacking a unique identifier, a particle can locally identify any given neighbor by its labeled port corresponding to the edge between them. The particles are assumed to have a common *chirality* (i.e., notion of clockwise direction), which allows each particle to label its ports in clockwise order. However, particles do not share a coordinate system or global compass and may have different offsets for their port labels, as in Fig. 1c.

Each particle has a constant-size local memory partitioned into internal memory and one addressed memory for each neighboring node. A particle can only write into its own memory, but can read all the addressed and internal memories of its neighbors for communication. A particle's internal memory stores whether it is expanded or contracted, its local port labeling (including which ports are incident to its head versus its tail), and any other application-specific information. Particles do not have any global information and — due to the limitation of constant-size memory — cannot know the total number of particles in the system nor any estimate of this value.

The system progresses through *atomic actions* according to the standard $\mathcal{A}$SYNC model of computation from distributed computing (see, e.g., [36]). A classical result under this model states that for any concurrent asynchronous execution of atomic actions, there exists a sequential ordering of actions producing the same end result, provided conflicts that arise in the concurrent execution are resolved. In the amoebot model, an atomic action corresponds to the activation of a single particle. Once activated, a particle can (*i*) perform an arbitrary, bounded amount of computation involving information it reads from its local memory and its neighbors' memories, (*ii*) write to its local memory, and (*iii*) perform at most one expansion or contraction. Conflicts involving simultaneous particle expansions into the same unoccupied node are assumed to be resolved arbitrarily such that at most one particle moves to some unoccupied node at any given time[4]. Thus, while in reality many particles may be active concurrently, it suffices when analyzing algorithms under the amoebot model to consider a sequence of activations where only one particle is active at a time. The resulting activation sequence is assumed to be *fair*: for any inactive particle P at time t, P will be activated again at some time $t' > t$. An *asynchronous round* is complete once every particle has been activated at least once. Unless otherwise specified, a *round* refers to an asynchronous round.

[4] A particle can only write into its own memory in the amoebot model's publishing-based communication, so no conflicts of concurrent writes to the same memory location are possible.

2.2 Rationale

We now provide some intuition behind the amoebot model and its details. At the highest level, we seek to answer the question: *what complex, collective behaviors are achievable by extremely simple, restricted programmable particles?* The amoebot model was designed to restrict the capabilities of the individual particles as much as possible in hope of developing algorithms that could be useful to many implemented systems across task domains and scales. For example, it may seem unnecessarily restrictive to a swarm robotics engineer to consider robots with only constant-size memory, since commodity hardware often supports $\mathcal{O}(\log n)$ or even $\mathcal{O}(n)$ memory, for reasonable swarm sizes n. However, this assumption makes algorithms under the amoebot model applicable both to swarm systems with extra memory as well as systems at the more restrictive micro- or nano-scales. Moreover, the resulting systems can be arbitrarily scalable to any number of units, a desirable property for programmable matter.

Communication. Restricting particle communication and vision to immediate neighbors captures the local nature of unit interactions in programmable matter. The amoebot model's communication scheme is a publishing-based version of standard message passing protocols in the $\mathcal{A}$SYNC model. If a particle P wants to send some information x to its neighbor Q, it writes x to its addressed memory facing Q. Particle P must wait for Q to activate, read x, and acknowledge its receipt before P can know the information was communicated. Situations where multiple neighbors try to send information to the same particle concurrently must be resolved by the recipient. (See Section 2.3 for a simpler variant of this publishing-based communication model).

Chirality. The chirality assumption, which states that particles have a common sense of clockwise direction, is reasonable in many settings. Having a shared chirality is essentially equivalent to the system's ability to break spatial symmetry, such as distinguishing between "up" and "down". This is usually fairly simple to decide; for example, if a particle system were deployed in any medium subject to gravity, the system's top and bottom would be trivially distinguishable. Recent results by Di Luna et al. suggest that this assumption may not be necessary for all applications [23, 24]. We will discuss this further at the end of Section 3.1.

Connectivity. A particle system is *connected* if the subgraph of G induced by the occupied nodes of V is also connected. This notion does not imply any particular kind of connectivity in a physical programmable matter system; connections could be physical bonds, points of contact between neighboring units, or even wireless communication links. Although the amoebot model does not require that a system remains connected, this is often a desirable property that its algorithms maintain. If a particle system disconnects, there is little hope the resulting components could ever reconnect. Since each particle can only see and communicate with its immediate neighbors and does not have a global compass, disconnected components have no way of knowing their relative positions and thus cannot intentionally move toward one another to reconnect.

Space. To aid system connectivity, we chose the triangular lattice G_Δ to represent space in the geometric amoebot model. In the other regular two-dimensional lattices (square and hexagonal), particles are often forced to momentarily disconnect from the rest of the system even to perform moves as simple as shifting "around" another particle by one position (see Fig. 2).

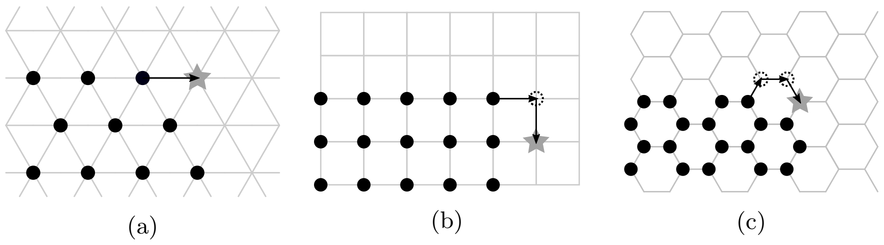


Fig. 2: Illustration of a particle moving "around" a neighboring particle to get to the next position on the surface, depicted as a gray star, on the (a) triangular lattice G_Δ, (b) square lattice, and (c) hexagonal lattice.

Movement. Modeling movements as expansions, contractions, and handovers also has roots in connectivity. Splitting a particle's movement from one node to another into an expansion and a contraction can be thought of as a look-ahead mechanism in which the particle reserves a space and examines its new surroundings before deciding whether or not to go through with the movement. This is vaguely similar to human walking, where we put one foot forward before completely shifting our weight to take another step. By looking ahead, a particle can determine whether its move might break system connectivity before committing to it. Handovers, as described in Section 2.1, allow the system to maintain connectivity while moving. These movements were not simply included for convenience; there are tasks — such as moving through a static tunnel of width 1 — which are impossible without handovers if connectivity must also be maintained at all times.

2.3 Extensions

Many papers on the amoebot model utilize techniques and assumptions that extend the core model described in Section 2.1. These extensions can be thought of as modules which combine and repackage core model features into useful, higher-level functionalities.

Leader Particle. Some algorithms under the amoebot model assume the existence of a unique *leader particle* (or *seed*) at initialization which can be used to coordinate the rest of the system. This assumption is reasonable, since the leader election algorithm in [16] can be used as a preprocessing step for obtaining this leader particle (see Section 3.1 for details). Notably, a leader can impose its labeling scheme on all other particles in the system to establish a global compass. In the other direction, if the system establishes a global compass without a leader, it becomes trivial to solve leader election (e.g., "elect the south-most, west-most particle"). Thus, any algorithm under the amoebot model which hopes to run in sublinear time — faster than the leader election algorithm of [16], which matches the worst-case lower bound — must restrict itself only to local compasses.

Static Objects. An *object* is a finite, connected, static set of nodes $O \subset V$. We model objects as a collection of contracted particles in a special *object state* which do not move, communicate, or perform any computation throughout the execution of an algorithm. These object particles simply represent some fixed surface or entity in space and are usually not considered members of the particle system. For example, the coating algorithm in Section 4.3 assumes the existence of an object to be coated, and the stochastic algorithm for shortcut bridging in Section 5.2 considers two objects that the particle system must bridge between. Particles can differentiate between object and non-object particles.

Node Differentiation. It is sometimes useful to consider physical spaces that have heterogeneous properties, such as a marsh with both land and water locations or a tree-lined path with some parts exposed to sunlight and others in shade. *Node differentiation* models these differences by considering an assignment $\Phi : V \to \{1, \dots, k\}$ that maps each node of the graph G to one of k types, where k is a constant. A contracted particle occupying a node $u \in V$ can read $\Phi(u)$, but cannot alter it. Analogously, an expanded particle occupying adjacent nodes $u, v \in V$ can read but not write $\Phi(u)$ and $\Phi(v)$. This assignment Φ need not be static, but any dynamics controlling its evolution should reflect changes in the environment and not the actions of the particle system. This extension should not be used to encode global information for the particles to utilize.

Token Passing. A *token* is a constant-size message that can be passed from particle to particle. More specifically, a particle P can pass a token t to a neighbor Q by publishing t to its addressed memory facing Q, wait for Q to read, copy, and acknowledge t, and then delete t from its own memory. Due to the constant-size memory constraint of the amoebot model, each particle can hold only a constant number of tokens at once. Rules on whether tokens must be passed in a pipelined fashion, merge together, or interact in more complex ways may vary by each algorithm's need.

Many algorithms under the amoebot model use token passing to relay information beyond a particle's immediate neighborhood. For example, the leader election algorithm in Section 3.1 uses tokens extensively to facilitate communication and competition between candidates which are often far from one another. Although token passing often makes algorithms more complex, especially when tokens of different protocols interact, its flexibility and direct compatibility with the model make it a viable tool for many applications.

Direct Write Communication. In the publishing-based communication scheme of the amoebot model, a particle only has write access to its own memory, but has read access to all addressed and internal memories of its neighbors. As described in Section 2.2, if a particle P wants to send some information x to its neighbor Q, a multi-step process of publishing and acknowledging x is initiated. However, this can greatly complicate the presentation of algorithms that rely heavily on writes (e.g., any algorithm that uses token passing), as every write is split over multiple particle activations that must be locally synchronized.

One could greatly simplify communication descriptions by employing a variant of the publishing-based communication model. In the *direct write communication* model, a particle can do the following in one activation: (*i*) perform an arbitrary, bounded amount of computation involving information it reads from its local memory and its neighbors' memories, (*ii*) write to its local memory, (*iii*) directly write updates to at most one neighbor's memory, and (*iv*) perform at most one expansion or contraction. However, in the asynchronous setting of the amoebot model, this direct write communication allows for *write conflicts*, where multiple particles concurrently attempt to write to the internal memory of a common neighbor. These conflicts are assumed to be resolved arbitrarily such that each particle is involved in at most one write at any given time (i.e., at any given time, either a particle P is writing to a neighbor, a neighbor is writing to P, or neither). This direct write communication model can be faithfully emulated by the publishing-based communication scheme of the amoebot model via a simple emulation primitive, described fully in [17].

Random Number Generation. It is often assumed that each particle has access to random bits with which it can generate random values. However, due to the constant-size memory constraint of the core model, each particle can only hold a constant number of random bits and thus can only store constant precision random values. It is left to the algorithm designer to ensure that constant precision is sufficient for their application; see [1, 11] for examples of such arguments.

Agent Emulation. It can be useful for a particle to run multiple instances of an algorithm at once, especially in settings where it needs to participate in different phases of an algorithm concurrently. In the leader election algorithm in Section 3.1, for example, a particle executes up to three instances concurrently (one per boundary it is incident to). To accommodate this, a single particle can emulate up to a constant number of *agents*, each with its own memory, running its own instance of a given algorithm. This respects the constant-size memory constraint as each algorithm instance requires only constant memory and each particle emulates at most a constant number of agents.

3 Algorithmic Primitives for the Amoebot Model

Several algorithmic primitives exist under the amoebot model, acting as reusable building blocks for the algorithms of Sections 4 and 5. These include *leader* *election* (Section 3.1), which is used by other algorithms as a black box for obtaining a unique leader particle, the *spanning forest primitive* (Section 3.2), which is used to locally organize and move a particle system along a specified path, and *distributed binary counters* (Section 3.3), which enable n particles to collectively emulate a binary counter storing unsigned values up to $2^n - 1$.

3.1 Leader Election

To date, there have been two approaches to the classical problem of leader election under the amoebot model: the token-based approach by Derakhshandeh and Daymude et al. [16, 22], and the erosion-based approach by Di Luna et al. [23]. We will focus primarily on the algorithm in [16], which simplifies and extends the algorithm and analysis of [22] to a fully local, distributed, asynchronous setting. At a glance, the algorithm in [16] elects a leader in $\mathcal{O}(L)$ asynchronous rounds with high probability[5], where L is the length of the outer boundary of the system and w.h.p. applies to both correctness and runtime. A brief comparison to the erosion-based approach is made at the end of this section.

Problem Description An algorithm is said to solve the leader election problem if for any connected particle system of initially contracted particles, eventually a single particle *irreversibly* declares itself the *leader* (e.g., by setting a dedicated bit in its memory) and no other particle ever declares itself to be the leader. The running time of a leader election algorithm is defined to be the number of asynchronous rounds until a leader is declared. The algorithm is not required to terminate for particles other than the leader, though a leader could broadcast its existence to the rest of the system to trigger termination, if desired.

Algorithm We begin with a high-level overview of the algorithm's six *phases*. These phases are not strictly synchronized among each other, i.e., at any point in time, different parts of the particle system may execute different phases. Furthermore, a particle can be involved in the execution of multiple phases at the same time. The first phase is *boundary setup*. In this phase, each particle locally checks whether it is part of a *boundary* of the particle system. Only the particles on a boundary participate in the leader election. Particles occupying a common boundary organize themselves into a directed cycle. The remaining phases operate on each boundary independently. In the *segment setup* phase, the boundaries are divided into *segments*. Each particle flips a fair coin: particles that flip heads become *candidates* and compete for leadership whereas particles that flip tails become *non-candidates* and assist the candidates in their competition. A segment consists of a candidate and all subsequent non-candidates along the boundary up to the next candidate. The *identifier setup* phase assigns a random identifier to each candidate. The identifier of a candidate is stored distributedly among the particles of its segment. In the *identifier comparison* phase, the

[5] An event occurs *with high probability (w.h.p.)* if the probability of success is at least $1 - 1/n^c$, where $c > 0$ is a constant; in our setting, n is the number of particles.

candidates compete for leadership by comparing their identifiers using a token passing scheme. Whenever a candidate sees an identifier higher than its own, it revokes its candidacy. Whenever a candidate sees its own identifier, the *solitude verification* phase is triggered. In this phase, the candidate checks whether it is the last remaining candidate on the boundary. If so, it initiates the *boundary identification* phase to check if it occupies the unique *outer boundary* of the system. In that case, it becomes the leader; otherwise, it revokes its candidacy.

Boundary Setup. The boundary setup phase organizes the particle system into a set of *boundaries*, as in Fig. 3a. Let A be the set of nodes in G_Δ that are occupied by particles, and consider the graph $G_\Delta|_{V\setminus A}$ induced by the unoccupied nodes in G_Δ. An *empty region* is a maximal connected component of $G_\Delta|_{V\setminus A}$. Let $N(R)$ be the neighborhood of an empty region R in G_Δ; that is, $N(R) = \{u \in V \setminus R : \exists v \in R \text{ such that } (u, v) \in E\}$. Note that by definition, all nodes in $N(R)$ are occupied by particles. We refer to $N(R)$ as the *boundary* of the particle system corresponding to R. Since A corresponds to a finite set of particles, exactly one empty region has infinite size while any others have finite size. The boundary corresponding to the infinite empty region is the unique *outer boundary*, and any boundary corresponding to a finite empty region is an *inner boundary*.

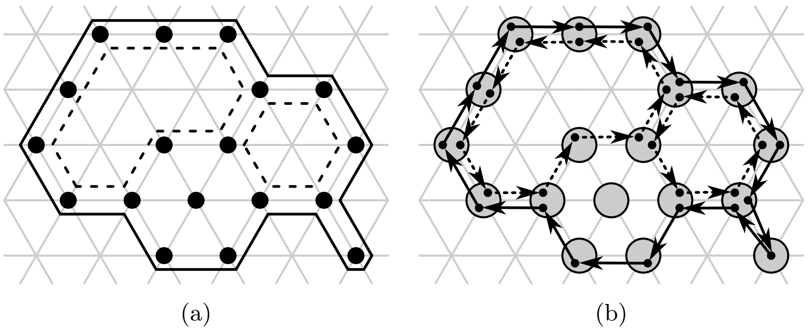


Fig. 3: (a) Boundaries of a particle system. The solid line represents the unique outer boundary and the dashed lines represent the inner boundaries. (b) Agents (black dots) of particles (gray circles) organized into directed cycles along the boundaries of (a).

Next, the particles of each boundary organize into a directed cycle. Upon its first activation, each particle determines its place in these cycles using only local information as follows. If P has no neighbors, then since the particle system is connected, P must be the only particle. So P immediately declares itself the leader and terminates. If P is surrounded (i.e., it has six neighbors), P is not part of any boundary and simply terminates.

Otherwise, the neighborhood of P must contain at least one occupied and one unoccupied node. For each maximal, connected sequence of unoccupied nodes

S in the neighborhood of P (of which there can be at most three; see Fig. 4), let P act as a distinct *agent* a_S that independently executes the remainder of the leader election algorithm. This ensures that the leader election algorithm runs on each boundary independently, since P cannot locally decide which such sequences belong to which boundary. Each agent a_S chooses the particle immediately clockwise (resp., counterclockwise) of S to be its *successor* (resp., *predecessor*).

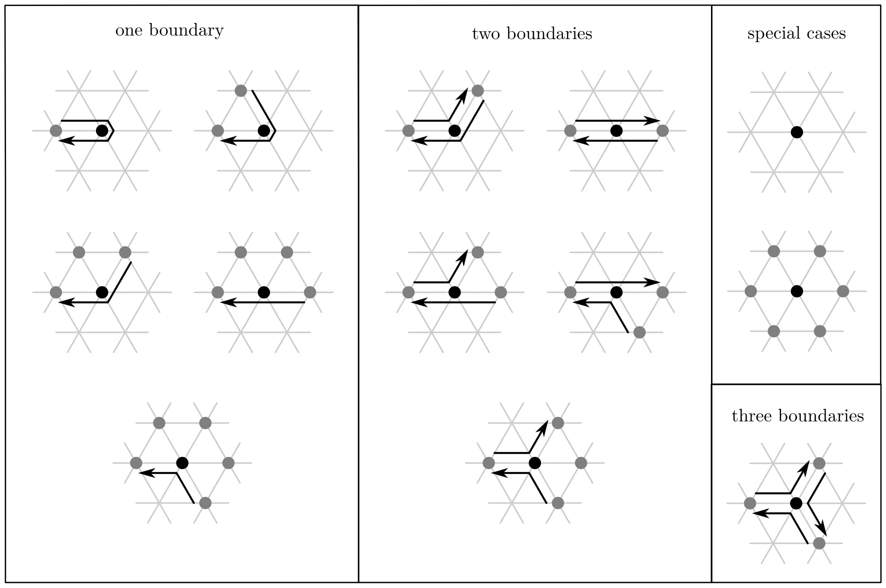


Fig. 4: Possible results (up to rotation) of the boundary setup phase depending on the neighborhood of a particle. For each boundary, the depicted arrow starts at the particle's predecessor and ends at its successor.

Fig. 4 shows all possible neighborhoods of a particle (up to rotation) and the corresponding predecessor and successor assignments of its agents. These assignments organize the set of all agents into disjoint cycles spanning the boundaries of the particle system (see Fig. 3b). It is possible that a particle can occur up to three times on the same boundary as different agents. While this property can be ignored for most of the remaining phases, it will remain a cause for special consideration in the solitude verification phase.

Segment Setup. All remaining phases (including this one) execute exclusively on each boundary independently. Therefore, we only consider a single boundary for the remainder of the algorithm description. The segment setup phase divides the boundary into disjoint "segments" as follows. Each agent flips a fair coin; those that flip heads become *candidates* and those that flip tails become *non-*

candidates. In the following phases, candidates compete for leadership while non-candidates assist this competition. A *segment* is a maximal sequence of agents $(a_1, a_2, \ldots, a_k)$ such that a_1 is a candidate, a_i is a non-candidate for $i > 1$, and a_i is the successor of a_{i-1} for $i > 1$. We refer to the segment starting at a candidate c as the segment of c (denoted c.seg) and denote its length as $|c.\text{seg}|$. In the following phases, each candidate uses its segment as a distributed memory.

Identifier Setup. After the segments have been set up, each candidate generates a random *identifier* for use in the competition of the next phase by assigning a random digit to each agent in its segment. Note that the term identifier is slightly misleading in that two distinct candidates can have the same identifier.

To generate its random identifier c.id, a candidate c sends a token (recall token passing from Section 2.3) along its segment in the direction of the boundary. As the token traverses the segment, it assigns a value chosen uniformly at random from $\{0, 1\}$ to each visited agent[6]. The resulting identifier is a binary number consisting of $|c.\text{seg}|$ bits where c holds the most significant bit and the last agent of c.seg holds the least significant bit.

After generating c.id, each candidate c creates a copy of c.id that is stored in *reverse digit order* in its segment. This copy is used in the next phase to compare against the identifiers of other candidates. The token that generated c.id is reused in creating the reversed copy as follows. It first reads the digit of the last agent of the segment c.seg. It is then passed to the beginning of c.seg (to candidate c) and stores a copy of that digit. It then reads the digit of c and is passed back to the end of the segment where it stores a copy of that digit. This continues in a similar fashion with the second to last and second agent and so on until c.id is completely copied. Finally, the token is passed to c to signal the end of this identifier setup phase.

Identifier Comparison. During the identifier comparison phase, the candidate agents use their identifiers to compete with each other. When comparing identifiers of different lengths, longer identifiers are defined to be higher than shorter ones; otherwise, the identifiers are compared directly. A candidate with the highest identifier eventually progresses to the solitude verification phase, described in the next section, while any candidate with a lower identifier withdraws its candidacy. To achieve the comparison, the non-reversed copies of the identifiers remain stored in their respective segments while the reversed copies move backwards along the boundary as a sequence of tokens. More specifically, a *digit token* is created for each digit of a reversed identifier. A digit token created by the last agent of a segment is marked as a *delimiter token*. We define the *token sequence* of a candidate c as the sequence of digit tokens created by the agents in c.seg. Once created, digit tokens traverse the boundary against the direction of the cycle spanning it. Each agent is allowed to hold at most two tokens at a time, and can forward at most one token per activation. Tokens are not allowed

[6] In [16], the digits are chosen uniformly at random from $[0, r-1]$ where r is a fixed constant. The resulting identifiers are numbers with radix r.

to overtake each other. Furthermore, an agent can only receive a token after it creates its own digit token. This ensures that token sequences of distinct candidates remain separated and the tokens within a token sequence maintain their relative order along the boundary.

We give a high level description of the token passing scheme for identifier comparison, illustrated in Fig. 5, using two successive candidates c and c'. Here, the token sequence of c' is compared with c.id. Initially, agents are *active* and tokens are *inactive*, as in Fig. 5a. Whenever a token is forwarded by a candidate into a new segment, the token becomes active, as in Fig. 5b. When an active agent receives an active token, they *match*, storing the result of their digit comparison ($<$, $>$, or $=$) in the agent and both becoming inactive (Fig. 5c). A matched (inactive) token is then simply passed on without incident until reaching c, who reactivates it when forwarding it into the next segment (Fig. 5d).

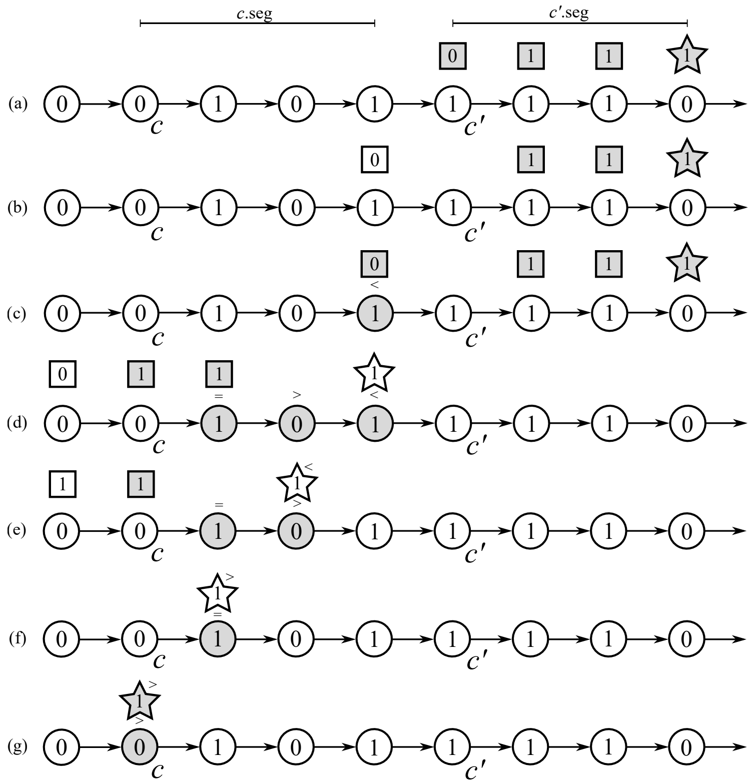


Fig. 5: Illustration of identifier comparison between c.id $= 0101$ and c'.id $= 1110$. Active elements are white while inactive elements are gray. The digit tokens are depicted as squares, while the star depicts the special delimiter token.

The delimiter token of c', say $d_{c'}$, eventually enters c.seg (Fig. 5d). As $d_{c'}$ traverses c.seg, it sees the results of the previous digit comparisons from least to most significant and updates its record of the overall comparison accordingly (Fig. 5e–f). When candidate c eventually receives $d_{c'}$, it locally compares identifier lengths as follows. If c already matched with a non-delimiter token of c', then $|c.\text{seg}| < |c'.\text{seg}|$ and c withdraws its candidacy. If the delimiter token $d_{c'}$ already matched with some agent before c, then $|c.\text{seg}| > |c'.\text{seg}|$ and c remains a candidate. Finally, if c matches with $d_{c'}$ (as in Fig. 5g), we have $|c.\text{seg}| = |c'.\text{seg}|$.

In this last case, c must use the record of the overall comparison stored in $d_{c'}$ in combination with its own digit comparison with $d_{c'}$ to decide the comparison result. If $c.\text{id} < c'.\text{id}$, c withdraws its candidacy. If $c.\text{id} > c'.\text{id}$, c remains a candidate. Finally, if $c.\text{id} = c'.\text{id}$, c may have just compared against its own identifier and thus initiates the solitude verification phase to determine if it is the only remaining candidate on the boundary.

As an aside, candidates who withdraw candidacy still reactivate inactive tokens when forwarding them. The delimiter token also resets inactive agents as it passes over them, preparing them for future identifier comparisons (Fig. 5e–g).

Solitude Verification. The goal of the solitude verification phase is for a candidate c to check whether it is the last remaining candidate on its boundary. Solitude verification is triggered during the identifier comparison phase whenever a candidate detects equality between its own identifier and the identifier of a token sequence that traversed its segment. Such a token sequence can either be a candidate's own or that of another candidate with the same identifier. Once the solitude verification phase is started, it runs in parallel to the identifier comparison phase and does not interfere with it.

A necessary (but insufficient) condition for candidate c to be the only remaining candidate on its boundary is if the next candidate along the boundary occupies the same node as c. The following algorithm checks this condition. Treat the directed edges of the boundary cycles as vectors in the two-dimensional Euclidean plane. The next candidate along the boundary, say c', occupies the same node as c if and only if the sum of the vectors corresponding to boundary edges from c to c' is 0. To decide if this is the case in a local manner, c defines a local two-dimensional coordinate system (e.g., as in Fig. 6) and uses a token passing scheme to check whether the x-components of the vectors sum to 0. An analogous scheme is used for the y-components, which runs in parallel.

First, c sends an *activation token* in the direction of the cycle towards the next candidate. Whenever the token moves in the positive (resp., negative) direction of the locally defined x-axis, it creates a *positive token* (resp., *negative token*). These tokens are sent back towards c. Positive and negative tokens move independently of each other, but cannot overtake tokens of the same type. Once these tokens either reach c or cannot move any closer to c, they become *settled.* Note that once all positive (or negative) tokens are settled, they form a consecutive sequence whose length corresponds to the number of tokens, as in Fig. 6a–c.

When the activation token reaches the next candidate, it reverses its movement back towards c, staying behind any positive or negative tokens that have

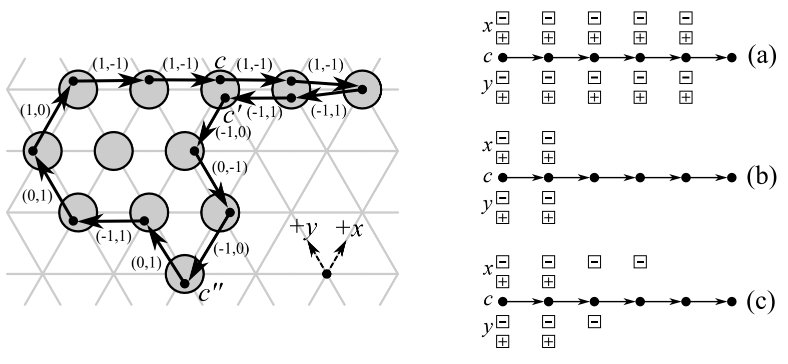


Fig. 6: The local vector construction used in solitude verification. The logical positions of the positive and negative tokens after they have settled are shown on the right, for the situation where the only remaining candidate(s) is/are (a) c, (b) c and c', and (c) c and c''.

not settled. Once they have settled, deciding whether the vectors sum to 0 can be done in a local manner: the vectors from c to the next candidate sum to 0 if and only if the length of the positive and negative token sequences are equal; i.e., if the last settled tokens in these sequences are held by the same agent. For example, this is the case in Fig. 6a–b, but not in Fig. 6c. Thus, the activation token simply observes whether or not this is the case and then moves back towards c to report the result. On the way, it deletes all positive and negative tokens.

However, as hinted before, this is not sufficient to decide whether c is the last remaining candidate on the boundary. For agents belonging to the same particle on the same boundary, as with c and c' in Fig. 6b, the vectors will sum to 0 despite there being at least two agents remaining. To handle this case, each particle assigns a locally unique identifier from $\{1, 2, 3\}$ to each of its agents in an arbitrary way. When the activation token reaches the next candidate, it reads its agent identifier and carries this information back to c. It is not hard to see that c is the last remaining candidate on the boundary if and only if the vectors sum to 0 and the agent identifier stored in the activation token equals the agent identifier of c.

Finally, we address the interaction between the solitude verification and identifier comparison phases. If solitude verification is triggered for a candidate c while c is still performing a previously triggered execution of solitude verification, it ignores this trigger and simply continues with the already ongoing execution. Candidate c may also be eliminated by the identifier comparison phase while it is performing solitude verification. In this case, c waits for the ongoing solitude verification to finish and only then withdraws its candidacy.

Boundary Identification. Once a candidate c determines that it is the only remaining candidate on its boundary, it initiates the boundary identification phase to check if it lies on the unique outer boundary. If so, the particle acting as c

declares itself the leader; otherwise, c revokes its candidacy. This phase uses the fact that, due to the boundary setup phase, the outer boundary is oriented clockwise while any inner boundary is oriented counterclockwise (see Fig. 3b).

To distinguish between clockwise and counterclockwise oriented boundaries, a candidate c sends a token along its boundary that sums the angles of the turns it takes according to Fig. 7, storing the results in a counter α. When the token returns to c, there are two cases: $\alpha = 360°$ for the unique outer boundary, and $\alpha = -360°$ for any inner boundary. We encode α as $k \in \mathbb{Z}$ such that $\alpha = k \cdot 60°$. It is sufficient to store k modulo 5 so that we have $k = 1$ for the outer boundary and $k = 4$ for an inner boundary, requiring only three bits of memory.

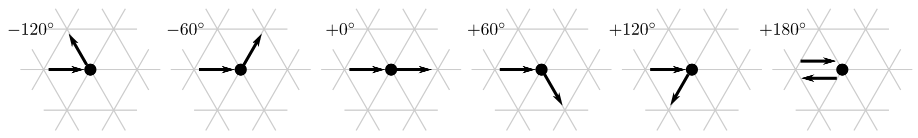


Fig. 7: Determining α. The incoming and outgoing arrows represent the directions the token enters and leaves an agent, respectively, up to rotation.

Analysis We now briefly discuss the correctness and runtime of the leader election algorithm, stating the main results.

Correctness. Recall that for a leader election algorithm to be correct, a single particle must irreversibly declare itself the leader and no other particle can ever do so. The boundary identification phase ensures that no candidate on an inner boundary can ever declare itself the leader, so the analysis focuses only on the outer boundary. For a single candidate agent c^* on the outer boundary to become the leader, it must have the highest identifier on the boundary; i.e., $c^*.\mathrm{id} > c.\mathrm{id}$ for every other candidate $c \neq c^*$ on the outer boundary. The analysis upper bounds the probability of another candidate having the same highest identifier by an inverse polynomial in n, the number of particles in the system. Barring this event, a unique candidate c^* with the highest identifier emerges. It eventually compares its own identifier with itself, triggering solitude verification. When solitude verification succeeds, boundary identification will indicate that c^* is on the unique outer boundary. Thus, we can state the following result.

Theorem 1. *The algorithm correctly solves the leader election problem, w.h.p.*

Runtime. The first two phases of the algorithm (boundary setup and segment setup) are performed by each particle in their first activation, and thus are completed in the first round. The identifier setup phase takes $\mathcal{O}(\ell^2)$ rounds for a segment of length ℓ, and the length of a segment on the outer boundary is $\mathcal{O}(\log n)$, w.h.p. Combining these results, the identifier setup phase completes for all candidates on the outer boundary in $\mathcal{O}(\log^2 n)$ rounds. Token sequences of the identifier comparison phase are then shown to traverse the outer boundary in $\mathcal{O}(L)$ rounds, where L is the number of agents on the outer boundary. By a similar argument, the solitude verification phase is shown to take $\mathcal{O}(\ell)$ rounds, where ℓ is the number of agents between the current and next candidates. Finally, the boundary identification phase on the outer boundary it proven to take $\mathcal{O}(L)$ rounds. Therefore, all together, we obtain the following.

Theorem 2. *The algorithm solves the leader election problem in $\mathcal{O}(L) = \mathcal{O}(n)$ rounds, w.h.p., where L is the number of agents on the outer boundary and n is the number of particles in the system.*

Comparison to Leader Election by Erosion Di Luna et. al [23] take another approach to leader election. At a high level, all particles originally start as candidates. Using local rules that depend on the number and configuration of a particle's neighbors, a particle may decide to withdraw its candidacy. Importantly, these rules are carefully designed so that the set of candidate particles always forms a connected component. At the end of this *erosion* process, there are one to three remaining candidates in a symmetric configuration. They give a deterministic protocol for attempting to break this symmetry, but it is possible that it may fail and a simple coin-flipping scheme must be used instead.

In comparison to the token-based algorithm described above, the erosion-based algorithm does not need long-range communication in the form of token passing, nor does it require that the system has a common chirality (i.e., notion of clockwise direction). However, it cannot handle particle systems which contain empty regions ("holes"), and was not shown to be extensible to situations where the particle system is moving, as is shown for the token-based algorithm in [21] (see Section 4.3 for more details). Their algorithm achieves the same runtime bound of $\mathcal{O}(n)$ rounds w.h.p., where n is the number of particles in the system.

3.2 Spanning Forest Primitive

Without a global compass or shared coordinate system, particles of a particle system must use some local mechanisms to coordinate their movements. Many algorithms under the amoebot model solve this problem using the *spanning forest primitive*, originally introduced in [22]. This primitive organizes the particle system into one or more "trees", each of which is composed of *follower* particles following a single *root* particle. The root is responsible for directing the movement of its tree; the followers simply perform a follow-the-leader protocol to trail along behind the root. For simplicity, we will present this primitive with respect to a single spanning tree; in general, this primitive executes on each tree of the spanning forest concurrently and independently.

Problem Description Consider an initially connected particle system $\mathcal{P}$ of contracted, *idle* particles, a designated *root* particle R, and a (simple) path

$\mathcal{L} \subset G_\Delta$ beginning at the node occupied by R. We desire for the particle system to traverse the path $\mathcal{L}$ exactly without becoming disconnected.

Two caveats are needed for considering this problem in practice. First, the root particle R is not usually predetermined. A root can either arises as the result of some local mechanism (e.g., "if adjacent to an object, become a root") or can be elected using leader election (Section 3.1) as a subprimitive. Second, $\mathcal{L}$ is usually not given explicitly; it is more often the path the root particle R traverses as it executes some local algorithm. Nevertheless, for this standalone presentation of the primitive, we assume R and $\mathcal{L}$ are given.

Algorithm Particles can be in one of three states: idle, follower, or root. All particles (except the unique root R), are initially idle. When an idle particle P is activated, it checks if it has a follower or root neighbor Q. If so, P sets its parent pointer P.parent $\leftarrow Q$ and becomes a follower; otherwise, P does nothing.

When a follower particle P is activated, it first checks whether it is contracted or expanded. If P is contracted and its parent Q (pointed at by P.parent) is expanded, P expands in a push handover with Q, forcing Q to contract. In doing so, it may need to update P.parent so it still points to Q. Otherwise, if P is expanded, there are two cases. First, if P has no idle neighbors and no children — i.e., no neighbors Q such that Q.parent points to the tail of P — P simply contracts. Otherwise, if P has a child Q that is contracted, P contracts in a pull handover with Q, forcing Q to expand. Similar to the push handover, P may need to update Q.parent so it still points to P.

When the root particle R is activated, it also checks whether it is contracted or expanded. The rules for when it is expanded are the same as for followers: if it has no idle neighbors and no children, it contracts; otherwise, if it has a contracted child, it performs a pull handover. If R is contracted, on the other hand, it checks if it has reached the end of the given path $\mathcal{L}$. If so, it does nothing; otherwise, it simply expands into the next node of $\mathcal{L}$.

Example. Consider an example run of the spanning forest primitive, illustrated in Fig. 8. All particles except the root R are initially idle (Fig. 8a). Particle P_1 has the root R as a neighbor and becomes a follower, setting P_1.parent $\leftarrow R$, while R expands into the next node of the path $\mathcal{L}$ (Fig. 8b). Eventually, all idle particles become followers; a handover occurs between R and P_1 (Fig. 8c). Root R moves into the final node of $\mathcal{L}$ via expansions and handovers that propagate out through the rest of the tree (Fig. 8d–8f). Eventually, all particles become contracted, and $\mathcal{L}$ has been traversed by the particle system (Fig. 8g).

Root Swaps. One caveat is necessary: it is possible that as R traverses $\mathcal{L}$, it may be blocked by another part of the particle system, as in Fig. 9a. In this case, it will not be able to expand into the next node of $\mathcal{L}$ since this node is occupied by another particle Q. Instead, R performs a *root swap* with Q (if Q is contracted), in which R transfers the contents of its memory to Q, promotes Q to become the new root, demotes itself to become a follower, and sets R.parent $\leftarrow Q$ (see

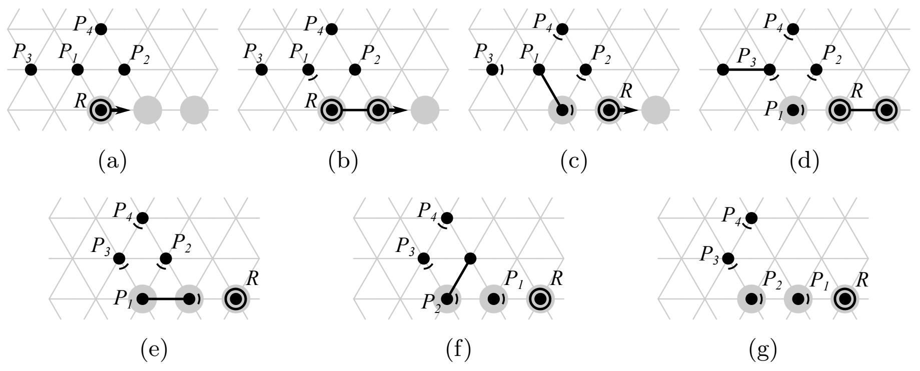


Fig. 8: Example of the spanning forest primitive. The root R is shown as a black dot with a black circle, and the path $\mathcal{L}$ it follows is shown in gray. The black arcs point from a follower's head to its parent, and expanded particles have a black line connecting their head and tail.

Fig. 9b). Note that this does not disrupt the tree structure of the particle system, even if Q was idle or has idle neighbors.

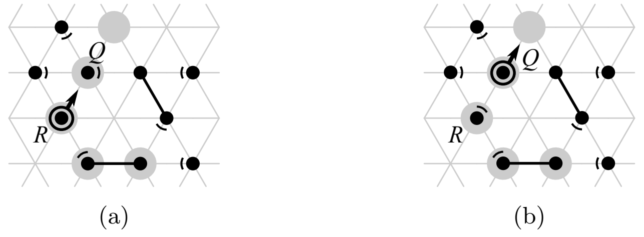


Fig. 9: Example of using a root swap to overcome a blocked path.

Analysis We now state the main correctness and runtime results for the spanning forest primitive.

Correctness. Two properties must hold for the particle system $\mathcal{P}$ to correctly traverse path $\mathcal{L}$: (*i*) $\mathcal{P}$ always makes eventual progress along $\mathcal{L}$ until $\mathcal{L}$ has been entirely traversed, and (*ii*) $\mathcal{P}$ never becomes disconnected. These are the *liveness* and *safety* conditions for the spanning forest primitive, respectively. Liveness depends on the root's ability to continue expanding and performing handovers along $\mathcal{L}$. In [21, 22], Derakhshandeh et al. prove that every expanded particle (including the root) eventually contracts. Thus, a contracted root will always be able to either expand into the next node of $\mathcal{L}$ or perform a root swap with the particle blocking it, since this blocking particle is also guaranteed to eventually be contracted. Safety follows more immediately. The only way $\mathcal{P}$ can become disconnected is if a particle with children or idle neighbors contracts outside a handover, breaking their connectivity to the tree. However, the protocol explicitly disallows this, so we have the following.

Theorem 3. *A particle system $\mathcal{P}$ following the spanning forest primitive will correctly traverse any given simple path $\mathcal{L}$.*

Runtime. A *dominance argument* [14, 20] is used to analyze the runtime of the spanning forest primitive. These arguments first consider a parallel execution of an algorithm where all particles make progress in lock-step. This is often much easier to reason about. For the spanning forest primitive, it is shown that in $\mathcal{O}(n)$ parallel rounds, all n particles are the root or its followers and the resulting tree forms a pattern of alternating expanded and contracted particles. Once in this configuration, the tree makes exactly one node of progress along $\mathcal{L}$ every 2 parallel rounds. Together, this implies that the spanning forest primitive traverses the entire path $\mathcal{L}$ in $\mathcal{O}(|\mathcal{L}|)$ parallel rounds.

Using careful case analysis, the concurrent, asynchronous execution of an algorithm is then shown to always make at least as much progress per round as its parallel counterpart. This implies that the runtime bound for the parallel execution is an upper bound on the runtime of the asynchronous execution. For the spanning forest primitive, this argument yields the following bound.

Theorem 4. *A particle system $\mathcal{P}$ following the spanning forest primitive will traverse a given path $\mathcal{L}$ in $\mathcal{O}(|\mathcal{L}|)$ rounds, where $|\mathcal{L}|$ is the length of path $\mathcal{L}$.*

3.3 Distributed Binary Counters

Many behaviors for programmable matter are realized by efficient algorithms under the amoebot model that only use constant-size state variables and messages. However, it can be useful in many applications to work with values that are on the order of $\mathcal{O}(\log n)$ or even $\mathcal{O}(n)$, where n is the number of particles in the system. An example of such values appeared in leader election (Section 3.1), where candidates generated identifiers of logarithmic length to compete for leadership. Other applications could include, for example, measuring the size of an object with a particle system in order to replicate its shape.

Due to the constant-size memory constraint of the amoebot model (Section 2.1), individual particles cannot keep these larger values in memory by themselves. However, with the help of a leader particle, a system of n particles can be organized into *distributed memory* in the form of a *distributed binary counter* that stores unsigned values up to $2^n - 1$ and supports increments and decrements by one as well as zero-testing. Porter and Richa first introduced an increment-only binary counter under the amoebot model in [42]; Daymude et al. extended this work to support decrements and zero-tests in [15].

Problem Description Consider an initially connected particle system $\mathcal{P}$ of contracted particles organized in a simple path $P_0, P_1, \ldots, P_{n-1}$ with a leader particle $\ell = P_0$ at its start. We desire for the particle system to self-organize into a distributed binary counter that supports increments and decrements by one (initiated by the leader ℓ). Additionally, the distributed binary counter should support reliable zero-testing, i.e., the leader ℓ should reliably be able to determine whether or not the counter's value is equal to zero.

Although the particles could move while maintaining the binary counter (e.g., according to the spanning forest primitive of Section 3.2), for ease of presentation we will assume they are static. It is also assumed that the leader ℓ never causes the counter to reach negative values; i.e., at any given time, ℓ has initiated at least as many increments as decrements. Presumably, ℓ could perform a zero-test before initiating a decrement to ensure the counter will not go negative, but for our presentation this nonnegativity assumption will suffice.

Algorithm Each particle P_i (for $0 \leq i < n$) has a bit value $P_i.\text{bit} \in \{\emptyset, 0, 1\}$, where $P_i.\text{bit} = \emptyset$ implies P_i is not part of the counter; i.e., it is beyond the most significant bit. A *final token* f represents the end of the counter. If a particle P_i holds f, the counter value is represented by the bits of each particle from the leader ℓ (holding the least significant bit) up to and including P_{i-1} (holding the most significant bit). Although not necessary for increments and decrements, utilizing f will allow the leader to zero-test the counter locally and efficiently.

The leader ℓ is responsible for initiating counter operations, while the other particles carry these operations out using only local information. To increment the counter, the leader ℓ simply generates an increment token c^+ (assuming it was not already holding a token). Now consider this operation from the perspective of any particle P_i holding a c^+ token, where $0 \leq i < n$. If $P_i.\text{bit} = 0$, P_i can simply consume c^+ and set $P_i.\text{bit} \leftarrow 1$. Otherwise, if $P_i.\text{bit} = 1$, this increment needs to be carried over to the next most significant bit. As long as P_{i+1} is not already holding a token, P_i can forward c^+ to P_{i+1} and set $P_i.\text{bit} \leftarrow 0$. Finally, if $P_i.\text{bit} = \emptyset$, this increment has been carried over past the counter's end, so P_i must also be holding the final token f. In this case, P_i simply forwards f to P_{i+1} and sets $P_i.\text{bit} \leftarrow 1$.

Decrements are similar; when considering this operation from the perspective of any particle P_i holding a c^- token, where $0 \leq i < n$, the cases for $P_i.\text{bit} \in \{0, 1\}$ are anti-symmetric to those for the increment, with two exceptions. First, we only allow P_i to consume c^- and set $P_i.\text{bit} \leftarrow 0$ if P_{i+1} is not also holding a c^-. While not necessary for the correctness of the decrement operation, this will enable conclusive zero-testing. Second, if $P_i.\text{bit} = 1$ and P_{i+1} is holding f, then P_i is the most significant bit. So this decrement shrinks the counter by one bit; thus, P_i consumes c^-, takes f from P_{i+1}, and sets $P_i.\text{bit} \leftarrow \emptyset$.

Finally, the zero-test operation: if P_1 is holding a decrement token c^- and $P_1.\text{bit} = 1$, ℓ cannot perform the zero-test conclusively. Otherwise, the counter value is 0 if and only if $\ell.\text{bit} = 0$, P_1 is holding the final token f, and P_1 is not holding an increment token c^+.

Analysis We only present the correctness and runtime results for the distributed binary counter primitive from [15], as these subsume those in [42].

Correctness. For the distributed binary counter primitive to be correct, it must eventually yield the same values as a centralized counter, assuming both are given the same sequence of operations as input. Because the algorithm ensures that the increment and decrement tokens are processed in the same order that their respective operations are initiated, the resulting distributed counter will correctly process an arbitrary number of increment and decrement operations (assuming the counter value remains in $\{0, 1, \ldots, 2^n - 1\}$). The zero-test operation is shown to always eventually be available (i.e., P_1 is not holding a decrement token c^- or $P_1.\text{bit} \neq 1$) and is reliable whenever it is available. Together, this yields:

Theorem 5. *A particle system $\mathcal{P}$ running the distributed binary counter primitive will maintain a distributed counter that eventually yields the same values as a centralized counter, given the same sequence of increment, decrement, and zero-test operations.*

Runtime. The following runtime bound is proven using a careful dominance argument (see, e.g., Section 3.2) applied to the progress of the increment and decrement tokens in the counter.

Theorem 6. *Given any nonnegative sequence of m operations, a particle system $\mathcal{P}$ running the distributed binary counter primitive processes all operations in $\mathcal{O}(m)$ rounds.*

Applications We briefly discuss some applications of the distributed binary counter primitive whose details are beyond the scope of this chapter. In [42], Porter and Richa give an algorithm using the increment-only counter for matrix-vector multiplication. For an $a \times b$ matrix (i.e., a rows and b columns), this algorithm requires $\mathcal{O}(ab)$ rounds to set up and an additional $\mathcal{O}(a + b)$ rounds to perform the multiplication. By performing a sequence of these matrix-vector multiplications, an $a \times b$ matrix can be multiplied by an $b \times c$ matrix in $\mathcal{O}(ab + c(a + b))$ rounds. Applications of these matrix multiplication algorithms to edge detection and color transformation in image processing are described in [42].

In [15], Daymude et al. use the distributed binary counter primitive to aid in *convex hull formation*, in which a particle system must seal an object using as few particles as possible. At a very high level, a leader particle traverses the surface of the given object, updating its estimate of the object's convex hull as it goes. Using its followers as a distributed binary counter, it stores the distances from its current position to each of the six half-planes constituting the convex hull. Once it completes its estimation, it can use these stored distances to lead its followers along the convex hull itself, eventually sealing the object as desired.

4 Deterministic Algorithms Under the Amoebot Model

This section is devoted to the deterministic algorithms under the amoebot model. These algorithms use the full capabilities of the amoebot model, including several of its extensions (Section 2.3) and algorithmic primitives (Section 3). We focus on two basic problems for programmable matter: forming a shape using all the particles in the system, and coating an object. In *basic shape formation* (Section 4.1), a follow-the-leader type protocol is used to construct regular shapes such as lines and hexagons. In *general shape formation* (Section 4.2), a more complex algorithm is given for constructing a much broader range of shapes. Finally, in *object coating* (Section 4.3), an algorithm is given for coating surfaces in one or more layers of particles as evenly as possible.

4.1 Basic Shape Formation

Shape formation is one of the most immediate and natural applications of programmable matter. A "lump" of programmable matter should be able to reconfigure into new shapes based on user input or autonomous sensing of its environment. Moreover, the final shape should scale with the size of the initial "lump". In *basic shape formation*, a particle system self-organizes to form regular, geometric shapes. Here, we focus on lines, hexagons, and triangles; however, the general framework can be applied to obtain other shapes as well. Line formation was first investigated in [22], while hexagon and triangle formation were introduced in [19]. Detailed versions of all three algorithms can be found in [52].

Problem Description An instance of a *shape formation* problem has the form $(I, \mathcal{G})$, where I is the initial configuration of the particle system and $\mathcal{G}$ is a set of goal configurations. An instance is *valid* if (i) I and all configurations of $\mathcal{G}$ are each connected, and (ii) I is composed of all contracted, idle particles and a unique *seed* particle. An algorithm solves a valid instance of the shape formation problem if, starting from initial configuration I, the algorithm terminates in a configuration of $\mathcal{G}$, after which all particles are contracted and no longer move.

The specific *line formation*, *hexagon formation*, and *triangle formation* problems simply define the desired set of goal configurations $\mathcal{G}$ as all configurations of straight lines, (almost) regular hexagons, and (almost) regular triangles, respectively. Note that, depending on the number of particles in I, the outermost layer of a hexagon or triangle may not be complete.

Algorithm Particles can be in one of four states: idle, follower, root, and retired. All particles are initially idle except the unique seed particle, which is always retired. At a high level, this algorithm constructs the desired shape one particle at a time in a snake-like fashion, starting at the seed. Thus, at any point in time before termination, the structure of retired particles partially forms the goal configuration.

The spanning forest primitive (Section 3.2) is used to organize the system. In basic shape formation, root particles always traverse the structure of retired particles in a clockwise direction, trailing their followers behind them. The most recent particle to retire (starting with the seed), say P, keeps a pointer P.retireDir to the next node to be filled by a retired particle. When a contracted root Q finds that it occupies this node (by seeing P.retireDir pointing to it), it retires, locally calculates the next node to be filled by a retired particle, and sets Q.retireDir to point to it. As other roots continue their traversal and their followers become roots when they touch the surface of retired particles, eventually all particles retire, forming the desired shape.

It remains to specify how a newly retired particle calculates the next node to add to the structure. Recall from Section 2.1 that each particle keeps a set of ports (one for each edge incident to the node(s) it occupies) labeled in clockwise order. Suppose a particle P has already retired and set P.retireDir, and another particle Q has just retired in the position referenced by P.retireDir. Let i be the port label of Q pointing to P. For line formation, Q sets Q.retireDir $\leftarrow$ $(i+3) \bmod 6$; this specifies that the next node to join the line should be opposite the direction of the existing structure, resulting in a straight line (see Fig. 10). For hexagon formation, Q sets Q.retireDir to the label of the first port clockwise from i that does not point to another retired particle; this causes the hexagon to be formed in counterclockwise order (see Fig. 11).

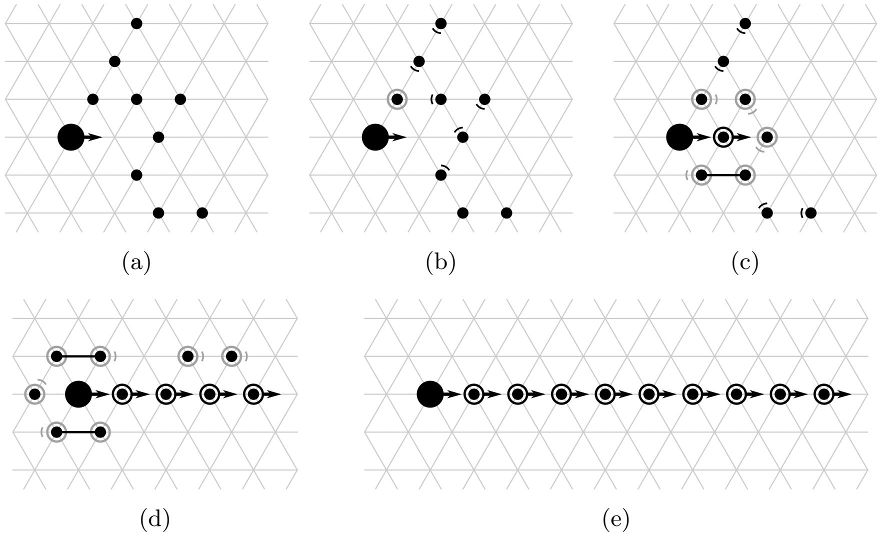


Fig. 10: Example of line formation with 10 particles, where the seed is depicted as a large black circle. Followers are shown with black arcs pointing to their parents, roots are shown with a gray circle, and retired particles are shown with a black circle. A retired particle's retireDir is shown as a black arrow.

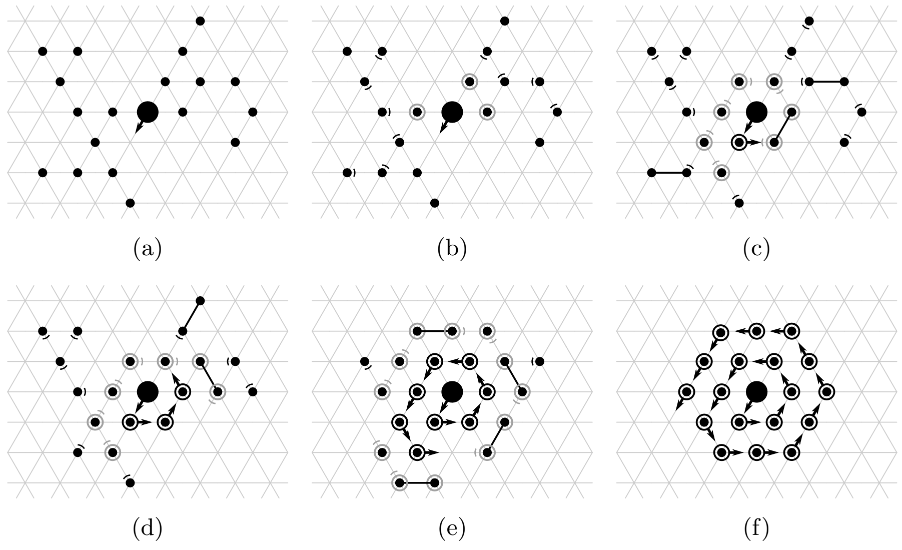


Fig. 11: Example of hexagon formation with 18 particles.

Finally, for triangle formation, a two-step mechanism is used[7]. Particle Q first checks the position immediately clockwise from i, say $j = (i + 1) \bmod 6$. If j does not point to a retired particle (as in Fig. 12c), Q is on a new side of the triangle and sets Q.retireDir $\leftarrow j$ to grow a new layer on this side. Otherwise, if j points to a retired particle (as in Fig. 12d), Q is simply extending an existing layer. So it sets Q.retireDir $\leftarrow (j + 2) \bmod 6$.

Analysis We now briefly motivate and state the main correctness and runtime results for basic shape formation.

Correctness. Showing that the basic shape formation algorithms correctly form their desired shapes can be split into guaranteeing (*i*) *liveness*; i.e., that the particle system $\mathcal{P}$ makes eventual progress towards terminating in the desired shape, and (*ii*) *safety*; i.e., that $\mathcal{P}$ never disconnects. Both liveness and safety follow from the correctness of the spanning forest primitive (Theorem 3), though proving that the final shape is the desired one relies on inspection of the retiring rules described above.

Theorem 7. *The basic shape formation algorithms correctly solve the line formation, hexagon formation, and triangle formation problems.*

[7] For this presentation, we use a simplified scheme that results in a triangle with the seed at its center; the original scheme given in [19, 52] is significantly more complex and results in a triangle with the seed at one vertex.

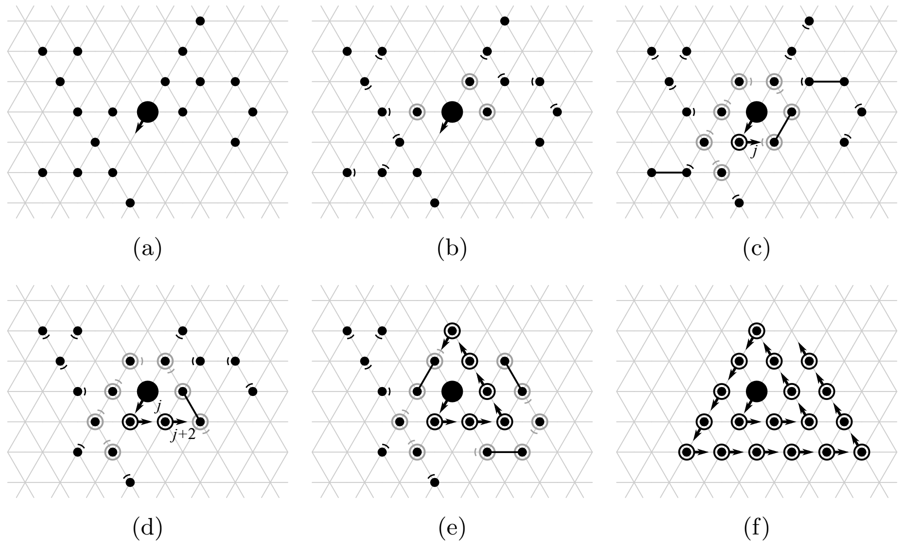


Fig. 12: Example of triangle formation with 18 particles. Unlike in Fig. 11, the resulting shape in (f) is not fully regular due to the number of particles. Additionally, note that (a)–(c) are the same as Fig. 11a–11c; the basic shape formation algorithms only differ in the way retired particles set their pointers.

Runtime. In [19, 22], Derakhshandeh et al. prove runtime bounds for the basic shape formation algorithms in terms of *work*, or the total number of particle movements required for an algorithm to terminate. It is shown that the worst-case amount of work required by any algorithm to solve any of the three basic shape formation problems (line formation, hexagon formation, or triangle formation) is $\Omega(n^2)$, where n is the number of particles in the system. The basic shape formation algorithm described above is shown to match this worst case bound in all three cases, terminating in $\mathcal{O}(n^2)$ work.

In his Ph.D. thesis, Strothmann uses a dominance argument (see, e.g., Section 3.2) to prove runtime bounds for these three algorithms in terms of asynchronous rounds [52]. Formally, his thesis gives the following theorem.

Theorem 8. *The total number of rounds required by the basic shape formation algorithms to solve the line formation, hexagon formation, or triangle formation problem is $\mathcal{O}(n)$, where n is the number of particles in the system.*

4.2 General Shape Formation

The basic shape formation algorithm of Section 4.1 has two major disadvantages that the *general shape formation algorithm* of this section seeks to alleviate. First, it can only construct simple shapes that are amenable to being built one particle at a time along a continuous path. Second, as stated in Theorem 8, it generally requires $\mathcal{O}(n)$ rounds to construct a shape of n particles, even when the shape could be formed more efficiently. The general shape formation algorithm [20,29] can form a much broader class of shapes and, assuming the particle system is well initialized, can do so as fast as any other local-control algorithm.

Problem Description Let S be a finite set of faces in the triangular lattice G_Δ. S is a *shape* if the faces of S are connected — i.e., there exists a path from any face in S to any other via pairs of faces that share a side — and the number of faces $s = |S|$ is constant. A shape S is *sequentially constructible*[8] if there exists a permutation $(a_1, a_2, \ldots, a_s)$ of the faces of S such that for every $1 \leq i \leq s$, the subset of faces $(a_1, a_2, \ldots a_i)$ is itself a shape S_i and the face a_i has a side on the outer boundary of S_i. This means that a sequentially constructible shape can be built by adding triangular faces to the outside of an intermediate shape.

Recall from basic shape formation (Section 4.1) that a shape formation problem $(I, \mathcal{G})$ defines the initial configuration I and the set of goal configurations $\mathcal{G}$. For the general shape formation problem, I forms a triangle consisting of n contracted particles, each with a binary representation of a sequentially constructible shape S to form stored in memory. The set of goal configurations $\mathcal{G}$ contains all transformations of S, where a transformation can be any combination of a translation, rotation by a multiple of $60°$, and isotropic scaling that still coincides with the triangular lattice G_Δ (see, e.g., Fig. 13). An algorithm solves the general shape formation problem if, starting from I, the algorithm terminates in a configuration of $\mathcal{G}$. Note that the final configuration can contain both expanded and contracted particles.

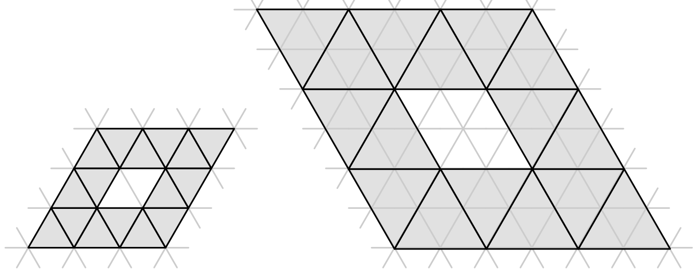


Fig. 13: A sequentially constructible shape consisting of 16 faces (left) and one of its transformations (right) involving a $120°$ rotation counterclockwise and an isotropic $2\times$ scale-up.

[8] The original publication on "universal" shape formation [20] claimed the algorithm could construct any shape with a constant number of faces. However, Gmyr corrected an oversight in this paper's analysis in his thesis [29] and, as a result, the class of shapes had to be restricted to sequentially constructible shapes.

Algorithm We present the general shape formation algorithm in several parts. First, we describe several movement primitives that are used throughout the algorithm. Next, we give highlights of an algorithm that transforms the initial triangle of particles into a useful intermediate structure. Finally, we describe the actual formation process. Our presentation prioritizes the algorithm's main ideas over its details, and we refer the interested reader to [20, 29] for a more thorough description.

Movement Primitives. Several movement primitives are used throughout the algorithm to move large sets of particles in a coordinated and efficient manner. The first of these is *chain movement*, in which a "chain" of particles moves along a simple path $\mathcal{L}$ without disconnecting. This is essentially the spanning forest primitive described in Section 3.2, but the set of particles moving along $\mathcal{L}$ is already organized into a simple path instead of a tree.

The remaining movement primitives operate on a set of contracted particles that form a triangle. Four primitives are available to such triangles: *expansion*, *contraction*, *rotation*, and *shift*. Fig. 14 depicts the first three of these primitives. An expansion of a triangle results in an *expanded triangle*, which is a rhombus composed of two triangles that share a side and are each the same size as the original triangle. A contraction of a triangle transforms an expanded triangle back into a triangle. Together, an expansion and contraction of a triangle rotates a triangle by 60° about one of its vertices. Finally, a shift of a triangle moves all of its particles by one node in a common direction.

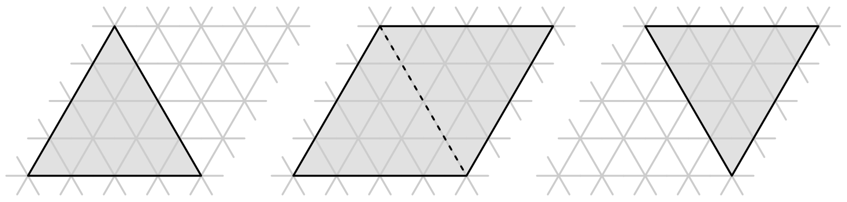


Fig. 14: Expansion, contraction, and rotation of a triangle. A triangle (left) can expand to form an expanded triangle (middle), which in turn can contract back to a triangle (right), rotating the original triangle by 60°.

At a high level, the triangle movement primitives are initiated by a *triangle coordinator* that occupies one of the triangle's three vertices. The coordinator organizes the particles in each row of its triangle into particle chains that can then be moved according to the chain movement primitive; for example, Fig. 15 depicts how the particle chains are moved in an expansion of a triangle. The coordinator initiates and completes these chain movements using a token passing scheme whose details we omit.

Reaching the Intermediate Structure. The first step of the general shape formation algorithm is to reconfigure the particle system from its initial triangle shape

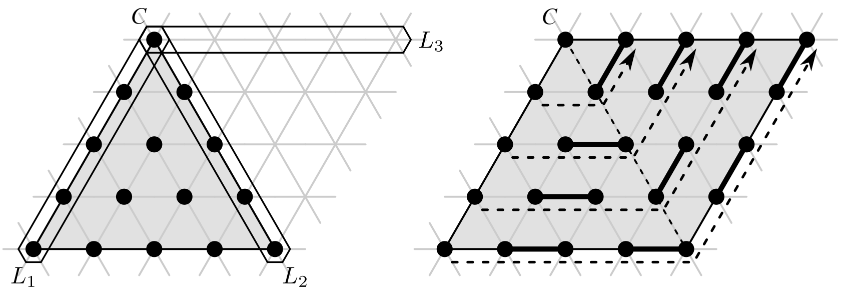


Fig. 15: A counterclockwise expansion of a triangle. The coordinator C is located at the top vertex of the triangle. The rows of the triangle expand as independently moving particle chains along the paths shown as dashed arrows.

to the intermediate structure depicted in Fig. 16. This intermediate structure is composed of Δ equilateral triangles of side length ℓ arranged in a line and a remainder composed of too few particles to form an additional triangle. All particles in the intermediate structure should be contracted.

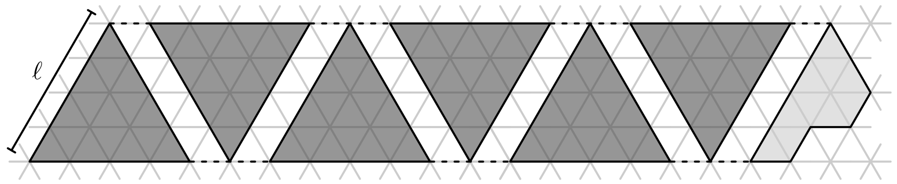


Fig. 16: The intermediate structure for general shape formation. The triangles (dark gray) form a straight line, and the remainder (light gray) is not large enough to form another triangle.

The side length ℓ must be chosen carefully so that the resulting number of triangles Δ in the intermediate structure is sufficient to construct the desired shape S. More specifically, Δ should satisfy $(3/4)s+1 \leq \Delta \leq s-3$, where $s = |S|$ is the number of faces in S; the choice of these particular bounds is explained in [29]. If L is the side length of the initial triangle of all n particles, then ℓ is chosen to be $\ell = \lceil L/\lfloor c \cdot \sqrt{s} \rfloor \rceil$, where $c < 1$ is a constant. It is shown in [29] that there is a c which yields the desired bounds on Δ, assuming s and L are sufficiently large.

To form this intermediate structure, the particle system first performs leader election (Section 3.1) to elect a unique leader particle on the boundary of the initial triangle. The leader then uses a token to transfer its leadership to a particle at one of the triangle's vertices. Next, the particle system determines the value of ℓ. However, since $\ell = \Omega(\sqrt{n})$, a single particle cannot keep ℓ in its constant-size memory. So the leader initiates the following token passing scheme to store ℓ

in a distributed fashion over multiple particles. The leader generates a *counter token* and sends it down one of the initial triangle's sides. The counter token stores the number of steps it has taken modulo $\lfloor c \cdot \sqrt{s} \rfloor$, which is a constant since c and s are. Whenever the token's count is 0 (starting at the leader), a *marker token* is generated that moves back towards the leader without overtaking other marker tokens. When the counter token is consumed by the particle at the end of the triangle's side, exactly ℓ marker tokens will have been generated. Thus, when all marker tokens have moved back towards the leader as far as possible — which can be detected by the leader using a process similar to "settling" in the solitude verification phase of leader election (Section 3.1) — exactly ℓ particles, starting with the leader, will be holding marker tokens.

Using ℓ, the leader coordinates a recursive process to form the intermediate structure. The process first splits the current triangle into a smaller triangle and an isosceles trapezoid with legs of length ℓ (see Fig. 17). The trapezoid immediately becomes part of the intermediate structure, while the smaller triangle must be rotated, shifted twice, and rotated again to be placed in line with the next part of the intermediate structure. This process then recurs on the smaller triangle until a triangle with side length at most ℓ is moved, completing the intermediate structure.

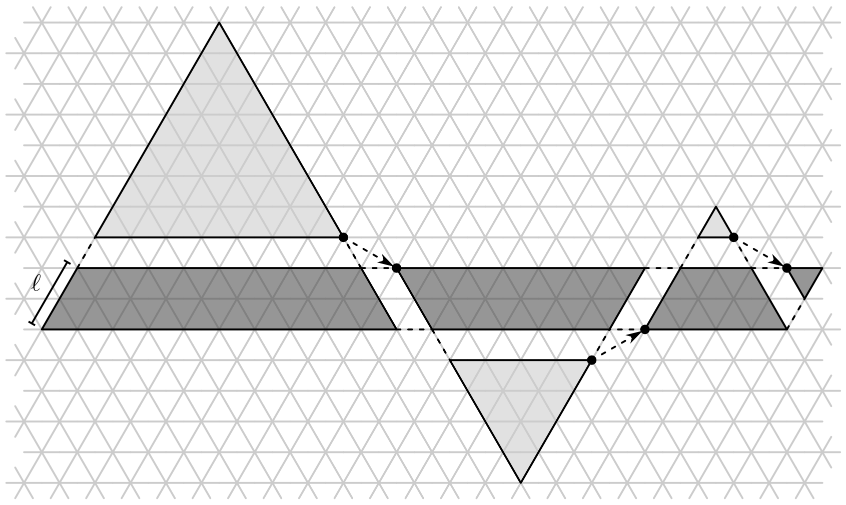


Fig. 17: Constructing the intermediate structure. The parts shown in dark gray form the intermediate structure, while the triangles that are moved are shown in light gray. The dashed arrows show these triangle movements. For example, the leftmost triangle is rotated 60° about the marked vertex, shifted once right and once down-right, and then rotated 120° about the next marked vertex.

Forming the Final Shape. To form the desired shape S, the leader coordinates a process that sequentially adds triangles from the intermediate structure to the outside of the shape under construction. In [29], Gmyr first describes a "simple algorithm" which captures the main ideas of the general shape formation algorithm. However, there are three issues the simple algorithm does not address. First, faces of S have overlapping edges while triangles formed by particles do not, since each node of G_Δ can be occupied by at most one particle. Thus, realizing a shape may require pruning triangles to different side lengths (as in Fig. 18) and reincorporating the pruned particles elsewhere. Second, the remainder of particles in the intermediate structure that did not form a complete triangle must somehow be incorporated into the final shape. Finally, the algorithm for constructing the intermediate structure constructs at most $s - 3$ triangles, but the desired shape S has s faces. Thus, some expanded triangles need to make up for the missing triangles. We will only present the simple algorithm for general shape formation, and refer the interested reader to [29] for the full algorithm.

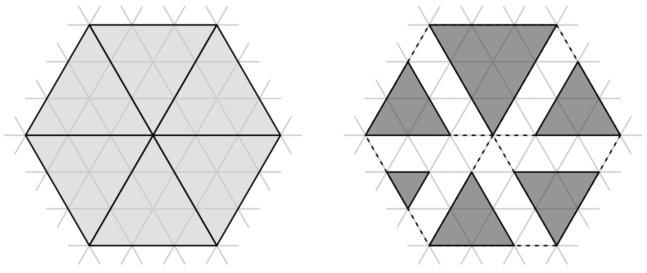


Fig. 18: A shape composed of six faces with overlapping edges (left) and one possible realization requiring triangles of three different side lengths (right).

In the simplified setting, suppose that the particles pruned to create smaller triangles do not need to be reincorporated and that the intermediate structure consists of exactly s triangles without a remainder. The algorithm constructs a scaled representation of the desired shape S, where each face has side length ℓ.

The leader first computes a permutation $(a_1, a_2, \ldots, a_s)$ of the faces of S such that for every $1 \leq i \leq s$, the subset of faces $(a_1, a_2, \ldots a_i)$ is itself a connected shape S_i and the face a_i has a side on the outer boundary of S_i. Such a permutation is guaranteed to exist since S is sequentially constructible, and the leader can compute it since s is a constant.

The triangles of the intermediate structure are then mapped to the faces in the permutation $(a_1, a_2, \ldots, a_s)$. If needed, a triangle can be pruned by one or two rows of particles; the pruned rows are moved out of the way using chain movements. The triangles of the intermediate structure are then added to the outer boundary of the shape in the order defined by the permutation using triangle rotations and shifts. If the intermediate structure or pruned rows obstruct the placement of a triangle, they are simply moved out of the way.

Analysis As described earlier, the intermediate structure with a desired number of triangles is constructed correctly, assuming the number of faces s in the goal shape and the number of particles in the system n are sufficiently large. It is shown in [20,29] that this intermediate construction completes in $\mathcal{O}(\sqrt{n})$ rounds, w.h.p. The w.h.p. qualifier is inherited from leader election (Section 3.1).

A triangle can be moved from the intermediate structure to its goal position in the final shape using only a constant number of triangle rotations and shifts, each of which requires $\mathcal{O}(\ell)$ rounds, where $\ell = \mathcal{O}(\sqrt{n})$ is the side length of the triangle. The number of rows pruned from triangles is constant, and each has length $\mathcal{O}(\ell) = \mathcal{O}(\sqrt{n})$. These pruned rows are moved out of the way a distance $\mathcal{O}(\sqrt{n})$ at most a constant number of times, so moving the pruned rows requires $\mathcal{O}(\sqrt{n})$ rounds in total, by Theorem 4. Therefore, all together, the following runtime bound is obtained.

Theorem 9. *The general shape formation algorithm constructs any sequentially constructible shape in $\mathcal{O}(\sqrt{n})$ rounds, w.h.p.*

We conclude with the following theorem, which shows that the general shape formation algorithm achieves the optimal bound.

Theorem 10. *Any local-control algorithm for forming a non-triangular shape S requires $\Omega(\sqrt{n})$ rounds in the worst case.*

4.3 Object Coating

Object coating is another natural application of programmable matter. For example, one can imagine a particle system coating remote parts of a bridge to identify stress points, or coating a vehicle as a layer of smart paint. Instead of developing a class of coating algorithms that each coats a specific object, a more elegant approach might seek one general algorithm that dynamically adapts to any given object. We present such an algorithm in this section. The *universal coating algorithm* was defined and proven to be correct in [21], and its runtime analysis and proofs of worst-case optimality appeared in [14].

Problem Description We begin with some terminology. *Layer i* of an object O is the set of unoccupied nodes in G_Δ whose shortest path to O has length i. Let B_i denote the number of nodes in layer i. An object O has a *tunnel of width k* if the subgraph of G_Δ induced by the non-object nodes is not k-connected; for example, Fig. 19 depicts an object with a tunnel of width 1.

An instance of the object coating problem has the form $(\mathcal{P}, O)$, where $\mathcal{P}$ is a system of n particles and O is a static object to be coated. An instance is *valid* if: (*i*) all particles in $\mathcal{P}$ are initially contracted and in an idle state, (*ii*) the nodes occupied by particles of $\mathcal{P}$ and the object O induce a connected subgraph of G_Δ, (*iii*) O does not contain holes, and (*iv*) O does not contain any tunnels of width $2(\lceil n/B_1 \rceil + 1)$. Coating an object with narrow tunnels requires technical mechanisms that complicate the protocol without contributing to the main idea of coating, so these types of objects are not considered. An algorithm solves a valid instance $(\mathcal{P}, O)$ of the object coating problem if it terminates in a configuration where all particles of $\mathcal{P}$ are as close to the object O as possible, after which no particle ever moves or changes state. Intuitively, this means that $\mathcal{P}$ coats O as evenly as possible.

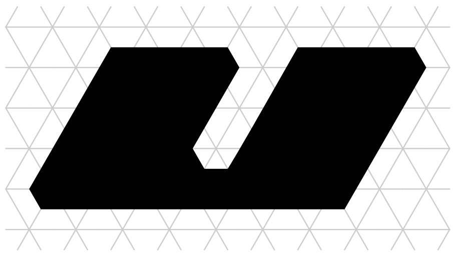


Fig. 19: An object with a tunnel of width 1.

Algorithm The universal coating algorithm is composed of several algorithmic primitives that run concurrently without any underlying synchronization. The first of these is the spanning forest primitive described in Section 3.2, which orients the particles towards the object. The *complaint-based coating primitive* is responsible for coating layer 1 of the object (also called the *surface layer*). The *node-based leader election primitive* is a variant of the leader election algorithm described in Section 3.1 that, instead of directly electing a leader particle, elects a leader node on the surface layer. After the surface layer is completely coated, the particle occupying the leader node becomes the leader particle. This leader particle triggers the *general layering primitive*, which allows each layer $i \geq 2$ to form once layer $i-1$ is complete. We describe each of these primitives in detail after introducing some preliminaries.

Preliminaries. Particles can be in one of five states: idle, follower, root, marker, and retired. Throughout the coating algorithm, a particle P keeps track of its current layer number, denoted P.layer. However, to respect the constant-size memory constraint, P.layer is stored modulo 4 in a particle's memory. A layer is said to be *filled* if all nodes in that layer are occupied by retired particles.

Spanning Forest Primitive. The spanning forest primitive for coating extends the spanning forest primitive in Section 3.2. Instead of assuming the root particles are predetermined, idle particles become roots if they are adjacent to the object or a retired particle. Additionally, if the new root was adjacent to the object, it makes the node it occupies a *leader candidate node* and begins to assist in leader election, described below. As usual, the root is responsible for leading its tree of followers; the path it traverses is defined by the complaint-based coating and general layering primitives, described below. It is possible that a root may be blocked by particles of another tree during its path traversal. Instead of using the root swap operation defined in Section 3.2, blocked roots in the coating algorithm simply wait. An unblocked root is called a *super-root*.

Complaint-Based Coating. The complaint based-coating primitive is responsible for coating the surface layer with particles. When an idle particle becomes a follower according to the spanning forest primitive, it generates a *complaint flag*. Complaint flags are forwarded from particles to their parents through the spanning forest. In more detail, each particle can hold at most two complaint flags. Whenever a particle P is holding at least one complaint flag, it forwards one flag to its parent as long as its parent is holding less than two flags.

These complaint flags eventually accumulate at and behind a super-root of the spanning forest. A super-root can only expand along its traversal path if it is holding a complaint flag and, when it expands, it consumes one complaint flag (see Fig. 20b–20c). No other particles (i.e., roots or followers) need complaint flags to perform their movements. When a root is in a situation where it could perform a handover with either another root on the object or a follower not yet on the object, it gives preference to the follower (see Fig. 20d–20e). These movement rules ensure that a particle not yet in the surface layer will eventually join, provided the surface layer is not already filled.

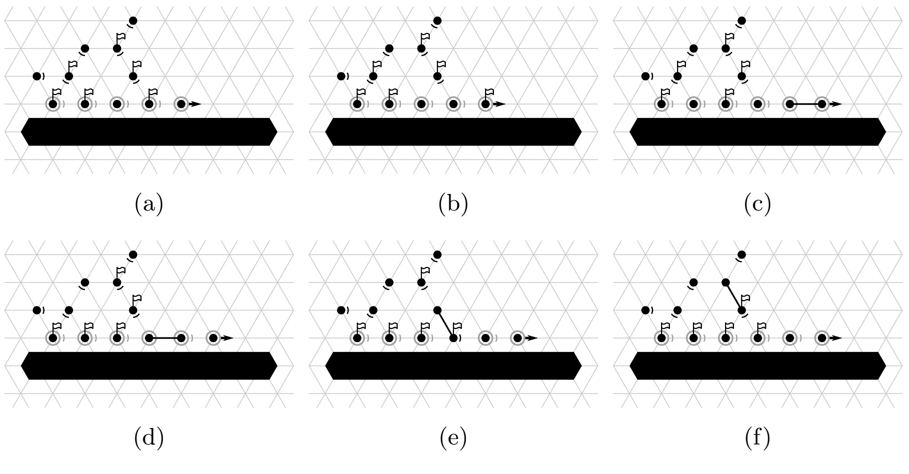

(a) (b) (c) (d) (e) (f)

Fig. 20: Example of the complaint-based coating primitive. The roots (black dots with gray circles) are coating the object (black polygon) behind the super-root (gray circle with black arrow). Complaint flags are forwarded to the super-root to enable movement and allow more followers to join the surface layer.

Node-Based Leader Election. The node-based leader election primitive runs in parallel with the complaint-based coating primitive to elect a leader node on the surface layer. This primitive uses a variant of the algorithm presented in Section 3.1, where the leader candidates are nodes instead of static particles. The roots in the surface layer facilitate the competition between leader node candidates by storing and transferring all the tokens and state information used by the leader election algorithm for the node(s) they occupy. Root movements are handled carefully so that no leader election information is lost or moved to a different node; for example, during a handover between an expanded particle P and a contracted particle Q, P transfers all leader election information about the node occupied by its tail to Q.

The node-based leader election primitive will not successfully elect a leader node until all nodes of the surface layer are occupied. Once a leader node emerges, the first contracted particle to occupy it becomes both retired and a marker. This marker particle designates a neighboring node in layer 2 as a *marker node*, which will act as the starting point for the next layer. If a contracted root is following a retired particle, it also becomes retired, causing the surface layer to fill with retired particles in counterclockwise order. Once the surface layer is completely filled with retired particles, the general layering primitive is activated.

One caveat is necessary before presenting the final primitive. If the surface layer is longer than the number of particles in the system (i.e., $B_1 > n$), a leader node will never be elected. However, the complaint-based coating primitive will still continue until all complaint flags are consumed by the super-root, bringing all particles in the system into the surface layer. This results in a successful coating of the object; all particles are as close to the object as possible.

General Layering. The general layering primitive handles the coating of all layers $i \geq 2$. Following the spanning forest primitive, followers become roots whenever they become adjacent to a retired particle. In the general layering primitive, roots perform a clockwise traversal of their layer if their layer number is odd (see, e.g., Fig. 21a), and a counterclockwise traversal otherwise.

One of three cases eventually occurs for a root P. First, if P encounters an unoccupied node in the layer below it, it moves into the lower layer, causing it to change the direction of its traversal (Fig. 21b–21c). Second, if P is contracted and encounters a retired particle in its layer, it also retires (Fig. 21d). Finally, if P is contracted and occupies the marker node designated by the marker particle in the layer below, it waits until the marker particle signals that layer $P.\text{layer} - 1$ is completely filled (which it can determine locally). Once signalled, P retires and becomes the marker particle for layer $P.\text{layer}$, designating a neighboring node in layer $P.\text{layer} + 1$ as the next marker node. This continues until all particles become retired.

Analysis The correctness of the universal coating algorithm follows from the *safety* condition, i.e., that the set nodes occupied by particles and objects remains connected at all times, and the *liveness* condition, i.e., that the system eventually makes progress. Progress is made if an idle particle becomes active (i.e., a root or follower), a movement is executed, or an active particle retires. Both safety and liveness are proven by Derakshandeh et al. in [21].

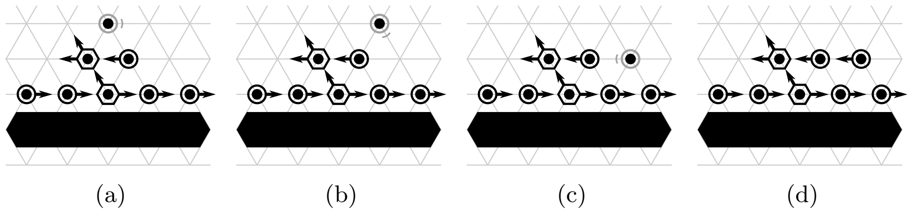


Fig. 21: Example of the general layering primitive. Retired particles are shown in black circles, and marker particles are shown in black hexagons. A root (gray circle) (a) traverses layer 3 in a clockwise direction, (b) encounters an unoccupied location in layer 2, (c) enters layer 2 and changes directions, and (d) retires.

In [14], Daymude et al. use a careful dominance argument to bound the runtime of the universal coating algorithm. To coat the surface layer, the particle system first organizes into a spanning forest. By Theorem 4, this is achieved in $\mathcal{O}(n)$ rounds. The surface layer is then coated in $\mathcal{O}(B_1) = \mathcal{O}(n)$ rounds, where B_1 is the length of the surface layer, assuming there are enough particles to do so. By Theorem 2, a leader node is elected in an additional $\mathcal{O}(B_1)$ rounds, w.h.p, allowing the particles of the surface layer to retire in $\mathcal{O}(B_1)$ more rounds. By a similar argument, coating the higher layers is shown to take another $\mathcal{O}(n)$ rounds in the worst case. All together, we have the following theorem.

Theorem 11. *The universal coating algorithm correctly solves a valid instance of the object coating problem $(\mathcal{P}, O)$ in $\mathcal{O}(n)$ rounds in the worst case, w.h.p, where n is the number of particles in $\mathcal{P}$.*

We conclude with the following theorem, which shows that the universal coating algorithm achieves the optimal bound.

Theorem 12. *The worst-case runtime required by any local-control algorithm to solve the object coating problem for a system of n particles is $\Omega(n)$ rounds.*

5 Stochastic Algorithms Under the Amoebot Model

In the *stochastic approach to self-organizing particle systems*, algorithms under the amoebot model are designed and analyzed using concepts from statistical physics and stochastic processes. Instead of using careful state management and communication to drive particle computation and movement as in the algorithms of Section 4, these stochastic algorithms are stateless, use almost no communication, and depend only on probabilistic decisions. Designing these algorithms begins by defining an energy function that captures the objectives for the particle system. One then designs *Markov chains* that, in the long run, favor particle system configurations with desirable energy values. Although Markov chains are usually defined at a global, system level, these must be designed carefully so that they can be translated into fully distributed, local, asynchronous algorithms run by each particle individually.

The motivation underlying the design of these Markov chains comes from statistical physics, where ensembles of particles similar to those considered in the amoebot model represent physical systems. Previous studies on these systems have shown that local micro-behaviors can induce global, macro-scale changes to a system [3,4,46], yielding the kind of emergent phenomena desirable for programmable matter. Like a spring relaxing, physical systems favor configurations that minimize energy. Each system configuration σ is assigned an energy value by a Hamiltonian $H(\sigma)$ and a corresponding weight $w(\sigma) = e^{-B \cdot H(\sigma)}$, where $B = 1/T$ is inverse temperature. Markov chains have been well-studied as a tool for sampling system configurations with probabilities proportional to their weight $w(\sigma)$, where configurations with the least energy $H(\sigma)$ are the most likely to be sampled.

The stochastic approach to self-organizing particle systems utilizes a Hamiltonian $H(\sigma)$ over particle system configurations σ that assigns the lowest values to desirable configurations; a corresponding Markov chain algorithm is then designed to favor these configurations with small $H(\sigma)$. Each problem uses a different Hamiltonian. Each problem also uses a bias parameter $\lambda = e^B$. The weight of a configuration then becomes $w(\sigma) = \lambda^{-H(\sigma)}$. Thus, raising λ (by increasing B, effectively lowering temperature) increasingly favors configurations with lower energy values, yielding the desired particle system configurations.

Before introducing the stochastic algorithms under the amoebot model, we give a brief primer on the terminology and techniques used in their design.

Terminology. A particle system *configuration* is the set nodes (locations) of G_Δ occupied by particles. An *edge* of a configuration is an edge of G_Δ where both endpoints are occupied by particles. When referring to a *path*, we mean a path of such edges. Two particles are *connected* if there exists a path between them, and a configuration is *connected* if all pairs of particles are[9]. A *hole* in a configuration is a maximal finite connected component of unoccupied locations.

Markov Chains. A *Markov chain* $\mathcal{M}$ is a memoryless stochastic process defined on a state space Ω. Here, Ω is a set of particle system configurations; thus, we only consider state spaces that are finite and discrete. The transition matrix $Q : \Omega \times \Omega \to [0, 1]$ is defined so that $Q(\sigma, \tau)$ is the probability of moving from state σ to state τ in one step, for any pair of states $\sigma, \tau \in \Omega$. In Markov chains for particle systems, transitions correspond to one particle moving one unit in one direction, and the transition probabilities are chosen carefully to achieve some objective. The t-step transition probability $Q^t(\sigma, \tau)$ is the probability of moving from σ to τ in exactly t steps.

A Markov chain is *irreducible*, or its state space is *connected*, if there is a sequence of valid transitions from any state to any other state; i.e., for all

[9] This definition of configuration connectivity is equivalent to that of system connectivity given in Section 2.2.

$\sigma, \tau \in \Omega$, there is a t such that $Q^t(\sigma, \tau) > 0$. A Markov chain is *aperiodic* if for all $\sigma, \tau \in \Omega$ we have $\gcd\{t : Q^t(\sigma, \tau) > 0\} = 1$. A Markov chain is *ergodic* if it is both irreducible and aperiodic. Any finite, ergodic Markov chain converges to a unique *stationary distribution* π defined as $\lim_{t\to\infty} Q^t(\sigma, \tau) = \pi(\tau)$, for all $\sigma, \tau \in \Omega$. Any distribution π' satisfying the *detailed balance condition* — $\pi'(\sigma)Q(\sigma, \tau) = \pi'(\tau)Q(\tau, \sigma)$ for all $\sigma, \tau \in \Omega$ — must be this unique stationary distribution (see, e.g., [27]).

Given a state space Ω, a set of allowable state transitions, and a desired stationary distribution π on Ω, the celebrated Metropolis-Hastings algorithm [32] defines a Markov chain on Ω that uses only allowable transitions and has stationary distribution π. This algorithm sets the probabilities of state transitions as follows. Starting at a state $\sigma \in \Omega$, choose a neighboring state $\tau \in \Omega$ (one with an allowable transition from σ to τ) uniformly with probability $1/(2\Delta)$, where Δ is the maximum number of neighbors of any state. Move from σ to τ with probability $\min\{1, \pi(\tau)/\pi(\sigma)\}$; with the remaining probability stay at σ and repeat. Assuming the allowable transitions connect the state space, then detailed balance will verify that this algorithm yields a Markov chain with π as its stationary distribution. Although calculating $\pi(\tau)/\pi(\sigma)$ seems to require global knowledge, this ratio can often be calculated with only local information when many terms cancel out, as is the case for the algorithms presented here.

Obliviousness and Robustness. Compared to the mostly deterministic algorithms in Section 4, these stochastic algorithms are *nearly oblivious*, requiring very little memory. Markov chains are naturally memoryless, and thus the distributed algorithms derived from them also have very little dependence on memory and communication. As we will see in Sections 5.1–5.3, only 1–2 bits of memory per particle are required in the resulting distributed algorithms. This is an artifact of translating the Markov chain algorithms — where a particle moves from one node to a neighboring node in a single step — into ones that fully respect the constraints of the amoebot model, where a particle can perform at most one expansion or contraction per activation.

Our stochastic algorithms are also significantly more *robust* (i.e., have a higher tolerance to failures) than those in Section 4. A distributed algorithm's *fault tolerance* is its ability to correctly solve a problem in spite of potential failures, a highly desirable and relevant property for programmable matter. One can imagine that in a system of thousands of particles, a small number of them may die and cease to move, compute, or communicate (i.e., a *crash failure*) or become corrupted and move, compute, or communicate erroneously (i.e., a *Byzantine failure*). Even a single fault of either type would cause complete failure in the algorithms in Section 4. In contrast, the stochastic algorithms have robustness built in. Because these algorithms are stateless and do not rely on communication, crashed particles and particles attempting to communicate arbitrary or malicious information have no real effect on the behavior of non-faulty particles. However, crashed particles act as fixed points that will affect the resulting particle system configurations.

5.1 Compression

The original publication on compression was the first to introduce the stochastic approach to self-organizing particle systems [11]. This line of work continued with shortcut bridging (Section 5.2) and separation (Section 5.3), both of which extend the original algorithm design of compression to achieve more complex behaviors. We will thus give more motivation and details for compression, while focusing more on the distinguishing features of the later algorithms.

In the *compression* problem, a particle system gathers together as tightly as possible, as in a disc or its equivalent in the presence of some underlying geometry. There are many metrics that capture this behavior; e.g., system diameter or average particle distance to the system's center of mass. Here, a particle system must reorganize to minimize its *perimeter*, where a configuration's perimeter is measured by the length of the walk along its boundary. Several examples of this behavior exist in nature, particularly in social insects: fire ants gather to form floating rafts [38], cockroach larvae perform self-organizing aggregation [34], and honeybees choose hive locations based on decentralized swarming and recruitment [9]. While no individual insect can view the whole group when making decisions and soliciting information, it can take cues from its immediate neighbors to achieve cooperation.

Problem Description Let $p(\sigma)$ denote the perimeter of a particle system configuration σ. For a system of n particles, the minimum possible perimeter is $p_{min} := p_{min}(n) = \Theta(\sqrt{n})$. A configuration σ with no holes is said to be α-*compressed* if $p(\sigma) < \alpha \cdot p_{min}$, for any $\alpha > 1$. An algorithm solves the compression problem if, given any particle system in an initially connected configuration and any $\alpha > 1$, eventually the system reaches and remains in a set of α-compressed configurations with all but a probability exponentially small in n.

Analogously, the maximum possible perimeter for a system of n particles is $p_{max} := p_{max}(n) = 2n - 2$. A configuration σ with no holes is said to be β-*expanded* if $p(\sigma) > \beta \cdot p_{max}$, for any $0 < \beta < 1$. An algorithm solves the expansion problem if, given any particle system in an initially connected configuration and any $0 < \beta < 1$, eventually the system reaches and remains in a set of β-expanded configurations with all but a probability exponentially small in n.

Markov Chain $\mathcal{M}$ We first present the Markov chain $\mathcal{M}$ for compression and will later show its translation into a distributed, local, asynchronous algorithm $\mathcal{A}$ that can be run by individual particles. $\mathcal{M}$ is defined on the state space Ω of all connected particle system configurations of n contracted particles. Both $\mathcal{M}$ and $\mathcal{A}$ start in an arbitrary configuration $\sigma_0 \in \Omega$ and take a bias parameter $\lambda > 1$ as input, where λ controls the preference for having small perimeter.

Recall that, for each stochastic algorithm, a Hamiltonian $H(\sigma)$ is defined which assigns the lowest energy values — and thus the largest weight $w(\sigma) = e^{-B \cdot H(\sigma)} = \lambda^{-H(\sigma)}$ — to desirable configurations. To achieve compression, the lowest energy values should be assigned to the configurations with the smallest perimeter. In [11], Cannon et al. prove that minimizing configuration perimeter $p(\sigma)$ is equivalent to maximizing the number of configuration edges $e(\sigma)$. Thus, by setting $H(\sigma) = -e(\sigma)$, we obtain $w(\sigma) = \lambda^{e(\sigma)}$. This implies that $\lambda > 1$ corresponds to particles favoring having more neighbors while $\lambda < 1$ corresponds to particles favoring having fewer neighbors.

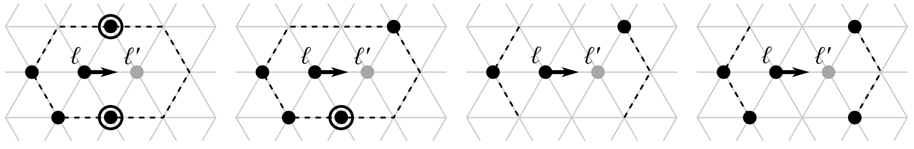
Markov chain $\mathcal{M}$ is carefully designed to maintain several critical properties that are necessary for its correctness and for applying certain tools from Markov chain analysis. First, $\mathcal{M}$ must keep the particle system connected and hole-free throughout its execution, assuming it starts in a connected, hole-free configuration. Next, $\mathcal{M}$ should be ergodic and, after a move is made, there should be a nonzero probability that it is undone in the next step. Finally, in order to solve the compression problem, $\mathcal{M}$ must eventually reach a stationary distribution that favors system configurations proportional to their weight $w(\sigma) = \lambda^{e(\sigma)}$ using only local information and local particle moves.

Local Properties for Simple Connectivity. Two local properties are used to ensure the particle system remains connected and hole-free throughout the execution of $\mathcal{M}$. Together, these properties ensure that a particle's local connectivity with respect to its neighbors does not change as a result of its move. Moreover, they ensure that every move can be undone, as desired.

We use the following notation. For a location ℓ of G_Δ, let $N(\ell)$ denote the set of particles adjacent to ℓ. For adjacent locations ℓ and ℓ', we use $N(\ell \cup \ell')$ to denote the set $N(\ell) \cup N(\ell')$, excluding particles occupying ℓ or ℓ'. Let $\mathbb{S} = N(\ell) \cap N(\ell')$ be the particles adjacent to both locations; note that $|\mathbb{S}| \in \{0, 1, 2\}$. The following properties can be locally checked by an expanded particle occupying both ℓ and ℓ', and are symmetric with respect to these locations (see Fig. 22).

Property 1. $|\mathbb{S}| \in \{1, 2\}$ and every particle in $N(\ell \cup \ell')$ is connected to a particle in $\mathbb{S}$ by a path through $N(\ell \cup \ell')$.

Property 2. $|\mathbb{S}| = 0$, ℓ and ℓ' each have at least one neighbor, all particles in $N(\ell) \setminus \{\ell'\}$ are connected by paths within this set, and all particles in $N(\ell') \setminus \{\ell\}$ are connected by paths within this set.



(a) Prop. 1: ✓ (b) Prop. 1: X (c) Prop. 2: ✓ (d) Prop. 2: X

Fig. 22: Examples of particle neighborhoods with respect to Properties 1 and 2. Particles of $\mathbb{S}$ are drawn with black circles around them.

We can now present the Markov chain $\mathcal{M}$ for compression (Algorithm 1).

Algorithm 1 Markov Chain $\mathcal{M}$ for Compression

```
    Beginning at any connected configuration $\sigma_0$ of $n$ contracted particles, repeat:
 1: Select particle $P$ uniformly at random from among all particles; let $\ell$ be its location.
 2: Choose a neighboring location $\ell'$ and $q \in (0,1)$ uniformly at random.
 3: if $\ell'$ is unoccupied then
 4:     $P$ expands to occupy both $\ell$ and $\ell'$.
 5:     Let $e = |N(\ell)|$ be the number of neighbors $P$ had when it was contracted at $\ell$,
        and let $e' = |N(\ell')|$ be the number of neighbors $P$ would have if it contracts to $\ell'$.
 6:     if (i) $\ell$ and $\ell'$ satisfy Property 1 or 2, (ii) $e < 5$, and (iii) $q < \lambda^{e'-e}$ then
 7:         $P$ contracts to $\ell'$.
 8:     else $P$ contracts back to $\ell$.
```

Distributed, Local, Asynchronous Algorithm $\mathcal{A}$ We now present the local, distributed, asynchronous algorithm $\mathcal{A}$ that each particle runs. Recall from Section 2.1 that during a single activation of a particle P, P can perform an arbitrary amount of computation and at most one expansion or contraction. In particular, P cannot do both an expansion and a contraction in one activation as $\mathcal{M}$ does in a single step. Thus, $\mathcal{A}$ must decouple a single step of $\mathcal{M}$ into two (not necessarily consecutive) particle activations and carefully handle the way in which the particle's neighborhood may change between its two activations (see [11] for full details of this decoupling).

Algorithm 2 Distributed Algorithm $\mathcal{A}$ for Compression run by Particle P

```
    If $P$ is contracted:
 1: Let $\ell$ denote $P$'s current location.
 2: Particle $P$ chooses neighboring location $\ell'$ uniformly at random from the six pos-
    sible choices, and generates a random number $q \in (0,1)$.
 3: if $\ell'$ is unoccupied and $P$ has no expanded neighboring particle at $\ell$ then
 4:     $P$ expands to occupy both $\ell$ and $\ell'$.
 5:     if there are no expanded particles adjacent to $\ell$ or $\ell'$ then
 6:         $P$ sets a flag $f = $ TRUE in its local memory.
 7:     else $P$ sets $f = $ FALSE.

    If $P$ is expanded:
 8: Let $N^*(\cdot) \subseteq N(\cdot)$ be the set of neighboring particles excluding any heads of ex-
    panded particles.
 9: Let $e = |N^*(\ell)|$ be the number of neighbors $P$ had when it was contracted at $\ell$,
    and let $e' = |N^*(\ell')|$ be the number of neighbors $P$ would have if it contracts to $\ell'$.
10: if (i) $\ell$ and $\ell'$ satisfy Property 1 or 2 with respect to $N^*(\cdot)$, (ii) $e < 5$, (iii)
    $q < \lambda^{e'-e}$, and (iv) $f = $ TRUE then
11:     $P$ contracts to $\ell'$.
12: else $P$ contracts back to $\ell$.
```

Each particle P continuously runs Algorithm $\mathcal{A}$, executing Steps 1–7 if P is contracted and Steps 8–12 if P is expanded. Conditions (i)–(iii) in Step 10 of $\mathcal{A}$ are the same as those in Step 6 of $\mathcal{M}$. The additional Condition (iv) ensures P is the only particle in its neighborhood potentially moving to a new position since it last expanded. Any conflicts arising from two particles concurrently attempting to expand into the same location are assumed to be resolved arbitrarily (Section 2.1). Hence, any concurrent movements will cover pairwise disjoint neighborhoods and the respective actions will be mutually independent.

However, in an asynchronous setting, one cannot typically assume the next particle to be activated is equally likely to be any particle, as in Step 1 of $\mathcal{M}$. This assumption is made in order to explicitly calculate the stationary distribution of $\mathcal{M}$ (Lemma 1) and rigorously analyze it (Theorems 13 and 14), but the system's behavior is not expected to differ substantially if this requirement was relaxed.

These random sequences of particle activations can be approximated using Poisson clocks with mean 1. That is, each particle can activate and execute Algorithm $\mathcal{A}$ at a random real time drawn from the exponential distribution e^{-t}. After each action, a particle could then compute another random time drawn from the same distribution e^{-t} and activate again after that amount of time has elapsed. The exponential distribution is unique in that, if particle P has just activated, it is equally likely that any particle will be the next particle to activate, including particle P (see, e.g., [27]). Moreover, a particle updates without requiring knowledge of any other particle's clock. As an aside, the analysis can be modified to accommodate each clock having its own constant mean; however, for ease of presentation, we assume here that they are all i.i.d.

Results Cannon et al. give a detailed analysis of Markov chain $\mathcal{M}$ in [11]; here, we briefly present the major results on some properties of $\mathcal{M}$, its stationary distribution π, and its correctness in solving compression. We conclude with results on using $\mathcal{M}$ to also solve expansion. Note that all results for $\mathcal{M}$ extend to the local algorithm $\mathcal{A}$.

Invariants of $\mathcal{M}$. Cannon et al. prove that if the particle system is initially connected, $\mathcal{M}$ maintains system connectivity. Moreover, once a connected configuration with no holes is reached, $\mathcal{M}$ will never introduce new holes to the system. Together, these imply that once $\mathcal{M}$ reaches the subspace $\Omega^* \subset \Omega$ of all connected, hole-free configurations of n particles, it will remain in Ω^* forever. Since $\mathcal{M}$ is finite and ergodic on Ω^* (a result shown in [11]), it converges to a unique stationary distribution π on Ω^* that can be verified by detailed balance.

Lemma 1. *Markov chain $\mathcal{M}$ for compression has a unique stationary distribution π given by:*

$$\pi(\sigma) = \begin{cases} \lambda^{e(\sigma)}/Z & \text{if } \sigma \in \Omega^*; \\ 0 & \text{otherwise.} \end{cases}$$

where $Z = \sum_{\tau \in \Omega^*} \lambda^{e(\tau)}$ is the normalizing constant or partition function.

Achieving Compression and Expansion. Markov chain $\mathcal{M}$ — and, by extension, local algorithm $\mathcal{A}$ — solve both the compression and expansion problems depending on the value of bias parameter λ. Although compression or expansion could occur before $\mathcal{M}$ even converges to its stationary distribution π, the proofs in [11] rely on analyzing π. First, it is shown that for any $\alpha > 1$ and provided λ and n are large enough, a configuration chosen at random according to π is α-compressed with all but a probability that is exponentially small in n.

Theorem 13. *For any $\alpha > 1$, let $\lambda^* = (2+\sqrt{2})^{\alpha/(\alpha-1)}$. There exists $n^* \geq 0$ and $\zeta < 1$ such that for all $\lambda > \lambda^*$ and $n > n^*$, the probability that a random sample σ drawn according to the stationary distribution π of $\mathcal{M}$ is not α-compressed is exponentially small:*

$$\Pr_{\sigma\sim\pi}\left[p(\sigma) \geq \alpha \cdot p_{min}\right] < \zeta^{\sqrt{n}}.$$

It can also be shown that there is some constant α for which α-compression occurs when $\lambda > 2+\sqrt{2}$ is fixed. However, there is a tradeoff: smaller values of λ require larger values of α and vice versa.

The algorithm also provably achieves expansion for different values of the bias parameter λ. It is shown that, for all $0 < \lambda < 2.17$ and provided n is large enough, there is a constant $0 < \beta < 1$ such that a configuration chosen at random according to the stationary distribution of $\mathcal{M}$ is β-expanded with all but exponentially small probability in n. This is counter-intuitive, since it implies that $\lambda > 1$ (i.e., favoring more neighbors) is not sufficient to guarantee particle compression.

Theorem 14. *For any $0 < \beta < 1$, let $\lambda^* = (\sqrt{2}/(2+\sqrt{2})^{\beta})^{1/(1-\beta)}$. There exists $n^* \geq 1$ and $\zeta < 1$ such that for all $\lambda < \lambda^*$ and $n \geq n^*$, the probability that a random sample σ drawn according to the stationary distribution π of $\mathcal{M}$ is not β-expanded is exponentially small:*

$$\Pr_{\sigma\sim\pi}\left[p(\sigma) \leq \beta \cdot p_{max}\right] < \zeta^{\sqrt{n}}.$$

Similar to compression, it is also shown that there is a constant β for which β-expansion occurs when $0 < \lambda < 2.17$ is fixed. Again, there is a tradeoff in larger values of λ requiring smaller values of β and vice versa.

Convergence Time of $\mathcal{M}$. Although compression provably occurs with all but exponentially small probability once $\mathcal{M}$ converges to its stationary distribution, no explicit bounds are given on the time required for this to occur. However, simulations support the conjecture that the worst case number of steps of $\mathcal{M}$ needed to observe compression is $\Omega(n^3)$ and $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ rounds of $\mathcal{A}$.

Simulations In practice, Markov chain $\mathcal{M}$ yields good compression. Fig. 23 depicts a simulation of $\mathcal{M}$ for $\lambda = 4$ on 100 particles that begin in a line and become compressed. In contrast, $\lambda = 2$ (which still favors having more particle neighbors), does not yield compression; see Fig. 24, where even after 20 million steps of $\mathcal{M}$, the particles have not compressed. Cannon et al. conjecture there is a phase transition in λ, i.e., a critical value λ_c such that for all $\lambda > \lambda_c$ the particles compress and for all $\lambda < \lambda_c$ they do not. Such phase transitions exist for similar statistical physics models (e.g., [8]). Theorems 13 and 14 indicate that if λ_c exists, then $2.17 \leq \lambda_c \leq 2 + \sqrt{2}$.

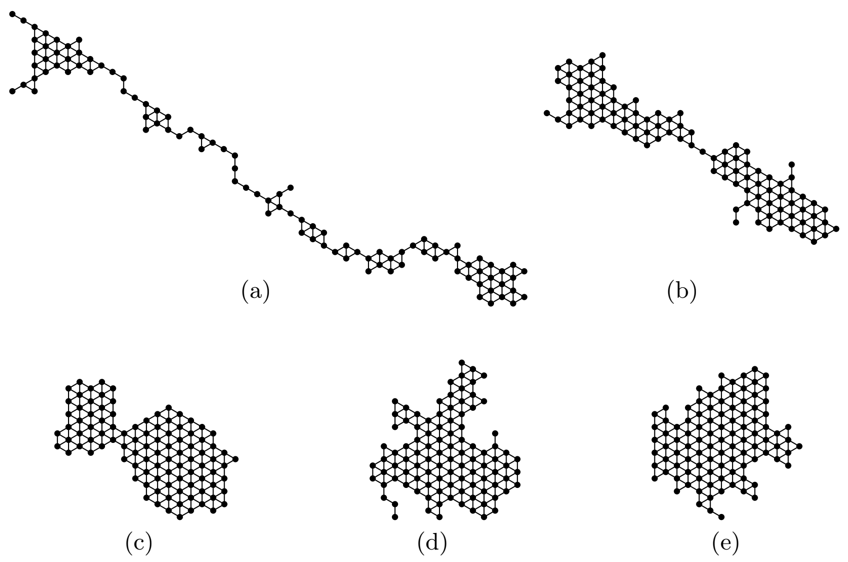


Fig. 23: 100 particles in a line with occupied edges drawn, after (a) 1 million, (b) 2 million, (c) 3 million, (d) 4 million, and (e) 5 million steps of $\mathcal{M}$ with bias $\lambda = 4$.

5.2 Shortcut Bridging

Andrés Arroyo et al. further validated the stochastic approach to self-organizing particle systems in an investigation of *shortcut bridging* [1]. This work is inspired by an entomological study [43] that found army ants of the genus *Eciton* continuously modify the shape and position of foraging bridges — constructed and maintained by their own bodies — across holes and uneven surfaces on the forest floor. These bridges appear to stabilize in a structural formation that balances the “benefit of increased foraging trail efficiency” with the “cost of removing workers from the foraging pool to form the structure” [43]. Shortcut bridging is an attractive goal for programmable matter, which may need to make similar tradeoffs when maintaining bridges over terrain with structural irregularities.

To consider this problem in the amoebot model, two model extensions (Section 2.3) are employed. First, static objects are used to anchor the particle system to certain fixed sites. Second, the locations of G_Δ are considered either

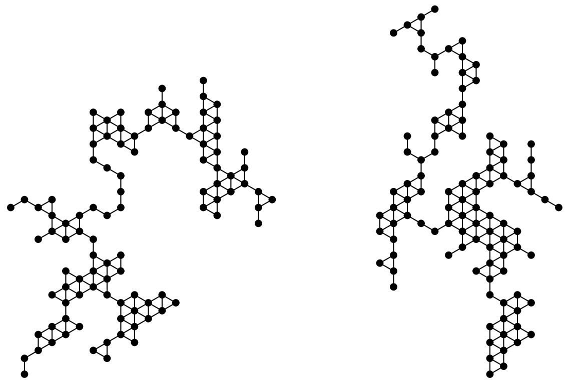


Fig. 24: 100 particles in a line with occupied edges drawn, after (a) 10 million and (b) 20 million steps of $\mathcal{M}$ with bias $\lambda = 2$.

gap (unsupported) or *land* (supported) using node differentiation. The notion of configuration perimeter used in compression (Section 5.1) is extended to address this new land/gap setting as follows. The *weighted perimeter* $\overline{p}(\sigma, c)$ of a particle system configuration σ is the summed weight of the edges on the boundary of σ, where edges between land locations have weight 1, edges between gap locations have weight $c \geq 1$, and edges with one endpoint on land and one endpoint in the gap have weight $(1 + c)/2$.

Problem Description An instance of the shortcut bridging problem has the form $(L, O, \sigma_0, c, \alpha)$, where $L \subseteq V$ is the set of land locations, O is the set of (two) objects to bridge between, σ_0 is the initial configuration for the particle system, $c \geq 1$ is a fixed weight for edges between gap locations, and $\alpha > 1$ is a parameter capturing error tolerance. An instance is *valid* if (*i*) the objects of O and particles of σ_0 all occupy locations in L, (*ii*) σ_0 connects the objects, and (*iii*) σ_0 is connected. Let $\overline{p}_{min} := \overline{p}_{min}(L, O, \sigma_0, c)$ be the minimum possible weighted perimeter of a configuration. An algorithm solves a valid instance if, beginning from σ_0, with all but exponentially small probability it reaches and remains in a set of configurations Σ^* such that any $\sigma \in \Sigma^*$ has $\overline{p}(\sigma, c) < \alpha \cdot \overline{p}_{min}$.

Algorithm As an extension of compression, the Markov chain and resulting distributed algorithm for shortcut bridging share many characteristics with the Markov chain $\mathcal{M}$ (Algorithm 1) and algorithm $\mathcal{A}$ (Algorithm 2) for compression. For brevity, we focus on the main differences.

To solve the shortcut bridging problem, an algorithm must yield configurations that have small weighted perimeter. In essence, an algorithm must balance the competing objectives of having a short path between the two objects while not forming too large of a bridge. These competing objectives are captured by preferring configurations σ that have both small perimeter $p(\sigma)$, the length of the walk around the boundary of σ, and small *gap perimeter* $g(\sigma)$, the number of perimeter edges in the gap. Using the weights defined above, weighted perimeter becomes $\overline{p}(\sigma, c) = p(\sigma) + (c-1)g(\sigma)$; thus, minimizing weighted perimeter is equivalent to simultaneously minimizing both perimeter and gap perimeter.

Two bias parameters are used: λ and γ. The desired Markov chain for shortcut bridging should sample configurations σ proportional to weight $w(\sigma) = \lambda^{-p(\sigma)}\gamma^{-g(\sigma)}$. Setting $\lambda > 1$ corresponds to favoring having small perimeter, as it did for compression, while $\gamma > 1$ corresponds to favoring having small gap perimeter. Arithmetic shows $\lambda^{-\overline{p}(\sigma,c)} = \lambda^{-p(\sigma)-(c-1)g(\sigma)} = \lambda^{-p(\sigma)}(\lambda^{c-1})^{-g(\sigma)}$, so setting $\gamma = \lambda^{c-1}$ results in the desired relationship between perimeter, gap perimeter, and weighted perimeter.

The same local properties from compression (Properties 1 and 2) are used in shortcut bridging to ensure the particle system remains connected, no new holes form, and every move made can be undone. Only one small change is made: the definitions of location neighborhoods (e.g., $N(\ell)$, $N(\ell \cup \ell')$, etc.) are extended to include objects, ensuring that the system does not break away from the points it is supposed to bridge between.

We can now present the Markov chain $\mathcal{M}_B$ for shortcut bridging. It follows the same procedure as Markov chain $\mathcal{M}$ for compression (Algorithm 1), with two differences. First, instead of counting neighbors as in Step 5 of $\mathcal{M}$, $\mathcal{M}_B$ simply refers to the configuration with particle P at its original location ℓ as σ and the configuration with P at its new location ℓ' as σ'. Then, $\mathcal{M}_B$ replaces the probability condition in Step 6 of $\mathcal{M}$ with $q < \lambda^{p(\sigma)-p(\sigma')}\gamma^{g(\sigma)-g(\sigma')}$. One might observe that this probability is not defined locally; however, an argument in [1] shows that it can be calculated by an expanded particle occupying ℓ and ℓ' using only local information from its neighborhood.

Markov chain $\mathcal{M}_B$ can be directly translated to a fully distributed, local, asynchronous algorithm $\mathcal{A}_B$ for shortcut bridging using the same decoupling and Poisson clock mechanisms described for compression.

Results In [1], Andrés Arroyo et al. give a detailed analysis of the Markov chain $\mathcal{M}_B$ and resulting distributed algorithm $\mathcal{A}_B$ for shortcut bridging. Here, we will only present the highlights. As for compression, $\mathcal{M}_B$ is shown to maintain system connectivity. It is also shown to eventually reach the set of configurations with no holes, after which all configurations will remain hole-free.

The resulting stationary distribution π_B over Ω_B (the set of all configurations reachable from σ_0, the initial configuration, via valid transitions of $\mathcal{M}_B$) is shown to be $\pi_B(\sigma) = \lambda^{-p(\sigma)}\gamma^{-g(\sigma)}/Z$, where $Z = \sum_{\tau \in \Omega_B} \lambda^{-p(\tau)}\gamma^{-g(\tau)}$ is the normalizing constant. Analyzing this stationary distribution with a careful Peierls argument results in the following theorem, which shows $\mathcal{M}_B$ correctly solves the shortcut bridging problem.

Theorem 15. *Consider any $\alpha > 1$ and let $\lambda^* = (2+\sqrt{2})^{\alpha/(\alpha-1)}$. There exists $n^* > 0$ such that for all $\lambda > \lambda^*$ and $n > n^*$, if $\gamma = \lambda^{c-1}$, the probability that a random sample σ drawn from the stationary distribution of $\mathcal{M}_B$ has weighted* *perimeter $\overline{p}(\sigma, c) \geq \alpha \cdot \overline{p}_{min}$ is exponentially small in n, where n is the number of particles in the system.*

Simulation results supporting the findings of Theorem 15 can be seen in Figs. 25 and 26, where $\mathcal{M}_B$ was run with biases $\lambda = 4$ and $\gamma = 2$ (i.e., $c = 3/2$) on a V-shaped and N-shaped land mass, respectively.

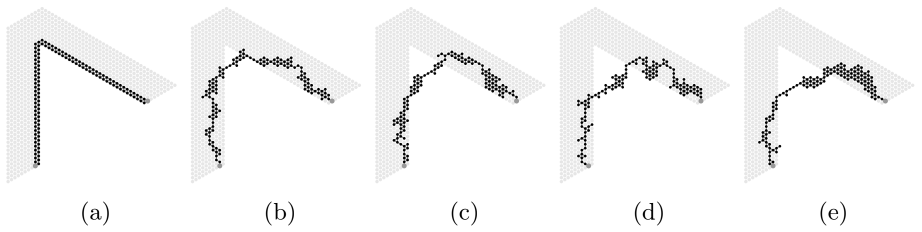


Fig. 25: A particle system, beginning in configuration (a), using biases $\lambda = 4$ and $\gamma = 2$ to shortcut a V-shaped land mass after (b) 2 million, (c) 4 million, (d) 6 million, and (e) 8 million steps of Markov chain $\mathcal{M}_B$. The two objects (large, dark gray) anchor the particle system (black) to land (gray).

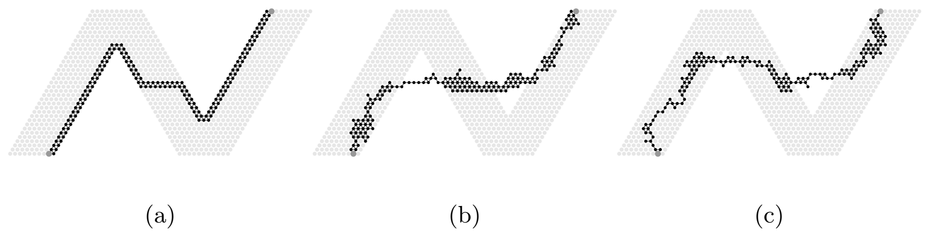


Fig. 26: A particle system, beginning in configuration (a), using biases $\lambda = 4$ and $\gamma = 2$ to shortcut an N-shaped land mass after (b) 10 million and (c) 20 million steps of Markov chain $\mathcal{M}_B$.

Dependence on Gap Angle. The shortcut bridging algorithm is also shown in [1] to have a dependence on the internal angle θ of the gap similar to that of the army ant bridges studied in [43]. Informally, it is shown that when θ is sufficiently small, with all but exponentially small probability the bridge constructed by the particles stays close to the bottom of the gap (away from the apex of angle θ). For some large values of θ, it is shown that the bridge constructed by the particles stays close to the top of the gap (nearer to land) with all but exponentially small probability. Although the bounds obtained in [1] are relatively narrow (e.g., the proofs for small θ only hold for gap angles less than $\theta_1 = 0.0879 \sim 5.03°$), simulations suggest these bounds are far from tight. Fig. 27 depicts simulation results that are consistent with the proven angle dependence behaviors, but were obtained using angles and bias parameter values outside the proven bounds.

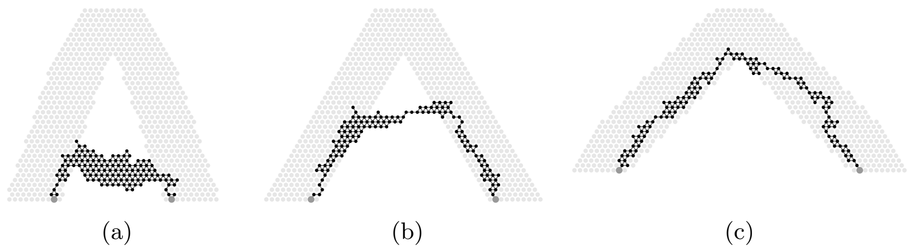


Fig. 27: A particle system using biases $\lambda = 4$ and $\gamma = 2$ to shortcut a V-shaped land mass with gap angle (a) $\pi/6$, (b) $\pi/3$, and (c) $\pi/2$ after 20 million steps of Markov chain $\mathcal{M}_B$.

5.3 Separation

Examples of heterogeneous entities separating and integrating exist at many scales, from molecules exhibiting attractive and repulsive forces, to mixed solutions of varying viscosities, to inherent human biases that influence how we form and maintain social groups. This fundamental behavior of heterogeneous entities separating or integrating in response to environmental stimuli spans remarkably diverse areas of study. Of particular relevance to the stochastic approach to self-organizing particle systems are the Schelling model [49,50] — which explores how micro-motives of individuals can induce macro-phenomena such as racial segregation in residential neighborhoods — and the Ising model of ferromagnetism from statistical physics [55].

In the *separation* problem, a *heterogeneous particle system* — i.e., one in which particles have different immutable *colors* — must self-organize to form monochromatic clusters, resulting in observable color class separation. Each particle P keeps a color $c(P) \in \{c_1, \ldots, c_k\}$ in its memory that is visible to itself and its neighbors, where $k < n$ is some small constant. An edge between two neighboring particles P and Q is *homogeneous* if $c(P) = c(Q)$ and is *heterogeneous* otherwise. Cannon et al. give a Markov chain algorithm $\mathcal{M}_S$ for separation in a two-color particle system in [10], though the algorithm has also been shown to generalize to k-color particle systems in simulations (for a constant k). We will focus on the two-color case for the problem statement, results, and simulations, but will describe the Markov chain $\mathcal{M}_S$ and corresponding distributed algorithm $\mathcal{A}_S$ for separation in full k-color generality.

Problem Description Informally, a two-color particle system configuration is *separated* if there is a set R of particles such that R mostly contains particles of color c_1, its complement $\overline{R}$ mostly contains particles of color c_2, and the boundary between R and $\overline{R}$ is small. If this is the case, R and $\overline{R}$ are called *clusters*. More formally, a configuration is (β, δ)-*clustered*, for $\beta > 0$ and $\delta < 1/2$, if there are at most $\delta|R|$ particles of color c_2 in R, at most $\delta|\overline{R}|$ particles of color c_1 in $\overline{R}$, and the boundary between R and $\overline{R}$ is of size at most $\beta\sqrt{n}$, where n is the number of particles in the system.

An instance of the separation problem has the form $(\sigma_0, \beta, \delta)$ where σ_0 is a connected initial configuration of colored particles and $\beta > 0$ and $0 < \delta < 1/2$ are constants. An algorithm solves an instance if, beginning from configuration σ_0, with all but exponentially small probability it reaches and remains in a set of configurations that are (β, δ)-clustered.

Algorithm As was the case for shortcut bridging (Section 5.2), the Markov chain $\mathcal{M}_S$ and corresponding distributed algorithm $\mathcal{A}_S$ for separation are also extensions of the compression algorithm (Section 5.1) and follow the stochastic approach to self-organizing particle systems. To achieve separation, an algorithm should favor configurations with many edges (inducing small perimeter, as in Section 5.1) and large monochromatic clusters. These objectives are achieved by sampling configurations σ proportional to their weight $w(\sigma) = \lambda^{e(\sigma)}\kappa^{a(\sigma)}$ where $e(\sigma)$ is the number of edges and $a(\sigma)$ is the number of homogeneous edges in σ. Bias parameter λ controls the system's preference for having small perimeter, as in compression and shortcut bridging; larger values of λ increasingly favor compressed configurations while for small λ the opposite is true. The additional bias parameter κ controls separation, favoring clustered/separated configurations when κ is large and well-integrated configurations when κ is small.

A new *swap move* is introduced to enable adjacent particles of different colors to switch places. For two neighboring contracted particles P and Q, either P or Q can initiate a swap to exchange colors, which can be implemented as follows: P reads $x \leftarrow c(Q)$ from the memory of Q, overwrites $c(Q) \leftarrow c(P)$ in the memory of Q, and finally updates its own color $c(P) \leftarrow x$. Adding this swap move enables faster convergence of the separation algorithm in practice, but is not necessary for any of its formal results.

In addition to the usual location neighborhood definitions (e.g., $N(\ell)$, $N(\ell \cup \ell'))$ used in compression and shortcut bridging, separation also uses color-specific location neighborhoods. More precisely, for a location ℓ, $N_i(\ell)$ denotes the set of particles of color c_i adjacent to location ℓ. For neighboring locations ℓ and ℓ', $N_i(\ell \cup \ell')$ denotes the set $N_i(\ell) \cup N_i(\ell')$, excluding particles occupying ℓ and ℓ'. These color-specific neighborhoods are used when calculating the difference in the number of homogeneous edges between a particle's new and old position.

Separation also uses local Properties 1 and 2 from compression to ensure the particle system remains connected and no new holes form. However, these properties need not be verified for swap moves, which do not change the set of occupied nodes and thus cannot disconnect the system or create a hole. We can now present the Markov chain $\mathcal{M}_S$ for separation (Algorithm 3).

Algorithm 3 Markov Chain $\mathcal{M}_S$ for Separation

```
    Beginning at any connected configuration $\sigma_0$ of $n$ contracted particles, repeat:
 1: Select particle $P$ uniformly at random from among all particles; let $c_i$ be its color
    and $\ell$ its location.
 2: Choose a neighboring location $\ell'$ and $q \in (0,1)$ uniformly at random.
 3: if $\ell'$ is unoccupied then
 4:     $P$ expands to occupy both $\ell$ and $\ell'$.
 5:     if (i) $\ell$ and $\ell'$ satisfy Property 1 or 2, (ii) $|N(\ell)| < 5$, and (iii) $q < \lambda^{|N(\ell')|-|N(\ell)|}\kappa^{|N_i(\ell')|-|N_i(\ell)|}$ then
 6:         $P$ contracts to $\ell'$.
 7:     else $P$ contracts back to $\ell$.
 8: else if $\ell'$ is occupied by particle $Q$ of color $c_j$ then
 9:     $P$ calculates $|N_i(\ell)|$ and $|N_j(\ell) \setminus \{Q\}|$ and sends these values to $Q$.
10:     $Q$ calculates $|N_i(\ell') \setminus \{P\}|$ and $|N_j(\ell')|$.
11:     if $q < \gamma^{|N_i(\ell')\setminus\{P\}|-|N_i(\ell)|+|N_j(\ell)\setminus\{Q\}|-|N_j(\ell')|}$ then $Q$ swaps with $P$.
```

While $\mathcal{M}_S$ has much in common with the Markov chain $\mathcal{M}$ for compression (Algorithm 1), it also has important differences. In particular, condition (*iii*) in Step 5 of $\mathcal{M}_S$ specifically addresses homogeneous edges and Steps 8–11 of $\mathcal{M}_S$ implement the new swap move. Moreover, the translation of $\mathcal{M}_S$ to a fully distributed, local, asynchronous algorithm $\mathcal{A}_S$ for separation does not follow trivially from the translation given for compression. Although decoupling a particle's expansion and contraction in Steps 3–7 of $\mathcal{M}_S$ can be done in a similar manner to that of compression, locally synchronizing the swap move of Steps 8–11 of $\mathcal{M}_S$ requires additional mechanisms. Details can be found in [10].

Results Cannon et al. rigorously analyze the Markov chain $\mathcal{M}_S$ and corresponding distributed algorithm $\mathcal{A}_S$ for k-color separation in [10]. As for compression and shortcut bridging, $\mathcal{M}_S$ is shown to maintain system connectivity and remain hole-free throughout its execution, assuming it begins at a connected and hole-free configuration σ_0. Additionally, its stationary distribution π_S over Ω_S (the set of all connected, hole-free configurations with the same number of particles of each color as σ_0) is shown to be $\pi_S(\sigma) = \lambda^{e(\sigma)}\kappa^{a(\sigma)}/Z$, where $Z = \sum_{\tau\in\Omega_S} \lambda^{e(\tau)}\kappa^{a(\tau)}$ is the normalizing constant.

While these initial results follow from standard techniques, proving that $\mathcal{M}_S$ achieves separation requires significantly heavier machinery. Using a Markov chain analysis technique known as *bridging* [37] (not to be confused with the shortcut bridging behavior described in Section 5.2), it is shown that, among two-color configurations with the same small external boundary, a configuration sampled according to the stationary distribution π_S of $\mathcal{M}_S$ will be clustered/separated as desired with all but exponentially small probability. We present the corresponding theorem in its full formality below, but refer the reader to [10] for a more detailed explanation.

Theorem 16. *For any $\alpha > 1$, $\beta > 4\alpha$, and $\delta < 1/2$, there exists κ^* and n_0 (which depend on α, β, and δ) such that for all $\kappa > \kappa^*$ and $n > n_0$, for any α-compressed boundary $\mathcal{B}$, the probability that a configuration sampled according to π_S from among two-color configurations with n particles and boundary $\mathcal{B}$ is not (β, δ)-clustered is at most $\zeta^{\sqrt{n}}$ for some constant $\zeta < 1$.*

Simulations In simulation, $\mathcal{M}_S$ exhibits the expected separation behavior for large λ and κ, as well as integration behaviors for other parameter values. Fig. 28 shows a simulation of $\mathcal{M}_S$ on a two-color system with 50 particles of each color using biases $\lambda = 4$ and $\kappa = 4$, the regime in which the system should compress and individual color classes should separate. Much of the progress towards a compressed and separated system occurs in the first million steps, though the simulation runs for much longer. Fig. 29 compares the resulting configurations after running $\mathcal{M}_S$ from the same initial configuration for the same number of steps, varying only the values of λ and κ. Four distinct phases appear: expanded-integrated, expanded-separated, compressed-integrated, and compressed-separated. Thus, although Theorem 16 only proves separation for two-color systems with small, fixed boundaries, $\mathcal{M}_S$ appears to be capable of a diverse set of dynamic behaviors in practice.

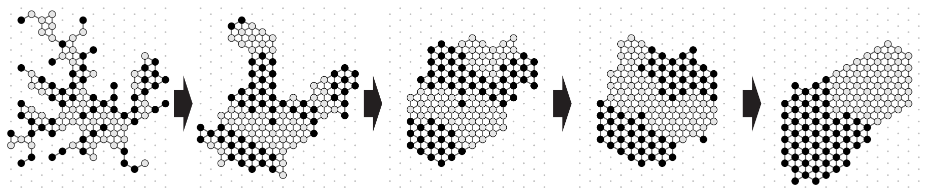


Fig. 28: A two-color heterogeneous particle system starting in an arbitrary state after — from left to right — 0; 50,000; 1,050,000; 17,050,000; and 68,250,000 steps of $\mathcal{M}_S$ with $\lambda = 4$ and $\kappa = 4$.

6 Hybrid Programmable Matter

In this section, we discuss a variant of the amoebot model known as *hybrid programmable matter*, in which a collection of *active robots* operate on a connected system of *passive tiles*. The robots have similar capabilities to the particles of the amoebot model, but the tiles — which are uniform and stateless — cannot move themselves nor perform any computation. To change its (relative) position, a tile must be lifted and moved by a robot. Unlike in the amoebot model, system connectivity is defined with respect to the structure of tiles (including tiles

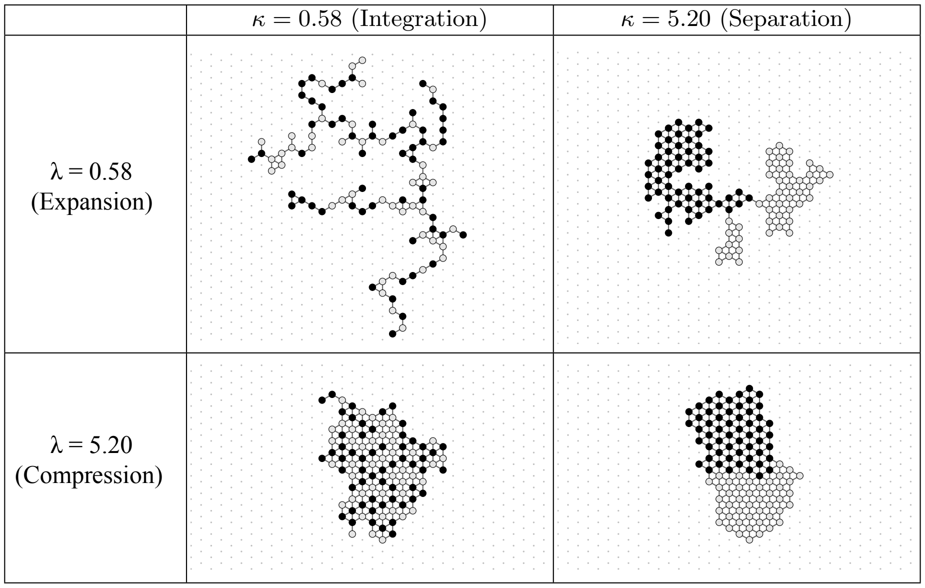


Fig. 29: A two-color heterogeneous particle system starting in the leftmost configuration of Fig. 28 after 50 million steps of $\mathcal{M}_S$ for various values of the parameters λ and κ.

being carried by robots) and must be maintained at all times. Since the set of robots does not need to stay connected, we can abstract away from any specific locomotion primitive such as the amoebot model's expansions and contractions.

Although most of the algorithms presented in this section focus on systems with only one robot, we present the complete model with respect to multiple robots, as it is the basis of ongoing research. A system of hybrid programmable matter is composed of k active robots operating on a set of n passive hexagonal tiles. Each tile occupies exactly one node of the triangular lattice $G_\Delta = (V, E)$ (see, e.g., Fig. 30a). A *configuration* (T, P) consists of a set $T \subset V$ of all nodes occupied by tiles, and the robots' positions $P \subset V$. Each node can be occupied by at most one robot. We describe the relative positions of adjacent nodes by the six compass directions *N*, *NE*, *SE*, *S*, *SW*, and *NW* (see Fig. 30a). As in the amoebot model, we assume that the robots have a common chirality, but do not share a coordinate system or global compass. For this chapter, we will present the algorithms using a single robot as if the robot's orientation was a global one; for ease of presentation, we will also assume a common orientation for the distributed algorithms utilizing multiple robots.

Each robot must occupy or be adjacent to a node occupied by a tile. Additionally, the subgraph of G_Δ induced by $T \cup P_c$ must stay connected, where $P_c \subseteq P$ is the set of positions occupied by robots carrying a tile. In a scenario where a tile structure swims in a liquid, for example, this restriction prevents

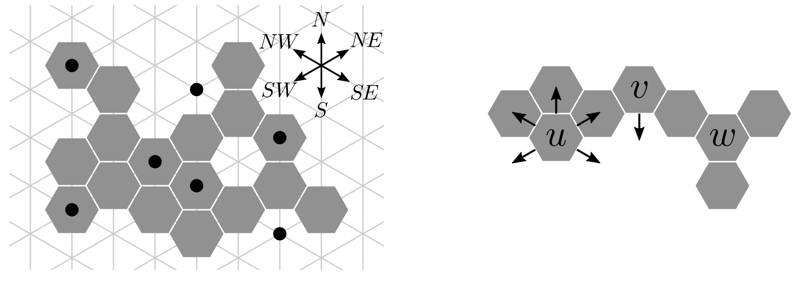


Fig. 30: (a) A connected set of tiles positioned on the triangular lattice G_Δ. The black dots indicate robot positions. (b) Possible movements of tiles u, v, and w. Tile w cannot be moved anywhere without violating connectivity.

the robots or parts of the tile structure from floating apart. Some examples of possible tile-moving steps are shown in Fig. 30b.

The robots act as finite automata and operate in rounds of `Look-Compute-Move` cycles. In the `Look` phase, a robot observes the node it occupies, say p, and the six nodes adjacent to p. For each of these nodes, the robot can determine if it is occupied by a tile, if it is occupied by a robot, and, in the latter case, the state of that robot. In the `Compute` phase, a robot potentially changes its state and determines its next move according to the observed information. Furthermore, it may change the state of any robots occupying adjacent nodes. In the `Move` phase, a robot can either (*i*) lift a tile from p if $p \in T$, (*ii*) place a tile it is carrying on p if $p \notin T$, or (*iii*) move to an adjacent node while possibly carrying a tile with it. Each robot can carry at most one tile.

A robot is additionally allowed to carry a constant number of *pebbles*, which can be placed on tiles in order to *mark* them. More specifically, in the `Look` phase, a robot can additionally observe whether any tile in its neighborhood is marked by a pebble. In the `Move` phase, in addition to its other options, a robot can either pick up a pebble (if its current tile is marked by a pebble) or place a pebble (if the current tile is not already marked and the robot has a pebble at its disposal). Pebbles are stateless and indistinguishable.

We assume the standard $\mathcal{A}$SYNC model from distributed computing in which robots are activated in an arbitrary sequence of activations, where a robot performs exactly one `Look-Compute-Move` cycle before the next robot is activated. A *round* is complete whenever each robot has been activated at least once.

7 Algorithms for Hybrid Programmable Matter

As the research in hybrid programmable matter is still in its infancy, only a few results have been established so far. In this section, we focus on two problems: *shape formation* and *shape recognition*. The former is concerned with *transforming* the tile structure into some desired shape, while the latter considers the problem of *recognizing* a given shape. As in the amoebot model, the main difficulty in designing algorithmic solutions for these problems lies in the robot's limited memory and visibility. Here, we investigate how these challenges can be overcome for hybrid programmable matter.

7.1 Algorithmic Primitives

Before presenting the shape formation and recognition algorithms, we first outline some basic results and present a set of helpful primitives.

Exploring the tile structure. The hybrid programmable matter model essentially reduces to a set of robots moving in a possibly dynamic but discrete environment. A natural question to ask is whether the robots are able to gather information about the environment. For example, one may ask whether a single robot is able to explore the tile structure, i.e., visit each node at least once, and then halt. It is known that, if the tile structure is assumed to be compact, a single robot can fully traverse the structure by visiting the adjacent *columns* (i.e., consecutive connected tiles from N to S) of any column in clockwise order. An additional pebble (or a second robot) suffices to let the algorithm terminate. However, when considering arbitrary tile structures, even a single pebble does not suffice; this follows from the observation that, when the tile structure is static, the hybrid model reduces to Finite Automata in Labyrinths [33]. On the other hand, this problem can be solved with two pebbles [5].

The exploration problem relates to many other practical problems: *tile movement safety*, i.e., deciding whether the removal of a tile would disconnect the tile set, *hole detection*, i.e., deciding whether a structure is simply connected, and *boundary detection*, i.e., reaching the structure's outer boundary. Similar to the construction given by Gmyr et al. [31], it can be shown that none of these problems can be solved by a single robot without using a pebble. The same authors show that while tile movement safety can be decided with a single pebble, the best known solutions for hole and boundary detection require two pebbles. However, these results only hold if the robots are not allowed to move tiles. Once this is allowed, these problems can be easily solved using approaches similar to the algorithms we present in Sections 7.2 and 7.3.

Safe Tile Movements. Given the previous discussion on the difficulty of deciding tile movement safety, it may seem that a complex, multi-robot strategy for identifying tiles to be moved may be necessary for ensuring tile structure connectivity. However, a different — and surprisingly simple — strategy to maintain connectivity under tile movements is the following. If the tiles in the neighborhood of a tile t satisfy *local connectivity* — i.e., they form a connected component around t — t can be safely moved, as in Fig. 31a. If the neighboring tiles form two connected components separated by a single empty node u in the neighborhood of t, then t can be safely placed onto u without violating connectivity (Fig. 31b). Otherwise, tile movement safety cannot be locally decided (Fig. 31c). Using these local rules, a tile t that is safe to move can always be found by moving NW, SW, or N (in that precedence) until reaching a tile that has no adjacent tile in any of these directions. If t satisfies local connectivity, it can be lifted and moved anywhere; otherwise, it must have adjacent tiles to the northeast and south but no adjacent tile to the southeast, so t can be placed onto the empty node to the southeast. By repeating this strategy, a robot is guaranteed to eventually find a tile that is locally connected, and therefore safely removable.

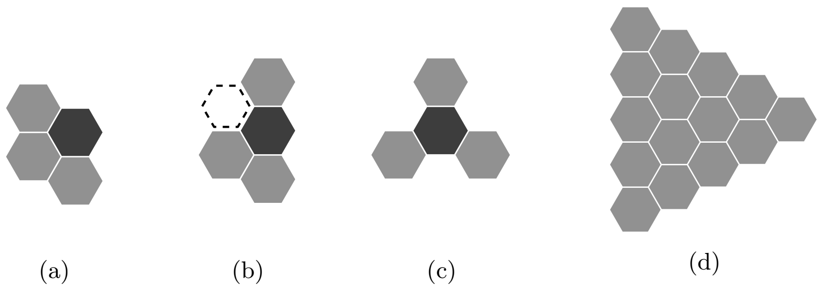


Fig. 31: The black tile can (a) be safely removed, (b) only be moved to the adjacent node marked by a dashed outline, or (c) not be moved at all. (d) The triangle to be constructed by the shape formation algorithm in Section 7.2.

7.2 Shape Formation

Arguably one of the most interesting problems for hybrid programmable matter is shape formation. Shape formation problems may consider different shapes to form (e.g., a hexagon as in Section 4.1, or a sequentially constructible shape as in Section 4.2), different optimization goals (e.g., runtime, or distance of moved tiles), and side conditions (e.g., avoiding moving tiles beyond the initial structure's convex hull). In this section, we consider the problem of forming a triangle. We first present an algorithm for triangle formation by a single robot, and then show how it can be modified to handle a multi-robot setting. Finally, we present two variants of the algorithm that aim at minimizing the algorithm's running time and only moving tiles within the initial structure's convex hull, respectively. The section is based on the work of Gmyr et al. [31], where proofs and more detailed descriptions of the algorithms can be found.

Problem Description Consider a connected set of tiles and a single robot that is initially placed on some tile. An algorithm solves the triangle formation problem for hybrid programmable matter if it transforms the initial tile structure into a triangle that is axis-aligned along the robot's north-south axis and grows from east to west (see Fig. 31d).

Algorithm In a naive approach to shape formation, the robot could iteratively search for a tile that can be removed without disconnecting the tile structure and then move that tile to some position such that the shape under construction is extended. Although a safely removable tile always exists, a robot may not be able to find it as previously discussed. Instead, the safe tile movements discussed in Section 7.1 are used to first transform the structure into a *line* (i.e., a sequence of connected tiles from north to south). From this intermediate structure, a triangle can easily be constructed in a second stage.

To construct a line, the robot first moves *S* as far as possible, i.e., as long as there is a tile in direction *S*. Then, it alternates between a *tile searching phase*, in which it moves *N*, *NW*, and *SW* (in that precedence) until there is no longer a tile in any of these directions; and a *tile moving phase*, in which it lifts the tile, moves one step *SE*, moves *S* until it reaches an empty node, and then places the tile. The line is complete once the robot does not encounter any adjacent tiles to the east or west in the tile searching phase. Fig. 32 shows the first several steps of this algorithm.

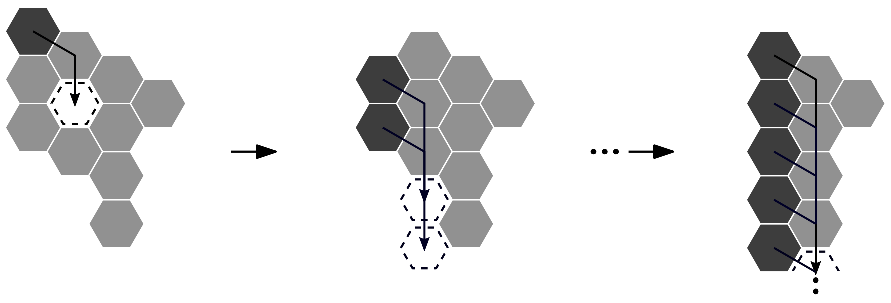


Fig. 32: First several steps of line formation. The black tiles are moved to the positions marked by dashed outlines.

In the second stage, the triangle is built by repeatedly taking the northernmost tile of the line, carrying it south to the *vertex* of the forming triangle, and adding it to the westernmost *layer* of the triangle (see Fig. 33). More specifically, the robot places the first tile *NW* of the triangle's vertex. Every other tile of the triangle is then placed as follows. The robot first takes the northernmost tile of the line and brings it to the vertex. It then walks *NW* and *S* (in that precedence) until an empty node is reached. If there is a tile to the southeast, the robot moves one step *S* and places the tile. Otherwise, the robot moves *N* to the top of the layer, takes one step *NW*, and places the tile. In this manner, the robot continues to extend the triangle tile by tile until the line only consists of the triangle's vertex.

Analysis We now state the main results for the triangle formation algorithm.

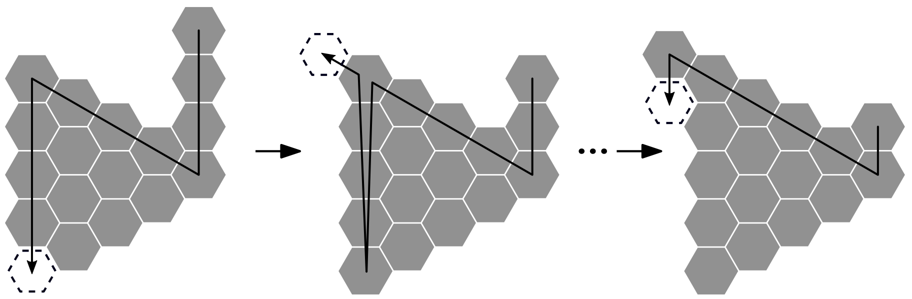


Fig. 33: Snapshots of triangle formation. If the number of tiles is not triangular, the final layer will not be completely filled.

Correctness. The two stages of the algorithm can be analyzed separately. The correctness of the line formation stage follows from (*i*) the tile searching phase always leads the robot to discover a safe tile to move, (*ii*) the tile moving phase never disconnects the tile structure, and (*iii*) the algorithm terminates when a line is formed. The correctness of the triangle formation stage is easily established, resulting in the following theorem.

Theorem 17. *The algorithm correctly transforms any connected tile structure into a triangle.*

Runtime. The two stages are again analyzed separately. In the first stage, each tile is moved by at most n steps. This is due to the fact that the easternmost column of the initial configuration is never moved, and since tiles are only moved *S* and *SE*. As there are n tiles, the total number of rounds that account to moving tiles is bounded by $\mathcal{O}(n^2)$. Additionally, the robot has to move through the structure in order to search for tiles. By assigning coordinates to the nodes and using the coordinates of the robot as a potential function, it can be shown that the total number of rounds that account to searching for tiles is also bounded by $\mathcal{O}(n^2)$. In the second stage, each tile is carried by a distance of at most n, and moving to the next tile takes an additional n steps at most. Therefore, we have the following theorem.

Theorem 18. *The algorithm constructs a triangle within $\mathcal{O}(n^2)$ rounds.*

It is not hard to see that $\Omega(n^2)$ rounds are necessary to rearrange an arbitrary initial tile configuration into a triangle using a single robot. If the initial configuration is a line, then a constant fraction of the tiles must be moved by a distance linear in n and thus, in total, $\Omega(n^2)$ move steps are necessary.

Distributed Algorithm In order to extend the algorithm to construct a triangle with multiple robots that work in coordination, several challenges must be overcome. First, robots which are *hanging* off the edge of the tile structure

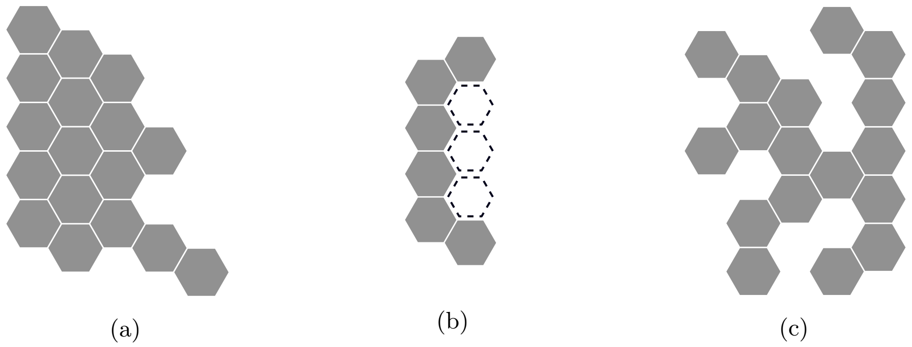


Fig. 34: A (a) block, (b) overhang, and (c) tree.

may be disconnected if another robot lifts the tile they were hanging on. Thus, a robot checks for any hanging robots before lifting a tile. Second, the line formation stage must be modified so that all robots eventually learn the line has been formed. Finally, robots may obstruct one another's progress when forming the line and triangle, requiring them to either communicate or simply wait in order to become unblocked.

Although correctness can be proven for this multi-robot approach, it is difficult to make any runtime guarantees. This is due to the fact that, when there are many robots compared to the number of tiles, many robots are blocked by others and must wait to make progress. However, simulation results for distributed line formation [31] suggest a reasonable speedup for few robots in randomly generated initial configurations. In order to guarantee a speedup for multiple robots, we believe different formation strategies that better utilize coordination between the robots will be useful.

Alternative Intermediate Structures Although a line can be constructed efficiently, its linear *diameter* (i.e., the maximal length of a shortest path between any two tiles) may make it an undesirable intermediate structure. In fact, if both the initial diameter D and the diameter of the desired shape are small, moving tiles by a linear distance seems to be an excessive effort. Therefore, we briefly describe how to construct two alternate intermediate structures, namely a *block* and a *tree*, noting their advantages and disadvantages.

Block Formation. In a block, all tiles except those farthest to the west have a neighbor to the northwest. A block has only one westernmost column, and every row begins with a tile from that column (see Fig. 34a).

As in the line formation algorithm, constructing a block alternates between searching and moving phases. The robot first searches for a locally northwesternmost tile by repeatedly moving *NW*, *SW*, or *N* (in that precedence). The robot then lifts the tile, moves *SE* until it reaches an empty node, and places the tile there. Although this simple algorithm correctly constructs a block, detecting its completion requires a series of more complex tests performed alongside the block's construction.

Theorem 19. *The algorithm constructs a block within $\mathcal{O}(nD)$ rounds and ensures that no tile is ever moved by more than a distance D.*

Note that since tiles are exclusively moved *SE*, the resulting block has at most D rows consisting of at most D tiles each, and therefore has diameter $\mathcal{O}(D)$. Therefore, by using a block as an intermediate structure, a triangle can be constructed in $\mathcal{O}(nD)$ rounds. When the initial configuration's diameter is low, i.e., $D = \mathcal{O}(\sqrt{n})$, a triangle can thus be formed in $\mathcal{O}(n^{3/2})$ rounds.

Tree Formation. While the block-based approach focuses on quickly constructing a suitable intermediate structure, it may also be desirable to minimize the required work space. Both the line and block are, in many cases, built almost completely outside the initial configuration's convex hull (where we refer to the convex hull of the corresponding set of hexagonal tiles in the Euclidean plane). We briefly describe an algorithm that builds a tree in time $\mathcal{O}(n^2)$ by exclusively moving tiles inside the structure's convex hull. A *tree* is a connected tile configuration that does not contain an *overhang*, i.e., a set of vertically adjacent empty nodes bounded by tiles to the north, west, and south. Examples of an overhang and a tree can be found in Figs. 34b and 34c, respectively.

We describe the tree formation algorithm at a high level and refer the interested reader to [31] for a detailed description. Roughly speaking, the robot traverses the columns of the tile structure from west to east until it encounters an overhang. It then fills the overhang by retrieving tiles from western columns. Here, the robot exploits the property that the western columns no longer have overhangs, which allows it to find safely removable tiles efficiently and to bring them back to the overhang. After filling the overhang, the algorithm recurs. Correctness of this approach is established by proving that the algorithm (i) fills an overhang if one exists and (ii) terminates after a complete traversal. Together with a detailed runtime analysis, we have the following theorem.

Theorem 20. *The algorithm constructs a tree within $\mathcal{O}(n^2)$ rounds without ever placing a tile outside the initial structure's convex hull.*

7.3 Shape Recognition

For many applications of programmable matter such as shape formation, transformation, repair, or sealing, it may be useful or even necessary to first determine whether the initial structure has a certain shape. However, as already argued, the detection capabilities of a single robot are very limited: it is, for example, not able to distinguish (i) a spiral from a hollow hexagon [31], or (ii) a parallelogram whose longer side is, say, the square of its shorter side, from a larger parallelogram [30]. Therefore, we begin by presenting a set of simple structures that can be detected by a single robot. Afterwards, we present some results for the recognition of parallelograms of certain side ratios with the help of pebbles. Note that instead of using pebbles the robot may also cooperate with other robots, each mimicking the behaviour of a pebble. The details of the results presented in this chapter can be found in [30].

Problem Description Consider a single robot that is placed on some tile of an arbitrary initial tile configuration. The shape recognition problem tasks the robot with deciding if the tile structure is of a certain shape, e.g., a line, triangle, hexagon, or parallelogram. If the shape is a parallelogram, the robot must additionally decide if its longer side has length $\ell = f(h)$, for a given function $f(\cdot)$, where h is the length of its shorter side. In this section, we ask whether $\ell = ah + b$ for some constants a and b.

Algorithm All four types of shapes can be recognized using a similar strategy. For example, to test if a given tile shape is a line, the robot first chooses a direction in which there is a tile (say, w.l.o.g., N), walks in that direction as far as possible, and then traverses the structure in the opposite direction until no longer possible. If it ever encounters a tile to the east or west of any traversed tile, the structure is not a line.

To test if a given tile shape is a (filled) parallelogram axis-aligned along the robot's N and NE directions, the robot first moves to a locally southernmost tile by moving S and SW as long as there is a tile in either of these directions. It then traverses the shape column by column in a snake-like fashion (see Fig. 35a) by repeating the following movements: move N as far as possible; move one step NE; move S as far as possible, and finally move one step NE. The above procedure is repeated until a NE movement is impossible. The robot can decide whether the structure is a parallelogram by performing a sequence of checks alongside these movements. Any other axis-aligned parallelogram, and all other aforementioned shapes, can be tested in a similar fashion.

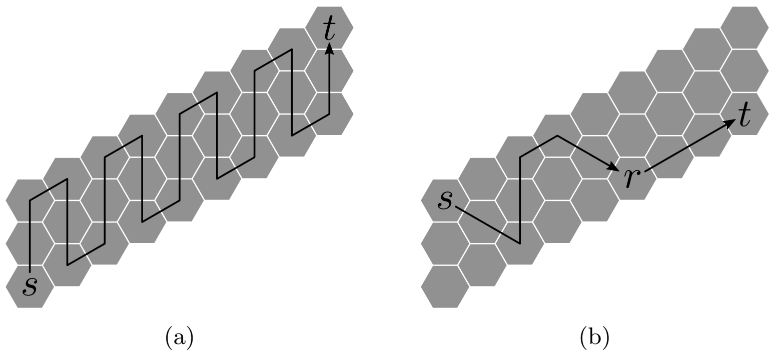


Fig. 35: (a) The snake-like traversal to detect a parallelogram. (b) The traversal to decide whether $\ell = ah + b$ (where in this case $a = 2$ and $b = 3$).

Now consider the problem of determining whether $\ell = ah+b$. W.l.o.g., assume that the longer side of the parallelogram is in the northeast direction. The longer side can be determined by moving to the northernmost tile of column 0 (where the columns are numbered from 0 to $\ell - 1$ from west to east), and then moving *SE* as far as possible; if there is a tile to the southeast (resp., to the south), then the longer side is in the northeastern (resp., southern) direction. If there is no tile to the northeast or south, then the parallelogram is a rhombus.

To decide whether $\ell = ah + b$, the robot first moves to the northernmost tile of column 0. It then traverses the tile structure in two stages to verify the ratio of the sides. In the first stage, the robot "measures" the distance ah along the length of the parallelogram by moving in a zig-zag fashion as depicted in Fig. 35b. In the second stage, the robot measures b. More specifically, in the first stage, the robot repeats the following movements in a loop: (*i*) move *SE* as far as possible, (*ii*) move *N* as far as possible, and (*iii*) move one step *NE*. After having performed the complete sequence of *SE* movements a times, the robot moves on to the second stage, in which it makes an additional b *NE* steps.

If the robot reaches the easternmost column before completing the above procedure, or halts on a tile with a neighboring tile to the northeast, it terminates with a negative result. Otherwise, it terminates with a positive result.

Analysis We have the following theorem for shape recognition.

Theorem 21. *A single robot can detect whether the structure is a line, a triangle, a hexagon, or a parallelogram. Furthermore, it can detect whether the structure is a parallelogram with $\ell = ah + b$ for any constants $a, b \in \mathbb{N}$.*

However, the following theorem shows that a single robot by itself cannot hope to verify any side ratios of a parallelogram that are not linear.

Theorem 22. *A single robot without any pebbles cannot decide whether the tile configuration is a parallelogram with $\ell = f(h)$, where $f(x) = \omega(x)$.*

Theorem 22 can be proven using the following observation: to correctly detect the length of a parallelogram of length $f(h)$, the robot must move from the westernmost to the easternmost column (or vice versa) at least once (otherwise it would be unaware of any elongation of the parallelogram). Therefore, if h is chosen such that $\lfloor (f(h) - 2)/h \rfloor > k$, where k is the given number of states of the robot, there must be a row of the parallelogram in which the robot steps on more than k tiles of columns between column 1 and $\ell - 2$, which implies that it visits at least two of these tiles in the same state. Therefore, there must be a repetition in the traversal between column 0 and $\ell - 1$, and if the length of the parallelogram is elongated by a multiple of the distance between these two tiles, the robot will not recognize this change. This can be used to construct an elongated parallelogram that is indistinguishable from the original one by a single particle with no pebbles.

The Power of Pebbles For the problem of recognizing parallelograms of certain side ratios, equipping a robot with a set of pebbles tremendously increases its capabilities. For example, whereas a single robot without any pebbles cannot decide any superlinear function, by Theorem 22, a single pebble suffices to decide any polynomial of constant degree and with constant coefficients. Furthermore, two pebbles suffice to decide certain exponential functions. An overview of some results for a robot with pebbles, which can be found in full detail in the work of Gmyr et al. [30], is shown in Table 1.

Pebbles	Possible	Impossible	Remarks
1	$f(x) = a_n x^n + \ldots + a_0$	$f(x) = \omega(x^{6k+2})$	n, a_i constant for all i; k is the number of the robot's states
2	$f(x) = \underbrace{2^{2^{\cdot^{\cdot^{2^x}}}}}_{s+1}$	—	s constant; asymptotic lower bound does not exist
n	$f_n(x)$	$f_{n+1}(x)$	for some function family f_n

Table 1: A summary of results for recognizing whether a given parallelogram has height h and length $\ell = f(h)$ given a certain number of pebbles.

8 Conclusion and Future Work

This chapter presented a comprehensive review of the distributed algorithms for programmable matter defined under the amoebot model, and surveyed some initial results in the budding new model of hybrid programmable matter. For the amoebot model, we presented two distinct types of algorithms: those that are (mostly) deterministic and heavily utilize state management and particle communication, and those that are fully stochastic, keeping little to no state and requiring very little communication between particles. There is an inherent tradeoff between these two approaches. The former often yields algorithms that provably terminate within a linear number of rounds or better, but are more complex to design, more difficult to implement, and have single points of failure. On the other hand, the stochastic algorithms are very difficult to analyze in terms of convergence times and are observed to be relatively slow in simulations; however, they are inherently robust and are relatively easy to design.

Looking forward, there are several intriguing research directions for the amoebot model. First, any physical implementation of programmable matter would need to meaningfully address the challenge of power management as it expends energy moving and computing. Extending the amoebot model to incorporate energy costs for particle actions could lead to interesting modeling questions, such as how individual particles obtain energy and share it among the collective. Moreover, considering energy usage as an alternative to time complexity when analyzing algorithm efficiency could yield new paradigms for algorithm design. Second, developing a general framework for fault tolerant algorithms under the amoebot model would be a huge step towards realizing programmable matter systems that can handle unpredictable and potentially hazardous application domains. Third, generalizing the amoebot model to three-dimensional space is an exciting goal and a current research direction that may bring our theoretical investigations closer to physically realizable systems.

This is also of interest for hybrid programmable matter; in particular, considering three-dimensional tiles that are hollow (e.g., skeletal polyhedra) would allow active robots to move through the resulting structure, generalizing the movement primitives used in two-dimensional space. One could additionally imagine that the hollow tiles could fold into a condensed shape, enabling robots to transport tiles through other tiles. Other directions for future work on hybrid programmable matter include developing algorithms for multiple robots that provably benefit from the power of coordination, considering settings where tiles may break or need to be repaired, and settings where tiles may attach and detach at random based on environmental changes outside of the robots' control.

Acknowledgements

Our warmest gratitude belongs to all of our wonderful collaborators, both past and present, without whom this research would not have been possible. We would like to thank Robert Gmyr, Thim Strothmann, and Zahra Derakhshandeh for their trailblazing work on self-organizing particle systems during their PhD studies. We would especially like to thank Robert for letting us use materials from his PhD thesis for this chapter (in particular, his excellent images). To Dana Randall and Sarah Cannon, thank you for leading us into a new paradigm by showing us just how much one can do with a whole lot of randomness. To Irina Kostitsyna and Dorian Rudolph, thank you for all your work in developing hybrid programmable matter. Finally, to our undergraduate research assistants, especially Alexandra Porter: thank you for your enthusiasm, energy, and effort.

References

1. Andrés Arroyo, M., Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A stochastic approach to shortcut bridging in programmable matter. In: DNA Computing and Molecular Programming — 23rd International Conference. pp. 122–138. DNA23 (2017), an updated journal version is awaiting publication and available online at `https://arxiv.org/abs/1709.02425`
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing **18**(4), 235–253 (2006)
3. Baxter, R.J., Enting, I.G., Tsang, S.K.: Hard-square lattice gas. Journal of Statistical Physics **22**, 465–489 (1980)

4. Blanca, A., Chen, Y., Galvin, D., Randall, D., Tetali, P.: Phase coexistence for the hard-core model on $\mathbb{Z}^2$. Combinatorics, Probability and Computing pp. 1–22 (2018)
5. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: 19th Annual Symposium on Foundations of Computer Science. pp. 132–142. SFCS '78 (1978)
6. Bonato, A., Nowakowski, R.J.: The Game of Cops and Robbers on Graphs. AMS (2011)
7. Bonifaci, V., Mehlhorn, K., Varma, G.: Physarum can compute shortest paths. Journal of Theoretical Biology **309**, 121–133 (2012)
8. Borgs, C., Chayex, J.T., Kim, J.H., Frieze, A., Tetali, P., Vigoda, E., Vu, V.H.: Torpid mixing of some Monte Carlo Markov chain algorithms in statistical physics. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science. pp. 218–229. FOCS '99 (1999)
9. Camazine, S., Visscher, P.K., Finley, J., Vetter, R.S.: House-hunting by honey bee swarms: Collective decisions and individual behaviors. Insectes Sociaux **46**(4), 348–360 (1999)
10. Cannon, S., Daymude, J.J., Gokmen, C., Randall, D., Richa, A.W.: Brief announcement: A local stochastic algorithm for separation in heterogeneous self-organizing particle systems. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. pp. 483–485. PODC '18 (2018), a full version is available online at `https://arxiv.org/abs/1805.04599`
11. Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A Markov chain algorithm for compression in self-organizing particle systems. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. pp. 279–288. PODC '16 (2016), a significantly updated journal version is in preparation and available online at `https://arxiv.org/abs/1603.07991`
12. Chirikjian, G.S.: Kinematics of a metamorphic robotic system. In: Proceedings of the 1994 IEEE International Conference on Robotics and Automation. ICRA '94, vol. 1, pp. 449–455 (1994)
13. Das, S.: Mobile agents in distributed computing: Network exploration. Bulletin of the European Association for Theoretical Computer Science **109**, 54–69 (2013)
14. Daymude, J.J., Derakhshandeh, Z., Gmyr, R., Porter, A., Richa, A.W., Scheideler, C., Strothmann, T.: On the runtime of universal coating for programmable matter. Natural Computing **17**(1), 81–96 (2018)
15. Daymude, J.J., Gmyr, R., Hinnenthal, K., Kostitsyna, I., Scheideler, C., Richa, A.W.: Convex hull formation for programmable matter (2018), preprint online at `https://arxiv.org/abs/1805.06149`
16. Daymude, J.J., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Improved leader election for self-organizing programmable matter. In: Algorithms for Sensor Systems. pp. 127–140. ALGOSENSORS '17 (2017)
17. Daymude, J.J., Richa, A.W., Scheideler, C.: The amoebot model (2017), available online at `https://sops.engineering.asu.edu/sops/amoebot`
18. Derakhshandeh, Z., Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Brief announcement: Amoebot - a new model for programmable matter. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 220–222. SPAA '14 (2014)
19. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: An algorithmic framework for shape formation problems in self-organizing particle systems. In: Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication. pp. 21:1–21:2. NANOCOM '15 (2015)

20. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 289–299. SPAA '16 (2016)
21. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal coating for programmable matter. Theoretical Computer Science **671**, 56–68 (2017)
22. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R.A., Richa, A.W., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. In: DNA Computing and Molecular Programming — 21st International Conference. pp. 117–132. DNA21 (2015)
23. Di Luna, G.A., Flocchini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Shape formation by programmable particles. In: 21st International Conference on Principles of Distributed Systems. OPODIS '17, vol. 95, pp. 31:1–31:16 (2018)
24. Di Luna, G.A., Flocchini, P., Prencipe, G., Santoro, N., Viglietta, G.: Line recovery by programmable particles. In: Proceedings of the 19th International Conference on Distributed Computing and Networking. pp. 4:1–4:10. ICDCN '18 (2018)
25. Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C.: Ameba-inspired self-organizing particle systems (2013), workshop paper at Biological Distributed Algorithms (BDA) 2013. Available online at `https://arxiv.org/abs/1307.4259`
26. Doty, D.: Theory of algorithmic self-assembly. Communications of the ACM **55**(12), 78–88 (2012)
27. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. 1. Wiley, New York (1968)
28. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theoretical Computer Science **399**(3), 236–245 (2008)
29. Gmyr, R.: Distributed Algorithms for Overlay Networks and Programmable Matter. Ph.D. thesis, Paderborn University (2017)
30. Gmyr, R., Hinnenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C.: Shape recognition by a finite automaton robot. In: 43rd International Symposium on Mathematical Foundations of Computer Science. pp. 52:1–52:15. MFCS '18 (2018)
31. Gmyr, R., Hinnenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C., Strothmann, T.: Forming tile shapes with simple robots. In: DNA Computing and Molecular Programming — 24rd International Conference. DNA24 (2018)
32. Hastings, W.K.: Monte carlo sampling methods using Markov chains and their applications. Biometrika **57**(1), 97–109 (1970)
33. Hoffmann, F.: One pebble does not suffice to search plane labyrinths. In: Fundamentals of Computation Theory. pp. 433–444. FCT '81 (1981)
34. Jeanson, R., Rivault, C., Deneubourg, J.L., Blanco, S., Fournier, R., Jost, C., Theraulaz, G.: Self-organized aggregation in cockroaches. Animal Behaviour **69**(1), 169–180 (2005)
35. Lund, K., Manzo, A.J., Dabby, N., Michelotti, N., Johnson-Buck, A., Nangreave, J., Taylor, S., Pei, R., Stojanovic, M.N., Walter, N.G., Winfree, E., Yan, H.: Molecular robots guided by prescriptive landscapes. Nature **465**(7295), 206–210 (2010)
36. Lynch, N.: Distributed Algorithms. Morgan Kauffman (1996)
37. Miracle, S., Randall, D., Streib, A.: Clustering in interfering binary mixtures. In: 15th International Workshop on Randomization and Approximation Techniques in Computer Science. pp. 652–663. RANDOM '11 (2011)

38. Mlot, N.J., Tovey, C.A., Hu, D.L.: Fire ants self-assemble into waterproof rafts to survive floods. Proceedings of the National Academy of Sciences **108**(19), 7669–7673 (2011)
39. Omabegho, T., Sha, R., Seeman, N.C.: A bipedal DNA Brownian motor with coordinated legs. Science **324**(5923), 67–71 (2009)
40. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Natural Computing **13**(2), 195–224 (2014)
41. Pelc, A.: Deterministic rendezvous in networks: A comprehensive survey. Networks **59**(3), 331–347 (2012)
42. Porter, A., Richa, A.: Collaborative computation in self-organizing particle systems. In: Unconventional Computation and Natural Computation. pp. 188–203. UCNC '18 (2018)
43. Reid, C.R., Lutz, M.J., Powell, S., Kao, A.B., Couzin, I.D., Garnier, S.: Army ants dynamically adjust living bridges in response to a cost–benefit trade-off. Proceedings of the National Academy of Sciences **112**(49), 15113–15118 (2015)
44. Reid, C.R., Latty, T.: Collective behaviour and swarm intelligence in slime moulds. FEMS Microbiology Reviews **40**(6), 798–806 (2016)
45. Reif, J.H., Sahu, S.: Autonomous programmable DNA nanorobotic devices using dnazymes. Theoretical Computer Science **410**, 1428–1439 (2009)
46. Restrepo, R., Shin, J., Tetali, P., Vigoda, E., Yang, L.: Improving mixing conditions on the grid for counting and sampling independent sets. Probability Theory and Related Fields **156**, 75–99 (2013)
47. Şahin, E.: Swarm robotics: From sources of inspiration to domains of application. In: Swarm Robotics. pp. 10–20 (2005)
48. Savoie, W., Cannon, S., Daymude, J.J., Warkentin, R., Li, S., Richa, A.W., Randall, D., Goldman, D.I.: Phototactic supersmarticles (2018), to appear in Artificial Life and Robotics; preprint available online at `https://arxiv.org/abs/1711.01327`
49. Schelling, T.C.: Models of segregation. The American Economic Review **59**(2), 488–493 (1969)
50. Schelling, T.C.: Dynamic models of segregation. The Journal of Mathematical Sociology **1**(2), 143–186 (1971)
51. Shin, J.S., Pierce, N.A.: A synthetic DNA walker for molecular transport. Journal of the American Chemical Society **126**(35), 10834–10835 (2004)
52. Strothmann, T.F.: Self-* Algorithms for Distributed Systems: Programmable Matter & Overlay Networks. Ph.D. thesis, Paderborn University (2017)
53. Thubagere, A.J., Li, W., Johnson, R.F., Chen, Z., Doroudi, S., Lee, Y.L., Izatt, G., Wittman, S., Srinivas, N., Woods, D., Winfree, E., Qian, L.: A cargo-sorting DNA robot. Science **357**(6356) (2017)
54. Toffoli, T., Margolus, N.: Programmable matter: Concepts and realization. Physica D: Nonlinear Phenomena **47**(1), 263–272 (1991)
55. Vinković, D., Kirman, A.: A physical analogue of the Schelling model. Proceedings of the National Academy of Sciences **103**(51), 19261–19265 (2006)
56. Walter, J.E., Tsai, E.M., Amato, N.M.: Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. IEEE Transactions on Robotics **21**(4), 621–631 (2005)
57. Wickham, S.F., Bath, J., Katsuda, Y., Endo, M., Hidaka, K., Sugiyama, H., Turberfield, A.J.: A DNA-based molecular motor that can navigate a network of tracks. Nature Nanotechnology **7**(3), 169–173 (2012)

58. Woods, D., Chen, H.L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science. pp. 353–354 (2013)
59. Yim, M., Shen, W.M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular self-reconfigurable robot systems [grand challenges of robotics]. IEEE Robotics Automation Magazine **14**(1), 43–52 (2007)