Fast Reconfiguration for Programmable Matter

Irina Kostitsyna
TU Eindhoven, The Netherlands

Tom Peters
TU Eindhoven, The Netherlands

Bettina Speckmann
TU Eindhoven, The Netherlands

Abstract

The concept of programmable matter envisions a very large number of tiny and simple robot particles forming a smart material. Even though the particles are restricted to local communication, local movement, and simple computation, their actions can nevertheless result in the global change of the material's physical properties and geometry.

A fundamental algorithmic task for programmable matter is to achieve global shape reconfiguration by specifying local behavior of the particles. In this paper we describe a new approach for shape reconfiguration in the *amoebot* model. The amoebot model is a distributed model which significantly restricts memory, computing, and communication capacity of the individual particles. Thus the challenge lies in coordinating the actions of particles to produce the desired behavior of the global system.

Our reconfiguration algorithm is the first algorithm that does not use a canonical intermediate configuration when transforming between arbitrary shapes. We introduce new geometric primitives for amoebots and show how to reconfigure particle systems, using these primitives, in a linear number of activation rounds in the worst case. In practice, our method exploits the geometry of the symmetric difference between input and output shape: it minimizes unnecessary disassembly and reassembly of the particle system when the symmetric difference between the initial and the target shapes is small. Furthermore, our reconfiguration algorithm moves the particles over as many parallel shortest paths as the problem instance allows.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Self-organization

Keywords and phrases Programmable matter, amoebot model, shape reconfiguration

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.27

Related Version *Full Version*: `https://arxiv.org/abs/2202.11663` [22]

1 Introduction

Programmable matter is a smart material composed of a large quantity of robot particles capable of communicating locally, performing simple computation, and, based on the outcome of this computation, changing their physical properties. Particles can move through a programmable matter system by changing their geometry and attaching to (and detaching from) neighboring particles. By instructing the particles to change their local adjacencies, we can program a particle system to reconfigure its global shape. Shape assembly and reconfiguration of particle systems have attracted a lot of interest in the past decade and a variety of specific models have been proposed [5, 20, 25, 28, 18, 12, 24, 26]. We focus on the *amoebot* model [13], which we briefly introduce below. Here, the particles are modeled as independent agents collaboratively working towards a common goal in a distributed fashion. The model significantly restricts computing and communication capacity of the individual particles, and thus the challenge of programming a system lies in coordinating local actions of particles to produce a desired behavior of the global system.

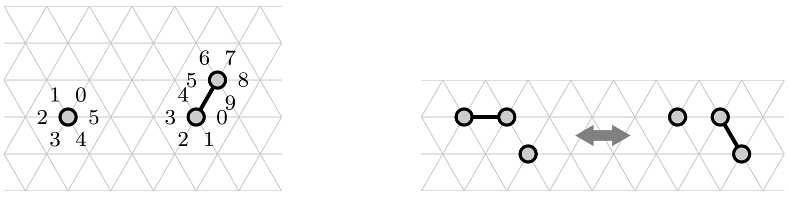

Figure 1 Left: particles with ports labeled, in contracted and expanded state. Right: handover operation between two particles.

A fundamental problem for programmable matter is shape reconfiguration. To solve it we need to design an algorithm for each particle to execute, such that, as a result, the programmable matter system as a whole reconfigures into the desired target shape. Existing solutions first build an intermediate *canonical configuration* (usually, a line or a triangle) and then build the target shape from that intermediate configuration [14, 16]. However, in many scenarios, such as shape repair, completely deconstructing a structure only to build a very similar one, is clearly not the most efficient strategy.

We propose the first approach for shape reconfiguration that does not use a canonical intermediate configuration when transforming between two arbitrary shapes. Our algorithm exploits the geometry of the symmetric difference between the input shape I and the target shape T. Specifically, we move the particles from $I \setminus T$ to $T \setminus I$ along shortest paths through the overlap $I \cap T$ over as many parallel shortest paths as the problem instance allows. In the worst case our algorithm works as well as existing solutions. However, in practice, our approach is significantly more efficient when the symmetric difference between the initial and the target shape is small.

Amoebot model. Particles in the amoebot model occupy nodes of a plane triangular grid G. A particle can occupy either a single node or a pair of adjacent nodes: the particle is contracted and expanded, respectively. The particles have constant memory space, and thus have limited computational power. They are disoriented (no common notion of orientation) and there is no consensus on chirality (clockwise or counter-clockwise). The particles have no ids and execute the same algorithm, i.e. they are identical. They can reference their neighbors using six (for contracted) or ten (for expanded particles) *port identifiers* that are ordered clockwise or counter-clockwise modulo six or ten (see Figure 1 (left)). Particles can communicate with direct neighbors by sending messages over the ports. Refer to Daymude et al. [10] for additional details.

Particles can move in two different ways: a contracted particle can *expand* into an adjacent empty node of the grid, and an expanded particle can *contract* into one of the nodes it currently occupies. Each node of G can be occupied by at most one particle, and we require that the particle system stays connected at all times. To preserve connectivity more easily, we allow a *handover* variant of both move types, a simultaneous expansion and contraction of two neighboring particles using the same node (Figure 1 (right)). A handover can be initiated by any of the two particles; if it is initiated by the expanded particle, we say it *pulls* its contracted neighbor, otherwise we say that it *pushes* its expanded neighbor.

Particles operate in activation cycles: when activated, they can read from the memory of their immediate neighbors, compute, send constant size messages to their neighbors, and perform a move operation. Particles are activated by an asynchronous adversarial but fair scheduler (at any moment in time t, any particle must be activated at some time in the future $t' > t$). If two particles are attempting conflicting actions (e.g., expanding into the same node), the conflict is resolved by the scheduler arbitrarily, and exactly one of these actions succeeds. We perform running time analysis in terms of the number of *rounds*: time intervals in which all particles have been activated at least once.

We say that a *particle configuration* $\mathcal{P}$ is the set of all particles and their internal states. Let $G_{\mathcal{P}}$ be the subgraph of the triangular grid G induced by the nodes occupied by particles in $\mathcal{P}$. Let a *hole* in $\mathcal{P}$ be any interior face of $G_{\mathcal{P}}$ with more than three vertices. We say a particle configuration $\mathcal{P}$ is *connected* if there exists a path in $G_{\mathcal{P}}$ between any two particles. A particle configuration $\mathcal{P}$ is *simply connected* if it is connected and has no holes.

Related work. The amoebot model, which is both natural and versatile, was introduced by Derakhshandeh et al. [13] in 2014 and has recently gained popularity in the algorithmic community. A number of algorithmic primitives, such as leader election [15, 16, 17], spanning forests [15], and distributed counters [8, 27], were developed to support algorithm design.

Derakhshandeh et al. [14] designed a reconfiguration algorithm for an amoebot system which starts from particles forming a large triangle and targets a shape consisting of a constant number of unit triangles (such that their description fits into the memory of a single particle). In their approach the initial large triangle is partitioned into the unit triangles, which move in a coordinated manner to their corresponding position within the target shape. Derakhshandeh et al. make some assumptions on the model, including a sequential activation scheduler (at every moment in time only one particle can be active), access of particles to randomization, and common particle chirality. Due to these assumptions, and the fact that the initial shape is compact, the reconfiguration process takes $O(\sqrt{n})$ number of rounds for a system with n particles.

Di Luna et al. [16] were the first to reconfigure an input particle system into a line (a canonical intermediate shape). They then simulate a Turing machine on this line, and use the output of the computation to direct the construction of the target structure in $O(n \log n)$ rounds. Their main goal was to lift some of the simplifying assumptions on the model of [14]: their algorithm works under a synchronous scheduler, is deterministic, does not rely on the particles having common chirality, and requires only the initial structure to be simply connected. However, just as the work by Derakhshandeh et al. [14], their method only works for structures of constant description size.

Cannon et al. [4] consider a stochastic variation of the amoebot model. Viewed as an evolving Markov chain, the particles make probabilistic decisions based on the structure of their local neighborhoods. In this variant, there exist solutions for compressing a system into a dense structure [4], simulating ant behavior to build a shortcut bridge between two locations [1], and separating a system into multiple components by the *color* of particles [3].

Problem description. An instance of the *reconfiguration problem* consists of a pair of simply connected shapes (I, T) embedded in the grid G (see Figure 2). The goal is to transform the initial shape I into the target shape T. Initially, all particles in I are contracted. The problem is solved when there is a contracted particle occupying every node of T.

We make a few assumptions on the input and on the model. Most of our assumptions fall into at least one of the following categories: they are natural for the problem statement, they can be lifted with extra care, or they are not more restrictive than existing work.

Assumptions on I and T. We assume that the input shape I and the target shape T have the same number n of nodes. We call $I \cap T$ the *core* of the system, the particles in $I \setminus T$ the *supply*, and the particles in $T \setminus I$ the *demand*. In our algorithm, the core nodes are always occupied by particles, and the supply particles move through the core to the demand.

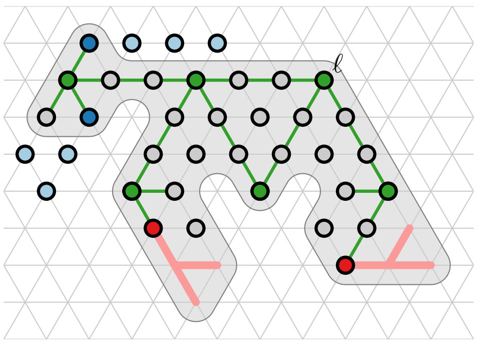

Figure 2 The particles form the initial shape I. The target shape T is shaded in gray. Supply particles are blue, supply roots dark blue. Demand roots (red) store spanning trees of their demand components. The graph G_L on the coarse grid is shown in green and leader ℓ is marked. Particles on G_L that are green are grid nodes, other particles on G_L are edge nodes.

We assume that

A1 the core $I \cap T$ is a non-empty simply connected component. This is a natural assumption in the shape repair scenario when the symmetric difference between the initial and the target shape is small.

A2 each demand component D of $T \setminus I$ has a constant description complexity.

In Section 6 we discuss a possible strategy for lifting A1. Below A6 we explain why A2 is not more restrictive than the input assumptions in the state-of-the art.

Assumptions on initial particle state. A standard assumption is that initially all particles have the same state. In this paper we assume that a preprocessing step has encoded T into the particle system and thus the states of some particles have been modified from the initial state. The reasons for this assumption are twofold: first, to limit the scope of this paper we chose to focus on the reconfiguration process; and second, the encoding preprocessing step, whose specification we omit, can be derived in a straightforward manner from existing primitives and algorithms. Below we describe more precisely what our assumptions are on the outcome of the preprocessing step.

To facilitate the navigation of particles through the core $I \cap T$, and in particular, to simplify the crossings of different flows of particles, we use a coarsened (by a factor of three) version of the triangular grid. Let G_L be the intersection of the coarse grid with $I \cap T$ (the green nodes Figure 2). We assume that after the preprocessing step

A3 a coarsened grid G_L has been computed, by definition every node in the core is either in G_L or is adjacent to a node in G_L. For simplicity of presentation we assume that the shape of the core is such that G_L is connected;

A4 particles know whether they belong to the supply $I \setminus T$, the demand $I \cap T$, or to the core $I \cap T$;

A5 every connected component C of the supply has a representative particle in the core $I \cap T$ adjacent to C, the *supply root* of C; correspondingly, every connected component D of the demand has one designated particle from the core adjacent to D, the *demand root* of D;

A6 each demand root d stores a complete spanning tree rooted at d of the corresponding demand component D in its memory explicitly or implicitly, by encoding construction rules for D.

If the core does not naturally induce a connected G_L, we can restore connectivity by locally deviating from the grid lines of the coarse grid and adapting the construction and use of G_L accordingly. Assumptions A2 and A4–A6 allow us to focus our presentation on the reconfiguration algorithm itself. Encoding a general target shape into the initial structure remains a challenging open problem. However, compared to prior work, our assumptions are not very strong: all other existing algorithms support only fixed, simple target shapes which can be easily encoded in the initial shape. For example, Derakhshandeh et al. [14] assume that the initial shape I is a large triangle and that the target shape T consists of only a constant number of unit triangles. Here a leader particle can easily compute the information necessary for the pre-processing step and broadcast it through the system.

Contribution and organization. We present the first reconfiguration algorithm for programmable matter that does not use a canonical intermediate configuration. Instead, our algorithm introduces new geometric primitives for amoebots which allow us to route particles in parallel from supply to demand via the core of the particle system. A fundamental building block of our approach are *feather trees* [21] which are a special type of *shortest path trees* (SP-trees). SP-trees arrange particles in a tree structure such that the paths from leaves to the root are shortest paths through the particle structure. The unique structure of feather trees allows us to use multiple overlapping trees in the particle system to enable particle navigation along shortest paths between multiple sources of supply and demand. In Section 2 we give all necessary definitions and show how to efficiently construct SP-trees and feather trees using a grid variant of shortest path maps [23].

In Section 3 we then show how to use feather trees to construct a *supply graph*, which directs the movement of particles from supply to demand. In Section 4 we describe in detail how to navigate the supply graph and also discuss the coarse grid G_L which we use to ensure the proper crossing of different flows of particles travelling in the supply graph. Finally, in Section 5 we summarize and analyze our complete algorithm for the reconfiguration problem. We show that using a sequential scheduler we can solve the particle reconfiguration problem in $O(n)$ activation rounds. When using an asynchronous scheduler our algorithm takes $O(n)$ rounds in expectation, and $O(n \log n)$ rounds with high probability. All omitted proofs and missing details can be found in the full version of our paper [22].

In the worst case our algorithm is as fast as existing algorithms, but in practice our method exploits the geometry of the symmetric difference between input and output shape: it minimizes unnecessary disassembly and reassembly of the particle system. Furthermore, if the configuration of the particle system is such that the feather trees are balanced with respect to the amount of supply particles in each sub-tree, then our algorithm finishes in a number of rounds close to the diameter of the system (instead of the number of particles). Our reconfiguration algorithm also moves the particles over as many parallel shortest paths as the problem instance allows. In Section 6 we discuss these features in more detail and also sketch future work.

2 Shortest path trees

To solve the particle reconfiguration problem, we need to coordinate the movement of the particles from $I \setminus T$ to $T \setminus I$. Among the previously proposed primitives for amoebot coordination is the *spanning forest primitive* [15] which organizes the particles into trees to facilitate movement while preserving connectivity. The root of a tree initiates the movement, and the remaining particles follow via handovers between parents and children. However, the

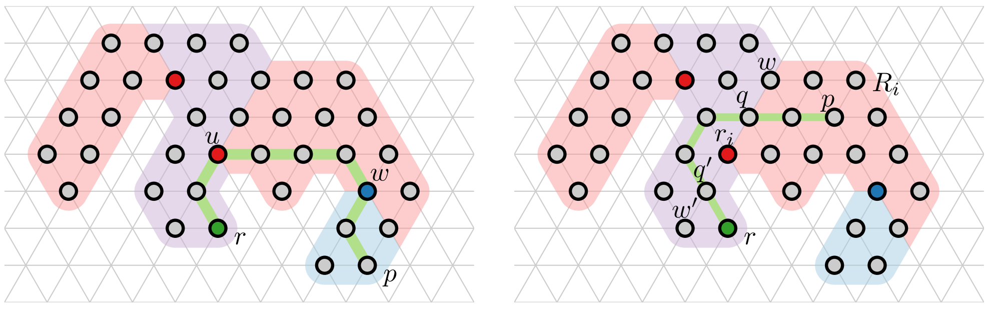

Figure 3 Shortest path map of node r. Any shortest path between r and p must pass through the roots of the respective SPM regions (u, w, and r_i). The region R_0 (in purple) consists of the particles $\mathcal{P}$-visible to r. The red and the blue particles are the roots of the corresponding SPM regions. Right: any path not going through the root of a visibility region can be shortened.

spanning forest primitive does not impose any additional structure on the resulting spanning trees. We propose to use a special kind of shortest path trees (SP-trees), called *feather trees*, which were briefly introduced in [21].

To ensure that all paths in the tree are shortest, we need to control the growth of the tree. One way to do so, is to use breadth-first search together with a token passing scheme, which ensures synchronization between growing layers of the tree, this is also known as a β-synchronizer [2]. See the full version of our paper for details [22].

▶ **Lemma 1.** *Given a connected particle configuration $\mathcal{P}$ with n particles, we can create an SP-tree using at most $O(n^2)$ rounds.*

2.1 Efficient SP-trees

To create SP-trees more efficiently for simply connected particle systems, we describe a version of the shortest path map (SPM) data structure [23] on the grid (see Figure 3 (left)). We say that a particle $q \in \mathcal{P}$ is *$\mathcal{P}$-visible* from a particle $p \in \mathcal{P}$ if there exists a shortest path from p to q in G that is contained in $G_{\mathcal{P}}$. This definition of visibility is closely related to staircase visibility in rectilinear polygons [6, 19]. Let $\mathcal{P}$ be simply connected, and let $R_0 \subseteq \mathcal{P}$ be the subconfiguration of all particles $\mathcal{P}$-visible from some particle r. By analogy with the geometric SPM, we refer to R_0 as a *region*. If $R_0 = \mathcal{P}$ then SPM(r) is simply R_0. Otherwise, consider the connected components $\{\mathcal{R}_1, \mathcal{R}_2, \dots\}$ of $\mathcal{P} \setminus R_0$. The *window* of $\mathcal{R}_i$ is a maximal straight-line chain of particles in R_0, each of which is adjacent to a particle in $\mathcal{R}_i$ (e.g., in Figure 3 (right), chain (r_i, w) is a window). Denote by r_i the closest particle to r of the window W_i of $\mathcal{R}_i$. Then SPM(r) is recursively defined as the union of R_0 and SPM(r_i) in $\mathcal{R}_i \cup W_i$ for all i. Let $R_i \subseteq \mathcal{R}_i \cup W_i$ be the set of particles $\mathcal{P}$-visible from r_i. We call R_i the *visibility region* of r_i, and r_i the *root* of R_i. Note that by our definition the particles of a window between two adjacent regions of a shortest path map belong to both regions.

▶ **Lemma 2.** *Let r_i be the root of a visibility region R_i in a particle configuration $\mathcal{P}$. For any particle p in R_i, the shortest path from r to p in $\mathcal{P}$ passes through r_i.*

▶ **Corollary 3.** *Any shortest path π between r and any other particle p in $\mathcal{P}$ must pass through the roots of the SPM regions that π crosses.*

If a particle p is $\mathcal{P}$-visible from r then there is a 60°-*angle monotone path* [11] from r to p in $G_{\mathcal{P}}$. That is, there exists a 60°-cone in a fixed orientation, such that for each particle q on the path from q to p lies completely inside this cone translated to q (see Figure 4 (left)).

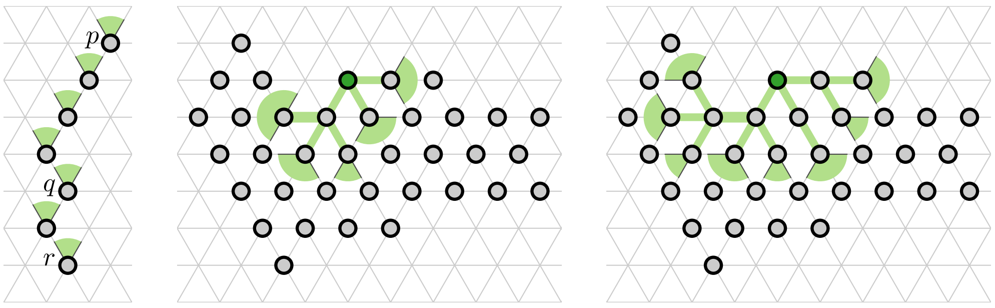

Figure 4 Left: An angle monotone path from r to p. For every particle q, the remainder of the path lies in a 60°-cone. Middle: Growing an SP-tree using cones of directions. The particle on the left just extended its cone to 180°. Right: A couple activations later.

We use a version of such cones to grow an SP-tree efficiently. Each node that is already included in the tree carries a cone of valid growth directions (see Figure 4 (middle)). When a leaf of the tree is activated it includes any neighbors into the tree which are not part of the tree yet and lie within the cone. A cone is defined as an interval of ports. The cone of the root r contains all six ports. When a new particle q is included in the tree, then its parent p assigns a particular cone of directions to q. Assume parent p has cone c and that q is connected to p via port i of p. By definition $i \in c$, since otherwise p would not include q into the tree. We intersect c with the 120°-cone $[i-1, i+1]$ and pass the resulting cone c' on to q. (Recall that the arithmetic operations on the ports are performed modulo 6.) When doing so we translate c' into the local coordinate system of q such that the cone always includes the same global directions. This simple rule for cone assignments grows an SP-tree in the visibility region of the root r and it does so in a linear number of rounds.

▶ **Lemma 4.** *Given a particle configuration $\mathcal{P}$ with diameter d which is $\mathcal{P}$-visible from a particle $r \in \mathcal{P}$, we can grow an SP-tree in $\mathcal{P}$ from r using $O(d)$ rounds.*

We now extend this solution to arbitrary simply-connected particle systems using the shortest-path map SPM(r). The SP-tree constructed by the algorithm above contains exactly the particles of the visibility region R_0 of SPM(r). As any shortest path from a window particle to r passes through the root of that window, any window incident to R_0 forms a single branch of the SP-tree. To continue the growth of the tree in the remainder of $\mathcal{P}$, we extend the cone of valid directions for the root particles of the regions of SPM(r) by 120°. A particle p can detect whether it is a root of an SPM(r)-region by checking its local neighborhood. Specifically, let the parent of p lie in the direction of the port $i+3$. If (1) the cone assigned to p by its parent is $[i-1, i]$ (or $[i, i+1]$), (2) the neighboring node of p in the direction $i+2$ (or $i-2$) is empty, and (3) the node in the direction $i+1$ (or $i-1$) is not empty, then p is the root of an SPM(r)-region, and thus p extends its cone to $[i-1, i+2]$ (or $[i-2, i+1]$) (see Figure 4 (middle)). Note that an extended cone becomes a 180°-cone.

▶ **Lemma 5.** *Let* SPM(r) *be the shortest-path map of a particle r in a simply-connected particle system $\mathcal{P}$. A particle $u \in \mathcal{P}$ extends its cone during the construction of an SP-tree if and only if it is the root of a region in* SPM(r).

Lemmas 4 and 5 together imply Theorem 6.

▶ **Theorem 6.** *Given a simply-connected particle configuration $\mathcal{P}$ with diameter d and a particle $r \in \mathcal{P}$ we can grow an SP-tree in $\mathcal{P}$ from r using $O(d)$ rounds.*

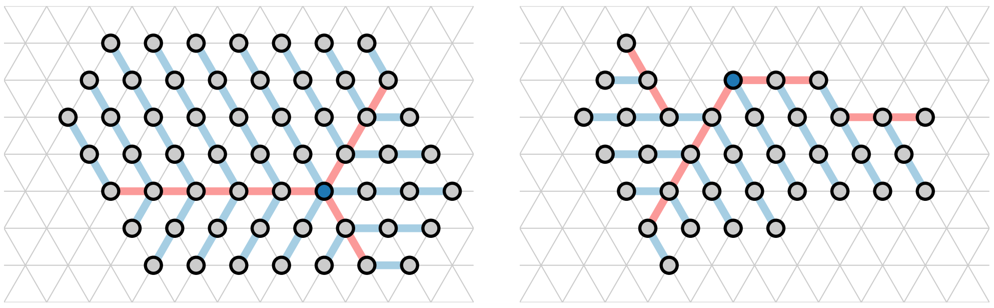

Figure 5 Two feather trees growing from the dark blue root. Shafts are red and branches are blue. Left: every particle is reachable by the initial feathers; Right: additional feathers are necessary.

2.2 Feather trees

These SP-trees, although efficient in construction, are not unique: the exact shape of the tree depends on the activation sequence of its particles. Our approach to the reconfiguration problem is to construct multiple overlapping trees which the particles use to navigate across the structure. As the memory capacity of the particles is restricted, they cannot distinguish between multiple SP-trees by using ids. Thus we need SP-trees that are unique and have a more restricted shape, so that the particles can distinguish between them by using their geometric properties. In this section we hence introduce *feather trees* which are a special case of SP-trees that use narrower cones during the growth process. As a result, feather trees bifurcate less and have straighter branches.

Feather trees follow the same construction rules as efficient SP-trees, but with a slightly different specification of cones. We distinguish between particles on *shafts* (emanating from the root or other specific nodes) and *branches* (see Figure 5 (left)). The root r chooses a maximal independent set of neighbors N_{ind}; it contains at most three particles and there are at most two ways to choose. The particles in N_{ind} receive a standard cone with three directions (a 3-cone), and form the bases of shafts emanating from r. All other neighbors of r receive a cone with a single direction (a 1-cone), and form the bases of branches emanating from r. For a neighbor p across the port i, p receives the cone $[i-1, i+1]$, translated to the coordinate system of p, if p is in N_{ind}, and the cone $[i]$ otherwise. The shaft particles propagate the 3-cone straight, and 1-cones into the other two directions, thus starting new branches. Hence all particles (except for, possibly, the root) have either a 3-cone or a 1-cone. The particles with 3-cones lie on shafts and the particles with 1-cones lie on branches.

We extend the construction of the tree around reflex vertices on the boundary of $\mathcal{P}$ in a similar manner as before. If a branch particle p receives a 1-cone from some direction $i+3$, and the direction $i+2$ (or $i-2$) does not contain a particle while the direction $i+1$ (or $i-1$) does, then p initiates a growth of a new shaft in the direction $i+1$ (or $i-1$) by sending there a corresponding 3-cone (see Figure 5 (right)).

Feather trees are a more restricted version of SP-trees (Theorem 6). For every feather tree, there exists an activation order of the particles such that the SP-tree algorithm would create this specific feather tree. This leads us to the following lemma.

▶ **Lemma 7.** *Given a simply connected particle configuration $\mathcal{P}$ with diameter d and a particle $r \in \mathcal{P}$, we can grow a feather tree from r in $O(d)$ rounds.*

Every particle is reached by a feather tree exactly once, from one particular direction. Hence a feather tree is independent of the activation sequence of the particles. In the following we describe how to navigate a set of overlapping feather trees. To do so, we first identify a useful property of shortest paths in feather trees.

We say that a vertex v of $G_{\mathcal{P}}$ is an *inner vertex*, if v and its six neighbors lie in the core $I \cap T$. All other vertices of the core are *boundary vertices.* A *bend* in a path is formed by three consecutive vertices that form a 120° angle. We say that a bend is an *boundary bend* if all three of its vertices are boundary vertices; otherwise the bend is an *inner bend*.

▶ **Definition 8** (Feather Path). *A path in $G_{\mathcal{P}}$ is a* feather path *if it does not contain two consecutive inner bends.*

We argue that every path π from the root to a leaf in a feather tree is a feather path. This follows from the fact that inner bends can occur only on shafts, and π must alternate visiting shafts and branches.

▶ **Lemma 9.** *A path between a particle s and a particle t is a feather path if and only if it lies on a feather tree rooted at s.*

Navigating feather trees. Consider a directed graph composed of multiple overlapping feather trees with edges pointing from roots to leaves. Due to its limited memory, a particle cannot store the identity of the tree it is currently traversing. Despite that, particles can navigate down the graph towards the leaf of some feather tree and remain on the correct feather tree simply by counting the number of inner bends and making sure that the particle stays on a feather path (Lemma 9). The number of inner bends can either be zero or one, and thus storing this information does not violate the assumption of constant memory per particle. Thus, when starting at the root of a feather tree, a particle p always reaches a leaf of that same tree. In particular, it is always a valid choice for p to continue straight ahead (if feasible). A left or right 120° turn is a valid choice if it is a boundary bend, or if the last bend p made was boundary.

When moving against the direction of the edges, up the graph towards the root of some tree, we cannot control which root of which feather tree a particle p reaches, but it still does so along a shortest path. In particular, if p's last turn was on an inner bend, then its only valid choice is to continue straight ahead. Otherwise, all three options (straight ahead or a 120° left or right turn) are valid.

3 Supply and demand

Each supply root organizes its supply component into an SP-tree; the supply particles will navigate through the supply roots into the core $I \cap T$ and towards the demand components along a *supply graph.* The supply graph, constructed in $I \cap T$, serves as a navigation network for the particles moving from the supply to the demand along shortest paths. Let $G_{I \cap T}$ be the subgraph of G induced on the nodes of $I \cap T$. We say a supply graph S is a subgraph of $G_{I \cap T}$ connecting every supply root s to every demand root d such that the following three *supply graph properties* hold:

1. for every pair (d, s) a shortest path from d to s in S is also a shortest path in $G_{I \cap T}$,
2. for every pair (d, s) there exists a shortest path from d to s in S that is a feather path,
3. every particle p in S lies on a shortest path for some pair (d, s).

We orient the edges of S from demand to supply, possibly creating parallel edges oriented in opposite directions. For a directed edge from u to v in S, we say that u is the predecessor of v, and v is the successor of u.

To create the supply graph satisfying the above properties, we use feather trees rooted at demand roots. Every demand root initiates the growth of its feather tree. When a feather tree reaches a supply root, a *supply found token* is sent back to the root of the tree. Note that if several feather trees overlap, a particle p in charge of forwarding the token up the tree cannot always determine which specific direction the corresponding root of the tree is. It thus sends a copy of the token to all *valid* parents (predecessors of p on every possible feather path of the token), and so the token eventually reaches all demand roots reachable by a feather path from the node it was created in. To detect if a token has already made an inner bend, the supply found token carries a flag β that is set once the token makes an inner bend. Specifically, a particle p that receives a supply found token t does the following:

1. p marks itself as part of the supply graph S, and adds the direction i that t came from as a valid successor in S,
2. from the set of all its predecessors (for all incoming edges), p computes the set U of valid predecessors. When p is not a reflex vertex, $U = [i + 3]$ if $\beta = 1$, and $U = [i + 2, i + 4]$ if $\beta = 0$. For reflex vertices, β is reset to 0 and more directions might be valid. Particle p then adds U to the set of its valid predecessors in S,
3. p sends copies of t (with an updated value of β) to the particles in U, and
4. p stores t in its own memory.

Each particle p, from each direction i, stores at most one token with flag β set to 1 and one with β set to 0. Hence p can store the corresponding information in its memory. If a particle p already belongs to the supply graph S when it is reached by a feather tree F, then p checks if its predecessor q in F is a valid parent for any token t stored in p, and, if so, adds q to the set of its valid predecessors in S (as in step (2)). For each of these tokens, but for at most one for each values of β, p sends a copy of t to q (as in step (3)).

▶ **Lemma 10.** *Given a simply-connected particle configuration $\mathcal{P}$ with diameter d, a set of particles marked as supply roots, and a set of particles marked as demand roots, a supply graph can be constructed in $O(d)$ rounds.*

Bubbles. Particles move from supply to demand. However, for ease of presentation and analysis, we introduce the abstract concept of demand *bubbles* that move from demand to supply, in the direction of edges of S, see Figure 6. Let us assume for now that the supply graph S has been constructed (in fact, its construction can occur in parallel with the reconfiguration process described below). Starting with a corresponding demand root d, each demand component D is constructed by particles flowing from the core $I \cap T$, according to the spanning tree of D stored in d. Every time a leaf particle expands in D, it creates a *bubble* of demand that needs to travel through d down S to the supply, where it can be

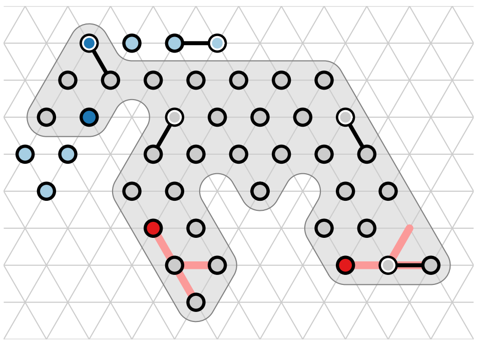

Figure 6 A reconfiguration process with five expanded particles holding bubbles (outlined in white). Supply particles are blue, supply roots dark blue. Demand roots (red) store spanning trees of their demand components. The supply graph is not shown.

resolved. Bubbles move via a series of handovers along shortest paths in S. An expanded particle p holding a bubble b stores two values associated with it. The first value β is the number of inner bends b took since the last boundary bend ($\beta \in \{0, 1\}$), and is used to route the bubbles in S. The second value δ stores the general direction of b's movement; $\delta = \textsc{s}$ if b is moving forward to supply, and $\delta = \textsc{d}$ if b is moving backwards to demand.

If a particle p holding a bubble b wants to move b to a neighboring particle q, p can only do so if q is contracted. Then, p initiates a pull operation, and thereby transfers b and its corresponding values to q. Thus the particles are pulled in the direction of a demand root, but the bubbles travel along S from a demand root towards the supply.

A supply component may become empty before all bubbles moving towards it are resolved. In this case, the particles of S have to move the bubbles back up the graph. Particles do not have sufficient memory to store which specific demand root bubbles came from. However, because of the first supply graph property, every demand root has a connection to the remaining supply. While a bubble is moved back up along S, as soon as there is a different path towards some other supply root, it is moved into that path. Then, the edges connecting to the now empty supply are deleted from S. Moreover, other edges that now point to empty supply or to deleted edges are themselves deleted from S. As the initial and target shapes have the same size, the total number of bubbles equals the number of supply particles. Therefore, once all bubbles are resolved, the reconfiguration problem is solved.

In the remainder of the paper we may say "a bubble activates" or "a bubble moves". By this we imply that "a particle holding a bubble activates" or "a particle holding a bubble moves the bubble to a neighboring particle by activating a pull handover".

4 Navigating the supply graph

When the demand and supply roots are connected with the supply graph S, as described in Section 3, the reconfiguration process begins. Once a demand root d has received a supply found token from at least one successor (d is added to S), it begins to construct its demand component D along the spanning tree $\mathcal{T}_D$ that d stores, and starts sending demand bubbles into S. The leafs of the partially built spanning tree $\mathcal{T}_D$ carry the information about their respective sub-trees yet to be built, and pull the chains of particles from d to fill in those sub-trees, thus generating bubbles that travel through d into S.

With each node v of S, for every combination of the direction i to a predecessor of v and a value of β, we associate a value $\lambda(i, \beta) \in \{\text{true}, \text{false}\}$, which encodes the liveliness of feather paths with the corresponding value of β from the direction i through v to some supply. If $\lambda(i, \beta) = \text{false}$ then, for a given value of β, there are no feather paths through v to non-empty supply in S. Note that here we specifically consider nodes of S, and not the particles occupying them. When particles travel through S, they maintain the values of λ associated with the corresponding nodes. Initially, when S is being constructed, $\lambda = \text{true}$ for all nodes, all directions, and all β. When a bubble b travels down S to a supply component that turns out to be empty, b reverses its direction. Then, for all the nodes that b visits while reversing, the corresponding value λ is set to false, thus marking the path as dead.

For an expanded particle p occupying two adjacent nodes of S, denote the predecessor node as v_a, and the successor node as v_b. By our convention, we say the bubble in p occupies v_b. When particle p with a bubble b activates, it performs one of the following operations. It checks them in order and performs the first action available.

1. If $\delta = \textsc{s}$ (b is moving to supply) and v_b is inside a supply component, then p pulls on any contracted child in the spanning tree of the supply; if v_b is a leaf, then p simply contracts into v_a, thus resolving the bubble.
2. If $\delta = \textsc{s}$ and $v_b \in S$, p checks which of the successors of v_b in S lie on a feather path for b and are alive (i.e., their corresponding $\lambda =$ true). If there is such a successor q that is contracted, p pulls q and sends it the corresponding values of β and δ (while updating β if needed), i.e., p transfers b to q. Thus the bubble moves down S to supply.
3. If $\delta = \textsc{s}$, $v_b \in S$, and v_b does not have alive successors in S that lie on a feather path for b, then p reverses the direction of b to $\delta = \textsc{d}$, and sets the value $\lambda(i, \beta)$ of v_b to false, where i is the direction from v_b to v_a. The bubble does not move.
4. If $\delta = \textsc{d}$ (b is moving to demand) and there exists an alive successor node of either v_b or v_a that lies on a feather path for b and is occupied by a contracted particle q, then p switches the direction of b to $\delta = \textsc{s}$, pulls on q, and transfers to q the bubble b with its corresponding values (while updating β if needed). The bubble changes direction and moves onto a feather path which is alive.
5. If $\delta = \textsc{d}$ and none of the successors of v_b or v_a in S are alive for b, then p sets the corresponding values λ of v_b and v_a to false, and checks which predecessors of v_a lie on a feather path for b (note, this set is non-empty). If there is such a predecessor q that is contracted, then p pulls q and transfers it the bubble b. Thus the bubble moves up in S.

Coarse grid. Particles can only make progress if they have a contracted successor. To ensure that flows of particles along different feather paths can cross without interference, we introduce a coarsened grid, and devise a special crossing procedure.

Rather than constructing supply graph S on the triangular grid G, we now do so using a grid G_L that is coarsened by a factor of three and is laid over the core $I \cap T$ (see Figure 2). Then, among the nodes of G we distinguish between those that are also *grid nodes* of G_L, *edge nodes* of G_L, and those that are neither. By our assumptions on the input, the graph G_L is connected. Note, that then, every node of the particle system is either a part of G_L (is a grid or an edge node), or is adjacent to a node of G_L. To ensure that all particles agree on the location of G_L, we assume that we are given a leader particle ℓ in the core $I \cap T$, that initiates the construction of G_L.

In between two grid nodes of G_L, there are exactly two edge nodes of G_L. While bubbles are in the core $I \cap T$, we let them occupy only edge nodes of G_L. To resolve crossing paths in S, every time a bubble wants to cross a grid node of G_L occupied by a particle p, the bubble sends a request to p. Then if p receives multiple of these requests, it decides which of the bubbles can cross. To avoid potential head-on collisions of bubbles on a single edge of G_L corresponding to a bidirectional edge of S, our algorithm temporarily disallows the movement of bubbles in one of the two directions. Only after the chosen direction is marked as dead, the other direction becomes available for bubble traversal. That is, at any moment in time, the supply graph, does not contain bidirectional edges. See Appendix B for details.

5 Algorithm

To summarize, our approach consists of three phases. In the first phase, the leader particle initiates the construction of the coarse grid G_L over the core $I \cap T$. In the second phase, the particles grow feather trees, starting from the demand roots. If a feather tree reaches supply, that information is sent back up the tree and the particles form the supply graph S. In the last phase, particles move from supply to demand along S. Note that, for the particle system as a whole, these phases may overlap in time. For example, the reconfiguration process may begin before the supply graph is fully constructed. Each individual particle can move on to executing the next phase of the algorithm once the previous phase for it is finished.

For the purposes of analysis, we view the reconfiguration as bubbles of demand traveling along S from demand to supply. Bubbles turn around on dead paths where all supply has been consumed. To ensure proper crossing of different bubble flows, we let the grid nodes of S to act as traffic conductors, letting some bubbles cross while others wait for their turn.

Correctness. For simplicity of presentation, we first show correctness of the algorithm under a sequential scheduler. We then extend the algorithm and its analysis to the case of an asynchronous scheduler. To show correctness, we need to show two properties, *safety* and *liveness*. The algorithm is safe if $\mathcal{P}$ never enters an invalid state, and is live if in any valid state there exists a particle that, when activated, can make progress towards the goal. Lemma 10 proves the correctness of the phase of the construction of the supply graph. As the particles start executing the reconfiguration phase only after the construction of the supply graph is finished for them locally, for the purposes of proving the correctness of the reconfiguration phase, we may assume that the supply graph has been constructed in the particle system as a whole. A state of $\mathcal{P}$ is valid when it satisfies the following properties:

- Particle configuration $\mathcal{P}$ is connected.
- It holds that $\#b + \#d = \#s$, where $\#b$, $\#d$, and $\#s$ are the number of bubbles, demand spots, and supply particles respectively. That is, the size of supply matches exactly the size of demand. This assumes that initial and target shapes have the same size.
- There are no bidirectional edges in S allowed for traversal.
- For every pair of demand root d and supply root s with a non-empty supply, there exists a feather path from d to s in S; furthermore, all such feather paths are alive (i.e., the corresponding values of λ are set to true).
- Any expanded particle has both nodes on a single feather path of S.
- Any expanded particle on an alive feather path moves to supply ($\delta = \mathrm{s}$).
- Any node of S with $\lambda(i, \beta) =$ true, for some combination of direction i and value β, is connected by an alive feather path to some demand root d.

For the property of liveness, we need to show that progress can always be made. We say the particle system makes progress whenever (1) a bubble moves on its path (this includes resolving the bubble with supply, or when a demand root creates a new bubble), and (2) a bubble changes its value for direction δ. In the full version of our paper [22], we show by induction that during the execution of our algorithm, the configuration stays valid at any point in time, and that at any point in time, there is a bubble that can make progress.

▶ **Lemma 11.** *A particle configuration $\mathcal{P}$ stays valid and live at all times.*

Running time analysis. We begin by arguing that the total distance traveled by each bubble is linear in the number of particles in the system. We first limit the length of the path each bubble takes.

▶ **Lemma 12.** *The total path of a bubble b in a particle configuration $\mathcal{P}$ with n particles has size $O(n)$.*

Next, we give a lower bound on the number of rounds it takes each bubble to traverse its path. We analyze the progress made by the particle system by creating a specific series of $O(n)$ configurations that represent a lower bound on the overall progress; we show that any activation sequence chosen by an adversarial scheduler results in at least as much progress. This approach has been previously used for analyzing the running time of a moving line of particles [14], and for a tree moving from the root [7]. Together with the proof of correctness, we conclude with the following theorem:

▶ **Theorem 13.** *The particle reconfiguration problem for particle configuration $\mathcal{P}$ with n particles can be solved using $O(n)$ rounds of activation under a sequential scheduler.*

Extending the analysis to an asynchronous scheduler. Unlike in the case of a sequential scheduler, when at each moment only one particle can be active, under an asynchronous scheduler multiple particles may be active at the same time. This may lead to concurrency issues. Specifically, when actions of two or more particles conflict with each other, not all of them can be finished successfully. To extend our algorithm to the case of an asynchronous scheduler, we explore what kind of conflicts may arise, and ensure that our algorithm can deal with them. There can be three types of conflicts:

1. Two particles try to expand into the same empty node.
2. Two contracted particles try to push the same expanded particle.
3. Two expanded particles try to pull on the same contracted particle.

The first two kinds of conflicts never arise in our algorithm, as (1) every empty node (demand spot) can be moved into from only a single direction, according to the spanning tree stored by the corresponding demand root, and (2) no particle ever performs a push operation. The third conflict could potentially arise in our approach when two bubbles traveling on crossing paths pass the junction node. However, as crossing of a junction is controlled by the junction particle, only one particle at a time is given permission to pull on the junction node. Thus, under an asynchronous scheduler, all actions initiated by the particles always succeed.

We are left to analyze whether the asynchronous setting can lead to deadlocks in our algorithm, where concurrency issues result in some particle not being able to make progress. As mentioned above, crossings of junctions are controlled by the junction particles, and thus cannot lead to deadlocks. Furthermore, the algorithm forbids the simultaneous existence of bidirectional edges in S, thus there cannot be deadlocked bubbles moving in the opposite directions over the same edge. The only remaining case of a potential deadlock is when the algorithm temporarily blocks one of the two directions of a bidirectional edge. In a sequential schedule, this choice is made by one of the two edge nodes of G_L corresponding to the bidirectional edge of S. However, in an asynchronous schedule, both edge nodes may become active simultaneously, and choose the opposite directions of S. To resolve this case, and to break the symmetry of two nodes activating simultaneously and choosing the opposite directions, we need to utilize the power of randomness. We assume that the particles have access to a constant number of random bits. Daymude et al. [9] show that, in that case, a mechanism exists that allows the particles to lock their local neighborhood, and to perform their actions as if the neighboring particles were inactive. Such a locking mechanism increases the running time only by a constant factor in expectation, and by a logarithmic factor if the particle needs to succeed with high probability. Thus, with a locking mechanism, we can ensure that our algorithm can select one of the two directions of a bidirectional edge in S.

▶ **Theorem 14.** *The particle reconfiguration problem for particle configuration $\mathcal{P}$ with n particles can be solved in $O(n)$ rounds of activation in expectation (or in $O(n \log n)$ rounds of activation with high probability) under an asynchronous scheduler, if each particle has access to a constant number of random bits.*

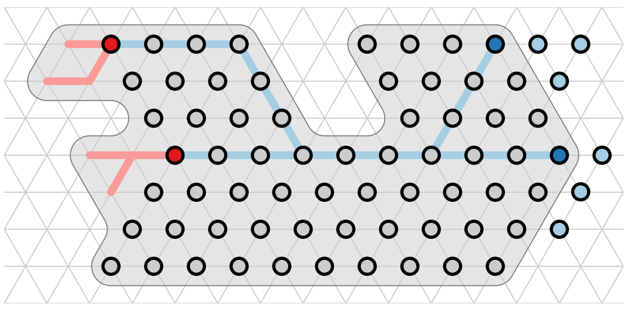

Figure 7 Shortest paths lead through a bottleneck, slowing down the algorithm in practice.

The running time of our algorithm is worst-case optimal: any algorithm to solve the particle reconfiguration problem needs at least a linear number of rounds.

▶ **Theorem 15.** *Any algorithm that successfully solves the particle reconfiguration problem needs $\Omega(n)$ rounds.*

6 Discussion

We presented the first reconfiguration algorithm for programmable matter that does not use a canonical intermediate configuration. In the worst-case, our algorithm requires a linear number of activation rounds and hence is as fast as existing algorithms. However, in practice, our algorithm can exploit the geometry of the symmetric difference between input and output and can create as many parallel shortest paths as the problem instance allows.

We implemented our algorithm in the Amoebot Simulator[1]. In the following screenshots and the accompanying videos of complete reconfiguration sequences[2] supply particles are colored green, demand roots red, and supply roots cyan. The dark blue particles are part of the supply graph and therefore lie on a feather path from a demand root to a supply root.

Figure 8 illustrates that our algorithm does indeed create a supply graph which is based on the shortest paths between supply and demand and hence facilitates parallel movement paths if the geometry allows. Activation sequences are randomized, so it is challenging to prove statements that capture which supply feeds which demand, but generally we observe that close supply and demand nodes will connect first. An interesting open question in this context is illustrated in Figure 7: here we see two supplies and two demands, but the shortest paths have a common bottleneck, which slows down the reconfiguration in practice. Is there an effective way to include near-shortest paths in our supply graph to maximize parallelism? One could also consider temporarily adding particles to the symmetric difference. Note, though, that additional parallelism leads only to constant factor improvements in the running time; this is of course still meaningful in practice.

As we already hinted in the introduction, the running time of our algorithm is linked to the balance of supply in the feather trees. Recall that the trees are rooted at demand and grow towards supply; bubbles move from demand to supply. Because bubbles decide randomly at each junction which path to follow, each sub-tree will in expectation receive a similar amount of bubbles. If all sub-trees of a junction contain an equal amount of supply particles, then we say this junction is *balanced*. If all junctions are balanced, then we say a feather tree is balanced; similarly we speak about a balanced supply graph.

[1] `https://github.com/SOPSLab/AmoebotSim`
[2] `https://github.com/PetersTom/AmoebotVideos`

The worst case running time of $O(n)$ rounds is triggered by unbalanced supply graphs. See, for example, Figure 9 (right), where at each junction the sub-tree rooted in the center carries all remaining supply, except for two particles. Such situations arise when there are many small patches of supply and only a few locations with demand. In the realistic scenario of shape repair we have exactly the opposite situation with many small damages (demand) and one large reservoir of supply stored for repairs. Here the supply graph will naturally be balanced and the running time is proportional to the diameter of the core $I \cup T$. Figure 9 (left) shows an example with only one supply and demand; here the running time of our algorithm is even proportional to the distance between supply and demand within $I \cup T$.

Feather trees are created to facilitate particles traveling on (crossing) shortest paths between supply and demand; they do not take the balance of the supply graph into account. A challenging open question in this context is whether it is possible to create supply graphs which retain the navigation properties afforded by feather tress but are at the same time balanced with respect to the supply.

As a final example, Figure 10 gives an impression of the natural appearance of the reconfiguration sequences produced by our algorithm. In the remainder of this section we discuss ways to lift A1: the core $I \cap T$ is non-empty and simply connected. If the core consist of more than one component, then we can choose one component as the core for the reconstruction. The other components can then be interpreted by the algorithm as both supply and demand. Consequently they will first be deconstructed and then reassembled. Clearly this is not an ideal solution, but it does not affect the asymptotic running time and does not require any major changes to our algorithm. If the core is not simply connected, that is, it contains holes, then our procedure for creating feather trees may no longer create shortest path trees or it might not even terminate. The slow $O(n^2)$ algorithm will however terminate and produce shortest path. A very interesting direction for future work are hence efficient algorithms for shortest path trees for shapes with holes.

References

1 Marta Andrés Arroyo, Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018. doi:10.1007/s11047-018-9714-x.

2 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

3 Sarah Cannon, Joshua J. Daymude, Cem Gökmen, Dana Randall, and Andréa W. Richa. A Local Stochastic Algorithm for Separation in Heterogeneous Self-Organizing Particle Systems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019)*, volume 145 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54:1–54:22, 2019. doi:10.4230/LIPIcs.APPROX-RANDOM.2019.54.

4 Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov Chain Algorithm for Compression in Self-Organizing Particle Systems. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 279–288, 2016. doi:10.1145/2933057.2933107.

5 Kenneth C. Cheung, Erik D. Demaine, Jonathan R. Bachrach, and Saul Griffith. Programmable Assembly With Universally Foldable Strings (Moteins). *IEEE Transactions on Robotics*, 27(4):718–729, 2011. doi:10.1109/TRO.2011.2132951.

6 Joseph C. Culberson and Robert A. Reckhow. Dent Diagrams: A Unified Approach to Polygon Covering Problems. Technical report, University of Alberta, 1987. TR 87–14.

7 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the Runtime of Universal Coating for Programmable Matter. *Natural Computing*, 17(1):81–96, 2016. doi:10.1007/s11047-017-9658-6.

8 Joshua J. Daymude, Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Christian Scheideler, and Andréa W. Richa. Convex Hull Formation for Programmable Matter. In *Proc. 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020. doi:10.1145/3369740.3372916.

9 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. Local Mutual Exclusion for Dynamic, Anonymous, Bounded Memory Message Passing Systems, November 2021. URL: http://arxiv.org/abs/2111.09449.

10 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. In *35th International Symposium on Distributed Computing (DISC)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:19, 2021. doi:10.4230/LIPIcs.DISC.2021.20.

11 Hooman R. Dehkordi, Fabrizio Frati, and Joachim Gudmundsson. Increasing-Chord Graphs On Point Sets. In *Proc. International Symposium on Graph Drawing (GD)*, LNCS 8871, pages 464–475, 2014. doi:10.1007/978-3-662-45803-7_39.

12 Erik D. Demaine, Jacob Hendricks, Meagan Olsen, Matthew J. Patitz, Trent A. Rogers, Nicolas Schabanel, Shinnosuke Seki, and Hadley Thomas. Know When to Fold 'Em: Self-assembly of Shapes by Folding in Oritatami. In *DNA Computing and Molecular Programming*, pages 19–36, 2018. doi:10.1007/978-3-030-00030-1_2.

13 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot—A New Model for Programmable Matter. In *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

14 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal Shape Formation for Programmable Matter. In *Proc. 28th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–299, 2016. doi:10.1145/2935764.2935784.

15 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader Election and Shape Formation with Self-organizing Programmable Matter. In *Proc. International Workshop on DNA-Based Computing (DNA)*, LNCS 9211, pages 117–132, 2015. doi:10.1007/978-3-319-21999-8_8.

16 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33:69–101, 2020. doi:10.1007/s00446-019-00350-6.

17 Fabien Dufoulon, Shay Kutten, and William K. Moses Jr. Efficient Deterministic Leader Election for Programmable Matter. In *Proc. 2021 ACM Symposium on Principles of Distributed Computing*, pages 103–113, 2021. doi:10.1145/3465084.3467900.

18 Cody Geary, Paul W. K. Rothemund, and Ebbe S. Andersen. A single-stranded architecture for cotranscriptional folding of RNA nanostructures. *Science*, 345(6198):799–804, 2014. doi:10.1126/science.1253920.

19 Subir Kumar Ghosh. *Visibility Algorithms in the Plane.* Cambridge University Press, 2007. doi:10.1017/CBO9780511543340.

20 Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming Tile Shapes with Simple Robots. In *Proc. International Conference on DNA Computing and Molecular Programming (DNA)*, pages 122–138, 2018. doi:10.1007/978-3-030-00030-1_8.

21 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Brief announcement: An effective geometric communication structure for programmable matter. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

22 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Fast reconfiguration for programmable matter. *arxiv*, August 2023. URL: `https://arxiv.org/abs/2202.11663`.

23 Joseph S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991. `doi:10.1007/BF01530888`.

24 Andre Naz, Benoit Piranda, Julien Bourgeois, and Seth Copen Goldstein. A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In *Proc. 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 254–263, 2016. `doi:10.1109/NCA.2016.7778628`.

25 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014. `doi:10.1007/s11047-013-9379-4`.

26 Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42(8):1619–1633, 2018. `doi:10.1007/s10514-018-9710-0`.

27 Alexandra Porter and Andrea Richa. Collaborative Computation in Self-organizing Particle Systems. In *Proc. International Conference on Unconventional Computation and Natural Computation (UCNC)*, LNCS 10867, pages 188–203, 2018. `doi:10.1007/978-3-319-92435-9_14`.

28 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proc. 4th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 353–354, 2013. `doi:10.1145/2422436.2422476`.

A Simulation figures

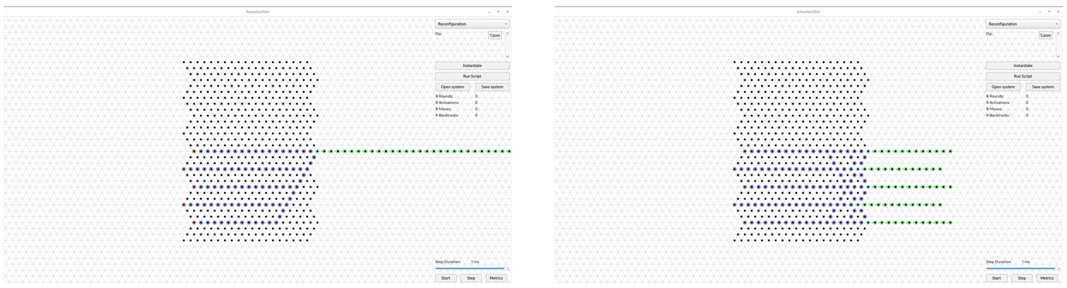

Figure 8 One supply and five demands vs. five supplies and five demands.

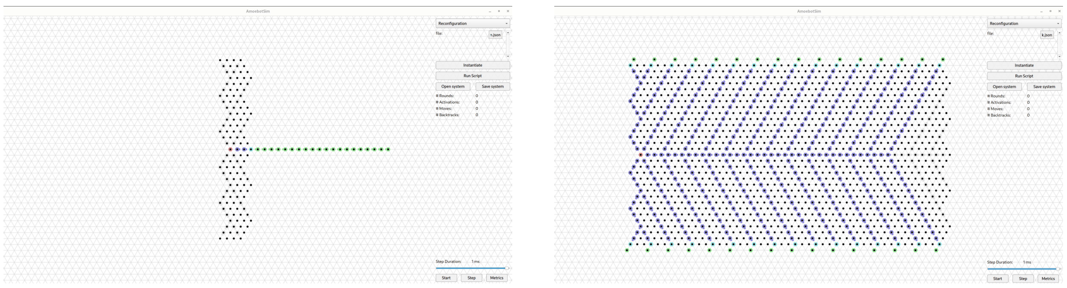

Figure 9 Left: Particles move directly between supply and demand. Right: Worst case configuration for back-tracking.

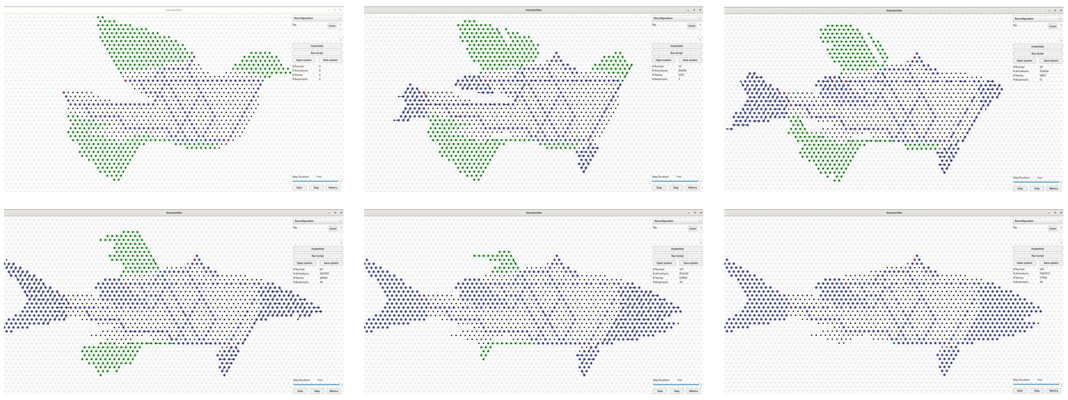

Figure 10 Dove to fish, full video at `https://github.com/PetersTom/AmoebotVideos`.

B Coarse grid

Particles can only make progress if they have a contracted successor. If there are crossing paths in S, these paths might interfere. To ensure that flows of particles along different feather paths can cross, we introduce a coarsened grid, and devise a special crossing procedure.

Rather than constructing supply graph S on the triangular grid G, we now do so using a grid G_L that is coarsened by a factor of three and is overlaid over the core $I \cap T$ (see Figure 2). Then, among the nodes of G we distinguish between those that are also *grid nodes* of G_L, *edge nodes* of G_L, and those that are neither. By our assumptions on the input, the graph G_L is connected. Note, that then, every node of the particle system is either a part of G_L (is a grid or an edge node), or is adjacent to a node of G_L. To ensure that all particles agree on the location of G_L, we assume that we are given a leader particle ℓ in the core $I \cap T$, that initiates the construction of G_L (see Figure 2 in Section 1).

Note that for two adjacent grid nodes v_1 and v_2 of G_L, the edge (v_1, v_2) does not have to be in G_L, if one or both corresponding edge nodes are not part of the core $I \cap T$. We say a grid node v in G_L is a boundary node if there exists a grid cell with corners v, v_1, v_2 where at least one of its edges is missing. Grid node v is a *direct boundary* node if this missing edge is incident to v, and an *indirect boundary* node if the missing edge is (v_1, v_2). All other grid nodes in G_L are inner nodes.

Now, feather trees only grow over the particles in G_L. Demand roots initiate the construction of their feather trees from a closest grid node in G_L. The supply roots organize their respective supply components in SP-trees as before (in the original grid), and connect to all adjacent particles in the supply graph S ($S \subseteq G_L$).

The growth of a feather tree in G_L is very similar to that in G. On direct boundary nodes, the cones propagate according to the same rules as before. On indirect boundary nodes, the creation of new shafts and branches is outlined in Figure 11. The resulting feather tree may now have angles of 60°. To still be able to navigate the supply graph, bubbles are allowed to make a 60° bend on indirect boundary nodes, resetting $\beta = 0$, only when both the start and end vertex of that bend are also boundary nodes.

A useful property of G_L that helps us ensure smooth crossings of the bubble flows is that there are at least two edge nodes in between any two grid nodes. We can now restrict the expanded particles carrying bubbles to mainly use the edge nodes of S, and only cross the grid nodes if there is enough room for them to fully cross.

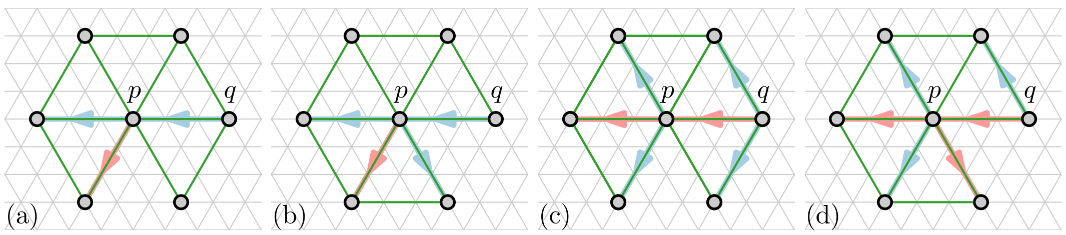

Figure 11 Construction of feather trees at indirect boundary nodes. G_L is shown in green, only the particles on nodes of G_L are shown. A branch ((a)–(b), in blue) and a shaft ((c)–(d), in red) of a feather tree grows from q to p. (a) A new shaft is emanated from p; (b) a new shaft and a new branch are emanated from p; (c) p behaves as an internal node; (d) a new shaft is emanated from p.

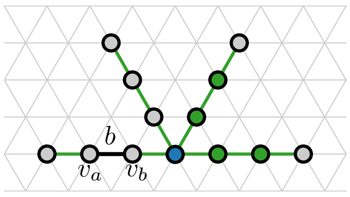

Figure 12 A bubble b on two edge nodes of G_L. Only particles on G_L are drawn. The blue particle is on $c(b)$, the nodes occupied by the two pairs of green particles together form $N(b)$.

With $p(x)$, we denote the particle occupying node x. For a directed edge from u to v in S, we say that u is the predecessor of v, and v is the successor of u, and denote this by $u \to v$. Consider a bubble b moving forward in S (with $\delta = \text{s}$), held by an expanded particle p occupying two edge nodes v_a and v_b, with $v_a \to v_b$ (see Figure 12). Let $c(b)$ be the grid node of S adjacent to v_b, such that $v_b \to c(b)$; $c(b)$ is the grid node that b needs to cross next. Let $\mathcal{D}(b)$ denote the set of valid directions for b in S from the position of $c(b)$. Let $N(b) = \{(r_1^i, r_2^i) \mid i \in \mathcal{D}(b)\}$ be the set of pairs of edge nodes lying in these valid directions for b. Specifically, for each $i \in \mathcal{D}(b)$, let $c(b) \to r_1^i$ in the direction i, and $r_1^i \to r_2^i$.

Crossings in G_L. The grid nodes of G_L that are part of the supply graph act as traffic conductors. We thus use terms grid nodes and junctions interchangeably. For bubble b, its particle p is only allowed to pull on the particle at $c(b)$, and thus initiate the crossing of $c(b)$, if there is a pair of nodes $(r_1, r_2) \in N(b)$ that are occupied by contracted particles. In this case, after at most three activation rounds, b will completely cross $c(b)$, the expanded particle now carrying it will occupy the edge nodes r_1 and r_2, and the junction $c(b)$ will be ready to send another bubble through itself. Assume for now that S has all edges oriented in one direction (there are no parallel edges in opposite directions). Below we discuss how to lift this assumption. The procedure followed by the junctions is the following. If an expanded particle p wants to pull on a particle at a junction, it first requests permission to do so by sending a *request* token containing the direction it wants to go after $c(b)$. Every junction node stores a queue of these requests. A request token arriving from the port i is only added to the queue if there are no requests from i in the queue yet. As every direction is stored only once, this queue is at most of size six. When particle $p(r_1)$ occupying an edge node r_1, with some grid node $c \to r_1$, activates, it checks if itself and the particle $p(r_2)$ at $r_1 \to r_2$ are contracted. If so, $p(r_1)$ sends an *availability* token to $p(c)$. Similarly, junction nodes need to store at most six availability tokens at once.

When the particle at a junction activates, and it is ready to transfer the next bubble, it grants the first pull request with a matching availability token by sending the acknowledgment token to the particle holding the corresponding bubble. The request and availability tokens are then consumed. Only particles with granted pull requests are allowed to pull and move their bubbles onto junctions. Junction queues are associated with the grid nodes of S, and not particles. Thus, if an expanded particle p occupying a grid node c pulls another particle q to c, the queues are transformed into the coordinate system of q and sent to q.

▶ Remark 16. Above, for simplicity of presentation, we assume that there are no bidirectional edges in S. These edges, however, can be treated as follows. During the construction of the supply graph one of the two opposite directions is chosen as a dominating one. For a corresponding edge of G_L, one of its two edge nodes that receives the supply found token first, reserves the direction of the corresponding feather path for this edge. If the other edge node eventually receives the supply found token from the opposite direction, it stores the information about this edge as being inactive. While the branch of the dominating direction is alive, the dominated branch is marked as unavailable. As soon as, and if, supply runs out for the dominating branch, and it is marked dead, the dominated branch is activated, and can now be used. This slows down the reconfiguration process by a number of rounds at most linear in the size of the dominating branch.