Fast Reconfiguration for Programmable Matter

Citation for published version (APA):
Kostitsyna, I., Peters, T., & Speckmann, B. (2022). Fast Reconfiguration for Programmable Matter. 365-371. Abstract from The 38th European Workshop on Computational Geometry, Perugia, Italy.

Document status and date:
Published: 14/03/2022

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Fast Reconfiguration for Programmable Matter

Irina Kostitsyna\textsuperscript{1}, Tom Peters\textsuperscript{1}, and Bettina Speckmann\textsuperscript{1}

\textsuperscript{1} TU Eindhoven, the Netherlands
\{i.kostitsyna|t.peters1|b.speckmann\}@tue.nl

Related Version A full version of the paper is available at arxiv.org/abs/2202.11663.

1 Introduction

The concept of programmable matter envisions a very large number of tiny and simple robot particles forming a smart material that can change its physical properties and shape based on the outcome of computation and movement performed by the individual particles in a concurrent manner. The ultimate goal is to have programmable matter that is indistinguishable from any other material. Thus, when modeling it, we assume a very small size of the particles and greatly restrict their computation, communication, and movement capabilities. 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 \cite{1,8,11,13,7,3,10,12}. Here we focus on the amoebot model which was introduced in \cite{4} and refined in \cite{2}. Refer to \cite{2} for additional details on the model description. For reconfiguration in the amoebot model, the approach taken by existing solutions is to build the target shape from scratch, ignoring any possible similarities between the initial shape and the target shape \cite{5,6}. However, in some scenarios (e.g. shape repair) where the initial and target shapes are similar, this might not be the most efficient strategy. We focus on an approach for reconfiguration that takes this similarity into account. In the worst case our algorithm works as well as the existing solutions, but it is natural to expect our approach to be more advantageous in the case where there are only small changes necessary in the system.

\textbf{Amoebot model.} Particles occupy nodes of a plane triangular grid $G$. A particle can occupy one (contracted particle) or two (expanded particle) adjacent nodes of the grid, and can communicate with its neighboring particles. The particles have constant memory space, and thus have limited computational power. They have no common notion of orientation, and no common notion of clockwise or counter-clockwise order. The particles are identical, i.e., they have no IDs and execute the same algorithm, but they can locally distinguish between neighbors using six (for contracted) or ten (for expanded particles) port identifiers (see Figure 1 (left)). Ports are labeled in order (either cw or ccw) modulo six or ten, respectively. Particles communicate by sending messages to the neighbors using the ports.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{Left: particles with ports labeled, in contracted and expanded state. Right: handover operation between two particles.}
\end{figure}

38th European Workshop on Computational Geometry, Perugia, Italy; March 14–16, 2022. 
This is an extended abstract of a presentation given at EuroCG'22. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.
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 (see Figure 1 (right)). The handover can be initiated by any of the two particles: an expanded particle can pull its contracted neighbor, and a contracted particle can push 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 \( t \), for any particle, it must be activated at some time in the future \( t' > t \)). If two particles are attempting at 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: the time intervals in which all particles have been activated at least once.

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

**Problem description.** An instance of the reconfiguration problem consists of a pair of simply connected shapes \((I,T)\) embedded in the grid \( G \). We assume that \( I \) and \( T \) have the same number of nodes, and that \( I \cap T \), which we call the core, is non-empty and simply connected. Initially, all particles in \( I \) are contracted. The problem is solved when every node of \( T \) contains a contracted particle.

We call the particles in \( I \setminus T \) the supply particles (see Figure 2), and assume that every connected component of \( I \setminus T \) has a designated particle in the core \( I \cap T \) adjacent to it, which we call the root of that component. Similarly, we say that \( T \setminus I \) are demand nodes. For every connected component \( D \) of \( T \setminus I \), we designate one particle from the core \( I \cap T \) adjacent to \( D \) as the demand root of \( D \). We assume that each demand root \( d \) stores a spanning tree of the corresponding component \( D \) in its memory. The root \( d \) will pull supply particles through the core \( I \cap T \) to fill \( D \).

![Figure 2](image-url) Initial shape \( I \) (formed by the particles), target shape \( T \) (gray), supply particles (blue), supply roots (dark blue). Demand roots (red) store a spanning tree of their demand component.
Contribution and organization. We propose a new approach for fast shape reconfiguration in the amoebot model, based on the symmetric difference between the initial and the target shapes. Our goal is to design a reconfiguration algorithm, for the case when the initial and target shapes are similar, that is faster and more natural than constructing the target shape from scratch. To this extent, we propose a new primitive: a special case of the shortest path tree (SP-tree) which we call a feather tree. We use feather trees to construct a graph, which the particles can use to move along shortest paths and reconfigure the particle system.

2 Feather trees

To solve the particle reconfiguration problem, we need to coordinate the movement of the particles. Among the previously proposed primitives for amoebot coordination is the shortest path tree (SP-tree) primitive [9] which facilitates movement of particles between the root and the leaves along shortest paths. Our approach to reconfiguration is to use multiple overlapping trees to guide the particles between the supply and demand regions. To do so we need trees with a more restricted shape than arbitrary SP-trees. In this section we hence introduce feather trees which are a special case of SP-trees (Figure 3).

Feather trees largely follow the same construction as the SP-trees. A feather tree consists of shafts and branches. Shafts are straight connections emanating from the root (and sometimes reflex nodes) that grow branches on either side. Branches are straight connections in the tree that do not branch further. To grow a feather tree, the root particle chooses a maximal independent set of neighbors; this set contains at most three particles. The particles in the independent set grow the shafts (in red) emanating from the root. All other neighbors of the root are the beginning of a blue branch. If a particle \( p \) at the end of a shaft or branch activates, it first extends the tree straight. Specifically, if \( i \) is the port from \( p \) to its parent, \( p \) extends the tree into the direction \( i + 3 \). Recall that all arithmetic on ports and directions is modulo six. The particle \( q \) in direction \( i + 3 \) becomes a child of \( p \) and \( p \) becomes the parent of \( q \). Next, if \( p \) lies on a shaft, it starts branches in the directions \( i + 2 \) and \( i + 4 \).

To reach all the particles of \( \mathcal{P} \) with a feather tree, and not just those within one bend from the root, we extend our construction around reflex vertices on the boundary of \( \mathcal{P} \). If for a particle \( p \), direction \( i \) is the direction to its parent, and the direction \( i + 1 \) (or \( i - 1 \)) does not contain a particle, while the direction \( i + 2 \) (or \( i - 2 \)) does, then \( p \) lies on a reflex vertex of the boundary of \( \mathcal{P} \). If in addition \( p \) lies on a branch, \( p \) starts a new shaft in the direction \( i + 2 \) (or \( i - 2 \)), see Figure 3 (right). We hence have the following lemma:

Lemma 1. Given a simply connected particle configuration \( \mathcal{P} \) with \( n \) particles and a
particle $r \in \mathcal{P}$, we can grow a feather tree from $r$ in $O(n)$ rounds.

Feather trees are unique, which helps with navigating the particles. Next we describe how to navigate multiple overlapping feather trees. First we identify a useful property of shortest paths in feather trees.

We say that a vertex $v$ of $G_P$ 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 an angle of 120°. We say that a bend is an inner bend if the middle vertex is an inner vertex; otherwise the bend is a boundary bend.

**Definition 2 (Feather Path).** A path in $G_P$ is a feather path if it does not contain two consecutive inner bends.

We now argue that every path from the root to a leaf in a feather tree is a feather path. A root-to-leaf path bends either on a shaft or on a branch; the path can transition from a shaft to a branch or vice versa only at a bend. If the bend occurs on a shaft, then it can be either an inner or a boundary bend; the path leaves the shaft and continues on a branch. Branches grow straight with one exception: if they detect a reflex vertex on the boundary of $\mathcal{P}$. In this case a bend occurs on the branch. This bend is always a boundary bend; the path leaves the branch and continues on the new shaft.

**Lemma 3.** Every path from the root to a leaf in a feather tree is a feather path.

**Navigating feather trees.** Due to its limited memory, a particle cannot identify different trees. However, particles can navigate feather trees by counting inner bends, even in presence of multiple overlapping trees. Therefore, while moving down a specific tree, a particle always knows if it is on a shaft or a branch and which directions belong to the tree. Starting from the root of a feather tree, a particle can always reach one of its leaves. We cannot control which leaf it reaches, but it will do so along a shortest path from the root. In particular, if feasible, it is always a valid choice for the particle to continue straight ahead. A left or right 120° bend is a valid choice if the particle is on a shaft, or if this is a boundary bend.

When moving up from leaves, we cannot control which root of which feather tree a particle will reach, but it will always travel along a shortest path. In particular, if the particle is moving along a shaft, 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

We now explain how to use feather trees to create a supply graph in the core $I \cap T$ of the particle system that connects supply roots and demand roots along shortest paths. This graph serves as a navigation network for the particles moving from supply to the demand.

Let $G_{I \cap T}$ be the subset of $G$ induced by the nodes of $I \cap T$. We say a supply graph $S$ is a directed 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 construct supply graph $S$ from feather trees as follows. First, every demand root initiates the growth of a feather tree. When a feather tree reaches a supply root $s$, a supply
found token is sent back to the root of the tree. Note that if several feather trees overlap, a node in charge of forwarding the token up the tree cannot determine which specific tree the token belongs to. However, it can identify and forward the token to all valid parents that lie on valid feather paths for this token. To count inner bends, the supply found token carries a flag $\beta$ which is updated at each 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 $S$,
2. $p$ stores the direction $i$ that $t$ came from as a valid child in $S$,
3. from all of its parents in all the feather trees, $p$ computes the set $U$ of valid parents by checking the flag $\beta$ of $t$,
4. $p$ adds $U$ to the set of its parents in $S$,
5. $p$ forwards $t$ to the particles in $U$, updating the flag $\beta$ if necessary, and
6. $p$ stores the complete information about $t$ for the future.

Note that a particle $p$ receives at most two supply found tokens from each direction, one for each value of $\beta$. Hence $p$ can store the corresponding information in its memory. When a feather tree $F$ reaches a particle $p$ that is already marked as part of the supply graph $S$ then $p$ first checks if $F$ would have been included in $U$ for at least one of the supply found tokens $t$ stored in $p$. If that is the case, $p$ sends a copy of $t$ towards the root of $F$, and sets its parent in $F$ to be a parent in $S$ as well. Otherwise, $p$ grows $F$ as normal.

**Lemma 4.** Given a simply connected particle configuration $\mathcal{P}$ with $n$ particles, a set of particles marked as supply roots, and a set of particles marked as demand roots, we can create a supply graph using $O(n)$ rounds.

**Algorithm.** We present a high level overview of our algorithm. A complete description and analysis can be found in the full version.

In the first phase of the algorithm, the particles form the supply graph, by creating feather trees starting from the demand roots, and sending tokens back up the trees if a branch finds supply. Each supply root organizes the corresponding supply component into a tree; these supply trees are connected to the supply graph $S$ via the supply roots. After the supply graph $S$ has been formed, demand roots pull particles from $S$ and fill the demand components; the pulling of particles propagates through $S$ to the supply components. Particles moving in $S$ store the number of inner bends they take. An extended particle checks the number of inner bends of its children in $S$ to determine which of them are valid choices to pull. Eventually, pulling propagates to the supply particles at the leaves of the corresponding spanning trees. These particles simply contract, reducing the outstanding supply. If a supply component becomes empty, some extended particles may need to revert and pull back against the prescribed direction in $S$. Then the corresponding edges get removed from $S$, thus rerouting the movement of extended particles towards supply that still exists.

Recall that the particles follow feather paths through $S$. This ensures that even if the particles themselves do not know which supply component they are pulling from, they do so via a shortest path. Moreover, even in the case where particles have to move back because a supply component was already empty, the total path taken by each particle will be at most linear in the size of the particle configuration.

**Theorem 5.** The particle reconfiguration problem can be solved using $O(n)$ rounds of activation.
4 Conclusion

We have presented a fast reconfiguration algorithm for a system of amoebot particles. Our solution currently only works on simply-connected particle systems with a simply-connected intersection of the initial and the target shapes. An interesting direction to explore is to extend our results for a larger class of shapes. Furthermore, our worst-case running time matches the efficiency bounds of existing solutions, however we expect our algorithm to be more advantageous for the case when the initial and target shapes are similar. Thus it would be interesting to evaluate our solution experimentally on realistic scenarios.

References

1. 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.

2. 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.

3. 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.

4. Zahra Derakhshande, Shlomi Dolev, Robert Gmry, 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.

5. Zahra Derakhshande, Robert Gmry, 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.

6. 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.

7. 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.

8. Robert Gmry, Kristian Hinzenhal, 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.

9. Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Coordinating Programmable Matter via Shortest Path Trees. In *Book of Abstracts, 37th European Workshop on Computational Geometry*, pages 32:1–32:7, 2021.

10. André Naz, Benoit Piaranda, 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.

11. 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.
12 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.

13 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabry, 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.