Universal Coating by Programmable Matter in 3D

Traversat, Wayan K.R.

Award date:
2020

Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Universal Coating by Programmable Matter in 3D

Wayan Traversat

Supervisors:
Irina Kostitsyna

version 1.0

Eindhoven, August 2020

Abstract

Matter dictates the physical properties of everything around us. Orchestrated by atoms and the laws of nature, these physical properties are unique to each material. These properties are often static and hard to modify. The objective of programmable matter, is to create a physical material that is programmable, scalable, autonomous, versatile, reconfigurable, robust to failures, and nanoscopic. Currently, programmable matter is not conceivable since particles are macroscopic in size, the advances of technology show promising results for the years to come. While we overcome the engineering challenges on our quest for miniaturization, engineers often sacrifice capabilities for size.

To model these limited particles, we use the geometric amoebot model. The amoebot model provides a framework that defines how particles can interact with each other to solve a given problem. In the amoebot model it is agreed that particles: are anonymous entities; operate on a given graph; have limited computational power; have constant memory; interact and communicate strictly locally; have limited locomotion capabilities; and are activated by a scheduler.

Under this model, we define a set of two problems: the *Filling Problem* and the *Coating Problem*. We will solve these problems under the amoebot model and under a sequential scheduler. We will prove that for any concurrent execution (asynchronous scheduler) with neighborhood locking, there exists a sequential ordering of actions that yield the same output. In the *Filling Problem*, an object forms a 2D perimeter of a bounded area to be filled by programmable particles. In this thesis, we present an algorithm that solves the *Filling Problem* in $\mathcal{O}(n \cdot R)$ asynchronous rounds, where R is the length of the longest chain of connected particles.

In the *Coating Problem*, the surface of an object O must be coated with uniform layers of programmable particles. The *Coating Algorithm* builds upon the *Filling Algorithm* to present a novel algorithm which solves the *Coating Problem* in $\mathcal{O}(n \cdot R)$ asynchronous rounds, where R is the length of the longest chain of connected particles. The algorithm only assumes that the initial set of particles in contact with the object to be coated is connected.

Contents

List of Figures

Chapter 1

Introduction

Computer components have grown exponentially in computing power while simultaneously decreasing exponentially in size. The computing power needed to send Neil Armstrong to the Moon can now fit in a space smaller than a smartwatch. This progress has enabled advances in the field of programmable matter. Programmable matter is the notion that a material can be programmed to alter its physical properties (strength, shape, conductivity, etc.). In our world, the properties of matter are controlled by atoms that constitute up everything around us. In the concept of programmable matter, atoms are particles that can autonomously self-organize to achieve a collective goal.

Programmable particles are small robots whose size is linked to the dimensions of their internal components (e.g. sensors, processors, memory, power). In the quest to miniaturize these components, the functionalities and capabilities of particles are often sacrificed to gain space. As these robots are pushed to their smallest size possible, with today's technology, they are limited in their operation (e.g. 2D locomotion, contact communication, shared power).

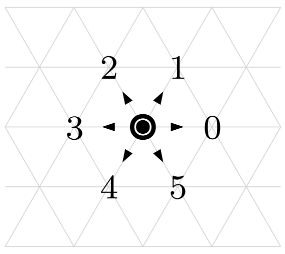
If programmable particles could become small enough, one hopes that they could mimic atoms and assemble themselves in specific structures. The different structures created by programmable particles aim to acquire certain properties, for example, programmable particles could arrange themselves to form a knife, but at the click of a button, could morph into a glass, a helmet, or anything one can think of, the main constraint being the number of programmable particles.

Just like the room-sized computers of the '60s, today's programmable particles are impractical partially due to their size. The size of a particle is linked to the degree of precision to which it can carry out its task. Particles 1 cm in size, reconfiguring into a knife won't be able to reproduce the serrated edge of the blade whereas particles 1 millimeter in size can form that serrated edge. When they are 1 micrometer in size, they could arrange themselves in a crystalline structure to give the knife its strength or sharpness. At the nanometer scale, particles can alter the way light is reflected off an object to modify its color on command.

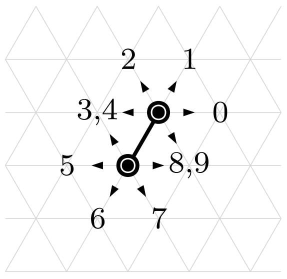
This ability to program the physical properties of programmable matter opens a wide gamut of applications. Their size and autonomous behavior make programmable matter ideal for remote, dangerous, or inaccessible areas. In space, where weight is paramount, an astronaut with programmable particles could virtually have any tool he might need. These programmable particles could morph into a screwdriver, or the screw itself, they could be used to patch a hole in the hull of the station or a hole in the astronaut suit while acting as a solar array tracking the sun. In medicine, the particles' size and autonomy could facilitate difficult treatments: by forming filters in arteries that help the fight against leukemia; by targeting tumors or infectious bacteria and covering them once identified; and by delivering drugs with superior precision. Most of these applications can be boiled down to two main problems, coating and shape formation.

In this thesis, we have chosen to focus on the universal *Coating Problem*: using a system of autonomous particles, how can we coat any object uniformly?

This *Coating Problem* is often referred to as smart coating [6] or universal coating problem [7]. In this thesis, we will present two algorithms under the Amoebot model [1] that address the universal coating problem, one in 2D and the other in 3D.



(a) Contracted particle with ports labeled 0 to 5



(b) Expanded particle with ports labeled 0 to 9

Figure 1.1: Comparing particle's ports

Amoebot Model. The *amoebot model*, introduced by Z. Derakhshandeh, S. Dolev, R. Gmyr and A. W. Richa [5], is a computational model used to represent a collective system of nanoscopic particles. This particle system is represented by a graph, where vertices represent all the possible positions a particle can occupy and edges represent all the possible atomic transitions a particle can perform from one position to another.

Particles can be in one of two states: *contracted*, or *expanded*, see Figure 1.1. In a contracted state, particles occupy one vertex and expanded particles occupy two adjacent vertices. Two adjacent particles are connected by a *bond*. A bond is used to exchange information and maintain the connectivity of the system.

A particle can transition from a contracted state to an expanded state by expanding from its current vertex to an adjacent vertex, furthermore an expanded particle can transition back into a contracted state by contracting into a single vertex.

Via a series of expansions and contractions, a particle can move through the graph. To preserve connectedness, adjacent particles coordinate their movement in one of two methods, namely *pull* and *push*. When a particle is contracted, it can expand into a vertex occupied by an expanded neighbor and push that neighbor to contract. When a particle is expanded it can contract by pulling a neighboring particle, forcing that neighbor to expand into the freed vertex. Both processes are called handover and they help particles move through the graph while maintaining connectivity.

To execute these handovers, both particles must synchronize their movement through communication via the bonds that connect them. Depending on the shape, particles make contact on several faces with adjacent neighbors. Each face of a particle exposes a *port*, where adjacent faces can make contact and form a port. Particles in 2D have six ports while in a contracted state and ten ports while in an expanded state. Particles are assumed to have a common *chirality*. This allows particles to order their ports in the same direction (i.e. clockwise direction). However, this does not mean that particles share the same global orientation.

Choosing particles' activation order is the role of the *scheduler*. There are two main types of schedulers: *synchronous* schedulers and *asynchronous* schedulers. Firstly, under synchronous schedulers, the timeframes for a particle's activation cycle are fixed. There are three cases of synchronous schedulers and they differ in the number of activated particles for each timeframe. Under the *semi-synchronous* model at least one particle is activated in each timeframe whereas in the *sequential* model, exactly one robot is activated for each timeframe. Under the *fully synchronous* model all particles are activated each timeframe. Lastly, under the *asynchronous* model the timeframes are not fixed, particle's activation is random and independent, and any number of particles can be active at any time.

Communication between particles occurs by writing to or reading from adjacent particles' memory. It is assumed that the particle's memory is of constant size (e.g. particles are *anonymous*, because the number of particles is not constant). Under all schedulers except the sequential model, concurrent operations can lead to conflicts. Memory access can be an example of such conflicting operations, where two particles attempt to access the same memory address, we assume that memory conflicts are resolved arbitrarily.

When a particle activates, it operates in a look-compute-move cycle, where a particle observes its local environment, performs an arbitrary bounded number of computations involving its own local memory, its neighbors' memory, and the environment; effectuates at most one movement; then goes to sleep awaiting its next activation cycle. We assume that all particles are reliable, meaning that all actions performed by the particles are faultless, and any cycle started will be completed.

It is through these atomic activations that the system progresses. For each of these activations, particles need static information from at most N-neighborhoods depending on the algorithm. To preserve the concurrent execution of multiple particles, information from a particle's N-neighborhood must be locked until its cycle completes. From research [15], when the system's actions are atomic and isolated, the set of actions can be serialized; this applies for any set of actions performed by this system. For any concurrent execution, there exists a sequential ordering of actions that yield the same output.

While in reality such a system of particles may run concurrently, we can analyze algorithms under the amoebot model where at most one particle is active at any time. A round under this model assumes that each particle has activated at least once and the rounds are fair. Fairness ensures that every particle will be reactivated in the future, this property ensures that the system can make progress.

Universal Coating Problem. In the *Universal Coating Problem* [7], a particle system must coat an object with uniform layers of particles. While solutions have been explored in 2D under the amoebot model, no paper addresses the 3D variant of this problem under the amoebot model.

In this problem, the object to be coated must be immutable, meaning the object is fixed and cannot change shape during the execution of the algorithm, and the surface to be coated must be connected.

The coating problem is solved once all particles make contact with the surface of the object and particles are evenly distributed throughout all the layers. Furthermore, all particles must be contracted and in a stable state, meaning that particles will not change state from the moment of completion.

1.1 Related Work

The *Universal Coating Problem* has been solved under the Amoebot model in 2D [7]. In this section, we will describe the model that is used to solve this problem as well as the different subroutines used to solve the problem in 3D.

The solution found to the *Universal Coating Problem* cannot be directly used in 3D. The solution found utilizes the fact that there exists only one path along the boundary of a 2D object whereas in 3D, this path is not unique.

Research under the Amoebot is still an emerging technology and little research has been done on the topic. There are some subroutines that are used in this paper, most notably: *tree formation* and *leader election*. Furthermore, a pre-existing simulator was used to visualize the algorithm created in this paper.

Existing solutions to the *leader election* problem vary in the pre-conditions. Some solutions assume that the particles have a common chirality, others use randomness to find a leader, whereas some assume that the particle system does not have holes, and others yet may find multiple leaders [1, 3, 8, 11, 12]. Recent work has managed to solve the leader election problem assuming a sequential scheduler in $O(Ln^2)$ rounds [10].

While the work for leader election has solely been conducted in 2D, we will be reducing the 3D leader election problem to a 2D problem; the leader will be elected in the subset S from the initial set of particles P where S is the set of particles adjacent to object O.

The *tree formation problem* has been addressed in a paper under the name of *Spanning Forest Primitive*[7]. The paper proposes a solution to the tree formation problem and can be modified to work for both the *Filling Problem* and the *Coating Problem.* The *Universal Coating Algorithm* in the same paper, uses the *Spanning Forest Primitive* subroutine to organize particles in a spanning forest. This structure is used to guide particles throughout the coating process while maintaining connectivity.

1.2 Contribution

In this thesis we solve the *Universal Coating Problem* by self-organizing particle systems in 3D by proposing a set of two algorithms. A simulator was developed to visualize and test the created 3D coating algorithm. The first algorithm reduces the *Universal Coating Problem* on self-organizing particle systems from a 3D problem to a 2D problem. We will refer to the 2D problem as the *Filling Problem*, defined in Section 2.5. We know that by filling a 2D space representing the net of a 3D shape, this 2D texture map can be folded to reconstruct a 3D coat. This technique is inspired by UV mapping: the process of projecting 2D planes onto a 3D object.

The second algorithm, called *Coating Algorithm* found in Chapter 5, will be inspired by the *Filling Algorithm* found in Chapter 1. While the 3D algorithm is built on the same core concept as the 2D algorithm, it greatly improves upon it. The *Coating Algorithm* uses, for example, multiple leaders and multiple layer coating. The *Coating Algorithm* is a solution to the *Universal Coating Problem* in 3D.

To visualize and test the algorithm, a 3D simulator was created. The simulator was extended to allow other researchers to visualize their 3D algorithms. Tools in the simulator were added to provide users with statistics, debugger functionality, and execution history.

Chapter 2

Preliminaries

2.1 Definitions

Particle. In this thesis, we present two algorithms: one solves the *Filling Problem* in 2D and the other solves the *Coating Problem* in 3D, both problems are solved under the amoebot model, defined in Section 2.2. For each problem, we will define the graph G on which the problem is solved. In 2D, $G = G_{2D}$ where G_{2D} is an equilateral triangular lattice, whereas in 3D, $G = G_{3D}$ where G_{3D} is a rhombic dodecahedral honeycomb which is the Voronoi diagram of the face-centered cubic sphere-packing. For any particle $p \in P_{2D}$ or $p \in P_{3D}$, p has a state s where $s \in S$ where:

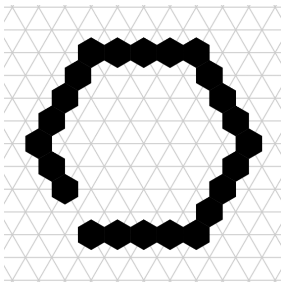
$$S = \{ \text{ \textit{Contracted}, \textit{Expanded} } \}$$

$\forall$ p has a role q where $q \in Q_{2D}$ in 2D and $q \in Q_{3D}$ in 3D. We will define Q_{2D} as the set of roles exclusive to the *Filling Algorithm* and Q_{3D} to be the roles exclusive to the *Coating Algorithm.* We will further define Q_{2D} and Q_{3D} in their respective sections.

A particle also maintains a constant size memory that can be read/written by any adjacent particle. The content of a p particle's local memory will be denoted as $p.x$ where x represents a piece of fixed information. The content of this memory will be implementation-specific and will be described in the respective section for each algorithm. Notations like *p.parent* and *p.child* will also be used, and while this information is not stored in local memory, it can be inferred from the local context and makes the ownership of memory being described easier to explain.

Object & Layer. We will define A_{2D} as the area for the 2D problem and L_{3D} to define the layer for the 3D problem. For A_{2D}, the object O_{2D} is a perimeter represented by a set of boundary vertices $v \in V(O)$ forming an inner region to be filled. This finite empty region is the area A_{2D} to be filled by the *Filling Algorithm.* $V(L)$ denotes the available positions a particle can occupy and $E(A)$ denotes the possible path from one vertex to another. One boundary vertex $v \in V(O)$ will be empty and reserved for the leader particle. For L_{3D}, the object O_{3D} is an immutable connected shape. In 3D, multiple layers exist, hence we will denote L_i where i denotes the layer number and distance to object O. We define $d(u, v)$ as the minimum distance between vertex u and vertex v. Hence L_1 is composed of all vertices v where $d(v, V(O)) = 1$. A layer L_i is deemed complete, when L is connected, when all vertices $v \in L_i$ are occupied by a contracted particle $p \in P$ where p is contracted and p is stable (in our case, when p is in role *retired*, p is stable). We call a particle p *stable* when p does not perform any role change or movement.

Configurations. A configuration c represents a snapshot of the particle system at a time t, and will be designated as c_t. The set of all configurations will be called C. Information such as the state of all particles, local memory, and the object is stored in a configuration c. We will define a configuration c to be *stable* if, from a moment t onwards, no particle $p \in c$ ever performs an


(a) O_{2D} a hexagon of size 4

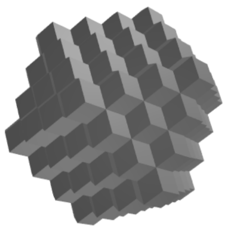

(b) O_{3D} a rhombic dodecahedron of size 4

Figure 2.1: Example of objects in both dimensions

operation or a movement. This is commonly achieved by having a role $s \in Q$, where a particles with role s clears its local memory and cannot perform a role transition.

In 2D, we will define a configuration to be *legal* if for all particles $p \in P(L)$ are contracted and stable. In 3D, we will define a configuration to be legal if for all particle $p \in \{L_1 \cup ... \cup L_{i-1}\}$ for all $i > 1$, are contracted and stable.

We will define an initial configuration c at $t = 0$ for both problems. In 2D, there must exist at least one particle p_1, where $p_1 \in C_0$ and $d(p_1, L) = 1$ (p_1 is adjacent to the layer). In 3D, there must exist at least one particle p_1, where $p_1 \in C_0$ and $d(p_1, V(O)) = 0$ (p_1 is on the layer). The layer must be valid for the given problem.

Tree Definitions. There are three tree structures used throughout this thesis: T_I standing for Tree of Initial particles, T_S standing for Tree of Surface particles, and T_L standing for Tree of Leader particles. The first two trees are used in the 2D *Filling Algorithm* and all three are used in the 3D *Coating Algorithm*. The first two trees, T_S and T_I are illustrated in Figure 2.2. T_I is created by the *Tree Formation Algorithm*, see Section 2.3, and it is used to link inactive particles to the leader particle. T_S is used to link the particles as they fill/coat the area/layer. The tree T_L similarly to the tree T_I, is rooted at a specific particle. In 3D, the initial blob of particles can have more than one contact point with the object, and hence all these surface particles (leaders) must be rooted at a specific particle, which we denote as a super leader in the *Coating Algorithm*. The tree T_L is used to link all the leader particles on the surface of O to the super leader.

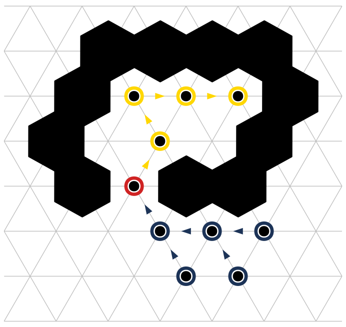


Figure 2.2: T_I set of blue vertices and T_S set of yellow vertices. Both trees include the *leader* vertex in red

2.2 Amoebot Model

In this section, we will formally define the Amoebot model for both problems. In the first section, we will define the Amoebot model for the 2D filling algorithm and in the second section, we will explain how the model differs for the 3D *Coating Algorithm*.

2D Particles denoted as p, make up a particle system P where the positions of each particle are mapped to a vertex on an infinite triangular lattice graph $G_{2D} = (V_{2D}, E_{2D})$ in 2D, where V_{2D} represents all possible positions and E_{2D} represents all possible interactions from a vertex to another. Vertices $v \in V(G_{2D})$ can be occupied by a particle, an object, or neither.

When a particle p expands, as seen in Figure 1.1, from a vertex n_1 into a vertex n_2, we call the part of p in n_2 the *head* of p and we call the part of p in n_1 the *tail* of p. Since a contracted particle p occupies a single vertex n, its head and tail are the same single vertex n that the particle p occupies. When a particle expands, we assume it is always the head of the particle moving into the new adjacent vertex and when a particle contracts its tail moves to the adjacent vertex.

Particles have six ports labeled 0 through 5 when in a contracted state and ten ports labeled 0 through 9 when in an expanded state.

The programmable particle's memory is constant. This memory limitation has multiple implications, such as: particles are unable to maintain a unique identifier (i.e. are *anonymous*), particles are unaware of the total number of particles in the system. Any information that increases with the size of the problem or number of particles cannot be stored in a particle. There are ways to circumvent this limit, some algorithms may need to keep track of a counter. In the coating problem, for example, particles are interested in knowing whether a particle is on the current or on the previous layer. While the layer counter is not constant, the modulo of that counter is. This way, a particle can identify whether a particle is on the current layer or previous layer.

Given that a particle $p \in P$ activates at a time t, we know by our fairness assumption that such particle will activate again at $t' > t$.

3D In the 3D variant of the problem, particles are not 2D hexagons but rather are represented by a 3D Rhombic Dodecahedron. To arrange rhombic dodecahedrons on a graph, we use a different graph $G = G_{3D}$ where $G_{3D} = (V_{3D}, E_{3D})$ is an infinite face-centered cubic (fcc) lattice graph. A Rhombic Dodecahedron is a polygon with 12 congruent faces, for any particle $p \in P$ in 3D, p has 12 ports when contracted, and 22 ports when expanded.

2.3 Existing Subroutines

The algorithms present in this section have been taken from existing papers. Algorithms have had slight modifications made to them while maintaining their validity. The *leader election* algorithm runs its routine on all particles in the system, while we only want particles on the surface to be eligible for leadership. In the *tree formation* algorithm, the root of the tree is the particle actively coating, where as in our scenario, the position of the root of the tree remains fixed, while other particles perform the coating/filling task. The leader election subroutine must complete before the tree formation can start. Once the tree formation subroutine has begun, the *Filling Algorithm* or the *Coating Algorithm* can start.

Leader Election. In this thesis, we will assume the existence of a leader particle. Extensive research on the leader election problem [1, 3, 8, 11, 12] shows that in our initial configuration, a leader can be found. For our algorithm, we are interested in a leader algorithm which does not use movement and finds exactly one leader. The work presented in this paper [8] presents an algorithm that meets these conditions.

This leader election algorithm relies on probability and communication to elect a leader in a static system. Initially, all particles in the system consider themselves a potential leader candidate. The particles first identify if they are on the boundary of the particle system and the subset of particles on this boundary form a directed cycle. This cycle is divided into segments which decide the particles' successors and predecessors. Particles along the segments decide through *coin flips* if a particle will revoke its own candidacy and in turn promote the candidacy of its successor. A segment consists of one candidate and particles promoting that candidate, and each segment maintains an identifier which is used to compete with other segments. When a segment recognizes its own identifier, the final solitude check begins to identify whether that segment is the last. The algorithm has been proven to converge towards exactly one leader with high probability. A variation of the algorithm, in the same paper, proposes a solution that converges to one leader with a probability of 1.

For the *Coating Algorithm*, we make a slight modification to the algorithm by choosing the subset of particles which participate in the leader election algorithm. Instead of choosing all particles as initial candidates, only the subset of particles on the surface of the object O will be chosen as potential leader candidates for the system. This ensures that the set of potential candidates is a 2D set on which we can run the leader election algorithm.

Tree Formation. The algorithm used for tree formation is presented in the following paper [8], it has also been thoroughly explained in this book [14].

The algorithm relies on the existence of a special particle p_s, called a *root* in the tree formation algorithm, but called a *leader* in our algorithm. The purpose of the algorithm is to structure the particle system into a tree, such that particles are oriented toward p_s. Such a tree structure is formed with the interaction of two groups of particles: active particles and inactive particles. An inactive particle p_1 becomes active by following an active particle p_2. By following p_2, p_1 creates a connection to p_2. These connections can only be created by adjacent particles. The conversion process is fairly straightforward, when a particle p wakes up, if p is inactive, it will scan its adjacent ports looking for an adjacent active particle (i.e. follower or leader in our case). If an active particle is found, particle p will follow the newly found particle. Initially, there is only one active particle, p_s, chosen by the *Leader Election Algorithm* and only particles adjacent to p_s will become active and are assigned a *follower* role. Subsequently, *inactive* particles will be able to follow other *followers.* If no active particle is present in its surrounding, particle p will sleep and await its next activation. This iterative process and the fairness property of the system ensures that all particles will eventually become active. The connection created by the active particles creates a tree that is rooted at p_s.

2.4 Amoebot Simulator

Visualization and experimental results for the 2D filling algorithm were created using *AmoebotSim* [1]. AmoebotSim is a visual simulator operating under the amoebot model. This simulator was used to explore ways to solve the Universal Coating Problem under the amoebot model. This research led to the creation of the Filling Algorithm, an approach to the Universal Coating Problem.

It is worth noting that the *AmoebotSim* is using a specific scheduler. We will call this scheduler *sequential multiple activations.* This scheduler activates a single particle at random for each activation and a round is incremented once each particle has been activated at least once. We will discuss in Section 5.1 the implications of using such a scheduler.

[1]https://amoebotsim.readthedocs.io/en/latest/

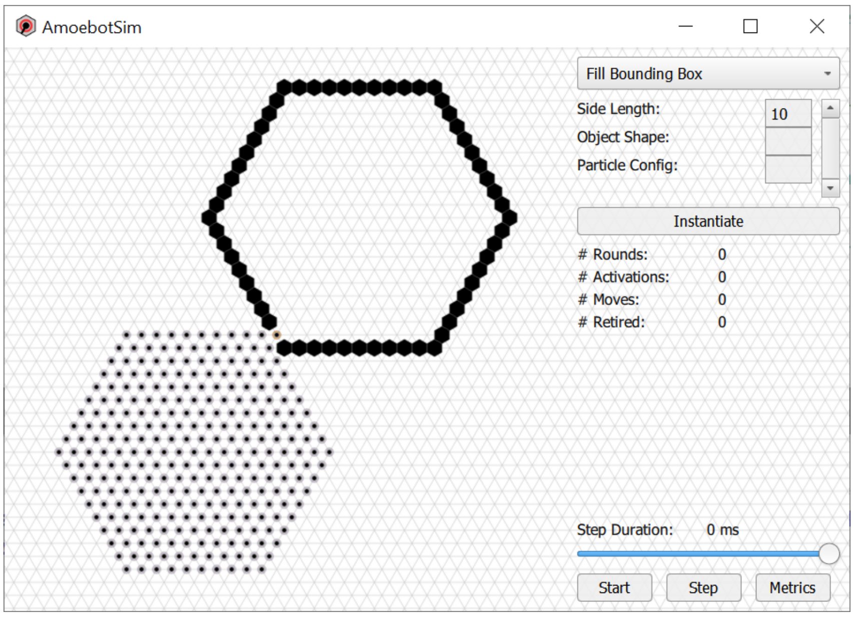


Figure 2.3: AmoeBot simulator by SOPS lab

2.5 Problem Definition

2D Filling Problem. In this paragraph, we will formally define the *2D Filling Problem.* In this problem we are given a 2D object O. This object is a set of vertices $V(O)$ which forms a boundary. The object O is connected and immutable. The area formed by O is the area A and this area A is also connected. For all $p \in P$, we denote the subset of particles in A as $p \in T_S$. We say that a system of particles has filled the area A, if:

$$\forall\, p\ \{\ \ p \in T_S\ \mid p.state = contracted\ \wedge\ p.role = retired\ \}\ \wedge\ |T_S| = |A|$$

3D Coating Problem. In this paragraph, we will formally define the *3D Coating Problem.* In this problem, we are given a 3D object O. This object is a set of vertices $V(O)$ which forms a 3D polyhedron. The object O is connected and immutable. The surface of O is the first layer L_1 and this layer L_1 is also connected. When a layer L_i is complete, the surface of L_i becomes layer L_{i+1}.

For all $p \in P$, we denote the subset of particles in L as $p \in T_S \cup T_L$. We say that a system of particles has coated layer L_i if:

$$\forall\, p\ \{\ p \in \{T_{S_i} \cup T_{L_i}\} \mid p.state = contracted \wedge p.role = retired \wedge d(p, V(O)) = i\ \} \wedge |T_S \cup T_L| = |L_i|$$

The statement states, that for all particles participating in coating layer L_i, the layer is complete once all participants have retired in a contracted state.

Chapter 3

The Filling Algorithm

In this thesis, we will first simplify the Universal Coating Problem to a 2D filling problem. From geometry, we know that some 3D objects can be unfolded onto a 2D area. A popular 3D rendering technique called UV Mapping, illustrated in Figure 3.1, is used to wrap 2D texture maps onto 3D meshes and vice-versa. Using this technique, we will simplify the coating problem to a filling problem, where an algorithm will fill the 2D net of a 3D shape. The purpose of this algorithm is to establish a better understanding of the problem.

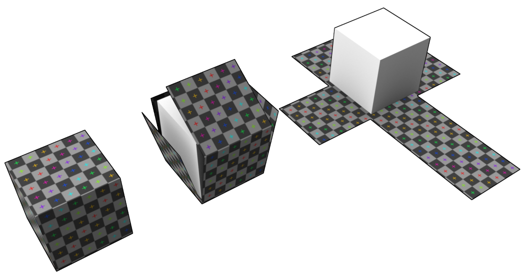


Figure 3.1: UV Mapping a 3D cube by *Wikipedia* [17]

There are limitations to this technique, as not all 3D objects can be unwrapped into connected self-non-overlapping 2D maps. There are two known problems related to this issue: Edge-Unfolding Convex Polyhedra [9, 16] and Vertex-Unfolding Polyhedra [4]. We know that the Edge-Unfolding of Non-Convex Polyhedra has been proven to be unsolvable [2] and we know that there exists a vertex unfolding for simplicial polyhedra [4]. The overall hunch is that Edge/Node Unfolding of Convex Polyhedra is solvable [13] but no formal proof for all cases has emerged.

For cases where a 2D map exists for a polyhedron, we can represent this map with a connected, non-overlapping net which can be projected onto a discrete triangular graph G_{2D}, as seen in Figure 3.2. An available position, that a particle can occupy, is represented by a vertex $v \in V(G_{2D})$. Edges represent the available paths between vertices. The boundaries of this 2D map are marked with object vertices $v \in V(O)$. These vertices can be seen as black vertices in Figure 3.2). On one hand, the perimeter of the inner region is delimited by black vertices. On the other hand, the inner region to be filled is represented by grey vertices (which we will commonly refer to as the filling area), both are shown in Figure 3.1). The net shown in the figure can be folded to obtain a 3D cube.

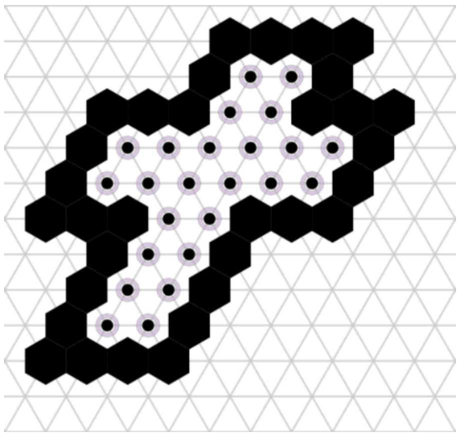


Figure 3.2: 2D unfolding of a 3D cube projected on a triangular lattice. Black vertices area bordering the region to be filled, grey vertices identify possible positions to be occupied by particles.

3.1 The Algorithm

For a higher-level overview of the algorithm, please refer to Figure 3.3 which explains the main phases of the algorithm. The *Filling Algorithm* uses a set of two trees; one tree T_I, connects all *follower* particles to the *leader*, and another tree T_S is created as particles fill the area, ensuring that leaf particles can pull particles through the leader. In essence, the filling task is accomplished by *leaf* particles which expand through the area to be filled. For each expansion a *leaf* particle makes, it initiates a series of handovers from itself through the *leader* to a leaf particle in T_I, see green particles Figure 3.3b. Each expansion/contraction cycle completed by a *leaf* increases the area covered by the particles and reduces the size of the problem. To accelerate the process, some *filler* particles can branch off from their path and hence, create a new *leaf* particle. Where the new *leaf* branched off, a *branch* particle sits at the bifurcation to funnel *filler* particles in either direction, see brown particles in Figure 3.3c. When a *leaf* particle p can not expand into any direction, all adjacent ports are occupied by either particles or the boundary of the object O, then that particle becomes *retired*. Subsequently, the parent of p becomes a *leaf*. This recursive handover of the *leaf* role ensures that no blank holes are left inside the area to be filled. Once the child of the *leader* particle has *retired*, we know that the filling process is complete. The *leader* now becomes a *retired* particle and hands his role to his parent *follower* particle.

Roles. Any particle p has a finite set of roles Q, below are enumerated the possible roles in which p can be, along with a description for each of these roles. Furthermore, Figure 3.3 showcases the different roles in an example. There are two main types of roles: *static* and *dynamic*. *Static* roles are tied to a vertex v, meaning that any particle p which has its head or tail on that vertex v will have that role, and when a particle leaves that vertex, it loses that role. *Dynamic* roles are tied to a particle, while a particle fulfills a set of conditions, the particle will keep its role during movement. Movement in this thesis will only occur through one of the two handovers: pull. An expanded particle p can only pull a contracted parent. Note that in this thesis we will mention roles in italic, this is to distinguish between the role *leaf* and the leaf of a tree in data structures. Throughout these descriptions, we will be referring to memory registers, which will be explained thoroughly in the next paragraph section. Memory registers for particle p will follow this format, where $p.x$ refers to the memory register labeled x, we will also use $p.parent$ and $p.child$ to refer to the parent and child of p respectively. Combining a parent reference with a memory registry can yield; $p.parent.role$ which refers to the role of the parent of p. We will now distinguish p particle's behavior based on its role:

- ***Leader*** is a *static* role. As a *leader*, p acts as a gateway through which all the particles flow. After the leader election phase, a leader is elected at vertex v where $d(v, V(L)) = 1$. A leader will always have either its head or tail on vertex v. When an expanded particle p has its tail on v, p is a *leader* particle, and when p only has its head on v, p is a *small leader* for reasons explained in the paragraph below. When an expanded *leader* p_1, initiates a handover, pulling a contracted particle p_2, the leader p_1 will hand its role to p_2 which will be converted to a *small leader*. When the small leader contracts, it will become a *leader* particle and will not hand the *leader* role to another particle. A contracted leader l checks for $l.child$ particle, via the $p.childDir$ register. If $l.child.state$ is contracted and $l.child.role$ is *retired*, l can safely retire and this terminates the *Filling Problem*. A contracted leader can also have no child, in that case $p.childDir$ is -1, this will trigger a special condition enabling the *leader* to expand inside the shape and upon its next contraction will become a *leaf* particle.

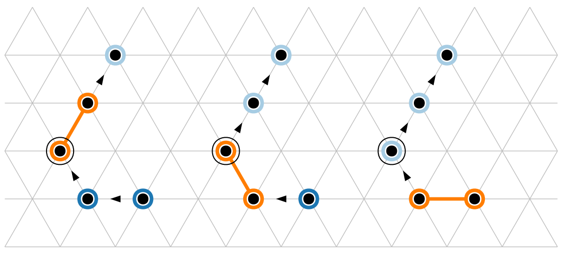
Pulling: when p pulls a particle p_1, p_1 will always be converted to a *small leader*. *Leader* p will either become a *leaf* if $p.child$ does not exist, else it will become a *filler* particle.

Pulled: when p is pulled by a *leaf* particle or a *filler* particle, p will always remain a *leader* particle.

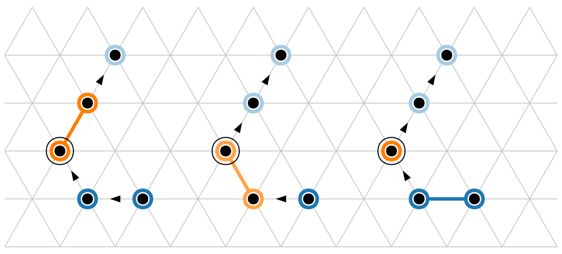
- ***Small Leader*** is a *static* role. *Small leader* is a precursor to the *leader* role. In Figure 3.4, we can observe what happens when the *leader* hands over its role after every pull handover. To prevent the particle's position from shifting, which happens when the *leader* pulls another particle, the *leader* will converted its follower to a *small leader* role. This is because, when a *small leader* pulls another particle, it will not hand its role to its follower.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role $p_1.role$. *Small leader* p will always become a *leader*.

  Pulled: a *small leader* cannot be pulled because it only exists as an expanded particle.



(a) The leader's position is shifted



(b) Using a small leader locks leader position

Figure 3.4: Using a small leader to lock the position of the leader to a vertex (circled in black)

- ***Leaf*** is a *dynamic* role. A *leaf* is the only particle able to expand into a new position (with the exception of the first expansion of a *leader*). *Leaf* particles are also the only particles able to transition to a *retired* role. This ensures that a particle cannot retire while there exists an empty adjacent vertex. When a *leaf* p retires, its role is handed over to $p.parent$.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role $p_1.role$. One exception being when a *leaf* initiates a branch, its first pull will convert a *filler* particle into a *small branch* particle. *Leaf* particle p will always remain a *leaf* after the handover.

  Pulled: a *leaf* p cannot be pulled because $p.child$ does not exist (we defined that the pull handover can only occur when a parent particle).

- ***Filler*** is a *dynamic* role. A *filler* particle only exists in T_S. When $p.state$ is *expanded*, p pulls its parent if $p.parent.state$ is contracted. When $p.state$ is contracted, p checks whether both $p.parent.role$ and $p.child.role$ are *filler*, if both are *fillers*, p then checks for each port in $p.port$ if p can expand into any adjacent position to its ports. Finally, if such a position exists, p becomes a *leaf* and expands into an available adjacent position. At the next pull initiated by the *leaf*, $p.parent$ will become a *small branch*.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role $p_1.role$. *Filler* particle p will always remain a *filler* particle after the handover.

  Pulled: since *filler* particles only exist in T_S, a *filler* particle p can be pulled by a *branch* or a branching *leaf* such that p will become a *small branch*. *Filler* particle p can also be pulled by another *filler*, *leaf* or *small branch*, these handover will not alter $p.role$.

- ***Follower*** is a *dynamic* role. A *follower* is a particle in T_I. T_I is a tree created by the *tree formation algorithm* is rooted at the *leader* particle. Any *follower* $f \in T_I$ is connected to the leader particle l. When $p.state$ is expanded, f can either pull on $f.parent$, if such parent exists, else f can contract if $f.parent$ does not exist, as it is a leaf in T_I.

  Pulling: when p pulls a particle p_1, p_1 will always be a *follower* and p_1 will always keep its role $p1.role$. *Follower* particle p will always keep its role after the handover.

  Pulled: since *follower* particles only exist in T_I, a *follower* particle p can be pulled by a *leader* such that p will become a *small leader*. Particle p can also be pulled by another *follower*, where it will keep its own $p.role$ role.

- ***Branch*** is a *static* role: Similarly to the *leader* role, the *branch* role is linked to a vertex v in the graph. The *branch* role marks a bifurcation in the tree T_S. Each *branch* marks the beginning of the path of either two *leaf* particles, the beginning of both path are remembered using two memory registers $b.childDir$ and $b.branchChildDir$ which marks the port adjacent to either particles at the beginning of either paths. For a *branch* particle b, when either particles at $b.childDir$ or $b.branchChildDir$ retires, p becomes a *filler* and keeps the $b.childDir$ register pointing to the remaining active particle. Similar to the *leader*, when a *branch* pulls $b.parent$, $p.parent.role$ will be *small branch*. This prevents the *branch* from shifting from its position v.

  Pulling: when p pulls a particle p_1, p_1 will always become a *small branch*. *Branch* particle p will always become a *filler* after the handover.

  Pulled: since *filler* particles only exist in T_S, a filler particle p can be pulled by a *branch* or a branching *leaf* such that p will become a *small branch*. When particle p is pulled by a *small branch*, it will keep its role after the handover.

- ***Small Branch*** is a *static* role: Small branch is a sub role of the *branch* role. To prevent the *branch* from shifting position, when a *branch* pulls another particle p_1, $p_1.role$ will become *small branch*. We can see in the *small leader* description an explanation for why that is the case.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role $p_1.role$. *Small branch* p will always become a *branch*.

  Pulled: a *small branch* cannot be pulled because it only exists as an expanded particle.

- ***Retired*** : When a *leaf* particle lf can not expand into any position adjacent $lf.port$, then $lf.role$ becomes *retired*. When a particle retires, the content of its local memory is cleared and no further transition can occur. Particle p is said to then be stable and inactive. Particles can distinguish the difference between the initial inactive particle and a *retired* particle.

  Pulling & Pulled: a *retired* particle can neither pull a particle nor be pulled by any particle.

Memory. Any particle p has a constant memory. A particle p can read the information in its own memory as well as read/write any adjacent particle's local memory. We will describe the different memory registers held by a particle p:

State: *p.state* will either return *contracted* or *expanded.*

Role: *p.role* will return the role of a particle p. A particle p must have exactly one role $r \in Q_{2D}$ where Q_{2D} is a finite set of states:

$Q_{2D} = \{$ *Leader*, *Small Leader*, *Branch*, *Small Branch*, *Follower*, *Leaf*, *Retired*, *Inactive Filler* $\}$

A particle can transition from one role to another according to Figure 3.5. Roles are defined in the previous paragraph.

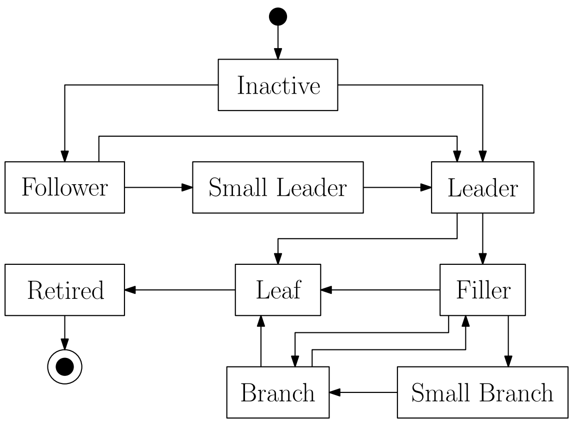


Figure 3.5: Possible role transitions in the *Filling Algorithm*

Port: *p.ports* is an array that uniquely labels each port of particle p. When *p.state* is *contracted*, *p.port* will return 6 unique labeled ports and when *p.state* is *expanded*, *p.port* will return 10 unique labeled ports (it is worth noting that, when *p.state* is expanded, 2 pairs of ports point to the same adjacent vertex, as seen in Figure 1.1).

ChildDir: If a particle p follows a particle p_1, then we call p_1 the child of p and p the parent of p_1. In order to remember this relationship, particle p uses a register called *p.childDir* which saves the *port* adjacent to p_1. The register is initialized at -1, and this also indicates that a particle does not have a child.

BranchChildDir: *p.branchChildDir* is exclusively used by *branch* particles. With *childDir* and *branchChildDir*, a *branch* particle can be pulled by either of its two children. By definition, a branch only keeps track of exactly 2 children, and when either child retires, p becomes a *filler* particle.

Algorithm 1 Contracted Particle Activation vs Expanded Particle Activation

Particle is Contracted

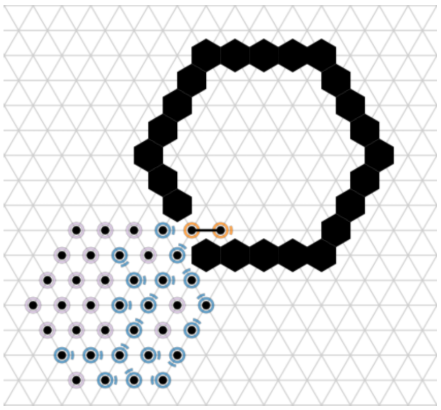
```
switch Role do
    case Fill
        if child is retired then
            if particle can Expand then
                particle becomes a Leaf
            else
                particle becomes Retd
        else if parent is Fill and child is Fill then
            if particle can Expand then
                particle becomes Leaf
                add pointer to child
                particle Expands

    case Brch
        if either child is Retd then
            Become Fill

    case Leaf
        if particle can Expand then
            Expand
        else
            particle becomes Retd

    case Lead
        if child is Retd then
            Done filling
        else if child is does not exist then
            Expand
```

Particle is Expanded

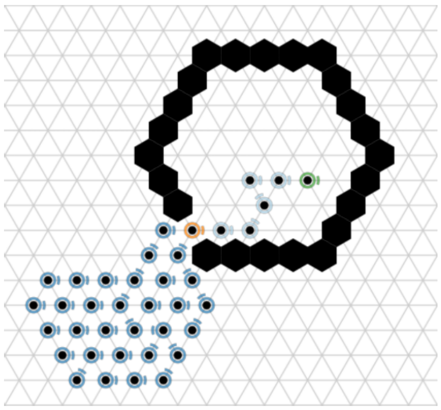
```
switch Role do
    case Fill
        if particle has a ready parent then
            pull parent

    case Brch
        if particle has a ready parent then
            pull parent
            parent becomes SBrch
            update parent pointers
            become Fill

    case Leaf
        if particle has a ready parent then
            pull parent
        else if particle has second pointer then
            pull parent
            parent role SBrch
            drop second pointer

    case Lead
        pull a ready parent
        parent becomes a SLead
        if particle has no parent then
            particle becomes Leader
        else
            particle becomes Fill

    case SBrch
        if particle has a ready parent then
            pull parent
            particle becomes Brch

    case SLead
        if particle has no parent then
            Contract
            particle becomes Lead
        else
            pull a parent
            particle becomes Lead
```

Algorithm 2 Particle p_1 pulling particle p_2

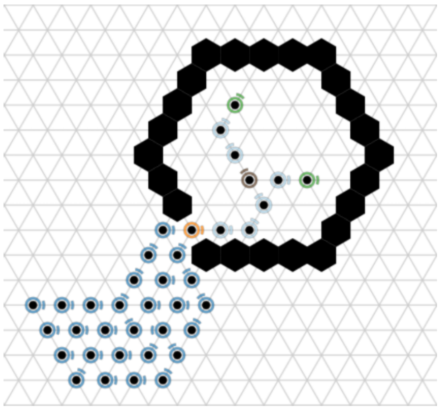
```
if $p_1$ is expanded then
    if $p_2$ is contracted then
        if $p_1.parent$ is $p_2$ then
            contract $p_1$
            $p_2$ expands into $p_1$ direction
            update $p_2.childDir$
```



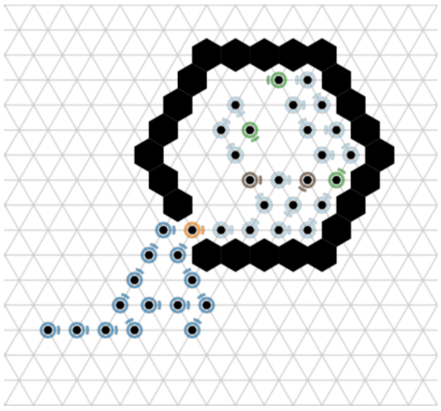
(a) Inactive particles (in grey) become active (in blue) as they join the tree T_I. The tree formed, funnels the particles through the *leader* (in orange).



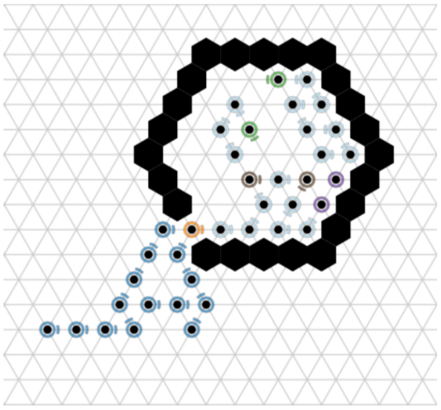
(b) Particles flow from T_I through the leader (in orange) to the filling area forming T_S which will be more evident with the branching in the next Subfigure. The flow is controlled by the leaves (in green) which initiate the pull handovers to fill the inner region.



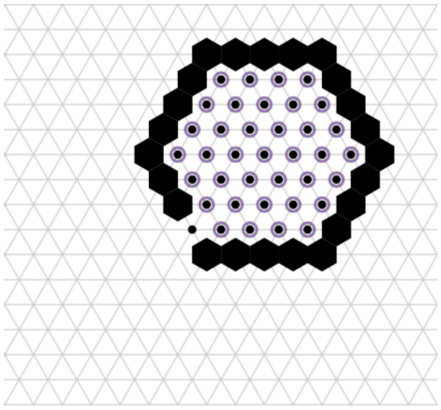
(c) It is possible for *filler* particles (in light blue) to branch off under certain conditions. When that happens a *branch* (in brown) particle will appear at the bifurcation until both branches retire.



(d) *Leaf* particle retires when it has no adjacent space to expand into. We can see that the bottom right most *leaf* l cannot expand and will retire at its next activation



(e) The *leaf* l has retired handing its role to its parent. That parent also retired during its activation because it had no adjacent space to expand into.



(f) Final configuration, all particles in the filling area are contracted and stable

Figure 3.3: Overview of the Filling Algorithm's main stages

3.2 Proof of Correctness

In this section, we will describe the building block required to prove the correctness of the algorithm. We will begin by proving that the system is connected and that connectivity is maintained as the system progresses. We will then prove that the particle system progresses and such progress reduces the size of the problem. Finally, we will prove that this progress will lead to a final state, and when the particle system terminates, that it finds a solution to the *Filling Problem.*

Let $V(\cdot)$ denote the vertex representation, likewise for $E(\cdot)$ with edge representation, we will denote $|\cdot|$ as the size. The particle system P will fill an area A bounded by an object O. To differentiate between the particles in the initial blob and particles actively filling the area A, we use two tree structures T_I and T_S respectively. The *tree formation algorithm* forms the tree T_I, used by *follower* particles to flow through the gate leader particle. The *leader* particle always resides on vertex g where g is at the hole marking the opening of the shape (this hole can be seen in Figure 2.1). As *follower* particles flow through the gate particle, they enter the area A to be filled. *Leaf* particles direct this expansion into the area and as they do, they form T_S, rooted at the *leader* g.

In the *Filling Algorithm*, only the *pull* handover will be used. A particle p can only pull *p.parent* if such parent exists, and *p.parent.state* is contracted. For p to identify which adjacent partiles are its parent, p uses the registry *p.port* to know where to look for its *child* or *parent*. Movement without handover is only possible with expansion for certain roles.

For the following lemma, we will assume that $|P| \geq |A|$. The section below will create small building blocks which will lead to the theorem proving the correctness of the algorithm.

Lemma 1. *At any moment t, particles in T_I form a directed tree.*

Proof. **Claim 1.** *Particles that join the directed tree T_I maintain connectivity.*

In order to prove this Lemma, we will use a similar structure to Claim 1 in Lemma 2 from [7]. At some time t, the tree is valid with a single particle: the leader particle. We will show that this property of the graph will hold at $t+1$. Suppose that at $t+1$, a *follower* particle f becomes active. If f is adjacent to another active particle p_1, f will set its register *f.childDir* to the port adjacent to p_1. As a parent of p_1, f extends the tree as a new leaf. At time t, T_I was a tree, at time $t' > t$, f joins the graph, maintains connectivity and the tree property.

Claim 2. *Particle movement in the directed tree T_I maintains connectivity.*

From Claim 3.2, we know that particles joining the tree maintain connectivity. At some initial time t, the tree is valid as defined in Claim 3.2, we now want to prove that for all times $t+1$ a movement will occur in the tree, and this movement will also maintain connectivity. At some time $t' > t$, the first particle movement will occur in the tree. For the tree to disconnect, a particle in *p.state* expanded with an existing *p.parent* must contract without causing *p.parent* to expand. We know from our initial assumptions that the contraction of a particle can only occur in a handover. There is however one exception to this rule, the *follower* role can contract its tail under certain conditions. Only when a follower particle f does not have a parent, can a *follower* contract. This condition ensures that no parent is disconnected from f.

In the previous two claims we show that when a particle joins the tree and when a particle moves that the connectivity is maintained at all times t. □

Lemma 2. *At any moment t, particles in the filling area A form a directed tree T_S.*

Proof. **Claim 1.** *Particles that join the directed tree T_S maintain connectivity.*

Initially, T_S contains a single particle: the leader particle chosen during by the *leader election algorithm.* There exists a time $t' > t$ when a new particle joins T_S. We want to prove that for all times $t' > t$, connectivity will be maintained. When the *leader* l first pulls a *follower* f, the leader becomes the first *leaf* to join T_S. At this moment, f has a register *f.childDir* pointing to its child l. Every time the *leader* pulls a particle, a new particle joins T_S in a similar process. Connectivity is maintained as the expanded *leader* pulls a follower and connectivity is maintained when the *small leader* pulls another follower.

Claim 2. *Particles movement in the directed tree T_S maintain connectivity.*

From Claim 3.2, we know that particles joining the tree maintain connectivity. At some initial time t, the tree is valid as defined in Claim 3.2, we now want to prove that for all times $t' > t$ a movement will occur in the tree, and this movement will also maintain this connectivity. At some time $t' > t$, the next particle movement will occur in the tree. For the tree to disconnect, a particle in *p.state* expanded with an existing *p.parent* must contract without causing *p.parent* to expand. We know from our initial assumptions, that the contraction of a particle can only occur in a handover. Furthermore, the *follower* role which can perform a contraction outside a handover cannot exist in T_S, for reasons explained in Lemma 3.

In the previous two claims we show that when a particle joins the tree and when a particle moves in the tree, connectivity is maintained at all times t.

□

Corollary 1. *$T_I \cup T_S$ forms a single connected component linked by a single leader particle.*

Lemma 3. *When a particle occupies a new position $v \in A$, this position v will remain occupied.*

Proof. We recall that for clarity of presentation, we assume that $|P| \geq |A|$. Assume that this statement is false. At some time t, vertex v becomes occupied by some particle. For this proposition to be false, there would exist a time $t' > t$ when v becomes unoccupied. We know that when a particle contracts, by definition the tail collapses on the head position. For this scenario to occur, a particle p must be expanded while having its tail on v and subsequently contract at time t'.

In this algorithm, there exists only one role which is able to contract itself outside of a handover, this role is *follower*. However, the *follower* particle only exists in T_I. There exists only one vertex x, where $x \in V(T_I) \cap V(A)$, and this vertex is g, occupied by the leader. We know from the algorithm that any particle with either its tail or head on g is a *leader* or a *small leader*.

We assumed that it would be possible for a particle p to contract its tail away from v, but we have shown that such an operation can only be performed by a particle in a role that cannot exist in the filling area A. This contradicts our initial assumption. □

Lemma 4. *When a particle p retires, it holds that p is contracted, and that there does not exist an unoccupied position adjacent to a port label $l \in p_{port}$.*

Proof. By observing the behaviors of a *leaf* particle l, we know that a contracted leaf will scan each position adjacent to each port in *l.port*. If there at exist at least one unoccupied position which l can expand into, then l will expand into any unoccupied position, there is no pre-condition preventing l from expanding. If there does not exist an available position, then for all positions x adjacent to *p.port* there is either a particle or a boundary object. We will distinguish the possible states of position x:

Case 1: position x is occupied by a boundary object: by definition, we assumed O to be immutable, hence once x is occupied by a boundary object, it will remain occupied for the whole filling duration.

Case 2: position x is occupied by a particle: in Lemma 3, we proved that once a vertex position x is occupied by a particle at a time t, this position x will remain occupied for all $t' > t$.

□

Lemma 5. *When all particles $p \in A$ retire, there does not exist an empty region $R \subseteq A$ where there exists a vertex $v \in R$ that is unoccupied.*

Proof. Let us assume that such a region R exists. Then, there exists a configuration c_t, where there is at least one vertex $v \in V(A)$, where v is unoccupied and v must also have an adjacent vertex occupied by a particle. We will now look at time $t' < t$ when the last *leaf* particle l adjacent to v retires. At $t' - 1$, particle l was a *leaf* particle. From Lemma 4, for l to retire there must not exist a position adjacent to v that l can expand into. From our original assumptions we know that

l is adjacent to position v where l can expand to at $t' - 1$, this contradicts the assumption that l will be retired at t'. Our initial assumption that a region R exists is thus wrong. This reasoning applies for any region R in c_{final}. □

Lemma 6. *While there exists a leaf particle l, either l will retire or l will expand at a time $t' > t$.*

Proof. We recall that for clarity of presentation, we assume that $|P| \geq |A|$. Let us assume that l cannot retire. We will observe what happens when l is contracted and when l is expanded:

Case 1: l is expanded. We know from Corollary 1 that $T_I \cup T_S$ are both connected and linked by leader le. Therefore, there exists a path between any particle in T_I and le. Knowing $|P| \geq |A|$; there exists a path connecting all *follower* particles to l, this entails that l will eventually contract. We can now look and what happens to l when it is contracted in Cases 2 & 3.

Case 2: l is contracted and has an available adjacent position: l expands at its next activation, because the scheduler is fair we know this will happen and hence l makes progress by increasing the total number of particles in the layer.

Case 3: l is contracted and does not have an available adjacent position to expand into: l retires at its next activation, because the scheduler is fair we know this will happen and hence l makes progress by increasing the number of retired particles in the layer.

□

Theorem 1. *The algorithm solves the Filling Problem.*

Proof. To solve the *Filling Problem*, the particle system must reach a *legal configuration.*

The algorithm will terminate in a legal configuration which will solve the filling problem. From Lemma 6, we know the particle system progresses by increasing the total number of particles or increasing the number of retired particles in the filling area. Because the size of the filling area is constant, there exists a time t, when no more particles can fit in $|A|$. In such a case, Lemma 6 ensures that all particles $p \in A$ will eventually retire at a time $t' > t$. We now know that at time t' the particle system is *stable*, particles cannot change state or perform a move. Through Lemma 5, we know that such configuration cannot contain an empty region R. This makes $c_{t'}$ a final stable legal configuration that solves the *Filling Problem.* □

Theorem 2. *Tree formation has a running time complexity of $O(n)$ rounds.*

Proof. From Lemma 13 in [6] we know the running time tree formation is $O(n)$. We would like to highlight that this bound could be made tighter using the in-degree of edges in the tree. We define the degree of a vertex as the number of edges connecting to a vertex. In a directed tree, the in-degree of a vertex v is the number of edges coming into v. In the worst-case, the maximum in-degree of a particle is 1. In that case, only one particle transitions from inactive to active (i.e. when one particle joins the tree). □

Theorem 3. *Our algorithm has a running time complexity of $O(n \cdot R)$ rounds where R is* $\max\ d(u, v)$ *for any u, v.*

Proof. We define $d(u, v)$ as the distance separating two particles. Because u and v exist on either trees (T_I or T_S), there exists only one path between u and v, and the number of particles on that path signifies the distance between u and v. The proof of running time can be divided into two sections; the time it takes to form T_I and the time it takes for all particles $p \in P$ to join A (fill/coat the area).

For the filling problem, we have $|P|$ potential filling particles which can be in an area A. Assuming the $|A| = |P|$, then all particles must transition to the area to fill. In the worst case, throughout this filling process, there is only one *leaf* particle p_1 expanding/contracting. For p_1 to complete this contraction/expansion, the worst case is determined by the longest path of particles

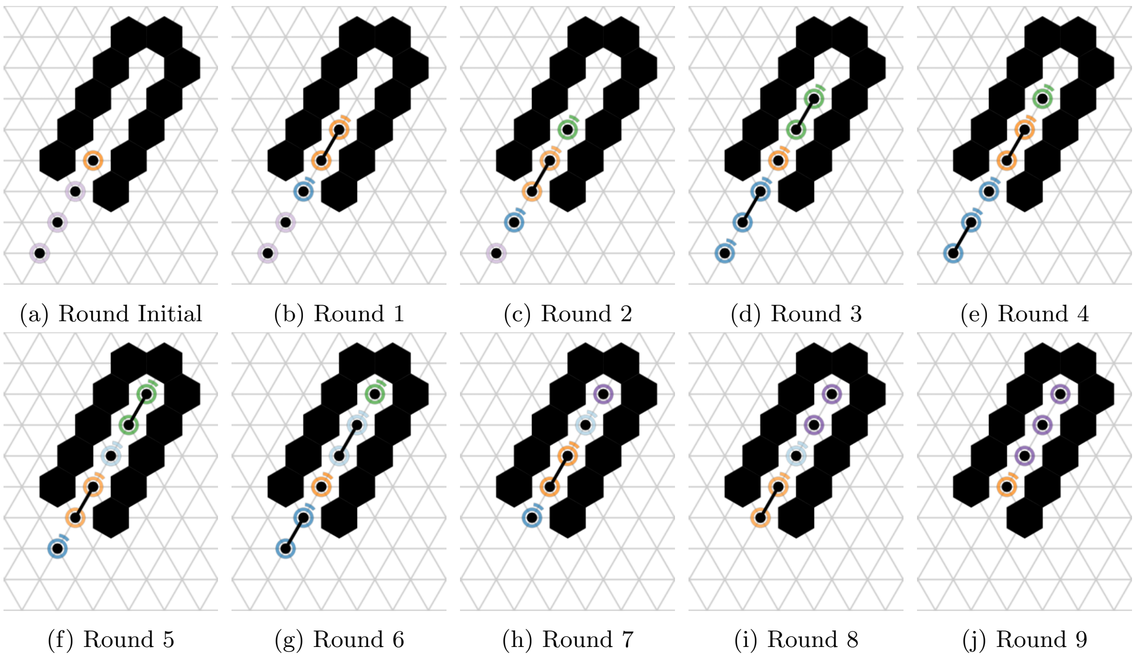

(a) Round Initial (b) Round 1 (c) Round 2 (d) Round 3 (e) Round 4
(f) Round 5 (g) Round 6 (h) Round 7 (i) Round 8 (j) Round 9

Figure 3.6: Example of an adverse scheduler choosing a bad sequence of activations

from p_1 to a leaf p_2 in T_I, if all particles are in a line, this chain is at most of size $|R|$. Hence, it would take a $O(|R|)$ for a contraction/expansion to propagate through the longest path; at round r, particle p_1 expands, all particles between p_1 and p_2 are expanded, at round $r+1$, particle p_2 contracts, at round $r+2$, $p_2.child$ can contract by pulling p_2, this repeats until p_1 can contract at round $r+|R|$. At the end of this cycle, the number of available positions in A are reduced by one. Since there are $|A| = |P|$ positions to be filled, the filling algorithm has a run time complexity of $O((|R|) \cdot n)$ to complete the filling phase.

The $O(n^2)$ rounds complexity only applies to a single input when all particles are lined up behind each other. For any other configurations, we can use $O((|R|) \cdot n)$.

□

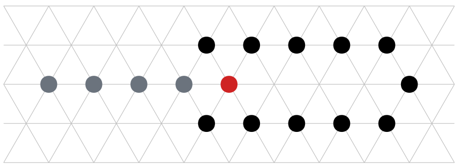


Figure 3.7: Worst case input scenario for C_0. All particles are in a line and fill a line type object, at any moment there can only be one leaf particle p and every expansion/contraction cycle of p, p will initiate a handover that will travel through all particles in the system.

3.3 Experimental Results

This section will present the experimental results of the *Filling Algorithm*. This section will explain the bridge between the theoretical analysis made in this thesis and the empirical results obtained through testing.

To test and visualize the algorithm, the *AmoebotSim* [1] software was used. From the theoretical

[1]https://amoebotsim.readthedocs.io/en/latest/

analysis of the developed algorithm, we developed the first two hypotheses. As we ran the experiment some findings led to more hypotheses to further investigate the results found. For each hypothesis a battery of tests was prepared; using different initial particle shapes and different shapes for the object.

Hypothesis 1. *Changing the shape of O while maintaining an equal-sized area A to fill yields the same running time.*

Setup. To test the impact of an object's shape, we first came up with extreme shapes, one is maximizing the diameter and another minimizing the diameter of O. We also prepared some pseudo-random shapes to see if the running time would fall in between the running times from the extreme shapes.

To maximize the diameter, we used a line object as shown in Subfigure 3.10c, and to minimize the diameter of the object, we used a hexagon object as shown in Subfigure 3.10a. In the two extreme cases, we scaled the size of the area to be filled to observe how the run time would grow with the size of the area and the shape of the object. We used an equal number of particles for each size used and ensured that the area of both shapes was equal. For each of the 16 sizes used, 10 trials were made and averaged out to yield a data point for each size. These 160 trials were run for each of the two extreme shapes.

For each of the pseudo-random shapes shown in Figure A.2 and Figure A.3, we ran a total of 4 trials. Each set of 4 trials was averaged to yield a data point.

Observations. While it was easy to compare the custom shapes performances with the line fill experiment, it is hard to compare with the hexagon fill experiment. This is because the hexagon area size is not granular enough. We can however observe in Figure 3.8 that the hexagon running times are stable enough to sketch a prediction line to estimate what its run time would be for the specific cases. What we observe is that both extreme shapes running time diverges significantly. This divergence increases at a constant rate. An interesting trend is distinguishable when looking at the run time for the pseudo-random shapes; we can see in Figure 3.9 that all pseudo-random shapes performed very similarly to the hexagon shape. A new Hypothesis 3 springs from this observation and it will explore the impact of the diameter of an object on the running time.

Hypothesis 2. *The run time of the Filling Algorithm is linked to the size of the longest path* $\max\ d(u, v)$ *for any u, v.*

Setup. To explore the hypothesis, a set we prepared a set of 64 experiments with the templates provided in Figure 3.10. Each experiment is a data point on the charts present in Figure 3.8. A total of 32 experiments were run to fill 16 different size hexagons, as seen in Subfigure 3.10a & 3.10b, and the other 16 were run on lines of matching sizes, as seen in Subfigure 3.10c & 3.10d. For both shape and each of the 16 sizes, we ran 10 trials with particles arranged in a line configuration and 10 trials with particles arranged in a hexagon configuration. The four different types of experiments are shown in Figure 3.10.

Observations. We can see from Figure 3.8 a clear distinction between filling a line and filling a hexagon. We can see a consistent improvement when the max path length is minimized in a shape by using a compact initial particle organization. The hexagon fill outperforms the line fill in every case after size 6. We can see that at size 5, line fill outperforms hexagon fill. This discrepancy can probably be due to an unlucky series of runs due to the low number of particles in the system.

Hypothesis 3. *The diameter of a shape has a linear impact on the running time.*

Setup. Since the particle system will often branch off and expansions are random, it is hard to control the size of the longest path. However, what we can control is the diameter of the shape created. We started with the smallest diameter of 1 (i.e. a line).

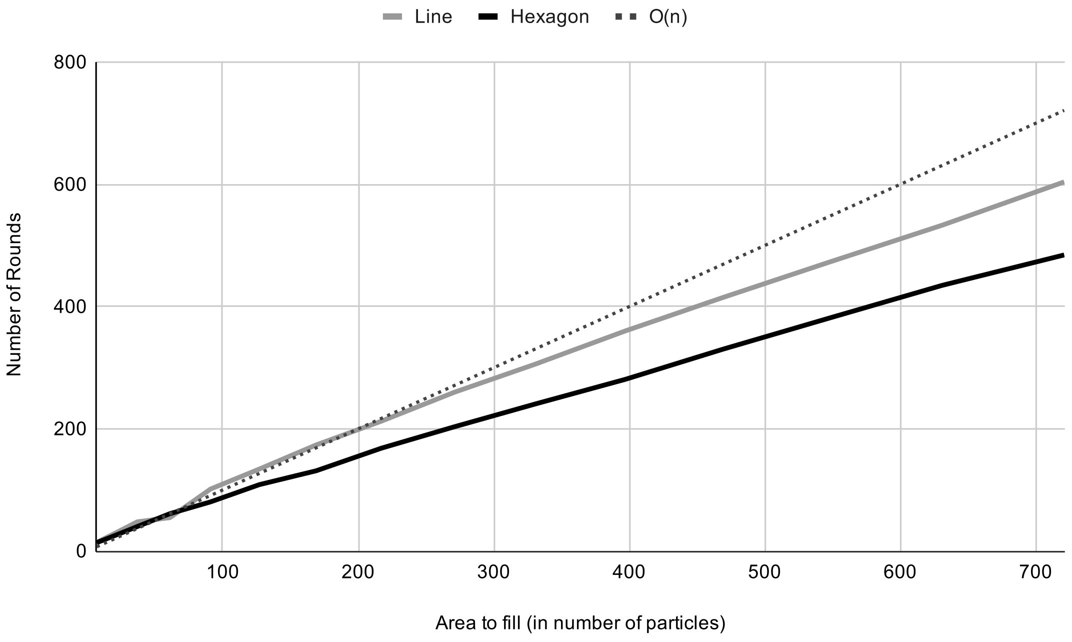
For this hypothesis, we prepared 22 experiments on shapes with a total filling area of 500 along with 500 particles. We began with a shape of diameter 1 and for each experiment, we increased the diameter of the shape by 1 until the shape became a square. Subsequent experiments past the square would yield the same results as the previous 22 experiments. For each experiment, we ran 5 trials and averaged the runs to exclude outliers.

Observations. We can see a steady decrease in the number of rounds needed to fill an area. As seen in Figure 3.11, the algorithm tends to form a wall of particles moving from the root towards the end of the shape. As the overall diameter of the shape increase, so does that wall, which minimizes the length of the longest path in the tree. It is worth noting that we did not expect to see a linear decrease in the number of rounds needed. As the overall diameter of the shape would increase, the average length of the path should have increased faster in the first few corridor sizes. As the linear decrease that we observed did not match our expectations, we did not run the rest of the battery of tests. We did however run corridor size 20 and 22 to ensure that the trend would continue. As it did follow the trend, it was safe to assume that the rest of the sizes would fall in line.

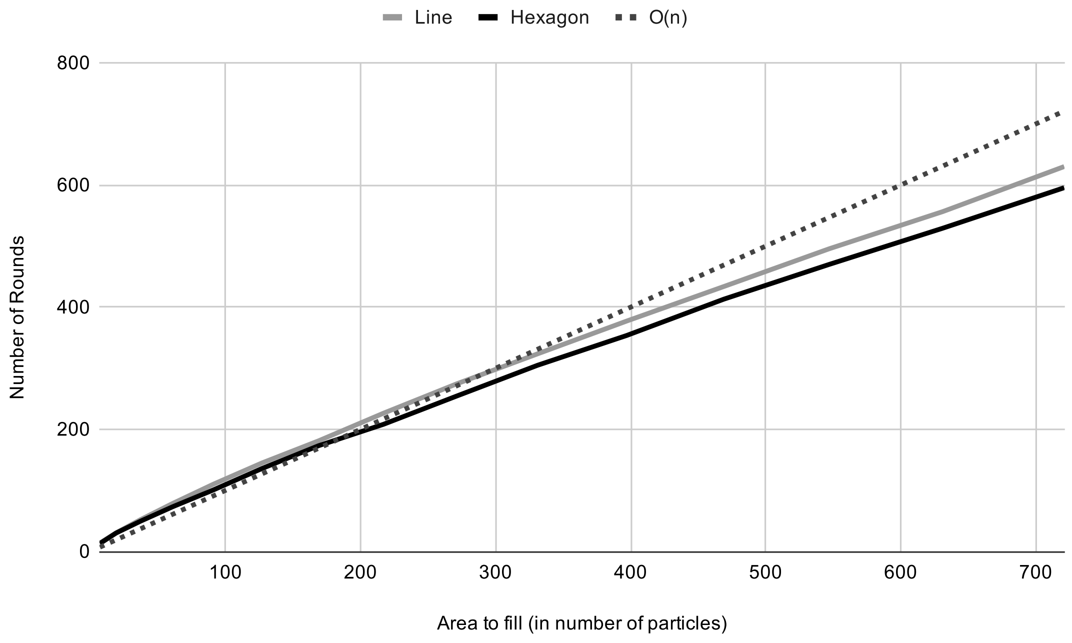
Hypothesis 4. *Increasing the total number of particles to coat an area of equal size will increase the average running time.*

Setup. To test this hypothesis we used a hexagon object of size 7, where 7 stands for the length of one side of the hexagon. This yields a total area of 139 particles. Initially, we ran the test with 139 particles, adding 25%, 50%, 100%, and finally 200% of the original number of particles.

Observations. As seen in Figure 3.13, we can see that the total number of rounds needed decreases as the total number of particles decrease. Our initial assumption was that the total number of rounds would increase but we failed to take into account the type of scheduler being used here. The simulator uses a *sequential multiple activation* scheduler which activates a particle at random until all particles were activated at least once. As the total number of particles increases, so does the probability that most particles will activate an increasing number of times.



(a) Filling a Hexagon with two different initial particle configuration, namely line and hexagon



(b) Filling a Line with two different initial particle configuration, namely line and hexagons

Figure 3.8: Comparing hexagon filling with line filling

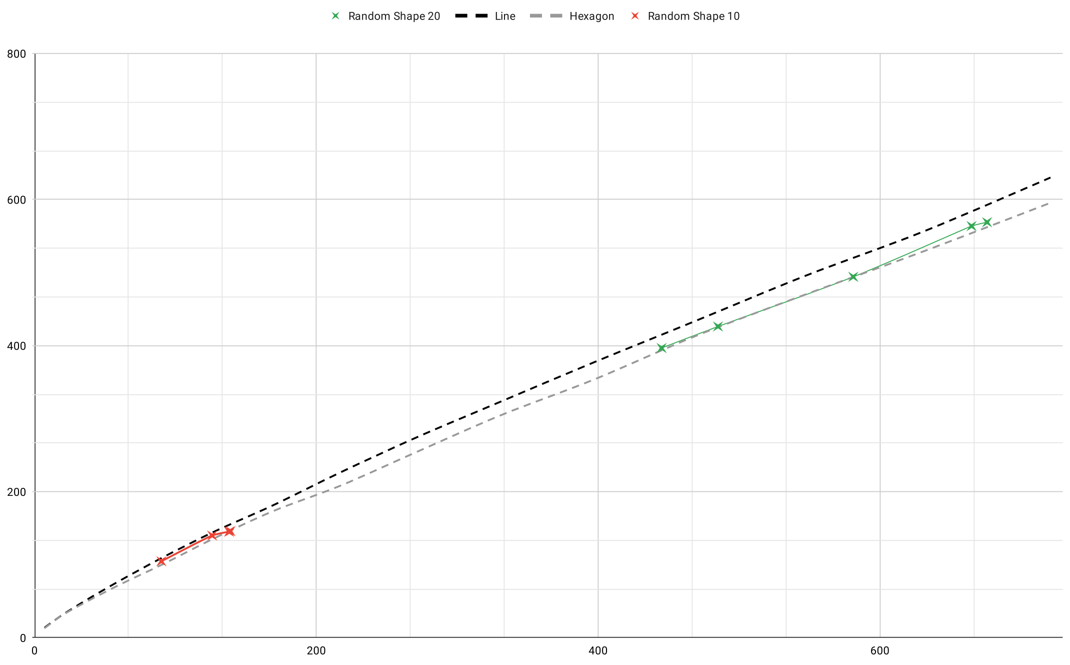


Figure 3.9: Showing pseudo random shapes of diameter 10 and diamter 20

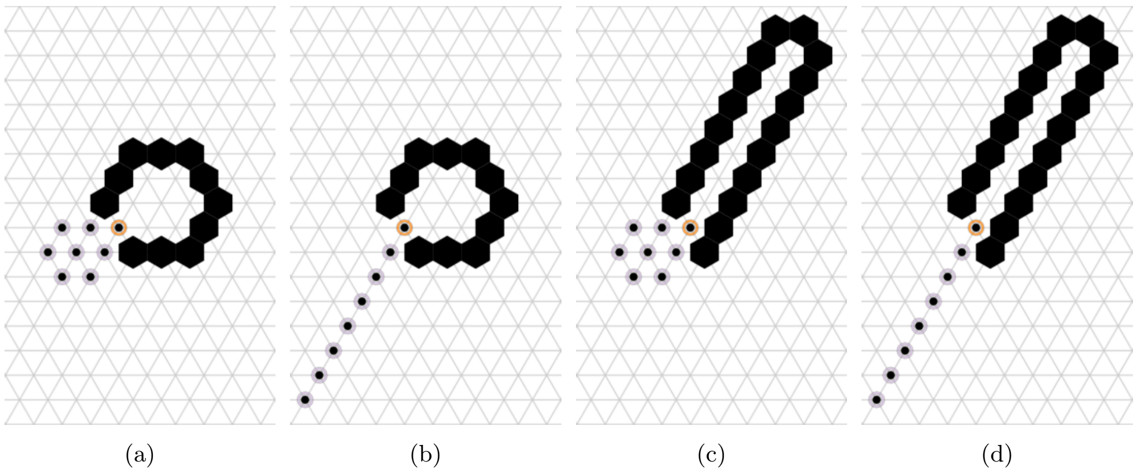


Figure 3.10: Templates used for testing, here size 2 is shown, where size is the length of the hexagon's size. Sizes 2-16 were used to makes the charts in Figure 3.8. The line's area matched the hexagon's area. Sub-Figure 3.10a and Sub-Figure 3.10b were the templates used to make the chart in Sub-Figure 3.8a. Sub-Figure 3.10c and Sub-Figure 3.10d were the templates used to make the chart in Sub-Figure 3.8b

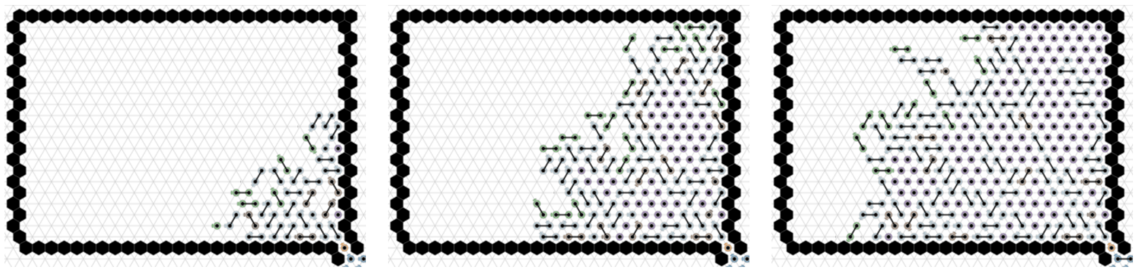


Figure 3.11: Progression of particles though a corridor. It is worth noting that particles cant extend far past the wall because if an arm of particles were to stretch out, it will branch

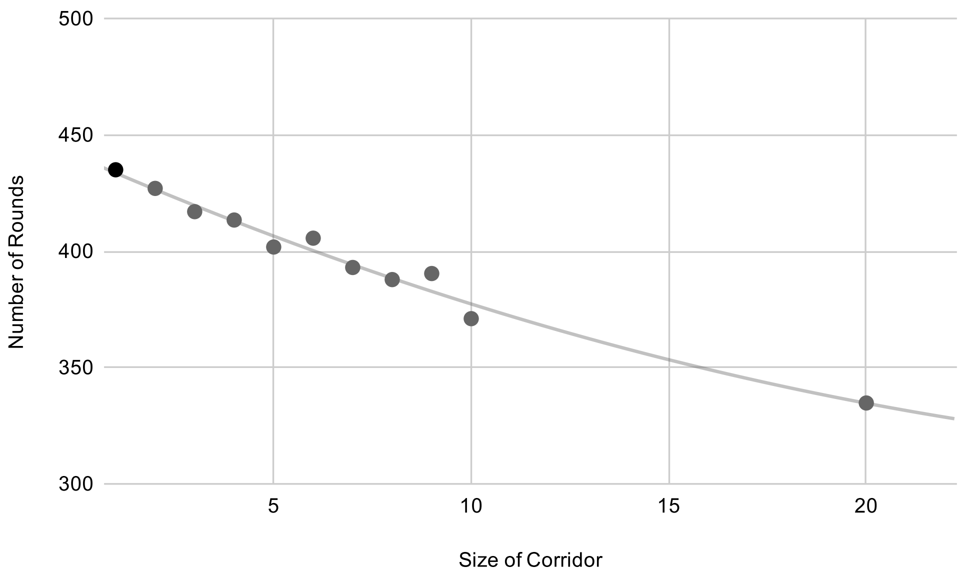


Figure 3.12: Impact of object width on run time

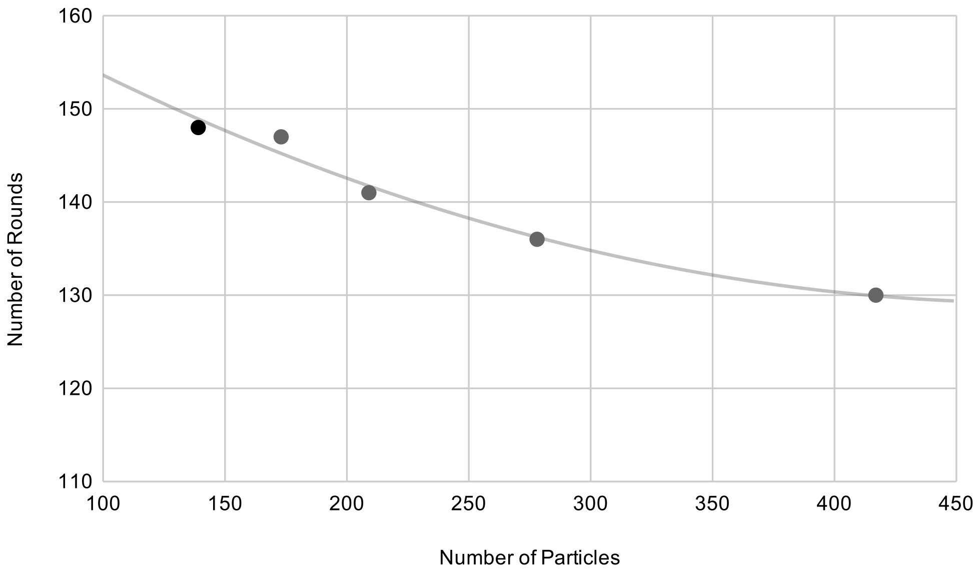
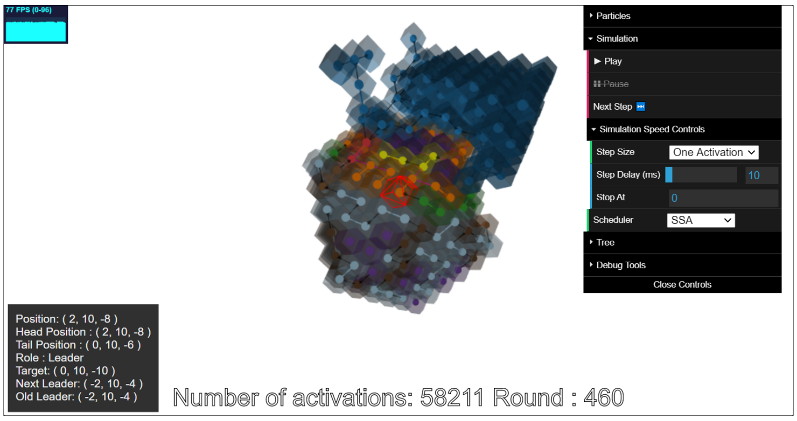


Figure 3.13: Impact of total number of particles on runtime on area of size 139. In black is shown the the baseline, where the trial was 139 particles to coat an area of 139.

Chapter 4

Simulator for 3D Amoebot Research

In this section we describe the simulator developed for the purpose of visualizing the 3D algorithmic solution to the 3D Universal Coating problem. Controls and custom interactions facilitate the use of this simulator for research under the Amoebot model.



Step Size: One Activation
Step Delay (ms): 10
Stop At: 0
Scheduler: SSA

Position: (2, 10, -8)
Head Position : (2, 10, -8)
Tail Position : (0, 10, -6)
Role : Leader
Target: (0, 10, -10)
Next Leader: (-2, 10, -4)
Old Leader: (-2, 10, -4)

Number of activations: 58211 Round : 460

Figure 4.1: Screenshot of simulator. Top right we have statistics about the performance of the system. On the right-hand side, we have the different control which enables interactions with the simulator. At the bottom, we have current information about the number of rounds elapsed as well as how many particles were activated. Lastly, in the bottom right corner we have debugging information about the particle highlighted with the wireframe.

4.1 Technologies Used

For the first algorithm, we used an already existing simulator; *AmoebotSim* [1]. This solution offers a robust simulator to work under the Amoebot model in 2D, which was satisfactory for the filling implementation. The same simulator has a 3D extension under development but in its current stage was not yet suitable for the 3D coating algorithm. The first option was to extend the simulator to work in 3D but since the project was at such an early stage, creating a new solution was preferred over extending this one.

When choosing which technology to use to implement the simulator, ease of use and accessibility was a priority. Installing the 2D simulator was a lengthy setup before one could implement a

[1]https://amoebotsim.readthedocs.io/en/latest/

prototype. Prototyping should be hassle-free and accessible to facilitate research in the field of programmable matter.

To facilitate access of the simulator to other researchers, we chose a web-based solution. The simulator was created using Three.js version *r120*, a cross-browser JavaScript 3D computer graphics library using WebGL. The advantage of a web-based simulator is that little to no setup (depending on whether or not the simulator is hosted) is required. The browser of choice can be utilized on all platforms supporting a browser [2]. For development, performance is slightly hindered due to the innate nature of web browsers not being able to maximize the performance of the hardware they run on (i.e. multi-threaded, hardware acceleration). The role of this simulator is mainly to provide visual feedback for algorithm design and compare the performances of different schedulers.

4.2 Schedulers

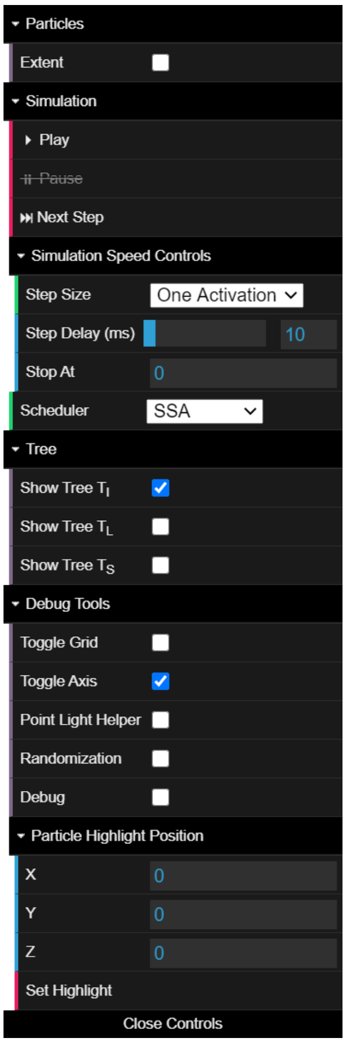
For the algorithm, a set of three schedulers were built; sequential single activation, sequential multiple activations, and sequential adversary. The sequential single activations randomly permutes the order of the particle system P at the beginning of each round r. This ensures that the number of activations after each round r is $|P|$. The sequential multiple activations activate a random particle in P and maintain a dictionary d of particles activated that round. When $|d| = |P|$, we know that each particle p was activated at least once, this marks the end of the round, the dictionary is initialized as empty and a new round starts. We cannot predict the number of activations for the round, we can monitor and record it once the round is complete. The final scheduler, sequential adversary scheduler, activates particles by attempting to maximize the total number of rounds from initial configuration c_1 to final stable configuration c_{end}. In Section 5.1, we explain how the sequential adverse scheduler chooses particles to maximize the number of rounds.

4.3 Tools

Particle & Tree. Enables the extent of a particle. While a particle in the 3D simulator is represented by a rhombic dodecahedron, it is impossible to see what is happening inside the initial set of particles when particles are tightly packed. This feature can alter the visibility of the particle's geometry. Setting a tree to visible can help visualize how the particles are linked together. While particles use a series of target markers to indicate which particles they are pointing to, the line between them enables visualization of the tree formed and facilitates pointer error detection while developing the algorithm. The tree labels are explained in Chapter 5.

Simulation. The first two options are used to start and stop the simulation. The next button runs one step of the algorithm, which corresponds to the step size chosen which can be either *one activation* or *one round*. An activation step indicates that one particle will be activated and one round step means that one round will be executed. A round is a series of activations defined by the scheduler which can be either *SSA* (*sequential single activation*), *SMA* (*sequential multiple activations*) or *adversary*. For more information about the schedulers, refer to Section 5.1. The *stop at* field allows the simulation to stop at a specific activation number, this can be useful when randomization is disabled at it allows us to observe predictable repeating behaviors. Step delay induces a delay in-between step, this can be used to fine-tune the speed at which the algorithm proceeds. It is worth noting that the speed of the simulation is capped at the computer monitor's refresh rate. For example, a monitor at 60Hz will run the simulation at 60 activations per second, this is an intended feature of Three.js as the framework will only update the scene as many times as can be displayed by the monitor. For long simulations, it is advised to run on a round by round basis for the first few rounds and then lower the speed to activation by activation basis.

[2]for supported browser check `https://threejs.org/docs/manual/en/introduction/Browser-support`



▾ Particles
Extent ☐
▾ Simulation
▸ Play
~~⏸ Pause~~
⏭ Next Step
▾ Simulation Speed Controls
Step Size One Activation
Step Delay (ms) 10
Stop At 0
Scheduler SSA
▾ Tree
Show Tree T_I ✓
Show Tree T_L ☐
Show Tree T_S ☐
▾ Debug Tools
Toggle Grid ☐
Toggle Axis ✓
Point Light Helper ☐
Randomization ☐
Debug ☐
▾ Particle Highlight Position
X 0
Y 0
Z 0
Set Highlight
Close Controls

Figure 4.2: This pictures shows all the controls exposed by the simulator

Debug Tools. Contains an array of tools needed for debugging purposes. The particle highlight is a useful tool which, at the click of a button, shows information about a particle. By pressing the ”n” key, it is also possible to activate the particle at that position indicated by the highlight (seen as the red wireframe in Figure 4.1). This feature enables the exploration of edge cases, and testing for specific activations sequences. The debug option enables or disables the particle highlight system. The toggle features are straightforward and enable the visualization of some of the elements of the simulator, such as the positions of the lights to understand the shadows; what is the direction of the axis; or having a perspective for the sizes in the simulation by enabling the grid.

Chapter 5

The Coating Algorithm

Note: This chapter will follow the same structure as Chapter 3. While both sections can be read independently, we have marked with an asterisk redundant information which can be skipped if Chapter 3 has been read.

In this section, we will be describing a solution to the Universal Coating Problem in 3D. This new solution is the *Coating Algorithm*, an algorithm created using the similar concepts as the *Filling Algorithm* presented in Chapter 3. While the *Coating Algorithm* is a generic solution to the Universal Coating Problem 3D, there are some preconditions for the initial configuration of the particle system.

In Chapter 1, we addressed some of the *Filling Algorithm* pitfalls: some 3D shapes are not mappable to 2D; one leader bottlenecking the filling process. The *Coating Algorithm* overcomes these limitations set by the *Filling Algorithm.* Firstly, the *Coating Algorithm* can use multiple leaders, if the initial configuration permits. All particles, initially on the surface, are chosen as *leader* particles and amongst them a *super leader* is designated. Secondly, the algorithm can coat multiple layers, as one layer L_i completes, particles $p \in L_i$ retire and form a new object to coat. The main challenge in this *Coating Algorithm* is to manage the multiple leaders actively coating the layer.

For the multiple leader setup to be efficient, particles need to be shared evenly between all leaders. This is a challenge of its own as the leaders are oblivious to the initial configuration of the set P of particles; particles linking to these leaders are also oblivious to the size of the subtree they are joining. We will see in the analysis the impact of this disparity.

Furthermore, we assume the set S of particles $p \in V(P) \cap V(L_1)$, where S represent the subset of initial particles adjacent to the layer L_1, is connected. This assumption enables us to use an existing *leader election algorithm* [8] to elect a super leader in S.

5.1 The Algorithm

The *Coating Algorithm* behaves very similarly to the *Filling Algorithm.* Once particles pass the leader or super leader, their behaviour is identical to the *Filling Algorithm. Leaf* particles lead the coating effort initiating a series of expansions and contractions to coat the layer. These series of handovers propagate through the tree structure, created in the earlier phases, to reach leader type particles. These leader gateway particles link the coating branches to the initial set of particles which has yet to enter the coating layer.

For a higher level overview of the algorithm, please refer to Figure 5.1 which shows the different roles of the algorithm.

The *Coating Algorithm* uses a set of three trees; one tree T_I, defined in Section 2.3, connects all *follower* particles to their respective *leader*, the second tree T_L connects the *leader* particles to the *super leader*, the third tree T_S is created as particles coat the layer ensuring that *leaf* particles can pull particles through the *leader* particles. Initially, the leader election algorithm is designed to find a leader in a 2D set of particles. For the leader election process to work with the 3D *Coating Algorithm*, a subset of particles on the surface is chosen for leader election. As assumed previously that the contact point between O, and the initial set P is connected. As the set is in 2D, the leader election algorithm from Chapter 3 can be used. However, instead of electing a *leader*, it will elect a *super leader*. We then proceed to run the *tree formation* subroutine on the particle system. The difference here is that particles on the surface can only connect to particles on the surface; instead of becoming a *follower* particle.

In essence, the coating task is accomplished by *leaf* particles which expand through the area to be filled, where for each expansion a *leaf* particle makes, it initiates a series of handovers from itself through the *leader* and onto a *leaf* particle in T_I, see Figure 5.1. Each expansion/contraction cycle completed by a *leaf* increases the area covered by the particles and reduces the size of the problem. In order to accelerate the process, some *coater* particles can branch off from their path and hence create a new *leaf* particle. Where the new *leaf* branched off, a *branch* particle sits at the bifurcation to funnel *coater* particles in either direction, see Figure 5.1. When a *leaf* particle p can not expand into any direction, all adjacent ports are occupied by either particles or the boundary of the object O, then that particle becomes *retired*. Subsequently, the parent of p becomes a *leaf*. This recursive handover of the *leaf* role ensures that no blank holes are left inside the area to be filled. Once the child of the *leader* particle has *retired*, we know that the filling process is complete. The *leader* can now become a *retired* particle and hand his role to his parent *follower* particle. Once leaders are done coating, they alter their role and become *bridge* particles. *Bridges* can move particles up and down T_L. We will discuss in the role description when and how *bridge* particles move particles in order to provide for other leaders/bridges.

When a layer is completed, the particles on the layer will use the *retired* particles of the previous layer to move toward the *super leader* of the previous layer. Once the first particle p_1 has reached the *super leader* of the previous layer, the *super leader* will hand its role to p_1, making p_1 the super leader of the new layer.

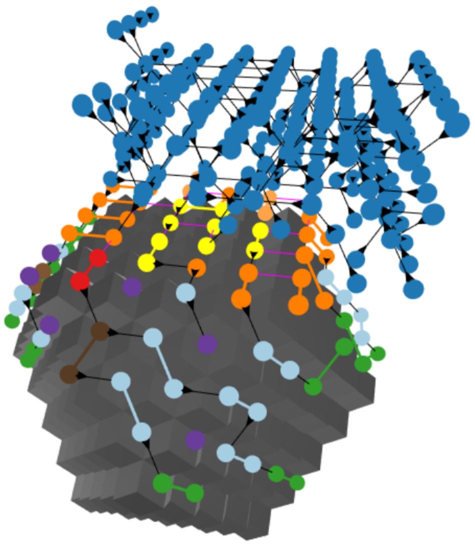


Figure 5.1: We can see all the roles of the *Coating Algorithm*. In red is the *super leader*, which is the root of tree T_L connecting all the *leader* particles in orange. The trapped leaders are converted to *bridge* particles, shown in yellow, which help other leaders complete their layer. In darker blue are the *follower* particles, representing the initial particles in T_I rooted at a leader particle. In light blue are the *coater* particles, led by the *leaf* particles in green which lead the tree T_S. There are bifurcation in T_S which marked by the *branch* particles in brown. Lastly, *retired* particles which are done coating are shown in purple.

Note: This section will follow the same structure as Chapter 3. While both sections can be read *independently, we have marked in grey the differences between both sections if Chapter 3 has been read.*

Roles. Any particle p has a finite set of roles Q, below are enumerated the possible roles in which p can be, along with a description for each of these roles. Furthermore, Figure 5.1 showcases the different roles in one example. There are two main types of roles: *static* and *dynamic*. *Static* roles are tied to a vertex v, meaning that any particle p which has its head or tail on that vertex v will have that role, and when a particle leaves that vertex, it loses that role. *Dynamic* roles are tied to a particle while a particle fulfills a set of conditions, it will keep its role during movement. Movement in this thesis will only occur through one of the two handovers: pull. An expanded particle p can only pull a contracted parent. Note that in this thesis we will mention roles in italic, this is to distinguish between the role *leaf* and the leaf of a tree in data structures. Throughout these descriptions we will be referring to memory registers to explain the roles. Memory registers for particle p will follow the following format, where $p.x$ refers to the memory register labeled x, we will also use $p.parent$ and $p.child$ to refer to the parent and child of p respectively. Combination of both usage can yield; $p.parent.role$ which refers to the role of the parent of p. We will now distinguish particle p behavior based on its role:

- **Leader** p acts as a gateway through which particles flow. A leader is not chosen by the leader *election algorithm*, but rather is a special kind of *follower*. A leader in the *Coating Algorithm* is a *follower* that is adjacent to the object O. Furthermore, a *leader* can connect to particles in the role *leader* or *super leader*. *Leader* particles participate in T_S tree formation. We will denote v as v the vertex where *leader* l joins the tree T_L. A leader will always have either its head or tail on vertex v. When an expanded particle p has its tail on v, p is a *leader* particle, and when p only has its head on v, p is a *small leader* for reasons explained in the paragraph below. When an expanded *leader* p_1, initiates a handover, pulling a contracted particle p_2, the leader p_1 will hand its role to p_2 which will be converted to a *small leader*. When the small leader contracts, it will become a *leader* particle and will not hand the *leader* role to another particle. A contracted leader l checks for $l.child$ particle, via the $p.target$ register. If $l.child.state$ is contracted and $p.target$ is *retired*, l will become a *bridge* to help other leaders finish their coaitng process. A contracted leader can also have no child, in that case $p.target$ is -1, this will trigger a special condition enabling the *leader* to expand inside the shape, and upon its next contraction will become a *leaf* particle. A *Leader* particle p will be converted to a *bridge* if $p.child.role$ is retired, or if p cant expand (e.g. surrounded by other leaders, adjacent positions occupied by particles)

  Pulling: when p pulls a particle p_1, p_1 will always be converted to a *small leader*. *Leader* p will either become a *leaf* if $p.child$ does not exist, or it will become a *coater*.

  Pulled: when p is pulled by a *leaf* or a *coater* particle, p will always remain a *leader*.

- **Small Leader** is a *static* role. *Small leader* is a precursor to the *leader* role. In Figure 3.4 we can observe what happens when the *leader* hands over its role after every pull handover. To prevent this shift, when the *leader* pulls another particle, it will convert it to a *small leader*. When a *small leader* pulls another particle, it will not hand its role over and will convert itself to a *leader*. There is a scenario when a *small leader* will convert its parent to a *bridge*. With the existence of T_L, there is a time $t' < t$ when p was a *leader*. When p was a *leader*, if p pulls a particle on the layer, then p converts a *bridge* to a *small leader*. Now back to time t when p is an expanded *small leader*, when p pulls a *follower* onto the layer, the *follower* must become a *bridge*

  Pulling: when p pulls a particle p_1, p_1 will always keep its role $p_1.role$. *Small leader* p will always become a *leader*. As discussed above, there is a scenario when p will convert P_1 to a *bridge* particle. To distinguish between both scenarios (leave $p_1.role$ as is or change $p_1.role$ to *bridge*), p will look at $p_1.leaderPointer$. If $p_1.leaderPointer$ is a valid pointer then p knows that p_1 was on the layer and hence must convert it to a *bridge*

Pulled: a *small leader* cannot be pulled because it only exists as an expanded particle.

- **Super Leader** is a *static* role. A *super leader* shares the same characteristics as the *leader*. The *super leader* is the root of T_L, T_S and T_I.

  Pulling: when p pulls a particle p_1, p_1 will always be converted to a *super small leader*. *Leader* p will either become a *leaf* if *p.child* does not exist, or it will become a *coater*.

  Pulled: when p is pulled by a *leaf* or a *coater* particle, p will always remain a *leader*.

- **Super Small Leader** is a *static* role. *Super Small leader* is a precursor to the *super leader* role and behaves in the same way as a *small leader*. The only modification is that p converts to a *super leader* when contracting. Just as for the *small leader*, there is a scenario when a *super small leader* will convert its parent to a *bridge*. With the existence of T_L, there is a time $t' < t$ when p was a *leader*. When p was a *leader*, if p pulls a particle on the layer, then p converts a *bridge* to a *super small leader*. Now back to time t when p is an expanded *super small leader*, when p pulls a *follower* onto the layer, the *follower* must become a *bridge*

- **Leaf*** is a *dynamic* role. A *leaf* is the only particle able to expand into a new position (with the exception of the first expansion of a *leader* or the super leader). *Leaf* particles are, with *bridge* particles, the only particles able to transition to a *retired* role. This ensures that a particle cannot retire while there exists an empty adjacent vertices. When a *leaf* p retires, its role is handed over to *p.parent*.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role p_1*.role*. One exception being when a *leaf* initiates a branch, its first pull will convert a *filler* into a *small branch*. *Leaf* particle p will always remain a *leaf* after the handover.

  Pulled: a *leaf* p cannot be pulled because *p.child* does not exist for any *leaf*.

- **Coater*** is a *dynamic* role. A *filler* particle only exists in T_S. When *p.state* is *expanded*, p pulls its parent if *p.parent.state* is contracted. When *p.state* is contracted, p checks whether both *p.parent.role* and *p.child.role* are *filler*, if both are *filler*, p then checks for each port in *p.port* if p can expand into any adjacent position to its ports, and finally if such a position exists, p becomes a *leaf* and expands into an available adjacent position. At the next pull initiated by the *leaf*, *p.parent* will become a *small branch*.

  Pulling: when p pulls a particle p_1, p_1 will always keep its role p_1*.role*. *Coater* particle p will always remain a *coater* after the handover.

  Pulled: since *Coater* particles only exist in T_S, a *coater* particle p can be pulled by a *branch* or a branching *leaf* such that p will become a *small branch*. *Coater* particle p can also be pulled by another *caoter* , *leaf* or *small branch*, these handovers will not alter *p.role*.

- **Follower*** is a *dynamic* role. A *follower* is a particle in T_I. Tree T_I is created by the *tree formation algorithm* rooted at the *leader* particle. Any *follower* $f \in T_I$ is connected to the super leader particle l. When *p.state* is expanded, f can either pull on *f.parent*, if such a parent exists, else f can contract if *f.parent* does not exist, as it is a leaf in T_I.

  Pulling: when p pulls a particle p_1, p_1 will always be *follower* and p_1 will always keep its role $p1$*.role*. *Follower* particle p will always keep its role after the handover.

  Pulled: since *follower* particles only exist in T_I, a *follower* particle p can be pulled by a *leader* such that p will become a *small leader*. Particle p can also be pulled by another *follower*, where it will keep its own *p.role* role.

- **Branch*** is a *static* role: Similarly to the *leader* role, the *branch* role is linked to a vertex v in the graph. The *branch* role marks a bifurcation in the tree T_S. Each *branch* marks the beginning of the paths for two *leaf* particles, these two paths are remembered using one memory register, *b.target* which marks the port adjacent to either particle at the beginning of either path. For a *branch* particle b, when either particle at *b.target* retires, p becomes a *filler* and keeps the *b.target* register pointing to the remaining active particle. Similar to the *leader*, when a *branch* pulls *b.parent*, *p.parent.role* will be *small branch.* This prevents the *branch* from shifting from its position v.

Pulling: when p pulls a particle p_1, p_1 will always become a *small branch. Branch* particle p will always become a *filler* after the handover.

Pulled: since *filler* particles only exist in T_S, a filler particle p can be pulled by a *branch* or a branching *leaf* such that p will become a *small branch.* When particle p is pulled by a *small branch*, it will keep its role after the handover.

- **Small Branch*** is a *static* role: Small branch is a sub role of the *branch* role. In order to prevent the *branch* from shifting position, when a *branch* pulls another particle p_1, $p_1.role$ will become *small branch.* We can see in the *small leader* description an explanation for why that is the case.

  Pulling: when p pulls a particle p_1, p_1 will always become a *small branch. Branch* particle p will always become a *filler* after the handover.

  Pulled: a *small branch* cannot be pulled because it only exists as an expanded particle.

- **Bridge** is a *static role.* When a *leader* or a *super leader* completes its coating process (marked when *p.child* is in *p.role retired*), p becomes a *bridge* particle. A *bridge* hands his *follower* particles down T_L. If p does not have any *follower* particle, it will wait until either its parent becomes a *bridge* and feeds p particles, or it will wait for *p.child* to become a reverse *bridge.* A reverse bridge is a *bridge* that sends particles up T_L. A *bridge* particle p only reverses: if *bridge* p is a child in T_L, and p has *p.parent.role follower*; if all parents of *bridge* p in T_L are reversed *bridges* or *retired.* To check if a bridge is reversed, a particle will check if *p.leaderPointer* matches a *p.target.* A *bridge* can retire in one of two scenarios: when it is a child in T_S and it does not have *p.parent.role follower* or if its parent in T_L has retired.

  Pulling: when p pulls a particle p_1: if $p_1.role$ is *follower*, the $p_1.role$ will be *small bridge*; if $p_1.role$ is *bridge*, then p_1 will keep its role. *Bridge* particle p will always become a *bridge* after the handover.

  Pulled: When a bridge is pulled by another *leader*, it is converted to a *small leader*

- **Small Bridge** is a *static role. Small bridge* is a sub role of the *bridge* role. In order to prevent the *bridge* from shifting position, when a *bridge* pulls another particle, it is converted to a *small branch.*

  Pulling: when p pulls a particle p_1, p_1 will always become a *small branch. Branch* particle p will always become a *filler* after the handover.

  Pulled: a *small bridge* cannot be pulled because it only exists as an expanded particle.

- **Retired** : When a *leaf* can not expand in any position adjacent to *p.ports*, then p retires. When a particle retires, its local memory is cleared and no further transition can occur. *Retired* is a stable role. Furthermore, a *retired* particle will keep its *p.leaderPointer* flag, this flag will be used by particles in the next layer L_{i+1}.

Memory. Any particle p has a constant memory. A particle p can read the information in its own memory as well as read/write any adjacent particle's local memory. We will describe the different memory registers held by a particle p:

State*: *p.state* will either return *contracted* or *expanded.*

Role*: *p.role* will return the role of a particle p. A particle p must have exactly one role $r \in Q$. Most roles remain similar to the ones presented in the *Filling Algorithm.*

Port*: *p.ports* is an array that uniquely labels each port of particle *p*. When *p.state* is *contracted*, *p.port* will return 12 uniquely labeled ports and when *p.state* is *expanded*, *p.port* will return 22 uniquely labeled ports (it is worth noting that, when *p* is expanded, 4 pairs of ports point to the same adjacent vertex).

Target: $p.target[i]$ is a fixed array used to save the *port* pointing to the particle p_1 that p is following. Particle p_1 is considered the child of p and if a particle p_2 has a $p_2.target[i]$ pointing to p, we call p_2 the parent of p. *Branch* and *Bridge* particles have more than one target and use both slots of Target to keep track of the bifurcation in their respective tree.

LeaderDir: *p.leaderDir* is exclusively used by *leader*, *super leader* and *bridge* particles. This flag is used to know which leader is next in-line when *p* becomes a *bridge* or $follower$. We will discuss in the *Bridge* role how this flag is used.

OldLeaderDir: *p.oldLeaderDir* is exclusively used by *leader*, *super leader* and *bridge* particles. This flag is used in conjunction with the *LeaderDir* regisry to know which leader is next in-line when *p.state* is expanded. We will discuss in the *Bridge* role how this flag is used.

Scheduler. For the 3D Simulator a set of 3 schedulers were used; sequential multiple activations, sequential single activation and adversary. The sequential multiple activations (used in the *Amoebot* simulator) activates a single particle $p \in P$ at random for each activation and does so until each particle p has been activated at least once. In the sequential single activation scheduler, at the beginning of each round all particles are shuffled in a random order, this means that each particle gets activated exactly once each round. The adversary scheduler aims to maximize the total number of rounds from initial configuration to final configuration. In this process, the adversary scheduler begins by activating all particle $p \in P$ where *p.state* is *contracted*, there is one exception, when *p.role* is in role *leaf* we check the following condition; if *p.parent.state* is contracted and p cant expand then first activate *p.parent* then p. Next, for all particles $p \in P$ where *p.state* is *expanded*, if p does not have a child or its child is in state *contracted*, then we activate p, else if p has a child p_1 where $p_1.state$ is expanded, we place these expanded particles in a queue $q = \{p, p_1, ..., p_n\}$ until we find a particle p_n where p_n either does not have a child (i.e. is a *leaf* particle) or p_n has a contracted child. We then activate all the particles in q in reverse order (i.e. from the last particle in the queue to the last particle in the queue). This ensures that only one particle in q makes progress. In reality, programmable particles operate asynchronously in look-compute-move cycles. Particles wake up, most often multiple at a time, to perform their routine and repeat their cycle after it has finished. Under this assumption, the differences in particle activation would be at most $t_{diff} = 2 \cdot \lfloor t_{max}/t_{min} \rfloor$ where t_{max} and t_{min} are the the maximum and minimum activation times respectively. Under this behaviour it is unlikely that particles' activations vary significantly in each round. When the total number of particles is small, then this behaviour is closely modeled by the sequential multiple activation scheduler. However, as the total number of particles in P increases, the sequential multiple activation scheduler increases the frequency that some number will be picked multiple times. Past a certain total number of activations, the sequential single activation better represents real particle activations.

Algorithm 3 Contracted Particle Activation vs Expanded Particle Activation

Particle is Contracted

```
switch Role do
    case Coat
        Identical

    case Brch
        Identical

    case Leaf
        Identical

    case Lead
        if child is Retd then
            particle becomes Bridge
        else if child is does not exist then
            Expand

    case SupLead
        if child is Retd then
            particle becomes Bridge
        else if child is does not exist then
            Expand

    case Bridge
        if child in $T_S$ is Retd then
            if parent in $T_I$ is Retd then
                if parent in $T_L$ is Retd then
                    particle becomes Retd
            else
                if all parent in $T_L$ is Retd or is Bridge reverse then
                    particle becomes Bridge reversed
```

Particle is Expanded

```
switch Role do
    case Coat
        Identical

    case Brch
        Identical

    case Leaf
        Identical

    case Lead
        pull a ready parent
        parent becomes a SLead
        if particle has no child then
            particle becomes Leaf
        else
            particle becomes Coat

    case SupLead
        pull a ready parent
        parent becomes a SSupLead
        if particle has no child then
            if all parent in $T_L$ are Retd then
                particle becomes Retd
            else
                particle becomes Leaf
        else
            particle becomes Coat

    case SSupLead
        if particle has no parent then
            Contract
            particle becomes Lead
        else
            pull a parent
            particle becomes SupLead

    case SBridge
        if particle has no parent then
            Contract
            particle becomes Lead
        else
            pull a parent
            particle becomes Bridge
            target adjacent Leader
    case SBrch
        if particle has a ready parent then
            pull parent
            particle becomes Brch

    case SLead
        if particle has no parent then
            Contract
            particle becomes Lead
        else
            pull a parent
            particle becomes Lead
```

5.2 Proof of Correctness

In this section, we will describe the building block required to prove the correctness of the algorithm. We will begin by proving that the system is connected and that connectivity is maintained as the system progresses. We will then prove that the progress reduces the size of the problem. Finally, we will prove that this progress will lead to a final state, and when the algorithm terminates, it finds a solution to the *Coating Problem.*

We will denote $V(\cdot)$ as the vertex representation, likewise for $E(\cdot)$ with edge representation, we will denote $|\cdot|$ as the size. The particle system P will coat layers L on object O. To differentiate between the particles in the initial blob and particles actively coating the layers L, we use T_I and T_S respectively. The *tree formation algorithm* forms the tree T_I, used by *follower* particles to flow through the gate leader particles. All *leader* particles are connected together in T_L The *super leader* particle always resides on vertex g where $g \in V(T_I) \cap V(T_S) \cap V(T_L)$. As *follower* particles flow through the gate particles, they enter the layer L to be coated. *Leaf* particles direct this expansion into the layer and as they do, they form T_S, rooted at the super leader.

In the *Coating Algorithm* only the *pull* handover is used. A particle p can only pull *p.parent* if such a parent exists, and *p.parent.state* is contracted and *p.parent* is adjacent to a port in *p.port*. Movement without handover is only possible with expansion for certain roles.

For the following lemma will assume that $|P| \geq |L_i|$.

Lemma 7. *At any moment t, particles in the initial form a directed tree T_I.*

Proof. **Claim 1.** *Particles that join the directed tree T_I maintain connectivity.*

In order to prove this Lemma, we will use a similar structure to Claim 1 in Lemma 2 from this paper [7]. At some time t, the tree is valid with a single particle: the super leader particle. We will show that this property of the graph will hold at $t+1$. Suppose that at $t+1$, a particle f becomes active. If f is adjacent to another active particle p_1, f will set its register *f.childDir* to the port adjacent to p_1 if it is a *follower*, else it will set its port *p.leaderDir*. As a parent of p_1, f extends the tree as a new leaf. At time t, T_I was a tree, at time $t' > t$, f joins the graph, and maintains connectivity and the tree property.

Claim 2. *Particles movement in the directed tree T_I maintain connectivity.*

From Claim 3.2, we know that particles joining the tree maintain connectivity. At some initial time t, the tree is valid as defined in Claim 3.2. We now want to prove that for all times $t+1$, a movement will occur in the tree, and this movement will also maintain this connectivity. At some time $t' > t$, the first particle movement will occur in the tree. For the tree to disconnect, a particle in *p.state* expanded with an existing *p.parent* must contract without causing *p.parent* to expand. We know from our initial assumptions that the contraction of a particle can only occur in a handover. There is however one exception to this rule, the *follower* role can contract its tail under certain conditions. Only when a follower particle f does not have a parent, can a *follower* contract. This condition ensures that no parent is disconnected from f.

In the previous two claims we show that when a particle joins the tree and when a particle moves in the tree, connectivity is maintained at all times t. □

Lemma 8. *At any moment t, particles on the coating layer L_i form a directed tree T_{S_i} rooted at the super leader.*

Proof. **Claim 1.** *Particles that join the directed tree T_{S_i} maintain connectivity.*

Initially, T_{S_i} contains a single particle: the super leader, a particle chosen during by the *leader election algorithm.* There exists a time $t' > t$, when a new particle joins T_{S_i}. We want to prove that for all time $t' > t$, connectivity will be maintained. During the tree formation phase, new leaders join T_I. We have shown in Lemma 7, that these leaders are connected to the *super leader.* When the *leader* l first pulls a *small leader* s, the leader becomes the first leaf to join T_{S_i}. At this moment, s has a register *s.childDir* pointing to its child l. Every time the *leader* pulls a particle, a new particle joins T_{S_i} in a similar process.

Claim 2. *Particles movement in the directed tree T_{S_i} maintain connectivity.*

From Claim 3.2, we know that particles joining the tree maintain connectivity. At some initial time t, the tree is valid as defined in Claim 5.2, we now want to prove that for all times $t+1$, a movement will occur in the tree, and this movement will also maintain this connectivity. At some time $t' > t$, the first particle movement will occur in the tree. For the tree to disconnect, a particle in *p.state* expanded with an existing *p.parent* must contract without causing *p.parent* to expand. We know from our initial assumptions that the contraction of a particle can only occur in a handover. Furthermore, the *follower* role which can perform a contraction outside a handover cannot exist in T_S, for reasons explained in Lemma 3.

In the previous two claims we show that when a particle joins the tree and when a particle moves in the tree, connectivity is maintained at all times t.

□

Lemma 9. *At any moment t, leader particles on the coating layer L_i form a directed tree T_{L_i} rooted at the super leader.*

Proof. **Claim 1.** *Particles that join the directed tree T_{L_i} maintain connectivity.*

Initially, T_{L_i} contains a single particle: the super leader, a particle chosen during by the *leader election algorithm.* There exists a time $t' > t$ when a new particle joins T_{S_i}. We want to prove that for all times $t' > t$, connectivity will be maintained. We proved in Lemma 5.2 that every time a *leader* joins it is connected to the *super leader* via T_I. It thus holds, that it is also connected to the *super leader* in T_L

Claim 2. *Particles movement in the directed tree T_{L_i} maintain connectivity.*

From Claim 5.2, we know that particles joining the tree maintain connectivity. At some initial time t, the tree is valid as defined in Claim 5.2, we now want to prove that for all times $t+1$, a movement will occur in the tree, and this movement will also maintain this connectivity. At some time $t' > t$, the first particle movement will occur in the tree. For the tree to disconnect, a particle in *p.state* expanded with an existing *p.parent* must contract without causing *p.parent* to expand. We know from our initial assumptions that the contraction of a particle can only occur in a handover. Furthermore, the *follower* role which can perform a contraction outside a handover cannot exist in T_L, for reasons explained in Lemma 3.

In the previous two claims we show that when a particle joins the tree and when a particle moves in the tree, connectivity is maintained at all times t.

□

Corollary 2. *$T_I \cup T_S \cup T_L$ form a single connected component linked by a single super leader particle.*

Lemma 10. *When a particle occupies a new position $v \in L$, this position v will remain occupied.*

Proof. Assume that is statement is false. At some time t, vertex v becomes occupied by some particle. For this proposition to be false, there would exist a time $t' > t$ when v becomes unoccupied. We know that when a particle contracts, it is always the case that the tail collapses on the head position. For this scenario to occur, a particle p must be expanded while having its tail on v and subsequently contract at time t'.

In this algorithm, there exists only one role which is able to contract itself outside a handover, this being the *follower* role. However, the *follower* only exists in T_I. There exists only one vertex x where $x \in V(T_I) \cap V(A)$, and that vertex is the vertex g occupied by the *leader.* We know from the algorithm that any particle with either its tail or head on g is a *leader* or a *small leader.*

We assumed that it would be possible for a particle p to contract its tail away from v, but we have shown that such an operation can only be performed by a particle in a role which cannot exist in the filling area A. This contradicts our initial assumption. □

Lemma 11. *When a particle p retires, it holds that p is contracted, that there does not exist a port label $l \in p_{port}$ for which p can expand to;*

Proof. By observing the behaviors of a *leaf* particle l, we know that a contracted leaf will scan each position adjacent to a port in *l.port*. If there is a position in which l can expand, then l will expand into that position, and there are no pre-conditions preventing l from expanding. If there does not exist an available positions then for all positions adjacent to a port in *p.port* there is either a particle or a boundary object:

Case 1: position x is occupied by a boundary object: by definition we assumed O to be immutable, hence, once x is occupied by a boundary object, it will remain occupied for the whole filling duration.

Case 2: position x is occupied by a particle: in Lemma 3, we have proven that once a vertex position x is occupied by a particle at a time t, this position x will remain occupied for all $t' > t$.

□

Lemma 12. *When all particles $p \in L$ retire, there does not exist an empty region $R \subseteq L_i$ where there exists a vertex $v \in R$ that is unoccupied.*

Proof. Let us assume that such a region R exists. Then there exists a configuration c_t, where there is at least one vertex $v \in V(A)$ where v is unoccupied and and v must be adjacent to a position occupied by a particle. We will now look at time $t' < t$, when the last leaf particle l adjacent to v retires. At $t' - 1$, particle l was a leaf particle. From Lemma 4, for l to retire there must not exist a position adjacent to *l.ports* that l can expand into. From our original assumptions, we know that l is adjacent to position v where l can expand to at $t' - 1$, this contradicts the assumption that l will be retired at t'. Our initial assumption that a region R exists is thus wrong. This reasoning applies for any region R in c_{final}. □

Lemma 13. *Leaders and super leader will have enough particle to coat L_i.*

Proof. We will assume that $|P| > |L_i|$. To prove this Lemma we will be using Figure 5.2. We can distinguish three trees namely, X, Y and Z. We will denote $|\cdot|$ to be the sum of *follower* and *leader* particles in the subtree.

Let us assume that the black particle in Figure 5.2 is the *super leader*. For the sake of simplicity, we will assume that a particle has two parents in T_L. Where X and Y are subtrees rooted at the parents of the *super leader*. We will show that the proof can expand to any number of children (bounded by the number of ports a particle has).

We will prove that enough *follower* particles exist in either 3 subtrees such that the *super leader sl* will eventually contract.

If Z has enough particles for *sl* to finish coating, then we are done. Else;

Case 1: Neither X nor Y has enough particles for *sl* to finish coating its coating process. This contradicts our original assumption that $|P| > |L_i|$ because $|X| + |Y| + |Z| < |L_i|$. Initially, all three subtrees represent all the particles in the system so $|X| + |Y| + |Z| = |P|$. Hence, it must hold that either Case 1 or Case 2 holds.

Case 2: X has enough *follower* particles. If X has enough *follower* particles, then we can repeat the scheme of Figure 5.2 to the root of X: claiming that the root will have enough particles in either of the 3 subtrees. We can repeat this process until we reach a leaf in X in which case, we know that there will be enough particles. As *leaders* in X finish their coating process, they become *bridge* particles. According to the *bridge* role description, *bridge* particles can direct *followers* up the tree. When a *bridge* sends particles up the tree, we call it a reverse *bridge* particle, a *bridge* can reverse if: it is a leaf in T_L and still has *follower* particles as parent(s);or all *bridge* parents in T_L are reversed. Once the root of X becomes a reverse bridge, the bridge at the super leader will pass particles down to Y continue the coating process.

Case 3: Y has enough *follower* particles. The same argument as Case 1 can be made.

□

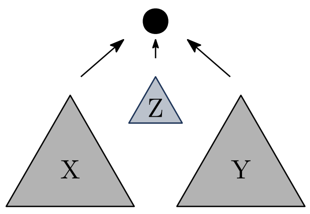


Figure 5.2: Particle p with its subtrees. The grey subtree represents the children of p in T_L. The blue subtree represents the *follower* particles of p

Lemma 14. *While there exists particles where l is a leaf in T_S, either l will retire or l will expand at a time $t' > t$.*

Proof. For this proof, we assume that $|P| \geq |A|$. Let us assume that l cannot retire. We will observe what happens when l is contracted and when l is expanded:

Case 1: l is expanded. From Lemma 13, we know that a leader will have enough *follower* particles to enable l to contract until the layer is complete. We can now look and see what happens to l when it is contracted in Cases 2 & 3.

Case 2: l is contracted and has an available adjacent position to expand into: l expands at its next activation, because the scheduler is fair we know this will happen and hence l makes progress.

Case 3: l is contracted and does not have an available adjacent position expand into: l retires at its next activation, because the scheduler is fair we know this will happen and hence l makes progress.

□

Theorem 4. *The algorithm solves the Coating Problem.*

Proof. To solve the *Coating Problem*, the particle system must reach a *legal configuration.*

The algorithm will terminate in a legal configuration which will solve the coating problem. We know the algorithm will terminate once the the *leader* particle retires. The root of the tree will retire once all particles in the coating layer have contracted and retired.

From Lemma 14, we know that the particle system progresses by increasing the total number of particles in the current coating layer. Because the size of the coating layer is constant, there exists a time t when no more particles can fit in $|L|$. In such a case Lemma 14 ensures that all particles $p \in L$ will eventually retire at a time $t' > t$. We now know that $c_{t'}$ is a final and *stable* configuration. Through Lemma 12, we know that such a configuration cannot contain an empty region R. This makes $c_{t'}$ a final stable legal configuration that solves the *Coating Problem* □

Theorem 5. *Tree formation has a running time complexity of $O(n)$ rounds.*

Proof. In the worst-case, the maximum in-degree of a particle is 1. In that case, only one particle transitions from inactive to active (i.e. when one particle joins the tree). □

Theorem 6. *Our algorithm has a running time complexity of $O(n \cdot R)$ rounds where R is $maxd(u, v)$ for any $u, v \in P$.*

Proof. The proof of running time can be divided into two sections; the time it takes to form T_I and the time to takes for all particles $p \in P$ to join A (fill/coat the area).

For the coating problem we have $|P|$ potential coating particles which can be in in an layer L. Assuming that $|L| = |P - 1|$, then all particles must transition to the layer to coat. In the worst case, throughout this coating process there is only one *leaf* particle p_1 expanding/contracting. For p_1 to complete this contraction/expansion cycle, the worst case is determined by the longest path of particles from p_1 to a leaf p_2 in T_I, if all particles are in a line, this chain is of size n. Hence, it would take a $O(n)$ to complete a cycle; at round r particle p_1 expands, all particles between p_1 and p_2 are expanded, at round $r+1$ particle p_2 contracts, at round $r+2$, p_2.*child* can contract by pulling p_2, this repeats until p_1 can contract at round $r+n$. At the end of this cycle, the number of available positions in A has reduced by one. The coating algorithm has a run time complexity of $O((|P-1|) \cdot n)$ equivalent to $O(n^2)$ rounds to complete the coating phase. The size of the handovers made by the bridge to shift particles in T_L does not impact the running time. As the size of T_S increase, so does the number of particles already on the layer (reduces the size of the problem).

The $O(n^2)$ rounds complexity only applies to a single input, we can make this bound tighter by taking into account the longest chain for any c_t. $\square$

5.3 Experimental Results

In order to test the *Coating Algorithm* we will be using the simulator presented in Chapter 4. This section will explain the bridge between the theoretical analysis made in this thesis and the empirical results obtained through testing.

Hypothesis 5. *The 2D Filling Algorithm performance should be similar to the 3D Coating Algorithm under similar conditions (SMA scheduler and one leader).*

Setup. To test this hypothesis we used an arbitrary shape for the object and organized the initial particles in a compact fashion. We set the initial set of particles such that only one leader would emerge (closely matches the situation in the 2D filling algorithm). We also used the same SMA scheduler as the 2D filling algorithm.

Observations. As seen in Figure 5.3, the 3D algorithm slightly outperforms the 2D filling algorithm. The difference is most likely due to the fact that a leader in the 3D layer can expand in any direction while the leader in the 2D is limited by the bounds of the object and the initial set of particles behind in the same dimensions. This on average, increases the path that particles must travel.

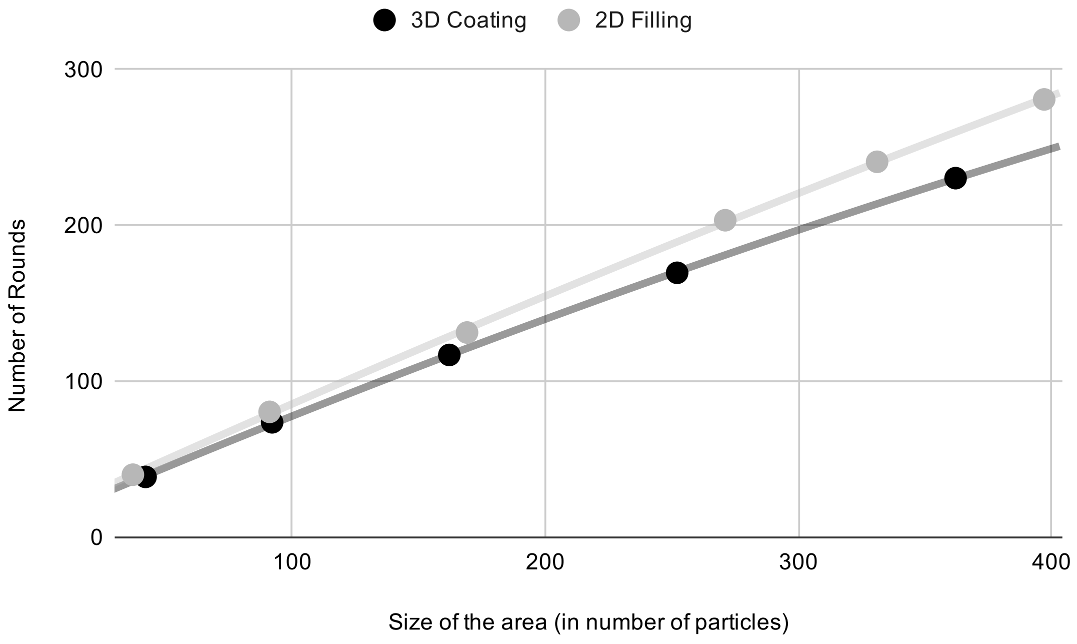


Figure 5.3: 2D Filling algorithm vs 3D Coating Algorithm while using similar initial parameters

Hypothesis 6. *Increasing the number of leaders will improve the average running time.*

Setup. To test this hypothesis, we will use a rhombic dodecahedron object of size 4. For each test we will increase the total number of leaders allowed in the initial set of particles.

Observations. As the number of leader increases beyond 8, it is worth noting that some leaders can trapped in between other leaders. This causes the efficiency of leaders to decrease as some leaders cannot contribute a single particle to the layer. This is reflected by the trend line of Figure 5.4 where we can see that as the number of leaders increases in the first few steps, the number of rounds greatly decreases. Beyond 9 leaders, the improvements seem marginal. When the number of leaders is 9, there are 8 boundary leaders, when there are 16 leaders there are 12 boundary leaders and when there are 25 leaders there are 16 boundary leaders. While the number of boundary leaders accounts for some of the decrease in performance, we can note that between 9 total leaders and 25 total leaders, the amount of boundary leaders differs by 8, while on

Figure 5.4 we see close to no difference. This would suggests that another factor is bottle-necking the performance of multiple leaders.

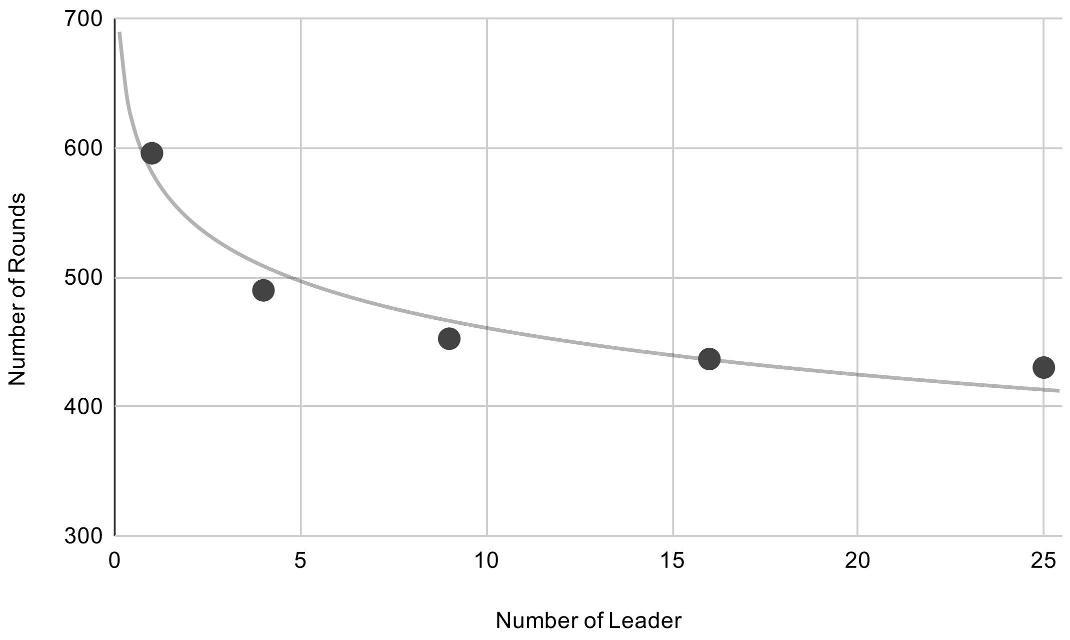
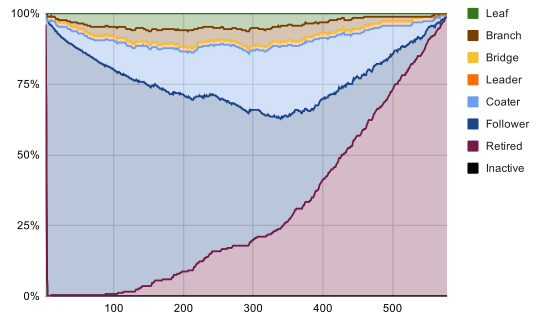


Figure 5.4: Multiple leaders coating a rhombic dodecahedron object shape of size 4 using an initial set of particles in orthotope configuration. The area is of the layer is of size

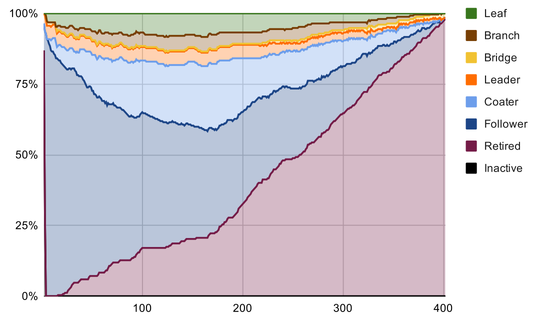
Hypothesis 7. *Increasing the number of boundary leaders improves the average running time*

Setup. We denote boundary leaders as leaders which are not surrounded on all side by other leaders. To test this hypothesis, we will be using the same setup as Hypothesis 6. But we will explore different metrics of the particle system.

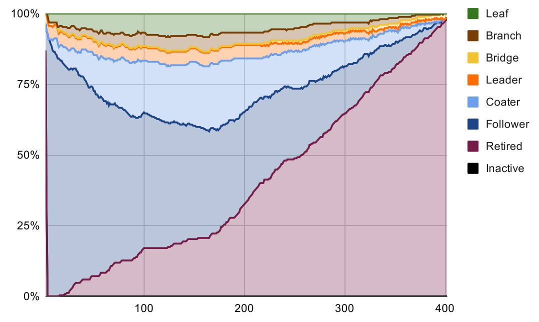
Observations. From Figure 5.5 we can observe in the 25 leaders case that the number of branches in the first 100 rounds is very low in comparison to the previous cases. One could suggest that this could be due the boundary leaders saturating the layer as they all attempt to expand, this in turn limits the possible branching of the path and increases the average length of the path on the layer. This limitation could be the second factor to the few performance gains from adding more leaders to the layer.



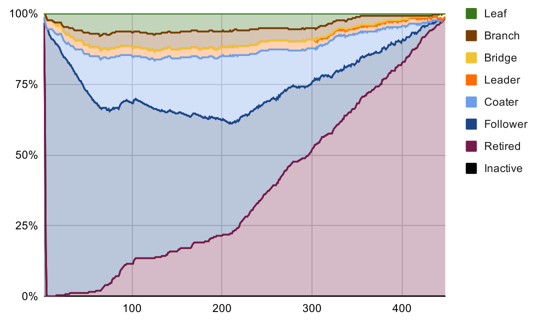
(a) 1 Leader



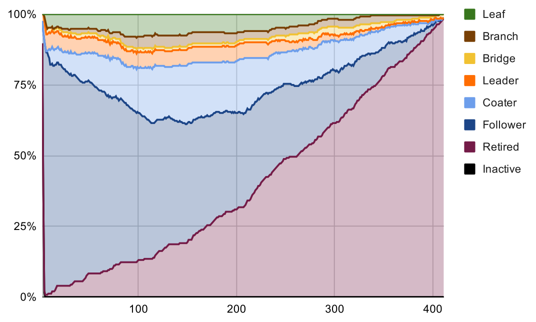
(b) 4 Leaders



(c) 9 Leader



(d) 16 Leaders



(e) 25 Leader

Figure 5.5: Comparing the distributions of roles along the number of rounds for the Hypothesis 7

Hypothesis 8. *How does SMA compare with SSA?*

Setup. To test this hypothesis we prepared a series of tests where we used two shapes of varying radii. We created a line type shape to reflect the shape shown in Subfigure 3.10a. We create a rhombic dodecahedron to reflect the shape shown in Subfigure 3.10c. For these setups, we will only be using one leader to evaluate the performances one to one.

Observations. In Figure 5.6, we can see that when using the SSA scheduler, the algorithms performs worse than the 2D *Filling Algorithm*. When using the SMA the algorithms performs better than the 2D *Filling Algorithm*. The multiple leader implementation has been shown to lower the number of rounds needed to coat the first layer. SMA will always outperform SSA because in SSA every particle is activated exactly once, SMA simply performs many more activations.

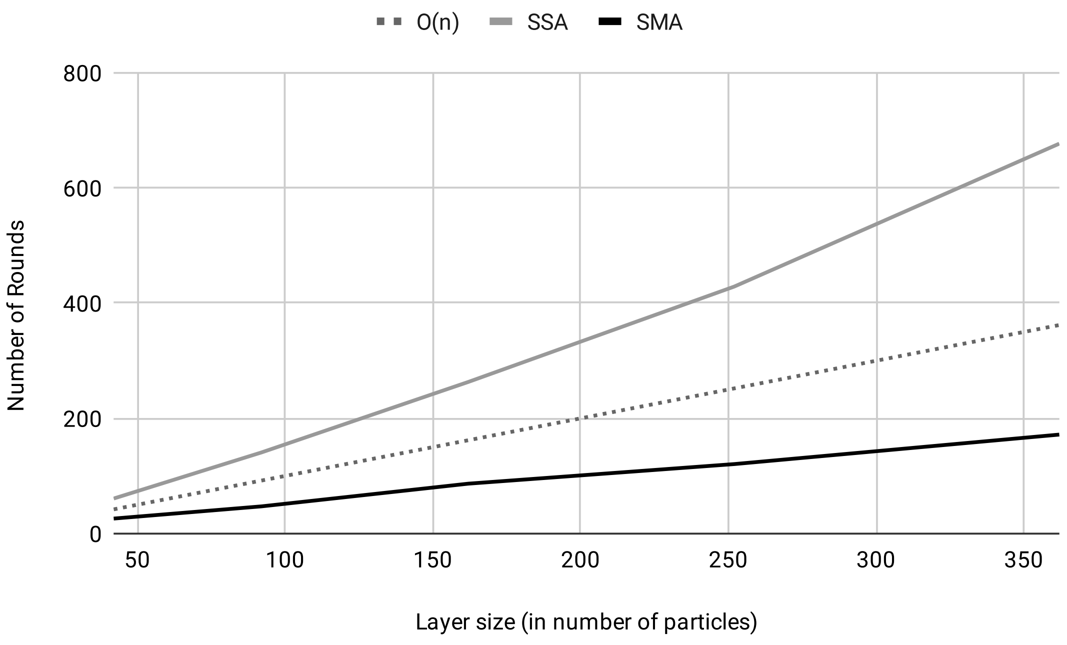
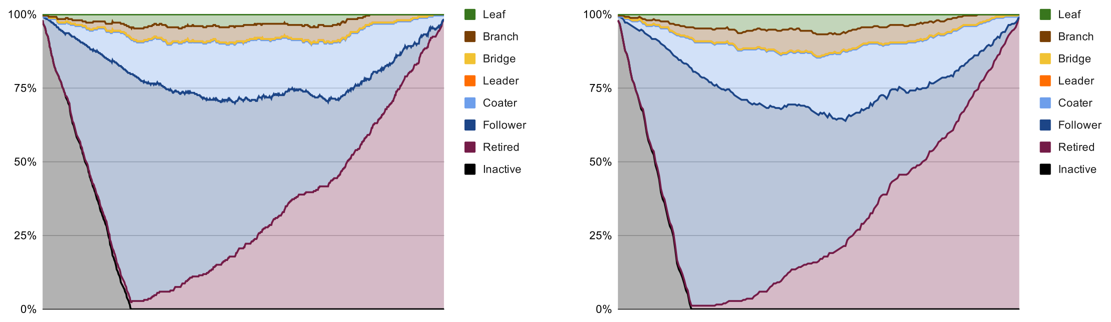


Figure 5.6: Comparing SMA performances against SSA when coating a rhombic dodecahedron of varying size using an othortope configuration for the initial set of particles.



(a) SSA

(b) SMA

Figure 5.7: Observing the evolution of particle roles over the number of rounds

Hypothesis 9. *Under SSA scheduler increasing the number of particles needed for the same sized layer will increase the average running time.*

Setup. To test this hypothesis, we will use a rhombic dodecahedron object of size 3. We will use an orthotope initial configuration with 9 leaders. We proceeded to run 20 trials for each data point.

Observations. From previous observations about SMA, we know that increasing the number of particles decreases the overall number of rounds needed to coat the layer. In Figure 5.8, we can see that increasing the total number of particles for the same object yields similar results. It is worth noting that the improvement in the number of rounds is marginal. Initially, 10 tests were run for each data point, but as we increased the number of particles, the distribution spread out.

In an attempt to iron out the outliers, we doubled the number of tests. Even then, the experiment with 324 particles still yielded results outside the expectations.

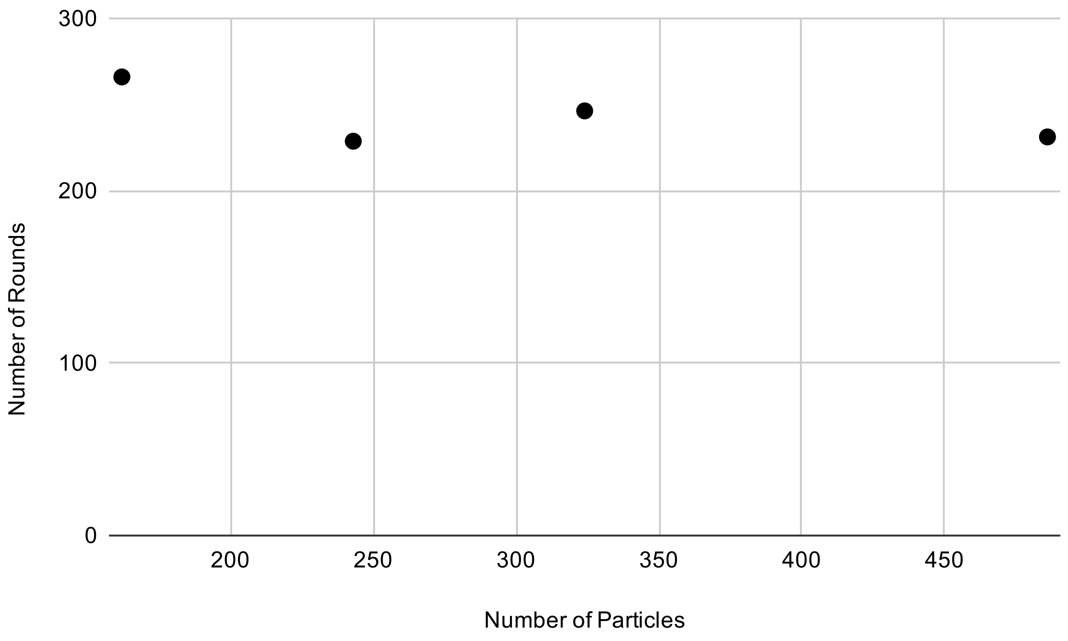


Figure 5.8: Coating an rhombic dodecahedron of size 3. Initially the number of particles need to coat the object is 162. Each point represents an average of 20 trials, for 243 (50% increase), 324 (100% increase) and 485 (200% increase)

Chapter 6

Conclusions

In this thesis we have described a solution to the *Filling Problem* that runs in at most $O(n^2)$ asynchronous rounds. We have shown that this bound only applies to a single input. As the bound was made tighter, the *Filling Algorithm* ran in at most $O(n \cdot R)$ asynchronous rounds, where R is the size of the longest chain of connected particles. The main idea of the algorithm is that particles form two tree structures. One tree is used to funnel particles to the hole in the boundary object, beyond which is the area to be filled. Once inside, a second tree directs particles to the leaves in the tree. These leaves direct the filling effort by expanding/contracting to fill the area. The particle at the hole is called a leader.

In the experimental results, we have explored how different factors such as the diameter of the object, number of particles and edge cases affect the running time. We found that increasing the number of particles decreased the number of rounds needed to fill an area, we discussed that it was due to the nature of the scheduler. We observed that the diameter of a shape impacts the average height of the filling tree. Furthermore, we also found that the single leader particle bottlenecks the *Filling Algorithm.* We will explore in the *Coating Algorithm* experimental results the implications of using multiple leaders.

In this thesis we have described a solution to the *Coating Problem* called the *Filling Algorithm* that runs in at most $O(n \cdot R)$ asynchronous rounds, where R is the size of the longest chain of connected particles. The *Coating Algorithm* builds upon the *Filling Algorithm.* In the *Coating Problem*, a particle system is connected to an object. The particles on the surface of that object form a connected set of particles. When this set contains more than one particle, multiple leaders can be elected. Each leader behaves in the same way as in *Filling Algorithm.* In the *Coating Algorithm*, all leaders are part of a tree structure rooted at a super leader. When a leader completes its coating effort, it converts itself to a bridge role. The bridge role helps other leaders complete their coating routine by handing over their spare particles. When testing the *Coating Algorithm* we began by fine tuning a configuration which would match the *Filling Algorithm.* Using this configuration, we proceeded to explore the implications of using multiple leaders. We discovered that the improvements of multiple leaders soon plateaus. When we explored the impact of boundary leaders, leaders not trapped by other surrounding leaders, we found that the performance was bottle necked by another factor. When looking at the role distribution over the number of rounds, we found that as the number of boundary leaders increases, the number of branches decreases. This suggested that nearby leaders are competing on the same layer. When testing under the SSA and SMA scheduler, we discovered that increasing the number of particles for the same layer did not increase the number of rounds needed to coat the layer. When exploring the running time of multiple layer coating, the sum area of all the layers will impact the running time, but the number of particles to coat these layers will not impact the running time.

While both algorithms have been developed under sequential schedulers, research has shown that when the system's actions are atomic and isolated the set of actions can be serialized; this applies to any set of actions performed by this system. Hence, for any concurrent execution, there exists a sequential ordering of actions that yield the same output.

To visualize the *Coating Algorithm* a simulator was created. In its current state, the simulator can only be used to explore the *Coating Algorithm*.

Future work. For the *Filling Algorithm* the bottleneck is the number of leaders. We have seen that a single leader is the current bottleneck in the system. We have shown that the running time for tree formation can be tighter using the in-degree of a node.

In the *Coating Algorithm* we make the assumption that the contact point between the set of particles P and the object O is connected. This assumption can be removed by exploring the potential implications of having a particle on the layer: a super leader that can be elected for each contact group. This solution opens up new problems as the different group of *super leaders* might disconnect from one another.

Bibliography

[1] Rida A. Bazzi and Joseph L. Briones. Brief announcement: Deterministic leader election in self-organizing particle systems. *Computer Science Stabilization, Safety, and Security of Distributed Systems*, page 381–386, 2018. 1, 3, 7

[2] Marshall Bern, Erik D. Demaine, David Eppstein, Eric Kuo, Andrea Mantler, and Jack Snoeyink. Ununfoldable polyhedra with convex faces. *Comput. Geom. Theory Appl.*, pages 24(2):51–62, 2003. 11

[3] Joshua J. Daymude, Robert W. Gmyr, Andréa undefined Richa, Christian undefined Scheideler, and Thim undefined Strothmann. Improved leader election for self-organizing programmable matter. *International Symposium on Algorithms and Experiments for Sensor Systems*, page 127–140, 2017. 3, 7

[4] Erik D. Demaine, David Eppstein, George W. Hart Jeff Erickson, and Joseph O'Rourke. Vertex-unfolding of simplicial manifolds. *Proceedings of the 18th Annual ACM Symposium on Computational Geometry*, pages 237–243, 2002. 11

[5] Z. Derakhshandeh, S. Dolev, R. Gmyr, and A. W. Richa. Brief announcement: Amoebot-a new model for programmable matter. *SPAA 14'*, pages 220–222, 2014. 2

[6] Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Lecture Notes in Computer Science DNA Computing and Molecular Programming*, page 148–164, 2016. 1, 20

[7] Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017. 1, 3, 4, 18, 40

[8] Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. *International Workshop on DNA Computing and Molecular Programming*, page 117–132, 2015. 3, 7, 8, 33

[9] Albrecht Dürer. The painter's manual: A manual of measurement of lines, areas, and solids by means of compass and ruler assembled by albrecht dürer for the use of all lovers of art with appropriate illustrations arranged to be printed in the year mdxxv. *Abaris Books*, 1525. 11

[10] Y. Emek, S. Kutten, Jr William K., and R. Lavi. Deterministic leader election in programmable matter. *arXiv:1905.00580v1*, 2019. 3

[11] Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed leader election and computation of local identifiers for programmable matter. *Algorithms for Sensor Systems*, page 159–179, 2019. 3, 7

[12] Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *21st International Conference on Principles of Distributed Systems*, 2017. 3, 7

[13] Joseph O'Rourke. Folding and unfolding in computational geometry. *Japan Conf. Discrete Comput. Geom*, pages 258–266, 2000. 11

[14] Giuseppe Prencipe Paola Flocchini and Nicola Santoro. *Distributed Computing by Mobile Entities.* Springer International, 2019. 8

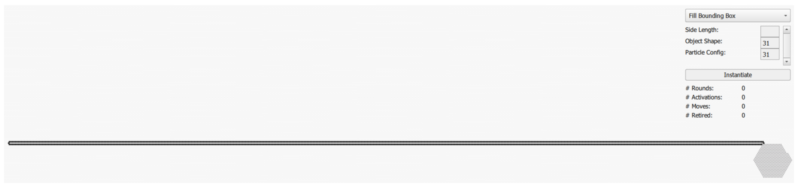
[15] Vassos Hadzilacos Philip Bernstein and Nathan Goodman. Concurrency control and recovery in database systems. *Addison-Wesley*, 1987. 3

[16] Geoffrey C. Shephard. Convex polytopes with convex nets. *Math. Proc. Cambridge Philos. Soc.*, page 78(3):389–403, 1975. 11

[17] Wikipedia. UV mapping — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=UV%20mapping&oldid=958558656`, 2020. [Online; accessed 28-May-2020]. vi, 11

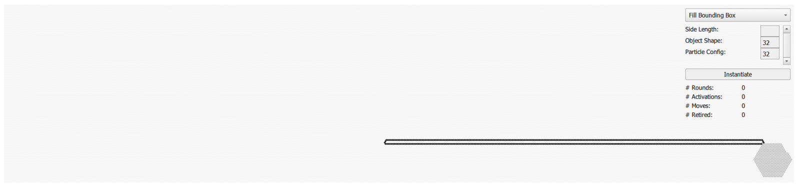
Appendix A

2D Test Figures



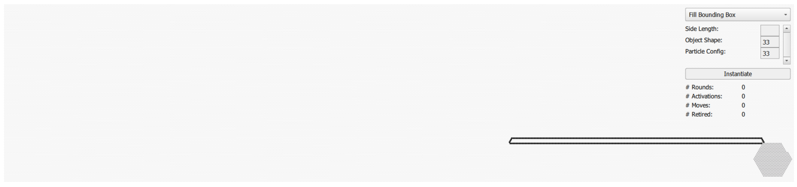
Fill Bounding Box
Side Length:
Object Shape: 31
Particle Config: 31
Instantiate
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(a)



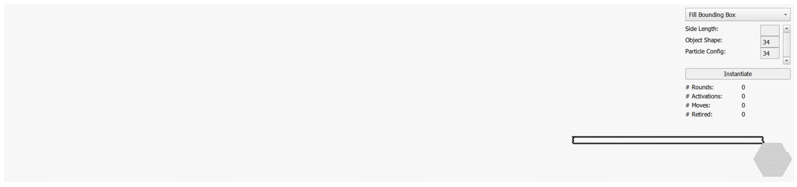
Fill Bounding Box
Side Length:
Object Shape: 32
Particle Config: 32
Instantiate
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(b)



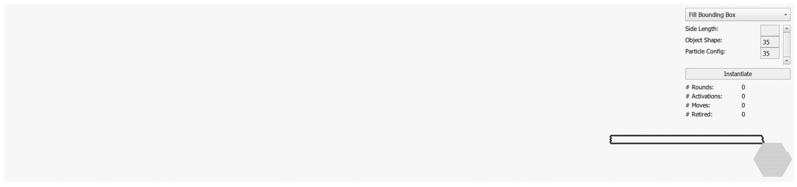
Fill Bounding Box
Side Length:
Object Shape: 33
Particle Config: 33
Instantiate
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(c)



Fill Bounding Box
Side Length:
Object Shape: 34
Particle Config: 34
Instantiate
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

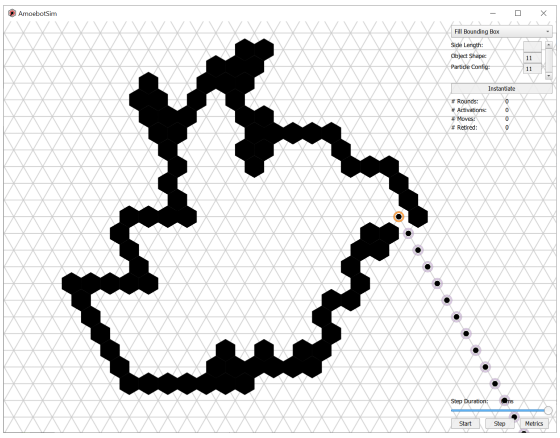
(d)



Fill Bounding Box
Side Length:
Object Shape: 35
Particle Config: 35
Instantiate
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

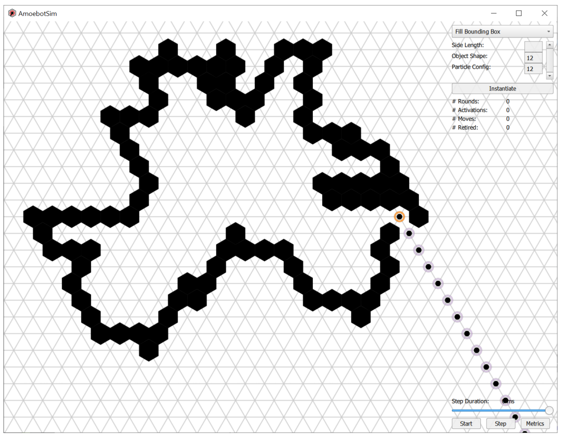
(e)

Figure A.1: Shapes with varying size corridor for testing



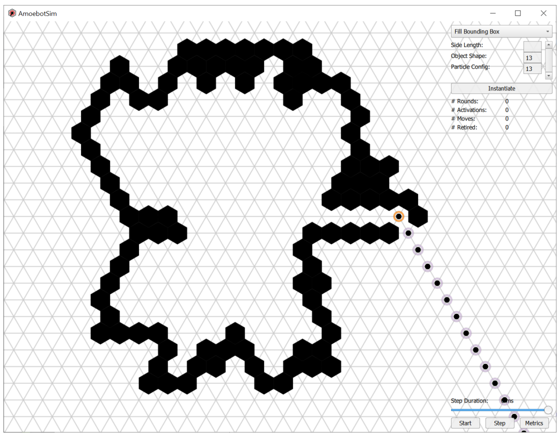
Side Length:
Object Shape: 11
Particle Config: 11
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(a)



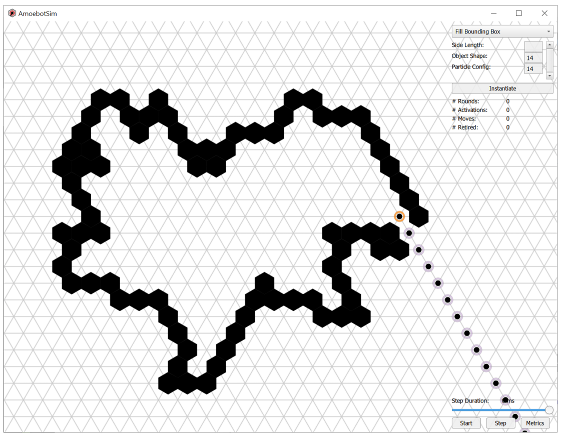
Side Length:
Object Shape: 12
Particle Config: 12
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(b)



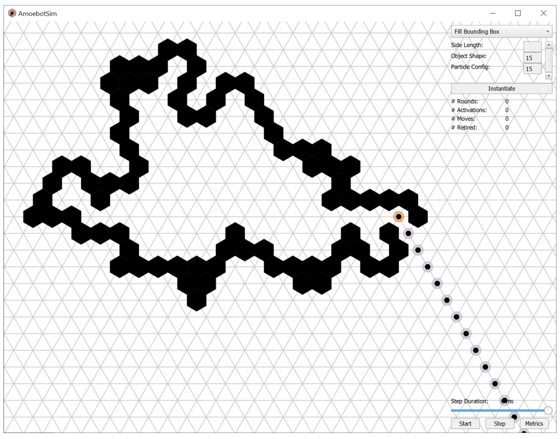
Side Length:
Object Shape: 13
Particle Config: 13
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

(c)



Side Length:
Object Shape: 14
Particle Config: 14
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

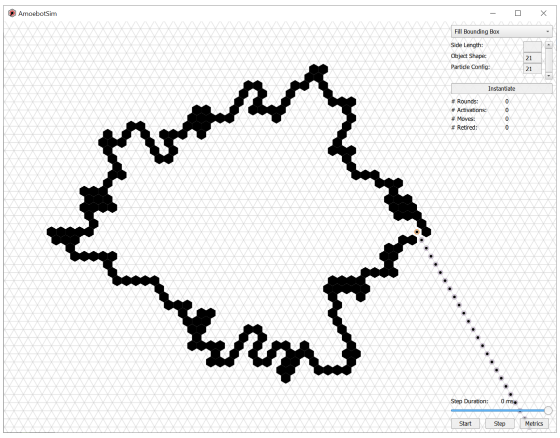
(d)



Side Length:
Object Shape: 15
Particle Config: 15
Rounds: 0
Activations: 0
Moves: 0
Retired: 0

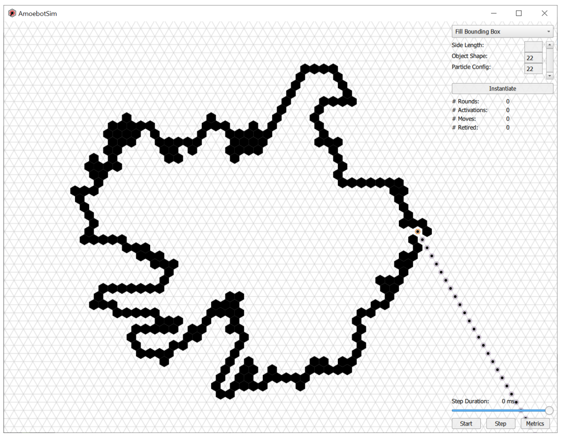
(e)

Figure A.2: Random shapes used for testing



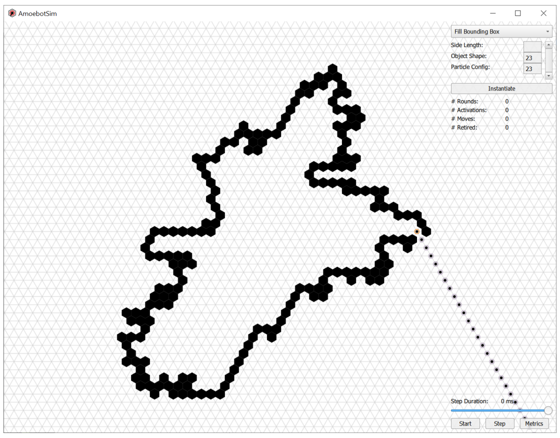
Fill Bounding Box
Side Length:
Object Shape: 21
Particle Config: 21
Rounds: 0
Activations: 0
Moves: 0
Retired: 0
Step Duration: 0 ms

(a)



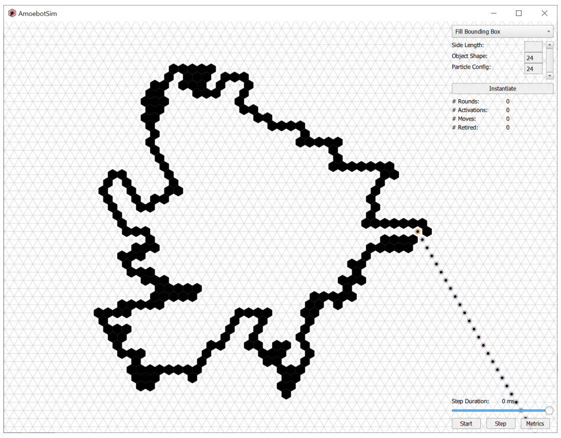
Fill Bounding Box
Side Length:
Object Shape: 22
Particle Config: 22
Rounds: 0
Activations: 0
Moves: 0
Retired: 0
Step Duration: 0 ms

(b)



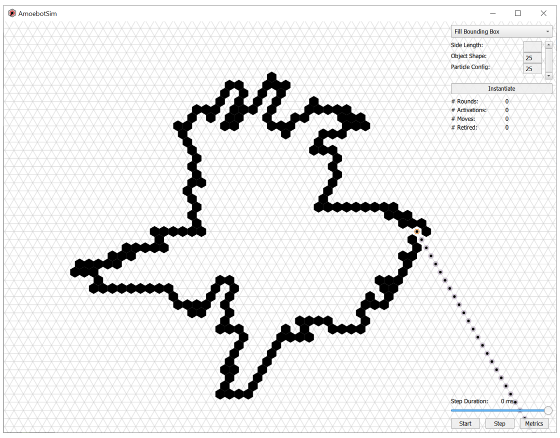
Fill Bounding Box
Side Length:
Object Shape: 23
Particle Config: 23
Rounds: 0
Activations: 0
Moves: 0
Retired: 0
Step Duration: 0 ms

(c)



Fill Bounding Box
Side Length:
Object Shape: 24
Particle Config: 24
Rounds: 0
Activations: 0
Moves: 0
Retired: 0
Step Duration: 0 ms

(d)



Fill Bounding Box
Side Length:
Object Shape: 25
Particle Config: 25
Rounds: 0
Activations: 0
Moves: 0
Retired: 0
Step Duration: 0 ms

(e)

Figure A.3: Random shapes used for testing

Appendix B

Code

Provided that the code base for each of these algorithm would not fit nicely in an Appendix, a link to each repository is shared below:

- 2D Filling Algorithm
- 3D Coating Algorithm