Compression in Self-Organizing Particle Systems

Joshua J. Daymude

APPROVED:

Dr. Andréa Richa
Director x_____

Dr. Henry Kierstead
Second Committee Member x_____

Compression in Self-Organizing Particle Systems

Joshua J. Daymude

April 15, 2016

Abstract

Many programmable matter systems have been proposed and realized recently, each often tailored toward a particular task or physical setting. In our work on self-organizing particle systems, we abstract away from specific settings and instead describe programmable matter as a collection of simple computational elements (to be referred to as particles) with limited computational power that each perform fully distributed, local, asynchronous algorithms to solve system-wide problems of movement, configuration, and coordination. In this thesis, we focus on the *compression* problem, in which the particle system gathers as tightly together as possible, as in a sphere or its equivalent in the presence of some underlying geometry. While there are many ways to formalize what it means for a particle system to be compressed, we address three different notions of compression: (1) *local compression*, in which each individual particle utilizes local rules to create an overall convex structure containing no holes, (2) *hole elimination*, in which the particle system seeks to detect and eliminate any holes it contains, and (3) α-*compression*, in which the particle system seeks to shrink its perimeter to be within a constant factor α of the minimum possible value. We analyze the behavior of each of these algorithms, examining correctness and convergence where appropriate. In the case of the Markov Chain Algorithm for Compression, we provide improvements to the original bounds for the bias parameter λ which influences the system to either compress or expand. Lastly, we briefly discuss contributions to the problem of *leader election*—in which a particle system elects a single leader—since it acts as an important prerequisite for compression algorithms that use a predetermined *seed* particle.

1 Introduction

Many programmable matter systems have recently been proposed and realized—modular and swarm robotics, synthetic biology, DNA tiling, and smart materials form an incomplete list—and each is often tailored toward a specific task or physical setting. In our work on self-organizing particle systems, we abstract away from specific settings and instead describe programmable matter as a collection of simple computational elements (to be referred to as *particles*) with limited computational power that each perform fully distributed, local, asynchronous algorithms to solve system-wide problems of movement, configuration, and coordination. In this thesis, we discuss different approaches to the problem of for *compression*, in which the particle system gathers as tightly together as possible, as in a sphere or its equivalent in the presence of some underlying geometry.

Compression of systems appears in many phenomena, and resulting configurations are often useful for communication and information propagation. In natural systems, compression often leads to some additional utility; ants form bridges and floating rafts by gathering in such a manner, and honey bees communicate foraging patterns using swarming and recruitment. Compression also applies to potential synthetic biological systems which could mimic the behavior of a macrophage by gathering around and engulfing foreign bodies, or act as clotting agents by gathering and sealing minor lacerations. In civil engineering, intelligent, self-organizing sensors could detect, report, and temporarily patch structural weaknesses in infrastructure. The algorithms presented here provide a generalized, theoretical framework from which these various motivations and applications can be addressed.

2 Related Work

Problems akin to compression have been studied previously in a variety of contexts. When considering physical systems and models, one can differentiate between active and passive systems. Particles in passive systems have no explicit control over their movements, and in some cases do not have any computational power. A notable example of a passive system is DNA self-assembly via folding, which as described in [30] could be used to create a compressed configuration. In active systems, particles have control over their behavior and—depending on the model—can achieve some directed locomotion. *Swarm robotics* is one example; within a swarm each individual autonomous robot is able to gather information, communicate, and move within some limitations. The abilities of specific swarms vary, but notable examples such as the *kilobots* [25] or those described in [17] are able to achieve shape formation and collection after some pre-processing is performed to establish a kind of global orientation. The *nubot* model [31] addresses a framework for biomolecular-inspired models which—although allowing for some non-local movements—provides additional means of creating two dimensional shapes in polylogarithmic time. Similarly, pattern formation and creation of convex structures has been studied within the *cellular automata* domain (e.g. [7, 14]), but differs from our model by assuming more powerful computational capabilities, such as knowledge of the size of the system or a global sense of orientation.

Nature offers a variety of examples in which gathering and cooperative behavior is apparent [18, 29]. For example, social insects often exhibit compression-like characteristics in their collective behavior; fire ants form floating rafts by interlocking into a mass that

evenly distributes their weight over the surface tension of water [22]; cockroach larvae perform self-organizing aggregation based on local propagation of pheromones and odors [20, 24]; and honey bees choose hive locations based on a decentralized process of swarming and recruitment [4]. Individual units in these natural systems are able to achieve and utilize compressed formations for physical tasks that inspire this work, but in general have more powerful mechanisms and communication than those considered here.

The "rendezvous" (or "gathering") problem—first described in [26]—seeks to gather mobile agents together on some node of a graph, whose structure can be anonymous [2, 3, 6] or have some known topology such as a tree [1, 8] or ring [15, 21, 27]. Bampas et al. [2] consider the gathering of two mobile agents with a limited field of vision moving asynchronously in δ-dimensional Euclidean space, and propose an algorithm for their trajectories which guarantees their meeting within a set distance traveled. Fault tolerant versions of the problem have also been considered [6, 9, 15]. Chalopin et al. [6] discuss how the gathering problem can function in the presence of dangerous or faulty edges which destroy agents that move along them, guaranteeing the gathering of some minimum threshold of agents, while Dobrev et al. [15] study the situation of having a "black hole" node on a ring which destroys any agents that visit it. In comparison, our particles occupy physical space instead of gathering at a single node and are computationally simpler than the agents considered in the works above.

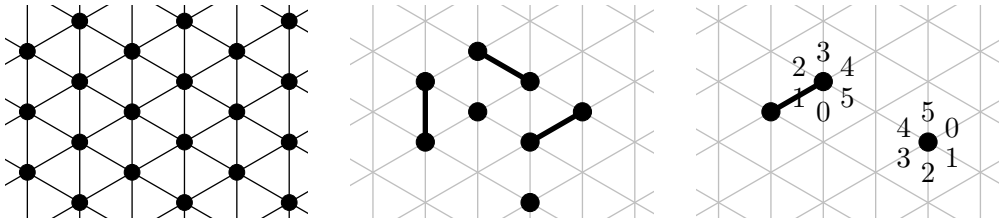
Lastly, in [11], we presented an algorithm for hexagon shape formation in the amoebot model, in which particles follow a snake-like pattern which sequentially places particles to create successive layers over an elected seed particle. Although creating a hexagon does gather particles as desired, it relies heavily upon the sequential placement of particles and has need for an underlying organization of the particles (the algorithm in [11] organizes the particles in a spanning forest rooted at a seed particle, which is maintained throughout the execution of the algorithm). In this thesis, however, compression algorithms aim to take a more fully decentralized and local approach, which is naturally self-stabilizing, forgoing the need for a seed particle or heavy reliance on an underlying organization of the set of particles.

3 Background and Model

We begin by describing the amoebot model for programmable matter, which was originally proposed in [10]. After defining some additional terminology, we conclude by discussing the different ways of formalizing what it means for a particle system to be compressed.

3.1 The Amoebot Model

In the *general amoebot model*, programmable matter consists of individual computational units known as particles, whose properties we now detail. An infinite, undirected graph $G = (V, E)$ is used to represent the particle system, where V is the set of all possible locations a particle could occupy relative to its structure and E is the set of all possible atomic transitions between locations in V [10, 12]. For the purposes of this thesis, we assume the *geometric* variant of the amoebot model, which imposes an underlying geometric structure $G = G_{eqt}$, where G_{eqt} is the *triangular lattice* (Figure 1a).



(a) (b) (c)

Figure 1: (a) A section of the triangular lattice G_{eqt}; (b) expanded and contracted particles; (c) two non-neighboring particles with different offsets for their port labels.

Each particle is either *contracted*, occupying a single location, or *expanded*, occupying a pair of two adjacent locations. Figure 1b depicts both contracted and expanded particles. Particles achieve movement via a series of *expansions* and *contractions*; a contracted particle may expand into a neighboring unoccupied location to become expanded, and completes its movement by contracting to once again occupy only a single location. Moreover, particles can coordinate their movements via *handovers*, in which two scenarios are possible: (1) a contracted particle p can "push" a neighboring expanded particle q and expand into one of the nodes previously occupied by q, forcing a contraction of q or conversely, (2) an expanded particle p can "pull" a neighboring contracted particle q to a node it currently occupies, forcing an expansion of q. This handover mechanism allows particles to move in an amoeba-like fashion, providing an efficient and safe way to move without severing their connection.

Two particles occupying adjacent locations in $V(G_{eqt})$ are said to be *neighbors*. These particles are *anonymous*—that is, they have no unique identifiers—but each has a collection of ports corresponding to edges incident to its location which have unique labels and via which bonds are formed with their neighbors. While the particles do not share any sense of global orientation, we additionally assume particles have a common *chirality*, meaning they share the same notion of a *clockwise direction*. This allows the particles to agree on a way to order their port labels, as illustrated in Figure 1c.

Every particle has a constant-size, local, shared memory which both it and its neighbors are able to read from and write to for communication. Due to the limitation of constant-size memory, particles know neither the total number of particles in the system nor any estimate of this quantity. We assume the standard asynchronous model from distributed computing, where progress is achieved through a series of *particle activations* in which only one particle is acting at a time.

3.2 Compression of Particle Systems

Our objective is to find a solution to the compression problem for self-organizing particle systems. There are many ways to formalize what it means for a system to be *compressed*; for example, one could try to minimize the diameter of a system, maximize the number of edges, maximize the number of triangles, eliminate any holes it contains, etc. This thesis explores three different notions of compression: *local compression*, in which a particle system seeks for each of its particles to satisfy some local rules, resulting in a convex configuration

containing no holes; *hole elimination*, in which a particle system simply eliminates holes it contains; and α-*compression*, which is defined in terms of minimizing the perimeter of the system. While formal definitions of each of these notions will be left to their respective sections, we here define further terminology and notation that is common to all of them.

Definition 3.1. A *hole* in a particle system is a finite, maximal location set $H \subseteq V(G_{eqt})$ such that the subgraph of G_{eqt} induced by H is connected, contains no occupied locations, and is bounded by occupied locations.

Definition 3.2. The neighborhood of a location $l \in V(G_{eqt})$, denoted $N(l)$, is the set of locations incident to l along edges in $E(G_{eqt})$.

Definition 3.3. The neighborhood of a particle p, denoted $N(p)$, is the set of particles occupying locations adjacent to that of p. We refer to any particles p and $q \in N(p)$ as *neighbors*.

It is important to note that, as defined, $l \notin N(l)$ and $p \notin N(p)$. Additionally, when referring to a particle p which has expanded and occupies two locations, we wish to exclude p from its own neighborhood. Thus, for an expanded particle p which occupies positions l and l', we refer to the neighborhood of its location as $N(l \cup l') = N(l) \cup N(l') \setminus \{l, l'\}$.

4 Early Approaches

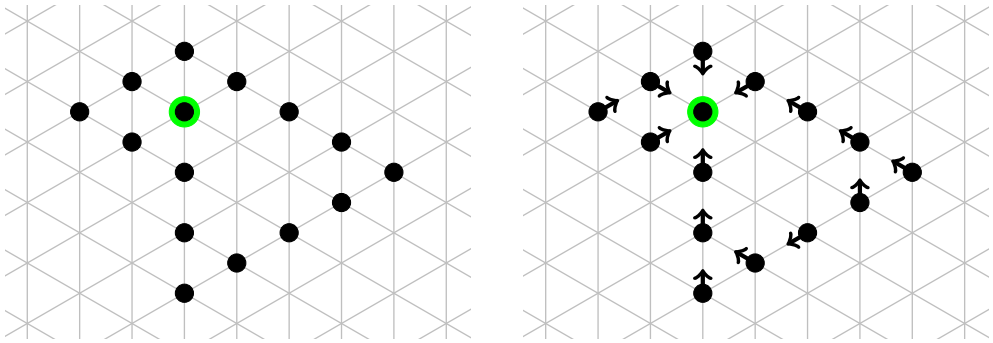
In the course of this work on compression, we have explored many different algorithms and strategies. Two significant results are the *Local Compression* algorithm—referred to in previous work as the *compaction* algorithm—and the *Hole Elimination* algorithm. Both of these algorithms share characteristics with the shape formation algorithms for self-organizing particle systems [11], such as the predetermination of a "seed" particle which acts as a leader for the system, and the utilization of the spanning forest primitive which provides a local method for obtaining a sense of direction in an otherwise compass free system.

4.1 Local Compression

We first present the Local Compression algorithm (Algorithm 1), a fully distributed, asynchronous, nondeterministic algorithm for compression. In Algorithm 1, each individual particle is responsible for satisfying a set of conditions and—when failing to satisfy them—can initiate a series of coordinated movements with other particles to achieve a better configuration. The following definition formalizes this set of local conditions and what notion of compression this algorithm achieves.

Definition 4.1. A particle system is said to be *locally compressed* if every particle p is *particle compressed*; that is, (1) p does not have exactly five neighbors, and (2) the graph induced by $N(p)$ is connected.

We now fully describe Algorithm 1 in a high-level, narrative format. Complimentary pseudocode can be found below. For convenience, we refer to particles which have one or more children in the spanning forest as *parents*, and particles with no children as *leaves*.



(a) (b)

Figure 2: (a) Initial configuration without orientation; (b) orientation via spanning trees.

We assume that the particle system is initially connected, has a unique "seed" particle s, and that all other particles are initially inactive and without orientation.

From the perspective of some particle p, this compression algorithm takes place in two phases: orientation and movement. During orientation, a sense of direction is propagated throughout the system by means of spanning trees, as depicted in Figure 2. In detail, if p is not yet oriented, it checks to see if the seed s is in its neighborhood. If so, it orients itself toward s; otherwise, if it has an oriented neighbor q, it orients itself toward q. These checks are repeated until p obtains an orientation.

Both p and its neighbors must complete the orientation phase before beginning movement; this ensures that the system does not become disconnected. If p is not currently taking part in a movement, it checks for two criteria: if its parent in the spanning tree, say q, is a "follower" or "leader", and otherwise, if it is not particle compressed and is a parent. In the first case, p becomes a follower and expands into the position currently occupied by q. When q is also a follower, q remains the parent of p in the spanning tree (Figure 3d); however, when q is a leader, p is assigned a new parent according to information left behind by q, as in Figure 3c (the source of this information is detailed in the second case). In the second case, p detects that it is not particle compressed and will attempt to fix this gap in its neighborhood by moving into an neighboring empty position as depicted in Figures 3a–3b. It achieves this by becoming a leader and expanding into the first empty position found counter-clockwise from its parent. Before this expansion, however, it communicates the relative position of its parent to each of its children. This is done such that whichever child takes its place will be able to deduce where p's old parent was and adopt it as its own, thus maintaining the spanning tree structure. After p expands, it sets its new parent to be whichever child takes its place.

By rules of the algorithm, a particle must be a parent to initiate a movement which fills a potential concavity or hole so as to avoid disconnecting the system. However, it is possible to have a hole whose border is entirely composed of leaves, as illustrated in Figure 4a. Thus, a third movement is used: if a leaf is not involved in some other movement, it may choose a random particle from its neighborhood to become its new parent, which gives the new parent the necessary child for initiating movements (Figure 4b). This *leaf switching* is otherwise inconsequential; since only leaves in the tree are allowed to switch parents, the

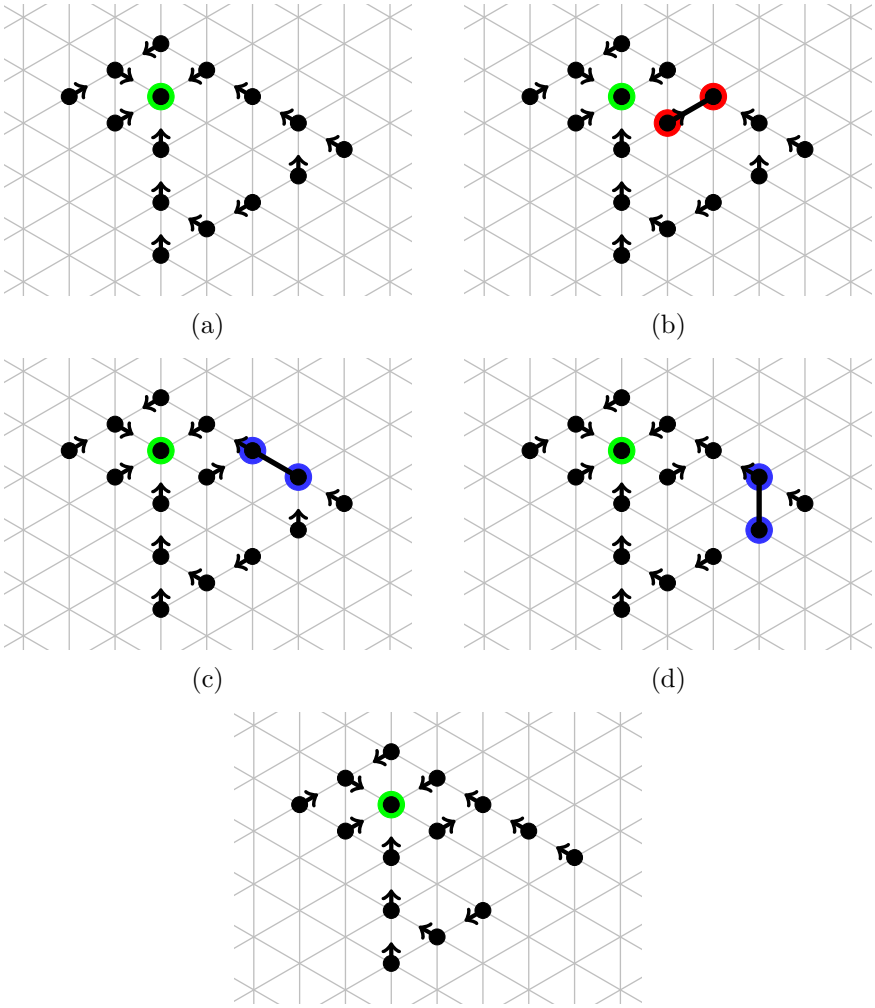
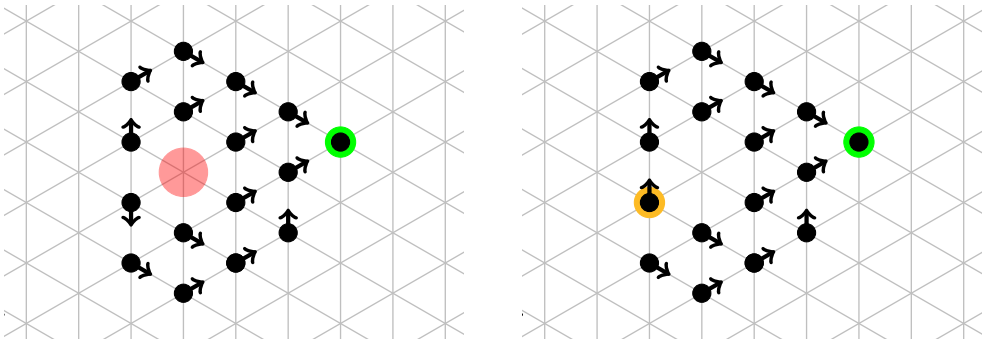


Figure 3: (a) The particle p directly to the right of the seed (green) detects local non-compactness; (b) p expands and becomes a leader (red); (c) a child of p becomes a follower (blue) and adopts p's old parent as its own during expansion; (d) the child of the follower also becomes a follower and expands; (e) this last follower contracts, completing this movement.



(a) (b)

Figure 4: (a) A hole bounded completely by leaves; (b) a leaf which has switched parents.

spanning tree structure is maintained and the system remains connected.

These movements continue indefinitely, continuously attempting to detect and correct holes and other local concavities in the system. This additionally allows the system to adapt if particles are added or removed to the system on-the-fly.

We now prove an important lemma which links the distributed nature of local compression to geometric characteristics of the particle system as a whole. Specifically, we show that local compression implies a particle system forming a *convex* configuration—that is, every external angle α on the outer border of the system has $\alpha \geq 180^\circ$—which contains no holes. Notably, this is not true in the other direction; a straight line contains no holes and is also convex—the external angles are all 180° or 360°—but every particle except those at the endpoints have graphs induced on their neighborhoods which are disconnected.

Lemma 4.1. *If a particle system is locally compressed, then it forms a convex configuration containing no holes.*

Proof. Suppose to the contrary that a particle system is locally compressed but is not convex or contains a hole. If the system is not convex, then there exists an external angle α on the outer border of the system with $\alpha < 180^\circ$. Let p be the particle at the vertex of α. Then, in fact, $\alpha = 120^\circ$; otherwise p is not on the outer border. Now p has at least two neighbors (those with which it forms α) and has an unoccupied location between them. But here we reach a contradiction: if all three remaining neighbor locations are occupied by particles, then $|N(p)| = 5$; otherwise, p is incident to two borders, forcing the graph induced on $N(p)$ to be disconnected.

If the system contains a hole, when traversing the hole's border from its interior, there must be an angle $\beta < 180^\circ$; otherwise, the hole's border cannot be closed and thus is no hole at all. But then the same argument holes as above, and a contradiction is reached. □

While this global geometric property provides us insight into the overall structure of a locally compressed particle system, it also emphasizes how loose this particular interpretation of compression really is. For example, any straight line configuration which is two particles thick qualifies as convex and containing no holes, but depending on length is certainly far less gathered than originally desired (compare Figures 5a–5b as an example). This difficulty is compounded by the algorithm's practical performance and behavior;

Algorithm 1 Local Compression on a Particle p

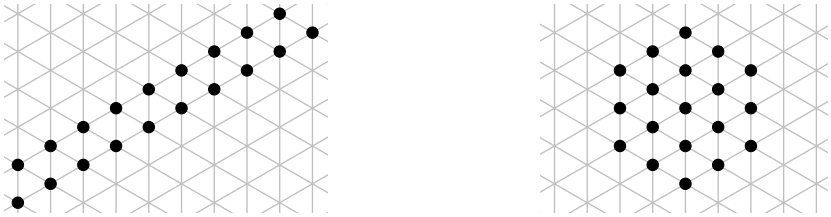
Phase 1: Orientation
while $p.state = inactive$ **do**
  if seed $s \in N(p)$ **then** $p.parent \leftarrow s$
  else if $\exists q \in N(p)$ with $q.state = active$ **then** $p.parent \leftarrow q$

Phase 2: Movement
while true do
  if (p is expanded) $\wedge$ ($!\exists q \in N(p)$ with $q.state = inactive$) **then**
    if p is a leaf **then** contract
    else if (q is a child of p) $\wedge$ ($q.state = follower$) **then**
      handover contract with q
  else if ($p.state = active$) $\wedge$ ($!\exists q \in N(p)$ with $q.state = inactive$) **then**
    if $p.parent.state = follower$ **then**
      $p.state \leftarrow follower$
      handover expand with $p.parent$
    else if $p.parent.state = leader$ **then** .
      $p.state \leftarrow follower$
      handover expand with $p.parent$
      $p.parent \leftarrow flag$
    else if (p is not particle compressed) $\wedge$ (p is a parent) $\wedge$ ($!\exists q \in N(p)$ with $q.state = follower$) **then**
      $p.state \leftarrow leader$
      communicate relative position of $p.parent$ to each neighbor as $flag$
      expand into first unoccupied position counter-clockwise from $p.parent$
      $p.parent \leftarrow p$
    else if (p is a leaf) $\wedge$ ($!\exists q \in N(p)$ with $q.state = follower$ or $leader$) $\wedge$ ($|N(p)| \neq 6$) **then**
      choose $q \in N(p)$ uniformly at random
      $p.parent \leftarrow q$

while it generally achieves some gathering very quickly, sequences of moves such as those depicted in Figure 6 can open up new holes in the system or enlarge the very concavities it was meant to mend. This often causes oscillation throughout the system; particles make productive and unproductive moves with equal probability, and go between creating issues and fixing them ad infinitum. Thus, while the local nature and simplicity of the algorithm are greatly desirable, its acceptance of only loosely gathered particle systems as valid goal configurations and its inability to stabilize are glaring shortcomings that were kept in mind when designing its successors.

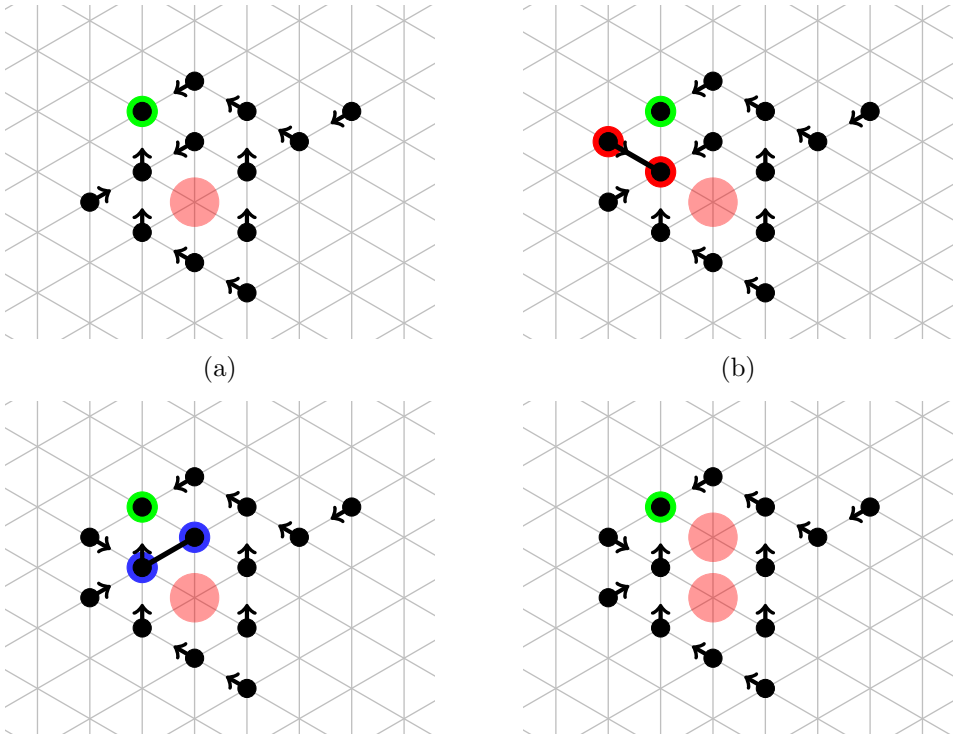
4.2 Hole Elimination

In a joint work with Robert Gmyr at the Universität Paderborn, we developed the Hole Elimination algorithm (Algorithm 2) to address the issues with instability and oscillations



(a) (b)

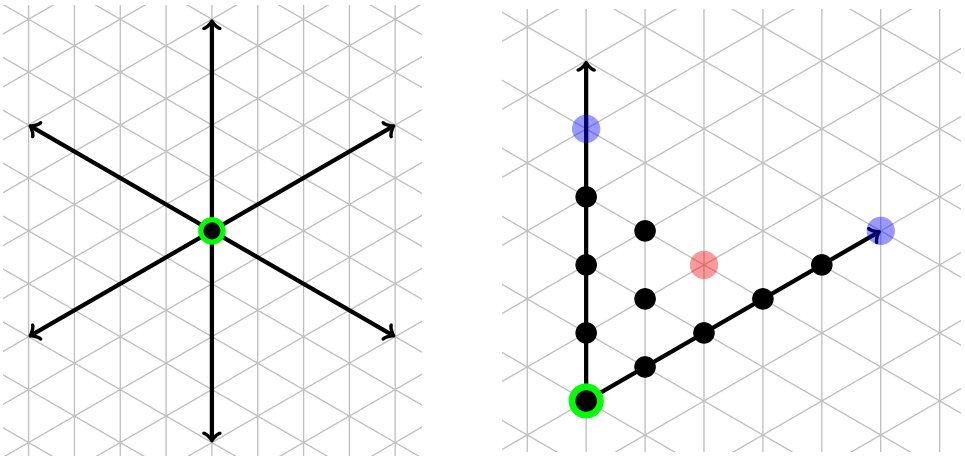
Figure 5: (a) A particle system on 19 particles which is convex and contains no holes, but has compressed poorly; (b) optimal compression on 19 particles.



(a) (b)

(c) (d)

Figure 6: (a) A particle p to the up-left of a hole detects adjacency to two potential concavities; (b) unable to locally determine the optimal choice, p expands outward; (c) a child of p becomes a follower and adopts p's old parent; (d) the child contracts, completing the movement.



(a) (b)

Figure 7: (a) The six axes directed outwards from the seed; (b) examples of docking locations, where those in blue satisfy (1) and the one in red satisfies (2).

found in Algorithm 1. Here, we investigate a much weaker version of compression, in which a particle system merely eliminates any holes that it contains. The property separating hole elimination from more optimal compression is the degree of convexity in the final configuration of the system; optimal compression results in a regular, convex sphere while hole elimination is much more permissive, allowing curves and irregular surfaces with concave borders so long as they contain no holes.

At a high level, this algorithm operates in two phases: first, particles are organized into a spanning forest, as in Algorithm 1; then, particles follow their parents to the border of the system in order to traverse it in the clockwise direction, searching for a safe location to finish without creating permanent holes. The following definitions formally describe the conditions under which a particle will finish at such a location, which is said to be *docking*.

Definition 4.2. The set $A \subseteq V$ of locations lying along an infinite ray directed outwards from the seed along a direction of the tiling is said to be an *axis* (Figure 7a).

Definition 4.3. A location l is said to be *docking* if (1) $l \in A$, for some axis A, and the preceding axis location is occupied by a finished particle, or (2) $N(l)$ contains three consecutive locations occupied by finished particles (Figure 7b).

Definition 4.4. The set $F \subseteq V$ of locations occupied by finished particles that are not completely surrounded by other finished particles is said to be the *finished border* of the system.

In more detail, we assume that the particle system is initially connected, has a unique and predetermined "seed" particle, and that all other particles are initially unoriented. Algorithm 2 executes from the perspective of some particle p as follows (accompanying pseudocode is included below). As in Algorithm 1, p must first gain orientation; so, on activation, if p is not yet oriented it checks to see if any $q \in N(p)$ is oriented. If so, it orients itself toward q. Additionally, p sets its initial state based on that of q: if q

is the seed or is "walking"—a state reserved for particles traversing the border—p also becomes "walking"; otherwise, p becomes a "follower". Whether p becomes "walking" or a "follower", it performs careful checks on its movements to ensure that $N(p)$ does not contain any unoriented particles, since moving away from such particles before they gain orientation can disconnect the system.

As a "follower", p performs a series of handover expansions and contractions with its parents and children in the spanning forest as it is led toward the finished border of the system. Once p reaches the border, which it detects by finding a finished particle in its neighborhood, it also becomes "walking". As a "walking" particle, p first uses local rules to determine if its current location is docking. If so, p becomes "finished" and ceases to perform any further actions; otherwise, it expands to the next clockwise location along the finished border if it is unoccupied. Particle p continues to traverse the border in this way until a docking location is found, which at worst is at the end of the first axis it encounters.

Algorithm 2 Hole Elimination on a Particle p

Phase 1: ORIENTATION
while $p.state = unoriented$ **do**
  if $\exists q \in N(p)$ with $q.state \neq unoriented$ **then**
    $p.parent \leftarrow q$
    if $p.parent.state = seed$ or $walking$ **then** $p.state \leftarrow walking$
    else $p.state \leftarrow follower$

Phase 2: MOVEMENT
while $p.state \neq finished$ **do**
  if p is contracted **then**
    if $p.state = follower$ **then**
      if $\exists q \in N(p)$ with $q.state = finished$ **then**
        $p.parent \leftarrow q$
        $p.state \leftarrow walking$
      else if $p.parent$ is expanded **then**
        handover expand with $p.parent$
    else if $p.state = walking$ **then**
      if p occupies a docking location **then** $p.state \leftarrow finished$
      else if the next clockwise location l on the finished border is unoccupied **then**
        expand toward l
  else if (p is expanded) $\wedge$ ($!\exists q \in N(p)$ with $q.state = unoriented$) **then**
    if p has a child q **then** handover contract with q
    else contract

4.2.1 Correctness

We now show that at termination of Algorithm 2, a particle system cannot contain any holes, thus solving hole elimination.

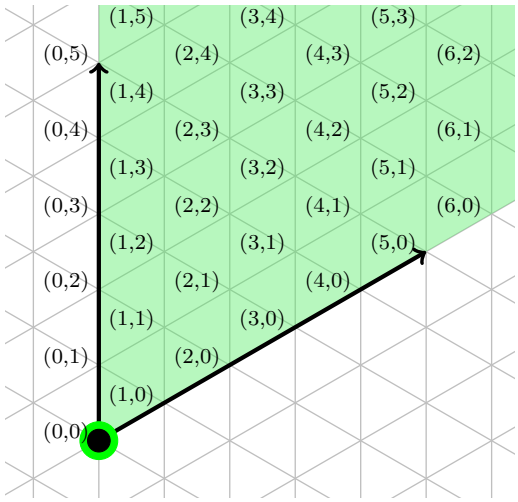


Figure 8: Locations composing a wedge and their corresponding coordinates from $\mathcal{C}$.

Definition 4.5. The set $W \subseteq V$ of locations lying between two adjacent axes, including the nodes of the axes themselves, is said to be a *wedge*.

For any wedge W, we can define a coordinate system $\mathcal{C} : \{(x, y) : x, y \in \mathbb{N}\} \to W$ as depicted in Figure 8. The seed is always positioned at the origin, and the axes of W also act as the axes of the coordinate system. As such, it suffices to show that the forthcoming arguments hold in the context of a single wedge, since they can be applied by rotational symmetry to any other wedge as well.

We define a relation $\leq$ on the locations $(x, y), (x', y') \in \mathcal{C}$ as follows:

$$(x, y) \leq (x', y') \iff x \leq x' \wedge y \leq y'.$$

It is easily seen that $\leq$ is a partial ordering. From this, we can define the strict interpretation of the relation:

$$(x, y) < (x', y') \iff (x, y) \leq (x', y') \wedge (x, y) \neq (x', y').$$

We use this relation to prove a lemma that is central to our proof of correctness.

Lemma 4.2. *If a location (x, y) in a wedge W is docking, then every location (x', y') such that $(x', y') < (x, y)$ is occupied by a finished particle.*

Proof. Argue by induction on n, the number of finished particles in W. First suppose that $n = 1$. Then the only finished particle in W is the seed positioned at (0,0), and the only docking locations are $(1, 0)$ and $(0, 1)$. Since $(0, 0) < (1, 0)$ and $(0, 0) < (0, 1)$, the statement holds. Now suppose that $n > 1$. Consider any docking location in W, say (x, y). We examine the newly created docking location(s) l when a particle p finishes at (x, y), becoming the $(n + 1)$-th finished particle.

Case 1: l satisfies (1). Then l is on an axis, and p is its preceding axis particle; suppose that p finishes at $(x, 0)$. Then l is location $(x+1, 0)$. The statement holds for $(x+1, 0)$

since by the induction hypothesis, every location $(x', 0)$ with $0 \leq x' < x$ is occupied by a finished particle and p has just finished at $(x, 0)$. An analogous argument holds if p instead finishes at $(0, y)$.

Case 2: l satisfies (2). After p has finished, all locations (x', y') with $(x', y') < (x, y)$ are occupied by finished particles by the induction hypothesis, so the candidates for l are $(x-1, y+1), (x, y+1), (x+1, y)$ and $(x+1, y-1)$. Observe that none of these candidates can be on an axis, as they would have already become docking due to (1). Suppose to the contrary that some axis location, say $(x', 0)$, becomes docking by satisfying (2) only; that is, its preceding axis location $(x'-1, 0)$ is unoccupied. Then one of its three finished neighbors must occupy location $(x'+1, 0)$; more pertinently, $(x'+1, 0)$ was docking before $(x', 0)$ was. This is a contradiction of the induction hypothesis, since $(x'-1, 0) < (x'+1, 0)$ but $(x'-1, 0)$ is unoccupied. An analogous argument holds for such a location on the y-axis.

Suppose $(x+1, y)$ becomes docking by satisfying (2). None of its three consecutive finished neighbors may occupy locations (x', y') such that $(x', y') > (x, y)$ as this would contradict the induction hypothesis, since p has only just become finished. Thus, its finished neighbors must occupy $(x, y), (x+1, y-1)$, and $(x+2, y-1)$. Therefore, the statement holds for $(x+1, y)$ since by the induction hypothesis all locations (x', y') such that $(x', y') \leq (x, y)$ are occupied by finished particles, as are all locations (x', y') such that $(x', y') \leq (x+1, y-1)$. A symmetric argument shows that the statement holds true for $(x, y+1)$ as well.

Now suppose $(x-1, y+1)$ becomes docking by satisfying (2). There cannot be a finished neighbor at location $(x, y+1)$, since $(x, y+1) > (x, y)$ and p has only just become finished, contradicting the induction hypothesis. Similarly, there cannot be a finished neighbor at $(x-1, y+2)$, since $(x-1, y+2) > (x-1, y+1)$ and $(x-1, y+1)$ has only just become docking. Since this location satisfies (2), two of its three consecutive finished neighbors must occupy locations $(x-2, y+1)$ and $(x-1, y)$. Therefore, the statement holds for $(x-1, y+1)$ since by the induction hypothesis all locations (x', y') such that $(x', y') \leq (x-2, y+1)$ are occupied by finished particles, as are all locations (x', y') such that $(x', y') \leq (x-1, y)$. A symmetric argument applies to $(x+1, y-1)$.

□

Theorem 4.3. *A particle system entirely composed of finished particles contains no holes.*

Proof. Suppose to the contrary that there exists a hole H in G. Then there must exist a location in H with coordinates (x, y) in a wedge it is contained in such that $(x+1, y)$ is occupied by a finished particle; otherwise, all locations (x', y) with $x' > x$ are unoccupied which causes H to be infinite, a contradiction. Thus, there is indeed such an unoccupied location (x, y) with $(x+1, y)$ occupied by a finished particle. However, this is a contradiction of Lemma 4.2 since $(x+1, y)$ is docking and $(x+1, y) > (x, y)$. We conclude that no hole could possibly exist in the finished structure. □

4.2.2 Convergence

We now show that the actions taken by Algorithm 2 preserve the connectivity and spanning forest structures of the particle system throughout its execution, and go on to prove a tight bound on the algorithm's complexity.

Definition 4.6. An action taken by a particle is said to be *safe* if it maintains both the connectivity of the overall particle system as well as the spanning forest of trees rooted at the finished border built during orientation.

Lemma 4.4. *The actions of Algorithm 2 are safe.*

Proof. Argue by induction on i, where a_i is the i-th action performed by particles in the system. Clearly, the initial configuration before a_1 is connected by assumption, and the spanning forest is trivially maintained as it is nonexistent. Now suppose that for some $i > 0$, each action $a_1, a_2, \ldots, a_i$ was safe. First, we suppose a_{i+1} only affects one particle, say p. If p was idle, then it becomes either a walking particle rooted to the finished border or a follower in a spanning tree. In either case, the system remains connected since p does not move and the spanning forest is only added to, which is safe. If p was a follower, it either becomes walking or contracts. In the former case, p merely acknowledges adjacency to the finished border by becoming a root; in the latter, p has no child in its spanning tree, nor does it have idle neighbors. Thus, its contraction is safe. If p was walking, then it can either expand—which certainly is safe—or contract, which is safe by the same reasoning as above for contractions of followers.

In the case that a_{i+1} affects two particles, a handover is performed between them. Handovers are guaranteed to maintain connectivity, and specifically do not change the parent/child relationships between particles in this algorithm, thus maintaining the spanning forest structure. □

Definition 4.7. An asynchronous *round* is complete after every unfinished particle has been given a chance to act at least once.

The proof of the following lemma follows from a similar analysis of spanning tree behavior for universal coating in self-organizing particle systems, which can be found in [13].

Lemma 4.5. *In Algorithm 2, every particle in a spanning tree of size k will become either finished or walking after $O(k)$ rounds.*

Theorem 4.6. *Algorithm 2 terminates in the worst case of $\Theta(n)$ rounds, where n is the number of particles in the system.*

Proof. A lower bound of $\Omega(n)$ rounds is easily shown; consider a system of idle particles initially in a line, say $p_0, p_1, p_2, \ldots, p_{n-1}$, where p_0 is the seed. If each round has the activation order of $p_{n-1}, p_{n-2}, \ldots, p_0$, then in round i only one new particle (namely, p_i) will become oriented. Thus, since orientation is achieved in the worst-case of $\Theta(n)$ rounds, the entire algorithm can be no faster.

Consider any wedge W and the set of spanning trees rooted in this wedge, say $\mathcal{T}_W = \{T_1, T_2, \ldots, T_m\}$, where $i < j$ implies that the root of T_i is positioned further clockwise along the finished border than that of T_j. Now consider any particle $p \in W$. As in the case

for the lower bound, after at most $O(n)$ rounds p will become oriented, so suppose $p \in T_k$. We first consider the case that p becomes walking in wedge W. p might be temporarily blocked by elements of other trees sometime before it becomes walking; in the worst case, T_k may be blocked by T_{k-1}, which in turn is blocked by T_{k-2} and so on until T_1, which cannot be blocked. By Lemma 4.5, p needs at most $O(\sum_{i=1}^{k} |T_i|)$ rounds to become walking, which is effectively $O(n)$ rounds since the number of particles contained in the first k trees is less than the total number of particles. In the other case, p may become walking in a different wedge—for depending on the shape of T_k, another axis in the system may have grown to intersect it as it was clearing. This effectively prunes branches from T_k, allowing it to clear in a lesser amount of rounds. As a result, p may now be contained in one of these pruned branches, say B_k, which now acts as smaller tree rooted to this intersecting axis. Since $|B_k| < |T_k|$, by Lemma 4.5 p certainly waits no longer than it would have before the intersection to become walking.

To become finished, p must traverse the finished border until it finds a docking position. The wedge containing p certainly has docking locations at the ends of its axes, and others with three finished particles in their neighborhoods may appear during the algorithm's execution. In the worst case, however, the only reachable docking position is at the end of the x-axis, since all unfinished particles in this wedge must traverse the border clockwise without passing one another and finish at the end of the axis one-by-one. Now, since all trees rooted in front of T_k (or B_k) have cleared, p can only be blocked by other walking particles, some of which must be unblocked and advancing toward the docking position in every round. As the distance along the finished border from p to the end of the x-axis is at most the number of particles in the system, p needs at most $O(n)$ rounds to discover a docking position and become finished.

Thus, p requires at most $O(n)$ rounds to become oriented, $O(n)$ rounds to become walking, and $O(n)$ rounds to become finished, resulting in a total of $O(n)$ rounds. As the choice of p was arbitrary, we conclude that every particle in the system will be finished in a linear number of rounds. □

4.2.3 Practical Performance and Competitive Analysis

Lastly, we performed a competitive analysis between Algorithm 2 and the Hexagon Shape Formation algorithm [11], which technically achieves optimal compression by creating even layers of particles over a seed particle. This algorithm was shown in [11] to terminate in $O(n^2)$ work for a particle system of size n, and served as a baseline comparison for Algorithm 2, which also requires $\Theta(n^2)$ work. This tight bound for work follows from the proofs of worst-case work requirements for line formation in [12].

The experiment was performed as follows. Both algorithms were implemented on our self-organizing particle systems simulator, which—among other things—keeps track of the total number of particle movements during an algorithm's execution. Both algorithms were instantiated with the same randomly generated initial configurations which varied from 32–2048 particles in size, on expectation. At the termination of each run, the size of the particle system and the total number of movements (work) performed were recorded as a pair.

Figure 9 depicts the results of about 12,000 runs of hexagon shape formation, and just

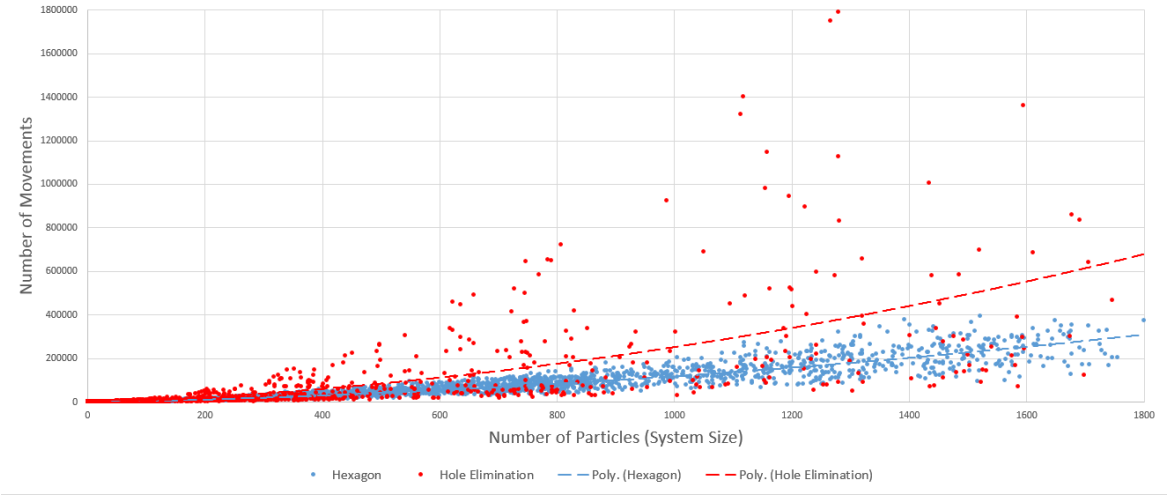


Figure 9: Particle system size versus work for hexagon shape formation (blue) and hole elimination (red).

under 2,300 runs of hole elimination. The discrepancy in the number of runs between the two algorithms is closely linked to the resulting trends; for small particle systems—say, $n \leq 300$—the algorithms perform roughly the same, but as the sizes scale larger, the hole elimination algorithm begins to vary widely, taking significantly more time to complete runs. While occasionally outperforming the hexagon formation algorithm slightly, hole elimination often takes twice to five times as much work to terminate. In accordance with the theoretical result that both algorithms require at most $O(n^2)$ work, second-order polynomial trendlines were fitted to each algorithm's data. These trendlines indicate that hole elimination performs consistently worse than the hexagon formation algorithm by a factor of about 2.25 and is far less consistent in the amount of work required.

5 A Markov Chain Algorithm for Compression

We now present a Markov chain algorithm for compression (Algorithm 3) [5], which was developed as a joint work with Dr. Dana Randall and Sarah Cannon at the Georgia Institute of Technology. This algorithm takes a significantly different approach than the previous two algorithms described; it does not use a predetermined seed particle which the rest gravitate toward, nor does it require an underlying structure to be maintained throughout its execution. Furthermore, this algorithm does not require particles to communicate with one another or set states; they need only be able to examine their local neighborhood for occupied and vacant locations. Instead, this algorithm uses a Markov chain $\mathcal{M}$ and a bias parameter $\lambda > 0$ which influences how strongly the particles favor being incident to triangles; $\lambda > 1$ influences particles to favor triangles, while $0 < \lambda < 1$ influences particles to avoid them. We will show in Lemma 5.2 that maximizing the number of triangles in the system is equivalent to minimizing the perimeter—and thus to compressing the system—and will show that for $\lambda > 5$, α-compression is achieved with high probability. For a particle system with n particles in a connected configuration σ, let the *perimeter* of σ, denoted $p(\sigma)$,

be the length of the walk around the (unique) outer border of the particles. Note that for any σ on n particles, $\Theta(n) = p_{min}(n) \leq p(\sigma) \leq p_{max}(n) = 2n - 2$, where $p_{min}(n)$ and $p_{max}(n)$ denote the minimum possible perimeter (i.e. that of a hexagon or a hexagon with the outermost layer incomplete) and the maximum possible perimeter (that of a tree with no induced triangles), respectively.

Definition 5.1. Given an $\alpha > 1$, a connected configuration σ on n particles is said to be *α-compressed* if $p(\sigma) \leq \alpha \cdot p_{min}(n)$.

Despite its apparent simplicity, this algorithm is able to maintain several important properties throughout its execution which will be used to apply important tools from Markov chain analysis: (1) the particle system remains connected and without holes, (2) all moves made are reversible; i.e. for any move made, there is a nonzero probability that it will be undone in the next step, and (3) the state space of the Markov chain is connected; that is, for any two possible configurations of the particle system, there is a series of moves in the Markov chain that can take one to the other.

5.1 A Brief Overview of Markov Chains

Algorithm 3 implements a Markov chain, which is a memoryless stochastic process defined on a finite set of configurations Ω. The transition matrix P on $\Omega \times \Omega \to [0, 1]$ is defined such that for any $x, y \in \Omega$, $P(x, y)$ is the probability of moving from state x to state y in one step. The t-step transition probability $P^t(x, y)$ is the probability of moving from x to y in exactly t steps.

Definition 5.2. A Markov chain is said to be *ergodic* if it is *irreducible*—that is, for any $x, y \in \Omega$ there exists a t such that $P^t(x, y) > 0$—and *aperiodic*, that is, for any $x, y \in \Omega$, the g.c.d. $\{t : P^t(x, y) > 0\} = 1$.

Any finite, ergodic Markov chain converges to a unique distribution π, i.e., for all $x, y \in \Omega$, $\lim_{t\to\infty} P^t(x, y) = \pi(y)$. To reason about this unique limiting probability distribution, known as the *stationary distribution*, we rely on the *detailed balance condition* in the following lemma (See, e.g., [16]).

Lemma 5.1. *Let $\mathcal{M}$ be an ergodic Markov chain on a finite state state space Ω with transition probabilities $P(\cdot, \cdot)$. If $\pi' : \Omega \to [0, 1]$ is any function satisfying the detailed balance condition:*

$$\pi'(x)P(x, y) = \pi'(y)P(y, x),$$

and if it also satisfies $\sum_{x\in\Omega} \pi'(x) = 1$, then π' is the unique stationary distribution of $\mathcal{M}$.

A chain satisfying detailed balance for some π' is called *time-reversible*.

When we have a desired stationary distribution π on Ω, the Metropolis-Hastings algorithm [19] explains how to define the transition probabilities. Starting at a state x, we pick a neighbor $y \in \Omega$ uniformly with probability $1/(2\Delta)$, where Δ is the maximum number of neighbors of any configuration, and move to to y with probability $\min\{1, \pi(y)/\pi(x)\}$; with remaining probability we stay at x, and repeat. Using detailed balance, it is easy to verify if the state space is connected, then π must be the stationary distribution. While calculating

$\pi(x)/\pi(y)$ seems to require global knowledge, this ratio can often be calculated using only local information when many terms cancel out. In our case, the Metropolis probabilities are simply $\min(1, \lambda^{\delta(x,y)})$, where $\delta(x, y)$ is the local change in the number of triangles in the two configurations.

See [23, 28] for further details on Markov chains.

5.2 Algorithm Description

Before detailing the Markov chain for compression, we define two properties which ensure that the particle system remains connected and without holes throughout the execution of the algorithm. Let $l, l' \in V(G_{eqt})$ be neighboring locations, and let $\mathbb{S}$ be the set of particles adjacent to both l and l'. Note that $\mathbb{S}$ does not include particles possibly located at l or l', and that $|\mathbb{S}| \in \{0, 1, 2\}$.

Definition 5.3. Locations l and l' are said to satisfy *Property 1* if $|\mathbb{S}| \in \{1, 2\}$ and every particle in $N(l \cup l')$ is connected to a particle in $\mathbb{S}$ by a path contained in $N(l \cup l')$.

Definition 5.4. Locations l and l' are said to satisfy *Property 2* if $|\mathbb{S}| = 0$, l and l' each have at least one neighbor, all particles in $N(l) \setminus l'$ are connected by paths contained in this set, and all particles in $N(l') \setminus l$ are connected by paths contained in this set.

We now present the Markov chain $\mathcal{M}$ for compression, which is written from the global perspective of the whole particle system. Note that $\mathcal{M}$ is easily translated to a fully distributed, local, asynchronous protocol; each particle can run lines 4–8 of Algorithm 3 independently and indefinitely, resolving any contentions in which two particles try to expand into the same location with a coin toss or some other method of arbitration.

Algorithm 3 Markov chain $\mathcal{M}$ for Compression

1: **Input:** starting configuration σ_0 with no holes and bias parameter $\lambda > 1$.
2: **while true do**
3: Activate particle P, chosen uniformly at random from among all particles; let l be its location.
4: Choose a neighboring location l' of P uniformly at random from the six possibilities; generate a random number $q \in (0, 1)$.
5: **if** l' is a vacant location **then** P expands to occupy both l and l'.
6: Let t be the number of triangles P is incident to at position l, and t' be the number of triangles P is incident to at position l'.
7: **if** (l does not have exactly five neighboring particles) $\wedge$ (locations l and l' satisfy Property 1 or Property 2) $\wedge$ ($q < \lambda^{t'-t}$) **then** P contracts to l'.
8: **else** P contracts back to l.

5.3 Relevant Proofs

In [5], we provide detailed proofs for many aspects of Algorithm 3, including maintenance of connectivity and holelessness, the ergodicity of $\mathcal{M}$, and ranges of λ for which α-compression is possible and impossible. For sake of clarity and brevity, we leave many arguments and

statements of supporting lemmas in [5] for reference, and focus here on the most critical results for compression.

In line 7 of Algorithm 3, we provide the conditions on which a particle will commit to a movement that changes its position. While the first two conditions have to do with the safety of a movement—i.e. not disconnecting or creating a hole in the system and preserving reversibility—the third addresses the geometric motivation for this algorithm. Specifically, for a $\lambda > 1$, $\lambda^{t'-t}$ will always be greater than 1 if $t' \geq t$, and still may be greater than 1 when $t' < t$ if λ is large enough. Thus, when a particle locally detects that an available movement can increase the number of triangles it is incident to, it always moves, and even when a movement could temporarily decrease the triangle count, it might move anyway to possibly access more optimal moves in the future. The following lemma proves that maximizing the number of triangles contained in the system minimizes the perimeter, which is the local property we will exploit when progressing toward compression as desired.

Lemma 5.2. *For a connected configuration σ on n particles with no holes, the number of triangles $t(\sigma) = 2n - p(\sigma) - 2$.*

Proof. We count particle-triangle incidences, of which there are $3t(\sigma)$. Counting another way, every particle has six incident triangles, except for those along the perimeter. Consider a (clockwise) traversal of the perimeter, starting at an arbitrary location. At each particle this perimeter traversal passes, the exterior angle is $120^\circ, 180^\circ, 240^\circ, 300^\circ$, or 360°. These correspond to the particle "missing" 2, 3, 4, 5, or 6 of its possible six incident triangles, respectively. If the traversal passes the same vertex multiple times, we count the appropriate exterior angle each time the vertex is visited. We now use the fact the the sum of the exterior angles in this traversal is $180 \cdot p(\sigma) + 360$, so the total number of missing triangles for the vertices on the perimeter is $3p(\sigma) + 6$. Counting this way, the number of particle-triangle incidences is $6n - 3p(\sigma) - 6$. This implies $3t(\sigma) = 6n - 3p(\sigma) - 6$, implying the lemma. □

Corollary 5.3. *$t(\sigma)$ is maximized when $p(\sigma) = p_{min}(n)$.*

Before turning our attention to proofs of correctness for Algorithm 3, we state in passing that Markov chain $\mathcal{M}$ is indeed ergodic, implying that its stationary distribution is unique. Furthermore, we use detailed balance to give an expression for the stationary distribution π of $\mathcal{M}$. A proof of the ergodicity of $\mathcal{M}$ is given in [5].

Lemma 5.4. *The stationary distribution π of $\mathcal{M}$ is given by*

$$\pi(\sigma) = \frac{\lambda^{t(\sigma)}}{Z}, \text{ where } Z = \sum_{\sigma} \lambda^{t(\sigma)} \text{ is the normalizing constant.}$$

Proof. We confirm that π is the stationary distribution with detailed balance. Let σ and τ be any particle configurations with $\sigma \neq \tau$ such that $P(\sigma, \tau)$ is nonzero. In fact, by Lemma 5.1, we conclude $P(\tau, \sigma) > 0$. Suppose particle p moves from location l in σ to neighboring location l' in τ. Let t be the number of triangles on which p is incident when it is in location l, and let t' be the number of triangles on which p is incident when it is in location l'. This implies $t(\sigma) - t(\tau) = t - t'$. W.l.o.g., let $t' < t$. Then,

$$P(\sigma, \tau) = \frac{1}{n} \cdot \frac{1}{6} \cdot \lambda^{t'-t} \text{ and } P(\tau, \sigma) = \frac{1}{n} \cdot \frac{1}{6} \cdot 1.$$

We now show σ and τ satisfy the detailed balance condition.

$$\pi(\sigma)P(\sigma,\tau) = \frac{\lambda^{t(\sigma)}}{Z} \cdot \frac{\lambda^{t'-t}}{6n} = \frac{\lambda^{t(\tau)}}{Z \cdot 6n} = \pi(\tau)P(\tau,\sigma).$$

We conclude π is the stationary distribution of $\mathcal{M}$. □

Corollary 5.5. *The stationary distribution π of $\mathcal{M}$ is also given by*

$$\pi(\sigma) = \frac{\lambda^{-p(\sigma)}}{Z}, \text{ where } Z = \sum_{\sigma} \lambda^{-p(\sigma)} \text{ is the normalizing constant.}$$

Proof. We use Lemmas 5.2 and 5.4:

$$\pi(\sigma) = \frac{\lambda^{t(\sigma)}}{\sum_\sigma \lambda^{t(\sigma)}} = \frac{\lambda^{2n-p(\sigma)-2}}{\sum_\sigma \lambda^{2n-p(\sigma)-2}} = \frac{\lambda^{2n-2}}{\lambda^{2n-2}} \cdot \frac{\lambda^{p(\sigma)}}{\sum_\sigma \lambda^{p(\sigma)}} = \frac{\lambda^{-p(\sigma)}}{\sum_\sigma \lambda^{-p(\sigma)}} = \frac{\lambda^{-p(\sigma)}}{Z}.$$

□

We are now ready to show that, provided λ and n are large enough, if $\mathcal{M}$ is at stationarity then with high probability the particles are in an α-compressed configuration for some $\alpha > 1$. Note that while α can be as arbitrarily close to 1 as desired, other parameters must shift accordingly to achieve compression. We begin with a crucial counting lemma.

Lemma 5.6. *The number of connected configurations with no holes and perimeter k is at most 5^k.*

Proof. Consider any connected configuration σ. Consider a clockwise traversal of its perimeter, beginning at the lowest leftmost particle of σ. For every edge of this perimeter traversal, note the location left of the edge is unoccupied; otherwise, the edge would not be traversed in a clockwise perimeter walk. At each step of the perimeter traversal, the perimeter can continue straight, turn left by 60°, turn right by 60°, turn right by 120°, or turn right by 180°. The perimeter can never turn left by 120° or 180°. Thus, at each step of the perimeter traversal, there are at most five possible locations for the next particle on the perimeter. We conclude that there are at most 5^k configurations of perimeter k, each specified by the directions of the turns on the walk traversing its perimeter. □

In the next section, we will discuss at greater length how this bound is by no means tight, and how improvements on the base of this exponent will lead directly to better (lower) values for λ^* in the result below. The proof of this result is included in Appendix **??**.

Theorem 5.7. *For any $\alpha > 1$, there exists $\lambda^* = 5^{\frac{\alpha}{\alpha-1}} > 5$, $n^* \geq 0$, and $\gamma < 1$ such that for all $\lambda > \lambda^*$ and $n > n^*$, the probability that a random sample σ drawn according to the stationary distribution π of $\mathcal{M}$ is not α-compressed is exponentially small:*

$$\mathbb{P}(p(\sigma) \geq \alpha \cdot p_{min}(n)) < \gamma^{\sqrt{n}}.$$

Proof. Let S_α be the set of configurations of perimeter at least $\alpha \cdot p_{min}(n)$. Let σ_{min} be a configuration of n particles achieving the minimum perimeter $p_{min}(n)$. For convenience, we

define the *weight* of a configuration σ to be $w(\sigma) = \lambda^{-p(\sigma)}$ and the weight of a set $S \subseteq \Omega$ to be $w(S) = \sum_{\sigma \in S} w(\sigma)$. We show

$$\pi(S_\alpha) = \frac{w(S_\alpha)}{Z} < \frac{w(S_\alpha)}{w(\sigma_{min})} \leq \gamma^{\sqrt{n}}.$$

The first equality is the definition of π; the next inequality follows from the definitions of Z and w. We focus on the last inequality. By Lemma 5.6, there are at most 5^k configurations of perimeter k; the weight of any configuration of perimeter k is $\lambda^{-p(\sigma)} = \lambda^{-k}$. We use this to sum over the configurations in S_α.

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \leq \frac{\sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^k \lambda^{-k}}{\lambda^{-p_{min}}} = \sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^k \lambda^{-k} \lambda^{p_{min}} = \sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^{(1-\log_5 \lambda)k + (\log_5 \lambda) p_{min}}.$$

Using the inequality $p_{min} \leq k/\alpha$, it follows that

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \leq \sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^{(1-\log_5 \lambda)k + (\log_5 \lambda)(k/\alpha)} = \sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^{(1-(1-1/\alpha)\log_5 \lambda)k}.$$

Provided $\lambda > \lambda^* = 5^{\frac{\alpha}{\alpha-1}}$, the constant $-c_1 = 1 - (1 - 1/\alpha)\log_5 \lambda < 0$, and we can again use the inequality $k \geq \alpha \cdot p_{min}$. We also note that $p_{min}(n) > 2\sqrt{n}$:

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \leq \sum_{k=\lceil \alpha p_{min} \rceil}^{2n-2} 5^{-c_1 \cdot 2\alpha\sqrt{n}} \leq 2n \left(5^{-2\alpha c_1}\right)^{\sqrt{n}}.$$

It follows that there exists $\gamma < 1$ and n^* such that for all $n \geq n^*$,

$$\mathbb{P}\left(p(\sigma) \geq \alpha \cdot p_{min}\right) = \pi(S_\alpha) \leq \frac{w(S_\alpha)}{w(\sigma_{min})} \leq 2n \left(5^{-2\alpha c_1}\right)^{\sqrt{n}} < \gamma^{\sqrt{n}}.$$

□

Corollary 5.8. *For any $\lambda > 5$, there exists $\alpha = \log_5 \lambda / (\log_5 \lambda - 1), n^* \geq 0$, and $\gamma < 1$ such that for all $n > n^*$, a random sample σ drawn according to the stationary distribution π of $\mathcal{M}$ satisfies*

$$\mathbb{P}(p(\sigma) \geq \alpha \cdot p_{min}(n)) < \gamma^{\sqrt{n}}.$$

For small λ, $\mathcal{M}$ can also be used to achieve α-expansion of the particle system, which is formalized below. Notably, if a configuration is α-expanded then it has linear perimeter, implying that it cannot be β-compressed for any constant β. We show that, provided n is large enough, for all $0 < \lambda < \sqrt{2}$ there is a constant α such that if $\mathcal{M}$ is at stationarity then with high probability the configuration is α-expanded. This is important because it shows that although $\lambda > 1$ causes particles to favor being incident to triangles, it is not enough to guarantee compression. The proof of this result is given in Appendix **??**.

Definition 5.5. Given an $0 < \alpha < 1$, a connected configuration σ on n particles is said to be α-expanded if $p(\sigma) \geq \alpha \cdot p_{max}(n)$.

Theorem 5.9. *For all $0 < \alpha < 1$, there exists $\lambda^* = \lambda^* \cdot \alpha < \sqrt{2}, n^* \geq 0$, and $\gamma < 1$ such that for all $\lambda < \lambda^*$ and $n > n^*$, the particles achieve α-expansion with high probability: for a configuration σ drawn at random according to the stationary distribution π,*

$$\mathbb{P}(p(\sigma) < \alpha \cdot p_{max}(n)) < \gamma^{\sqrt{n}}.$$

Proof. We let

$$\lambda < \lambda^* = 5^{\frac{\alpha - (1/2)\log_5 2}{\alpha - 1}} < 5^{\frac{-(1/2)\log_5 2}{-1}} = \sqrt{2}.$$

Let S_α be the set of configurations of perimeter at most $\alpha \cdot p_{max}$. Let σ_{max} be a configuration achieving the maximum perimeter $p_{max} = 2n - 2$. We note the number N of configurations achieving this maximum is bounded below by $2^{n-1} = 2^{p_{max}/2}$; this is the number of paths where every step is up or up-right, all of which have no triangles and thus maximal perimeter. We show

$$\pi(S_\alpha) = \frac{w(S_\alpha)}{Z} < \frac{w(S_\alpha)}{N \cdot w(\sigma_{max})} \leq \gamma^{\sqrt{n}}.$$

We focus on proving the last of the inequalities above. The number configurations of perimeter k is at most 5^k by Lemma 5.6; the weight of any configuration of perimeter k is $\lambda^{-p(\sigma)} = \lambda^{-k}$. Applying this, we see

$$\frac{w(S_\alpha)}{N \cdot w(\sigma_{max})} \leq \frac{\sum_{k=p_{min}}^{\lfloor \alpha p_{max} \rfloor} 5^k \lambda^{-k}}{2^{p_{max}/2} \cdot \lambda^{p_{max}}} = \sum_{k=p_{min}}^{\lfloor \alpha p_{max} \rfloor} 5^k \lambda^{-k} \lambda^{p_{max}} \cdot 2^{-\frac{p_{max}}{2}} < \sum_{k=p_{min}}^{\lfloor \alpha p_{max} \rfloor} 5^k \lambda^{-k} \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}}.$$

Recalling that $\lambda < \sqrt{2}$ and $k < \alpha \cdot p_{max}$, we see

$$\frac{w(S_\alpha)}{N \cdot w(\sigma_{max})} \leq \sum_{k=p_{min}}^{\lfloor \alpha \cdot p_{max} \rfloor} 5^k \lambda^{-k} \left(\frac{\lambda}{\sqrt{2}}\right)^{(k/\alpha)} = \sum_{k=p_{min}}^{\lfloor \alpha \cdot p_{max} \rfloor} 5^{k\left(1 - \log_5 \lambda + \frac{\log_5 \lambda}{\alpha} - \frac{\log_5 2}{2\alpha}\right)}$$

We let c_3 be

$$-c_3 = 1 - \log_5 \lambda + \frac{\log_5 \lambda}{\alpha} - \frac{\log_5 2}{2\alpha}.$$

We are interested in the case $-c_3 < 0$, so we solve for λ and see that $-c_3 < 0$ precisely when $\lambda < \lambda^*$, a condition we know to hold. It follows there exists constant $c_3 > 0$ such that

$$\frac{w(S_\alpha)}{N \cdot w(\sigma_{max})} \leq \sum_{k=p_{min}}^{\lfloor \alpha \cdot p_{max} \rfloor} 5^{-c_3 k} \leq \sum_{k=p_{min}}^{\lfloor \alpha \cdot p_{max} \rfloor} 5^{-c_3 \cdot 2\sqrt{n}} \leq 2n \cdot 5^{-2c_3\sqrt{n}}.$$

We conclude there exists $n^* \geq 0$ and $\gamma < 1$ such that for all $n \geq n^*$,

$$\mathbb{P}\left(p(\sigma) \leq \alpha \cdot p_{max}\right) = \pi(S_\alpha) \leq \frac{w(S_\alpha)}{N \cdot w(\sigma_{max})} \leq 2n \cdot 5^{-c_3 \cdot 2 \cdot \sqrt{n}} < \gamma^{\sqrt{n}}.$$

□

Corollary 5.10. *For all $\lambda < \sqrt{2}$, there exists a constant α such that with high probability a sample drawn according to the stationary distribution π of $\mathcal{M}$ is α-expanded.*

Proof. This follows from Theorem 5.9; for $\lambda < \sqrt{2}$, there exists an alpha such that $\lambda < \lambda^* \cdot \alpha$. □

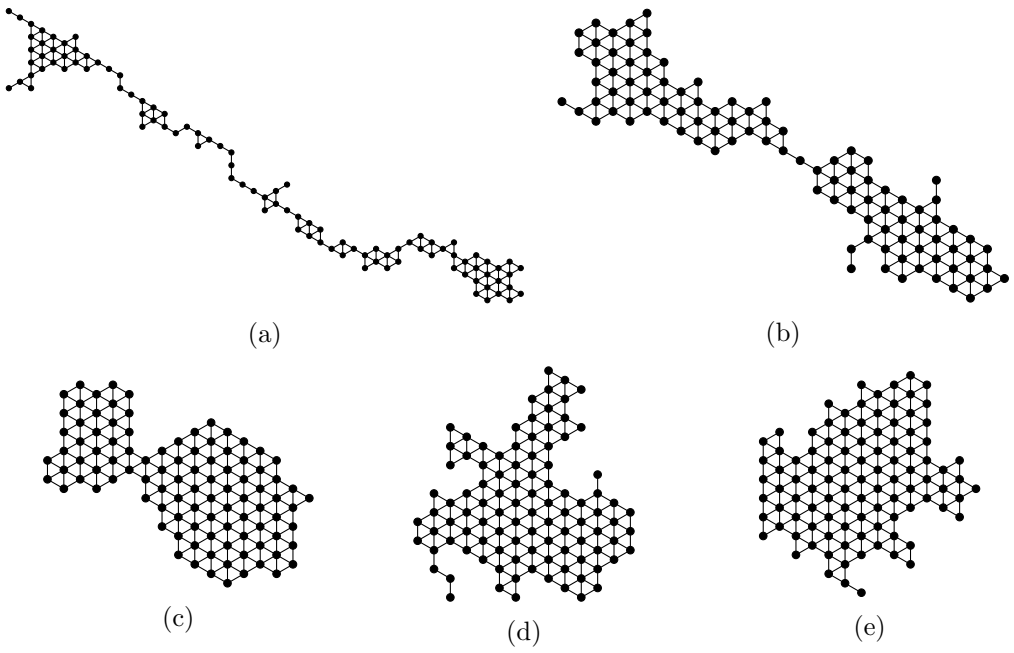


Figure 10: 100 particles initially in a line after (a) 1 million, (b) 2 million, (c) 3 million, (d) 4 million, and (e) 5 million iterations of Markov chain $\mathcal{M}$ with bias $\lambda = 4$.

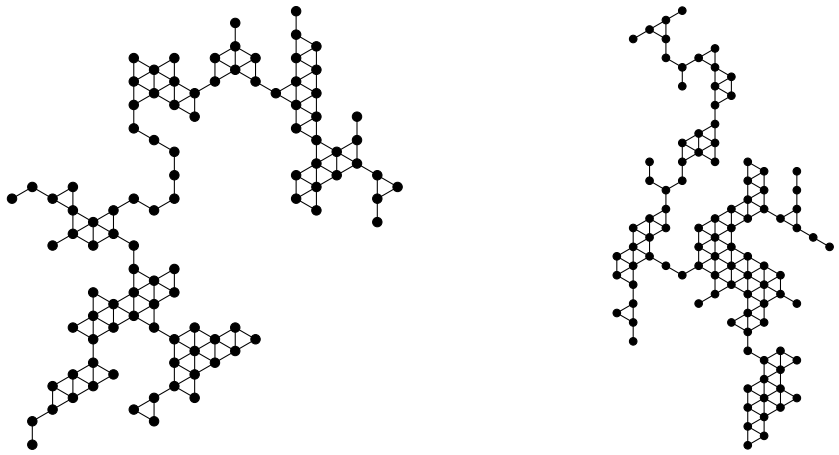
5.4 Improvements to Bounds on λ

Corollaries 5.8 and 5.10 create bounds for the bias parameter λ that categorize the behavior of the Markov chain. More specifically, we showed that for any $\lambda > 5$, there exists a constant α such that the probability of drawing a non-α-compressed random sample according to the stationary distribution of $\mathcal{M}$ is exponentially small. Additionally, for any $0 < \lambda < \sqrt{2}$, we have shown that there exists a constant fraction α such that if $\mathcal{M}$ is at stationarity then with high probability the particle system is α-expanded. These bounds are by no means tight, and in practice Markov chain $\mathcal{M}$ yielded good compression for values such as $\lambda = 4$, which was outside of our proven range. We simulated $\mathcal{M}$ with $\lambda = 4$ on a particle system of 100 particles initially configured as a line, and—as Figure 10 illustrates—compression occurs after only a relatively small number of iterations. Conversely, $\lambda = 2$ which theoretically still favors particles forming triangles does not appear to yield compression in practice. Starting from the same initial configuration of 100 particles in a line, Figure 11 shows that even after 20 million simulated steps of $\mathcal{M}$, the particles have not compressed.

In [5], we conjectured that there is a critical value λ_c such that for all $\lambda < \lambda_c$ the particles do not compress and for all $\lambda > \lambda_c$ compression is achieved. We now discuss improvements which close the gap between $\sqrt{2}$ and 5.

5.4.1 Lower Bound for Compression

To obtain the bound $\lambda > 5$, we used Lemma 5.6, which relied on counting the number of connected configurations with no holes and perimeter k. Since we assume valid configurations of particle systems contain no holes, it sufficed to count the number of walks of length



(a) (b)

Figure 11: 100 particles initially configured in a diamond shape with side lengths 10 after (a) 10 million and (b) 20 million iterations of $\mathcal{M}$ with bias $\lambda = 2$.

k around the perimeter. We concluded that, at most, there could be 5^k unique walks of length k.

It is not difficult to see, however, that many of these walks do not form valid perimeters of length k. On one extreme, the walk which always continues in the same direction—corresponding to a straight line of k particles—is not closed and therefore cannot possibly form a perimeter. On the other, the walk which chooses at every step to turn 180° overlaps itself many times, bouncing between only two particles. Thus, the count of 5^k perimeter walks—and the corresponding bound of $\lambda > 5$—is much too large. Here, we discuss a method for obtaining better bases for this exponent, which lead directly to lower λ_i such that for any $\lambda > \lambda_i$ there exists a constant α such that the probability of drawing a non-α-compressed random sample from the stationary distribution of $\mathcal{M}$ is exponentially small.

Once again, we count perimeter walks of length k. However, instead of considering the number of directional choices a walk W can make at any one particle, we count the number of valid sequences of turns that an i-particle subset of W could take; call this number $c(i)$. We then can build perimeter walks of length k by chaining together $\lceil k/i \rceil$ i-particle subset walks, and conclude that the number of valid perimeter walks is at most $c(i)^{k/i}$. Therefore, by Theorem 5.7, we obtain a better lower bound for λ: $\lambda > \lambda_i = c(i)^{k/i}$.

The following are preliminary experimental results from a program we implemented to compute $c(i)$ for any $i \in \mathbb{N}$. The C++ code for generating these results can be found in Appendix A.1.

i	$c(i)$	5^i	λ_i
1	5	5	5.00000
2	23	25	4.79583
3	94	125	4.54684
4	374	625	4.39762
5	1,436	3,125	4.27987
6	5,450	15,625	4.19501
7	20,373	78,125	4.12648
8	75,572	390,625	4.07188
9	277,930	1,953,125	4.02607
10	1,016,475	9,765,625	3.98758
11	3,696,783	48,828,125	3.95435

From these experimental results alone, we obtain the better lower bound $\lambda > 3.955$, and expect that with larger test cases—and some appropriate optimization to the program's performance—these λ_i will converge to some tight bound λ_c.

5.4.2 Upper Bound for Non-Compression

To obtain the bound $\lambda < \sqrt{2}$, we counted the number of triangle-free, connected configurations on n particles (to be referred to as *trees*) which could be formed by placing particles successively either up or up-right of their predecessors. After placing the first particle, each of the following $n-1$ particles can be placed either up or up-right of their predecessors, so we conclude that there are at least $2^{n-1} = \sqrt{2}^{2n-2}$ unique trees on n particles. As is demonstrated in Theorem 5.9, since $2n-2$ is the maximum perimeter of a particle system on n particles, $\lambda < \sqrt{2}$.

Once again, we find that this counting method is not as accurate as it could be. For this upper bound for non-compression, 2^{n-1} drastically undercounts the number of trees on n particles. Certainly many trees exist that use placements outside of up and up-right; for example, a straight line of particles all placed down-right is a tree. This provides another opportunity for improving bounds on λ, this time by including more trees in our count to obtain a better upper bound for non-compression, say λ'_i, such that for all $0 < \alpha < 1$ and any $\lambda < \lambda'_i$, particles achieve α-expansion with high probability.

We proceed to count the number of trees on n particles, but instead of building the entire tree at once by sequentially placing single particles in set directions, we first count the number of trees with $i < n$ particles; call this number $c'(i)$. We can then construct a full tree with n particles by sequentially attaching any one of the $c'(i)$ smaller trees' upper-left-most or lower-left-most particle to the previous one's right-most particle, guaranteeing that the new structure is still triangle-free. This can be achieved with $\lceil (n-1)/i \rceil$ subtrees, so we conclude that the number of trees on n particles is at least $(2c'(i))^{(n-1)/i} = \sqrt[2i]{2c'(i)}^{2n-2}$. Therefore, by Theorem 5.9, we obtain a better upper bound for λ: $\lambda < \lambda'_i = \sqrt[2i]{2c'(i)}$.

The following are preliminary experimental results from a program we implemented to compute $c'(i)$ for any $i \in \mathbb{N}$. Due to the code's efficiency being relatively better than that for decreasing the $\lambda > 5$ bound, we were able to run this program on larger i. Once again, full C++ code for this program can be found in Appendix A.2.

i	$c'(i)$	λ'_i
1	1	1.41421
2	3	1.56508
3	9	1.61887
4	29	1.66123
5	95	1.68996
6	321	1.71380
7	1,102	1.73301
8	3,802	1.74810
9	13,138	1.76007
10	45,439	1.76979
11	157,254	1.77784
12	544,505	1.78461
13	1,885,989	1.79038
14	6,534,175	1.79536
15	22,643,242	1.79970
16	78,482,367	1.80351
17	272,068,967	1.80689
18	943,298,064	1.80991

From these results, we obtain the better upper bound for non-compression $\lambda < 1.810$. Once again, larger test cases do promise better bounds, but only marginally so. Thus, we have improved the window of uncertainty on behavior from $\sqrt{2} < \lambda < 5$ to $1.810 < \lambda < 3.955$.

6 Leader Election

Algorithms 1 and 2 rely on a predetermined seed particle to aid in propagating an orientation throughout the system. However, this skirts the issue of providing the anonymous, directionless particles with a way of determining a single leader. Moreover, this leader needs to know that it is the sole leader in the system; this problem is known as the *decision version of leader election*. In [12], we developed an algorithm for leader election in self-organizing particle systems which has the particles compete along borders they are adjacent to. While a particle may be incident to up to three different borders, there is only one outer border. The last remaining candidate particle on the outer border performs some additional checks to assure its solitude, and then declares leadership over the whole system.

In collaboration with Robert Gmyr at Universität Paderborn, Germany, we developed a simulation which accurately handles and depicts the complicated token-passing schemes that the leader election algorithm uses for particle communication. This simulation was the first attempt at providing detailed protocols for how the tokens were used, and was shown in the algorithm's presentation at DNA21.

7 Future Work

In this thesis, we have suggested three different notions of compression and provided algorithms which solve each of them. While each algorithm's analysis is fairly comprehensive, there are several areas in which compression could be further investigated.

In the case of Algorithm 3, we have shown its correctness in Theorem 5.7 for sufficiently large λ and n, and have additionally proven bounds for λ that control the behavior of the Markov chain $\mathcal{M}$ in Corollaries 5.8 and 5.10. However, even after considering the improvements to these bounds discussed in Section 5.4, we remain in search of a critical value λ_c above which λ influences a particle system to become α-compressed and below which λ influences a system to become α-expanded with sufficient iterations. Improving these bounds remains an active research effort.

Additionally, no effort has been made in this thesis to provide proofs of running time for Algorithm 3. While experimental simulations like the one depicted in Figure 10 suggest that the work (iterations of $\mathcal{M}$) required to achieve α-compression is something polynomial, like $O(n^3)$, nothing has been proven. We currently are working to analyze Algorithm 3 from the perspective of distributed computing (as opposed to Markov chain analysis) to obtain bounds on its convergence time.

More generally, compression has only been examined in the geometric variant of the amoebot model. In order to be even more robust and applicable to physical situations, efforts could be undertaken to solve compression in three dimensional space, in which particles attempt to gather into a sphere-like configuration.

8 Acknowledgments

I want to thank the many involved in contributing to this work. Firstly, many thanks to my thesis committee, Dr. Andréa Richa and Dr. Hal Kierstead. Dr. Richa not only advised this honors thesis, but also guided my undergraduate research as a whole for the past two years. Dr. Kierstead played a critical role in culturing my interest in graph theory and helped to find a new name for this problem when it became apparent that "compaction" already carried a mathematical meaning. Thanks also to Zahra Derakhshandeh at Arizona State University for her aid in the early stages of the Local Compression algorithm.

I'm deeply grateful to our collaborators Dr. Christian Scheideler, Robert Gmyr, and Thim Strothmann at the Universität Paderborn, Germany for hosting me as a research assistant in Summer 2015. In collaboration with Robert and Thim, we developed the Hole Elimination algorithm, proved its correctness and convergence results, and performed its competitive analysis experiment against Hexagon Shape Formation. Additionally, Robert's help and expertise in the self-organizing particle systems simulator allowed me to develop the simulation for Leader Election.

Finally, I want to thank Dr. Dana Randall and Sarah Cannon at the Georgia Institute of Technology for their formidable efforts on the Markov Chain Algorithm for Compression. The ideas driving the improvements to the bounds for λ were also Sarah's, and her suggestions and writeups were central to code I developed to find the improved bounds shown in this thesis.

References

[1] Daisuke Baba, Tomoko Izumi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Linear time and space gathering of anonymous mobile agents in asynchronous trees. *Theoretical Computer Science*, 478:118–126, 2013.

[2] Evangelos Bampas, Jurek Czyzowicz, Leszek Gąsieniec, David Ilcinkas, and Arnaud Labourel. Almost optimal asynchronous rendezvous in infinite multidimensional grids. In *Distributed Computing: 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, pages 297–311, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] Lali Barriere, Paola Flocchini, Pierre Fraigniaud, and Nicola Santoro. Rendezvous and election of mobile agents: Impact of sense of direction. *Theory of Computing Systems*, 40(2):143–162, 2005.

[4] S. Camazine, K. P. Visscher, J. Finley, and S. R. Vetter. House-hunting by honey bee swarms: collective decisions and individual behaviors. *Insectes Sociaux*, 46(4):348–360.

[5] Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A markov chain algorithm for compression in self-organizing particle systems. *CoRR*, abs/1603.07991, 2016.

[6] Jérémie Chalopin, Shantanu Das, and Nicola Santoro. Rendezvous of mobile agents in unknown graphs with faulty links. In *Distributed Computing: 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007. Proceedings*, pages 108–122, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Arturo Chavoya and Yves Duthen. Using a genetic algorithm to evolve cellular automata for 2D/3D computational development. In *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*, pages 231–232, 2006.

[8] Jurek Czyzowicz, Adrian Kosowski, and Andrzej Pelc. Time versus space trade-offs for rendezvous in trees. *Distributed Computing*, 27(2):95–109, 2013.

[9] Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin-Parvédy. Fault-tolerant and self-stabilizing mobile robots gathering. In *Distributed Computing: 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006. Proceedings*, pages 46–60, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot - a new model for programmable matter. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 220–222, 2014.

[11] Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM' 15, Boston, MA, USA, September 21-22, 2015*, pages 21:1–21:2, 2015.

[12] Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida A. Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*, pages 117–132, 2015.

[13] Zahra Derakshandeh, Robert Gmyr, Alexandra Porter, and booktitle = Submitted manuscript to DNA22 year = 2016 André]a W. Richa and Christian Scheideler and Thim Strothmann, title = On the Runtime of Universal Coating for Programmable Matter.

[14] Andreas Deutsch and Sabine Dormann. *Cellular Automaton Modeling of Biological Pattern Formation: Characterization, Applications, and Analysis*. Modeling and Simulation in Science, Engineering and Technology. Birkhäuser Boston, 2007.

[15] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Multiple agents rendezvous in a ring in spite of a black hole. In *Principles of Distributed Systems: 7th International Conference, OPODIS 2003, La Martinique, French West Indies, December 10-13, 2003, Revised Selected Papers*, pages 34–46, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[16] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 1968.

[17] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(13):412 – 447, 2008.

[18] Simon Garnier, Jacques Gautrais, and Guy Theraulaz. The biological principles of swarm intelligence. *Swarm Intelligence*, 1(1):3–31, 2007.

[19] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[20] Raphael Jeanson, Colette Rivault, Jean-Louis Deneubourg, Stephane Blanco, Richard Fournier, Christian Jost, and Guy Theraulaz. Self-organized aggregation in cockroaches. *Animal Behaviour*, 69(1):169 – 180, 2005.

[21] Ralf Klasing, Adrian Kosowski, and Alfredo Navarra. Taking advantage of symmetries: Gathering of asynchronous oblivious robots on a ring. *Theoretical Computer Science*, 411:3235 – 3246, 2010.

[22] Nathan J. Mlot, Craig A. Tovey, and David L. Hu. Fire ants self-assemble into waterproof rafts to survive floods. *Proceedings of the National Academy of Sciences*, 108(19):7669–7673, 2011.

[23] Dana Randall. Slow mixing of glauber dynamics via topological obstructions. In *17th ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 870–879, 2006.

[24] Colette Rivault and Ann Cloarec. Cockroach aggregation: discrimination between strain odours in Blattella germanica. *Animal Behaviour*, 55(1):177 – 184, 1998.

[25] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[26] Thomas C. Schelling. *The strategy of conflict.* Oxford University Press, 1960.

[27] Masahiro Shibata, Shinji Kawai, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Partial gathering of mobile agents in asynchronous unidirectional rings. *Theoretical Computer Science*, 617:1–11, 2016.

[28] Alistair Sinclair. *Algorithms for Random Generation & Counting: a Markov Chain Approach.* Birkhäuser, Boston, 1993.

[29] D.J.T Sumpter. The principles of collective animal behaviour. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 361(1465):5–22, 2006.

[30] Erik Winfree, Furong Liu, Lisa A. Wenzler, and Nadrian C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, 1998.

[31] Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science, Berkeley, California, USA, January 9-12th, 2013*, pages 353–354, 2013.

A Code for Improving Bounds on λ

The following are the two C++ implementations for improving both bounds on λ. They are commented extensively for readability.

A.1 Lower Bound for Compression

```
/* SOPS:         Compression - finding a better bound for lambda > 5
*  FILENAME:     lambdalowerbound.cpp
*  AUTHOR:       Joshua J. Daymude
*  DESCRIPTION: In our paper submitted to PODC'16, we proved formally that for lambda
*               > 5, compression occurs and for 0 < lambda < sqrt(2) we do not get
*               compression. We are seeking to close the gap between sqrt(2) and 5 by
*               counting the possible walks around the perimeter of a structure. To do
*               so, we consider a k-step subset of a walk and examine the number of
*               valid/invalid walks that come out of it. Then, using a similar
*               methodology as in the paper, we compute the kth root of the number of
*               valid walks to get a better bound on lambda.
*/

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

using namespace std;

typedef struct {
    pair<int,int> pos;
    int edges[6];
} Node;

// edge values: -1 -> unknown, 0 -> open to the perimeter, 1 -> occupied by edge
const Node default_node = {make_pair(0,0), {-1,-1,-1,-1,-1,-1}};

vector<Node>::iterator nodePosSearch(vector<Node> * nodes, const pair<int,int> target) {
    for(vector<Node>::iterator it = nodes->begin(); it != nodes->end(); ++it) {
        if(it->pos.first == target.first && it->pos.second == target.second) {
            return it;
        }
    }
    return nodes->end(); // position pair not found
}

vector<pair<int,int>> getNeighborPositions(const pair<int,int> pos) {
    vector<pair<int,int>> neighborPos;
    neighborPos.push_back(make_pair(pos.first + 1, pos.second));
    neighborPos.push_back(make_pair(pos.first, pos.second + 1));
    neighborPos.push_back(make_pair(pos.first - 1, pos.second + 1));
    neighborPos.push_back(make_pair(pos.first - 1, pos.second));
    neighborPos.push_back(make_pair(pos.first, pos.second - 1));
    neighborPos.push_back(make_pair(pos.first + 1, pos.second - 1));

    return neighborPos;
}

bool markEdge(Node * node, const int edge, const int value) {
    // used to mark edges with values in {-1,0,1}
    // if it attempts to override a 0 with a 1 or a 1 with a 0, it detects a fault

    if(node->edges[edge] == -1 || node->edges[edge] == value) {
        node->edges[edge] = value;
        return true;
    }
    else {
        return false;
    }
}

bool testWalkSubset(const string s) {
    // create node storage and set up n0
    vector<Node> nodes;
    nodes.push_back(default_node);
    nodes.back().edges = {1,-1,-1,-1,-1,0};

    // create variable to track the last known position and expected position of
    // the next node
    pair<int,int> lastNodePos = make_pair(0,0);
```

```
    pair<int,int> nextNodePos = make_pair(1,0);

    for(size_t i = 0; i < s.length(); ++i) {
        // if the next position is new, make a new node; otherwise find the existing node
        Node * currentNode;
        vector<Node>::iterator it = nodePosSearch(&nodes, nextNodePos);
        if(it == nodes.end()) {
            // add new node with the expected position
            nodes.push_back(default_node);
            nodes.back().pos = nextNodePos;
            currentNode = &nodes.back();

            // if existing nodes are adjacent to this new node, update their edges accordingly
            vector<pair<int,int>> neighborPositions = getNeighborPositions(currentNode->pos);
            for(auto neighborPos = neighborPositions.begin(); neighborPos != neighborPositions.end
                (); ++neighborPos) {
                vector<Node>::iterator neighborIt = nodePosSearch(&nodes, *neighborPos);
                int neighborEdge;

                if(neighborIt != nodes.end()) {
                    // for case comments, let (x,y) be the neighbor and (x',y') be the new node
                    if(neighborPos->first < currentNode->pos.first) {
                        if(neighborPos->second == currentNode->pos.second) {
                            neighborEdge = 0; // (x,y) = (x'-1,y)
                        }
                        else {
                            neighborEdge = 1; // (x,y) = (x'-1,y'+1)
                        }
                    }
                    else if(neighborPos->first == currentNode->pos.first) {
                        if(neighborPos->second > currentNode->pos.second) {
                            neighborEdge = 2; // (x,y) = (x',y'+1)
                        }
                        else {
                            neighborEdge = 5; // (x,y) = (x',y'-1)
                        }
                    }
                    else { // neighborPos->first > currentNode->pos.first
                        if(neighborPos->second == currentNode->pos.second) {
                            neighborEdge = 3; // (x,y) = (x'+1,y')
                        }
                        else {
                            neighborEdge = 4; // (x,y) = (x'+1,y'-1)
                        }
                    }
                    if(!markEdge(&(*neighborIt), neighborEdge, 1) || !markEdge(currentNode, (
                        neighborEdge + 3) % 6, 1)) {
                        return false; // if marking fails it is because it has detected a fault
                    }
                }
            }
        }
        else {
            // use the node that already exists at the desired position
            currentNode = &(*it); // convert iterator to Node pointer
        }

        // use the last and next node positions to find the incoming edge on the current node
        int incomingEdge;
        if(currentNode->pos.first < lastNodePos.first) {
            if(currentNode->pos.second == lastNodePos.second) {
                incomingEdge = 0;
            }
            else {
                incomingEdge = 1;
            }
        }
        else if(currentNode->pos.first == lastNodePos.first) {
            if(currentNode->pos.second > lastNodePos.second) {
                incomingEdge = 2;
            }
            else {
                incomingEdge = 5;
            }
        }
        else { // currentNode->pos.first > lastNodePos.first
            if(currentNode->pos.second == lastNodePos.second) {
                incomingEdge = 3;
            }
            else {
                incomingEdge = 4;
            }
        }
        if(!markEdge(currentNode, incomingEdge, 1)) {
            return false;
        }
```

```
        // calculate the outgoing edge from the current node
        int outgoingEdge = (incomingEdge + (s[i] - '0')) % 6; // convert external angle value to
            int
        if(!markEdge(currentNode, outgoingEdge, 1)) {
            return false;
        }

        // mark unoccupied edges (those between incoming and outgoing, clockwise)
        for(int edge = (incomingEdge + 1) % 6; edge != outgoingEdge; edge = (edge + 1) % 6) {
            if(!markEdge(currentNode, edge, 0)) {
                return false;
            }
        }

        // using current node's position and the outgoing edge, find the next node's position
        lastNodePos = currentNode->pos;
        switch(outgoingEdge) {
            case 0:
                nextNodePos = make_pair(lastNodePos.first + 1, lastNodePos.second);
            break;
            case 1:
                nextNodePos = make_pair(lastNodePos.first + 1, lastNodePos.second - 1);
            break;
            case 2:
                nextNodePos = make_pair(lastNodePos.first, lastNodePos.second - 1);
            break;
            case 3:
                nextNodePos = make_pair(lastNodePos.first - 1, lastNodePos.second);
            break;
            case 4:
                nextNodePos = make_pair(lastNodePos.first - 1, lastNodePos.second + 1);
            break;
            case 5:
                nextNodePos = make_pair(lastNodePos.first, lastNodePos.second + 1);
            break;
            default:
                cout << "ERROR: outgoing edge " << outgoingEdge << " is not in [0,5]." << endl;
        }
    }

    // do calculations on final node, where there is no external angle
    vector<Node>::iterator finalNodeIt = nodePosSearch(&nodes, nextNodePos);
    // note: most of this code is similar to the above section where new nodes are added
    if(finalNodeIt == nodes.end()) {
        // add new node with the expected position
        nodes.push_back(default_node);
        nodes.back().pos = nextNodePos;
        Node * currentNode = &nodes.back();

        vector<pair<int,int>> neighborPositions = getNeighborPositions(currentNode->pos);
        for(auto neighborPos = neighborPositions.begin(); neighborPos != neighborPositions.end();
            ++neighborPos) {
            vector<Node>::iterator neighborIt = nodePosSearch(&nodes, *neighborPos);
            int neighborEdge;

            if(neighborIt != nodes.end()) {
                if(neighborPos->first < currentNode->pos.first) {
                    if(neighborPos->second == currentNode->pos.second) {
                        neighborEdge = 0;
                    }
                    else {
                        neighborEdge = 1;
                    }
                }
                else if(neighborPos->first == currentNode->pos.first) {
                    if(neighborPos->second > currentNode->pos.second) {
                        neighborEdge = 2;
                    }
                    else {
                        neighborEdge = 5;
                    }
                }
                else { // neighborPos->first > currentNode->pos.first
                    if(neighborPos->second == currentNode->pos.second) {
                        neighborEdge = 3;
                    }
                    else {
                        neighborEdge = 4;
                    }
                }
                if(!markEdge(&(*neighborIt), neighborEdge, 1) || !markEdge(currentNode, (
                    neighborEdge + 3) % 6, 1)) {
                    return false;
                }
            }
        }
    }
```

```
    return true;
}

string getKString(const int k, const int i) {
    // returns the ith string with k characters from [2,6]
    // order is 2...22, 2...23, ..., 2...26, 2...32

    int intString[k];
    for(int j = 0; j < k; ++j) {
        intString[j] = 0;
    }
    int reverseIndex = k - 1;
    int temp = i;

    while(temp / 5 > 0) {
        intString[reverseIndex] = temp % 5;
        temp = temp / 5;
        --reverseIndex;
    }
    intString[reverseIndex] = temp; // == (temp % 5)

    // convert the integer array to a string
    ostringstream converter;
    for(int j = 0; j < k; ++j) {
        converter << (intString[j] + 2);
    }
    return converter.str();
}

int main(int argc, char * argv[]) {
    // argument checking
    if(argc != 2) {
        cout << "ERROR, usage: " << argv[0] << " <k value>" << endl;
        return 1;
    }
    else if(atoi(argv[1]) < 1) {
        cout << "ERROR: the value for k must be positive." << endl;
        return 1;
    }

    const int k = atoi(argv[1]);
    int numValid = 0;
    int numInvalid = 0;

    // test generated strings of k external angles
    for(int i = 0; i < pow(5,k); ++i) {
        if(testWalkSubset(getKString(k,i))) {
            ++numValid;
        }
        else {
            ++numInvalid;
        }
    }

    // check the final values and calculate lambda
    double lambdabound = pow(numValid,1.0/k); // compute kth root
    cout << "By computing all " << k << "-step subsets of walks, " << numValid << "/" << pow(5,k)
         << " were valid." << endl;
    cout << "Bound: lambda > " << lambdabound << endl;

    return 0;
}
```

A.2 Upper Bound for Compression

```
/* SOPS:         Compression - finding a better bound for lambda > sqrt(2)
*  FILENAME:     lambdaupperbound.cpp
*  AUTHOR:       Joshua J. Daymude
*  DESCRIPTION: In our paper submitted to PODC'16, we proved formally that for lambda > 5,
*               compression occurs and for 0 < lambda < sqrt(2) we do not get compression.
*               We are seeking to close the gap between sqrt(2) and 5 by counting the
*               total number of triangle-free, connected configurations that n particles
*               could take. Thus, instead of looking at attaching particles one by one to
*               a triangle-free configuration, we can attach full size-n chunks, thereby
*               getting a higher bound than lambda > sqrt(2).
*/

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <tuple>
```

```
#include <utility>
#include <vector>

using namespace std;

// global variables;
int **tree;
int totalParticles;
vector<tuple<int,int,int,int>> branchStack;

// helper functions
const int posToXCoord(const int px) {
    return px;
}

const int posToYCoord(const int py) {
    return totalParticles - 1 - py;
}

const vector<pair<int,int>> getNeighborPos(const int px, const int py) {
    vector<pair<int,int>> neighborPos;

    if(px + 1 < totalParticles) {
        neighborPos.push_back(make_pair(px+1,py));
    }
    if(px + 1 < totalParticles && py - 1 > -1*totalParticles) {
        neighborPos.push_back(make_pair(px+1,py-1));
    }
    if(py - 1 > -1*totalParticles) {
        neighborPos.push_back(make_pair(px,py-1));
    }
    if(px - 1 >= 0) {
        neighborPos.push_back(make_pair(px-1,py));
    }
    if(px - 1 >= 0 && py + 1 < totalParticles) {
        neighborPos.push_back(make_pair(px-1,py+1));
    }
    if(py + 1 < totalParticles) {
        neighborPos.push_back(make_pair(px,py+1));
    }

    return neighborPos;
}

const bool isRightUpOfS(const int px, const int py) {
    // we suppose that particle S is at (0,0)
    if(px < 0) {
        return false;
    }
    else if(px == 0 && py < 0) {
        return false;
    }

    return true;
}

// using this function relies on a tricky insight: since we're building these trees particle
// by particle, if any potential position already has two occupied neighboring positions,
// then putting the current particle down results in making a triangle or cycle, depending
// on how those occupied positions are oriented.
const bool doesFormTriangleOrCycle(const int px, const int py) {
    vector<pair<int,int>> neighborPos = getNeighborPos(px,py);
    int neighborCount = 0;

    for(auto i = neighborPos.begin(); i != neighborPos.end(); ++i) {
        if(tree[posToYCoord(i->second)][posToXCoord(i->first)] != -1) {
            ++neighborCount;
        }
    }

    return (neighborCount > 1);
}

// recursive function
int numValidTrees(const int px, const int py, const int branchDir, const int particlesRemaining) {
    // test for the validity of position (px,py)
    if(isRightUpOfS(px,py) && !doesFormTriangleOrCycle(px,py)) {
        int remaining = particlesRemaining - 1;
        tree[posToYCoord(py)][posToXCoord(px)] = branchDir; // add particle to tree

        if(remaining == 0) {
            if(branchStack.empty()) { // we've found a valid tree using all the particles
                tree[posToYCoord(py)][posToXCoord(px)] = -1; // remove particle from tree
                return 1;
            }
            else { // we've finished one branch, but due to a case (4) there are others still to
                do
```

```
                tuple<int,int,int,int> stackCall = branchStack.back();
                branchStack.pop_back();
                int temp = numValidTrees(get<0>(stackCall), get<1>(stackCall), get<2>(stackCall),
                    get<3>(stackCall));
                tree[posToYCoord(py)][posToXCoord(px)] = -1; // remove particle from tree
                return temp;
            }
        }
        else {
            // depending on the branch direction, the neighbors involved in recursion are
                different
            pair<int,int> dirPlus1, dirSame, dirMinus1;
            switch(branchDir) {
                case 0: {
                    dirPlus1 = make_pair(px,py+1);
                    dirSame = make_pair(px+1,py);
                    dirMinus1 = make_pair(px+1,py-1);
                    break;
                }
                case 1: {
                    dirPlus1 = make_pair(px-1,py+1);
                    dirSame = make_pair(px,py+1);
                    dirMinus1 = make_pair(px+1,py);
                    break;
                }
                case 2: {
                    dirPlus1 = make_pair(px-1,py);
                    dirSame = make_pair(px-1,py+1);
                    dirMinus1 = make_pair(px,py+1);
                    break;
                }
                case 3: {
                    dirPlus1 = make_pair(px,py-1);
                    dirSame = make_pair(px-1,py);
                    dirMinus1 = make_pair(px-1,py+1);
                    break;
                }
                case 4: {
                    dirPlus1 = make_pair(px+1,py-1);
                    dirSame = make_pair(px,py-1);
                    dirMinus1 = make_pair(px-1,py);
                    break;
                }
                case 5: {
                    dirPlus1 = make_pair(px+1,py);
                    dirSame = make_pair(px+1,py-1);
                    dirMinus1 = make_pair(px,py-1);
                    break;
                }
            }

            // make the recursive calls for each of the four possible cases
            int treesSum = 0;
            treesSum += numValidTrees(dirPlus1.first, dirPlus1.second, (branchDir+1) % 6,
                remaining);
            treesSum += numValidTrees(dirSame.first, dirSame.second, branchDir, remaining);
            treesSum += numValidTrees(dirMinus1.first, dirMinus1.second, (branchDir+5) % 6,
                remaining);
            for(int i = 1; i < remaining; ++i) {
                branchStack.push_back(make_tuple(dirMinus1.first, dirMinus1.second, (branchDir+5)
                    % 6, remaining - i));
                treesSum += numValidTrees(dirPlus1.first, dirPlus1.second, (branchDir+1) % 6, i);
            }

            // remove current particle from tree, since we've explored all subtrees using it
            tree[posToYCoord(py)][posToXCoord(px)] = -1;
            return treesSum;
        }
    }
    else { // there are no more trees to be found by adding (px,py)
        // if a position ends up being invalid, we still need to pop that branch off
        if(!branchStack.empty()) {
            branchStack.pop_back();
        }
        return 0;
    }
}

// main function
int main(int argc, char * argv[]) {
    // argument checking
    if(argc != 2) {
        cout << "ERROR, usage: " << argv[0] << " <n value>" << endl;
        return 1;
    }
    else if(atoi(argv[1]) < 1) {
        cout << "ERROR: number of particles (n) must be positive." << endl;
```

```
        return 1;
    }

    totalParticles = atoi(argv[1]);
    const int n = totalParticles;

    // instantiate tree space
    // the tree holds integers 0--5 which map to branch directions 0--300
    // where 0 is up-right and 300 is down-right; -1 is unoccupied
    tree = new int *[2*n-1];
    for(int i = 0; i < 2*n-1; ++i) {
        tree[i] = new int[n];
    }

    // set all values initially to false (no particles)
    for(int i = 0; i < 2*n-1; ++i) {
        for(int j = 0; j < n; ++j) {
            tree[i][j] = -1;
        }
    }

    // set up the recursive call
    int totalTrees = numValidTrees(0,0,0,n);

    // print results
    double lambdabound = pow(2*totalTrees,1.0/(2*n));
    if(totalTrees == 1) {
        cout << "There is " << totalTrees << " connected, triangle-free configuration on " << n <<
            " particle." << endl;
    }
    else {
        cout << "There are " << totalTrees << " connected, triangle-free configurations on " << n
            << " particles." << endl;
    }
    cout << "Bound: lambda < " << lambdabound << endl;

    return 0;
}
```