Collaborating in Motion

Distributed and Stochastic Algorithms

for Emergent Behavior in Programmable Matter

by

Joshua J. Daymude

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2021 by the
Graduate Supervisory Committee:

Andréa W. Richa, Chair
Christian Scheideler
Dana Randall
Theodore Pavlic
Stephanie Gil

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

The world is filled with systems of entities that collaborate in motion, both natural and engineered. These cooperative distributed systems are capable of sophisticated emergent behavior arising from the comparatively simple interactions of their members. A model system for emergent collective behavior is programmable matter, a physical substance capable of autonomously changing its properties in response to user input or environmental stimuli. This dissertation studies distributed and stochastic algorithms that control the local behaviors of individual modules of programmable matter to induce complex collective behavior at the macroscale. It consists of four parts.

In the first, the canonical amoebot model of programmable matter is proposed. A key goal of this model is to bring algorithmic theory closer to the physical realities of programmable matter hardware, especially with respect to concurrency and energy distribution. Two protocols are presented that together extend sequential, energy-agnostic algorithms to the more realistic concurrent, energy-constrained setting without sacrificing correctness, assuming the original algorithms satisfy certain conventions.

In the second part, stateful distributed algorithms using amoebot memory and communication are presented for leader election, object coating, convex hull formation, and hexagon formation. The first three algorithms are proven to have linear runtimes when assuming a simplified sequential setting. The final algorithm for hexagon formation is instead proven to be correct under unfair asynchronous adversarial activation, the most general of all adversarial activation models.

In the third part, distributed algorithms are combined with ideas from statistical physics and Markov chain design to replace algorithm reliance on memory and communication with biased random decisions, gaining inherent self-stabilizing and fault-tolerant properties. Using this stochastic approach, algorithms for compression, shortcut bridging, and separation are designed and analyzed.

Finally, a two-pronged approach to “programming” physical ensembles is presented. This approach leverages the physics of local interactions to pair theoretical abstractions of self-organizing particle systems with experimental robot systems of active granular matter that intentionally lack digital computation and communication. By physically embodying the salient features of an algorithm in robot design, the algorithm’s theoretical analysis can predict the robot ensemble’s behavior. This approach is applied to phototaxing, aggregation, dispersion, and object transport.

DEDICATION

For my redeemer Jesus Christ, my dearest Annie, and our beloved community
who continuously show me more clearly than any algorithm or theorem
what beauty can come from unpolished, imperfect things
willing to work alongside one another in love.

ACKNOWLEDGMENTS

I am immensely fortunate to have the love, support, and friendship of many truly good people, without whom I could not have become this version of myself or completed this dissertation. Perhaps unfortunately for the organization and length of these acknowledgements, I have tried to live a holistic and integrated life that does not delineate cleanly between family, friends, mentors, colleagues, and faith communities. In the true spirit of academic computer science writing, I have chosen to include my full acknowledgements as Appendix A due to space constraints. For each of you appearing there and also for those who I could not exhaustively list, know that you are loved and have my deepest gratitude.

TABLE OF CONTENTS

Page

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTER

1 INTRODUCTION 1

1.1 Introduction & Motivation 1

1.2 Related Work 6

2 THE CANONICAL AMOEBOT MODEL 8

2.1 Relationship to Other Models of Programmable Matter 9

2.2 Model Description 11

2.2.1 Amoebot Anatomy 14

2.2.2 Amoebot Operations 15

2.2.3 Amoebot Actions, Algorithms, and Executions 24

2.2.4 Time Complexity 26

2.3 Rationale 27

2.4 Extensions 29

2.5 Asynchronous Hexagon Formation 30

2.5.1 The Hexagon Formation Problem 31

2.5.2 The HEXAGON-FORMATION Algorithm 31

2.5.3 Analysis 34

2.5.4 Discussion 43

3 CONCURRENCY CONTROL 45

3.1 Algorithm Conventions 46

3.2 The Concurrency Control Protocol 49

3.3 Analysis 53
3.4 Discussion and Open Problems 72
4 ENERGY DISTRIBUTION 75
4.1 Biological Inspiration 77
4.2 The Energy Distribution Problem 79
4.3 The ENERGY-SHARING Algorithm 80
4.3.1 Algorithm Description 81
4.3.2 Analysis 84
4.4 Simulation Results 97
4.5 Extensions 98
4.5.1 The FOREST-PRUNE-REPAIR Algorithm 101
4.5.2 Analysis 102
4.5.3 Algorithm Composition 106
4.5.4 Using Energy For Reproduction 109
5 LEADER ELECTION 112
5.1 The Leader Election Problem 115
5.2 The IMPROVED-LEADER-ELECTION Algorithm 116
5.2.1 Boundary Setup 117
5.2.2 Segment Setup 119
5.2.3 Identifier Setup 120
5.2.4 Identifier Comparison 121
5.2.5 Solitude Verification 124
5.2.6 Boundary Identification 127
5.3 Analysis 128

5.4 Extensions 140

5.4.1 Expanded Amoebots 140

5.4.2 Termination for All Amoebots 141

5.4.3 Almost Sure Leader Election 142

6 OBJECT COATING 145

6.1 The Object Coating Problem 146

6.2 The Universal-Coating Algorithm 147

6.2.1 Preliminaries and Setup 148

6.2.2 Complaint Coating 151

6.2.3 Node Election 152

6.2.4 General Layering 153

6.3 Runtime Analysis 154

6.4 Lower Bounds 170

6.5 Simulations 174

7 CONVEX HULL FORMATION 178

7.1 The Convex Hull Formation Problem 180

7.2 The Single-Amoebot Algorithm 182

7.3 A Binary Counter of Amoebots 187

7.3.1 Correctness Analysis 189

7.3.2 Runtime Analysis 192

7.4 The Convex-Hull-Formation Algorithm 198

7.4.1 Learning the Convex Hull 199

7.4.2 Adapting Binary-Counter for Convex-Hull-Formation 202

7.4.3 Forming the Convex Hull 204

7.4.4 Correctness Analysis 210
7.4.5 Runtime Analysis 223
7.5 Forming the (Weak) $\mathcal{O}_\Delta$-Hull 230
8 COMPRESSION 240
8.1 Preliminaries 241
8.1.1 A Stochastic Approach to Self-Organizing Particle Systems . 242
8.1.2 Terminology for the Stochastic Approach 243
8.1.3 Compression of Amoebot Systems 245
8.1.4 Background on Markov Chains 249
8.2 Algorithms for Compression 252
8.2.1 The Markov Chain $\mathcal{M}_C$ for Compression 253
8.2.2 The Local, Distributed Algorithm $\mathcal{A}_C$ for Compression 255
8.2.3 Obliviousness and Robustness of $\mathcal{M}_C$ and $\mathcal{A}_C$ 259
8.2.4 Invariants for $\mathcal{M}_C$ 261
8.2.5 Eventual Ergodicity of $\mathcal{M}_C$ 264
8.2.6 The Stationary Distribution π of $\mathcal{M}_C$ 275
8.2.7 Convergence Time of $\mathcal{M}_C$ 278
8.3 Achieving Compression 279
8.3.1 Perimeter and Self-Avoiding Walks 280
8.3.2 Proof of Compression 285
8.4 Achieving Expansion 287
8.4.1 A Non-trivial Lower Bound on the Partition Function 289
8.4.2 Proof of Expansion 291
8.4.3 An Improved Lower Bound on the Partition Function 293

CHAPTER Page

8.4.4 Proof of Expansion for a Larger Range of Bias 298

9 SHORTCUT BRIDGING 301

9.1 Preliminaries 301

9.1.1 The Shortcut Bridging Problem 302

9.1.2 Applying the Stochastic Approach 304

9.2 Algorithms for Shortcut Bridging 305

9.2.1 The Markov Chain $\mathcal{M}_B$ for Shortcut Bridging 306

9.2.2 The Local, Distributed Algorithm $\mathcal{A}_B$ for Shortcut Bridging 308

9.3 Analysis of the Markov Chain $\mathcal{M}_B$ 312

9.4 Dependence on the Gap Angle 316

9.4.1 Proofs for Small θ 320

9.4.2 Proofs for Large θ 326

9.5 Simulations 332

10 SEPARATION 335

10.1 Preliminaries 339

10.1.1 Terminology and Results for Homogeneous Amoebot Systems 339

10.1.2 Heterogeneous Systems and The Separation Problem 342

10.2 Algorithms for Separation 343

10.2.1 The Markov Chain $\mathcal{M}_S$ for Separation 343

10.2.2 The Stationary Distribution of Markov Chain $\mathcal{M}_S$ 345

10.2.3 Simulations 350

10.3 Technical Overview 351

10.4 Proof of Separation 353

10.4.1 Lattice Duality and Contours 355

CHAPTER Page

10.4.2 Bridging Systems 357

10.4.3 Information Theoretic Argument for Separation 361

10.5 Proof of Integration 369

11 APPLICATIONS TO SWARM ROBOTICS AND GRANULAR ACTIVE MATTER 378

11.1 Phototactic Supersmarticles 379

11.1.1 Supersmarticle Design and Experimental Results 381

11.1.2 A Distributed, Stochastic Algorithm for Phototaxing 384

11.2 Aggregation, Dispersion, and Collective Transport in BOBbots . . . 389

11.2.1 The Markov Chain $\mathcal{M}_A$ for Aggregation and Dispersion 392

11.2.2 BOBbots: A Model Active Cohesive Granular Matter System 398

11.2.3 Clustering Dynamics in BOBbots and Theory 401

11.2.4 Object Transport in the Aggregated Phase 407

11.2.5 Discussion 410

12 CONCLUSION 414

REFERENCES 418

APPENDIX

A FULL ACKNOWLEDGEMENTS 435

LIST OF TABLES

Table Page

1. Assumption Variants in the Canonical Amoebot Model 12

2. Amoebot Operations in the Canonical Amoebot Model..................... 15

3. Local Variables for Hexagon-Formation 32

4. Parameters for Energy-Sharing 81

5. Local Variables for Energy-Sharing 81

6. Comparison of Amoebot Algorithms for Leader Election 113

7. Local Variables for Universal-Coating 148

LIST OF FIGURES

Figure Page

1. The Canonical Amoebot Model 13
2. The Communication Operations 17
3. The Contract and Expand Operations 19
4. The Handover Operations 20
5. Lattice Movements 28
6. The Hexagon-Formation Algorithm 32
7. The Monotonicity Convention 48
8. Amoebot Energy Anatomy 79
9. Simulation of Energy-Sharing 99
10. Simulation of Energy-Sharing Without Communication 100
11. Chase Cycles in Forest-Prune-Repair 105
12. Simulation of Energy-Sharing Composed with Basic-Shape-Formation 108
13. Runtime Experiments for Algorithm Composition 110
14. Simulation of Energy-Sharing with Reproduction 111
15. Boundaries and Agents 118
16. Amoebot Neighborhoods and Their Boundaries 119
17. Identifier Comparison 123
18. Solitude Verification 125
19. Boundary Identification 128
20. The Complaint Coating Phase 151
21. The General Layering Phase 154
22. Object Coating Instance for Linear Lower Bound 171
23. Object Coating Instance for Linear Competitive Ratio 172

Figure Page

24. Optimal Algorithm for Object Coating 174

25. Simulations of the Universal-Coating Algorithm........ 176

26. Restricted-Orientation Convex Hulls 181

27. The Half-Planes of a Convex Hull 183

28. The Single-Amoebot Algorithm for Convex Hull Formation 184

29. Safe Handovers in the Learning Phase 202

30. The Hull Closing Subphase 206

31. Freeing a Trapped Amoebot 209

32. Movements of Filler Amoebots 210

33. The O-Hull-Formation Algorithm 231

34. The Execution of a Greedy Parallel Cycle Schedule 236

35. An Amoebot System Configuration and Its Boundaries 245

36. Simulation of Markov Chain $\mathcal{M}_C$ with $\lambda = 4$ 255

37. The Necessity of Jump Moves in $\mathcal{M}_C$ 265

38. The Line Invariants for Ergodicity 266

39. Local Moves for Line Setup 267

40. Local Moves for Line Formation 270

41. Local Moves for Merging Lines 271

42. Self-Avoiding Walks in the Hexagonal Lattice 281

43. Self-Avoiding Walks and Particle System Configurations 282

44. Simulation of Markov Chain $\mathcal{M}_C$ with $\lambda = 2$ 288

45. Three-Particle System Configurations 294

46. Enumerating Tree Configurations. 296

47. Example Initial Configurations for Shortcut Bridging 303

Figure Page

48. Choosing an Optimization Metric to Produce Ant-Like Bridges 304

49. Construction of V-Shaped Land Masses by Gap Angle 318

50. Initial System Configurations on V-Shaped Land Masses by Gap Angle 319

51. Supporting Figures for the Analysis of Small Gap Angles 322

52. Calculating the Gap Depth 327

53. Supporting Figures for the Analysis of Large Gap Angles 328

54. Simulation of Markov Chain $\mathcal{M}_B$ on a V-Shaped Land Mass 333

55. Simulations of Markov Chain $\mathcal{M}_B$ Varying Gap Angle 333

56. Simulation of Markov Chain $\mathcal{M}_B$ on an N-Shaped Land Mass 333

57. Bounding the Minimum Perimeter of a System Configuration.............. 340

58. Simulation of Markov Chain $\mathcal{M}_S$ with $\lambda = 4$ and $\gamma = 4$ 350

59. The Four Phases of Markov Chain $\mathcal{M}_S$ 351

60. Lattice Duality and Contours 355

61. Bridging Systems 358

62. Diamond Partition of the Triangular Lattice 371

63. Smarticles and Supersmarticle Collectives 381

64. The Smarticle Square Gait 382

65. Supersmarticle Trajectories 383

66. Modeling Phototaxing in the Amoebot Model 385

67. Phototaxing in a System of Three Amoebots 388

68. Simulation of Algorithm $\mathcal{A}_P$ for Phototaxing 390

69. The Markov Chain $\mathcal{M}_A$ for Aggregation 395

70. BOBbots And Their Collective Motion 399

71. Evolution of BOBbot Clusters 402

Figure Page

72. Algorithmic Interpretation of BOBbot Clustering 404

73. Perimeter Scaling of BOBbot Clusters 405

74. Object Transport Using Aggregation 408

75. Mean-Squared Displacement of Object Transport 410

Chapter 1

INTRODUCTION

1.1 Introduction & Motivation

Our world is filled with systems of entities that collaborate in motion. From biological cells to social insects to human societies and nanorobotics to Bluetooth devices to autonomous vehicular networks, this fundamental paradigm spans a diverse array of sizes, form factors, and functions. These cooperative distributed systems are capable of sophisticated and surprising *emergent behavior* arising from the comparatively simple interactions of their members. For nearly thirty years, researchers spanning biology, physics, materials science, robotics, and computer science have worked toward a model system for emergent collective behavior known as *programmable matter* [186]. The grand vision for programmable matter is to design fundamental "particles" that collectively form a substance capable of changing its physical properties in response to user input or stimuli from its environment. If realized, programmable matter could be deployed in numerous domain spaces to address a wide gamut of problems: in construction, smart materials could self-monitor structural integrity and self-repair minor damage; in environmental science, smart particles could locate and metabolize air and water pollutants at the micro-scale; and in healthcare, smart medicine could deliver and apply treatment where it is most needed.

This dissertation primarily focuses on advancing programmable matter's formal algorithmic underpinnings, investigating how simple, local interactions can induce emergent, macroscopic behavior. Abstracting away from any particular instantiation of programmable matter, we model these systems as collectives of computational "particles" that are strictly limited in terms of actuation, sensing, memory, communication, and computation. Although individually incapable of meaningful behavior, these particles can cooperate by executing *distributed algorithms* that, when designed carefully, can produce useful system-level behaviors. Formal algorithmic research on programmable matter is not unique to this dissertation — as we will discuss in the related work of Section 1.2 — but this work contributes significant advancements in the modeling, algorithm design and analysis, and robotic applications of programmable matter systems.

The first results in this dissertation address critical gaps between existing theoretical models of programmable matter and its practical implementations. In particular, we focus on *concurrency* and *energy*. Most models of programmable matter restrict or abstract from the inherent concurrency of simultaneously acting particles to make algorithm design and analysis simpler. Similarly, nearly all existing works only analyze programmable matter algorithms' time and (occasionally) space complexities — the usual computer science metrics of efficiency — while entirely ignoring energy harvesting, distribution, and usage. This dissertation research focuses on addressing these issues in the context of the *amoebot model* for programmable matter [52, 59]. We introduce the *canonical amoebot model* as a new generalization that formalizes all amoebot communication and cooperation in the tradition of message passing systems, enabling a fine-grained treatment of time, concurrency, and algorithm executions (Chapter 2). As a case study in concurrent algorithm design, we revisit the classical problem of *shape formation* in the canonical amoebot model and give an algorithm that correctly forms a regular hexagon even under unfair asynchronous adversarial activation, the most general of all adversarial models. We next focus on two black box enhancements of existing amoebot results: *concurrency control* (Chapter 3) and *energy distribution* (Chapter 4). Our concurrency control protocol transforms correct sequential algorithms that satisfy certain conventions into algorithms that remain correct in the concurrent setting. We then achieve an analogous goal for energy-agnostic algorithms, providing a worst-case asymptotically optimal protocol for energy distribution that can be composed with existing algorithms to ensure their correct behavior even under energy constraints. These advancements not only bring theoretical work closer to physical reality, but emphasize rich areas for further rigorous analysis.

We next develop and analyze *stateful distributed algorithms* under the amoebot model, i.e., those that utilize amoebot memory and communication. The first is a randomized algorithm for *leader election* that uses the geometry of the underlying lattice and amoebot communication to elect a unique leader amoebot for any static (non-moving), connected system with high probability (Chapter 5). The second is an algorithm for *object coating* that organizes an amoebot system in even layers around a given object. While this algorithm was the subject of earlier work [61], this dissertation research contributes its runtime analysis that leverages a careful *dominance argument* between sequential and parallel executions to prove that this algorithm has a worst-case asymptotically optimal runtime (Chapter 6). The third amoebot algorithm presented is for *convex hull formation* where an amoebot system must enclose an object using the minimum number of amoebots (Chapter 7). Our algorithm achieves a linear runtime and — to our knowledge — is the first distributed approach to computing convex hulls that uses entities that do not have any global information or coordinates and use strictly local sensing and constant-size memory. All three of these algorithms predate the canonical amoebot model and the work on concurrency described above, instead using the less realistic sequential setting where only one amoebot acts per time.

We continue the dissertation's algorithmic results with *stochastic distributed algorithms* under the amoebot model. The amoebot model was formulated with implementation by micro-scale robots in mind: it limits amoebots to constant-size memory, communication with immediate neighbors, and strictly local information (e.g., no orientation, sense of position, estimate of system size, etc.). To investigate what behaviors can be achieved by even less capable entities, our stochastic approach to programmable matter algorithm design uses design principles from statistical physics and Markov chain and Monte Carlo methods to shift algorithm reliance away from memory and communication to biased random decisions that, when designed carefully, cause the system to converge to desirable configurations. We present and analyze three algorithms in this stochastic paradigm: one for *compression*, where the system must gather as compactly as possible while remaining simply connected (Chapter 8); another for *shortcut bridging*, where the system forms a connected bridge over a gap that optimizes a structural tradeoff inspired by the bridging behavior of army ants (Chapter 9); and finally one for *separation*, where a system of heterogeneous amoebots must compress not only as a collective but also in monochromatic clusters (Chapter 10). Compared to the stateful algorithms of Chapters 5–7, algorithms developed using this stochastic approach require only a few bits of memory per amoebot and have inherent self-stabilizing and fault-tolerant properties. This inherent robustness makes these stochastic algorithms more readily adapted to experimental systems of swarm robots with minimal capabilities, as we demonstrate in the next chapter.

The final technical chapter of this dissertation focus on the implementation of our algorithms by swarm robotic systems (Chapter 11). In particular, we demonstrate how our stochastic algorithms enable the reinterpretation of digital, logic-based programming as physical, analog features of robot design. We preserve only the most salient features of the algorithms and do not require that the robots adhere all that closely to the model's assumptions; nevertheless, we can establish tight relationships between the swarm's behaviors and those predicted by the theory. Inspired by the group dynamics of three-link robots called *smarticles* ("smart active particles"), we investigate a variant of the compression algorithm where amoebots on one side of the system are activated more often than the rest. We find that, just like the *supersmarticle* collectives, the amoebot systems exhibit noisy, *directed locomotion* towards the less active side (Section 11.1). We then establish a tight feedback loop between the theoretical predictions of separation and the robot ensemble behaviors of *aggregation*, *dispersion*, and *collective transport* in swarms of analog robots known as *BOBbots* (Section 11.2). We obtain a tight mapping between the bias parameter in our algorithm and the force of attraction between BOBbots that accurately predicts when the BOBbot collectives will aggregate or disperse.

Altogether, this dissertation spans the full spectrum of modeling, algorithmic theory, and swarm robotic applications for programmable matter. With advancements towards designing concurrent, energy-constrained algorithms that are compatible with minimally capable robots — some of which are entirely analog and dependent on the physics of local interactions — this research brings algorithmic theory closer to realistic systems. We conclude in Chapter 12 with several open problems and exciting directions for future work.

1.2 Related Work

In 1991, Toffoli and Margolus [186] defined *programmable matter* as a physical computing medium composed of simple, homogeneous modules that can be (*i*) assembled into "lumps" of arbitrary size, (*ii*) dynamically reconfigured into any regular structure, (*iii*) interactively controlled by user input or environmental stimuli, and (*iv*) accessed in real time for observation, analysis, or modification. Over the last three decades, many organic, synthetic, and theoretical systems have been proposed that meet these criteria to varying degrees. In this related work, we give a brief survey of some of these systems; each technical section in Chapters 2–11 will provide more thorough and specific reviews of relevant literature.

A useful distinguishing feature of a programmable matter system is the degree to which its members self-determine and enact their local behaviors, ranging from *passive* to *active*. Members of fully passive systems depend entirely on the environment and their physical structure to determine their interactions. Prominent examples include DNA self-assembly, molecular computing, chemical reaction networks, and tile self-assembly models [37, 74, 157]. Wireless sensor networks and the corresponding population protocols model [10] are also passive with respect to their movement, but engage in active communication and computation during interactions. Slime molds also fall in the in-between of passive and active systems, as their decisions and movements are self-enacted but their behavior is largely defined by the environment [26, 169]. Swarm robot systems whose ensemble behaviors rely on passive physical interactions as opposed to active digital sensing and computation, such as the supersmarticles [43, 177, 178] and BOBbots [128] considered in this dissertation and other recent examples [124, 129, 195], also belong in this in-between.

Our algorithmic work primarily focuses on fully active systems, where individual modules self-enact interactions and movements to cooperatively achieve some task. Most systems in swarm robotics (or at least, those utilizing microcontrollers) are active [20, 27, 73, 101, 176]. Of particular relevance is the subfield of self-reconfigurable modular robotics [197] that studies "robots made of robots." Each individual modular robot is its own active entity, and must cooperate with the other modules to achieve system-level tasks. While many modular robot systems use robots that have significantly more powerful sensing, communication, and computational capabilities than would be available at small scales (see, e.g., [77, 81, 93, 175]), a notable exception is the Claytronics project [160] that takes seriously the limitations of micro-scale robotics. Our amoebot model for programmable matter [52, 59] addresses this minimally capable setting by ensuring each module is strictly limited and capable of only local interactions and movements. The closely related model of autonomous mobile robots [87] — which we discuss at length in Section 2.1 — also examines this restricted setting. Other theoretical models of programmable matter include the nubot model [194] and metamorphic robots [42, 189], but these models each include non-local capabilities that prohibit a direct translation of results to our setting.

Chapter 2

THE CANONICAL AMOEBOT MODEL

The *amoebot model* is an abstract computational model of programmable matter intended to enable rigorous algorithmic analysis of collective systems at the micro- and nano-scales. Originally proposed as "amoeba-inspired self-organizing particle systems" [71], the model was polished and formally announced as the *amoebot model* in 2014 [59]. From 2015–2020, the amoebot model was used to study both fundamental problems — such as leader election [21, 54, 60, 66, 78, 92] and shape formation [32, 58, 62, 66] — as well as more complex behaviors including object coating [55, 61], convex hull formation [53], bridging [8], spatial sorting [31], and fault tolerance [51, 67]. With this growing body of amoebot model literature, it is evident that the model has evolved — and, to some extent, fractured — during its lifetime as assumptions were updated to support individual results, capture more realistic settings, or better align with other models of programmable matter. This makes it difficult to conduct any systematic comparison between results under the amoebot model (see, e.g., the overlapping but distinct features used for comparison of leader election algorithms in [21] and [78]), let alone between amoebot model results and those of related models (e.g., those from the established *autonomous mobile robots* literature [87]). To address the ways in which the amoebot model has outgrown its original rigid formulation, we propose the *canonical amoebot model* [50] that includes a standardized, formal hierarchy of assumptions for its features to better facilitate comparison of its results. Moreover, such standardization will more gracefully support future model generalizations by distinguishing between core features and assumption variants.

A key area of improvement addressed by the canonical amoebot model is *concurrency*. The original model treats concurrency at a high level, implicitly assuming an isolation property that prohibits concurrent amoebot actions from interfering with each other. Furthermore, amoebots are usually assumed to be *reliable*; i.e., they cannot crash or exhibit Byzantine behavior. Under these simplifying assumptions, most existing algorithms — including the majority of those in this dissertation — are analyzed for correctness and runtime as if they are executed *sequentially*, with at most one amoebot acting at a time. Notable exceptions include the recent work of Di Luna et al. [66, 67] that adopt ideas from the "look-compute-move" paradigm used in autonomous mobile robots to bring the amoebot model closer to a realistic, concurrent setting. Our canonical amoebot model furthers these efforts by formalizing all communication and cooperation between amoebots as message passing while also addressing the complexity of potential conflicts caused by amoebot movements. This careful formalization allows us to use standard adversarial activation models from the distributed computing literature to describe concurrency [5].

2.1 Relationship to Other Models of Programmable Matter

There are many theoretical models of programmable matter in the literature, ranging from the non-spatial *population protocols* [10] and *network constructors* [145] to the tile-based models of *DNA computing* and *molecular self-assembly* [37, 157, 194]. Most closely related to the amoebot model is the well-established literature on *autonomous mobile robots*, and in particular those using discrete, graph-based models of space (see Chapter 1 of [87] for a recent overview). Both models assume anonymous individuals that can actively move, lacking a global coordinate system or common orientation, and having strictly limited computational and sensing capabilities. In addition, stronger capabilities assumed by the amoebot model also appear in more recent variants of mobile robots, such as persistent memory in the $\mathcal{F}$*-state* model [14, 88] and limited communication capabilities in *luminous robots* [47, 48, 68].

There are also key differences between the amoebot model and the standard assumptions for mobile robots, particularly around their treatment of physical space, the structure of individuals' actions, and concurrency. First, while the discrete-space mobile robots literature abstractly envisions robots as agents occupying nodes of a graph — allowing multiple robots to occupy the same node — the amoebot model assumes *physical exclusion* that ensures each node is occupied by at most one amoebot at a time, inspired by the real constraints of self-organizing micro-robots and colloidal state machines [106, 124, 132, 195, 196]. Physical exclusion introduces conflicts of movement (e.g., two amoebots concurrently moving into the same space) that must be handled carefully in algorithm design.

Second, mobile robots are assumed to operate in *look-compute-move* cycles, where they take an instantaneous snapshot of their surroundings (look), perform internal computation based on the snapshot (compute), and finally move to a neighboring node determined in the compute stage (move). While it is reasonable to assume robots may instantaneously snapshot their surroundings due to all information being visible, the amoebot model — and especially the canonical version presented in this work — treats all inter-amoebot communication as asynchronous message passing, making snapshots nontrivial. Moreover, amoebots have *read and write* operations allowing them to access or update variables stored in the persistent memories of their neighbors that do not fit cleanly within the look-compute-move paradigm.

Finally, the mobile robots literature has a well-established and carefully studied hierarchy of *adversarial schedulers* capturing assumptions on concurrency that the amoebot model has historically lacked. In fact, other than notable recent works that adapt look-compute-move cycles and a semi-synchronous scheduler from mobile robots for the amoebot model [66, 67], most amoebot literature assumes only sequential activations. A key contribution of our canonical amoebot model presented in this chapter is a hierarchy of concurrency and fairness assumptions similar in spirit to that of mobile robots, though our underlying message passing design and lack of explicit action structure require different formalizations.

2.2 Model Description

We introduce the *canonical amoebot model* as an update to the model's original formulation [52, 59]. This update has two main goals. First, we model all amoebot actions and operations using message passing, leveraging this finer level of granularity for a formal treatment of concurrency. Second, we clearly delineate which assumptions are fixed features of the model and which have stronger and weaker variants, providing unifying terminology for future amoebot model research. Unless variants are explicitly listed, the following description of the canonical amoebot model details its core, fixed assumptions. The variants are summarized in Table 1; we anticipate that this list will grow as future research develops new adaptations and generalizations of the model.

In the canonical amoebot model, programmable matter consists of individual, homogeneous computational elements called *amoebots*. Any structure that an amoebot system can form is represented as a subgraph of an infinite, undirected graph $G = (V, E)$ where V represents all relative positions an amoebot can occupy and E represents all atomic movements an amoebot can make. Each node in V can be occupied by at most

Assumption	**Variants**
Space	**General.* G is any infinite, undirected graph. **Geometric.* $G = G_\Delta$, the triangular lattice.
Orientation	**Assorted.* Assorted direction and chirality. **Common Chirality.* Assorted direction but common chirality. *Common Direction.* Common direction but assorted chirality. *Common.* Common direction and chirality.
Memory	*Oblivious.* No persistent memory. **Constant-Size.* Memory size is $\mathcal{O}(1)$. *Finite.* Memory size is $\mathcal{O}(f(n))$, i.e., some function of the system size. *Unbounded.* Memory size is unbounded.
Concurrency	*Asynchronous.* Arbitrary sets of amoebots can be simultaneously active. **Synchronous.* Arbitrary sets of amoebots can be simultaneously active, but all active amoebots must return to idle before the next arbitrary set is activated. *k-Isolated.* No amoebots within distance k are simultaneously active. **Sequential.* At most one amoebot is active per time.
Fairness	*Unfair.* Some enabled amoebot is eventually activated. * *Weakly Fair.* Every continuously enabled amoebot is eventually activated. *Strongly Fair.* Every amoebot that is enabled infinitely often is activated infinitely often.

Table 1. Assumption Variants in the Canonical Amoebot Model. Summary of assumption variants, each organized from least to most restrictive. Variants marked with $*$ have been considered in existing literature.

one amoebot at a time. There are many potential variants with respect to space; the most common is the *geometric* variant that assumes $G = G_\Delta$, the triangular lattice (Figure 1a).

An amoebot has two *shapes*: CONTRACTED, meaning it occupies a single node in V, or EXPANDED, meaning it occupies a pair of adjacent nodes in V (Figure 1b). For a contracted amoebot, the unique node it occupies is considered to be its *head*; for an expanded amoebot, the node it has most recently come to occupy (due to movements) is considered its head and the other is its *tail*. Each amoebot keeps a collection of ports — one for each edge incident to the node(s) it occupies — that are labeled consecutively according to its own local *orientation*. An amoebot's orientation depends on its *direction* — i.e., which axis direction it perceives as "north" — and its *chirality*, or sense of clockwise and counter-clockwise rotation. Different variants

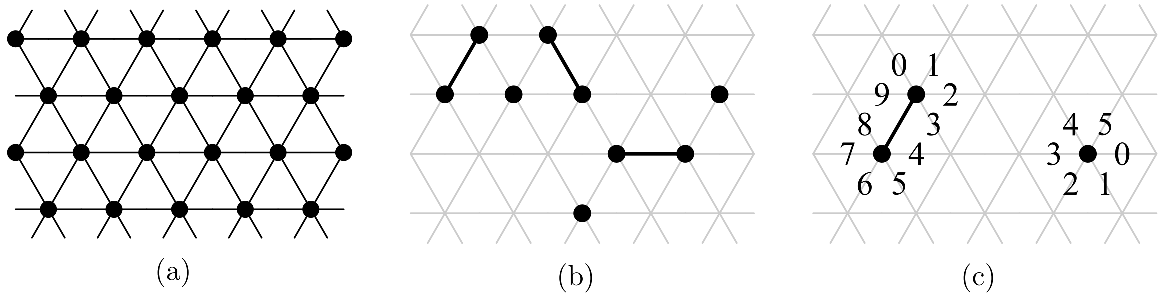


Figure 1. The Canonical Amoebot Model. (a) A section of the triangular lattice G_Δ used in the geometric variant; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted amoebots; G_Δ is shown in gray, and amoebots are shown as black circles. Amoebots with a black line between their nodes are expanded. (c) Two amoebots that agree on their chirality but not on their direction, using different offsets for their clockwise-increasing port labels.

may assume that amoebots share a common orientation with respect to both, one, or neither of their directions and chiralities (see Table 1); Figure 1c gives an example of the *common chirality* variant where amoebots share a sense of clockwise rotation but have different directions.

Two amoebots occupying adjacent nodes are said to be *neighbors*. Although each amoebot is *anonymous*, lacking a unique identifier, we assume an amoebot can locally identify its neighbors using their port labels. In particular, we assume that amoebots A and B connected via ports p_A and p_B each know one another's labels for p_A and p_B, whether or not they agree on chirality, and — in the case A or B are expanded — in which local direction they are expanded. This is sufficient for an amoebot to reconstruct which adjacent nodes are occupied by the same neighbor, but is not so strong so as to collapse the hierarchy of orientation assumptions. More details on an amoebot's anatomy are given in Section 2.2.1.

An amoebot's functionality is partitioned between a higher-level *application layer* and a lower-level *system layer*. Algorithms controlling an amoebot's behavior are designed from the perspective of the application layer. The system layer is responsible for an amoebot's core functions and exposes a limited programming interface of *operations* to the application layer that can be used in amoebot algorithms. The operations are defined in Section 2.2.2 and their organization into algorithms is described in Section 2.2.3.

2.2.1 Amoebot Anatomy

Each amoebot has persistent memory whose size is a model variant; the standard assumption is *constant-size* memories. An amoebot's memory consists of two parts: a *public memory* that is read-writeable by the system layer but only accessible to the application layer via communication operations (discussed in the next section), and a *private memory* that is inaccessible to the system layer but read-writeable by the application layer. The public memory of an amoebot A contains (i) the shape of A, denoted $A.\text{shape} \in \{\textsc{contracted}, \textsc{expanded}\}$, and (ii) publicly accessible copies of the variables used in the distributed algorithm being run by the application layer. An amoebot's private memory contains private copies of the distributed algorithm's variables that the application layer can modify as needed.

Neighboring amoebots (i.e., those occupying adjacent nodes) form *connections* via their ports facing each other. An amoebot's system layer receives instantaneous feedback whenever a new neighbor connects to it or an existing neighbor disconnects from it as these are physical, local interactions. Communication between connected neighbors is achieved via *message passing*. To facilitate message passing communication, each of an amoebot's ports has a FIFO *outgoing message buffer* managed by the system layer that can store up to a fixed (constant) number of messages waiting to be sent to the neighbor incident to the corresponding port. If two neighbors disconnect

Operation	**Return Value on Success**
Connected(p)	TRUE iff a neighboring amoebot is connected via port p.
Read(p, x)	The value of x in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise.
Write(p, x, x_{val})	Confirmation that the value of x was updated to x_{val} in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise.
Contract(v)	Confirmation of the contraction out of node $v \in \{\text{HEAD}, \text{TAIL}\}$.
Expand(p)	Confirmation of the expansion into the node incident to port p.
Pull(p)	Confirmation of the pull handover with the neighbor incident to port p.
Push(p)	Confirmation of the push handover with the neighbor incident to port p.
Lock()	Local identifiers of the amoebots that were successfully locked.
Unlock($\mathcal{L}$)	Confirmation that the amoebots of $\mathcal{L}$ were unlocked.

Table 2. Amoebot Operations in the Canonical Amoebot Model. Summary of operations exposed by an amoebot's system layer to its application layer.

due to some movement, their system layers immediately flush the corresponding message buffers of any pending messages. Otherwise, we assume that any pending message is sent to the connected neighbor in FIFO order in finite time.

2.2.2 Amoebot Operations

Operations provide the application layer with a programming interface for controlling the amoebot's behavior; the application layer calls operations and the system layer executes them. We assume the execution of an operation is *blocking* for the application layer; that is, the application layer can only execute one operation at a time. Each operation is designed to (*i*) send at most a constant number of messages per neighbor at a time and (*ii*) terminate — either successfully or in failure — in finite time. Combined with the blocking assumption, these design principles prohibit message buffer overflow, among other desirable properties. The communication, movement, and concurrency control operations are summarized in Table 2 and are formally defined in the following sections.

Communication Operations (Algorithm 1). An amoebot can check for the presence of a neighbor using the CONNECTED operation and exchange information with its neighbors using the READ and WRITE operations. When the application layer calls CONNECTED(p), the system layer simply returns TRUE if there is a neighbor connected via port p and FALSE otherwise. The application layer can call READ(p, x) to issue a request to read the value of a variable x in the public memory of the neighbor connected via port p. Analogously, the application layer can call WRITE(p, x, x_{val}) to issue a request to update the value of a variable x in the public memory of the neighbor connected via port p to a new value x_{val}. READ and WRITE can also be called with $p = \perp$ to access an amoebot's own public memory instead of a neighbor's.

Algorithm 1 Communication Operations for Amoebot A

```
function Connected($p$)
    if there is a neighbor connected via port $p$ then return TRUE.
    else return FALSE.
function Read($p, x$)
    On being called:
        if $p = \perp$ then return the value of $x$ in the public memory of $A$; success.
        else if Connected($p$) then send read_request($x$) via port $p$.
        else throw disconnect-failure.
    On receiving read_request($x$) via port $p'$:
        Let $x_{val}$ be the value of $x$ in the public memory of $A$.
        Send read_ack($x, x_{val}$) via port $p'$.
    On receiving read_ack($x, x_{val}$) via port $p$:
        return $x_{val}$; success.
    On disconnection via port $p$:
        throw disconnect-failure.
function Write($p, x, x_{val}$)
    On being called:
        if $p = \perp$ then update the value of $x$ in the public memory of $A$ to $x_{val}$; return success.
        else if Connected($p$) then send write_request($x, x_{val}$) via port $p$.
        else throw disconnect-failure.
    On write_request($x, x_{val}$) being sent:
        return success.
    On disconnection via port $p$:
        throw disconnect-failure.
    On receiving write_request($x, x_{val}$) via port $p'$:
        Update the value of $x$ in the public memory of $A$ to $x_{val}$.
```

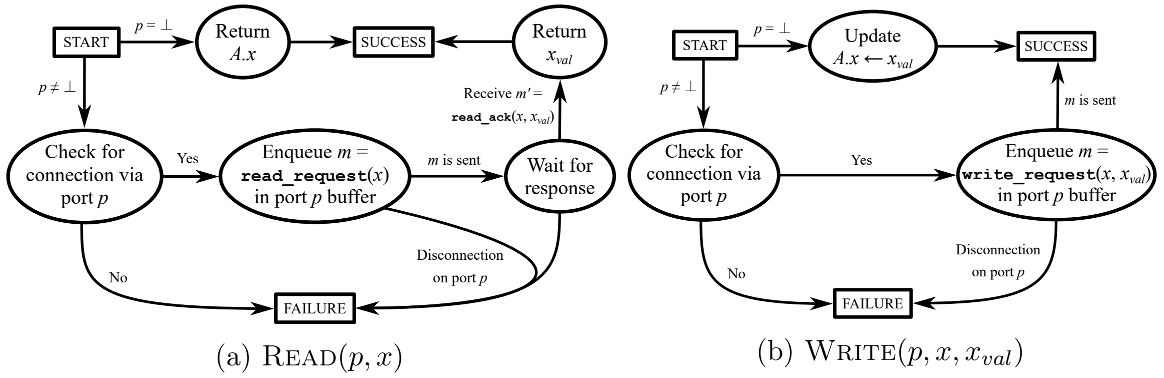

START
$p = \perp$
Return $A.x$
SUCCESS
Return x_{val}
$p \neq \perp$
Receive $m' =$ read_ack(x, x_{val})
Check for connection via port p
Yes
Enqueue $m =$ read_request(x) in port p buffer
m is sent
Wait for response
Disconnection on port p
No
FAILURE
(a) READ(p, x)
START
$p = \perp$
Update $A.x \leftarrow x_{val}$
SUCCESS
$p \neq \perp$
m is sent
Check for connection via port p
Yes
Enqueue $m =$ write_request(x, x_{val}) in port p buffer
Disconnection on port p
No
FAILURE
(b) WRITE(p, x, x_{val})

Figure 2. The Communication Operations. Execution flows of the READ and WRITE operations for the calling amoebot A.

Suppose that the application layer of an amoebot A calls READ(p, x), illustrated in Figure 2a. If $p = \perp$, the system layer simply returns the value of x in the public memory of A to the application layer and this READ succeeds. Otherwise, the system layer checks if there is a neighbor connected via port p: this READ fails if there is no such neighbor, but if there is, the system layer enqueues $m =$ `read_request`(x) in the message buffer on p. Let B be the neighbor connected to A via port p. Eventually, m is sent in FIFO order and the system layer of B receives it, prompting it to access variable x with value x_{val} in its public memory and enqueue $m' =$ `read_ack`(x, x_{val}) in the message buffer on its port facing A. Message m' is eventually sent in FIFO order by B and received by the system layer of A, prompting it to unpack x_{val} and return it to the application layer, successfully completing this READ. If A and B are disconnected (i.e., due to a movement) any time after A enqueues message m but before A receives message m', this READ fails.

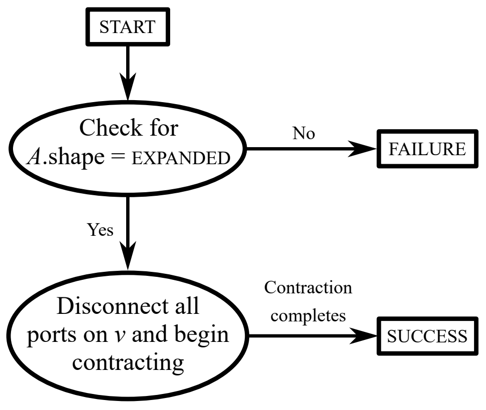
A WRITE(p, x, x_{val}) operation is executed analogously, though it does not need to wait for an acknowledgement after its write request is sent (see Figure 2b).

Movement Operations (Algorithms 2 and 3). The application layer can direct the system layer to initiate movements using the four movement operations CONTRACT, EXPAND, PULL, and PUSH. An expanded amoebot can CONTRACT into either node it occupies; a contracted amoebot occupying a single node can EXPAND into an unoccupied adjacent node. Neighboring amoebots can coordinate their movements in a *handover*, which can occur in one of two ways. A contracted amoebot A can PUSH an expanded neighbor B by expanding into a node occupied by B, forcing it to contract. Alternatively, an expanded amoebot B can PULL a contracted neighbor A by contracting, forcing A to expand into the neighbor it is vacating. We give the details of each of these movement operations below.

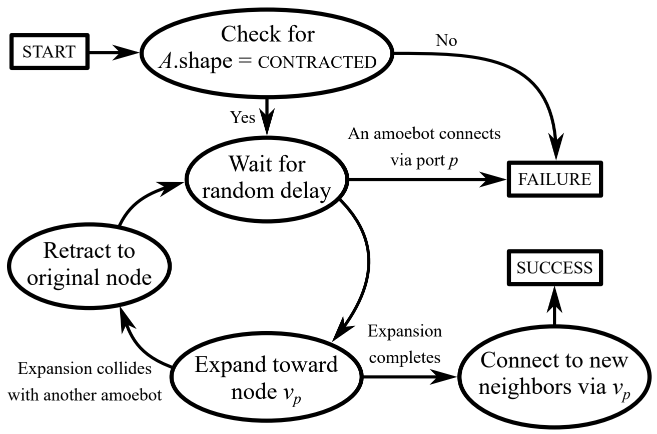
Algorithm 2 Movement Operations for Amoebot A: CONTRACT and EXPAND

```
 1: function CONTRACT($v$)
 2:     On being called:
 3:         if $A$.shape $\neq$ EXPANDED then throw shape-failure.
 4:         else if $A$ is involved in a handover then throw handover-failure.
 5:         else release all connections via ports on $v$ and begin contracting out of $v$.
 6:     On completing the contraction:
 7:         Update $A$.shape $\leftarrow$ CONTRACTED; return success.
 1: function EXPAND($p$)
 2:     Let $v_p$ denote the node that port $p$ faces.
 3:     On being called:
 4:         if $A$.shape $\neq$ CONTRACTED then throw shape-failure.
 5:         else if $A$ is involved in a handover then throw handover-failure.
 6:         else wait for a delay of 0.
 7:     After waiting for a delay:
 8:         if $\neg$CONNECTED($p$) then begin expanding into $v_p$.
 9:         else throw occupied-failure.
10:     On collision with another amoebot:
11:         Retract back out of $v_p$ and wait for a delay chosen u.a.r. from a sufficiently large interval.
12:     On connection via port $p$:
13:         throw occupied-failure.
14:     On completing the expansion:
15:         Establish connections with any new neighbors adjacent to $v_p$.
16:         Update $A$.shape $\leftarrow$ EXPANDED; return success.
```

Suppose that the application layer of an amoebot A calls CONTRACT(v), where $v \in \{\text{HEAD}, \text{TAIL}\}$ (see Figure 3a). The system layer of A first determines if this



(a) CONTRACT(v)



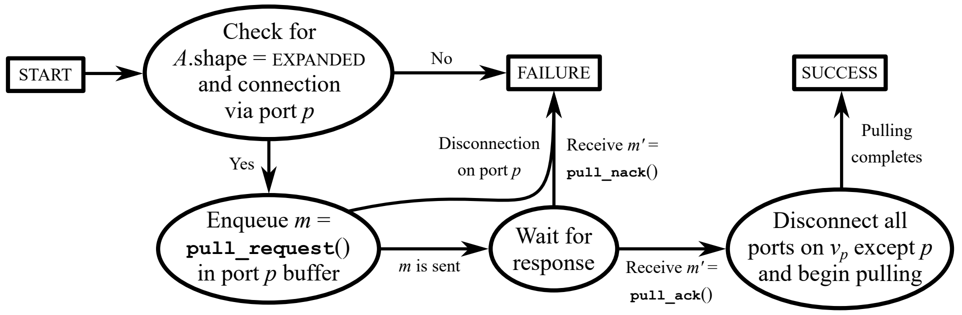
(b) EXPAND(p)

Figure 3. The Contract and Expand Operations. Execution flows of the CONTRACT and EXPAND operations for the calling amoebot A.

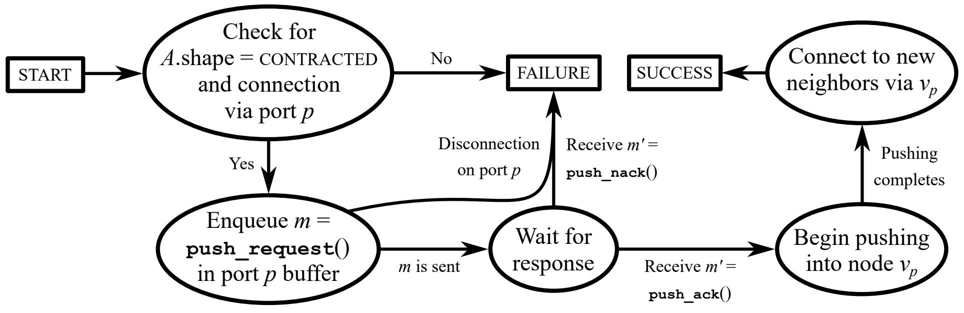
contraction is valid: if A.shape $\neq$ EXPANDED or A is currently involved in a handover, this CONTRACT fails. Otherwise, the system layer releases all connections to neighboring amoebots via ports on node v and begins contracting out of node v. Once the contraction completes, the system layer updates A.shape $\leftarrow$ CONTRACTED, successfully completing this CONTRACT.

Next suppose an amoebot A calls EXPAND(p) for one of its ports p (see Figure 3b); let v_p denote the node A is expanding into. If A.shape $\neq$ CONTRACTED, A is already involved in a handover, or v_p is already occupied by another amoebot, this EXPAND fails. Otherwise, A begins its expansion into node v_p. Once this expansion completes, the system layer establishes connections with all new neighbors adjacent to v_p, updates A.shape $\leftarrow$ EXPANDED, successfully completing this EXPAND.

However, A may *collide* with other amoebots while expanding into v_p. We assume that the system layer can detect when a collision has occurred. When A detects a collision, it retracts into its original node and then retries its expansion after a random time delay. If the delay is chosen uniformly at random from a sufficiently large time



(a) Pull(p)



(b) Push(p)

Figure 4. The Handover Operations. Execution flows of the Pull and Push operations for the calling amoebot A.

interval, then exactly one competing amoebot will successfully expand into v_p within $\mathcal{O}(\log n)$ expansion attempts, with high probability.[1] In [50], we show that for the geometric variant of the model on the triangular lattice, a time interval of $[5T, 10T]$ — where T is an upper bound on the time required for an amoebot to complete an expansion or retraction — is sufficient for this w.h.p. result. We omit the proof here as the analysis is a straightforward application of randomized backoff mechanisms for contention resolution in wireless networks [28, 33].

Finally, suppose an amoebot A calls Pull(p) for one of its ports p (see Figure 4a); let v_p denote the node A intends to vacate in this pull handover. If

[1]An event occurs *with high probability (w.h.p.)* if it occurs with probability at least $1 - 1/n^c$, where n is the number of amoebots in the system and $c > 0$ is a constant.

Algorithm 3 Movement Operations for Amoebot A: Pull and Push

```
For convenience, let $v_p$ denote the node that port $p$ faces.
function Pull($p$)
    On being called:
        if $A$.shape $\neq$ expanded then throw shape-failure.
        else if $A$ is involved in a handover then throw handover-failure.
        else if $\neg$Connected($p$) then throw disconnect-failure.
        else send pull_request() via port $p$.
    On receiving pull_request() via port $p'$:
        if $A$.shape $=$ contracted and $A$ is not involved in a move then set $m' \leftarrow$ pull_ack().
        else set $m' \leftarrow$ pull_nack().
        Send $m'$ via port $p'$.
    On sending pull_ack():
        Begin expanding into $v_p$.
    On completing the expansion into $v_p$:
        Establish connections with any new neighbors adjacent to $v_p$.
        Update $A$.shape $\leftarrow$ expanded.
    On receiving pull_ack() via port $p$:
        Release all connections via ports on $v_p$ except $p$ and begin contracting out of $v_p$.
    On receiving pull_nack() via port $p$ or on a disconnection via port $p$:
        throw nack-failure.
    On completing the contraction out of $v_p$:
        Update $A$.shape $\leftarrow$ contracted; return success.
function Push($p$)
    On being called:
        if $A$.shape $\neq$ contracted then throw shape-failure.
        else if $A$ is involved in a handover then throw handover-failure.
        else if $\neg$Connected($p$) then throw disconnect-failure.
        else send push_request() via port $p$.
    On receiving push_request() via port $p'$:
        if $A$.shape $=$ expanded and $A$ is not involved in a move then set $m' \leftarrow$ push_ack().
        else set $m' \leftarrow$ push_nack().
        Send $m'$ via port $p'$.
    On sending push_ack():
        Release all connections via ports on $v_p$ except $p$ and begin contracting out of $v_p$.
    On completing the contraction out of $v_p$:
        Update $A$.shape $\leftarrow$ contracted.
    On receiving push_ack() via port $p$:
        Begin expanding into $v_p$.
    On receiving push_nack() via port $p$ or on a disconnection via port $p$:
        throw nack-failure.
    On completing the expansion into $v_p$:
        Establish connections with any new neighbors adjacent to $v_p$.
        Update $A$.shape $\leftarrow$ expanded; return success.
```

A.shape $\neq$ EXPANDED, A is already involved in a handover, or A is not connected to a neighbor via port p, this PULL fails. Otherwise, the system layer of A enqueues $m = \texttt{pull\_request}()$ in the message buffer on port p. Let B be the neighbor connected to A via port p. Eventually, message m is sent in FIFO order and the system layer of B receives it. If B is not involved in another movement and B.shape $=$ CONTRACTED, its system layer prepares message $m' = \texttt{pull\_ack}()$; otherwise, it sets $m' = \texttt{pull\_nack}()$. In either case, the system layer of B enqueues m' in the message buffer on its port facing A. If A and B are disconnected any time after A enqueues message m but before A receives message m', this PULL fails; otherwise, message m' is eventually sent in FIFO order by B and received by the system layer of A. If $m' = \texttt{pull\_nack}()$, this PULL fails. Otherwise, if $m' = \texttt{pull\_ack}()$, A disconnects from all ports on node v_p (except for p) and A and B begin their coordinated handover of node v_p. When A completes its contraction, it updates A.shape $\leftarrow$ CONTRACTED; analogously, when B completes its expansion, it updates B.shape $\leftarrow$ EXPANDED and establishes connections to its new neighbors adjacent to node v_p. This successfully completes this PULL. A PUSH(p) operation is executed analogously (see Figure 4b).

Concurrency Control Operations. The concurrency control operations LOCK and UNLOCK address a variant of the mutual exclusion problem where an amoebot attempts to gain exclusive control over itself and its immediate neighborhood. In particular, all amoebots A have a variable A.lock $\in \{\text{TRUE}, \text{FALSE}\}$ in public memory that is TRUE if and only if A was locked by some amoebot's application layer calling LOCK(). A successful LOCK operation performed by A returns the set $\mathcal{L}$ of amoebots that were locked in this operation. Amoebot A can then call UNLOCK($\mathcal{L}'$) for any $\mathcal{L}' \subseteq \mathcal{L}$ to release its locks on amoebots in $\mathcal{L}'$.

At a high level, our LOCK operation realizes mutual exclusion in the asynchronous, anonymous, dynamic, constant-size memory message passing setting of the canonical amoebot model by combining the classical α-synchronizer [12] with a 2-phase locking mechanism. In the *preparation phase*, the amoebot A performing the LOCK operation chooses a priority $prio_A \in \{1, \ldots, K\}$ independently and uniformly at random for a fixed and sufficiently large constant K and then sends $prio_A$ to all its neighbors. In the *competition phase*, if $prio_A$ is the highest priority received by any of the neighbors of A, then A sets A.lock $\leftarrow$ TRUE and instructs its neighbors to do the same; otherwise, this LOCK fails. Of course, this high-level description neglects many issues. For example, some neighbors of A may already be locked by other LOCK operations or two amoebots may pick the same highest priority. While these issues are easily addressed, more severe are complications caused by amoebots moving during the competition, questions of when to close a competition so that it can be evaluated, and the implementation of an α-synchronizer using a constant-size round counter which might create deadlocks. Moreover, continuously running the α-synchronizer might cause an unnecessarily high message overhead. The details of this mutual exclusion algorithm underlying the LOCK operation are beyond the scope of this dissertation but can be found in [49]; we summarize its main properties in the following theorem.

Theorem 2.2.1. *Every execution of the* LOCK *operation eventually terminates. Whenever a* LOCK *operation by an amoebot A succeeds, there exists a subset $\mathcal{N}$ of the neighbors A had at the start of this operation for which all amoebots $B \in \mathcal{N} \cup \{A\}$ have $B.lock =$* TRUE*; whenever it fails, some other amoebot in the 3-neighborhood of A executed a* LOCK *operation that succeeded.*

The execution of an UNLOCK($\mathcal{L}$) call by the application layer of A is straightforward, behaving like a parallel version of the WRITE operation where `unlock`() messages are sent to all amoebots in $\mathcal{L}$ and failure occurs if any intended recipient disconnects from A during its execution.

2.2.3 Amoebot Actions, Algorithms, and Executions

Following the message passing literature, we specify distributed algorithms in the amoebot model as sets of *actions* to be executed by the application layer, each of the form $\langle \textit{label} \rangle : \langle \textit{guard} \rangle \to \langle \textit{operations} \rangle$. An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether an amoebot A can execute it based on the ports A has connections on — i.e., which nodes adjacent to A are (un)occupied — and information from the public memories of A and its neighbors. An action is *enabled* for an amoebot A if its guard is true for A, and an amoebot is *enabled* if it has at least one enabled action. An action's *operations* specify the finite sequence of operations and computation in private memory to perform if this action is executed. The control flow of this private computation may optionally include *randomization* to generate random values and *error handling* to address any operation executions resulting in failure.

Each amoebot executes its own algorithm instance independently and — as an assumption for the majority of this dissertation — *reliably*, meaning there are no crash or Byzantine faults. An amoebot is said to be *active* if its application layer is executing an action and is *idle* otherwise. An amoebot can begin executing an action if and only if it is idle; i.e., an amoebot can execute at most one action at a time. On becoming active, an amoebot A first evaluates which of its actions $\alpha_i : g_i \to ops_i$ are enabled. Since each guard g_i is based only on the connected ports of A and the public memories of A and its neighbors, each g_i can be evaluated using the CONNECTED and Read operations. If no action is enabled, A returns to idle; otherwise, A chooses the highest priority enabled action $\alpha_i : g_i \to ops_i$ — where action priorities are set by the algorithm — and executes the operations and private computation specified by ops_i. Recall from Section 2.2.2 that each operation is guaranteed to terminate (either successfully or with a failure) in finite time. Thus, since A is reliable and ops_i consists of a finite sequence of operations and finite computation, each action execution is also guaranteed to terminate in finite time after which A returns to idle. An action execution *fails* if any of its operations' executions result in a failure that is not addressed with error handling and *succeeds* otherwise.

We adopt standard terminology from the distributed computing literature (see, e.g., [5]) to characterize assumptions on the executions of distributed algorithms in the canonical amoebot model. In particular, we assume an *adversary* (or *daemon*) controls the timing of amoebot activations and the resulting action executions. The power of an adversary is determined by its *concurrency* and *fairness*. We distinguish between four concurrency variants: *sequential*, in which at most one amoebot can be active at a time; *k-isolated*, in which any set of amoebots for which no two are within distance k can be simultaneously active; *synchronous*, in which any set of amoebots can be simultaneously active but all active amoebots must return to idle before another set is activated; and *asynchronous*, in which any set of amoebots can be simultaneously active. Fairness restricts how often the adversary must activate enabled amoebots. We distinguish between three fairness variants: *strongly fair*, in which every amoebot that is enabled infinitely often is activated infinitely often; *weakly fair*, in which every continuously enabled amoebot is eventually activated; and *unfair*, in which an amoebot may never be activated unless it is the only one with an enabled action. An algorithm execution is said to *terminate* if eventually all amoebots are idle and no amoebot is enabled; note that since an amoebot can only become enabled if something changes in its neighborhood, termination is permanent.

The majority of the algorithms in this dissertation assume a sequential adversary, under which an active amoebot knows that all other amoebots are idle and thus its evaluation of a guard with Connected and Read operations must be correct since the operations cannot fail or have their results be outdated due to concurrent changes in the system. In the concurrent settings, however, these issues may lead to guards being evaluated incorrectly, potentially causing a disabled action to be executed or an enabled action to be skipped. We address these issues in two ways. In Section 2.5, we give an algorithm whose actions are carefully designed so that their guards are always evaluated correctly under any adversary, even without using locks. In Chapter 3, we present a concurrency control protocol that uses locks to ensure that guards can be evaluated correctly even in the asynchronous setting.

2.2.4 Time Complexity

It is difficult to apply standard measurements of time from the distributed computing literature to the canonical amoebot model since our model mixes message passing, movements, and action semantics. For example, a common unit of time in pure message passing systems is the delay of the slowest message. This measurement could be extended to instead consider the delay of the slowest message or movement. However, this fine-grained measurement of time is awkward when applied to amoebot algorithms defined at the application layer where no message passing is explicitly modeled. Distributed computing models that work at the equivalent of our application layer (e.g., the *atomic-state model* [70] or autonomous mobile robots [87]) that use the concepts of enabled actions and guards usually make stronger assumptions about the structure of a process's actions or time itself, partitioning time into "steps" or "rounds" that abstract from the lower level details of guard evaluation. This suggests that the canonical amoebot model may need to be revised from its current formulation to make clearer distinctions between the low-level message passing mechanics at its system layer and the high-level algorithm executions at its application layer before an elegant measurement of time can be defined.

For consistency with the runtime results presented in this dissertation under the original amoebot model, we conclude our model description with a grandfathered definition of time complexity. Here, we assume a sequential adversary (i.e., there can be at most one active amoebot at a time), though the adversary can activate any amoebot, not just those with enabled actions. We further assume that the adversary is fair, which in this context means that it must activate every amoebot infinitely often. Under these assumptions, a *(fair sequential) round* is complete once every amoebot has been activated at least once.

2.3 Rationale

Connectivity. An amoebot system is *connected* if the subgraph of G induced by the occupied nodes of V is also connected. This notion does not imply any particular kind of connectivity in a physical programmable matter system; connections could be physical bonds, points of contact between neighboring units, or even wireless communication links. Although the amoebot model does not require that a system remain connected, this is often a desirable property that its algorithms maintain.

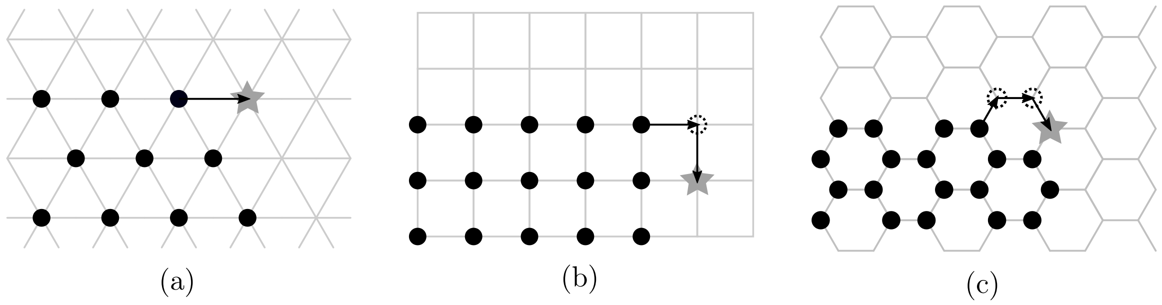


Figure 5. Lattice Movements. An amoebot moving "around" a neighbor to get to the next position on the system's boundary, depicted as a gray star, on the (a) triangular lattice G_Δ, (b) square lattice, and (c) hexagonal lattice.

Since amoebots can only interact with their immediate neighbors and exist on an infinite graph G, disconnection must be handled very carefully for there to be any hope that the resulting components could ever reconnect.

Space. It is well known that the only regular polygons that tile the two-dimensional plane are triangles, squares, and hexagons. The geometric amoebot model uses the triangular lattice G_Δ (corresponding to the hexagonal tiling) to represent space because it best supports system connectivity. In the other regular two-dimensional lattices, amoebots are forced to disconnect from the system even to perform moves as simple as shifting around another amoebot by one position (see Figure 5).

Movement. Modeling movements as expansions, contractions, and handovers also has roots in connectivity. Splitting an amoebot's movement from one node to another into an expansion and a contraction can be thought of as a look-ahead mechanism in which the amoebot reserves a space and examines its new surroundings before deciding whether or not to go through with the movement. This is vaguely similar to bipedal walking, where an organism puts one foot forward before shifting its weight to take another step. By looking ahead, an amoebot can try to determine whether its move might break system connectivity before committing to it. Handovers help maintain system connectivity by transferring occupancy of a node between neighboring amoebots without changing the set of occupied nodes. Without handovers, system behaviors such as moving in a line while preserving connectivity would be impossible.

2.4 Extensions

Many amoebot algorithms use techniques and assumptions that extend the core model described in Section 2.2. These extensions can be thought of as modules that combine and repackage core model features into useful, higher-level functionalities.

Leader Amoebot. Some amoebot algorithms assume the existence of a unique *leader amoebot* (or *seed*) at initialization that is used to coordinate the rest of the system. This assumption is reasonable under in certain settings since any of the existing leader election algorithms under the amoebot model (see Chapter 5) could be used as a preprocessing step for obtaining this leader amoebot. In particular, the recent algorithm by Emek et al. can elect a leader amoebot deterministically in polynomial time assuming geometric space, assorted orientations, constant-size memory, and a sequential, weakly fair adversary [78].

Static Objects. An *object* is a finite, connected, static set of nodes $O \subset V$ representing some fixed surface or entity in the amoebot system's environment. For example, the coating algorithm in Chapter 6 assumes the existence of an object to be coated, and the stochastic algorithm for shortcut bridging in Chapter 9 considers two objects that the amoebot system must bridge between. Object nodes do not move, communicate, or perform any computation throughout the execution of an algorithm. Amoebots can differentiate between neighboring amoebots and objects.

Token Passing. A *token* is a convenient abstraction used in algorithm design for passing information from amoebot to amoebot. More specifically, an amoebot A can pass a token t to a neighbor B by writing a copy of t into the memory of B and deleting its own copy. Rules governing how amoebots pass and process tokens may vary by each algorithm's need. In general, token passing is used to relay information beyond an amoebot's immediate neighborhood.

Random Number Generation. The canonical amoebot model assumes that an action's private computation may optionally include randomization; i.e., each amoebot has access to random bits with which it can generate random values. The only additional constraint is that the number of random bits available to an amoebot must be compatible with the memory size assumption; e.g., under constant-size memory, each amoebot can only have a constant number of random bits and thus can only generate constant precision random values. It is left to the algorithm designer to ensure that the available precision is sufficient for their application.

2.5 Asynchronous Hexagon Formation

In this section, we present an algorithm for *hexagon formation* as a concrete case study for algorithm design, pseudocode, and analysis in the canonical amoebot model. Our Hexagon-Formation algorithm is formulated in terms of actions, as specified in Section 2.2.3. Under the assumptions of geometric space, assorted orientation, and constant-size memory, we first prove that this algorithm correctly forms a hexagon under any unfair sequential adversary (Lemma 2.5.1). We then demonstrate that while locks are useful tools for designing correct amoebot algorithms under concurrent adversaries — as will be our approach in our concurrency control protocol (Chapter 3) — they are not always necessary. In particular, our Hexagon-Formation algorithm does not use locks but is guaranteed to correctly form a hexagon under any unfair asynchronous adversary, the most general of all possible adversaries (Theorem 2.5.6).

2.5.1 The Hexagon Formation Problem

The *hexagon formation problem* was originally proposed by Derakhshandeh et al. [58] and was later generalized as *basic shape formation* [52, 95]. This problem tasks an arbitrary, connected system of initially contracted amoebots with forming a regular hexagon (or as close to one as possible, given the number of amoebots in the system). We assume that there is a *unique seed amoebot* in the system and all other amoebots are initially *idle*. Note that the seed amoebot immediately collapses the hierarchy of orientation assumptions as it can impose its own local orientation on the rest of the system; thus, we will assume the amoebots share a common chirality (i.e., sense of clockwise orientation).

2.5.2 The Hexagon-Formation Algorithm

Following the sequential algorithm given by Derakhshandeh et al. [52, 58], the basic idea of our Hexagon-Formation algorithm is to form a hexagon by extending a spiral of amoebots counter-clockwise from the seed (see Figure 6). In addition

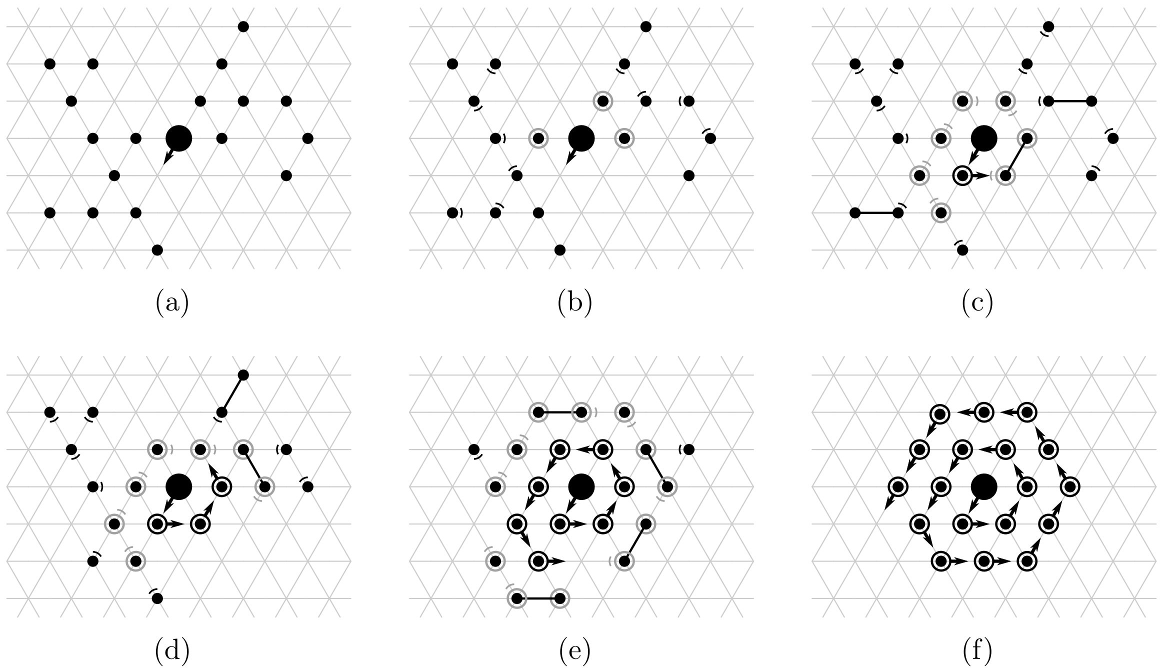


Figure 6. The Hexagon-Formation Algorithm. An example run with 19 amoebots. (a) All amoebots are initially idle (black dots), with the exception of a unique seed amoebot (large black dot). (b) Amoebots adjacent to the seed become roots (gray circles), and followers form parent-child relationships (black arcs) with roots and other followers. (c)–(f) Roots traverse the forming hexagon clockwise, becoming retired (black circles) when reaching the position marked by the last retired amoebot.

Variable	**Domain**	**Initialization**
state	$\{$SEED, IDLE, FOLLOWER, ROOT, RETIRED$\}$	one SEED, all others IDLE
parent	$\{\text{NULL}, 0, 1, \ldots, 9\}$	NULL
dir	$\{\text{NULL}, 0, 1, \ldots, 9\}$	0 if SEED, all others NULL

Table 3. Local Variables for Hexagon-Formation.

to the shape variable assumed by the amoebot model, Table 3 lists the variables each amoebot keeps in public memory. At a high level, the amoebot system first self-organizes as a spanning forest rooted at the seed amoebot using their parent ports. Follower amoebots follow their parents until reaching the surface of retired amoebots that have already found their place in the hexagon. They then become roots, traversing the surface of retired amoebots in a clockwise direction. Once they become

Algorithm 4 HEXAGON-FORMATION for Amoebot A

```
1:  $\alpha_1 : (A.\text{state} \in \{\textsc{idle}, \textsc{follower}\}) \wedge (\exists B \in N(A) : B.\text{state} \in \{\textsc{seed}, \textsc{retired}\}) \rightarrow$
2:      $\textsc{Write}(\perp, \text{parent}, \textsc{null})$.
3:      $\textsc{Write}(\perp, \text{state}, \textsc{root})$.
4:      $\textsc{Write}(\perp, \text{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.        ▷ See Algorithm 5.
5:  $\alpha_2 : (A.\text{state} = \textsc{idle}) \wedge (\exists B \in N(A) : B.\text{state} \in \{\textsc{follower}, \textsc{root}\}) \rightarrow$
6:      Find a port $p$ for which $\textsc{Connected}(p) = \textsc{true}$ and $\textsc{Read}(p, \text{state}) \in \{\textsc{follower}, \textsc{root}\}$.
7:      $\textsc{Write}(\perp, \text{parent}, p)$.
8:      $\textsc{Write}(\perp, \text{state}, \textsc{follower})$.
9:  $\alpha_3 : (A.\text{shape} = \textsc{contracted}) \wedge (A.\text{state} = \textsc{root}) \wedge (\forall B \in N(A) : B.\text{state} \neq \textsc{idle})$
10:       $\wedge\ (\exists B \in N(A) : (B.\text{state} \in \{\textsc{seed}, \textsc{retired}\}) \wedge (B.\text{dir is connected to } A)) \rightarrow$
11:     $\textsc{Write}(\perp, \text{dir}, \textsc{GetNextDir}(\text{clockwise}))$.
12:     $\textsc{Write}(\perp, \text{state}, \textsc{retired})$.
13: $\alpha_4 : (A.\text{shape} = \textsc{contracted}) \wedge (A.\text{state} = \textsc{root}) \wedge (\text{the node adjacent to } A.\text{dir is empty}) \rightarrow$
14:     $\textsc{Expand}(A.\text{dir})$.
15: $\alpha_5 : (A.\text{shape} = \textsc{expanded}) \wedge (A.\text{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\text{state} \neq \textsc{idle})$
16:       $\wedge\ (A \text{ has a tail-child } B : B.\text{shape} = \textsc{contracted}) \rightarrow$
17:     **if** $\textsc{Read}(\perp, \text{state}) = \textsc{root}$ **then** $\textsc{Write}(\perp, \text{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
18:     Find a port $p \in \textsc{TailChildren}()$ s.t. $\textsc{Read}(p, \text{shape}) = \textsc{contracted}$.  ▷ See Algorithm 5.
19:     Let $p'$ be the label of the tail-child's port that will be connected to $p$ after the pull handover.
20:     $\textsc{Write}(p, \text{parent}, p')$.
21:     $\textsc{Pull}(p)$.
22: $\alpha_6 : (A.\text{shape} = \textsc{expanded}) \wedge (A.\text{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\text{state} \neq \textsc{idle})$
23:       $\wedge\ (A \text{ has no tail-children}) \rightarrow$
24:     **if** $\textsc{Read}(\perp, \text{state}) = \textsc{root}$ **then** $\textsc{Write}(\perp, \text{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
25:     $\textsc{Contract}(\textsc{tail})$.
```

connected to a retired amoebot's dir port, they also retire and set their dir port to the next position of the hexagon. Algorithm 4 describes HEXAGON-FORMATION in terms of actions. We assume that if multiple actions are enabled for an amoebot, the enabled action with smallest index is executed. For conciseness and clarity, we write action guards as logical statements as opposed to their implementation with READ and CONNECTED operations. In action guards, we use $N(A)$ to denote the neighbors of amoebot A and say that an amoebot A has a *tail-child* B if B is connected to the tail of A via port B.parent.

Algorithm 5 Helper Functions for HEXAGON-FORMATION

```
function GetNextDir($c$)                          ▷ $c \in \{\text{clockwise}, \text{counter-clockwise}\}$
    Let $p$ be any head port.
    try:
        while (Connected($p$) = false) ∨ (Read($p$, state) $\notin$ {seed, retired}) do
            $p \leftarrow$ the next head port in orientation $c$.
    catch disconnect-failure do $p \leftarrow$ the next head port in orientation $c$; go to Step 4.
    try:
        while (Connected($p$) = true) ∧ (Read($p$, state) $\in$ {seed, retired}) do
            $p \leftarrow$ the next head port in orientation $c$.
    catch disconnect-failure do $p \leftarrow$ the next head port in orientation $c$; go to Step 8.
    return $p$.
function TailChildren( )
    Let $P \leftarrow \emptyset$.
    for each tail port $p$ do
        try:
            if (Connected($p$) = true) ∧ (Read($p$, parent) points to $A$) then
                $P \leftarrow P \cup \{p\}$.
        catch disconnect-failure do nothing.
    return $P$.
```

2.5.3 Analysis

We begin our analysis of the HEXAGON-FORMATION algorithm by showing it is correct under any sequential adversary. While the related hexagon formation algorithm of Derakhshandeh et al. has already been analyzed in the sequential setting in previous works [52, 58], HEXAGON-FORMATION must be proved correct with respect to its action formulation and the unfair adversary.

Lemma 2.5.1. *Any unfair sequential execution of the* HEXAGON-FORMATION *algorithm terminates with the amoebot system forming a hexagon.*

Proof. We first show that the system remains connected throughout the execution. Recall that the amoebot system is assumed to be initially connected. A disconnection can only result from a movement, and in particular, a CONTRACT movement. EXPAND movements only enlarge the set of nodes occupied by the system and handovers only change which amoebot occupies the handover node, not the fact that the node remains occupied. So it suffices to consider action α_6, the only action involving a CONTRACT operation. Action α_6 only allows an expanded follower or root amoebot to contract its tail if it has no idle neighbors or neighbors pointing at its tail as their parent. The only other possible tail neighbors are the seed, roots, or retired amoebots; however, all of these neighbors are guaranteed to be connected to the forming hexagon structure. Thus, the system remains connected throughout the algorithm's execution.

Now, suppose to the contrary that the HEXAGON-FORMATION algorithm has terminated — i.e., no amoebot has an enabled action — but the system does not form a hexagon. By inspection of action α_3, the retired amoebots form a hexagon extending counter-clockwise from the seed. Thus, for the system to not form a hexagon, there must exist some amoebot that is neither the seed nor retired.

First of all, there cannot be any idle amoebots remaining in the system; in particular, we argue that so long as there are idle amoebots in the system, there exists an idle amoebot for which α_1 or α_2 is enabled, and thus the algorithm cannot have terminated. Suppose to the contrary that there are idle amoebots in the system but none of them have non-idle neighbors, yielding α_1 and α_2 disabled. Then the idle amoebots must be disconnected from the rest of the system, since we assumed that the system contains a unique seed amoebot initially, a contradiction of connectivity. Thus, if the algorithm has terminated, all idle amoebots must have already become roots or followers.

For all remaining root or follower amoebots to not have enabled actions, we have the following chain of observations:

(i) No follower can have a seed or retired neighbor; otherwise, action α_1 would be enabled for that follower.

(*ii*) Since we have already established that there are no idle amoebots in the system, there must not be a contracted root occupying the next hexagon node; otherwise, action α_3 would be enabled for that root.

(*iii*) Every contracted root amoebot must have its clockwise traversal of the forming hexagon's surface blocked by another amoebot; otherwise, action α_4 would be enabled for some contracted root. Moreover, since there are no followers on the hexagon's surface by (*i*) and no contracted root has yet reached the next hexagon node by (*ii*), each contracted root must be blocked by another root.

(*iv*) By (*iii*), there must exist at least one expanded root amoebot A. Since actions α_5 and α_6 must be disabled for A by supposition — and, again, there are no idle amoebots remaining in the system — A must have one or more tail-children that are all expanded.

(*v*) By the same argument, actions α_5 and α_6 can only be disabled for the expanded tail-children of A if they also each have at least one tail-child, all of which are expanded.

The chain of expanded tail-children established by (*iv*) and (*v*) cannot continue ad infinitum since the amoebot system is finite. There must eventually exist an expanded root or follower amoebot that either has a contracted tail-child or no tail-children, enabling α_5 or α_6, respectively. In all cases, we reach a contradiction: so long as the amoebot system does not yet form a hexagon, there must exist an amoebot with an enabled action. The execution of any enabled action brings the system monotonically closer to forming a hexagon: turning idle amoebots into followers, bringing followers to the hexagon's surface, turning followers into roots, bringing roots closer to their final position, and finally turning roots into retired amoebots. Therefore, regardless of the (potentially unfair) sequential adversary's choice of enabled amoebot to activate, the system is guaranteed to reach and terminate in a configuration forming a hexagon, as desired. □

We now consider the behavior of Hexagon-Formation under an unfair asynchronous adversary, the most general of all possible adversaries. The Hexagon-Formation algorithm maintains the following invariants:

1. The state variable of an amoebot A can only be updated by A itself. This follows from actions α_1, α_2, and α_3.
2. Only follower amoebots have non-NULL parent variables. An idle amoebot sets its own parent variable when it becomes a follower. While an amoebot A is a follower, the only amoebot that can update A.parent is the amoebot indicated by A.parent. Finally, when a follower becomes a root, it updates its own parent variable back to NULL, after which its parent variable never changes again. This follows from actions α_1, α_2, and α_5.
3. Only root and retired amoebots have non-NULL dir variables. The dir variable of an amoebot A can only be updated by A itself. Once a dir variable is set by a retired amoebot, it never changes again. This follows from actions α_1, α_3, α_5, and α_6.
4. Seed, idle, and retired amoebots are always contracted and never move. Moreover, seed and retired amoebots never change their state.
5. The shape variable of a root or expanded follower A can only be updated by a movement operation initiated by A itself, while the shape variable of a contracted follower A can only be updated by a Pull operation initiated by the amoebot indicated by A.parent. This follows from actions α_4, α_5, and α_6.
6. No amoebot can disconnect from an idle neighbor. Moreover, a root will not

change its state if it has an idle neighbor. This follows from actions α_3, α_5, and α_6.

7. Root amoebots traverse the surface of the forming hexagon clockwise while follower amoebots are pulled by their parents. This follows from actions α_1, α_4, α_5, and α_6.

In an asynchronous execution of an algorithm that does not use locks, it is possible that an amoebot evaluates its guards based on outdated information. Nevertheless, in the following two lemmas, we show that HEXAGON-FORMATION has the key property that whenever an amoebot thinks an action is enabled, it remains enabled and will execute successfully, even when other actions are executed concurrently.

Lemma 2.5.2. *For any asynchronous execution of the* HEXAGON-FORMATION *algorithm, if an action α_i is enabled for an amoebot A, then α_i will remain enabled for A until A executes an action's operations.*

Proof. We use the invariants to prove the claim on an action-by-action basis.

α_1 : If A evaluates the guard of α_1 as TRUE, then it must be an idle or follower amoebot with a seed or retired neighbor. Invariant 1 ensures that A remains an idle or follower amoebot, and Invariant 4 ensures its seed or retired neighbor does not move or change state.

α_2 : If A evaluates the guard of α_2 as TRUE, then it must be an idle amoebot with a follower or root neighbor. Invariant 1 ensures that A remains an idle amoebot, and Invariant 6 ensures that any neighbor A has must remain its neighbor while it is idle. A follower neighbor of A can concurrently change its state to root by α_1; however, a root neighbor of A will not change its state while A is idle by Invariant 6.

α_3 : If A evaluates the guard of α_3 as TRUE, then it must be a contracted root with no idle neighbors and a seed or retired neighbor that indicates that the node A occupies is the next hexagon node. Invariants 1 and 5 ensure that A remains a contracted root, Invariant 4 ensures that A cannot gain any idle neighbors, and Invariants 3 and 4 ensure that the seed or retired neighbor continues to indicate the node A occupies as the next hexagon node.

α_4 : If A evaluates the guard of α_4 as TRUE, then it must be a contracted root with no neighbor connected via A.dir. Invariants 1 and 5 ensure that A remains a contracted root, and Invariant 7 ensures that no amoebot but A can move into the node adjacent to A.dir.

α_5 : If A evaluates the guard of α_5 as TRUE, then it must be an expanded follower or root with no idle neighbors and some contracted tail-child. Invariants 1 and 5 ensure that A remains an expanded follower or root, Invariant 4 ensures that A cannot gain any idle neighbors, and Invariants 2 and 5 ensure that any contracted tail-child of A remains so.

α_6 : If A evaluates the guard of α_6 as TRUE, then it must be an expanded follower or root with no idle neighbors and no tail-children. Invariants 1 and 5 ensure that A remains an expanded follower or root, and Invariants 2 and 4 ensure that A cannot gain any idle neighbors or tail-children.

Therefore, any action that A evaluates as enabled must remain enabled, as claimed. □

Lemma 2.5.3. *For any asynchronous execution of the* HEXAGON-FORMATION *algorithm, any execution of an enabled action is successful and unaffected by any concurrent action executions.*

Proof. We once again consider each action individually.

α_1 : Action α_1 first executes two WRITE operations to A's own public memory which cannot fail. It then executes a helper function GETNEXTDIR(counter-clockwise) which involves a sequence of CONNECTED and READ operations. CONNECTED operations always succeed, so it suffices to consider the READ operations. While it is possible that READ operations issued to follower or root neighbors may fail if those neighbors disconnect, these failures are caught by error handling and thus do not cause the action to fail. Moreover, the critical READ operations issued to seed or retired neighbors that the function depends on for calculating the correct direction must succeed by the guard of α_1 and Lemma 2.5.2. Once this direction is computed, α_1 then executes a WRITE operation to A's own memory which cannot fail.

α_2 : Action α_2 first executes CONNECTED and READ operations to find a follower or root neighbor. Such a neighbor must exist and the corresponding READ operations must succeed by the guard of α_2 and Lemma 2.5.2. Action α_2 then executes two WRITE operations to A's own public memory which cannot fail.

α_3 : Action α_3 first executes helper function GETNEXTDIR(clockwise) which must succeed by an argument analogous to that of α_1. Once this direction is computed, α_3 executes two WRITE operations to A's own public memory which cannot fail.

α_4 : Action α_4 executes an EXPAND operation toward port A.dir which must succeed because A is contracted and the node adjacent to A.dir must remain unoccupied, as ensured by the guard of α_4 and Lemma 2.5.2.

α_5 : Action α_5 first executes a conditional based on a READ operation issued to A's own public memory which cannot fail. It then executes helper function GETNEXTDIR(counter-clockwise) which must succeed by an argument analogous to that of α_1. The computed direction is then used in a WRITE operation to A's own public memory which cannot fail. Action α_5 then executes a helper function TAILCHILDREN() which, like GETNEXTDIR, involves CONNECTED and READ operations. It must succeed for similar reasons: any failed READ operations are caught by error handling, and the critical READ operations issued to tail-children must succeed by the guard of α_5 and Lemma 2.5.2. Once the ports connected to tail-children are computed, READ operations are executed to find a contracted tail-child B which once again must succeed by the guard of α_5 and Lemma 2.5.2. Finally, α_5 executes a WRITE to the public memory of B and performs a PULL handover with B; both operations must succeed because B remains connected to A and cannot be involved in another movement by Invariant 5.

α_6 : Action α_6 first executes the same conditional operation as α_5 and thus succeeds for an analogous reason. It then executes a single CONTRACT operation which must succeed because A is expanded, as ensured by the guard of α_6 and Lemma 2.5.2.

Therefore, any execution of an enabled action must be successful and unaffected by concurrent action executions, as claimed. □

We next show that the HEXAGON-FORMATION algorithm is serializable. We denote the execution of an action α by an amoebot A in an execution of the algorithm as a pair (A, α).

Lemma 2.5.4. *For any asynchronous execution of the* HEXAGON-FORMATION *algorithm, there exists a sequential ordering of its action executions producing the same final configuration.*

Proof. Argue by induction on i, the number of action executions in the asynchronous execution of HEXAGON-FORMATION. Clearly, if $i = 1$, the asynchronous execution of a single action is also a sequential execution, and we are done. So suppose that any asynchronous execution of Hexagon-Formation consisting of $i > 1$ action executions can be serialized, and consider any asynchronous execution $\mathcal{S}$ consisting of $i + 1$ action executions. One can partially order the action executions (A, α) of $\mathcal{S}$ according to the ideal wall-clock time the asynchronous adversary activated A; note that this is only used for this analysis, and the wall-clock time is never available to the amoebots. Let (A^*, α^*) be any action execution with the latest activation time; if there are multiple such executions because the asynchronous adversary activated multiple amoebots simultaneously, choose any such execution. If (A^*, α^*) was removed from $\mathcal{S}$ to produce a new asynchronous execution $\mathcal{S}^-$, we have by Lemmas 2.5.2 and 2.5.3 that the remaining i action executions must still be enabled and successful since all other action executions either terminated before (A^*, α^*) was initiated or were concurrent with it. By the induction hypothesis, there must exist a sequential ordering of the i action executions in $\mathcal{S}^-$ producing the same final configuration as $\mathcal{S}^-$. Append (A^*, α^*) to the end of this sequential execution to produce $\mathcal{S}^*$, a sequential execution of $i + 1$ action executions. Any actions that were concurrent with (A^*, α^*) in $\mathcal{S}$ have now terminated before (A^*, α^*) in $\mathcal{S}^*$. However, by Lemmas 2.5.2 and 2.5.3, this does not change the fact that α^* is enabled for A^* and its execution is successful and produces the same outcome in $\mathcal{S}^*$. Therefore, we conclude that there exists a sequential ordering of the action executions of $\mathcal{S}$ producing the same final configuration. $\square$

Finally, we show that the Hexagon-Formation algorithm is correct under any unfair asynchronous adversary.

Lemma 2.5.5. *Any unfair asynchronous execution of the* Hexagon-Formation *algorithm terminates with the amoebot system forming a hexagon.*

Proof. First suppose to the contrary that there exists an asynchronous execution of

Hexagon-Formation that does not terminate; i.e., there are an infinite number of executions of enabled actions. By Lemmas 2.5.2 and 2.5.3, any such action execution must succeed and do exactly what it would have in a sequential execution where there are no other concurrent action executions. But Lemma 2.5.1 implies that there can only be a finite number of successful action executions before no amoebot has any enabled actions left, a contradiction. So all asynchronous executions of Hexagon-Formation must terminate.

Now suppose to the contrary that there exists an asynchronous execution of Hexagon-Formation that has terminated but the system does not form a hexagon. By Lemma 2.5.4, there must exist a sequential execution that also produces this non-hexagon final configuration. However, this is a contradiction of Lemma 2.5.1 which states that every sequential execution of Hexagon-Formation must terminate with the system forming a hexagon. □

Since asynchronous adversaries are the most general (i.e., any sequential, k-isolated, or synchronous adversary is also an asynchronous adversary) and we made no assumptions on fairness, we obtain the following theorem.

Theorem 2.5.6. *Under the assumptions of geometric space, assorted orientations, and constant-size memory, the* Hexagon-Formation *algorithm solves the hexagon formation problem under any adversary.*

2.5.4 Discussion

We note that Lemmas 2.5.4 and 2.5.5 are in fact algorithm-agnostic to an extent, independent of the specific details of Hexagon-Formation. The proofs of these lemmas show that any algorithm for which one could prove statements analogous to Lemmas 2.5.1–2.5.3 must be both serializable and correct under any (unfair) asynchronous adversary. Thus, these three lemmas establish a set of sufficient conditions for amoebot algorithm correctness under asynchronous adversaries without the use of locks: correctness under any sequential adversary (Lemma 2.5.1), enabled actions remaining enabled despite concurrent action executions (Lemma 2.5.2), and enabled actions executing successfully and invariant from their sequential executions (Lemma 2.5.3). We are hopeful that these sufficient conditions can be applied to the analysis of existing amoebot algorithms under the new canonical amoebot model.

Chapter 3

CONCURRENCY CONTROL

Under a sequential adversary where only one amoebot is active at a time, operation failures are necessarily the fault of the algorithm designer: e.g., attempting to Read on a disconnected port, attempting to Expand when already expanded, etc. Barring these design errors, it suffices to focus only on the correctness of the algorithm — i.e., whether the algorithm's actions always produce the desired system behavior under any sequential execution — not whether the individual actions themselves execute as intended. This is the focus of most existing amoebot works [8, 31, 32, 53, 54, 58, 60, 61, 62, 78, 92, 162] and the majority of the results in this dissertation.

In this chapter, we focus is on asynchronous executions where concurrent action executions can mutually interfere, affect outcomes, and cause failures far beyond those of simple designer negligence. Ensuring algorithm correctness in spite of concurrency thus appears to be a significant burden for the algorithm designer, especially for problems that are challenging even in the sequential setting due to the constraints of constant-size memory, assorted orientation, and strictly local interactions. What if there was a way to ensure that correct, sequential amoebot algorithms could be lifted to the asynchronous setting without sacrificing correctness? This would give the best of both worlds: the relative ease in design from the sequential setting and the correct execution in a more realistic concurrent setting.

We introduce and rigorously analyze a *concurrency control protocol* [50] for transforming an algorithm $\mathcal{A}$ that works correctly for every sequential execution into an algorithm $\mathcal{A}'$ that works correctly for every asynchronous execution. We prove that our protocol achieves this goal so long as the original algorithms satisfy certain *conventions*. These conventions limit the full generality of the amoebot model in order to provide a common structure to the algorithms. We discuss interesting open problems regarding what algorithms are compatible with this protocol and whether it can be extended beyond these conventions in Section 3.4.

3.1 Algorithm Conventions

The first of these conventions requires that all actions of the given algorithm are executed successfully under a sequential adversary. For sequential executions, the *system configuration* is defined as the mapping of amoebots to the node(s) they occupy and the contents of each amoebot's public memory. Certainly, this configuration is well-defined whenever all amoebots are idle, and we call a configuration *valid* whenever the requirements of our amoebot model are met, i.e., every position is occupied by at most one amoebot, each amoebot is either contracted or expanded, its shape variable corresponds to its physical shape, and its lock variable is TRUE if and only if it has been locked in a LOCK operation. Whenever we talk about a system configuration in the following, we assume that it is valid.

Convention 3.1.1. *All actions of an amoebot algorithm should be valid, i.e., for all its actions α and all system configurations in which α is enabled for some amoebot A, the execution of α by A should be successful whenever all other amoebots are idle.*

The second convention keeps an algorithm's actions simple by controlling the order and number of operations they perform.

Convention 3.1.2. *Each action of an amoebot algorithm should structure its operations as:*

1. *A compute phase, during which an amoebot performs a finite amount of computation in private memory and a finite sequence of* CONNECTED*,* READ*, and* WRITE *operations.*
2. *A move phase, during which an amoebot performs at most one movement operation decided upon in the compute phase.*

In particular, no action should use LOCK *or* UNLOCK *operations.*

Convention 3.1.2 is similar in spirit to the *look-compute-move* paradigm used in the mobile robots literature (see, e.g., [87]), though message passing communication via READ and WRITE operations adds additional complexity in the amoebot model. As discussed in Section 2.1, the instantaneous snapshot performed in the mobile robots' look phase is not trivially realizable in the amoebot model where amoebots' public memories are included in the neighborhood configuration. Moreover, the amoebot model distinguishes between CONNECTED and READ operations that are performed during the evaluation of an action's guards and those that are performed during the action's execution (see Section 2.2.3).

Finally, due to our approach of using a serializability argument to show the correctness of algorithms using our protocol, we need one last convention. This final convention is significantly more technical and limits the generality of the model more strictly than the first two, which we discuss further in Section 3.4. Consider any action $\alpha : g \to ops$ of an algorithm $\mathcal{A}$ being executed by an amoebot A. Recall that the guard g is a Boolean predicate based on the *local configuration* of A; i.e., the ports A has established connections on and the contents of the public memories of A and

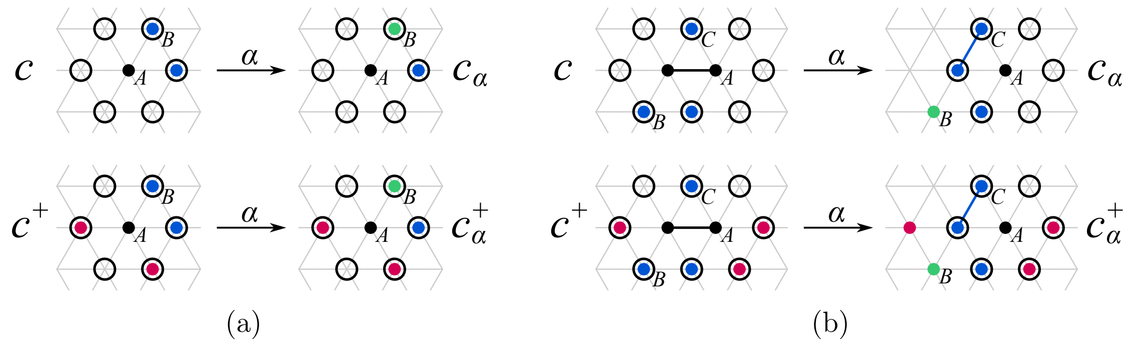


Figure 7. The Monotonicity Convention. We examine the local configuration c of amoebot A, an extension c^+ of c, and the corresponding outcomes c_α and c_α^+ reached by sequential executions of α. The nodes with black circles denote the $N(\cdot)$ sets of nodes adjacent to A. Neighbors occupying the "non-extended neighborhood" $O(c)$ are shown in blue and neighbors occupying the "extended neighborhood" $O(c^+) \setminus O(c)$ are shown in pink. Monotonicity requires that the executions of α make the same updates: in (a), amoebot A updates the public memory of B, shown in green; in (b), amoebot A updates the public memory of B (green) and pulls neighbor C in a handover.

its neighbors. For a local configuration c of A, let $N(c) = (v_1, \ldots, v_k)$ be the nodes adjacent to A; note that $k = 6$ if A is contracted and $k = 8$ if A is expanded. Let $O(c) \subseteq N(c)$ be the nodes adjacent to A that are occupied by neighboring amoebots. Letting $M_\mathcal{A}$ denote the set of all possible contents of an amoebot's public memory w.r.t. algorithm $\mathcal{A}$, we can write c as a tuple $c = (c_0, c_1, \ldots, c_k) \in M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k$ where $c_0 \in M_\mathcal{A}$ is the public memory contents of A and, for $i \in \{1, \ldots, k\}$, $c_i \in M_\mathcal{A}$ is the public memory contents of the neighbor occupying node $v_i \in N(c)$ if $v_i \in O(c)$ and $c_i = \text{NULL}$ otherwise. Thus, we can express the guard g as a function $g : M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k \to \{\text{TRUE}, \text{FALSE}\}$.

A local configuration c is *consistent* if $c_i = c_j$ whenever nodes $v_i, v_j \in N(c)$ are occupied by the same expanded neighbor. Local configurations c and c' are said to *agree on amoebots* X if for all nodes v_i occupied by an amoebot in X, we have $c_i = c'_i$. A local configuration c^+ is an *extension* of local configuration c if c^+ is consistent and c and c^+ agree on A and the neighbors occupying $O(c)$; intuitively, an extension

c^+ has all the same amoebots in the same positions with the same public memory contents as c, but may also have additional neighbors occupying nodes of $N(c) \setminus O(c)$. An extension c^+ of c is *expansion-compatible* with an execution of an action on c if any EXPAND operation in this execution would also be successful in c^+. An action $\alpha : g \to \mathit{ops}$ is *monotonic* (see Figure 7) if for any consistent local configuration c with $g(c) = \text{TRUE}$, any local configuration c_α reachable by a sequential execution of α on c, and any extension c^+ of c that is expansion-compatible with the execution reaching c_α and is reachable by a sequential execution of $\mathcal{A}$, then $g(c^+) = \text{TRUE}$ and there exists a local configuration c_α^+ reachable by a sequential execution of α on c^+ such that:

1. $N(c_\alpha) = N(c_\alpha^+)$; i.e., both executions of α yield the same set of nodes adjacent to A.
2. c_α and c_α^+ agree on A and the amoebots that occupied $O(c)$; i.e., both executions of α make identical updates w.r.t. A and its non-extended neighborhood.
3. c^+ and c_α^+ agree on the amoebots that occupied $O(c^+) \setminus O(c)$; i.e., the execution of α on c^+ does not make any updates to the extended neighborhood of A.

Convention 3.1.3. *All actions of an amoebot algorithm should be monotonic.*

3.2 The Concurrency Control Protocol

Our *concurrency control protocol* (Algorithm 6) takes as input any amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to \mathit{ops}_i] : i \in \{1, \ldots, k\}\}$ satisfying Conventions 3.1.1–3.1.3 and produces a corresponding algorithm $\mathcal{A}' = \{[\alpha' : g' \to \mathit{ops}']\}$ composed of a single action α'. The core idea of our protocol is to carefully incorporate locks in α' as a wrapper around the actions of $\mathcal{A}$, ensuring that $\mathcal{A}'$ only produces outcomes in concurrent settings that $\mathcal{A}$ can produce in the sequential setting. With locks, action guards that in general can only be evaluated reliably in the sequential setting can now also be evaluated reliably in concurrent settings.

To avoid any deadlocks that locking may cause, our protocol adds an *activity bit* variable $A.\mathrm{act} \in \{\textsc{true}, \textsc{false}\}$ to the public memory of each amoebot A indicating if any changes have occurred in the memory or neighborhood of A since it last attempted to execute an action. The single action α' of $\mathcal{A}'$ has guard $g' = (A.\mathrm{act} = \textsc{true})$, ensuring that α' is only enabled for an amoebot A if changes in its memory or neighborhood may have cause some actions of $\mathcal{A}$ to become enabled. As will become clear in the presentation of the protocol, Write and movement operations may enable actions of $\mathcal{A}$ not only for the neighbors the acting amoebot, but also for the neighbors of those neighbors (i.e., in the 2-neighborhood of the acting amoebot). The acting amoebot cannot directly update the activity bits of amoebots in its 2-neighborhood, so it instead sets its neighbors' *awaken bits* $A.\mathrm{awaken} \in \{\textsc{true}, \textsc{false}\}$ to indicate that they should update their neighbors' activity bits in their next action. Initially, $A.\mathrm{act} = \textsc{true}$ and $A.\mathrm{awaken} = \textsc{false}$ for all amoebots A.

Under our protocol, algorithm $\mathcal{A}'$ only contains one action $\alpha' : g' \to ops'$ where g' requires that an amoebot's activity bit is set to TRUE (Step 1). If α' is enabled for an amoebot A, A first attempts to lock itself and all its neighbors (Steps 2–3). Given that it locks successfully, there are two cases. If $A.\mathrm{awaken} = \textsc{true}$, then A must have previously been involved in the operation of some acting amoebot that changed the neighborhood of A but could not update the corresponding neighbors' activity bits (Steps 14, 17, 24, or 28). So A updates the intended activity bits to TRUE, resets A.awaken, releases its locks, and aborts (Steps 4–6). Otherwise, A obtains the necessary information to evaluate the guards of all actions in algorithm $\mathcal{A}$ (Steps 7–9). If no action from $\mathcal{A}$ is enabled for A, A sets A.act to FALSE, releases its locks, and

Algorithm 6 Concurrency Control Protocol for Amoebot A

```
    Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, k\}\}$ following Conventions 3.1.1–3.1.3.
 1: Set $g' \leftarrow (A.\text{act} = \text{TRUE})$ and $ops' \leftarrow$ "Do:
 2:     Perform a LOCK operation to lock amoebots $\mathcal{L} = \{A\} \cup \{\text{all neighbors of } A\}$.
 3:     if the LOCK operation fails then abort.
 4:     if $A.\text{awaken} = \text{TRUE}$ then
 5:         for all amoebots $B \in \mathcal{L}$ do WRITE $B.\text{act} \leftarrow \text{TRUE}$.
 6:         WRITE $A.\text{awaken} \leftarrow \text{FALSE}$, UNLOCK each amoebot in $\mathcal{L}$, and abort.
 7:     for all actions $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ do
 8:         Perform READ and CONNECTED operations to evaluate guard $g_i$ w.r.t. $\mathcal{L}$.
 9:         Evaluate $g_i$ in private memory to determine if $\alpha_i$ is enabled.
10:     if no action is enabled then WRITE $A.\text{act} \leftarrow \text{FALSE}$, UNLOCK each amoebot in $\mathcal{L}$, and abort.
11:     Choose an enabled action $\alpha_i \in \mathcal{A}$ and perform its compute phase in private memory.
12:     Let $W_i$ be the set of WRITE operations and $M_i$ be the movement operation in $ops_i$ based on
        its compute phase; set $M_i \leftarrow \text{NULL}$ if there is none.
13:     if $M_i$ is EXPAND (say, from node $u$ into node $v$) then perform the EXPAND operation.
14:         if the EXPAND operation succeeds then WRITE $A.\text{awaken} \leftarrow \text{TRUE}$.
15:         else reset private memory, UNLOCK each amoebot in $\mathcal{L}$, and abort.
16:     for all amoebots $B \in \mathcal{L}$ do WRITE $B.\text{act} \leftarrow \text{TRUE}$.
17:     for all $(B.x \leftarrow x_{val}) \in W_i$ do WRITE $B.x \leftarrow x_{val}$ and WRITE $B.\text{awaken} \leftarrow \text{TRUE}$.
18:     if $M_i$ is NULL or EXPAND then UNLOCK each amoebot in $\mathcal{L}$.
19:     else if $M_i$ is CONTRACT (say, from nodes $u, v$ into node $u$) then
20:         UNLOCK each amoebot in $\mathcal{L}$ that is adjacent to node $v$ but not to node $u$.
21:         Perform the CONTRACT operation.
22:         UNLOCK each remaining amoebot in $\mathcal{L}$.
23:     else if $M_i$ is PUSH (say, $A$ is pushing $B$) then
24:         WRITE $A.\text{awaken} \leftarrow \text{TRUE}$ and $B.\text{awaken} \leftarrow \text{TRUE}$.
25:         Perform the PUSH operation.
26:         UNLOCK each amoebot in $\mathcal{L}$.
27:     else if $M_i$ is PULL (say, $A$ in nodes $u, v$ is pulling $B$ into node $v$) then
28:         WRITE $B.\text{awaken} \leftarrow \text{TRUE}$.
29:         UNLOCK each amoebot in $\mathcal{L}$ (except $B$) that is adjacent to node $v$ but not to node $u$.
30:         Perform the PULL operation.
31:         UNLOCK each remaining amoebot in $\mathcal{L}$."
32: return $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$.
```

aborts; this disables α' for A until some future change occurs in its neighborhood (Step 10). Otherwise, A chooses any enabled action and executes its compute phase in private memory (Step 11). As a result of this computation, A knows which WRITE and movement operations, if any, it wants to perform (Step 12).

Before enacting these operations (thereby updating the system's configuration) amoebot A must be certain that no operation of α' will fail. It has already passed its first point of failure: the LOCK operation in Step 2. But α' may also fail during an EXPAND operation if it conflicts with some other concurrent expansion (Step 13). In either case, A resets its private memory, releases its locks, and aborts (Steps 3 and 15). As we will show in Lemma 3.3.5, these are the only two cases where α' can fail.

Provided neither of these failures occur, A can now perform operations that — without locks on its neighbors — could otherwise interfere with its neighbors' actions or be difficult to undo. This begins with A setting the activity bits of all its locked neighbors to TRUE since it is about to cause activity in its neighborhood (Step 16). It then enacts the WRITE operations it decided on during its computation, writing updates to its own public memory and the public memories of its neighbors. Since writes to its neighbors can change what amoebots in its 2-neighborhood see, it must also set the awaken bits of the neighbors it writes to to TRUE (Step 17).

The remainder of the protocol handles movements and releases locks. If A did not want to move or it intended to EXPAND — which, recall, it already did in Step 13 — it can simply release all its locks (Step 18). If A wants to contract, it must first release its locks on the neighbors it is contracting away from; it can then CONTRACT and, once contracted, release its remaining locks (Step 20–22). If A wants to perform a PUSH handover, it does so and then releases all its locks (Steps 24–26). Finally, pull handovers are handled similarly to contractions: A first releases its locks on the neighbors it is disconnecting from; it can then PULL and, once contracted, release its remaining locks (Steps 28–31).

3.3 Analysis

Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 3.1.1–3.1.3 and $\mathcal{A}'$ be the amoebot algorithm produced from $\mathcal{A}$ by our concurrency control protocol (Algorithm 6). Informally, we will show that if any sequential execution of $\mathcal{A}$ terminates, then any asynchronous execution of $\mathcal{A}'$ must also terminate and will do so in a configuration that was reachable by a sequential execution of $\mathcal{A}$ (Theorem 3.3.12). This analysis is broken into two stages: analyzing $\mathcal{A}'$ under sequential executions and then leveraging a serialization argument for the analysis of $\mathcal{A}'$ under asynchronous executions. In each stage, we show that executions of $\mathcal{A}'$ are finite; i.e., they must terminate (Lemmas 3.3.1 and 3.3.11). Since all executions of $\mathcal{A}'$ terminate, it suffices to show that the final configurations reachable by asynchronous executions of $\mathcal{A}'$ are contained in those reachable by sequential executions of $\mathcal{A}'$ (Lemma 3.3.10) which in turn are contained in those reachable by sequential executions of $\mathcal{A}$ (Lemma 3.3.3), yielding the desired result.

We say that an amoebot is $\mathcal{A}$-*enabled* if it has at least one enabled action $\alpha \in \mathcal{A}$. Note that under this definition, an amoebot A is $\mathcal{A}'$-enabled if and only if $A.\text{act} = \text{TRUE}$. Using this terminology, an execution of an algorithm $\mathcal{A}$ has terminated if there are no longer any $\mathcal{A}$-enabled amoebots.

We first analyze algorithm $\mathcal{A}'$ under sequential executions. Define a *sequential schedule* $\mathcal{S} = ((A_1, \alpha_1), (A_2, \alpha_2), \ldots)$ as the sequence of actions executed in a sequential execution, where α_i is the i-th action executed by the system and A_i is the amoebot that executed it. For a sequential schedule to be valid, α_i must be enabled for A_i in the configuration produced by executions $(A_1, \alpha_1), \ldots, (A_{i-1}, \alpha_{i-1})$, for all $i \geq 1$. Certainly, sequential schedules obfuscate various details; e.g., they ignore the precise timing of an action's operations and the specific choices produced by randomness or nondeterminism in an action's computation. So while a single sequential schedule may in fact represent a family of sequential executions, this abstraction suffices for our purposes.

Throughout this analysis, let C_0 be any (valid) initial system configuration such that all (valid) sequential schedules of $\mathcal{A}$ starting in C_0 are finite. Let $F_s(C_0)$ be the set of configurations in which $\mathcal{A}$ might terminate under any sequential schedule starting in C_0. Analogously, let C_0' be the corresponding initial configuration for $\mathcal{A}'$ that extends C_0 by adding $A.\text{act} = \textsc{true}$ and $A.\text{awaken} = \textsc{false}$ for all amoebots A, and let $F_s'(C_0')$ be the set of configurations in which $\mathcal{A}'$ might terminate under any sequential schedule starting in C_0'. We will show that all sequential schedules of $\mathcal{A}'$ starting in C_0' are finite — making $F_s'(C_0')$ well-defined — and that $F_s'(C_0') \subseteq_{\mathcal{A}} F_s(C_0)$, where $\subseteq_{\mathcal{A}}$ denotes containment when restricting configurations to only the variables used in $\mathcal{A}$ (i.e., when ignoring amoebots' activity and awaken bits used in $\mathcal{A}'$).

Lemma 3.3.1. *If every sequential schedule of $\mathcal{A}$ starting in C_0 is finite, then every sequential schedule of $\mathcal{A}'$ starting in C_0' is also finite.*

Proof. Suppose to the contrary that there exists an infinite sequential schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in configuration C_0'. When ignoring the handling of amoebots' activity and awaken bits, any execution of action α' of $\mathcal{A}'$ either makes no change to the system configuration or makes changes identical to those of some action $\alpha \in \mathcal{A}$. First suppose that $\mathcal{S}$ contains an infinite number of executions of α' executing actions of $\mathcal{A}$. Then by constructing a sequential schedule of $\mathcal{A}$ composed of only these $\mathcal{A}$ action executions, we obtain an infinite schedule of $\mathcal{A}$ starting in C_0, a contradiction.

It remains to consider the case that $\mathcal{S}$ contains only a finite number of executions of α' executing actions of $\mathcal{A}$. Since there are only a finite number of such executions, there must exist a time after which no amoebot is $\mathcal{A}$-enabled and the remaining infinite executions of α' only involve updates to amoebots' activity and awaken bits. Any activation of an amoebot A with A.awaken $=$ TRUE results in A setting the activity bits of its neighbors to TRUE — of which there can be at most 8 — and resetting A.awaken to FALSE (Steps 4–6). Otherwise, an activation of A with A.awaken $=$ FALSE must result in A resetting A.act to FALSE since it is not $\mathcal{A}$-enabled (Step 10). Then the potential function $\Phi(C) = \sum_A I_{A.\text{act}} + 9I_{A.\text{awaken}}$ over system configurations C where $I_{A.\text{act}} \in \{0, 1\}$ (resp., $I_{A.\text{awaken}} \in \{0, 1\}$) is equal to 1 if and only if A.act $=$ TRUE (resp., A.awaken $=$ TRUE) is both lower bounded by 0 and strictly decreasing. Therefore, $\mathcal{S}$ can only contain a finite number of executions of α' only involving updates to amoebots' activity and awaken bits, a contradiction of $\mathcal{S}$ being infinite. □

We next establish a key property for characterizing configurations reachable by $\mathcal{A}'$.

Lemma 3.3.2. *Consider any initial configuration C_0 and any sequential schedule $\mathcal{S}$. Then any $\mathcal{A}$-enabled amoebot A in the final configuration reached by $\mathcal{A}'$ under $\mathcal{S}$ starting in C_0' must either (i) be $\mathcal{A}'$-enabled or (ii) have an $\mathcal{A}'$-enabled neighbor B with B.awaken $=$ TRUE.*

Proof. Argue by induction on the length of $\mathcal{S} = ((A_1, \alpha_1), \ldots, (A_k, \alpha_k))$. If $k = 0$, then the lemma trivially holds since all amoebots A initially have A.act $=$ TRUE in C_0' and thus are all $\mathcal{A}'$-enabled. So suppose the lemma holds for schedules of any length $k \geq 0$, and consider any schedule $\mathcal{S}_{k+1} = ((A_1, \alpha_1), \ldots, (A_{k+1}, \alpha_{k+1}))$ of length $k + 1$. Let schedule $\mathcal{S}_k = ((A_1, \alpha_1), \ldots, (A_k, \alpha_k))$ be obtained by removing (A_{k+1}, α_{k+1}) from $\mathcal{S}_{k+1}$. Consider any $\mathcal{A}$-enabled amoebot A at the end of $\mathcal{S}_{k+1}$.

We first suppose that A was also $\mathcal{A}$-enabled at the end of $\mathcal{S}_k$. By the induction hypothesis, there are two cases to consider. If A is $\mathcal{A}'$-enabled at the end of $\mathcal{S}_k$, then the only scenario in which A.act is updated to FALSE is if $A = A_{k+1}$ and A is not $\mathcal{A}$-enabled (Step 10), contrary to our supposition. So A must remain $\mathcal{A}'$-enabled at the end of $\mathcal{S}_{k+1}$, satisfying (i). Otherwise, A must have an $\mathcal{A}'$-enabled neighbor B with B.awaken $=$ TRUE at the end of $\mathcal{S}_k$. The only scenario in which B.awaken is updated to FALSE is if $B = A_{k+1}$ and B sets all of its neighbors' activity bits, including that of A, to TRUE (Steps 4–6). So either B satisfies (ii) by remaining an $\mathcal{A}'$-enabled neighbor with B.awaken $=$ TRUE or A is $\mathcal{A}'$-enabled at the end of $\mathcal{S}_{k+1}$, satisfying (i).

Now suppose that A was not $\mathcal{A}$-enabled at the end of $\mathcal{S}_k$; i.e., the execution of action α_{k+1} by amoebot A_{k+1} causes a change in the neighborhood of A such that A becomes $\mathcal{A}$-enabled by the end of $\mathcal{S}_{k+1}$. Note that because A was not $\mathcal{A}$-enabled at the end of $\mathcal{S}_k$, we must have $A_{k+1} \neq A$. If A and A_{k+1} were neighbors at the end of $\mathcal{S}_k$, then A_{k+1} must update A.act to TRUE during its execution of α_{k+1} (Step 16), satisfying (i). Otherwise, if A and A_{k+1} were not neighbors at the end of $\mathcal{S}_k$, there are still two ways A_{k+1} could change the neighborhood of A by executing α_{k+1}. First, A_{k+1} could move into the neighborhood of A via an EXPAND or PUSH; in this case, A_{k+1} remains $\mathcal{A}'$-enabled and updates its own awaken bit to TRUE (Steps 14 and 24), satisfying (ii). Second, A_{k+1} could update the memory of a neighbor B of A via a WRITE; in this case, A_{k+1} must also update B.act and B.awaken to TRUE (Steps 16 and 17), also satisfying (ii). □

The following lemma concludes our analysis of sequential executions.

Lemma 3.3.3. *For any sequential schedule under which $\mathcal{A}'$ terminates when starting in C_0', the configuration C that $\mathcal{A}'$ terminates in must satisfy $C \in_{\mathcal{A}} F_s(C_0)$.*

Proof. Consider any valid sequential schedule $\mathcal{S}'$ of $\mathcal{A}'$ under which $\mathcal{A}'$ terminates and let C be the configuration it terminates in. By Convention 3.1.1, the executions of action α' in $\mathcal{S}'$ that involve executing actions of $\mathcal{A}$ form a valid sequential schedule $\mathcal{S}$ of $\mathcal{A}$ that makes the same sequence of changes to the system configuration w.r.t. the variables used in $\mathcal{A}$. So $\mathcal{S}$ certainly also reaches C. Moreover, $\mathcal{S}$ must terminate in C; otherwise, there exists an $\mathcal{A}$-enabled amoebot in C that, by Lemma 3.3.2, implies that there exists an $\mathcal{A}'$-enabled amoebot in C. But this contradicts $\mathcal{A}'$ terminating in C, so we conclude $C \in_{\mathcal{A}} F_s(C_0)$. □

Lemmas 3.3.1 and 3.3.3 yield the following corollary.

Corollary 3.3.4. *If every sequential execution of $\mathcal{A}$ starting in C_0 terminates, then every sequential execution of $\mathcal{A}'$ starting in C_0' also terminates. Moreover, $F_s'(C_0') \subseteq_{\mathcal{A}} F_s(C_0)$.*

We now consider the behavior of $\mathcal{A}'$ under asynchronous executions. Recall from Section 2.2.3 that while each amoebot executes at most one action at a time and executes that action's operations sequentially to completion, asynchronous executions allow arbitrarily many amoebots to execute actions simultaneously. We once again define an appropriate abstraction for our analysis. For this abstraction, we assume an ideal wall-clock that can be used to assign precise timing to each event associated with an algorithm's execution; however, we emphasize that this ideal time is only used in this analysis and is unavailable to the amoebots. An *asynchronous schedule* is an assignment of precise timing to every message sending and receipt, variable update in public memory, movement start and end, and operation failure in an asynchronous execution. Thus, in contrast to sequential schedules, we model asynchronous computation at the message passing level to capture all possible ways concurrent operation executions can interleave. In keeping with Sections 2.2.2 and 2.2.3, we make no assumptions on timing other than (i) the delay between every message's sending and receipt as well as every movement's start and end must be positive, and (*ii*) the time taken by every operation execution must be finite. We also assume that at any time while the asynchronous schedule has not yet terminated, there is at least one amoebot executing an action; note that any positive delay where all amoebots are idle could be truncated so that the last action execution's end time coincides with the next action execution's start time without changing the system configuration. In addition to timing, an asynchronous schedule must specify the operations, contents of all messages, and variable values accessed and updated; i.e., all details except private computations.

To ensure that an asynchronous schedule captures the actual system behavior of an amoebot system under an asynchronous adversary, we introduce the concept of validity. An asynchronous schedule is *valid* if there is an asynchronous execution of (enabled) actions producing the same events (timing and content wise) as in the given asynchronous schedule. In the remainder of our analysis, whenever we refer to an asynchronous schedule, we assume its timing is in the control of an adversary constrained by validity unless otherwise specified.

Recall that if C_0 is the initial configuration for $\mathcal{A}$, then C_0' is its extension for $\mathcal{A}'$ with amoebot activity and awaken bits. Let $F_a'(C_0')$ be the set of configurations in which $\mathcal{A}'$ might terminate under any asynchronous schedule starting in C_0'. We will show that all asynchronous schedules of $\mathcal{A}'$ starting in C_0' are finite (i.e., requiring finite time) — making $F_a'(C_0')$ well-defined — and that $F_a'(C_0') \subseteq F_s'(C_0')$. Combined with Corollary 3.3.4, this will yield our desired result. We first identify the points of failure in action α' of $\mathcal{A}'$.

Lemma 3.3.5. *An execution of action α' can only fail in its* Lock *or* Expand *operations.*

Proof. Consider any execution of α' by an amoebot A. We begin with an observation that follows immediately from the design of $\mathcal{A}'$ and Convention 3.1.2: ($*$) whenever an amoebot B is locked by A in an execution of α', only A can initiate a movement with or update the public memory of B.

Recall from Section 2.2.3 that an action execution fails if the execution of any of its operations results in a failure that is not caught by error handling. As α' does not include any error handling, it suffices to show that all operations in α' other than the initial LOCK operation and the possible EXPAND operation are guaranteed to succeed. The first operation A executes is a LOCK of itself and its neighbors which may fail, as claimed. If it fails, then α' immediately aborts, so no further operations are executed.

Hence, suppose that the initial LOCK succeeds, and let $\mathcal{L}_A$ denote the set of amoebots locked by A. Recall from Section 2.2.2 that a READ or WRITE operation by A can only fail if A is accessing a variable in the public memory of an amoebot $B \neq A$ that is disconnected from A at some time in that operation's execution. By ($*$), no amoebot in $\mathcal{L}_A$ can change its shape outside of a movement operation initiated by A. By inspection of α', A only executes READ and WRITE operations involving amoebots in $\mathcal{L}_A$ and does so before its movement operation; thus, they must succeed. Finally, UNLOCK operations cannot fail because they only involve locked amoebots, and CONNECTED operations always succeed.

It remains to consider the movement operations, all of which are determined by the execution of an enabled action $\alpha \in \mathcal{A}$. An EXPAND operation may fail, as claimed. A CONTRACT operation by A only fails if A.shape $\neq$ EXPANDED or A is already involved in a handover. By Convention 3.1.1, this contraction would succeed if all other amoebots were idle, so A must have been expanded when it evaluated the guard of α. Action α' does not contain any operations that change the shape of A between the guard evaluations and this CONTRACT operation, and by ($*$) no other action executions could have involved A in a handover and changed its shape since $A \in \mathcal{L}_A$. So A is still expanded and cannot be involved in a handover when starting this contraction, so the CONTRACT operation succeeds.

A PULL operation by A with a neighbor B only fails if A.shape $\neq$ EXPANDED, B.shape $\neq$ CONTRACTED, A and B are not connected, or A or B is already involved in another handover. By Convention 3.1.1, this pull handover would succeed if all other amoebots were idle, so A must have been expanded, B must have been contracted, and A and B must have been connected when A evaluated the guard of α. Once again, α' does not contain any operations that change the shape of A between the guard evaluations and this PULL operation. Moreover, by $(*)$ neither A nor B can be involved in another handover or could have changed their shape since $A, B \in \mathcal{L}_A$. So A is still expanded, B is still contracted, A and B are still neighbors, and neither A nor B are involved in another handover during this pull handover, so the PULL operation succeeds. An analogous argument holds for PUSH operations.

Therefore, an execution of α' can only fail in its LOCK or EXPAND operations. $\square$

We say that an execution of α' by an amoebot A is *relevant* in an asynchronous schedule of $\mathcal{A}'$ if it is successful and either A.awaken = TRUE or at least one action of $\mathcal{A}$ is enabled in α'. The next two lemmas show that we can *sanitize* any asynchronous schedule of $\mathcal{A}'$ by removing all timed events (i.e., message transmissions, variable updates, movement starts and ends, and operation failures) associated with irrelevant executions of α' — i.e., those that fail or have A.awaken = FALSE and have no action of $\mathcal{A}$ enabled — without changing the system's final configuration.

Lemma 3.3.6. *Let $\mathcal{S}$ be any asynchronous schedule of $\mathcal{A}'$ and let $\mathcal{S}_L$ be the asynchronous schedule obtained from $\mathcal{S}$ by removing all events except those associated with* LOCK *and* UNLOCK *operations and successful movements. Then $\mathcal{S}_L$ is valid w.r.t. its*

Lock *operations. Moreover, for any set S of successful* Lock *operations in $\mathcal{S}_L$, the asynchronous schedule $\mathcal{S}_S$ obtained from $\mathcal{S}_L$ by removing all events associated with* Lock *operations not in S is valid and all* Lock *operations of S are successful and lock the same amoebots they did in $\mathcal{S}_L$.*

We omit the details of this proof as it relies on the message passing implementation of the Lock operation. The full proof can be found in [50].

Lemma 3.3.7. *Let $\mathcal{S}$ be any asynchronous schedule of $\mathcal{A}'$ and let $\mathcal{S}^*$ be its sanitized version keeping only the events associated with relevant executions of α'. Then $\mathcal{S}^*$ is a valid asynchronous schedule that changes the system configuration exactly as $\mathcal{S}$ does except w.r.t. amoebots' activity bits, which have the property that the set of amoebots A with $A.act =$* TRUE *in $\mathcal{S}^*$ is always superset of those in $\mathcal{S}$.*

Proof. Let S be the set of Lock operations executed by relevant action executions in $\mathcal{S}$. By Lemma 3.3.6, when removing all events associated with irrelevant action executions from $\mathcal{S}$ to obtain $\mathcal{S}^*$, all Lock operations of S remain valid and successful in $\mathcal{S}^*$ and lock the same amoebots as they did in $\mathcal{S}$. By Lemma 3.3.5, an execution of α' by an amoebot A is irrelevant if and only if either (i) α' fails in its Lock or Expand operations or (ii) α' is successful in its Lock operation but A.awaken $=$ FALSE and no action in $\mathcal{A}$ is enabled in α'. In both cases, the only change α' can make to the system configuration is updating A.act to FALSE. So the set of amoebots A with A.act $=$ TRUE in $\mathcal{S}^*$ must be a superset of those in $\mathcal{S}$, implying that any relevant action execution of α' stays enabled in $\mathcal{S}^*$.

Since relevant executions of α' only issue Read and Write operations to the executing amoebot or its locked neighbors, the success and identical outcome of all Lock operations in $\mathcal{S}^*$ ensures that all Read and Write operations in $\mathcal{S}^*$ also succeed.

Moreover, because irrelevant executions of α' never perform WRITE operations, all READ and WRITE operations in $\mathcal{S}^*$ must access or update the same variable values as they did in $\mathcal{S}$ since the event timing is preserved. CONNECTED operations in $\mathcal{S}^*$ are also guaranteed to return the same results as in $\mathcal{S}$ since failed EXPAND operations discarded from $\mathcal{S}$ do not change amoebot connectivity.

It remains to show that all movement operations in $\mathcal{S}^*$ are successful. Any CONTRACT, PULL, or PUSH operations in $\mathcal{S}^*$ must have succeeded in $\mathcal{S}$, implying that they were unaffected by any failed EXPAND operations in $\mathcal{S}$ that are now removed. So the only movement operations in $\mathcal{S}^*$ that could have interacted with failed EXPAND operations in $\mathcal{S}$ are concurrent EXPAND operations that contended with failed EXPAND operations for the same nodes. But the fact that these EXPAND operations are in $\mathcal{S}^*$ implies that they succeeded in $\mathcal{S}$, completing their expansion into the contended nodes before all other competitors. This event timing is preserved in $\mathcal{S}^*$ and all contending expansions are removed, so these remain successful. □

Lemma 3.3.7 shows that it suffices to study algorithm $\mathcal{A}'$ under sanitized asynchronous schedules. Our next goal is to show that any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ can be *serialized*; i.e., the executions of action α' in $\mathcal{S}$ (and, by extension, the events therein) can be totally ordered while still producing the same final system configuration as $\mathcal{S}$. Our formalization of asynchronous schedules already totally orders the updates to any single variable in an amoebot's public memory and the occupancy of any single node in G_Δ; here, we focus on ordering entire action executions. Denote the (successful) executions of α' in $\mathcal{S}$ as pairs (A_i, α'_i), where amoebot A_i executes α'_i. Construct a directed graph D with nodes $\{(A_1, \alpha'_1), \ldots, (A_k, \alpha'_k)\}$ representing all executions of α' in $\mathcal{S}$ and directed edges $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ for $i \neq j$ if and only if one of the following hold:

1. Both (A_i, α'_i) and (A_j, α'_j) lock some amoebot B in their Lock operations, and (A_j, α'_j) is the first execution to lock B after B is unlocked by (A_i, α'_i).
2. (A_i, α'_i) executes a movement operation that overlaps in time with the Lock operation in (A_j, α'_j) and involves a node adjacent to A_j.
3. (A_j, α'_j) is the first execution to Expand into some node v after v is vacated by a Contract operation in (A_i, α'_i).

Lemma 3.3.8. *The directed graph D corresponding to the executions of α' in a sanitized asynchronous schedule of $\mathcal{A}'$ is a directed, acyclic graph (DAG).*

Proof. We will show that for any edge $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ in D, (A_i, α'_i) completes its Lock operation (i.e., all locks in $A_i.N_\alpha$ are set) before (A_j, α'_j) does. This immediately implies that D is acyclic; otherwise, the Lock operations of any two executions in a cycle of D must complete both before and after each other, a contradiction.

First suppose that $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ is an edge in D because both executions lock an amoebot B and (A_j, α'_j) is the first execution to lock B after B is unlocked by (A_i, α'_i). Clearly, A_j can only lock B after A_i has unlocked B and A_i can only unlock B after it locks B in its own Lock operation. Since these operations all involve message transfers requiring positive time, (A_i, α'_i) must complete its Lock operation before (A_j, α'_j) does.

Next suppose that $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ is an edge in D because (A_i, α'_i) executes a movement operation that overlaps in time with the Lock operation in (A_j, α'_j) and involves a node adjacent to A_j. Any movement operation in (A_i, α'_i) must start after its Lock operation completes; thus, (A_i, α'_i) must complete its Lock operation before (A_j, α'_j) does.

Finally, suppose that $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ is an edge in D because (A_j, α'_j) is the first execution to Expand into some node v after v is vacated by a Contract operation in (A_i, α'_i). It suffices to consider the case where the LOCK operation of (A_i, α'_i) does not lock A_j; otherwise, there exists a directed path of lock-based edges from (A_i, α'_i) to (A_j, α'_j) in D and the first case proves the claim. For (A_i, α'_i) to not lock A_j, A_j cannot be a neighbor of A_i at the time the LOCK operation of (A_i, α'_i) starts. We know that the LOCK operation of (A_i, α'_i) is successful, so A_i is locked and occupies v until the start of its CONTRACT operation out of v. But A_j must occupy a node adjacent to v at the start of (A_j, α'_j) and must succeed in its own LOCK operation in order to EXPAND into v. Thus, the LOCK operation of (A_j, α'_j) cannot complete until after A_i has started contracting out of v, which occurs strictly after the completion of the LOCK operation of (A_i, α'_i). □

Now we are ready to prove the following lemma.

Lemma 3.3.9. *Consider any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ and let (A_i, α'_i) be any sink in the corresponding DAG D. Let $\mathcal{S}_i^-$ be the asynchronous schedule obtained by removing all events associated with (A_i, α'_i) from $\mathcal{S}$. Then $\mathcal{S}_i^-$ is valid and the final configuration of $\mathcal{S}_i^-$ is expansion-compatible with (A_i, α'_i) and identical to that of $\mathcal{S}$ except for the amoebots locked by (A_i, α'_i) in $\mathcal{S}$, which appear exactly as they did just after the* LOCK *operation of (A_i, α'_i) completed in $\mathcal{S}$.*

Proof. Consider any action execution (A_j, α'_j) with $j \neq i$. We first show that (A_j, α'_j) must remain enabled in $\mathcal{S}_i^-$; i.e., A_j.act = TRUE at the time of this execution. This must have been the case in $\mathcal{S}$, so the only way for (A_j, α'_j) to not be enabled in $\mathcal{S}_i^-$ is if (A_i, α'_i) was responsible for enabling it in $\mathcal{S}$. But (A_i, α'_i) could only have updated A_j.act to TRUE if (A_i, α'_i) locked A_j, implying that (A_j, α'_j) could not have started until after (A_i, α'_i) unlocked A_j. Thus, there must exist a directed path in D from (A_i, α'_i) to (A_j, α'_j), contradicting the assumption that (A_i, α'_i) is a sink.

The remainder of this proof establishes that (A_j, α'_j) remains valid in $\mathcal{S}_i^-$. Let $\mathcal{L}_j(\mathcal{S})$ (resp., $\mathcal{L}_j(\mathcal{S}_i^-)$) denote the set of amoebots locked by (A_j, α'_j) in $\mathcal{S}$ (resp., in $\mathcal{S}_i^-$); we begin by showing $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$. Suppose that there is a directed path in D from (A_j, α'_j) to (A_i, α'_i). By the proof of Lemma 3.3.8, (A_j, α'_j) must complete its Lock operation before (A_i, α'_i) does, implying that (A_j, α'_j) completes its Lock operation before (A_i, α'_i) completes any operation. Since the timing of (A_j, α'_j) in $\mathcal{S}$ is preserved in $\mathcal{S}_i^-$, it follows that $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$. Now suppose that there is no directed path in D from (A_j, α'_j) to (A_i, α'_i). Then the amoebots locked by (A_j, α'_j) and (A_i, α'_i) in $\mathcal{S}$ are disjoint, so certainly (A_j, α'_j) can lock any amoebot in $\mathcal{S}_i^-$ that it did in $\mathcal{S}$; i.e., $\mathcal{L}_j(\mathcal{S}) \subseteq \mathcal{L}_j(\mathcal{S}_i^-)$. But suppose to the contrary that (A_j, α'_j) is able to lock some additional amoebot B in $\mathcal{S}_i^-$ that it did not lock in $\mathcal{S}$. This is only possible if (A_i, α'_i) caused B to move out of the neighborhood of A_j in $\mathcal{S}$. But then $(A_i, \alpha'_i) \rightarrow (A_j, \alpha'_j)$ must be a directed edge in D, contradicting the assumption that (A_i, α'_i) is a sink. Therefore, in all cases, (A_j, α'_j) locks the same set of amoebots in $\mathcal{S}$ and $\mathcal{S}_i^-$.

After completing its Lock operation, (A_j, α'_j) does one of two things. If A_j.awaken = TRUE, then it updates the activity bits of all the amoebots it locked to TRUE, updates its own awaken bit to FALSE, releases its locks, and aborts. Since $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$ and timing is preserved, these updates occur identically in $\mathcal{S}$ and $\mathcal{S}_i^-$. Otherwise, if A_k.awaken = FALSE, A_j evaluates the guards of actions in the original algorithm $\mathcal{A}$ which depend only on the positions, shapes, and public memories of the locked amoebots. Suppose to the contrary that there is an amoebot B locked by A_j whose position, shape, or public memory is different in $\mathcal{S}_i^-$ than it was in $\mathcal{S}$. Then (A_i, α'_i) must have locked B to perform the corresponding update in $\mathcal{S}$, implying that there is a directed path from (A_i, α'_i) to (A_j, α'_j) in D, contradicting the assumption that (A_i, α'_i) is a sink. So the outcomes of the guard evaluations must be identical in $\mathcal{S}$ and $\mathcal{S}_i^-$.

Since (A_j, α'_j) is in the sanitized schedule $\mathcal{S}$, it must be relevant, and thus there must exist an enabled action of $\mathcal{A}$. Whatever enabled action of $\mathcal{A}$ is chosen to perform, any WRITE, CONTRACT, PULL, or PUSH operations involved must occur identically in $\mathcal{S}$ and $\mathcal{S}_i^-$ since the locked amoebots and their positions, shapes, and public memories are the same in both schedules. The only remaining possibility is that (A_j, α'_j) causes A_j to EXPAND into an adjacent node v in $\mathcal{S}$ that is occupied in $\mathcal{S}_i^-$, causing the EXPAND operation to fail in $\mathcal{S}_i^-$. This implies that (A_i, α'_i) causes A_i to CONTRACT out of v. But then $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ must be a directed edge in D, contradicting the assumption that (A_i, α'_i) is a sink. Therefore, we conclude that $\mathcal{S}_i^-$ is valid and all action executions (A_j, α'_j) for which $j \neq i$ execute identically in $\mathcal{S}$ and $\mathcal{S}_i^-$.

Next, we show that the final configuration of $\mathcal{S}_i^-$ is expansion-compatible with (A_i, α'_i). Suppose to the contrary that (A_i, α'_i) performs a successful EXPAND operation into a node v in $\mathcal{S}$ but v is occupied by another amoebot at the end of $\mathcal{S}_i^-$. Since all executions other than (A_i, α'_i) are valid and execute identically in $\mathcal{S}$ and $\mathcal{S}_i^-$, another amoebot can only have come to occupy v at the end of $\mathcal{S}_i^-$ if A_i vacated v in some later execution in $\mathcal{S}$. But A_i can only change its shape if it is locked, contradicting the assumption that (A_i, α'_i) is a sink in D. So v must be unoccupied at the end of $\mathcal{S}_i^-$ and thus it is expansion-compatible with (A_i, α'_i).

It remains to show that the amoebots in $\mathcal{L}_i(\mathcal{S})$ — i.e., those locked by (A_i, α'_i) in $\mathcal{S}$ — appear in $\mathcal{S}_i^-$ exactly as they did after the LOCK operation of (A_i, α'_i) in $\mathcal{S}$. But this follows immediately from the assumption that (A_i, α'_i) is a sink: for an execution (A_j, α'_j) with $j \neq i$ to change the position, shape, or public memory of an amoebot $B \in \mathcal{L}_i(\mathcal{S})$, it would first have to lock B, implying that $(A_i, \alpha_i') \to (A_j, \alpha_j')$ is a directed edge in D. □

Lemma 3.3.9 and Convention 3.1.3 allow us to prove the following central result.

Lemma 3.3.10. *Any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ can be serialized; i.e., there exists a sequential ordering of its executions of α' that reaches a final configuration that is identical to that of $\mathcal{S}$ with the exception that the set of amoebots A with $A.act =$ TRUE or $A.awaken =$ TRUE is a superset of those in the final configuration reached by $\mathcal{S}$.*

Proof. Consider any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ and let D be its corresponding DAG (Lemma 3.3.8). We argue by induction on k, the number of executions of α' in $\mathcal{S}$, that any sequential ordering of the executions of α' in $\mathcal{S}$ consistent with a topological ordering of D satisfies the lemma.

The lemma trivially holds for $k = 1$, so suppose the lemma holds for any sanitized asynchronous schedule of $\mathcal{A}'$ with $k \geq 1$ executions of α'. Let $\mathcal{S}$ be any sanitized asynchronous schedule of $\mathcal{A}'$ consisting of $k + 1$ executions of α' and let (A_i, α_i') be any sink in the corresponding DAG D. By Lemma 3.3.9, the sanitized asynchronous schedule $\mathcal{S}_i^-$ obtained by removing all events associated with (A_i, α_i') from $\mathcal{S}$ is valid and reaches a final configuration that is expansion-compatible with (A_i, α_i') and is identical to that of $\mathcal{S}$ except for the amoebots locked by (A_i, α_i') in $\mathcal{S}$, which appear exactly as they did just after the LOCK operation of (A_i, α_i') completed in $\mathcal{S}$. By the induction hypothesis, there exists a sequential schedule $\bar{\mathcal{S}}_i$ that reaches a final configuration identical to that of $\mathcal{S}_i^-$ with the exception that the set of amoebots A with A.act $=$ TRUE or A.awaken $=$ TRUE is a superset of those in the final configuration reached by $\mathcal{S}_i^-$. This implies that (A_i, α_i') is enabled in the final configuration reached by $\bar{\mathcal{S}}_i$ since it was also enabled in that of $\mathcal{S}_i^-$.

The amoebots $\mathcal{L}_i(\mathcal{S})$ locked by (A_i, α_i') in $\mathcal{S}$ must still be neighbors of A_i at the end of $\bar{\mathcal{S}}_i$ by Lemma 3.3.9 and the induction hypothesis, but A_i may also have additional neighbors at the end of $\bar{\mathcal{S}}_i$ that were not originally present at the time of its LOCK operation in $\mathcal{S}$. Thus, we have $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$. There are three cases for the behavior of (A_i, α_i'); in each, we show that we can produce a sequential schedule $\bar{\mathcal{S}}$ combining $\bar{\mathcal{S}}_i$ and (A_i, α_i') whose final configuration satisfies the lemma.

Case 1. A_i.awaken = TRUE in both $\mathcal{S}$ and $\bar{\mathcal{S}}_i$. Let $\bar{\mathcal{S}}$ be the sequential schedule obtained by appending (A_i, α_i') to the end of $\bar{\mathcal{S}}_i$. In both $\mathcal{S}$ and $\bar{\mathcal{S}}_i$, (A_i, α_i') updates B.act to TRUE for all amoebots B that it locks, updates A_i.awaken to FALSE, releases its locks, and aborts. Since $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$, the only difference between the final configurations of $\mathcal{S}$ and $\bar{\mathcal{S}}$ is that the latter may have additional amoebots with their activity or awaken bits set to TRUE, so the lemma holds.

Case 2. A_i.awaken = FALSE in $\mathcal{S}$ but A_i.awaken = TRUE in $\bar{\mathcal{S}}_i$. Let $\bar{\mathcal{S}}$ be the sequential schedule obtained by activating A_i twice at the end of $\bar{\mathcal{S}}_i$. The first activation has the same effect as Case 1, potentially yielding more amoebots with their activity or awaken bits set to TRUE. It also resets the awaken bit of A_i, yielding A_i.awaken = FALSE in both $\mathcal{S}$ and $\bar{\mathcal{S}}_i + (A_i, \alpha_i')$. We address this in the following case.

Case 3. A_i.awaken = FALSE in both $\mathcal{S}$ and $\bar{\mathcal{S}}_i$. Let $\bar{\mathcal{S}}$ be the sequential schedule obtained by appending (A_i, α_i') to the end of $\bar{\mathcal{S}}_i$. Since (A_i, α_i') is an execution of $\mathcal{S}$, a sanitized schedule, we know that (A_i, α_i') is relevant and thus must have an enabled action $\alpha \in \mathcal{A}$ in $\mathcal{S}$. However, because the amoebots it locks in $\mathcal{S}$ and $\bar{\mathcal{S}}_i$ are not necessarily the same — indeed, we only know that $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$ — it is not immediately obvious that α is still enabled in $\bar{\mathcal{S}}_i$ or that it would make the same changes to the system configuration as it did in $\mathcal{S}$. But recall that by Convention 3.1.3, all actions of algorithm $\mathcal{A}$ are assumed to be monotonic. The remainder of this proof will show that the differences between the local configurations of A_i in $\mathcal{S}$ and $\bar{\mathcal{S}}_i$ are exactly those addressed by monotonicity, allowing us to use its guarantees when comparing the final configurations reached by $\mathcal{S}$ and $\bar{\mathcal{S}}$.

Let c be the local configuration of A_i with neighbors $\mathcal{L}_i(\mathcal{S})$ and A_i appearing as they did when the LOCK operation of (A_i, α_i') completed in $\mathcal{S}$ and let $\bar{c}$ be the local configuration of A_i at the end of $\bar{\mathcal{S}}_i$. Let $\alpha \in \mathcal{A}$ be the enabled action executed in (A_i, α_i') in $\mathcal{S}$, and let c_α be the local configuration of A_i reached by the execution of α on the locked neighborhood $\mathcal{L}_i(\mathcal{S})$. Since $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$, we know that c and $\bar{c}$ agree on $\mathcal{L}_i(\mathcal{S})$, making $\bar{c}$ an extension of c. Moreover, $\bar{c}$ is clearly reachable by the sequential schedule of $\mathcal{A}$ contained within $\bar{\mathcal{S}}_i$ and, by Lemma 3.3.9 and the induction hypothesis, is expansion-compatible with the execution reaching c_α from c. Thus, because α is monotonic by Convention 3.1.3, we conclude that α is also enabled for $\bar{c}$ and there is an execution of α on $\bar{c}$ that makes the same move (if any) and performs the same updates to the amoebots of $\mathcal{L}_i(\mathcal{S})$ as it did to reach c_α from c; moreover, it performs no updates to the amoebots of $\mathcal{L}_i(\bar{\mathcal{S}}_i) \setminus \mathcal{L}_i(\mathcal{S})$. The only other change (A_i, α_i') makes in reaching the final configuration of $\bar{\mathcal{S}}$ from the end of $\bar{\mathcal{S}}_i$ is updating the activity bits of all amoebots that it locked to TRUE, but as we saw in the previous case, this can only cause additional amoebots to have their activity or awaken bits set to TRUE since $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$. Therefore, in all cases, the lemma holds. □

It remains to show that all asynchronous schedules of $\mathcal{A}'$ are finite in a sense that they only require a finite amount of time.

Lemma 3.3.11. *If every sequential schedule of $\mathcal{A}'$ starting in C_0' is finite, then every asynchronous schedule of $\mathcal{A}'$ starting in C_0' is also finite.*

Proof. Suppose to the contrary that $\mathcal{S}$ is an infinite asynchronous schedule of $\mathcal{A}'$ starting in C_0'. If $\mathcal{S}$ contains an infinite sanitized asynchronous schedule, then — since every execution of a relevant action requires only finite time — $\mathcal{S}$ must consist of an infinite number of relevant action executions. Let D be the corresponding DAG (Lemma 3.3.8) with an infinite number of nodes. By supposition, there exists a finite upper limit f on the number of action executions in any sequential schedule of $\mathcal{A}'$ starting in C_0'. By iteratively applying Lemma 3.3.9, there must exist a sub-DAG $D' \subseteq D$ on $f + 1$ nodes corresponding to a valid sanitized asynchronous schedule $\mathcal{S}'$ obtained from $\mathcal{S}$ by removing all events associated with action executions that are nodes in D but not in D'. By Lemma 3.3.10, $\mathcal{S}'$ can be serialized to obtain a sequential schedule of $\mathcal{A}'$ starting in C_0' and consisting of the same $f + 1$ action executions, contradicting the fact that f is the upper limit on the number of action executions contained in any sequential schedule of $\mathcal{A}'$ starting in C_0'.

So instead suppose that $\mathcal{S}$ only contains finite sanitized asynchronous schedules. Then there must exist a finite time later than any event associated with a relevant action execution. Let t be the earliest such time; observe that for $\mathcal{S}$ to be infinite, there must be an infinite number of irrelevant action executions after t. Recall that an execution of α' is irrelevant if it fails or if it determines that no actions of $\mathcal{A}$ are enabled, causing it to set its activity bit to FALSE. By Lemma 3.3.5, executions of α' can only fail in their LOCK or EXPAND operations.

We first show that there exists a finite time $t' \geq t$ such that after time t', no execution of α' in $\mathcal{S}$ can fail in its LOCK operation because an $\mathcal{A}$-disabled execution succeeds in its own LOCK operation. Suppose to the contrary that there exists an infinite number of pairs $(A, \alpha'), (B, \alpha')$ such that (A, α') fails in its LOCK operation because (B, α') succeeded in its own. Because B is $\mathcal{A}$-disabled, it completes its execution (B, α') by resetting B.act to FALSE and releasing its locks. No irrelevant execution can ever set an amoebot's activity bit to TRUE because the corresponding WRITE operations occur after the points of failure in α' and do not occur for those that are $\mathcal{A}$-disabled. Thus, the number of enabled $\mathcal{A}$-disabled executions is monotonically decreasing and there are only a finite number of amoebots that may have them, so there cannot be an infinite number of these pairs, a contradiction.

So let $t' \geq t$ be the earliest time after which no execution of α' can fail in its LOCK operation due to an $\mathcal{A}$-disabled execution, and let (A, α') be the earliest execution (or, in the case of a tie, any of the earliest) in $\mathcal{S}$ that starts after time t' and results in failure. First suppose that (A, α') fails in its LOCK operation. Then by Theorem 2.2.1, there exists an amoebot B in the 3-neighborhood of A that succeeds in the LOCK operation of its own execution, say (B, α'). Since all relevant executions complete and release their locks before time t and since (A, α') starts after time $t' \geq t$, (B, α') must also result in failure. Execution (B, α') has already succeeded in its LOCK operation, so by Lemma 3.3.5 it must fail in its EXPAND operation, say, into an adjacent node v. Convention 3.1.1 ensures that B could not have called EXPAND if it was expanded or if v was occupied at the time of the corresponding guard evaluation, and B cannot be involved in a movement initiated by some other amoebot because it is locked. So the only remaining way for the EXPAND operation of (B, α') to fail is if another amoebot C successfully moves into v during an execution (C, α') that is concurrent with (B, α'). But if (C, α') succeeds in its movement operation, then the entire execution must be successful by Lemma 3.3.5; therefore, (C, α') is a successful execution that completes after time t, a contradiction. Therefore, in all cases, $\mathcal{S}$ cannot be infinite. □

Recall that if C_0 is the initial configuration for $\mathcal{A}$, then C_0' is its extension for $\mathcal{A}'$ with amoebot activity and awaken bits. Recall also that $F_s(C_0)$ is the set of configurations in which $\mathcal{A}$ might terminate under any sequential schedule starting in C_0 and $F'_a(C'_0)$ is the set of configurations in which $\mathcal{A}'$ might terminate under any asynchronous schedule starting in C'_0. Combining Corollary 3.3.4 with Lemmas 3.3.11, we obtain the following theorem.

Theorem 3.3.12. *Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 3.1.1–3.1.3 and $\mathcal{A}'$ be the amoebot algorithm produced from $\mathcal{A}$ by the concurrency control protocol (Algorithm 6). If every sequential execution of $\mathcal{A}$ starting in C_0 terminates, then every asynchronous schedule of $\mathcal{A}'$ starting in C'_0 also terminates. Moreover, $F'_a(C'_0) \subseteq_{\mathcal{A}} F_s(C_0)$.*

3.4 Discussion and Open Problems

The goal of our concurrency control protocol is to combine the convenience of sequential algorithm design with the realism of asynchronous executions. It is our hope that designing a correct sequential algorithm and then proving it satisfies the conditions of Theorem 3.3.12 to automatically obtain a correct asynchronous algorithm is easier than designing a correct asynchronous algorithm directly. Here, we discuss the degree to which our protocol achieves this goal and the degree of difficulty involved in showing a sequential algorithm is compatible with our protocol.

Theorem 3.3.12 provides a set of sufficient conditions for which a correct sequential algorithm $\mathcal{A}$ can be transformed into a correct asynchronous algorithm $\mathcal{A}'$. First, for any initial configuration C_0 that $\mathcal{A}$ may start in, $\mathcal{A}$ must be guaranteed to terminate under any unfair sequential adversary. This immediately renders algorithms that never terminate — such as the stochastic algorithms of Chapters 8–10 — incompatible with the protocol; thus, we instead focus on algorithms that do terminate. Unfair adversaries are the most powerful with respect to the fairness assumptions (see Table 1), since an adversary is never forced to activate an amoebot unless it is the only one in the system with an enabled action. Despite this challenge, there is some evidence indicating that termination under an unfair sequential adversary is not very restrictive. We know that the set of algorithms satisfying this condition is non-empty as it contains the Hexagon-Formation algorithm presented in Section 2.5. Moreover, it is likely that at least some of the existing algorithms in the amoebot literature would also satisfy this condition once translated from their current formulations into action semantics. Many of these algorithms are proven correct under a sequential adversary that activates all amoebots (enabled or not) infinitely often; one would only need to show that when these algorithms are formulated in terms of actions, the set of enabled amoebots shrinks such that an unfair adversary would be forced to activate any amoebot that would otherwise stall progress.

Second, $\mathcal{A}$ must satisfy Conventions 3.1.1–3.1.3. The validity convention (Convention 3.1.1) is trivially satisfied by any algorithm that is correct under an unfair sequential adversary. The phased action structure (Convention 3.1.2) that forces any movement operation to occur last and prohibits the use of Lock or Unlock operations is also relatively easy to satisfy. Moving last seems to be a common design paradigm in the literature. Additionally, the Lock and Unlock operations were only introduced in the canonical amoebot model (Section 2.2), so none of the existing algorithms use them.

The monotonicity convention (Convention 3.1.3), however, is significantly more restrictive. The serializability argument critically relies on it to show that when an action execution is removed from its place in an asynchronous schedule and moved into the future so that it is not concurrent with any other execution, it makes exactly the same changes to the system configuration that it did originally, regardless of any new amoebots that may have moved into its neighborhood in the meantime. In that light, it is easy to see that *static* algorithms that do not use movement trivially satisfy monotonicity. These include many of the existing algorithms for leader election [21, 54, 60, 66, 92] and the Energy-Sharing algorithm presented in the next chapter (Chapter 4, [51]). However, the majority of interesting collective behaviors for programmable matter require movement, and it is not clear if any of these algorithms satisfy monotonicity. In particular, even the relatively simple Hexagon-Formation algorithm of Section 2.5 which can be directly proven to be correct under any unfair asynchronous adversary is not monotonic.

Therefore, the monotonicity convention appears to be the convention that most severely limits the applicability of our concurrency control protocol. This highlights two critical open questions. Do there exist algorithms that are not correct under an asynchronous adversary but are compatible with our concurrency control protocol? Are there other, less restrictive sufficient conditions for correctness in spite of asynchrony? We are hopeful that this first step towards generalized concurrency control for programmable matter and answers to these open problems will advance the analysis of existing and future algorithms for programmable matter in the concurrent setting.

Chapter 4

ENERGY DISTRIBUTION

The composing modules of active programmable matter are often envisioned and designed to be simple, homogeneous units capable of internal computation, inter-module communication, and movement. These modules require a constant supply of energy to function, but as the number of modules per collective increases and individual modules are miniaturized from the centimeter/millimeter-scale [94, 100, 160] to the micro- and nano-scale [72, 124], traditional methods of robotic power supply such as internal battery storage and tethering become infeasible.

Programmable matter systems instead make use of an *external energy source* accessible by at least one module and rely on *module-to-module power transfer* to supply the system with energy [30, 94, 99, 160]. This external energy can be supplied directly to one or more modules in the form of electricity, as in [94], or may be ambiently available as light, heat, sound, or chemical energy in the environment [136, 152]. Since energy may not be uniformly accessible to all modules in the system, a strategy for *energy distribution* — or sharing energy between modules such that all modules eventually obtain the energy they need to function — is imperative but does not come for free. Significant energy loss can occur in module-to-module transfer depending on the method used, and even with perfect transfer successive voltage drops between modules can limit the number of modules that can be powered from a single source [94]. Module geometry may further complicate the problem by introducing short circuits, adding further constraints to power routing algorithms [30].

Algorithmic theory for programmable matter has largely ignored the role of energy (with notable exceptions, such as [72, 160]), focusing primarily on characterizing the minimal capabilities individual modules need to collectively achieve desired system-level self-organizing behaviors. Across models of active programmable matter — including population protocols [10], the nubot model [194], mobile robots [87], hybrid programmable matter [96, 97], and the amoebot model [52, 59] — most works either develop algorithms for a desired behavior and bound their time complexity or, on the negative side, prove that a given behavior cannot be achieved within the given constraints. To the extent of our knowledge, papers on these models have only mentioned energy to justify constraints (e.g., why a system should remain connected [144]) and have never directly treated the impact of energy usage and distribution on an algorithm's efficiency. In contrast, both programmable matter practitioners and the modular and swarm robotics literature view energy constraints as influential aspects of algorithm design [15, 120, 151, 159, 191].

In this chapter, we present an algorithm for energy distribution in the amoebot model [51] that is loosely inspired by the growth behavior of *Bacillus subtilis* bacterial biofilms [133, 163]. We assume that all amoebots in the system require energy to perform their actions but only some have access to an external energy source. Naive distribution strategies such as fully selfish or fully altruistic behaviors have obvious problems: in the former, amoebots with access to energy use it all and starve the others, while in the latter no amoebot ever knows when it is safe to use its stored energy. This necessitates a strategy in which amoebots shift between selfish and altruistic energy usage depending on the needs of their neighbors. Our algorithm mimics the way bacteria use long-range communication of their metabolic stress to temporarily inhibit the biofilm's energy consumption, allowing for nutrients to reach starving bacteria and effectively solving the energy distribution problem.

The remainder of this chapter is organized as follows. We begin with some details on the bacterial biofilms that inspired our approach in Section 4.1, followed by a formal statement of the energy distribution problem in Section 4.2. In Section 4.3, we present Energy-Sharing: a local, distributed algorithm that solves the energy distribution problem in $\mathcal{O}(n)$ sequential rounds (Theorem 4.3.8), where n is the number of amoebots in the system. This algorithm is asymptotically optimal when the number of external energy sources is fixed (Theorem 4.3.9). We then show simulation results in Section 4.4, demonstrating that without the biofilm-inspired communication of amoebots' energy states, Energy-Sharing fails to distribute sufficient energy throughout the system.

In Section 4.5, we consider the impact of crash faults on the correctness and runtime of our algorithm. Our fault mitigation strategy relies on a new algorithmic primitive called Forest-Prune-Repair that locally repairs the system's underlying communication structure after an amoebot crashes. This repair primitive is in fact of independent interest, as it extends the amoebot model's well-established *spanning forest primitive* [52] to be self-stabilizing in the presence of crash failures. Finally, we show how Forest-Prune-Repair can be used to compose other amoebot algorithms with our Energy-Sharing algorithm. This effectively generalizes all previous work on the amoebot model to also consider energy constraints.

4.1 Biological Inspiration

Our strategy of shifting between selfish and altruistic energy usage to achieve energy distribution is loosely inspired by the work of Liu and Prindle et al. [133, 163] on the growth behavior of colonies of *Bacillus subtilis* bacteria, which we summarize here for the sake of completeness. These bacteria form densely packed *biofilm colonies* when they become metabolically stressed (i.e., when they become nutrient scarce and begin to starve). These bacteria consume *glutamine*, which is produced from a combination of substrates *glutamate* and *ammonium*. Glutamate is sourced from the environment outside of the biofilm, whereas ammonium is produced by individual bacterium. However, because ammonium can freely diffuse across a bacterium's cell membrane and be lost to its surroundings, production of ammonium is known as the *futile cycle*. The futile cycle is detrimental for bacteria on the biofilm's periphery, as they lose all their ammonium to the external medium. Once a biofilm colony is formed, however, bacteria in the biofilm's interior are shielded from the futile cycle by those on the periphery. This creates a symbiotic co-dependence: bacteria in the interior are reliant on glutamate passed from the periphery, while bacteria on the periphery are reliant on ammonium produced by the interior.

As the biofilm grows, overall glutamate consumption in the periphery increases, limiting the amount of glutamate that permeates into the interior of the colony. This causes interior bacteria to become metabolically stressed. Thus, in order to regulate glutamate consumption on the periphery, interior bacteria communicate their metabolic states to the peripheral bacteria via a long-range electrochemical process known as *potassium ion-channel-mediated signaling* [163]. This sudden influx of potassium inhibits a bacterium's glutamate intake and ammonium retention, allowing more nutrients to pass into the biofilm's interior. As a result, the biofilm grows at an oscillating rate rather than a constant one, despite the fact that there is plentiful glutamate in the environment. This emergent oscillation enables continuous distribution of nutrients throughout the colony, effectively solving the energy distribution problem.

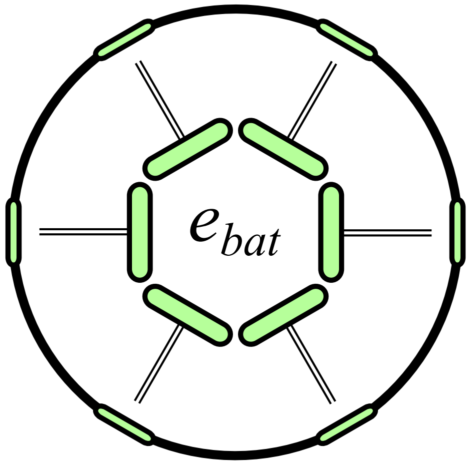


Figure 8. Amoebot Energy Anatomy. Energy is transferred between amoebots at their contact points, shown as green markers on the amoebot's periphery. An amoebot's battery e_{bat} stores energy for its own use and for sharing with its neighbors.

4.2 The Energy Distribution Problem

For this chapter, we assume the simplified sequential amoebot model in which at most one amoebot is active at a time and the adversary activates every amoebot infinitely often. We further assume geometric space, assorted orientations, and constant-size memory (see Table 1). Here, we introduce terminology specific to the problem of energy distribution. Each amoebot A has an *energy battery* denoted $A.e_{bat}$ with constant capacity $\kappa > 0$ (see Figure 8). The battery represents stored energy A can use for performing actions or for sharing with its neighbors. Amoebots with access to an external energy source can harvest energy into their batteries directly, while those that do not depend on their neighbors to share with them. In either case, each amoebot can transfer at most a constant $\alpha > 0$ units of energy per activation.

An instance of the *energy distribution problem* has the form $(\mathcal{S}, \kappa, \delta)$ where $\mathcal{S}$ is a finite connected amoebot system, κ is the capacity of each amoebot's battery, and energy demand $\delta(A, i)$ denotes the energy cost for an amoebot A to perform its i-th action. For convenience, we will use $\delta(A)$ to refer to the energy cost for A to perform its next action. An instance is *valid* if (i) $\mathcal{S}$ contains one or more "root" amoebots with access to external energy sources and all non-root amoebots are initially "idle" and (ii) for all amoebot actions, $\delta(\cdot, \cdot) \leq \kappa$; i.e., no energy demand exceeds the batteries' energy capacity. An amoebot A is *stressed* if the energy level of its battery is strictly less than the demand for its next action, i.e., if $A.e_{bat} < \delta(A)$. An action a of an amoebot A is *enabled* if, barring any energy considerations, A is able to perform action a. A local, distributed algorithm $\mathcal{A}$ *solves* a valid instance of the energy distribution problem in time t if, when each amoebot executes $\mathcal{A}$ individually, no amoebot remains stressed for more than t sequential rounds and at least one amoebot performs an enabled action every t sequential rounds.

4.3 The Energy-Sharing Algorithm

In this section, we present algorithm Energy-Sharing for energy distribution under the amoebot model. At a high level, this algorithm works as follows. After some initial setup, each amoebot continuously loops through a sequence of three phases: the communication phase, the sharing phase, and the usage phase. In the *communication phase*, amoebots propagate signals to communicate the energy states of stressed amoebots, analogous to the long-range electrochemical signaling via potassium ion channels in the biofilms. Amoebots then attempt to harvest energy from an external energy source or transfer energy to their neighbors in the *sharing phase*. Finally, amoebots spend their stored energy to perform actions according to their collective behavior in the *usage phase*. Note that the system is not synchronized and each amoebot progresses through these phases independently. Section 4.3.1 details the

Parameter	**Notation**	**Constraints**
Battery Capacity	$\kappa \in \mathbb{R}$	$\kappa > 0$
Energy Demand	$\delta : \mathcal{S} \times \mathbb{Z}_+ \to \mathbb{R}$	$\delta(\cdot, \cdot) \leq \kappa$
Transfer Rate	$\alpha \in \mathbb{R}$	$\alpha > 0$

Table 4. Parameters for ENERGY-SHARING.

Variable	**Notation**	**Domain**	**Initialization**
Battery Energy	e_{bat}	$[0, \kappa]$	0
Parent Pointer	parent	$\{\text{NULL}, 0, \ldots, 5\}$	NULL
Stress Flag	stress	$\{\text{TRUE}, \text{FALSE}\}$	FALSE
Inhibit Flag	inhibit	$\{\text{TRUE}, \text{FALSE}\}$	FALSE
Prune Flag	prune	$\{\text{TRUE}, \text{FALSE}\}$	FALSE

Table 5. Local Variables for ENERGY-SHARING.

setup and phases of ENERGY-SHARING. We then analyze this algorithm's correctness and runtime in Section 4.3.2.

4.3.1 Algorithm Description

Tables 4 and 5 list the parameters and local variables used by the ENERGY-SHARING algorithm. Complete distributed pseudocode is given in Algorithm 7.

The Setup Phase. Recall that amoebot system $\mathcal{S}$ is connected. Amoebots with access to an external energy source are roots, and the rest are idle. This phase organizes $\mathcal{S}$ as a spanning forest $\mathcal{F}$ of trees rooted at the root amoebots. These trees facilitate an analogy to the potassium ion signaling that the bacteria use to communicate when they are metabolically stressed (discussed further in the communication phase). To form $\mathcal{F}$, we make use of the well-established *spanning forest primitive* [52]. If an amoebot A is idle, it checks if it has a root or active neighbor B. If so, A becomes active and

Algorithm 7 Energy-Sharing for Amoebot A

```
if $A$ is idle then
    if $A$ has a neighbor $B$ that is a root or is active then
        $A$ becomes active.
        $A$.parent $\leftarrow B$.
else (i.e., $A$ is active or a root)
    Communicate( )
    ShareEnergy( )
    UseEnergy( )
function Communicate( )
    if $A$ is active then
        if $A.e_{bat} < \delta(A) \vee$ ($A$ has a child $B$ with $B$.stress = TRUE) then
            $A$.stress $\leftarrow$ TRUE.
        else $A$.stress $\leftarrow$ FALSE.
        $A$.inhibit $\leftarrow A$.parent.inhibit.
    else (i.e., $A$ is a root)
        if $A.e_{bat} < \delta(A) \vee$ ($A$ has a child $B$ with $B$.stress = TRUE) then
            $A$.inhibit $\leftarrow$ TRUE.
        else $A$.inhibit $\leftarrow$ FALSE.
function ShareEnergy( )
    if $A$ is a root then $A.e_{bat} \leftarrow \min\{A.e_{bat} + \alpha, \kappa\}$.
    if $A.e_{bat} \geq \alpha$ and $A$ has a child $B$ with $B.e_{bat} < \kappa$ then
        Choose an arbitrary child $B$ with $B.e_{bat} < \kappa$.
        $A.e_{bat} \leftarrow A.e_{bat} - \min\{\alpha, \kappa - B.e_{bat}\}$.
        $B.e_{bat} \leftarrow \min\{B.e_{bat} + \alpha, \kappa\}$.
function UseEnergy( )
    Let $a$ be the next action of $P$ and let $\delta(P)$ be its energy cost.
    if $P.e_{bat} \geq \delta(P)$ and $\neg P$.inhibit (i.e., $P$ is not inhibited) then
        Spend the required energy with $P.e_{bat} \leftarrow P.e_{bat} - \delta(P)$.
        Perform action $a$.
```

updates its parent pointer to A.parent $\leftarrow B$. This repeats until all amoebots are active, yielding a spanning forest $\mathcal{F}$.

The Communication Phase. The communication phase facilitates the long-range communication of amoebots' energy states analogous to the biofilm's potassium ion signaling. This is achieved by sending signals along an amoebot's tree in the spanning forest $\mathcal{F}$ constructed in the setup phase. In particular, any active amoebot A that is stressed — i.e., $A.e_{bat} < \delta(A)$ — sets a *stress flag* that remains until A is no longer stressed. Any amoebot that has a child in its tree with their stress flag set also sets their stress flag, effectively propagating this signal up to its tree's root amoebot. When the root amoebot receives this stress signal (or if it is itself stressed), it sets an *inhibit flag*, initiating a broadcast to the rest of the tree. Any amoebot whose parent in the tree has their inhibit flag set also sets their inhibit flag, propagating this inhibition signal throughout the tree. In the usage phase, inhibited amoebots are prohibited from spending their energy to perform actions, allowing more energy to pass on to the stressed amoebots. As we will show in the simulations of Section 4.4, omitting this phase can result in the indefinite starvation of many of the system's amoebots.

Signal resets behave analogously to how they are set. Any non-root amoebot that is not stressed — i.e., $A.e_{bat} \geq \delta(A)$ — and has no children with their stress flags set will reset its stress flag. Once a root no longer has any children with stress flags (and it is itself not stressed), it resets its inhibit flag. Any amoebot whose parent does not have its inhibit flag set resets its own inhibit flag, and so on.

The Sharing Phase. During the sharing phase, amoebots harvest energy from external energy sources and transfer energy to their neighbors, if possible. A root amoebot A begins the sharing phase by harvesting $\min\{\alpha, \kappa - A.e_{bat}\}$ units of energy from its external energy source. Any amoebot A — root or active — then checks to see if it has sufficient energy to share (i.e., $A.e_{bat} \geq \alpha$) and if any of its children in the spanning forest $\mathcal{F}$, say B, need energy (i.e., $B.e_{bat} < \kappa$). If so, A transfers $\min\{\alpha, \kappa - B.e_{bat}\}$ units of energy to B in keeping with the assumption from Section 4.2 that each amoebot can transfer at most α units of energy per activation.

The Usage Phase. In the usage phase, amoebots spend their energy to perform actions as required by their collective behavior. Suppose that a is the next action an amoebot A wants to perform; recall that its energy cost is given by $\delta(A)$. If A has sufficient stored energy to perform this action — i.e., $A.e_{bat} \geq \delta(A)$ — and A does not have its inhibit flag set, then A can spend the required energy and perform action a. Otherwise, A forgoes any action in this activation.

4.3.2 Analysis

We now prove the correctness and bound the runtime of the Energy-Sharing algorithm. We begin with two straightforward results regarding the setup and communication phases.

Lemma 4.3.1. *All idle amoebots in the system become active and join the spanning forest $\mathcal{F}$ within n sequential rounds, where n is the number of amoebots in the system.*

Proof. This follows directly from the analysis of the spanning forest primitive [52]. The amoebot system is connected, so as long as there are still idle amoebots in the system, at least one idle amoebot A must have an active or root amoebot as a neighbor. When A is next activated, it will become active and join the spanning forest by choosing one of its active or root neighbors as its parent. This is guaranteed to happen within one sequential round since every amoebot is activated at least once per round. Thus, at least one idle amoebot becomes active each round, and there are at most $n - 1$ idle amoebots since there is at least one root in the system initially. □

For the remainder of the analysis it suffices to focus on a single tree $\mathcal{T} \in \mathcal{F}$ since each tree in the spanning forest acts independently of the others.

Lemma 4.3.2. *Suppose an amoebot A in tree $\mathcal{T} \in \mathcal{F}$ is stressed; i.e., $A.e_{bat} < \delta(A)$. If tree $\mathcal{T}$ has depth $d_{\mathcal{T}}$, then all amoebots in $\mathcal{T}$ will have their inhibit flags set within $2d_{\mathcal{T}}$ sequential rounds.*[2]

Proof. Within one sequential round, amoebot A will be activated and will set its stress flag since $A.e_{bat} < \delta(A)$. Recall that stress flags are propagated up to the root by parents setting their stress flags when they see a child with its stress flag set. There can be at most $d_{\mathcal{T}} - 2$ ancestors of A strictly between A and the root. At least one more ancestor will set its stress flag per sequential round, so in at most $d_{\mathcal{T}} - 2$ rounds a child of the root will have its stress flag set.

Within one additional round, the root will be activated and will set its inhibit flag. Inhibit flags are then propagated from the root to all its descendants: in each round, any child that sees its parent's inhibit flag set will also set its own inhibit flag. The longest root-to-descendant path in $\mathcal{T}$ is of length $d_{\mathcal{T}}$, so in at most $d_{\mathcal{T}}$ rounds all amoebots in $\mathcal{T}$ will have their inhibit flags set. □

Lemma 4.3.2 shows that when a tree contains at least one stressed amoebot, every amoebot in the tree eventually becomes inhibited. This inhibition remains until all stressed amoebots *recharge*, i.e., until they receive the energy they need to perform their next action. The usage phase prohibits any inhibited amoebot from spending its energy on actions, so it suffices when bounding the recharge time to analyze how energy is shared within the tree.

In particular, we want to bound the worst case time for a stressed amoebot in a given tree $\mathcal{T}$ to recharge once all amoebots in $\mathcal{T}$ are inhibited. We make three

[2]The *depth* of an amoebot A in a tree $\mathcal{T}$ rooted at an amoebot R is the number of nodes in the (R, A)-path in $\mathcal{T}$ (i.e., the root R is at depth 1, and so on). The depth of a tree $\mathcal{T}$ is $\max_{A \in \mathcal{T}}\{\text{depth of } A\}$.

observations that make this analysis more tractable. First, we assume that all amoebots in $\mathcal{T}$ begin this recharging process with empty batteries and need to meet maximum energy demand; i.e., we assume $P.e_{bat} = 0$ and $\delta(A) = \kappa$ for all $A \in \mathcal{T}$. Although the amoebots of $\mathcal{T}$ may have obtained some energy before becoming inhibited, this assumption can only make recharging slower since more energy is needed. Second, we assume $\kappa/\alpha \in \mathbb{N}$, allowing us to assume all energy is transferred in units of size exactly α. This can be easily realized by rounding any given capacity κ up to the next multiple of α, as this can only increase the energy required in recharging. Third, we show in the following lemma that the recharge time in $\mathcal{T}$ is at most the recharge time in a simple path with the same number of amoebots.

Lemma 4.3.3. *Suppose $\mathcal{T}$ is a tree of k amoebots rooted at an amoebot R with access to external energy. If all k amoebots are inhibited and initially have no energy in their batteries, then the worst case number of sequential rounds to recharge all amoebots' batteries in $\mathcal{T}$ is at most the worst case number of rounds to do so in a path $\mathcal{L} = (A_1, \ldots, A_k)$ in which A_1 has access to external energy and $A_i.parent = A_{i-1}$ for all $1 < i \leq k$.*

Proof. Given any tree $\mathcal{U}$ of k inhibited amoebots rooted at an amoebot R with access to external energy and an activation sequence S of the amoebots in $\mathcal{U}$, let $t_S(\mathcal{U})$ denote the number of sequential rounds required to recharge all amoebots' batteries in $\mathcal{U}$ with respect to activation sequence S. We use $t(\mathcal{U}) = \max_S\{t_S(\mathcal{U})\}$ to denote the worst case recharge time for $\mathcal{U}$. With this notation, our goal is to show that $t(\mathcal{T}) \leq t(\mathcal{L})$.

Consider the maximal "non-branching" path $(R = A_1, \ldots, A_\ell = A)$ in tree $\mathcal{T}$ starting at the root R such that A_{i+1} is the only child amoebot of A_i in $\mathcal{T}$ for all $1 \leq i < \ell$. We argue by (reverse) induction on ℓ, the total number of amoebots in the maximal non-branching path of $\mathcal{T}$. If $\ell = k$, then $\mathcal{T}$ is already a path $\mathcal{L}$

of k amoebots and we have $t(\mathcal{T}) = t(\mathcal{L})$ trivially. So suppose that $\ell < k$ and for all possible trees $\mathcal{U}$ composed of the same k amoebots as $\mathcal{T}$ that are rooted at R and have at least $\ell + 1$ amoebots in their maximal non-branching paths starting at R, we have $t(\mathcal{U}) \leq t(\mathcal{L})$. Our goal is to modify $\mathcal{T}$ to form another tree $\mathcal{T}'$ that is composed of the same amoebots, is rooted at R, and has exactly one more amoebot in its maximal non-branching path such that $t(\mathcal{T}) \leq t(\mathcal{T}')$. Since $\mathcal{T}'$ has exactly $\ell + 1$ amoebots in its maximal non-branching path, we have by the induction hypothesis that $t(\mathcal{T}) \leq t(\mathcal{T}') \leq t(\mathcal{L})$.

With maximal non-branching path $(R = A_1, \ldots, A_\ell = A)$ of $\mathcal{T}$, $A = A_\ell$ is the "closest" amoebot to R with multiple children, say $B_1, \ldots, B_c$ for $c \geq 2$; note that such an amoebot A must exist since $\ell < k$. Form the tree $\mathcal{T}'$ by reassigning B_i.parent from A to B_1 for each $2 \leq i \leq c$. Then B_1 is the only child of A in $\mathcal{T}'$, and thus $(R = A_1, \ldots, A_\ell = A, B_1)$ is the maximal non-branching path of $\mathcal{T}'$ which has length $\ell + 1$. So it suffices to show that $t(\mathcal{T}) \leq t(\mathcal{T}')$.

Consider any activation sequence $S = (s_1, \ldots, s_f)$ where s_f is the first activation after which all amoebots in $\mathcal{T}$ have finished recharging; we must show that there exists an activation sequence S' such that $t_S(\mathcal{T}) \leq t_{S'}(\mathcal{T}')$. We construct S' from S so that the flow of energy through $\mathcal{T}'$ mimics that of $\mathcal{T}$. Consider each $s_i \in S$, for $1 \leq i \leq f$. In most cases, s_i has the same effect in both $\mathcal{T}$ and $\mathcal{T}'$ and thus $s_i' = s_i$ can be appended to S'. However, any activations s_i in which A passes energy to a child B_j, for $2 \leq j \leq c$, cannot be performed directly in $\mathcal{T}'$ since B_j is a child of B_1 — not of A — in $\mathcal{T}'$. We instead add a pair of activations $s_i' = (s_i^1, s_i^2)$ to S' that have the effect of passing energy from A to B_j but use B_1 as an intermediary. There are two cases. If B_1 has a full battery (i.e., $B_1.e_{bat} = \kappa$) at the beginning of s_i, then B_1 passes energy to B_j in s_i^1 and A passes energy to B_1 in s_i^2. Otherwise, A passes energy to B_1 in s_i^1 and B_1 passes energy to B_j in s_i^2.

Under this construction of S', if all amoebots start with empty batteries, the value of $A.e_{bat}$ after each $s_i \in S$ and $s'_i \in S'$ is the same in $\mathcal{T}$ and $\mathcal{T}'$, respectively, for all $1 \leq i \leq f$. Thus, the amoebots in $\mathcal{T}$ and $\mathcal{T}'$ only finish recharging after activations s_f and s'_f, respectively. Moreover, S' was obtained from S by adding activations which can only increase the number of sequential rounds in S'. Therefore, we have $t_S(\mathcal{T}) \leq t_{S'}(\mathcal{T}')$, and since the choice of S was arbitrary, we have $t(\mathcal{T}) \leq t(\mathcal{T}')$. □

By Lemma 4.3.3, it suffices to analyze the case where $\mathcal{T}$ is a simple path of k amoebots. To bound the recharge time in this setting, we use a *dominance argument* between sequential and parallel executions which is structured as follows. First, we prove that for any sequential execution, there exists a parallel execution that makes at most as much progress towards recharging the system in the same number of rounds. We then upper bound the recharge time in parallel rounds. Combining these results gives a worst case upper bound on the recharge time in sequential rounds, as desired. Remark 1 gives background on dominance arguments independent of the present focus on energy distribution.

Remark 1. Dominance Arguments.

The concept of *dominance* appears in many areas of mathematics. Perhaps the most ubiquitous examples in computer science are asymptotic dominating functions that give us big-$\mathcal{O}$, Ω, and Θ notation [122]. In that context, we say that a function f dominates a function g if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, it holds that $cf(n) \geq g(n)$. Another example comes from game theory, where dominating strategies are investigated for games between two individuals or parties with competing interests [141, 185]. Just as for the asymptotic dominating functions, invariant inequalities are sought to show that one strategy always outperforms the other.

In this dissertation, we leverage dominance to bound the runtime of our algorithms by comparing sequential executions to parallel executions. Because a sequential adversary may activate any amoebot in arbitrary order — so long as each amoebot is activated infinitely often — it can be difficult to directly measure progress, even within sequential rounds where each amoebot is guaranteed to be activated at least once. In contrast, a parallel execution activates all amoebots in lock-step, providing rigid structure that is more amenable to runtime analysis.

Informally, our dominance arguments prove two results: (*i*) for any sequential execution S of an algorithm $\mathcal{A}$, there exists a parallel execution P of $\mathcal{A}$ that is dominated by S; i.e., S always makes at least as much progress as P in the same number of rounds, and (*ii*) the parallel execution P of $\mathcal{A}$ is guaranteed to terminate in $\mathcal{O}(f(n))$ rounds, where n is the number of amoebots in the system. Combining these results, we have that S must also terminate in $\mathcal{O}(f(n))$ rounds. If S is chosen arbitrarily in (*i*), then $\mathcal{O}(f(n))$ is an upper bound on the runtime of any sequential execution of $\mathcal{A}$, as desired. While the measure of progress will vary depending on what process is being analyzed, our dominance arguments will always follow this structure.

Let a configuration C of the path $A_1, \ldots, A_k$ encode the battery values of each amoebot A_i as $C(A_i)$. A *schedule* is a sequence of configurations $(C_0, \ldots, C_t)$. Note that in the following definition for the parallel execution, we reduce each amoebot's battery capacity from κ to $\kappa' = \kappa - \alpha$. This does not apply to the sequential execution, and is just a proof artifact that will be useful in Lemma 4.3.5.

Definition 4.3.4. A *parallel energy schedule* $(C_0, \ldots, C_t)$ is a schedule such that for all configurations C_i and amoebots A_j we have $C_i(A_j) \in [0, \kappa']$ and, for every $0 < i \leq t$, C_i is reached from C_{i-1} using the following for each amoebot A_j:

- A_j is a root, so it harvests energy from the external energy source with:
    - $C_i(A_j) = C_{i-1}(A_j) + \min\{\alpha, \kappa' - C_{i-1}(A_j)\}$
- $C_{i-1}(A_j) \geq \alpha$ and $C_{i-1}(A_{j+1}) < \kappa'$, so A_j passes energy to its child with:
    - $C_i(A_j) = C_{i-1}(A_j) - \min\{\alpha, \kappa' - C_{i-1}(A_{j+1})\}$
    - $C_i(A_{j+1}) = C_{i-1}(A_{j+1}) + \min\{\alpha, \kappa' - C_{i-1}(A_{j+1})\}$

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible.

Now consider any fair sequential activation sequence S; i.e., one in which every amoebot is activated infinitely often. We compare a greedy parallel energy schedule to a *sequential energy schedule* $(C_0^S, \ldots, C_t^S)$ where C_i^S is the configuration of the path $A_1, \ldots, A_k$ at the completion of the i-th sequential round in S. For an amoebot A_i in a configuration C, let $\Delta_C(A_i)$ denote the total amount of energy in the batteries of amoebots $A_i, \ldots, A_k$ in C; i.e., $\Delta_C(A_i) = \sum_{j=i}^{k} C(A_j)$. For any two configurations C and C', we say C *dominates* C' — denoted $C \succeq C'$ — if and only if for all amoebots A_i in the path $A_1, \ldots, A_k$, we have $\Delta_C(A_i) \geq \Delta_{C'}(A_i)$.

Lemma 4.3.5. *Given any fair sequential activation sequence S beginning at a configuration C_0^S in which $A_i.e_{bat} = 0$ for all $1 \leq i \leq k$, there exists a greedy parallel energy schedule $(C_0, \ldots, C_t)$ with $C_0 = C_0^S$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq t$.*

Proof. Given a fair sequential activation sequence S and an initial configuration C_0^S, we obtain a unique sequential energy schedule $(C_0^S, \ldots, C_t^S)$. Our goal is to construct a parallel energy schedule $(C_0, \ldots, C_t)$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq t$. Let

$C_0 = C_0^S$; then, for $0 < i \leq t$, let C_i be obtained from C_{i-1} by performing one *parallel round*: each amoebot greedily performs the actions of Definition 4.3.4 if possible.

We now show $C_i^S \succeq C_i$ for all $0 \leq i \leq t$ by induction on i. Since $C_0 = C_0^S$, we trivially have $C_0^S \succeq C_0$. So suppose $i > 0$ and for all rounds $0 \leq r < i$ we have $C_r^S \succeq C_r$. Considering any amoebot A_j, we have $\Delta_{C_{i-1}^S}(A_j) \geq \Delta_{C_{i-1}}(A_j)$ by the induction hypothesis and want to show that $\Delta_{C_i^S}(A_j) \geq \Delta_{C_i}(A_j)$. First suppose the inequality from the induction hypothesis is strict and we have $\Delta_{C_{i-1}^S}(A_j) > \Delta_{C_{i-1}}(A_j)$, meaning strictly more energy has been passed into $A_j, \ldots, A_k$ in the sequential setting than in the parallel one after rounds $i-1$ are complete. Because all successful energy transfers pass α energy either from the external source to the root A_1 or from a parent A_j to its child A_{j+1}, we have that $\Delta_{C_{i-1}^S}(A_j) \geq \Delta_{C_{i-1}}(A_j) + \alpha$. But by Definition 4.3.4, an amoebot can receive at most α energy per parallel round, so we have:

$$\Delta_{C_i}(A_j) \leq \Delta_{C_{i-1}}(A_j) + \alpha \leq \Delta_{C_{i-1}^S}(A_j) \leq \Delta_{C_i^S}(A_j)$$

Thus, it remains to consider when $\Delta_{C_{i-1}^S}(A_j) = \Delta_{C_{i-1}}(A_j)$, meaning the amount of energy passed into $A_j, \ldots, A_k$ is exactly the same in the sequential and parallel settings after rounds $i-1$ are complete. It suffices to show that if A_j receives α energy in parallel round i, then it also does so in sequential round i.

We first prove that if A_j receives α energy in parallel round i, then $C_{i-1}^S(A_j) \leq \kappa - \alpha$; i.e., A_j has enough room in its battery to receive α energy whenever it is activated in sequential round i. There are two cases: either A_j already had enough room in its battery to receive α energy in parallel round i (i.e., $C_{i-1}(A_j) \leq \kappa' - \alpha$) or it had a full battery (i.e., $C_{i-1}(A_j) = \kappa'$) but passed α energy to A_{j+1} in parallel, "pipelining" energy to make room for the energy it received. In either case, it is easy to see that $C_{i-1}(A_j) \leq \kappa'$. By supposition we have $\Delta_{C_{i-1}^S}(A_j) = \Delta_{C_{i-1}}(A_j)$ and by the induction hypothesis we have $\Delta_{C^S_{i-1}}(A_{j+1}) \geq \Delta_{C_{i-1}}(A_{j+1})$. Combining these facts, we have:

$$\begin{aligned}
C^S_{i-1}(A_j) &= \sum_{\ell=j}^{k} C^S_{i-1}(A_\ell) - \sum_{\ell=j+1}^{k} C^S_{i-1}(A_\ell) \\
&= \Delta_{C^S_{i-1}}(A_j) - \Delta_{C^S_{i-1}}(A_{j+1}) \\
&\leq \Delta_{C_{i-1}}(A_j) - \Delta_{C_{i-1}}(A_{j+1}) \\
&= \sum_{\ell=j}^{k} C_{i-1}(A_\ell) - \sum_{\ell=j+1}^{k} C_{i-1}(A_\ell) \\
&= C_{i-1}(A_j) \leq \kappa' = \kappa - \alpha
\end{aligned}$$

Thus, regardless of whether A_j already had space for α energy or used pipelining in parallel round i, A_j must have space for α energy at the start of sequential round i, as desired.

Next, we show that if A_j receives α energy in parallel round i, then there is at least α energy for A_j to receive in sequential round i. If A_j is the root, this is trivial: the external source of energy is its infinite supply. Otherwise, $j > 1$ and we must show $C^S_{i-1}(A_{j-1}) \geq \alpha$. We have $\Delta_{C^S_{i-1}}(A_j) = \Delta_{C_{i-1}}(A_j)$ by supposition and $\Delta_{C^S_{i-1}}(A_{j-1}) \geq \Delta_{C_{i-1}}(A_{j-1})$ by the induction hypothesis, so:

$$\begin{aligned}
C^S_{i-1}(A_{j-1}) &= \sum_{\ell=j-1}^{k} C^S_{i-1}(A_\ell) - \sum_{\ell=j}^{k} C^S_{i-1}(A_\ell) \\
&= \Delta_{C^S_{i-1}}(A_{j-1}) - \Delta_{C^S_{i-1}}(A_j) \\
&\geq \Delta_{C_{i-1}}(A_{j-1}) - \Delta_{C_{i-1}}(A_j) \\
&= \sum_{\ell=j-1}^{k} C_{i-1}(A_\ell) - \sum_{\ell=j}^{k} C_{i-1}(A_\ell) \\
&= C_{i-1}(A_{j-1}) \geq \alpha
\end{aligned}$$

Thus, we have shown that if A_j receives α energy in parallel round i, then $C^S_{i-1}(A_j) \leq \kappa - \alpha$ and either $j = 1$ or $C^S_{i-1}(A_{j-1}) \geq \alpha$, meaning that at the end of sequential round $i - 1$ there is both α energy available to pass to A_j and A_j has room in its battery to receive it. Though we do not control the order of activations in sequential round i, additional activations can only increase the amount of energy available to pass to A_j (by, e.g., passing more energy to A_{j-1}) and increase the space available in $A_j.e_{bat}$ (by passing more energy to A_{j+1}). Since the activation sequence S was assumed to be fair, either $j = 1$ and A_j will be activated at least once in sequential round i or $j > 1$ and A_{j-1} will be activated at least once in sequential round i; in either case, A_j will receive α energy in sequential round i. Therefore, in all cases we have shown that $\Delta_{C_i^S}(A_j) \geq \Delta_{C_i}(A_j)$, and since the choice of A_j was arbitrary, we have $C_i^S \succeq C_i$ as desired. □

To conclude the dominance argument, we bound the number of parallel rounds needed to recharge a path of k amoebots. Combined with Lemma 4.3.5, this gives an upper bound on the worst case number of sequential rounds required to do the same.

Lemma 4.3.6. *Let $(C_0, \ldots, C_t)$ be a greedy parallel energy schedule where C_0 is the configuration in which $A_i.e_{bat} = 0$ for all $1 \leq i \leq k$ and C_t is the configuration in which $A_i.e_{bat} = \kappa' = \kappa - \alpha$ for all $1 \leq i \leq k$. Then $t = \frac{\kappa'}{\alpha}k = \mathcal{O}(k)$.*

Proof. We argue by induction on k, the number of amoebots in the path. If $k = 1$, then $A_1 = A_k$ is the root amoebot that harvests α energy per parallel round from the external source by Definition 4.3.4. Since A_1 has no children to which it may pass energy, clearly, within $\frac{\kappa'}{\alpha} = \mathcal{O}(k)$ rounds $A_1.e_{bat} = \kappa'$ will be satisfied.

Now suppose $k > 1$ and that for all $1 \leq j < k$, a path of j amoebots fully recharges in $\frac{\kappa'}{\alpha}j$ parallel rounds. Once an amoebot A_i has received energy for the first time, it is easy to see by inspection of Definition 4.3.4 that A_i will receive α energy from A_{i-1} (or the external energy source, in the case that $i = 1$) in every subsequent parallel round until $A_i.e_{bat}$ is full. Similarly, Definition 4.3.4 ensures that A_i will pass α energy to A_{i+1} in every subsequent parallel round until $A_{i+1}.e_{bat}$ is full. Thus, once A_i receives energy for the first time, A_i effectively acts as an external energy source for the remaining amoebots $A_{i+1}, \ldots, A_k$.

The root A_1 first harvests energy from the external energy source in parallel round 0, and thus acts as a continuous energy source for $A_2, \ldots, A_k$ in all subsequent rounds. By the induction hypothesis, we have that $A_2, \ldots, A_k$ will fully recharge in $\frac{\kappa'}{\alpha}(k-1)$ parallel rounds, after which A_1 will no longer pass energy to A_2. The root A_1 harvests α energy from the external energy source per parallel round and already has $A_1.e_{bat} = \alpha$, so in an additional $\frac{\kappa'}{\alpha} - 1$ parallel rounds we have $A_1.e_{bat} = \kappa'$. Therefore, the path $A_1, \ldots, A_k$ fully recharges in $1 + \frac{\kappa'}{\alpha}(k-1) + \frac{\kappa'}{\alpha} - 1 = \frac{\kappa'}{\alpha}k = \mathcal{O}(k)$ parallel rounds, as required. □

Lemmas 4.3.3, 4.3.5, and 4.3.6 show that an inhibited tree $\mathcal{T}$ of k amoebots will recharge all its stressed amoebots in at most $\mathcal{O}(k)$ sequential rounds. The following lemma shows that within a bounded number of additional rounds, there will be some amoebot that is neither inhibited nor stressed and thus can perform an enabled action (if it has one).

Lemma 4.3.7. *Suppose that the last stressed amoebot in $\mathcal{T}$ has just received the energy it needs to perform its next action. If $\mathcal{T}$ has depth $d_{\mathcal{T}}$, then within $2d_{\mathcal{T}}$ additional rounds some amoebot in $\mathcal{T}$ with a pending enabled action will be able to perform it.*

Proof. Let $\mathcal{T}_a$ be the set of amoebots in $\mathcal{T}$ that have enabled actions to perform. By supposition, all amoebots in $\mathcal{T}_a$ now have sufficient energy stored in their batteries to perform their actions (i.e., they are no longer stressed). It remains to bound the time for an amoebot in $\mathcal{T}_a$ to reset its inhibit flag, the only remaining obstacle to performing its action.

Let $\mathcal{T}_s \subseteq \mathcal{T}$ be the connected subtree of amoebots with their stress flags set. All leaves of $\mathcal{T}_s$ at the start of a sequential round are guaranteed to reset their stress flags by the completion of the round since they are no longer stressed and do not have children with stress flags set. A descendant-to-root path in $\mathcal{T}_s$ can have length at most $d_{\mathcal{T}}$; the depth of tree $\mathcal{T}$. So in at most $d_{\mathcal{T}}$ rounds, all amoebots in $\mathcal{T}$ will reset their stress flags.

In the first sequential round in which the root does not have any children with their stress flags set, the root resets its inhibit flag. In each subsequent round, any child whose parent has reset its inhibit flag will also reset its own inhibit flag. The longest root-to-descendant path in $\mathcal{T}$ is of length $d_{\mathcal{T}}$, so in at most $d_{\mathcal{T}}$ rounds there must exist an amoebot in $\mathcal{T}_a$ that resets its inhibit flag; let A be the first such amoebot. Amoebot A has an enabled action, has sufficient energy stored, and is not inhibited, so it performs its enabled action during its next usage phase. □

We conclude our analysis with the following two theorems. Recall from Section 4.2 that an algorithm solves the energy distribution problem in t sequential rounds if no amoebot remains stressed for more than t rounds and at least one amoebot is able to perform an enabled action every t rounds.

Theorem 4.3.8. *Algorithm* Energy-Sharing *solves the energy distribution problem in $\mathcal{O}(n)$ sequential rounds.*

Proof. By Lemma 4.3.1, all n amoebots in system $\mathcal{S}$ will join the spanning forest $\mathcal{F}$ within n sequential rounds. Since there is no communication or energy transfer between different trees of $\mathcal{F}$, it suffices to analyze an arbitrary tree $\mathcal{T} \in \mathcal{F}$. By Lemma 4.3.2, if $\mathcal{T}$ contains a stressed amoebot then all amoebots of $\mathcal{T}$ will be inhibited within $2d_{\mathcal{T}}$ rounds, where $d_{\mathcal{T}}$ is the depth of $\mathcal{T}$. Lemma 4.3.3 shows that assuming $\mathcal{T}$ has a path structure can only increase the time to recharge its stressed amoebots, and Lemmas 4.3.5 and 4.3.6 prove that even in the case that all amoebots have uniform, maximum demand — i.e., $\delta(A) = \kappa$ for all amoebots A — all stressed amoebots will be distributed enough energy to meet their demand within $\mathcal{O}(|\mathcal{T}|)$ rounds. Finally, Lemma 4.3.7 shows that within $2d_{\mathcal{T}}$ additional rounds some amoebot in $\mathcal{T}$ will use its energy to perform its next enabled action. Therefore, since the depth of $\mathcal{T}$ can be at most its size (if $\mathcal{T}$ is a path) and its size can be at most the number of amoebots in the system (if $\mathcal{T}$ is the only tree in $\mathcal{F}$), we conclude that Energy-Sharing solves the energy distribution problem in $n + 2d_{\mathcal{T}} + \mathcal{O}(|\mathcal{T}|) + 2d_{\mathcal{T}} = \mathcal{O}(n)$ sequential rounds. □

To establish a lower bound, observe that for a system of n amoebots each with a battery capacity of κ to fully recharge, the system needs to harvest and distribute $n\kappa$ total energy. Each amoebot with access to an external energy source may only be activated once per sequential round in the worst case. So in this worst case, a system with $s \leq n$ amoebots with energy access can harvest at most $s\alpha$ energy from external sources per sequential round. This yields the following theorem.

Theorem 4.3.9. *The worst case runtime for any local control algorithm to solve the energy distribution problem when $s \leq n$ amoebots have access to external energy sources is $\Omega(n/s)$ sequential rounds.*

Theorems 4.3.8 and 4.3.9 yield the following corollary.

Corollary 4.3.10. *Algorithm* Energy-Sharing *is asymptotically optimal when the number of amoebots with access to external energy sources is a fixed constant.*

4.4 Simulation Results

We now present simulations of the Energy-Sharing algorithm.[3] All figures in this section use color intensity to indicate the energy level of an amoebot's battery, with more intense color corresponding to more energy stored. Our first simulation (Figure 9) shows Energy-Sharing running on a system of 91 amoebots with a single root amoebot that has access to an external energy source. All amoebots have a capacity of $\kappa = 10$ and a transfer rate of $\alpha = 1$. To incorporate energy usage in the simulation, we assume that every amoebot has a uniform, repeating demand of $\delta(\cdot, \cdot) = 5$ energy per "action", though no explicit action is actually performed when the energy is used. The system is organized as a hexagon with the root at its center for visual clarity, but the resulting behavior is characteristic of other initial configurations, root placements, and parameter settings.

All amoebots are initially idle, with the exception of the root shown with a gray/black ring (Figure 9a). The setup phase establishes the spanning forest (or tree, in this case) rooted at amoebots with energy access; an amoebot's parent direction is shown as an arc. Since all amoebots start with empty batteries, stress flags (shown as red rings) quickly propagate throughout the system and inhibit flags soon follow (Figure 9b). As energy is harvested by the root and shared throughout the system, some amoebots (shown with yellow rings) receive sufficient energy to meet the demand for their next action but remain inhibited from using it (Figure 9c). This inhibition remains until all stressed amoebots in the system receive sufficient energy to meet their demands (Figure 9d), at which point amoebots (shown with green rings) can reset

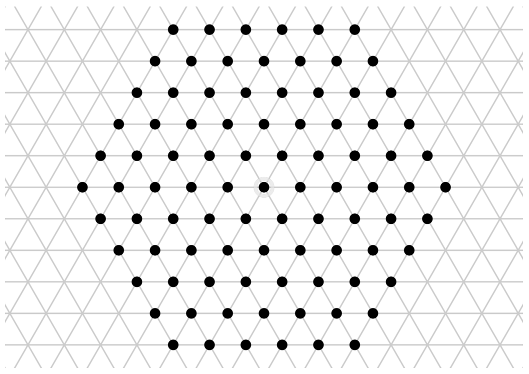
[3]Code for all simulations is openly available as part of AmoebotSim (https://github.com/SOPSLab/AmoebotSim), a visual simulator for the amoebot model of programmable matter. Enlarged videos of simulations can be viewed at https://sops.engineering.asu.edu/sops/energy-distribution.

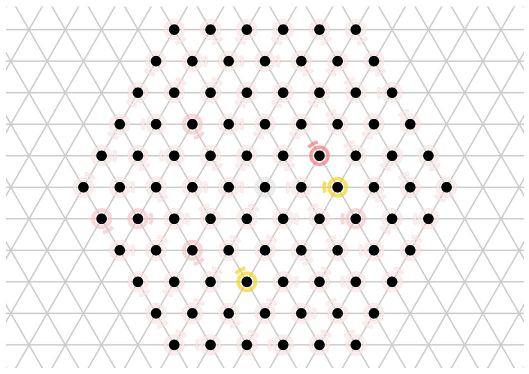
their inhibit flags and use their energy (Figure 9e). After using energy, these amoebots may again become stressed and trigger another stage of inhibition (Figure 9f).

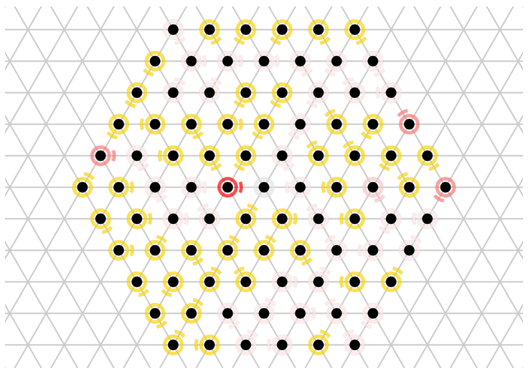
Our second simulation demonstrates the necessity of the communication phase for effective energy distribution. In Section 4.1, we motivated the need for a strategy that leverages the biofilm-inspired long-range communication of amoebots' energy states to shift between selfish and altruistic energy usage. Figure 10 shows a simulation with the same initial configuration and parameters as the first simulation (Figure 9), but with its communication phase disabled. Without the communication phase to inhibit amoebots from using energy while those that are stressed recharge, amoebots continuously share any energy they have with their descendants in the spanning forest. Thus, while the leaves of the spanning forest occasionally meet their energy demands (bold green amoebots in Figure 10b–10d), even after 1000 rounds most amoebots have still not met their energy demand even once.

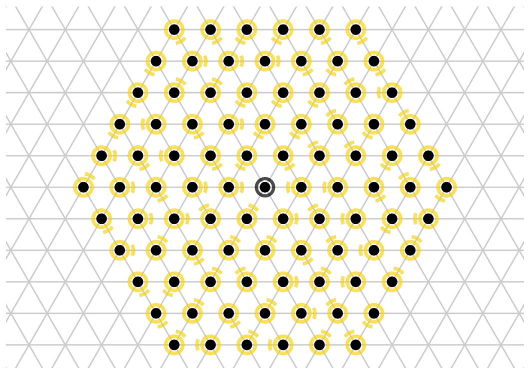
4.5 Extensions

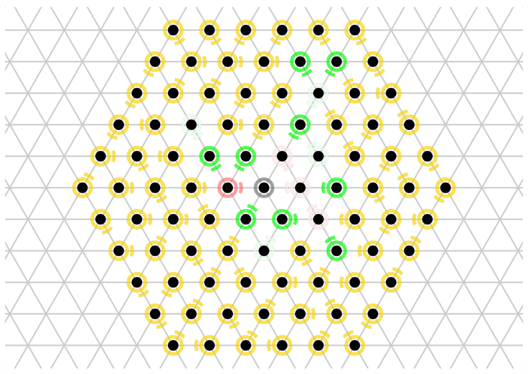
With our energy distribution algorithm in place, we now present useful extensions. We begin by considering amoebot *crash failures* in which an amoebot stops functioning and no longer participates in the collective behavior. Crash failures pose a key challenge for ENERGY-SHARING: they disrupt the structure of the spanning forest $\mathcal{F}$ that the amoebots use for routing energy and communicating their energy states. To achieve robustness to these crash failures, we present algorithm FOREST-PRUNE-REPAIR that enables the spanning forest to self-repair so long as certain assumptions on the locations of faulty amoebots hold (Sections 4.5.1–4.5.2). We then show how FOREST-PRUNE-REPAIR can be leveraged to compose ENERGY-SHARING with existing algorithms


(a) $t = 0$ seq. rounds


(b) $t = 10$


(c) $t = 100$


(d) $t = 190$


(e) $t = 191$

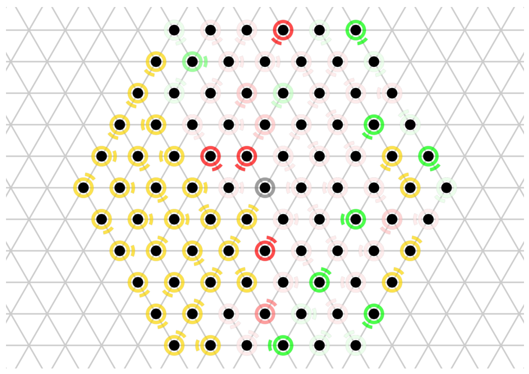

(f) $t = 192$

Figure 9. Simulation of ENERGY-SHARING. This system has 91 amoebots with one root, $\kappa = 10$, $\alpha = 1$, and a repeating uniform demand of $\delta(\cdot, \cdot) = 5$ for all amoebots. The black amoebot is the root, red amoebots have their stress flags and possibly also their inhibit flags set, yellow amoebots have only their inhibit flags set, and green amoebots have no flags set.

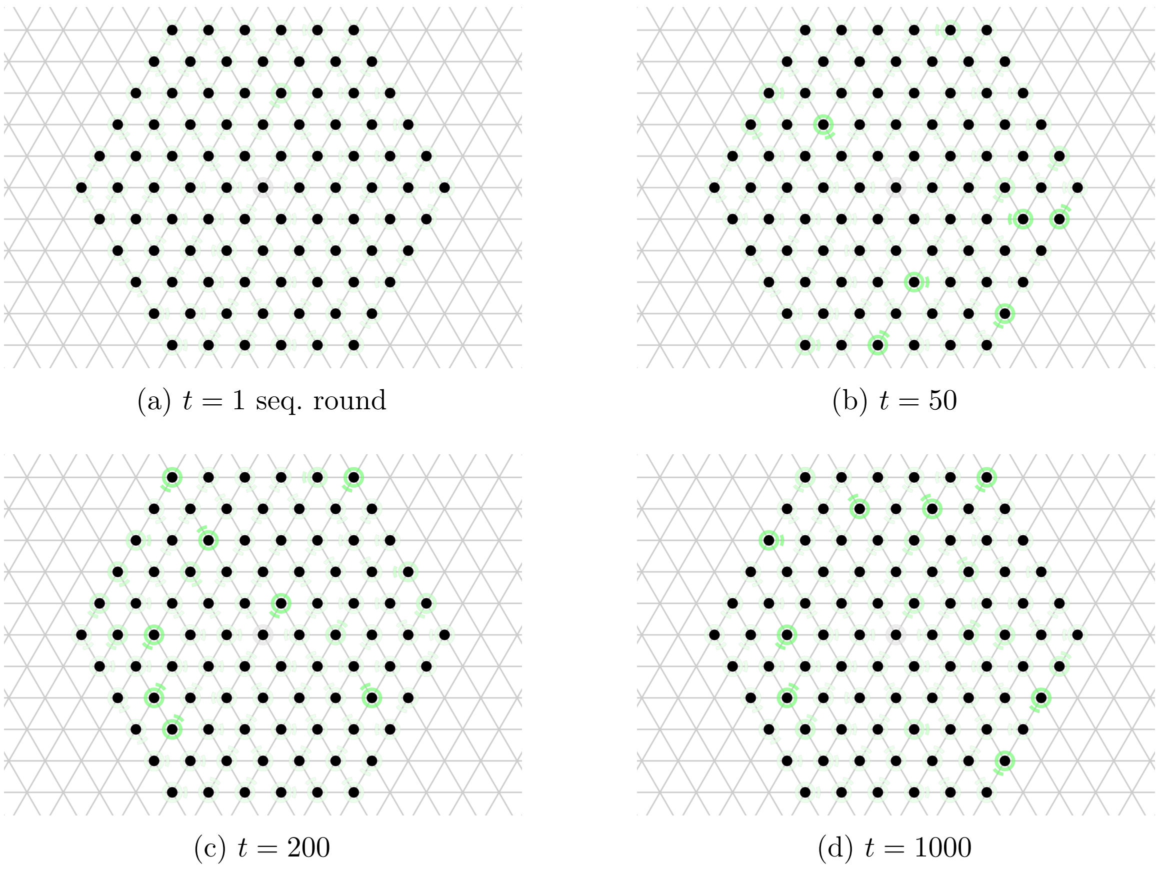

(a) $t = 1$ seq. round
(b) $t = 50$
(c) $t = 200$
(d) $t = 1000$

Figure 10. Simulation of Energy-Sharing Without Communication. Uses the same initial configuration and parameters as in Figure 9, but with the communication phase disabled. Without communication to set stress and inhibit flags, all amoebots remain uninhibited (green), but only the leaves of the spanning forest ever amass enough energy to meet their demands.

in the amoebot catalogue, effectively generalizing all previous work on the amoebot model to also consider energy constraints (Section 4.5.3).

We make three assumptions about crashed amoebots. First, the neighbors of a crashed amoebot can detect that it is crashed. Second, the subgraph induced by the positions of non-crashed amoebots must remain connected at all times; otherwise, there may be no way for components of non-crashed amoebots to communicate. Third, there must always be at least one non-crashed root amoebot; otherwise, the system would lose access to all external energy sources. We do not claim that these *detection*, *connectivity*, and *root-reliability* assumptions are necessary for fault tolerance, but each addresses a non-trivial challenge that is beyond the scope of this work.

4.5.1 The Forest-Prune-Repair Algorithm

In the context of our energy distribution algorithm, crash failures partition the spanning forest $\mathcal{F}$ into "non-faulty" trees $\mathcal{F}^*$ that are rooted at amoebots with energy access and "faulty" trees $\mathcal{F}'$ that are disconnected from any external energy source. Together, $\mathcal{F}^* \cup \mathcal{F}'$ form a forest that spans all non-crashed amoebots. To make our algorithms robust to these faults, we present Forest-Prune-Repair (Algorithm 8), a local, ad hoc reconstruction that self-repairs $\mathcal{F}$ to reform a spanning forest of trees rooted at amoebots with energy access.

Algorithm Forest-Prune-Repair works as follows. When an amoebot A finds that its parent has crashed, it knows it has become the root of a faulty tree in $\mathcal{F}'$. In response, A broadcasts a "prune signal" throughout this new tree by setting a *prune flag* in each of its children's memories, informing its descendants of the crash failure. It then clears its parent pointer, resets all flags, and becomes idle. Any amoebot that has its prune flag set does the same, effectively dissolving the faulty tree. Idle amoebots then rejoin an existing tree in a manner similar to the setup phase described in Section 4.3. When activated, an idle amoebot A considers all its root or active neighbors that do not have their prune flag set. Of these amoebots, A chooses one to be its parent in a round-robin manner; i.e., if A is ever pruned again, it chooses the next such amoebot to be its parent.

Integrating Forest-Prune-Repair with Energy-Sharing is straightforward. In the setting where the system is subject to crash faults, Forest-Prune-Repair

Algorithm 8 Forest-Prune-Repair for Amoebot A

```
1: if ($A$.prune) $\vee$ ($A$.parent is crashed) then
2:     for all amoebots $B$ such that $B$.parent $= A$ do $B$.prune $\leftarrow$ TRUE.
3:     $A$.parent $\leftarrow$ NULL.
4:     $A$.prune $\leftarrow$ FALSE.
5:     $A$ becomes idle.
6: else if ($A$ is idle) $\wedge$ ($A$ has a root or active neighbor $B$ such that $\neg B$.prune) then
7:     Choose a root or active neighbor $B$ with $\neg B$.prune by round-robin selection.
8:     Update $A$.parent $\leftarrow B$.
9:     $A$ becomes active.
```

simply replaces the setup phase described in Section 4.3. An amoebot proceeds with the communication, sharing, and usage phases if it is not idle and its parent is not crashed.

4.5.2 Analysis

We now analyze Forest-Prune-Repair, beginning with a simple proof of safety that shows certain properties of the non-faulty trees in $\mathcal{F}^*$ are always preserved.

Lemma 4.5.1. *If a non-faulty tree $\mathcal{T} \in \mathcal{F}^*$ is initially acyclic, then under* Forest-Prune-Repair *it will remain acyclic. Moreover, there will always be at least one tree in $\mathcal{F}^*$.*

Proof. By the root-reliability assumption, there is always at least one non-crashed root amoebot; thus, $\mathcal{F}^*$ can never be empty. The operations of Forest-Prune-Repair that change the structure of forest $\mathcal{F}$ are the removal of amoebots from their trees during pruning and the addition of idle amoebots to new trees during rejoining. It is easy to see that removing amoebots from any tree during pruning cannot create cycles where there were none before. An amoebot only rejoins a tree if it is idle, implying that it has no children. So an idle amoebot rejoining a tree $\mathcal{T} \in \mathcal{F}^*$ is like adding a new leaf vertex to $\mathcal{T}$, which cannot create a cycle because $\mathcal{T}$ was initially acyclic. $\square$

Lemma 4.5.2. *Suppose an amoebot crashes, yielding a new faulty tree $\mathcal{T} \in \mathcal{F}'$. For any amoebot A at depth d in tree $\mathcal{T}$, A will be pruned (i.e., set its children's prune flags, clear its memory, and become idle) in at most d sequential rounds.*

Proof. Suppose an amoebot crashes in round r, yielding a new faulty tree $\mathcal{T} \in \mathcal{F}'$. Let A be any amoebot at depth d in $\mathcal{T}$. If $d = 1$, then A is the root of $\mathcal{T}$. Since every amoebot is activated at least once per sequential round, A will activate, see its parent is crashed, and prune itself by the end of round $r + 1$. Now suppose $d > 1$ and every amoebot at depth at most $d - 1$ in $\mathcal{T}$ has been pruned by the end of round $r + d - 1$. If A has already been pruned (as is possible due to the activation order), we are done. So suppose A has not yet been pruned at the start of round $r + d$. The parent of A was at depth $d - 1$, and thus must have set the prune flag of A and become pruned by the end of the previous round. So whenever A is activated in round $r + d$, it sees its prune flag is set and is pruned. Thus, in all cases, A is pruned in at most d rounds. $\square$

Under Forest-Prune-Repair, a pruned amoebot A chooses its new parent B from among its root or active neighbors that do not have their prune flags set. There are two cases: (i) B is in a non-faulty tree, meaning A has rejoined $\mathcal{F}^*$ as desired, or (ii) B is in a faulty tree, say $\mathcal{T} \in \mathcal{F}'$. In the latter case, there must be prune flags propagating throughout $\mathcal{T}$ because $\mathcal{T} \in \mathcal{F}'$, so Lemma 4.5.2 shows A will now be pruned again, this time from $\mathcal{T}$.

An especially bad version of this case occurs when an amoebot continually rejoins the tree it is pruning by choosing one of its descendants as its new parent (see Figure 11). In fact, if this choice is not made carefully, it is possible that such an amoebot would always choose a descendant as its parent and thus never rejoin $\mathcal{F}^*$. We refer to this situation as a *chase cycle* due to the way the prune flag propagation "chases" the rejoining amoebots. However, since amoebots choose their new parents from among their eligible neighbors in a round-robin manner, chase cycles cannot continue for long. We have the following lemma.

Lemma 4.5.3. *Suppose an amoebot A in faulty tree $\mathcal{T} \in \mathcal{F}'$ has at least one neighbor in a non-faulty tree of $\mathcal{F}^*$. Then the number of times A will be pruned before it rejoins $\mathcal{F}^*$ is at most 6.*

Proof. Each time A is pruned, it chooses a new parent from among its active or root neighbors that do not have their prune flags set. By supposition, A has at least one such neighbor in a tree of $\mathcal{F}^*$. Moreover, its neighbor(s) in $\mathcal{F}^*$ will always be in the set of eligible new parents since every amoebot in a non-faulty tree is either a root or is active and is never pruned. By Lemma 4.5.2, A will be pruned again each time it chooses a parent in a faulty tree of $\mathcal{F}'$. In a round-robin selection, A can choose each neighbor in $\mathcal{F}'$ as its parent at most once before choosing a parent in $\mathcal{F}^*$, as desired. Every amoebot has at most 6 neighbors, so in the worst case the number of times A will be pruned before it rejoins $\mathcal{F}^*$ is 6. □

We conclude by bounding the stabilization time of Forest-Prune-Repair, which captures the time required for all amoebots to rejoin non-faulty trees once the last crash failure has occurred. We note that our bound does not directly depend on the number of crash failures f, but rather on the number of non-crashed amoebots m removed from non-faulty trees as a result of the crash failures.

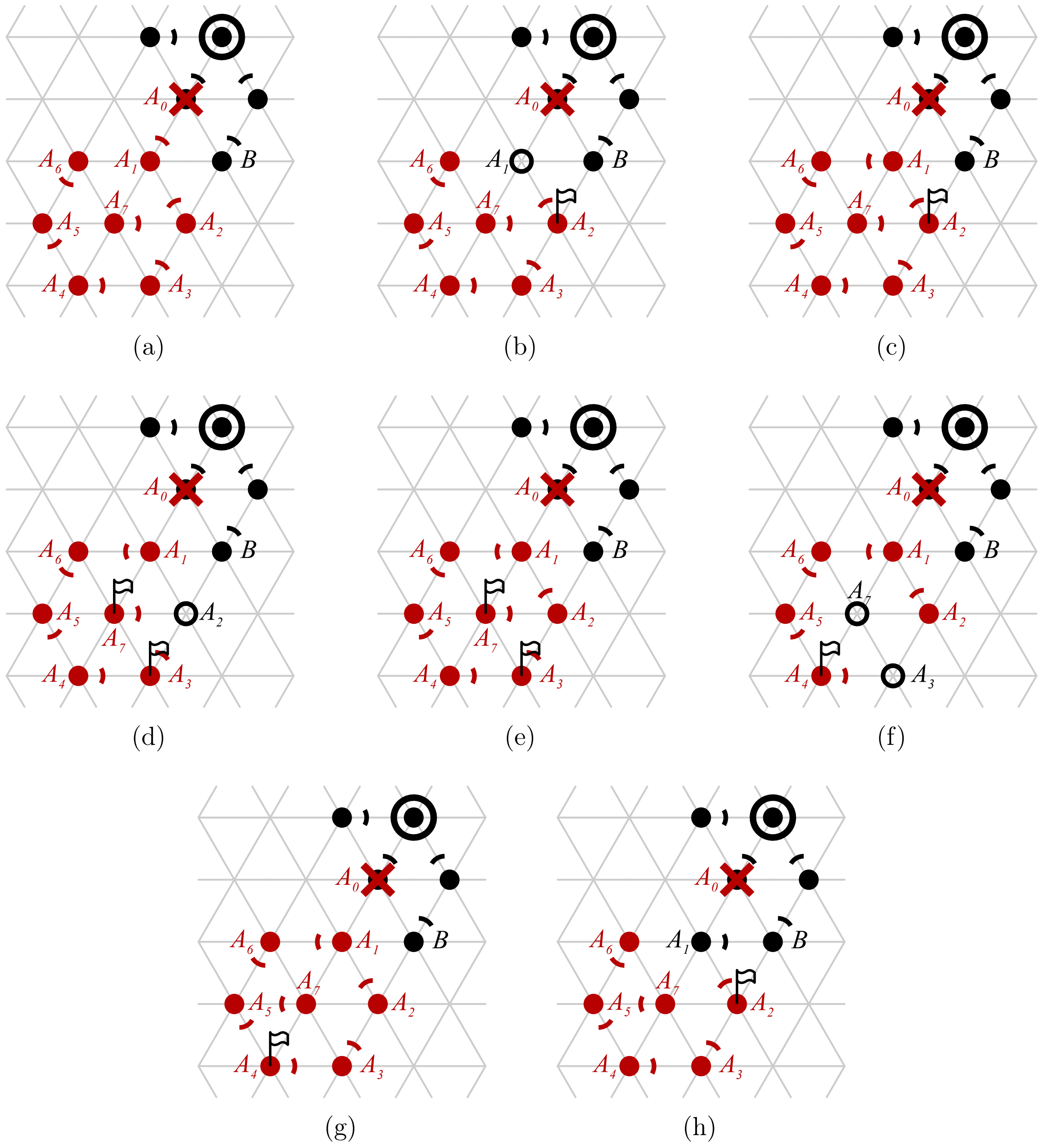


Figure 11. Chase Cycles in Forest-Prune-Repair. (a) A_0 crashes, removing the non-crashed amoebots in red from their non-faulty tree (in black) and disconnecting them from the root amoebot with access to external energy (shown with a black ring). (b) A_1 sees that its parent is crashed and prunes itself (black circle), setting its child's prune flag. (c) A_1 then chooses one of its descendants A_6 as its new parent, creating a chase cycle. (d)–(g) A_2 and A_3 do the same, continuing the chase cycle. (h) Later, A_1 chooses B as its parent, rejoining $\mathcal{F}^*$ and breaking the chase cycle.

Theorem 4.5.4. *Suppose $f < n$ amoebots crash (where n is the number of amoebots in $\mathcal{S}$), yielding faulty trees $\mathcal{F}'$. If no other amoebots crash, all $m = |\mathcal{F}'|$ non-crashed amoebots rejoin $\mathcal{F}^*$ in $\mathcal{O}(m^2)$ sequential rounds in the worst case.*

Proof. If $m = 1$, then by Lemma 4.5.1 and the connectivity assumption, all non-crashed neighbors of the single non-crashed amoebot $A \notin \mathcal{F}^*$ must be in $\mathcal{F}^*$. By Lemma 4.5.2, A will be pruned in one round; A will then choose a neighbor in $\mathcal{F}^*$ as its new parent in its next activation. So A rejoins $\mathcal{F}^*$ in $\mathcal{O}(1) = \mathcal{O}(m^2)$ rounds.

Now suppose $m > 1$. Again by Lemma 4.5.1 and the connectivity assumption, there must exist a non-crashed amoebot $A \in \mathcal{F}'$ with a neighbor in $\mathcal{F}^*$. By Lemma 4.5.2, A will be pruned in at most m rounds since the depth of A in its faulty tree can be at most the total number of amoebots in faulty trees. Amoebot A will then choose a new parent from among its eligible neighbors; if it chooses any neighbor in $\mathcal{F}'$ as its new parent, it will again be pruned in at most another m rounds by Lemma 4.5.2. By Lemma 4.5.3, A will in the worst case need to repeat this process 6 times before choosing a neighbor in $\mathcal{F}^*$ as its new parent. Thus, A rejoins $\mathcal{F}^*$ in $\mathcal{O}(m)$ rounds. This leaves $m - 1$ non-crashed amoebots in $\mathcal{F}'$ needing to rejoin $\mathcal{F}^*$. By the induction hypothesis, these amoebots rejoin $\mathcal{F}^*$ in $\mathcal{O}((m-1)^2)$ rounds, so we conclude that all m non-crashed amoebots in $\mathcal{F}'$ will rejoin $\mathcal{F}^*$ in $\mathcal{O}(m^2)$ rounds. □

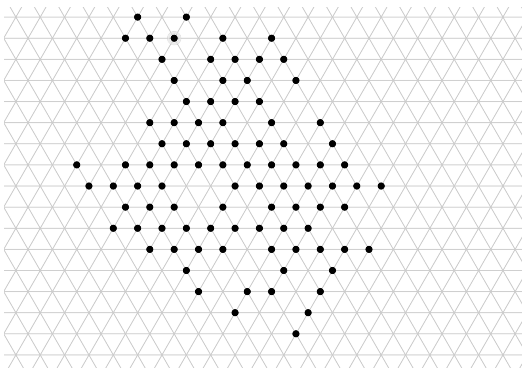
4.5.3 Algorithm Composition

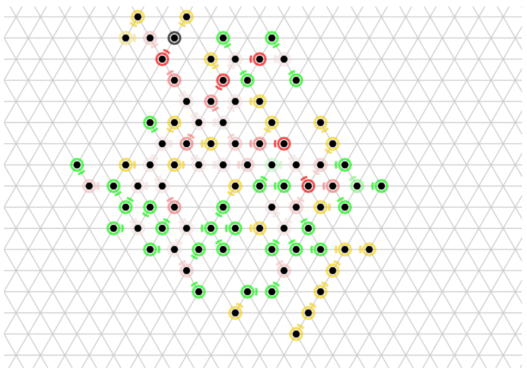
We ultimately envision Energy-Sharing as a subprocess that is executed continuously, handling the energy demands of higher level algorithms for the system's self-organizing behaviors. In particular, if every action of an amoebot algorithm was assigned an energy cost, Energy-Sharing must supply each amoebot with sufficient energy to meet these costs. However, many amoebot algorithms involve amoebot movements that would necessarily disrupt the spanning forest $\mathcal{F}$ maintained by ENERGY-SHARING for energy routing and communication. Just as was the case for crash failures (Section 4.5.1), this necessitates a protocol for repairing $\mathcal{F}$ as amoebots move, disconnecting from existing neighbors and gaining new ones.

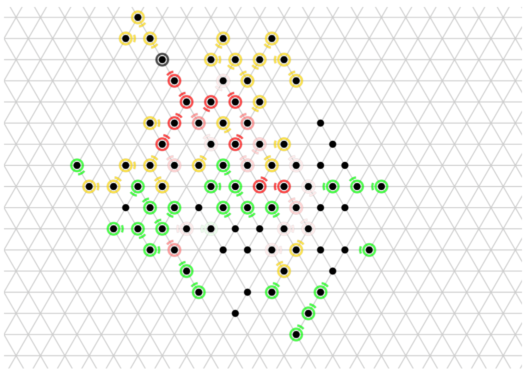
We can repurpose FOREST-PRUNE-REPAIR to address moving amoebots with a simple modification. In this setting, instead of an amoebot initiating the pruning of its subtree if it detects that its parent has crashed, it initiates the pruning of its subtree and additionally prunes itself (unless it is an energy root) whenever it moves according to the higher level algorithm. The rest of FOREST-PRUNE-REPAIR stays the same with the pruning broadcast dissolving the subtree and the resulting idle amoebots rejoining elsewhere.

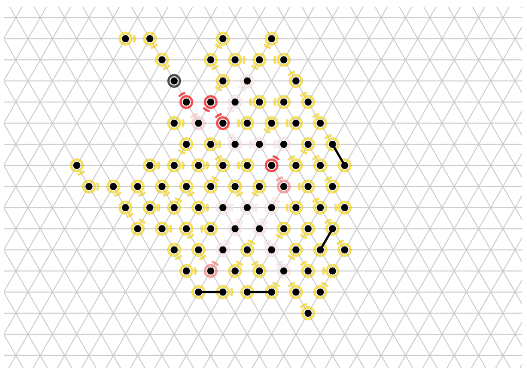
With this modification in place, ENERGY-SHARING can be composed with any amoebot algorithm $\mathcal{A}$ requiring energy distribution so long as (*i*) the battery capacity κ is at least as large as the demand of the most energy-intensive action in $\mathcal{A}$, and (*ii*) $\mathcal{A}$ maintains system connectivity at all times (this is sufficient to satisfy the connectivity assumption of Section 4.5 since no amoebots actually crash). Note that $\mathcal{A}$ need not satisfy the root-reliability assumption; since each root is not actually crashing when it moves, the system maintains its access to external energy sources so long as it remains connected.

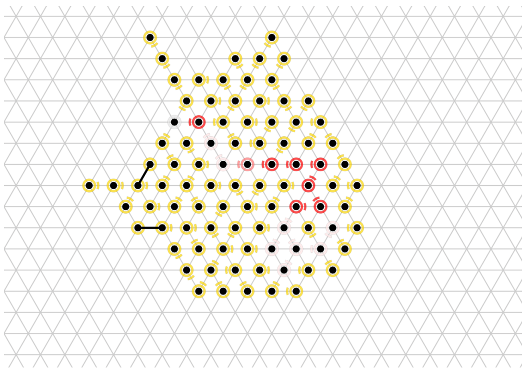
Actions required by algorithm $\mathcal{A}$ are handled in the usage phase of ENERGY-SHARING. If some amoebot A has an action to perform according to algorithm $\mathcal{A}$, then if A has sufficient stored energy and is not inhibited, it spends the energy and performs the action; otherwise, it foregoes its action this activation. For example, Figure 12 shows ENERGY-SHARING composed with the algorithm for *basic shape*


(a) $t = 0$ seq. rounds


(b) $t = 200$


(c) $t = 500$


(d) $t = 1000$


(e) $t = 1500$

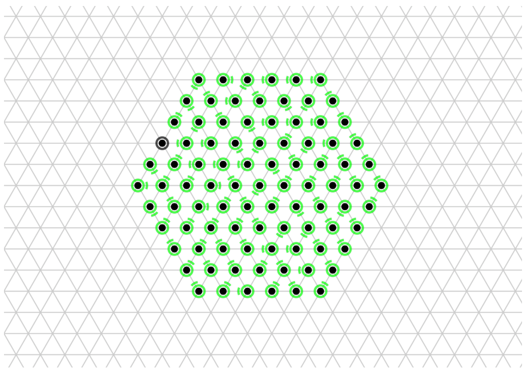
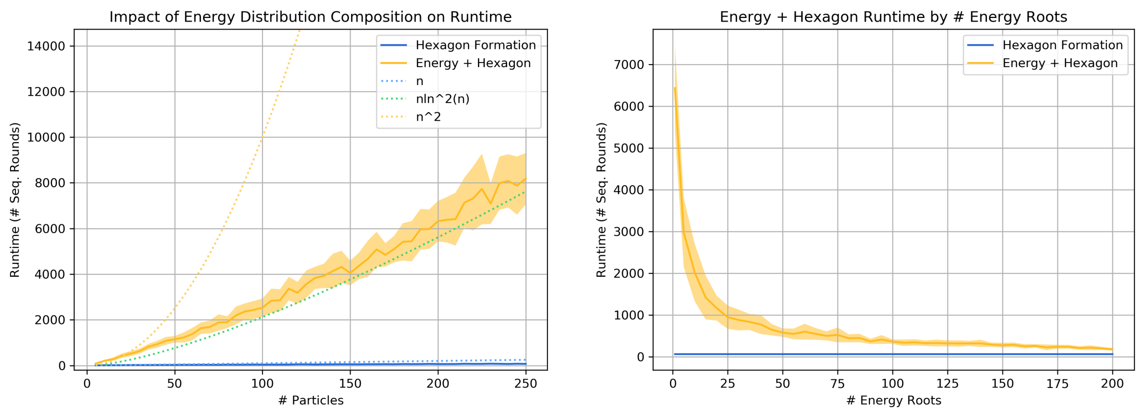

(f) $t = 2000$

Figure 12. Simulation of Energy-Sharing Composed with Basic-Shape-Formation. In particular, hexagon formation on 91 amoebots is composed with Energy-Sharing with one root, $\kappa = 10$, $\alpha = 1$, and action demand $\delta(\cdot, \cdot) = 5$. The communication structure is maintained by Forest-Prune-Repair. Amoebot color and parent directions are visualized with respect to Energy-Sharing, as in Section 4.4. The energy root (shown in black) moves according to the shape formation algorithm and need not be centered.

formation [52, 60] forming a hexagon. Theorem 4.3.8 ensures that all n amoebots will meet their energy needs and at least one amoebot will be able to perform an enabled action every $\mathcal{O}(n)$ sequential rounds. By Theorem 4.5.4, any disruption to the communication structure caused by actions involving movements will be repaired in $\mathcal{O}(m^2)$ sequential rounds, where m is the number of amoebots severed from the communication structure. Thus, Energy-Sharing will not impede the progress of $\mathcal{A}$ but — according to our proven bounds — may add significant overhead to its runtime. However, we observe reasonable performance in practice: for example, since hexagon formation terminates in $\mathcal{O}(n)$ rounds, our proven bounds suggest that the composed algorithm could terminate in time $\mathcal{O}(n^2)$ or worse but Figure 13a demonstrates an overhead that appears asymptotically sublinear. With the addition of more energy roots, the composed algorithm is dramatically faster, approaching the runtime achieved without energy constraints (see Figure 13b).

4.5.4 Using Energy For Reproduction

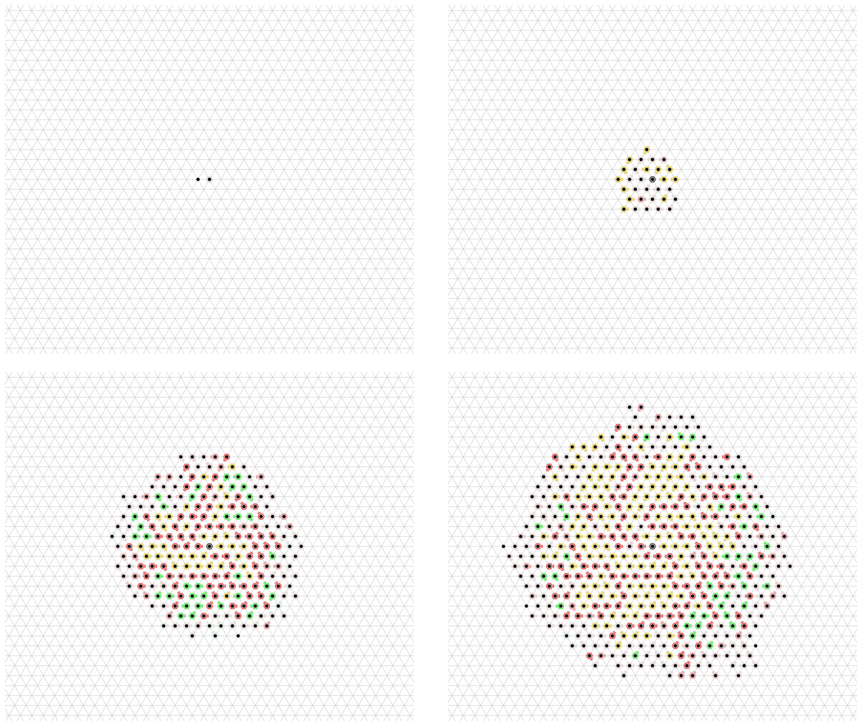
Our goal in this work was to meet the energy demands of fixed-sized amoebot systems as they execute algorithm actions. One could also consider using energy for system growth via *reproduction*, mimicking the bacterial biofilms that inspired our algorithm. Supposing an amoebot A has sufficient energy and is adjacent to some unoccupied position u, a reproduction action would split A into two (analogous to cellular mitosis), yielding a new amoebot A' occupying u. In preliminary simulations (see Figure 14), we obtain behavior that is qualitatively similar to the biofilm growth patterns observed by Liu and Prindle et al. [133, 163]; in particular, the use of communication and inhibition leads to an oscillatory growth rate. However, our



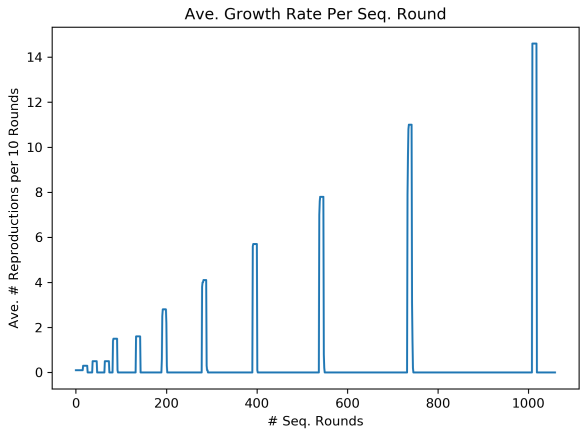
(a) (b)

Figure 13. Runtime Experiments for Algorithm Composition. Runtime experiment results for the composition of Energy-Sharing with the basic shape formation algorithm. Each experiment was repeated 20 times; average runtime is shown as a solid line and standard deviation is shown as an error tube. (a) Runtimes of the basic shape formation algorithm alone (blue) vs. when it is composed with Energy-Sharing (yellow) as a function of system size. Asymptotic runtime bounds are shown as dotted lines; the composed algorithm tracks most closely with $\mathcal{O}(n \log^2 n)$. (b) Runtimes of the composed algorithm for a system of 200 amoebots as a function of the number of energy roots in the system. With more energy roots, the composed algorithm approaches the runtime of basic shape formation with no energy constraints.

oscillations have an amplitude and period that increases with time while the biofilms' have relatively constant amplitude and period. This is due to the fact that the supply of energy remains constant in our simulated system while it scales with the periphery of the biofilms. Further work is needed to formally characterize our algorithm's behavior for these growing, dynamic systems.



(a) $t = 5$, 100, 550, and 1025 seq. rounds



(b)

Figure 14. Simulation of ENERGY-SHARING with Reproduction. (a) The system is initialized as a single root amoebot and uses the same parameters as the previous simulations. After 1025 sequential rounds, the system has grown to 507 amoebots. (b) The growth rate, shown here as the number of reproduction actions per round averaged over a 10-round sliding window, has an oscillating pattern: each recharging period is followed by a rapid burst of growth.

Chapter 5

LEADER ELECTION

Leader election is a fundamental and well studied problem in distributed computing in which exactly one member of the system must irreversibly declare itself the leader (e.g., by setting a dedicated bit in memory). The ability of a leader to break symmetry and coordinate the system via broadcasts makes it a powerful tool for distributed algorithms, and the domain of programmable matter is no exception. For population protocols, Angluin et al. showed that any semilinear predicate — i.e., any stable global property of the system — can be efficiently computed by a population with a constant number of states per agent when a leader is given, though it is unknown if this problem can be solved efficiently without a leader [9]. Only recently was a time- and space-optimal leader election protocol discovered [22]. In the tile assembly and nubot models for molecular self-assembly, leader-like seed structures are used ubiquitously for shape formation [39, 40, 41, 173, 194]. Finally, many amoebot algorithms assume or elect a leader, including those for shape formation [58, 62, 65, 66, 95], object coating (Chapter 6, [55, 61]), and convex hull formation (Chapter 7, [53]).

The Improved-Leader-Election algorithm presented in this chapter is historically the second leader election algorithm for the (geometric, sequential) amoebot model [54], subsuming the original Leader-Election algorithm [60]. Both algorithms utilize randomization and the geometry of the triangular lattice to break symmetry, electing a leader for any connected amoebot system in which the amoebots have common chirality. Leader-Election is (i) described at a high level, lacking local rules for each amoebot's execution, (ii) analyzed using a simplified treatment of time, and (*iii*) shown to achieve its linear runtime bound only in expectation. Improved-Leader-Election, on the other hand, is a fully local and distributed algorithm that elects a leader in $\mathcal{O}(L)$ sequential rounds w.h.p., where L is the length of the outer boundary of the system. This w.h.p. guarantee applies to both the algorithm's correctness and its runtime.

Algorithm	**Det.**	**Weak Sched.**	**Allows Holes**	**Removes Chirality**	**Static**	**Leaders Elected**	**Runtime**
Leader-Election [60]	No	No	Yes	No	Yes	1	$\mathcal{O}(L^*)$ exp.
Improved-Leader-Election	No	No	Yes	No	Yes	1, whp.	$\mathcal{O}(L)$ whp.
Di Luna et al. [65, 66]	Yes	Yes	No	Yes	Yes	$k \leq 3$	$\mathcal{O}(n^2)$
Gastineau et al. [92]	Yes	No	No	No	Yes	1	$\mathcal{O}(n)$
Bazzi and Briones [21]	Yes	Yes	Yes	No	Yes	$k \leq 6$	$\mathcal{O}(n^2)$
Emek et al. [78]	Yes	No	Yes	Yes	No	1	$\mathcal{O}(Ln^2)$

Table 6. Comparison of Amoebot Algorithms for Leader Election. Updated from the tables in [21, 78]. Algorithms are organized chronologically by first appearance. When $k \in \mathbb{Z}_+$ appears in the number of leaders elected, it refers to the amoebot system being k-symmetric. For the runtime bounds, L^* denotes the length of the longest system boundary, L denotes the length of the system's outer boundary, and n denotes the number of amoebots in the system.

This result spurred a flurry of recent research in the distributed computing community to obtain a leader election algorithm that removed reliance on randomization for symmetry breaking and other strong assumptions. Table 6 summarizes the known leader election algorithms, their assumptions, and their outcomes. These assumptions and outcomes are detailed as follows:

- *Deterministic vs. Randomized.* Randomization is a classical technique for symmetry breaking, but incurs a failure probability (with respect to correctness, runtime, or both) that is not present in deterministic algorithms. The Leader-Election algorithm by Derakhshandeh et al. [60] and the present Improved-Leader-Election algorithm are randomized while the rest are deterministic.

- *Amoebot Actions.* Recall from Section 2.2.4 that amoebots are activated sequentially and, when activated, can read information from their own memories and the memories of their neighbors, perform local computation, write updates to their memories or the memories of their neighbors, and perform at most one movement. The algorithms of Di Luna et al. [65, 66] and Bazzi and Briones [21] are proven to elect a leader even when amoebots' read/compute, write, and move operations are split across multiple (not necessarily consecutive) activations; i.e., they assume a *weak scheduler*.
- *Connectivity and Holes.* Recall from Section 2.3 that an amoebot system is connected if the subgraph of G_Δ induced by the occupied nodes is also connected. A hole in an amoebot system is a maximal, finite connected component of the subgraph of G_Δ induced by the unoccupied nodes. The algorithms of Di Luna et al. [65, 66] and Gastineau et al. [92] can only solve leader election for systems that do not contain holes while the rest work for any connected system. Note that solving leader election under for systems that remain disconnected throughout an algorithm's execution is impossible since the constraint of local sensing and communication prohibits disconnected components from interacting.
- *Chirality.* As described in Section 2.4, the chirality assumption states that all amoebots have a common sense of clockwise direction. For leader election, chirality provides useful geometric information. However, the algorithms of Di Luna et al. [65, 66] and Emek et al. [78] successfully elect leaders without the chirality assumption.
- *Movement.* Emek et al. use amoebots' movement capabilities as a novel way to break symmetry [78]. All other algorithms are static and only use communication to elect their leaders.

- *Number of Leaders Elected.* Due to spatial symmetry in the amoebot system's configuration, some of the deterministic algorithms may elect a constant number of leaders instead of a unique one. In particular, if the amoebot system is k-symmetric, then the algorithm of Di Luna et al. [65, 66] elects $k \in \{1, 2, 3\}$ leaders and the algorithm of Bazzi and Briones [21] elects $k \in \{1, 2, 3, 6\}$ leaders. All other algorithms elect a unique leader.
- *Runtime.* All bounds are given in (fair, sequential) rounds. Recall from Section 2.2.4 that a round ends when each amoebot has been activated at least once. Note that under a weak scheduler, this only guarantees that an amoebot has completed one of reading, writing, or moving per round.

The remainder of this chapter is organized as follows. We give the formal statement of the leader election problem and the assumptions in use in Section 5.1. We then present the Improved-Leader-Election algorithm in Section 5.2 and its correctness and runtime analysis in Section 5.3. Finally, Section 5.4 details several algorithm extensions, including an adaptation that improves the correctness guarantee from w.h.p. to with probability 1.

5.1 The Leader Election Problem

An algorithm solves the leader election problem if for any connected amoebot system of initially contracted amoebots, eventually a single amoebot *irreversibly* declares itself the *leader* (e.g., by setting a dedicated bit in its memory) and no other amoebot ever declares itself to be the leader. The algorithm presented in this chapter assumes geometric space, common chirality, constant-size memory, and the simplified sequential setting (Section 2.2.4). The running time of a leader election algorithm is defined as the number of (fair, sequential) rounds until a leader is declared. Note that an algorithm is not required to terminate for amoebots other than the leader. Variants of the leader election problem are discussed in Section 5.4, including allowing initial configurations with expanded amoebots and requiring termination for all amoebots.

5.2 The Improved-Leader-Election Algorithm

We now present the Improved-Leader-Election algorithm, beginning with an overview of its six *phases*. These phases are not synchronized; i.e., at any point in time, different amoebots may be executing different phases. Furthermore, an amoebot can be involved in the execution of multiple phases simultaneously. The first phase is *boundary setup*. In this phase, each amoebot locally checks whether it is part of a *boundary* of the amoebot system. Only boundary amoebots participate in the leader election. Amoebots occupying a common boundary organize themselves into a directed cycle. The remaining phases operate on each boundary independently.

In the *segment setup phase*, the boundaries are divided into *segments*. Each amoebot flips a fair coin: those that flip heads become *candidates* and compete for leadership whereas those that flip tails become *non-candidates* that assist the candidates in their competition. A segment consists of a candidate and all subsequent non-candidates along the boundary up to (but not including) the next candidate. The *identifier setup phase* assigns a random identifier to each candidate that is stored distributively among the amoebots in the candidate's segment.

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

5.2.1 Boundary Setup

The boundary setup phase organizes the amoebot system into a set of *boundaries*, as in Figure 15a. Let V_A be the set of nodes in G_Δ that are occupied by amoebots, and consider the graph $G_\Delta|_{V \setminus V_A}$ induced by the unoccupied nodes in G_Δ. An *empty region* is a maximal connected component of $G_\Delta|_{V \setminus V_A}$. Let $N(R)$ be the neighborhood of an empty region R in G_Δ; that is, $N(R) = \{u \in V \setminus R : \exists v \in R \text{ such that } (u, v) \in E\}$. Note that by definition, all nodes in $N(R)$ are occupied by amoebots. We refer to $N(R)$ as the *boundary* of the amoebot system corresponding to R. Since V_A corresponds to nodes occupied by a finite set of amoebots, exactly one empty region has infinite size while any others have finite size. The boundary corresponding to the infinite empty region is the unique *outer boundary*, and any boundary corresponding to a finite empty region is an *inner boundary.*

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

Otherwise, if neither of these special cases apply, then amoebot A must be adjacent

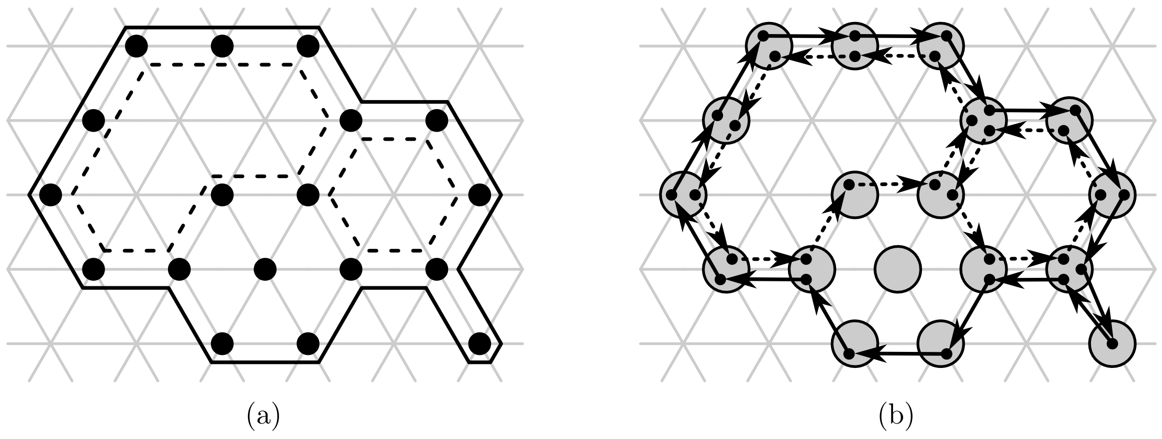


Figure 15. Boundaries and Agents. (a) Boundaries of an amoebot system. The solid line represents the unique outer boundary and the dashed lines represent the inner boundaries. (b) Agents (black dots) of amoebots (gray circles) organized into directed cycles along the boundaries of (a).

to at least one occupied and one unoccupied node. Consider each maximal, consecutive sequence of unoccupied nodes in the neighborhood of A. There can be at most three such *empty sequences* (see Figure 16) for A corresponding to up to three distinct boundaries. However, an amoebot cannot locally decide whether or not two empty sequences correspond to distinct boundaries. Thus, for each empty sequence S in its neighborhood, amoebot A acts as a distinct *agent* a_S that independently executes the remaining phases of Improved-Leader-Election, ensuring that the algorithm's executions on distinct boundaries are isolated.[4] Agent a_S chooses the amoebot occupying the node immediately clockwise (resp., counterclockwise) from S to be its *successor* (resp., *predecessor*). This organizes the set of all agents into disjoint, directed cycles spanning the boundaries of the system (see Figure 15b).

As a consequence of this approach, an amoebot can appear on the same boundary

[4]When activated, amoebot A simply executes Improved-Leader-Election for each of its agents in turn. Note that because the algorithm only requires constant memory per agent and there are at most three agents per amoebot, the entire algorithm respects the amoebot model's constant-size memory constraint.

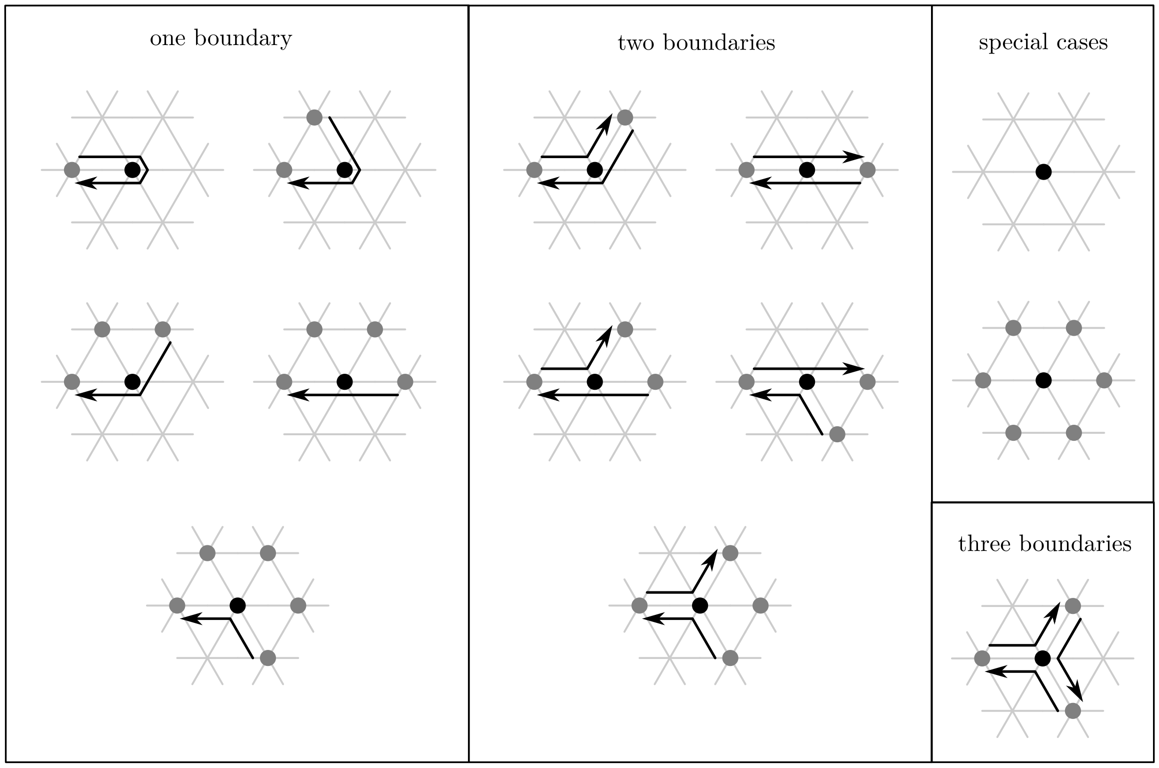


Figure 16. Amoebot Neighborhoods and their Boundaries. All possible results (up to rotation) of the boundary setup phase depending on the amoebot's neighborhood. For each boundary, the depicted arrow starts at the corresponding agent's predecessor and ends at its successor.

up to three times as different agents. While this can be ignored for most of the remaining phases, it will require special consideration during solitude verification.

5.2.2 Segment Setup

This and all subsequent phases operate exclusively on the system's boundaries and does so on each boundary independently. So it suffices to consider a single boundary for the rest of the algorithm description. The segment setup phase divides the boundary into disjoint "segments" as follows. Each agent flips a fair coin: those that flip heads become *candidates* and those that flip tails become *non-candidates.* In the following phases, candidates compete for leadership while non-candidates assist in this competition. A *segment* is a maximal sequence of agents $(a_1, \ldots, a_k)$ such that a_1 is a candidate, a_i is a non-candidate for all $1 < i \leq k$, and a_i is the successor of a_{i-1} for all $1 < i \leq k$. Note that the maximality of each segment implies that the successor of a_k is a candidate. We refer to the segment starting at a candidate c as the *segment of* c, denoted c.seg, and to the number of agents in this segment as its *length*, denoted $|c.\text{seg}|$. In the following phases, each candidate uses its segment as distributed memory.

5.2.3 Identifier Setup

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

Each candidate c generates its random identifier, denoted c.id, by passing a token along its segment in the direction of the boundary (recall token passing from Section 2.4). When an agent receives the token, it chooses a value uniformly at random from $\{0, \ldots, r-1\}$ as its digit in the identifier, where r is a constant that is fixed in the analysis. The resulting identifier c.id is a number with radix r consisting of $|c.\text{seg}|$ digits where c holds the most significant digit and the last agent of c.seg holds the least significant digit.

After generating its identifier c.id, each candidate c creates a copy of c.id that is stored in *reverse digit order* in its segment. This copy is used in the next phase to compare against the identifiers of other candidates. The token that generated c.id is reused in creating the reversed copy as follows. The last agent of c.seg writes its digit in the token and then passes it towards c (the beginning of c.seg), where c stores a copy of that digit. Candidate c then (over)writes its digit in the token and passes it back to the end of the segment, which stores a copy of that digit. This process is repeated by the second-to-last and second agents and so on until c.id is completely copied.[5] Finally, the token is passed to c to demarcate the end of this phase.

5.2.4 Identifier Comparison

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

[5] We deliberately opt for simplicity over speed in this method for creating a reversed copy of the identifier. As we will show in Section 5.3, the runtime of this phase will be dominated by that of identifier comparison, so this simpler approach does not affect the algorithm's asymptotic runtime.

after it creates its own digit token. This ensures that token sequences of distinct candidates remain separated and the tokens within a token sequence maintain their relative order along the boundary.

We now describe the comparison between the identifier of a candidate c and the token sequence of a candidate c'. Figure 17 serves as a running example, but note that c and c' need not be consecutive as depicted. Initially, all agents are *active* and tokens are *inactive* (Figure 17a). Whenever a candidate passes a token into a new segment, that token becomes active (Figure 17b). When an active agent receives an active token, they *match*: the agent stores the result of their digit comparison ($<$, $>$, or $=$) and both the agent and token become inactive (Figure 17c). Since the token sequence of c' stores the digits of c'.id in reverse order, the agent storing the least significant digit of c.id matches with the token storing the least significant digit of c'.id, the agent storing the second-to-least significant digit of c.id matches with the token storing the second-to-least significant digit of c'.id, and so on. Inactive tokens are simply passed on until reaching the next candidate that reactivates it when passing it into the next segment, as already described (Figure 17d).

The delimiter token of the token sequence of c' eventually enters c.seg (Figure 17d). As the delimiter token is passed through c.seg towards c, it sees the results of the previous matched digit comparisons from the least to most significant and updates its record of the overall comparison accordingly (Figure 17e–f). When candidate c eventually receives this delimiter token, the comparison can be decided as follows. If c already matched with a non-delimiter token of c', then $|c.\text{seg}| < |c'.\text{seg}|$ and c withdraws its candidacy. If the delimiter token already matched with a non-candidate agent in c.seg, then $|c.\text{seg}| > |c'.\text{seg}|$ and c remains a candidate. Finally, if c matches with the delimiter token (as in Figure 17g), then we have $|c.\text{seg}| = |c'.\text{seg}|$.

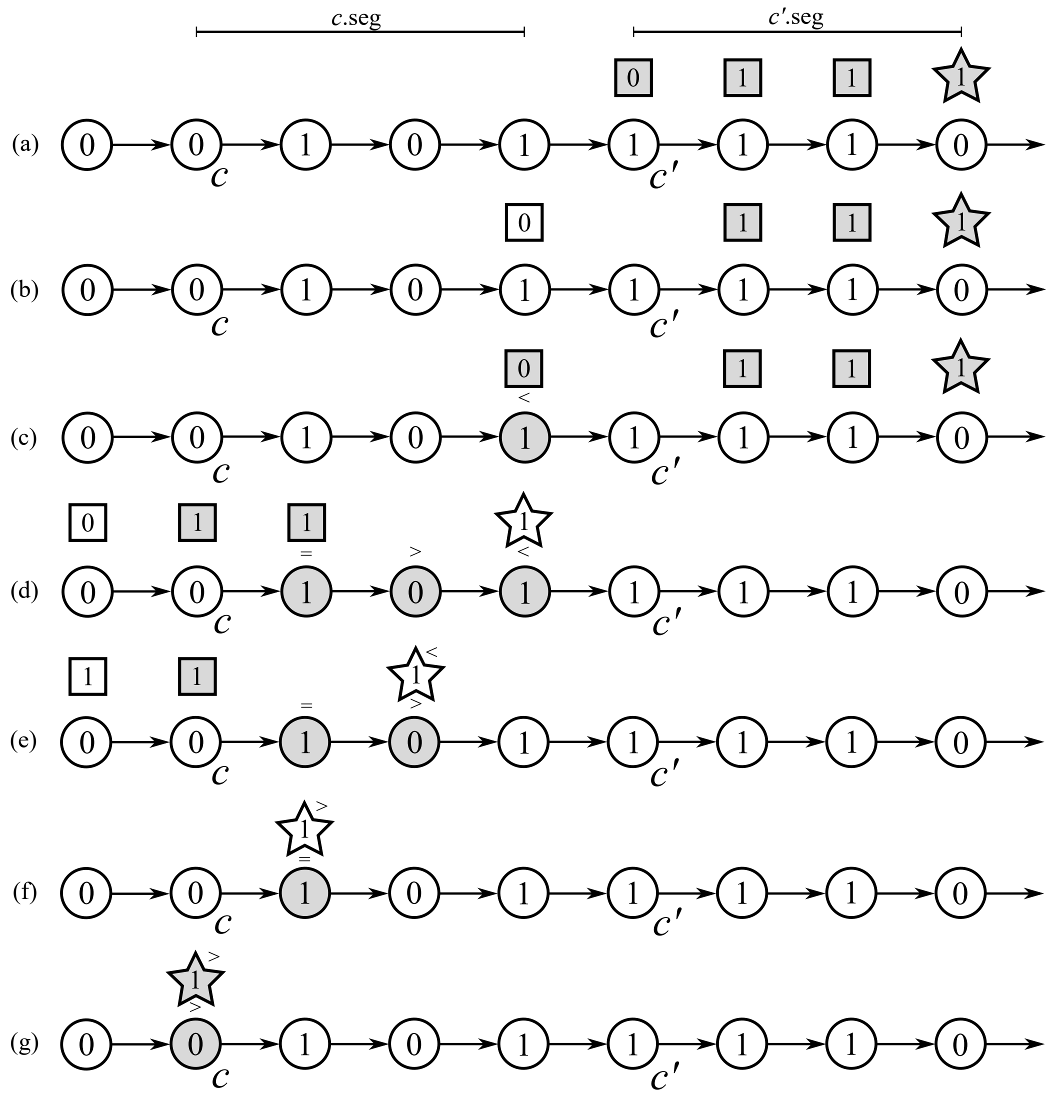


Figure 17. Identifier Comparison. Candidate c with identifier c.id = 0101 is being compared to the token sequence of candidate c' with c'.id = 1110. Inactive elements are shaded while active elements are not. The digit tokens are depicted as squares, while the star depicts the special delimiter token.

In this last case where c and c' have segments of equal length, the record of the overall comparison stored in the delimiter token is used to decide the comparison. If $c.\text{id} < c'.\text{id}$, c withdraws its candidacy. If $c.\text{id} > c'.\text{id}$, c remains a candidate. Finally, if $c.\text{id} = c'.\text{id}$, c may have just compared against its own identifier and thus initiates the solitude verification phase to determine if it is the only remaining candidate on the boundary. This concludes the comparison of c.id and c'.id.

It remains to describe how agents and tokens are reset to prepare for the next comparison. As already mentioned, when candidate c passes the tokens of c' into the next segment, it reactivates them — even if c has withdrawn its candidacy. Candidate c also deletes the comparison result from the delimiter token when it passes it into the next segment. Finally, whenever an agent passes a delimiter token, it reactivates and resets its comparison result (Figure 17e–g).

5.2.5 Solitude Verification

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

A necessary (but insufficient) condition for candidate c to be the only remaining candidate on its boundary is if the next candidate along the boundary, say c', occupies the same node as c. The following algorithm checks this condition. Let the *extended segment* of c refer to the span of agents from c up to but not including c'; note that this not only includes c.seg but also any segments of subsequent candidates that have already withdrawn their candidacy. Treat the directed edges of the boundary cycles as vectors in the two-dimensional Euclidean plane. The next candidate along

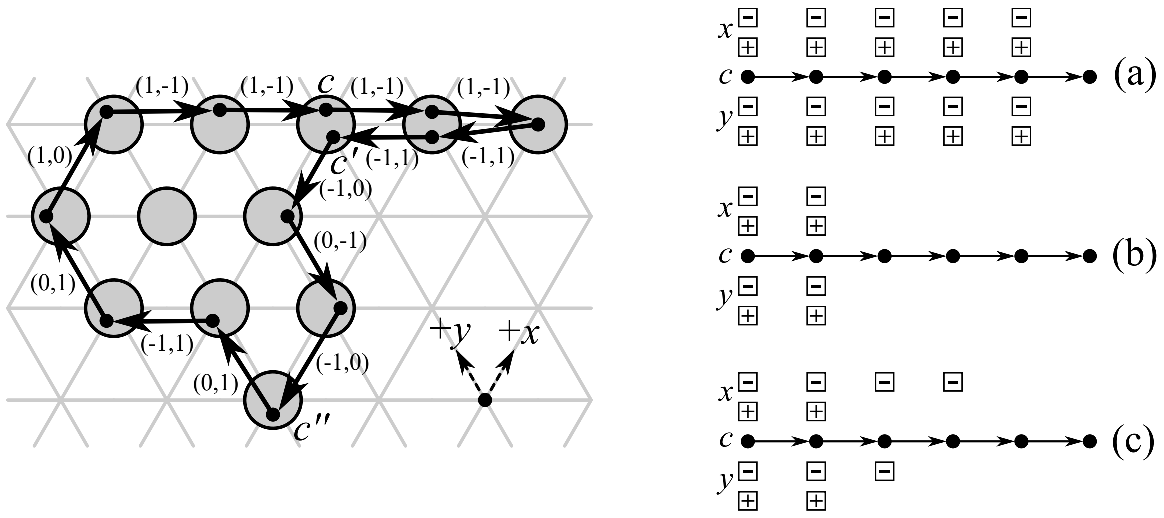


Figure 18. Solitude Verification. The vector interpretation of the system's outer boundary from the perspective of candidate c is shown on the left. The logical positions of the positive and negative tokens after they have settled are shown on the right, for the situation where the only remaining candidate(s) is/are (a) c, (b) c and c', and (c) c and c''.

the boundary occupies the same node as c if and only if the sum of the vectors corresponding to the boundary edges in the extended segment of c is 0. To decide if this is the case in a local manner, c defines a local two-dimensional coordinate system (e.g., as in Figure 18) and uses the following token passing scheme to generate and sum the x- and y-components in parallel. We only describe this scheme for the x-components since the scheme for the y-components is analogous.

First, candidate c creates an *activation token* and passes it in the direction of the cycle towards the next candidate. Whenever the activation token is passed to the right (resp., left) with respect to the locally defined x-axis, it creates a *positive token* (resp., *negative token*) that is passed back towards c. Positive and negative tokens move independently of each other, but cannot overtake tokens of the same type. Each agent can hold at most two tokens of each type.

Once these tokens reach c or cannot move any closer to c, they become *settled*. This is detected locally as follows. Each positive (or negative) token stores a bit specifying whether or not it is settled; this bit is initially FALSE. When a token is passed to c or to an agent whose predecessor already holds two settled tokens of that type, this bit is set to TRUE. Once all positive tokens are settled, they form a consecutive sequence whose length in tokens equals the magnitude of the $+x$-component in the summed vector; an analogous statement holds for the negative tokens (Figure 18a–c).

When the activation token reaches the next candidate, it reverses its movement back towards c, staying behind any unsettled positive or negative tokens. Once they have settled, deciding whether the vectors sum to 0 can be done in a local manner: the vectors corresponding to the extended segment of c sum to 0 if and only if the length of the positive and negative token sequences are equal; i.e., if the last settled tokens in these sequences are held by the same agent and that agent holds the same number of positive and negative tokens. For example, this is the case in Figure 18a–b but not in Figure 18c. Thus, the activation token simply observes whether or not this is the case and then continues towards c to report the result, deleting all positive and negative tokens on the way. When c receives the activation tokens for both the x- and y-components, it can decide whether the vectors sum to zero and thus whether the next candidate along the boundary occupies the same node.

However, as hinted before, this is not sufficient to decide whether c is the last remaining candidate on the boundary. As described in the boundary setup phase, an amoebot may appear on the same boundary up to three times as different agents. Thus, there may be distinct agents on the same boundary that occupy the same node (as with c and c' in Figure 18b) causing the vectors to sum to 0 despite there being multiple agents remaining. To handle this case, each amoebot assigns a locally unique identifier from $\{1, 2, 3\}$ to each of its agents in an arbitrary way. When the activation token reaches the next candidate, it reads its agent identifier and carries this information back to c. It is easy to see that c is the last remaining candidate on its boundary if and only if the corresponding vectors sum to 0 and the agent identifier stored in the activation token equals the agent identifier of c.

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

5.2.6 Boundary Identification

Once a candidate c determines that it is the only remaining candidate on its boundary, it initiates the boundary identification phase to check if it lies on the unique outer boundary. If so, the amoebot acting as c declares itself the leader; otherwise, c revokes its candidacy. This phase uses the fact that, due to the way agents' predecessors and successors are defined during boundary setup, the outer boundary is oriented clockwise while any inner boundary is oriented counterclockwise (see Figure 15b).

To distinguish between clockwise and counterclockwise oriented boundaries, a candidate c passes a token along its boundary that sums the angles of the turns it takes according to Figure 19, storing the results in a counter α. When the token returns to c, there are two cases: $\alpha = 360°$ for the unique outer boundary, and $\alpha = -360°$ for

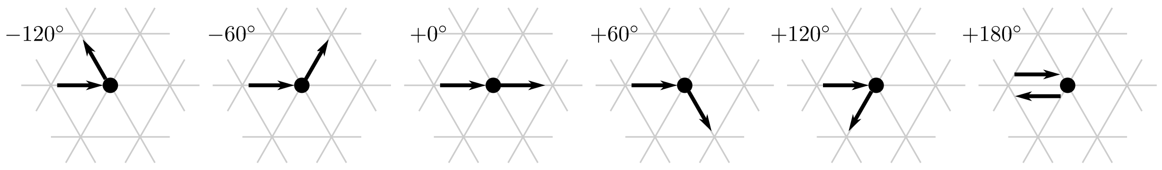


Figure 19. Boundary Identification. The external angle α is determined by summing the depicted turn angles. The incoming and outgoing arrows represent the directions the token enters and leaves an agent, respectively, up to rotation.

any inner boundary. To respect the constant-size memory constraint, we encode α as $k \in \mathbb{Z}$ such that $\alpha = k \cdot 60°$. It is sufficient to store k modulo 5 so that $k = 1$ for the outer boundary and $k = 4$ for an inner boundary, requiring only three bits of memory.

5.3 Analysis

We now first analyze the correctness of Improved-Leader-Election. Recall from Section 5.1 that an algorithm solves the leader election problem if exactly one amoebot irreversibly declares itself the leader. The boundary identification phase ensures that no agent on an inner boundary can ever declare itself the leader, so it suffices to focus on agents on the outer boundary. By the identifier comparison and solitude verification phases, a candidate on the outer boundary will only declare itself the leader if its identifier is strictly greater than the identifier of every other candidate on the outer boundary. The following sequence of lemmas establishes that there exists such a candidate with high probability. We begin with a lower bound on the length of the outer boundary.

Lemma 5.3.1. *Let n be the number of amoebots in the system and L be the length of the outer boundary, i.e., the number of agents in the cycle spanning the outer boundary. Then $L \geq \sqrt{n}$.*

Proof. Define a coordinate system by choosing an arbitrary node of G_Δ as the origin and orienting the $+x$-axis to the right and the $+y$-axis to the up-right. Label the nodes $V_A \subset V$ occupied by amoebots with their (x, y) coordinates, and let $x_{min} = \min\{x : (x, y) \in V_A\}$ and $x_{max} = \max\{x : (x, y) \in V_A\}$. Define y_{min} and y_{max} analogously. The *bounding box* of the amoebot system is the minimum area parallelogram containing all nodes of V_A, which is given by the parallelogram with corners $(x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{min})$, and (x_{max}, y_{max}). Since this bounding box contains all n amoebots, its area must satisfy $(x_{max} - x_{min})(y_{max} - y_{min}) \geq n$, implying that at least one of its sides must be of length at least $\sqrt{n}$. W.l.o.g., suppose $x_{max} - x_{min} \geq \sqrt{n}$. Then for each x-coordinate in $\{x_{min}, \ldots, x_{max}\}$ there exists an amoebot with a maximal y-coordinate; note that this amoebot must be on the outer boundary. Therefore, we conclude that the length of the outer boundary is at least as long as $x_{max} - x_{min} \geq \sqrt{n}$. □

The segment setup phase partitions the outer boundary into segments. We give a probabilistic lower bound on the length of the longest segment.

Lemma 5.3.2. *For sufficiently large n, the segment setup phase produces a segment on the outer boundary of length at least $(\log_2 n)/4$, w.h.p.*

Proof. Let a_1 be any agent on the outer boundary and label the remaining agents as $(a_1, a_2, \ldots, a_L)$ such that a_i is the successor of a_{i-1} for all $1 < i \leq L$. Partition a into subsequences of length $\ell = (\log_2 n)/4$, resulting in $k = \lfloor L/\ell \rfloor \geq \lfloor \sqrt{n}/\ell \rfloor$ subsequences by Lemma 5.3.1. Since the algorithm terminates in the boundary setup phase if $n = 1$, we can assume $n \geq 2$, implying that $\ell > 0$ and k is well-defined.

Let $a^{(i)} = (a_{(i-1)\ell+1}, \ldots, a_{i\ell})$ be the i-th subsequence of a, for $1 \leq i \leq k$. Let E_i be the event that all agents in $a^{(i)}$ flip tails during the segment setup phase. Then

$\Pr[E_i] = (1/2)^\ell = n^{-1/4}$. Since the events E_i are independent, we have:

$$\Pr\left[\bigcap_{i=1}^{k} \overline{E_i}\right] = \left(1 - n^{-1/4}\right)^k \leq \left(1 - n^{-1/4}\right)^{\lfloor 4\sqrt{n}/\log_2 n \rfloor}$$

Since $\lfloor x \rfloor > x/4$ for any $x > 1$ and $4\sqrt{n}/\log_2 n > 1$ for any $n \geq 2$, we have $\lfloor 4\sqrt{n}/\log_2 n \rfloor > \sqrt{n}/\log_2 n$. So:

$$\Pr\left[\bigcap_{i=1}^{k} \overline{E_i}\right] < \left(1 - n^{-1/4}\right)^{\sqrt{n}/\log_2 n}$$

Rewriting $\sqrt{n}$ as $n^{1/4} \cdot n^{1/4}$ in the exponent and applying the well-known inequality $(1 - 1/x)^x \leq 1/e$ yields:

$$\Pr\left[\bigcap_{i=1}^{k} \overline{E_i}\right] < \left(\left(1 - n^{-1/4}\right)^{n^{1/4}}\right)^{n^{1/4}/\log_2 n} \leq e^{-n^{1/4}/\log_2 n}$$

Thus for sufficiently large n, there exists a subsequence of $\ell = (\log_2 n)/4$ agents that all flip tails and become non-candidates in the segment setup phase w.h.p. It remains to show that there is at least one agent on the outer boundary that flips heads and becomes a candidate. By Lemma 5.3.1, this holds with probability $1 - (1/2)^L \leq 1 - (1/2)^{\sqrt{n}}$. Therefore, we conclude that for sufficiently large n, there exists a segment on the outer boundary of length at least $(\log_2 n)/4$, w.h.p. $\square$

We can now analyze the probability of there being a unique candidate on the outer boundary with the greatest identifier.

Lemma 5.3.3. *For sufficiently large n, there exists a candidate c^* on the outer boundary such that $c^*.id > c.id$ for all candidates $c \neq c^*$ on the outer boundary, w.h.p.*

Proof. Let C be the set of all candidates on the outer boundary, $M \subseteq C$ be the candidates with maximal segment length, and c^* be some candidate with the greatest identifier. Since longer identifiers are defined to be greater than shorter identifiers, we must have $c^* \in M$ and $c^*.\mathrm{id} > c.\mathrm{id}$ for all $c \in C \setminus M$. It remains to show that $c^*.\mathrm{id} > c.\mathrm{id}$ for all $c \in M \setminus \{c^*\}$. This is the case if the identifier of c^* is unique.

By definition, the identifiers of the candidates in M all consist of the same number of digits, which by Lemma 5.3.2 must be at least $(\log_2 n)/4$, w.h.p. Each digit is chosen independently and uniformly at random from $\{0, \ldots, r-1\}$, where r is a fixed constant of our choice. Thus, the probability that $c^*.\mathrm{id} = c.\mathrm{id}$ for any candidate $c \in M \setminus \{c^*\}$ is at most $r^{-(\log_2 n)/4}$. Applying the union bound shows that the probability there exists such a candidate is at most $(|M| - 1) \cdot r^{-(\log_2 n)/4} < n^{1-(\log_2 r)/4}$. So we conclude that c^* is the unique candidate with the greatest identifier, w.h.p., as claimed. □

We can now prove the correctness of the algorithm.

Theorem 5.3.4. *The* Improved-Leader-Election *algorithm solves the leader election problem, w.h.p.*

Proof. We must show that eventually a single amoebot irreversibly declares itself to be the leader and no other amoebot ever does so. We need only consider agents on the outer boundary since the boundary identification phase prohibits agents on inner boundaries from causing their amoebots to declare themselves as leaders. Once every amoebot has finished the boundary setup phase, every agent has finished the segment setup phase, and every candidate has finished the identifier setup phase, Lemma 5.3.3 shows that there is a unique candidate c^* on the outer boundary with the greatest identifier, w.h.p. Thus, c^* will not withdraw its candidacy during the identifier comparison phase. In contrast, every other candidate $c \neq c^*$ on the outer boundary eventually withdraws its candidacy because the token sequence of c^* eventually traverses $c.\mathrm{seg}$. Such agents cannot cause their amoebots to become leaders. So c^* will eventually trigger solitude verification when it is the last remaining candidate on the outer boundary because the token sequence of c^* eventually traverses c^*.seg while c^* is not already performing solitude verification. After verifying that it is the last remaining candidate, c^* executes the boundary identification phase and determines that it lies on the outer boundary. It then instructs its amoebot to declare itself the leader of the amoebot system. □

Next, we analyze the runtime of Improved-Leader-Election. Recall from Section 5.1 that the running time of a leader election algorithm is defined as the number of (sequential) rounds until a leader is declared. As with the correctness proofs, it suffices to analyze the outer boundary.

Each amoebot can complete the boundary setup and segment setup phases on its first activation since they only involve computation based on local neighborhood information. Since each amoebot is activated at least once per round, every amoebot completes these first two phases after a single round. Candidate agents then initiate the identifier setup phase.

Lemma 5.3.5. *A segment of length ℓ completes the identifier setup phase in $\mathcal{O}(\ell^2)$ sequential rounds.*

Proof. Recall that in the identifier setup phase, a segment's candidate creates a token that is passed (i) through the segment to assign digits to each agent, (ii) back and forth through the segment to create a reversed copy of the identifier, and (iii) back to the candidate to signal the end of the phase. The amoebot whose agent is holding this token is guaranteed to be activated at least once per round, so this token must be passed at least once per round since there are no other tokens blocking it. Thus, we can upper bound the runtime of this phase by the length of the token's trajectory.

For a segment of length ℓ, the token is passed at most $\mathcal{O}(\ell^2)$ times, so we conclude that the identifier setup phase completes in at most $\mathcal{O}(\ell^2)$ rounds. $\square$

To bound the number of rounds required for all segments on the outer boundary to complete the identifier setup phase, we bound the maximal length of a segment.

Lemma 5.3.6. *The length of any segment on the outer boundary is $\mathcal{O}(\log n)$, w.h.p.*

Proof. For any constant $k \in \mathbb{R}_+$, the probability that an agent becomes a candidate with a segment of length at least $k \log_2 n$ is at most $(1/2)^{k \log_2 n} = n^{-k}$. Since there are n amoebots in the system and each amoebot corresponds to at most 3 agents, there are at most $3n$ agents on the outer boundary. Applying the union bound shows that the probability of there being a segment of length at least $k \log_2 n$ is at most $3n^{1-k}$, which proves the lemma. $\square$

Combining Lemmas 5.3.5 and 5.3.6 yields the following corollary.

Corollary 5.3.7. *All candidates on the outer boundary complete the identifier setup phase after $\mathcal{O}(\log^2 n)$ rounds, w.h.p.*

In the identifier comparison phase, each segment generates a token sequence of digit tokens that are passed against the direction of the boundary for comparison with other segments' identifiers. Each agent holds at most two tokens that it forwards in first-in, first-out order whenever the next agent (in this case, its predecessor) can hold an additional token. Agents forward at most one token per activation.

We want to bound the time required for all digit tokens to progress some length along the outer boundary. However, since digit tokens can block one another's progress, this analysis is not as simple as it was for the identifier setup phase (Lemma 5.3.5). In the following lemma, we make use of a dominance argument (Remark 1) comparing the sequential execution of the identifier comparison phase with a parallel execution in which all tokens move in lock step.

Lemma 5.3.8. *Let T be the first round at the start of which every agent on the outer boundary has created its digit token. Then at the beginning of round $T + i$, for all $i \in \mathbb{N}$, each digit token on the outer boundary has been passed at least i times.*

Proof. Consider a parallel execution in which the digit tokens are passed in lock step, beginning from the configuration at which each digit token is held by the agent that created it. In one parallel round, each digit token is forwarded once; thus, every agent stores exactly one token per round in the parallel execution. For a digit token t, let $p_i(t)$ denote the number of times token t has been passed in the parallel execution by the beginning of parallel round i. Analogously, let $s_i(t)$ be the number of times t has been passed in the sequential execution by the beginning of sequential round $T + i$.

We argue by induction on $i \in \mathbb{N}$ that $s_i(t) \geq p_i(t)$ for all digit tokens t. If $i = 0$, then all digit tokens t are held by the agents that created them in both executions, so $s_i(t) = p_i(t)$. So suppose $i > 0$ and consider any digit token t. If $s_i(t) > p_i(t)$, then:

$$s_{i+1}(t) \geq s_i(t) \geq p_i(t) + 1 = p_{i+1}(t),$$

and thus we have $s_{i+1}(t) \geq p_{i+1}(t)$ as desired.

Suppose instead that $s_i(t) = p_i(t)$. For the sequential execution to make at least as much progress in sequential round $T + i$ as the parallel execution does in parallel round i, t must be passed at least once. The only reason this would not happen is if t is blocked by another digit token in the sequential execution. So consider the digit token t' that was created by the agent just ahead of the one that created t. Since all digit tokens move in lock step in the parallel execution, we have $p_i(t) = p_i(t')$. Combining this fact with the induction hypothesis and the assumption that $s_i(t) = p_i(t)$ yields:

$$s_i(t') \geq p_i(t') = p_i(t) = s_i(t)$$

Thus, t' has been passed at least as many times as t has by the start of sequential round $T + i$ and must be at least one agent ahead. By an analogous argument, the digit token t'' following t' must be at least two agents ahead of t in the sequential execution. Since the digit tokens preserve their order, there are no other agents between t, t', and t''. So at the start of sequential round $T + i$, we have: (*i*) t is the next token to be passed by its agent and (*ii*) the agent that t should be passed to is only holding one token, namely t'. Therefore, since every amoebot (and thus every agent) is activated at least once per sequential round, t will be passed in sequential round $T + i$. $\square$

We also analyze the runtime of the solitude verification phase using a dominance argument. Recall that the extended segment of a candidate c is the span of agents from c up to but not including the next non-withdrawn candidate in the direction of the boundary; this includes c.seg and the segments of any subsequent candidates that withdrew their candidacy in the identifier comparison phase.

Lemma 5.3.9. *An extended segment of length ℓ completes the solitude verification phase in $\mathcal{O}(\ell)$ rounds.*

Proof. W.l.o.g., consider the execution of the solitude verification phase for the x-components; the execution for the y-components occurs in parallel. The activation token is first passed through the extended segment creating positive and negative tokens. Since the activation token is never blocked, this traversal completes in at most ℓ rounds. The activation token is then passed back towards the candidate, but remains behind any unsettled positive and negative tokens. Once all positive and negative tokens have settled, the activation token can be passed unhindered back to the candidate, which again takes at most ℓ rounds. So it remains to bound the number of rounds until all positive and negative tokens have settled.

We once again employ a dominance argument. W.l.o.g., we analyze the time for all positive tokens to settle; the same bound also holds for the negative tokens, which are passed independently. Consider a parallel execution in which the positive tokens are passed in lock step, beginning from the configuration at which each positive token is held by the agent that created it. Let $a_1, \ldots, a_\ell$ be the agents of this extended segment, where a_1 is the candidate. For a positive token t, let $p_i(t)$ denote the index of the agent holding t at the beginning of parallel round i. Analogously, let $s_i(t)$ denote the index of the agent holding t at the beginning of sequential round $T' + i$, where T' is the first sequential round at the start of which all positive tokens have been created.

We argue by induction on $i \in \mathbb{N}$ that $s_i(t) \leq p_i(t)$ for all positive tokens t. If $i = 0$, all positive tokens t are held by the agents that created them in both executions, so $s_i(t) = p_i(t)$. So suppose $i > 0$ and consider any positive token t. If $s_i(t) < p_i(t)$, then:

$$s_{i+1}(t) \leq s_i(t) \leq p_i(t) - 1 \leq p_{i+1}(t),$$

and thus we have $s_{i+1}(t) \leq p_{i+1}(t)$ as desired.

Suppose instead that $s_i(t) = p_i(t)$. We need two observations: first, the order of the positive tokens is maintained; second, whenever an agent holds two positive tokens in the parallel execution, both tokens have reached their final position. If t is the closest positive token to the candidate, it will be passed unhindered in both executions, and the inequality holds. Otherwise, let t' be the next token from t in the direction of the candidate. Then we have one of three cases:

Case 1. If $p_i(t') = p_i(t)$, then by our second observation both t and t' are at their final position. Thus, t is never passed again in either execution since $s_i(t) = p_i(t)$:

$$s_{i+1}(t) = s_i(t) = p_i(t) = p_{i+1}(t)$$

Case 2. If $p_i(t') \leq p_i(t) - 2$, then by our first observation the agent t will be passed to in parallel round i is not holding a token. By the induction hypothesis we have $s_i(t') \leq p_i(t')$ and by our assumption we have $s_i(t) = p_i(t)$, so this is also true of the sequential execution. Thus, t is passed at least once in the sequential execution:

$$s_{i+1}(t) \leq s_i(t) - 1 = p_i(t) - 1 = p_{i+1}(t)$$

Case 3. If $p_i(t') = p_i(t) - 1$, then we distinguish between two subcases. If agent $p_i(t')$ holds two tokens, then t' has reached its final position and thus so has t. By the same argument as for Case 1, $s_{i+1}(t) = p_{i+1}(t)$. So suppose agent $p_i(t')$ only holds t' in the parallel execution. If there is no token t'' following t', then we know that agent $s_i(t)$ holds at most one token in the sequential execution. Otherwise, t'' is at least two agents ahead of t in the parallel execution and thus, by the induction hypothesis, also in the sequential execution. So in any case, agent $s_i(t) + 1$ holds at most one token in the sequential execution and t will be passed at least once:

$$s_{i+1}(t) \leq s_i(t) - 1 = p_i(t) - 1 = p_{i+1}(t)$$

So in all cases, we have $s_i(t) \leq p_i(t)$ for all positive tokens t. Thus, since all positive tokens reach their final position in $\mathcal{O}(\ell)$ parallel rounds, they do so within $\mathcal{O}(\ell)$ sequential rounds as well. Within an additional $\mathcal{O}(\ell)$ sequential rounds, all positive tokens have set their settled bits to TRUE. Therefore, in total, the solitude verification phase completes in $\mathcal{O}(\ell)$ sequential rounds. □

The boundary identification phase is only executed once a candidate determines that it is the last remaining candidate on its boundary. The following lemma upper bound the running time of this phase; recall that L is the length of the outer boundary.

Lemma 5.3.10. *The last remaining candidate on the outer boundary completes the boundary identification phase in $\mathcal{O}(L)$ rounds.*

Proof. The token summing the outer boundary's angles traverses the entire outer boundary. Since there are no other tokens blocking this one and the amoebot whose agent is holding this token must be activated at least once per round, this traversal completes in $\mathcal{O}(L)$ rounds. □

We conclude with the following runtime bound.

Theorem 5.3.11. *The* Improved-Leader-Election *algorithm solves the leader election problem in $\mathcal{O}(L)$ rounds, w.h.p.*

Proof. All amoebots complete the boundary setup and segment setup phases by the end of the first round. By Corollary 5.3.7, all candidates on the outer boundary complete the identifier setup phase in an additional $\mathcal{O}(\log^2 n)$ rounds, w.h.p. By Lemma 5.3.3, there exists a unique candidate c^* on the outer boundary with the greatest identifier, w.h.p. We show that c^* declares itself as the leader after an additional $\mathcal{O}(L)$ rounds.

At the end of one additional round, all candidates have created their digit tokens for identifier comparison. By Lemma 5.3.8, all digit tokens — and particularly the token sequence of c^* — have completely traversed the outer boundary after an additional L rounds. Thus, all candidates $c \neq c^*$ on the outer boundary have either already withdrawn their candidacy by this time or intend to do so after completing their ongoing executions of the solitude verification phase since $c.\text{id} < c^*.\text{id}$. Since the maximum length of an extended segment on the outer boundary is L, any ongoing executions of solitude verification will complete and all candidates waiting to withdraw their candidacy will do so after $\mathcal{O}(L)$ additional rounds by Lemma 5.3.9. If candidate c^* was also executing solitude verification, it will also complete — potentially unsuccessfully, if other candidates had not yet withdrawn their candidacy — within these $\mathcal{O}(L)$ rounds. In the worst case, Lemmas 5.3.8 and 5.3.9 show that in another $\mathcal{O}(L)$ rounds, the token sequence of c^* will once again traverse c^*.seg and trigger an execution of solitude verification that will establish c^* as the last remaining candidate on its boundary. Candidate c^* then executes the boundary identification phase, requiring a final $\mathcal{O}(L)$ rounds by Lemma 5.3.10. So after a total of $\mathcal{O}(\log^2 n) + \mathcal{O}(L) = \mathcal{O}(L)$ rounds, c^* determines that it is the only remaining candidate on the outer boundary and instructs its amoebot to declare itself the leader. □

Theorem 5.3.11 establishes the runtime of Improved-Leader-Election in terms of the number of agents on the outer boundary. Depending on the application, it might be desirable to specify this runtime in terms of numbers of amoebots. We have the following two corollaries.

Corollary 5.3.12. *The* Improved-Leader-Election *algorithm solves the leader election problem in $\mathcal{O}(n_L)$ rounds, w.h.p., where n_L is the number of amoebots on the outer boundary.*

Proof. Each amoebot on the outer boundary corresponds to at most three agents on the outer boundary, so $\mathcal{O}(n_L) = \mathcal{O}(L)$ and the corollary follows from Theorem 5.3.11. □

Corollary 5.3.13. *The* Improved-Leader-Election *algorithm solves the leader election problem in $\mathcal{O}(n)$ rounds, w.h.p.*

Proof. Clearly we have $n_L \leq n$, so the corollary follows from Corollary 5.3.12. □

We note that the $\mathcal{O}(n)$ bound given by Corollary 5.3.13 is quite pessimistic since n_L can be much smaller than n. For example, a solid hexagon or other similarly compact amoebot configurations have $n_L = \mathcal{O}(\sqrt{n})$ amoebots on their outer boundaries.

5.4 Extensions

The Improved-Leader-Election algorithm can be extended to achieve three desirable properties: (i) correctness when the system contains both contracted and expanded amoebots instead of just contracted ones, (ii) termination for all amoebots in the system instead of just for the leader, and (iii) correctness with probability 1 instead of just with high probability. These three extensions can be combined into a single algorithm that satisfies all of the above properties.

5.4.1 Expanded Amoebots

It is straightforward to extend Improved-Leader-Election to handle amoebot systems containing expanded amoebots. An expanded amoebot A simply simulates two distinct contracted amoebots, one for each node it occupies. Whenever A is activated, it simulates the activations of the corresponding contracted amoebots one after another in an arbitrary order, effectively reducing the problem of leader election with expanded amoebots to leader election without expanded amoebots.

Since every amoebot is activated at least once per round, every simulated amoebot is also activated at least once per round. Furthermore, the runtime analysis presented in Section 5.3 holds for arbitrary activation orders. So the analysis holds despite expanded amoebots always activating their two simulated amoebots consecutively.

Therefore, the statements of Theorem 5.3.11 and Corollaries 5.3.12 and 5.3.13 also hold for amoebot systems containing expanded amoebots.

5.4.2 Termination for All Amoebots

In the definition of the leader election problem given in Section 5.1, the leader is the only amoebot for which an algorithm must terminate. Accordingly, amoebots may enter infinite loops while executing Improved-Leader-Election in certain situations. For example, consider an amoebot system with an empty region R of size 1. With constant probability, all six agents on the inner boundary corresponding to R become candidates and create the same one-digit identifier. The identifier comparison phase will never eliminate any of these six candidates, yielding an infinite loop.

Depending on the application, it might be desirable to have a leader election algorithm that is guaranteed to terminate for all amoebots in the system. This can be achieved with the following extension. After the leader has been elected, it broadcasts a *termination signal* through the system. An amoebot receiving this message forwards it to each of its neighbors that have not already received it by writing it into their memories and then terminates. Let $V_A \subset V$ be the set of occupied nodes in G_Δ and let $G_\Delta|_{V_A}$ be the subgraph of G_Δ induced by V_A. The runtime of this termination broadcast is linear in the diameter D of $G_\Delta|_{V_A}$, so after $\mathcal{O}(L + D)$ rounds, a leader has been elected w.h.p. and the execution of Improved-Leader-Election has terminated for all amoebots in the system. We summarize this result in the following corollary.

Corollary 5.4.1. *There is a leader election algorithm that terminates for all amoebots in $\mathcal{O}(L + D)$ rounds, w.h.p.*

Note that L and D are, in general, independent of each other. Nevertheless, L and D are both clearly upper bounded by n, implying the following corollary.

Corollary 5.4.2. *There is a leader election algorithm that terminates for all amoebots in $\mathcal{O}(n)$ rounds, w.h.p.*

5.4.3 Almost Sure Leader Election

The Improved-Leader-Election algorithm elects a leader with high probability, as shown in Theorem 5.3.4. In particular, there is a small but nonzero probability that the algorithm fails to elect a leader, either because every agent on the outer boundary flips tails and becomes a non-candidate in the segment setup phase or because multiple candidates on the outer boundary generate the same greatest identifier in the identifier setup phase. In this final extension of Improved-Leader-Election, we describe how a leader can be elected *almost surely* (i.e., with probability 1) while maintaining a runtime bound of $\mathcal{O}(L)$ rounds, w.h.p.

The main idea of this extension is to run a second leader election algorithm called Almost-Sure-Leader-Election in parallel to Improved-Leader-Election. This algorithm begins by setting up the boundaries as in the boundary setup phase. Each agent is initially a candidate, and the candidates alternate between the following two phases. In the first phase, a candidate flips a coin and sends the result along its boundary to both its preceding and its succeeding candidate. A candidate withdraws its candidacy if it flips tails while both its predecessor and successor flip heads. Note that this competition locally synchronizes competing candidates. The second phase of the algorithm corresponds to the solitude verification phase.

Once a candidate determines that it is the last remaining candidate on its boundary, it executes the boundary identification phase. If the candidate lies on an inner boundary, it withdraws its candidacy. Otherwise, it sends a *stop token* along the outer boundary that stops the execution of Improved-Leader-Election and, at the same time, checks whether Improved-Leader-Election has already established a leader. If so, the candidate withdraws its candidacy; otherwise, it declares itself the leader. We have the following theorem.

Theorem 5.4.3. *The* Almost-Sure-Leader-Election *algorithm elects a leader in* $\mathcal{O}(L)$ *rounds, w.h.p., and eventually solves the leader election problem almost surely.*

Proof. As in previous analyses, it suffices to consider the execution of Almost-Sure-Leader-Election on the outer boundary. While there are still multiple candidates on the outer boundary, the first phase reduces the number of candidates with a probability that is lower bounded by a constant. When the last remaining candidate competes with itself, it cannot receive coin flip results different than what it flipped, so it does not withdraw its candidacy. Thus, eventually only a single candidate remains on the outer boundary almost surely.

This candidate eventually determines that it is the last remaining candidate on the outer boundary via the solitude verification and boundary identification phases. It then interacts with Improved-Leader-Election by sending a stop token along the outer boundary, producing one of three results: (*i*) Improved-Leader-Election establishes a leader and this candidate withdraws its candidacy, (*ii*) Improved-Leader-Election establishes a unique candidate on the outer boundary with the greatest identifier but the stop token reaches its amoebot before it declares leadership, so only Almost-Sure-Leader-Election declares a leader, or (*iii*) Improved-Leader-Election fails to establish a unique candidate on the outer boundary with the greatest identifier so Almost-Sure-Leader-Election eventually declares a leader.

By Theorem 5.3.4 either the first or second case holds with high probability. In the first case, Improved-Leader-Election establishes a leader in $\mathcal{O}(L)$ rounds by Theorem 5.3.11. In the second case, Almost-Sure-Leader-Election stops the execution of Improved-Leader-Election before it can establish a leader. Since Improved-Leader-Election establishes a leader in $\mathcal{O}(L)$ rounds, the stop token must have been created in $\mathcal{O}(L)$ rounds. By an argument analogous to those of Lemmas 5.3.5 and 5.3.10, the stop token requires at most L rounds to traverse the boundary and, therefore, Almost-Sure-Leader-Election establishes a leader in $\mathcal{O}(L)$ rounds. Finally, in the third case Almost-Sure-Leader-Election almost surely establishes a leader eventually. $\square$

Chapter 6

OBJECT COATING

Inspection of bridges, tunnels, wind turbines, and other large civil infrastructure for defects is a time-consuming, costly, and potentially dangerous task. In the future, *smart coating* technology could do the job more efficiently and without putting people in danger. The idea behind smart coating is to form a thin layer of tiny sensors on an object that can measure the condition of the surface — e.g., its temperature, stress and strain, etc. — and then report these metrics to a remote, centralized monitor for human review. Such technology promises a broad range of future applications including automated monitoring and repair of critical infrastructure, isolation and containment of harmful substance leaks, and stopping internal bleeding.

This chapter focuses on *object coating* under the amoebot assuming geometric space, common chirality, constant-size memory, and the simplified sequential setting (Section 2.2.4). Defined formally in Section 6.1, the object coating problem tasks an amoebot system with reconfiguring into even layers such that all amoebots are as close as possible to the given object. In 2017, Derakhshandeh et al. introduced the Universal-Coating algorithm for object coating but only proved its correctness [61]. In this chapter, we present this algorithm's runtime analysis and matching lower bounds [55]. This analysis depends in part on an update to the algorithm's leader election protocol based on Improved-Leader-Election (Chapter 5, [54]), so we first detail the Universal-Coating algorithm in Section 6.2 for completeness. In Section 6.3, we prove that Universal-Coating solves the object coating problem in $\mathcal{O}(n)$ sequential rounds w.h.p. (Theorem 6.3.13), where n is the number of amoebots in the system. We then prove two lower bounds in Section 6.4: local-control algorithms for object coating not only have at least a worst-case linear runtime (Theorem 6.4.1), but also are at least a linear factor slower than an optimal global algorithm in the worst-case (Theorem 6.4.2). These lower bounds imply that the Universal-Coating algorithm is worst-case asymptotically optimal both among local-control algorithms and in a competitive sense against algorithms with global information (Corollary 6.4.3). We conclude with simulation results in Section 6.5 that validate the Universal-Coating algorithm's linear runtime bound and demonstrate that its competitive ratio may be better than linear in practice.

6.1 The Object Coating Problem

Recall that an object is a finite, connected, static set of nodes $O \subset V$ (Section 2.4). For object coating, we further assume objects are *simply connected*, meaning the subgraphs induced by both O and $V \setminus O$ are connected (i.e., object O contains no holes). Let the *distance* between nodes $u, v \in V$, denoted $d(u, v)$, be the number of edges in the shortest (u, v)-path in G_Δ. Similarly, for $v \in V$ and $U \subset V$, we define $d(v, U) = \min_{u \in U} d(u, v)$. The *i-th layer of object* O is defined as $L_i = \{v \in V : d(v, O) = i\}$, i.e., the nodes with distance i to object O. Object O contains a *tunnel of width* i if the subgraph induced by $V \setminus O$ is i-connected but not $(i+1)$-connected.

An instance of the *object coating problem* has the form $(\mathcal{S}, O)$ where $\mathcal{S}$ is a finite amoebot system and $O \subset V$ is the object to be coated. An instance is *valid* if (i) all amoebots in $\mathcal{S}$ are initially contracted and "idle", (ii) object O is simply connected and does not contain tunnels of width less than $2(\lceil n/|L_1| \rceil + 1)$, and (iii) the subgraph induced by $O \cup V_{\mathcal{S}}$ is connected, where $V_{\mathcal{S}} \subset V$ are the nodes occupied by amoebots in

$\mathcal{S}$. The coating of objects with narrow tunnels requires added technical mechanisms that complicate the coating protocol without contributing to its basic concepts, so we exclude such objects for simplicity. A local, distributed algorithm $\mathcal{A}$ *solves* a valid instance of the object coating problem if, when each amoebot of $\mathcal{S}$ executes $\mathcal{A}$ individually to termination, all amoebots of $\mathcal{S}$ are contracted and the nodes $V_{\mathcal{S}}$ occupied by amoebots satisfy:

$$\min_{v \in V \setminus (O \cup V_{\mathcal{S}})} d(v, O) \geq \max_{v \in V_{\mathcal{S}}} d(v, O)$$

This means that in an optimally even coating of O, the closest unoccupied node to O is at least as far from O as the furthest occupied node from O.

6.2 The Universal-Coating Algorithm

In this section, we summarize the Universal-Coating algorithm of Derakhshandeh et al. for object coating under the amoebot model [61]. This algorithm assumes amoebots share a common chirality (i.e., sense of clockwise direction). Table 7 lists the local variables used by this algorithm and Algorithm 9 gives its complete distributed pseudocode. We emphasize that while this algorithm is not an original contribution of this dissertation, a sufficiently detailed summary is necessary to frame the runtime analysis contributed by this research. This dissertation also addresses several small inconsistencies and bugs in the presentation of the algorithm given in [55, 61].

We begin with an overview of the Universal-Coating algorithm's four phases which are integrated seamlessly without any underlying synchronization. The amoebot system is first organized as a spanning forest rooted at amoebots on the object's boundary (i.e., its first layer) in the *setup phase*. In the *complaint coating phase*, amoebots not yet on the boundary "complain" using the spanning forest structure to

Variable	**Notation**	**Domain**	**Initialization**
State	state	IDLE, FOLLOWER, ROOT, RETIRED, MARKER	IDLE
Complaints Stored	complaint	$\mathbb{Z}_2$	0
Checks Stored	check	$\mathbb{Z}_2$	0
Layer Index	layer	$\{-1\} \cup \mathbb{Z}_4$	-1
Parent Direction	parent	$\{-1\} \cup \mathbb{Z}_6$	-1
Movement Direction	dir	$\{-1\} \cup \mathbb{Z}_6$	-1
Marker Direction	marked	$\{-1\} \cup \mathbb{Z}_6$	-1

Table 7. Local Variables for UNIVERSAL-COATING.

advance their ancestors along the boundary while there is still room. This eventually either brings the entire amoebot system into the first layer (at which point the algorithm can terminate) or completely coats the first layer. The amoebots in the first layer participate in the *node election phase* to elect a *marker node* in the first layer. This marker node enables the amoebots in the first layer to determine when it has been completely coated. This triggers the *general layering phase* which ensures layer i of the object is only formed after layer $i - 1$ is complete, for $i \geq 2$.

6.2.1 Preliminaries and Setup

In the UNIVERSAL-COATING algorithm, an amoebot A can be in one of five states denoted A.state $\in \{\text{IDLE}, \text{FOLLOWER}, \text{ROOT}, \text{RETIRED}, \text{MARKER}\}$. The setup phase organizes the amoebot system into a spanning forest $\mathcal{F}$ using the well-established *spanning forest primitive* [52]. Due to amoebots' interactions with the object, we differentiate between more cases here than we did in the exposition of the spanning forest primitive in Chapter 4. Recall that all amoebots are initially idle. On activation, an idle amoebot A adjacent to the object becomes a root with A.state $\leftarrow$ ROOT. Otherwise, if A has a root or follower neighbor B, A becomes a follower with A.state $\leftarrow$

Algorithm 9 UNIVERSAL-COATING for Amoebot A

```
if $A$.state = IDLE then
    if $A$ is adjacent to the object $O$ then
        $A$.state ← ROOT and $A$.layer ← 1.
        $A$ makes its node a candidate and starts node-based IMPROVED-LEADER-ELECTION.
    else if $A$ has a neighbor $B$ with $B$.state ∈ {MARKER, RETIRED} then
        $A$.state ← ROOT.
    else if $A$ has a neighbor $B$ with $B$.state ∈ {ROOT, FOLLOWER} then
        $A$.state ← FOLLOWER, $A$.parent ← direction of $B$, and $A$.complaint ← 1.
else if $A$.state = FOLLOWER then
    if $A$ is contracted then
        if $A$ is adjacent to $O$ or has a neighbor $B$ with $B$.state ∈ {MARKER, RETIRED} then
            $A$.state ← ROOT.
        else FORWARDCOMPLAINT($A$.parent).                          ▷ See Algorithm 10.
    else if $A$ is expanded then
        if $A$ has a contracted child $B$ then $A$ pulls $B$ in a handover.
        else if $A$ has no children nor neighbors $B$ with $B$.state = IDLE then $A$ contracts.
else if $A$.state = ROOT then
    if $A$ is adjacent to $O$ then $A$ participates in node-based IMPROVED-LEADER-ELECTION.
    if $A$ is contracted then
        $A$.layer ← GETLAYER( ) and $A$.dir ← GETDIR( ).             ▷ See Algorithm 10.
        if $A$ has a neighbor $B$ in direction $A$.dir with $B$.state ∈ {MARKER, RETIRED} then
            $A$.state ← RETIRED.
        else if $A$ is adjacent to $O$ then
            if $A$ occupies the elected marker node then
                if $A$.check ≥ 1 then $A$.state ← MARKER.
                else if $A$ has a contracted neighbor $B$ in direction $A$.dir then $B$.check ← 1.
            else
                if the node in direction $A$.dir is unoccupied and $A$.complaint ≥ 1 then
                    $A$ expands toward $A$.dir and $A$.complaint ← $A$.complaint − 1.
                else
                    FORWARDCOMPLAINT($A$.dir).                      ▷ See Algorithm 10.
                    FORWARDCHECK($i$), where $i$ is the counter-clockwise direction.
        else
            if $A$ has a neighbor $B$ with $B$.marked pointing at $A$ then $A$.state ← MARKER.
            else if the node in direction $A$.dir is unoccupied then $A$ expands to $A$.dir.
    else if $A$ is expanded then
        $A$.check ← 0.
        if $A$ has a contracted child $B$ with $B$.state = FOLLOWER then
            $A$ pulls $B$ in a handover.
        else if $A$ has any contracted child $B$ then $A$ pulls $B$ in a handover.
        else if $A$ has no children nor neighbors $B$ with $B$.state = IDLE then $A$ contracts.
else if $A$.state = MARKER then
    if $A$ has a neighbor $B$ in direction $A$.dir with $B$.state = RETIRED then
        if $A$.layer is even then $A$.marked ← one direction clockwise from $A$.dir.
        else $A$.marked ← one direction counter-clockwise from $A$.dir.
        $A$.state ← RETIRED.
```

Algorithm 10 Universal-Coating for Amoebot A: Helper Functions

```
function GetLayer( )
    Let $N$ be the neighbors of $A$ with state MARKER or RETIRED.
    if $A$ is adjacent to the object $O$ then return 1.
    else return $(\min_{B \in N}\{B.\text{layer}\} + 1) \bmod 4$.
function GetDir( )
    Let $N$ be the nodes adjacent to $A$ occupied by $O$ or by neighbors $B$ with $B$.state = RETIRED and $A.\text{layer} = (B.\text{layer} + 1) \bmod 4$.
    Let $i \in \mathbb{Z}_6$ be any direction pointing from $A$ to a node in $N$.
    while $i$ points to a node in $N$ do
        if $A$.layer is even then $i \leftarrow$ the next clockwise direction from $i$.
        else $i \leftarrow$ the next counter-clockwise direction from $i$.
    return $i$.
function ForwardComplaint($i$)
    Let amoebot $B$ be the neighbor in direction $i$ from $A$.
    if $A.\text{complaint} \geq 1$ and $B.\text{complaint} < 2$ then
        $A.\text{complaint} \leftarrow A.\text{complaint} - 1$ and $B.\text{complaint} \leftarrow B.\text{complaint} + 1$.
function ForwardCheck($i$)
    if there is not a neighbor $B$ in direction $i$ from $A$ or $B$ is expanded then $A.\text{check} \leftarrow 0$.
    else if $A.\text{check} \geq 1$ and $B.\text{check} < 2$ then
        $A.\text{check} \leftarrow A.\text{check} - 1$ and $B.\text{check} \leftarrow B.\text{check} + 1$.
```

FOLLOWER and updates its parent pointer to $A.\text{parent} \leftarrow B$. A new follower also generates a *complaint token* to be used in the complaint coating phase. This repeats until all amoebots are either roots or followers, forming a spanning forest $\mathcal{F}$ of trees rooted at amoebots adjacent to the object.

Each root amoebot A maintains a variable $A.\text{dir} \in \mathbb{Z}_6$ indicating the direction it wants to move in. We refer to the unique root amoebot A in each tree with no neighboring amoebot in direction A.dir as the tree's *super-root.* The complaint coating and general layering phases will specify how the root and super-root amoebots choose their directions and move to lead their trees in coating the object. Regardless of phase, followers use the spanning forest primitive to follow their respective parents while maintaining tree connectivity. Consider any follower A. If A is contracted and adjacent to the object or a marker or retired amoebot, it becomes a root. Otherwise, if A is expanded, we distinguish between two cases: if A has a contracted child B,

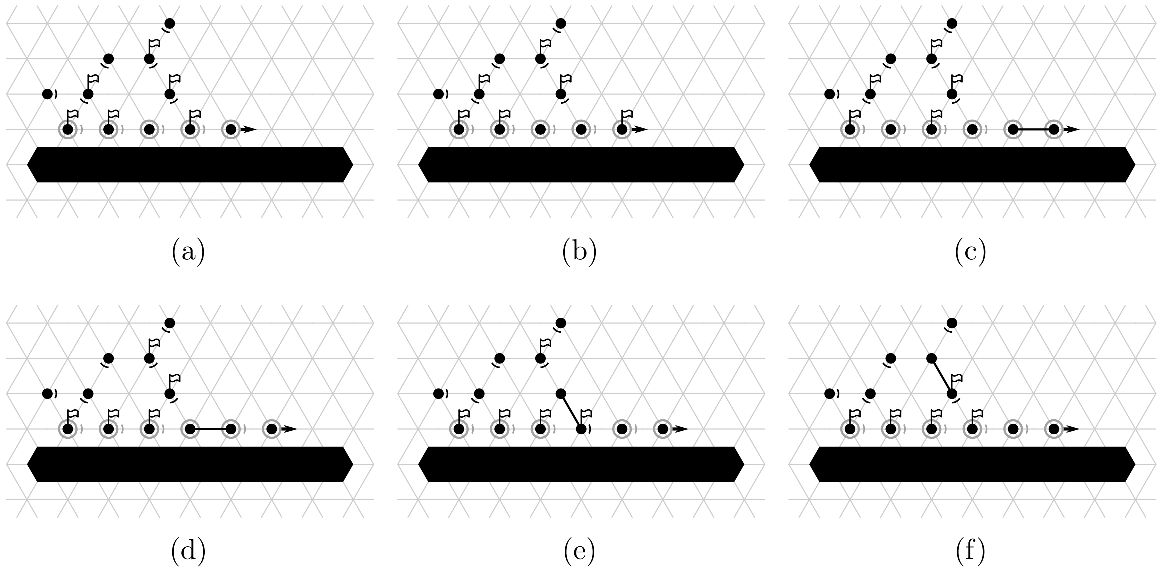


Figure 20. The Complaint Coating Phase. Root amoebots (black dots with gray circles) are coating the object (black polygon) by following the super-root (root with black arrow). Complaint tokens (flags) are forwarded to the super-root to enable movement and allow more followers to join the first layer. For simplicity, this illustration only shows amoebots holding one complaint token at a time though the algorithm specifies a capacity of two.

then A initiates a pull handover with B; otherwise, if A has no children and no idle neighbors, then A contracts.

6.2.2 Complaint Coating

In the complaint coating phase, roots advance clockwise around the first layer (i.e., the object's boundary) so long as there is room to do so and there are follower amoebots in their tree that are not yet on the boundary. This is achieved by each (super-)root A directing A.dir towards the next clockwise node along the first layer, which can be tracked locally.

Recall that whenever an amoebot transitions from idle to follower, it generates a complaint token. Thus, the total number of complaint tokens held by amoebots equals the total number of followers not yet in the first layer (Figure 20a). These complaint tokens are forwarded from contracted children to their parents towards each tree's super-root such that each amoebot holds at most two complaint tokens at a time (Figure 20b). Whenever a super-root A is contracted and holds at least one complaint token, it expands towards A.dir (i.e., into the next position clockwise along the first layer) and consumes one complaint token (Figure 20c). Regardless of complaint tokens, an expanded (super-)root performs a pull handover with a contracted child if it has one — preferring follower children to root children — or simply contracts if it has no children or idle neighbors (Figure 20d). Followers move as described above (Figure 20e–f).

The complaint coating phase terminates in one of two ways. If there are less amoebots in the system than there are positions in the first layer (i.e., $n < |L_1|$), then since super-roots can only expand into new positions when they have a complaint token to consume, eventually all complaint tokens are consumed, all amoebots are contracted and occupy nodes in L_1, and no amoebot ever moves again (see [61] for details). This solves the object coating problem. Otherwise, if $n \geq |L_1|$, the first layer is eventually filled with contracted amoebots. This is detected in the next phase.

6.2.3 Node Election

Root amoebots on the object's boundary participate in the node election phase to elect a marker node. This marker node is used to detect the the complete coating of the first layer by contracted amoebots and to mark the beginning and end of each successive layer. This phase uses the Improved-Leader-Election algorithm (Chapter 5, [54]) to elect a marker node with the key difference that the participating entities are the nodes of the first layer instead of static amoebots. However, since nodes are simply positions in space and cannot actually compete for leadership, the root amoebots occupying the first layer emulate this logical competition as they move. As a consequence of this approach, a marker node cannot be elected until the first layer is completely filled with (possibly expanded) amoebots.

Once the marker node has been elected, any contracted amoebot that comes to occupy it generates a *check-contracted* token that is used to determine when the first layer is completely filled with contracted amoebots. Check-contracted tokens are forwarded along the first layer in a counter-clockwise direction to contracted amoebots only. If a check-contracted token ever reaches an expanded amoebot, it is consumed. Thus, once a check-contracted token returns to the marker node, it is guaranteed that the first layer is filled with contracted amoebots. The contracted amoebot A occupying the marker node becomes the *marker amoebot* with A.state $\leftarrow$ MARKER and sets a pointer A.marked $\in \mathbb{Z}_6$ to a node of the next layer to indicate the starting node of layer 2. When a contracted root amoebot A sees a marker or retired amoebot in direction A.dir, it also becomes retired with A.state $\leftarrow$ RETIRED. This causes the amoebots in the first layer to retire in counter-clockwise order.

6.2.4 General Layering

The general layering phase handles the coating of higher layers (see Figure 21). Each amoebot A tracks its current layer index with variable A.layer $\in \mathbb{Z}_4$ which is stored modulo 4 to respect the constant-size memory constraint. If A is adjacent to the object, then A.layer $= 1$; otherwise, A sets A.layer $\leftarrow (B.\text{layer} + 1) \bmod 4$ for whichever of its neighbors B has the smallest layer index.

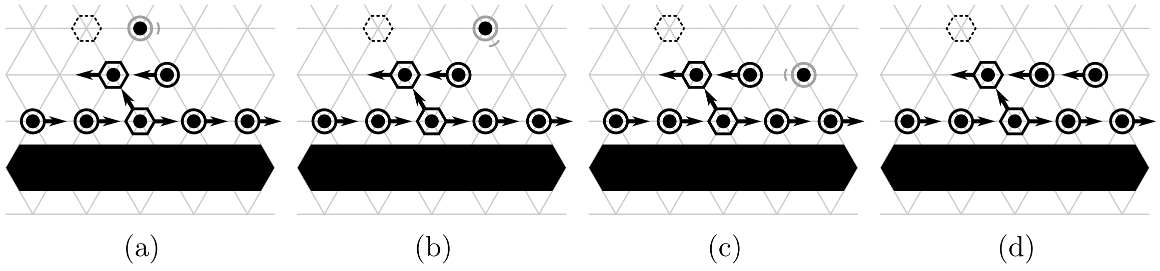


Figure 21. The General Layering Phase. The marker amoebots (black dots with black hexagons) delineate the start and end of each layer. The second layer is not yet complete, so its marker amoebot has not yet set a pointer to the marker node of layer 3 (dotted hexagon). Retired amoebots (black dots with black circles) form the coating. A root (black dot with gray circle) (a) traverses layer 3 clockwise, (b) encounters an unoccupied position in layer 2, (c) enters layer 2 and changes direction, and (d) sees a retired amoebot in its traversal direction and retires.

Root amoebots A traverse the structure of retired amoebots clockwise (if their layer index is odd) and counter-clockwise (if their layer index is even) by updating and following their A.dir directions. If a root changes layers during its traversal (i.e., by entering layer i from layer $i + 1$), it correspondingly updates its layer index and changes its traversal direction. This traversal completes in one of two ways. If a contracted root A has a marker or retired amoebot in direction A.dir, A can also retire. Otherwise, if a contracted amoebot A comes to occupy the node marked by the marker amoebot of the previous layer $A.\text{layer} - 1$, it becomes the marker amoebot of layer A.layer. However, it does not set A.marked for the next layer until A.layer is completely coated, which can be detected locally by ensuring A has retired neighbors in A.layer in both the clockwise and counter-clockwise directions.

6.3 Runtime Analysis

The Universal-Coating algorithm was proven correct by Derakhshandeh et al. in [61]. However, that formulation of the algorithm used the outdated and only partially analyzed LEADER-ELECTION algorithm [60] for its node election phase. When using the IMPROVED-LEADER-ELECTION algorithm for node election, this algorithm inherits its w.h.p. guarantees on both its correctness and runtime. An astute reader may observe that none of the subsequent algorithms for leader election discussed in Chapter 5, Table 6 existed at the time when the UNIVERSAL-COATING algorithm was developed. Unfortunately, though they boast deterministic guarantees on correctness and runtime, none of them are compatible with our present setting. The UNIVERSAL-COATING algorithm requires a competition between the static, cycle-forming nodes of the first layer to produce exactly one elected marker node, but none of the more recent algorithms simultaneously allow holes, remain static, and elect exactly one leader.[6] Thus, the IMPROVED-LEADER-ELECTION algorithm remains state of the art for our present setting.

In this section, we prove that the UNIVERSAL-COATING algorithm solves the object coating problem in $\mathcal{O}(n)$ sequential rounds, w.h.p., where n is the number of amoebots in the system. We begin with a straightforward analysis of the formation of the spanning forest.

Lemma 6.3.1. *All idle amoebots in the system become active and join the spanning forest within n sequential rounds.*

Proof. Recall from Section 6.1 that the subgraph induced by the occupied nodes and the nodes of the object is connected, so as long as there are still idle amoebots in the system, at least one idle amoebot A must be adjacent to the object or have a root or follower amoebot as a neighbor. When A is next activated, it will become a root

[6]The leader election algorithm by Bazzi and Briones [21] could be used in the restricted setting where the object to be coated (and thus the nodes in its first layer) are asymmetric. However, even with this restriction there is not a clear advantage for UNIVERSAL-COATING: its correctness and runtime guarantees become deterministic, but its runtime increases from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$ rounds.

or follower and join the spanning forest. This is guaranteed to happen within one sequential round since every amoebot is activated at least once per round. Thus, at least one idle amoebot joins the spanning forest per round, and there are initially n idle amoebots. $\square$

We next establish two separate *dominance arguments* (Remark 1) regarding the movement of trees in the spanning forest: one for their movement when regulated by complaint tokens (as in the coating of the first layer), and another for their unconstrained movement (as in the coating of higher layers). Together, these will upper bound the runtime of a sequential execution by the runtime of a parallel execution.

Let a configuration C of the amoebot system $\mathcal{S}$ encode the node(s) each amoebot occupies, the complaint token(s) each amoebot holds, and each amoebot's shape and algorithm variables. Let $\mathcal{F}(C)$ denote the forest induced by the nodes occupied by follower and root amoebots in configuration C. The forest $\mathcal{F}(C)$ has three types of directed edges: those from the tail of an expanded amoebot to its head, those from the head of a follower amoebot A to the tail of A.parent, and those from the head of a root amoebot A to the tail of the amoebot occupying A.dir, if one exists. As a technical detail we will use in the proofs, we assume a directed edge from a root amoebot A to the amoebot occupying A.dir is not included until A is first contracted. Since every node in $\mathcal{F}(C)$ has at most one outgoing edge, $\mathcal{F}(C)$ either forms a collection of disjoint trees or a ring of trees.

A *schedule* is a sequence of configurations $(C_0, \ldots, C_t)$. The following definition captures the unconstrained movement of a spanning forest in a parallel execution, as in the coating of the higher layers.

Definition 6.3.2. A *parallel forest schedule* $(C_0, \ldots, C_t)$ is a schedule such that each configuration C_i represents a connected amoebot system, $\mathcal{F}(C_0)$ forms a forest of one or more trees, and each amoebot A follows the unique path P_A defined by $\mathcal{F}$ from its initial position in C_0 as follows. For every $1 \leq i \leq t$, C_i is reached from C_{i-1} using the following for each amoebot A:

1. A expands into the next (adjacent) node in path P_A if it is unoccupied in C_{i-1}.
2. A contracts, leaving a node it occupied in C_{i-1} unoccupied in C_i.
3. A participates in a handover with a neighbor.
4. A occupies the same node(s) in C_{i-1} and C_i.

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible.

Greedy parallel forest schedules have a special property that ensures rapid progress. In particular, we prove that if a greedy forest schedule begins in a "well-behaved" initial configuration, then it remains well-behaved throughout its execution, guaranteeing progress is made every two parallel configurations.

Lemma 6.3.3. *Consider any greedy parallel forest schedule* $(C_0, \ldots, C_t)$*. If every expanded parent in* C_0 *has at least one contracted child, then every expanded parent in* C_i *also has at least one contracted child for all* $1 \leq i \leq t$*.*

Proof. Suppose to the contrary that C_i is the first configuration with an expanded parent A that has all expanded children; note that by supposition $i > 0$. We consider all possible expanded and contracted states of A and its children in C_{i-1} and show that none of them can result in A and its children all being expanded in C_i. First suppose A is expanded in C_{i-1}. Then by supposition, A has at least one contracted child. By Definition 6.3.2, the only move available to a contracted child of A is a handover with A. This yields A contracted in C_i, a contradiction.

Suppose instead that A is contracted in C_{i-1}. Since A is expanded in C_i by supposition, by Definition 6.3.2 it must either expand into the next unoccupied node of its path or perform a handover with its (expanded) parent. Regardless of which occurs, A does not interact with any of its children. Consider any child B of A. If B is contracted, then it cannot perform a movement and thus remains contracted in C_i. Otherwise, if B is expanded, then it either contracts if it has no children or performs a handover with one of its contracted children that it must have by supposition. So all children of A are contracted in C_i, a contradiction.

As a final observation, two trees of the forest $\mathcal{F}(C_{i-1})$ may merge if a super-root A of one tree expands into an unoccupied node adjacent to an amoebot B in another tree. But since A is a root, it only defines B as its parent after it becomes contracted, so the lemma holds in this case as well. $\square$

Now consider any fair sequential activation sequence S; i.e., one in which every amoebot is activated infinitely often. We compare a greedy parallel forest schedule to a *sequential forest schedule* $(C_0^S, \ldots, C_t^S)$ where C_i^S is the amoebot system configuration at the completion of the i-th sequential round in S. For an amoebot A in a configuration C of any type of forest schedule, we define the *head distance* $d_h(A, C)$ of A to be the number of edges in the path P_A from the head of A to the end of P_A. Its *tail distance* $d_t(A, C)$ is defined analogously. Depending on whether A is contracted or expanded, we have $d_h(A, C) \in \{d_t(A, C), d_t(A, C) - 1\}$. For any two configurations C and C' of a forest schedule, we say C *dominates* C' — denoted $C \succeq C'$ — if and only if for all amoebots A we have $d_h(A, C) \leq d_h(A, C')$ and $d_t(A, C) \leq d_t(A, C')$. We have the following lemma.

Lemma 6.3.4. *Given any fair sequential activation sequence S beginning at a configuration C_0^S in which every expanded parent has at least one contracted child, there* *exists a greedy parallel forest schedule* $(C_0, \ldots, C_t)$ *with* $C_0 = C_0^S$ *such that* $C_i^S \succeq C_i$ *for all* $0 \leq i \leq t$.

Proof. Let $M(A)$ be the sequence of movements amoebot A executes according to S. We iteratively construct a greedy parallel forest schedule $(C_0, \ldots, C_t)$ that mimics the sequential execution by greedily selecting moves from these movement sequences. As such, let $M_i(A)$ denote the subsequence of moves not yet selected from $M(A)$ after reaching C_i and let $m_i(A)$ denote the first movement in $M_i(A)$.

We claim that a greedy parallel forest schedule $(C_0, \ldots, C_t)$ can be constructed from configuration $C_0 = C_0^S$ such that C_i is reached from C_{i-1} by executing only the movements of a greedily selected, mutually compatible subset of $\{m_i(A) : A \in \mathcal{S}\}$ for all $1 \leq i \leq t$. Argue by induction on i, the current configuration number. The initial configuration C_0 is trivially obtained, so suppose $i > 0$ and $(C_0, \ldots, C_{i-1})$ is a greedy parallel forest schedule constructed as above. W.l.o.g., let $M_{i-1} = \{m_{i-1}(A_1), m_{i-1}(A_2), \ldots, m_{i-1}(A_k)\}$ be the greedily selected, mutually compatible subset of movements selected to reach C_i from C_{i-1}. Since all movements in M_{i-1} are mutually compatible and have not yet been executed, a greedy parallel forest schedule could certainly execute them; however, suppose to the contrary the schedule is additionally capable of executing some movement $m'(A) \notin M_{i-1}$ from C_{i-1}. We must have that $m'(A) \neq m_{i-1}(A)$; otherwise, we would have $m'(A) = m_{i-1}(A) \notin M_{i-1}$, implying that $m'(A)$ is incompatible with at least one movement in M_{i-1} since M_{i-1} is greedily selected. This leaves the following cases:

Case 1. Movement $m_{i-1}(A)$ is an expansion into an unoccupied node. Then A must be contracted and have no parent in C_{i-1}. This implies that $m'(A) = m_{i-1}(A)$ since there are no other movements A could execute, a contradiction.

Case 2. Movement $m_{i-1}(A)$ is a contraction. Then A must be expanded and have no children in C_{i-1}. This implies that $m'(A) = m_{i-1}(A)$ since there are no other movements A could execute, a contradiction.

Case 3. Movement $m_{i-1}(A)$ is a handover with one of its children B. By the induction hypothesis, all earlier movements in $M(A)$ must have already been executed in reaching C_{i-1}. Thus, B is a child of A in C_{i-1} and must be contracted at the time $m_{i-1}(A)$ is performed. If B is expanded in C_{i-1}, it must have become so before C_{i-1} was reached. But this yields a contradiction: since the parallel forest schedule is greedy, B should have already either contracted if it had no children or performed a handover with a contracted child whose existence is guaranteed by Lemma 6.3.3. Thus, B is also contracted in C_{i-1} and we once again have $m'(A) = m_{i-1}(A)$, a contradiction.

This establishes that a greedy parallel forest schedule $(C_0, \ldots, C_t)$ can be constructed by greedily mimicking the movements of the sequential execution. We conclude by showing that each configuration C_i is dominated by its sequential counterpart C_i^S, the configuration at the completion of the i-th sequential round in S. Argue by induction on i, the current configuration number. Since $C_0 = C_0^S$, we trivially have $C_0^S \succeq C_0$. So suppose $i > 0$ and for all rounds $0 \leq r < i$ we have $C_r^S \succeq C_r$. Consider any amoebot A. Since A executes the same movements in both the sequential and parallel schedules and A decreases either its head distance or tail distance by 1 each time it moves, it suffices to show that A has performed at least as many movements in the parallel schedule up to C_i as it has according to S up to C_i^S.

If A does not perform a movement in reaching C_i from C_{i-1}, then by the induction hypothesis we have $d_h(A, C_i) = d_h(A, C_{i-1}) \geq d_h(A, C_{i-1}^S) \geq d_h(A, C_i^S)$. An analogous argument holds for tail distances. Otherwise, A performs movement $m_{i-1}(A)$ in reaching C_i from C_{i-1}. If A has already performed movement $m_{i-1}(A)$ before reaching C_{i-1}^S in the sequential execution, then clearly $d_h(A, C_i^S) \leq d_h(A, C_i)$ and $d_t(A, C_i^S) \leq$

$d_t(A, C_i)$; in fact, at least one of these inequalities must be strict. Otherwise, $m_{i-1}(A)$ must be the next movement for A to perform according to S and we have $d_h(A, C_{i-1}^S) = d_h(A, C_{i-1})$ and $d_t(A, C_{i-1}^S) = d_t(A, C_{i-1})$. Thus, since A will perform $m_{i-1}(A)$ in parallel round i, we must show that A also performs $m_{i-1}(A)$ in sequential round i.

If $m_{i-1}(A)$ is an expansion, then A must be the super-root of its tree in both C_{i-1} and C_{i-1}^S and thus must be able to expand in both. Similarly, if $m_{i-1}(A)$ is a contraction, then A must have no children in both C_{i-1} and C_{i-1}^S and thus must be able to contract in both. Finally, if $m_{i-1}(A)$ is a handover with a contracted child B, then B must also be contracted in C_{i-1}^S; otherwise, $d_t(B, C_{i-1}^S) > d_t(B, C_{i-1})$, contradicting the induction hypothesis. So the handover can be executed in both the sequential and parallel schedules. Therefore, in any case the sequential schedule can perform the same movement for A as the parallel schedule does, and since the choice of A was arbitrary, we have $C_i^S \succeq C_i$. □

We can show a similar result for the movement of spanning forests that are regulated by complaint flags, as we have in the coating of the first layer. Note that in the following definition for the parallel execution, we reduce each amoebot's complaint token capacity from two to one. This does not apply to the sequential execution and is just a proof artifact that will be useful in Lemma 6.3.6.

Definition 6.3.5. A *parallel complaint-based forest schedule* $(C_0, \ldots, C_t)$ is a parallel forest schedule that, for every $1 \leq i \leq t$, reaches C_i from C_{i-1} using the following for each amoebot A:

1. A does not hold a complaint token in C_{i-1} and one of Property 2–4 of Definition 6.3.2 (a parellel forest schedule) holds.

2. A holds a complaint token t in C_{i-1} and expands into the next (adjacent) node in path P_A if it is unoccupied in C_{i-1}, consuming t.
3. A holds a complaint token in C_{i-1} and passes it to the neighbor specified by $\mathcal{F}(C_{i-1})$ if it either is not holding a complaint token in C_{i-1} or is also passing its complaint token. If multiple complaint tokens are being passed to the same neighbor (as is possible with multiple children passing their tokens to the same parent), preference is given to followers' tokens over roots'.

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible.

To extend the dominance argument between parallel and sequential executions for forest schedules to complaint-based forest schedules, define the *complaint distance* $d_c(t, C)$ of any complaint token t held by an amoebot A in configuration C of a forest schedule as the number of edges in the path P_A from the head of A to the end of P_A. For any two configurations C and C' of a complaint-based forest schedule, we say C *dominates* C' — denoted $C \succeq C'$ — if and only if C dominates C' with respect to head and tail distances and $d_c(t, C) \leq d_c(t, C')$ for all complaint tokens t.

It is also possible to construct a greedy parallel complaint-based forest schedule whose configurations are dominated by their sequential counterparts, as we did for greedy parallel forest schedules in Lemma 6.3.4. Since many of the details are the same, we only highlight the key differences. First, Definition 6.3.5 restricts amoebots to holding at most one complaint token at a time while the sequential executions allow a capacity of two. This enables the sequential executions to not "fall behind" the parallel execution when passing complaint tokens. Similar to the "pipelining" of energy described in Lemma 4.3.5 for Energy-Sharing, Definition 6.3.5 allows an amoebot A in a parallel schedule to pass its complaint token to the next amoebot B even if B also holds a complaint token, so long as B is also passing its token forward in this same parallel round. The sequential executions do not have this luxury of synchronized actions, but the mechanism of buffering up to two complaint tokens at a time allows them to mimic the pipelining in parallel schedules.

Another difference is that a contracted amoebot cannot expand into the next unoccupied node in its path unless it holds a complaint token to consume. However, this is true of both the sequential and parallel executions, so once again the greedy parallel complaint-based forest schedule can be constructed directly from the movements taken by the sequential execution. Moreover, since this restriction can only cause a contracted amoebot to remain comtracted, the conditions of Lemma 6.3.3 are still upheld. Thus, we obtain the following lemma:

Lemma 6.3.6. *Given any fair sequential activation sequence S beginning at a configuration C_0^S in which every expanded parent has at least one contracted child, there exists a greedy parallel complaint-based forest schedule $(C_0, \ldots, C_t)$ with $C_0 = C_0^S$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq t$.*

Together, the dominance argument results of Lemmas 6.3.4 and 6.3.6 imply that for any sequential execution of the first and higher layer coating process, there exists a greedy parallel execution that takes at least as many rounds. Formally, let S be any fair sequential activation sequence and let $S^* = (C_0, \ldots, C_f)$ denote the corresponding dominated greedy parallel schedule, where C_0 is the initial configuration of the amoebot system and C_f is the final coating configuration. In the remainder of this section we determine an upper bound for f which in turn, due to the dominance arguments, also upper bounds the running time of the sequential execution with activation sequence S. Since S was arbitrary, this upper bound serves as a runtime bound for any sequential execution of Universal-Coating, as desired.

To obtain an upper bound on f, the number of parallel rounds required by S^* to coat the first and higher layers, we bound the worst-case time required to coat layer i after layers $1, \ldots, i-1$ have already been coated. We will not differentiate between complaint-based and unconstrained forest schedules since analogous dominance arguments hold for both settings. Though our dominance arguments hold for any (complaint-based) forest schedule, our analysis focuses on a specific class that better supports our layer-by-layer decomposition. A *forest-path schedule* $((C_0, \ldots, C_t), \mathcal{P})$ is a forest schedule $(C_0, \ldots, C_t)$ such that (i) all trees of the forest $\mathcal{F}(C_0)$ are rooted at a path $\mathcal{P} = v_1, v_2, \ldots, v_\ell \subseteq G_\Delta$ and (ii) all amoebots traverse the path $\mathcal{P}$ in the same direction.

We now analyze the number of parallel rounds required by S^* to coat the first layer. For convenience, we assume that there are sufficient amoebots to coat the first layer (i.e., $n \geq |L_1|$) though the proofs could be extended to the setting where $n < |L_1|$. We also assume that the root of a tree generates a complaint token upon its first activation since this can only increase the number of complaint tokens generated. Let $S_1^* = ((C_0, \ldots, C_{t_1}), L_1)$ be the greedy parallel forest-path schedule where $(C_0, \ldots, C_{t_1})$ is a truncated version of S^* (i.e., $t_1 \leq f$) and C_{t_1} is the first configuration of S^* in which all nodes of layer 1 are occupied by contracted amoebots.

Lemma 6.3.7. *Consider any round i of the greedy parallel forest-path schedule S_1^* with $0 \leq i \leq t_1 - 2$. Then within the next two parallel rounds of S_1^*, (i) at least one complaint token is consumed, (ii) at least one complaint flag is passed from an amoebot outside layer 1 to an amoebot in layer 1, (iii) all remaining complaint flags move one amoebot closer to a super-root, or (iv) layer 1 is completely filled, possibly with some expanded amoebots.*

Proof. If layer 1 is completely filled, then (iv) is satisfied. Otherwise, there exists at least one super-root in $\mathcal{F}(C_i)$. We consider several cases:

Case 1. There exists a super-root A in $\mathcal{F}(C_i)$ that holds a complaint token c. Definition 6.3.5 delineates the following possibilities. If A is contracted, then it can expand and consume c in parallel round $i+1$. Otherwise, if A is expanded but has no children, then it can contract in parallel round $i+1$ and then expand and consume c in parallel round $i+2$. Finally, if A is expanded but has children, then by Lemma 6.3.3 it must have a contracted child with which it can perform a handover with in parallel round $i+1$. It can then expand and consume c in parallel round $i+2$. Thus, within two parallel rounds (i) is satisfied.

Case 2. No super-root in $\mathcal{F}(C_i)$ holds a complaint token and there exists a follower amoebot (outside layer 1) that holds a complaint token. Let $A_1, A_2, \ldots, A_k$ denote any sequence of amoebots in layer 1 such that A_j is the parent of A_{j+1} in $\mathcal{F}(C_i)$ for all $1 \leq j < k$, no ancestor of A_1 in $\mathcal{F}(C_i)$ holds a complaint token, only A_k has a follower child that holds a complaint token, and A_j holds a complaint token for all $1 \leq j \leq k$. By Definition 6.3.5, each amoebot in the sequence passes its complaint token to its parent in parallel, allowing for the follower child of A_k holding a complaint token to pass it to A_k. Thus, (ii) is satisfied.

Case 3. No super-root in $\mathcal{F}(C_i)$ holds a complaint token and all complaint tokens are held by amoebots in layer 1. Since there are no followers outside layer 1 holding complaint tokens that take preference over those already in layer 1, all complaint flags are forwarded from child to parent in parallel, satisfying *(iii)*. □

The following lemma uses Lemma 6.3.7 to show that all nodes of layer 1 are occupied by (possibly expanded) amoebots within $\mathcal{O}(|L_i|)$ parallel rounds. It then argues that within another $\mathcal{O}(|L_i|)$ rounds, all amoebots in layer 1 will be contracted.

Lemma 6.3.8. *The greedy parallel forest-path schedule S_1^* completes in $t_1 = \mathcal{O}(|L_1|)$ rounds; i.e., after $\mathcal{O}(|L_1|)$ rounds, layer 1 must be filled with contracted amoebots.*

Proof. Suppose to the contrary that after $8|L_1| + 2$ parallel rounds, there exist unoccupied nodes in layer 1. Then condition (*iv*) of Lemma 6.3.7 was not satisfied by any of these rounds, so one of conditions (*i*), (*ii*), or (*iii*) must have been satisfied every two parallel rounds. Condition (*i*) can be satisfied at most $|L_1|$ times (accounting for at most $2|L_1|$ parallel rounds) since a super-root expands into an unoccupied node of layer 1 each time a complaint token is consumed. Condition (*iii*) can also be satisfied at most $|L_1|$ times (accounting for at most another $2|L_1|$ parallel rounds) since once all complaint flags enter layer 1, they must all reach a super-root after being passed at most $|L_1|$ times. Thus, the remaining $8|L_1| + 2 - (2|L_1| + 2|L_1|) = 4|L_1| + 2$ parallel rounds must satisfy condition (*ii*) at least $2|L_1| + 1$ times, implying that $2|L_1| + 1$ complaint tokens have been passed from follower children to parents in layer 1. Each amoebot can hold at most one complaint token at a time, so at least $|L_1| + 1$ complaint tokens have been consumed by super-roots, implying that the super-roots have collectively expanded into $|L_1| + 1$ unoccupied nodes of layer 1, a contradiction.

So in at most $8|L_1| + 2$ parallel rounds, all nodes of layer 1 are occupied by amoebots, though some may be expanded. If an expanded amoebot in layer 1 does not have a contracted follower child, then once per round it must be able to pull its unique root child in a handover. This transfer of expansion occurs at most $|L_1|$ times before reaching an amoebot A in layer 1 with a contracted follower child B. Since root handovers give preference to follower children, A will pull B and B will then contract in the next two parallel rounds. Therefore, layer 1 will be filled with contracted amoebots after at most $8|L_1| + 2 + |L_1| + 2 = \mathcal{O}(|L_1|)$ parallel rounds. □

Once all nodes in layer 1 are occupied by amoebots, they compete to become the marker node in the node election phase. Recall that this phase uses a node-based variant of the Improved-Leader-Election algorithm (Chapter 5, [54]) for its competition. Thus, by Theorem 5.3.11, we have the following bound.

Lemma 6.3.9. *Once all nodes of layer 1 are occupied by amoebots, a marker node is elected in layer 1 within $\mathcal{O}(|L_1|)$ additional rounds, w.h.p.*

Lemmas 6.3.8 and 6.3.9 show that layer 1 is both filled with contracted amoebots and contains an elected marker node in $\mathcal{O}(|L_1|)$ parallel rounds, w.h.p. The contracted amoebot A occupying the marker node then generates a check-contracted token that is guaranteed to return to A in an additional $\mathcal{O}(|L_1|)$ parallel rounds since all amoebots in layer 1 are contracted and the token is passed once per parallel round. So A becomes the marker amoebot and in each subsequent parallel round, the next (counter-clockwise) amoebot in layer 1 retires. Once all $|L_1|$ contracted amoebots in layer 1 retire, the marker amoebot sets its pointer to indicate the starting node of layer 2 and retires. Combined with Lemma 6.3.1, this yields:

Lemma 6.3.10. *The worst-case number of parallel rounds for greedy parallel forest-path schedule S_1^* to coat layer 1 is $\mathcal{O}(n)$, w.h.p.*

Next, we turn to the higher layers. In particular, we bound the number of parallel rounds required by S^* to coat layer i once layer $i-1$ is coated, for $i > 1$. The following lemma establishes a more general result that we use for this purpose.

Lemma 6.3.11. *Consider any greedy forest-path schedule $((C_0, \ldots, C_t), \mathcal{P})$ with $\mathcal{P} = v_1, v_2, \ldots, v_\ell$ and any k such that $1 \leq k \leq \ell$. If every expanded parent in C_0 has at least one contracted child, then nodes $v_{\ell-k+1}, \ldots, v_\ell$ will be occupied by contracted amoebots within $2(\ell + k)$ parallel rounds.*

Proof. Let A be the super-root closest to v_ℓ and suppose there are at least k root or follower amoebots in C_0 (otherwise, there are insufficient amoebots to occupy k nodes of $\mathcal{P}$). Argue by induction on k, the number of nodes in $\mathcal{P}$ starting from v_ℓ that must be occupied by contracted amoebots. If $k = 1$, then by Lemma 6.3.3 every expanded parent has at least one contracted child in any configuration, so A is always able to either expand into the next unoccupied node of $\mathcal{P}$ if it is contracted or contract in a handover with one of its children if it is expanded. Thus, in at most $2\ell \leq 2(\ell + k) = 2\ell + 2$ parallel rounds, A has moved forward ℓ nodes, is contracted, and occupies its final node $v_{\ell-k+1} = v_\ell$.

Now suppose that $k > 1$ and that nodes $v_{\ell-x+1}, \ldots, v_\ell$ are all occupied by contracted amoebots in at most $2(\ell + x)$ parallel rounds for all $1 \leq x < k$. In particular, we have that nodes $v_{\ell-(k-1)+1} = v_{\ell-k+2}, \ldots, v_\ell$ are all occupied by contracted amoebots in at most $2(\ell + k - 1) = 2(\ell + k) - 2$ parallel rounds and want to show that $v_{\ell-k+1}$ is also occupied by a contracted amoebot within two additional parallel rounds. Let A be the amoebot currently occupying $v_{\ell-k+1}$; such an amoebot must exist because we assumed there were at least k root or follower amoebots and all amoebots follow the unique path $\mathcal{P}$ in a forest-path schedule. If A is contracted in parallel round $2(\ell + k) - 2$ then it remains so in parallel round $2(\ell + k)$ and we are done. Otherwise, if A is expanded in parallel round $2(\ell + k) - 2$, it either has no children or has a contracted child by Lemma 6.3.3. In either case, A can contract in parallel round $2(\ell + k) - 1$, proving the lemma. □

When coating any higher layer $i > 1$, each root amoebot either traverses layer i in the set direction d_i (either clockwise or counter-clockwise) or traverses layer $i+1$ in the opposite direction $\overline{d_i}$ over already retired amoebots in layer i until it finds an unoccupied node of layer i to enter. We bound the worst-case runtime for these two traversals independently in order to upper bound the time to coat layer i after layer $i-1$ is coated. W.l.o.g., let $L_i = v_1, \ldots, v_{|L_i|}$ denote the nodes of layer i in the order they appear starting at the marker node v_1 and proceeding in direction d_i. Let $S_i^* = ((C_{t_{i-1}+1}, \ldots, C_{t_i}), L_i)$ be the greedy parallel forest-path schedule where $(C_{t_{i-1}+1}, \ldots, C_{t_i})$ is the subschedule of S^* where $C_{t_{i-1}}$ is the first configuration of S^* in which all nodes of layer $i-1$ are occupied by contracted, retired amoebots and C_{t_i} is defined analogously. By Lemma 6.3.11, all root movements through layer i in direction d_i complete in $\mathcal{O}(|L_i|)$ parallel rounds; an analogous argument shows that all root movements through layer $i+1$ in direction $\overline{d_i}$ complete in $\mathcal{O}(|L_{i+1}|) = \mathcal{O}(|L_i|)$ parallel rounds. Once all nodes of layer i are occupied by contracted amoebots, these amoebots retire and the marker node for layer $i+1$ is set within an additional $\mathcal{O}(|L_i|)$ parallel rounds. This yields the following runtime bound for individual higher layers.

Lemma 6.3.12. *Starting from configuration $C_{t_{i-1}+1}$ — the first configuration after layer $i-1$ is completely coated — the greedy parallel forest-path schedule S^* coats layer i in $\mathcal{O}(|L_i|)$ additional parallel rounds.*

We conclude our analysis with the following theorem.

Theorem 6.3.13. *The* Universal-Coating *algorithm solves any valid instance $(\mathcal{S}, O)$ of the object coating problem in $\mathcal{O}(n)$ sequential rounds, w.h.p., where n is the number of amoebots in $\mathcal{S}$.*

Proof. Consider any valid instance $(\mathcal{S}, O)$ of the object coating problem and any fair sequential activation sequence S. By Lemmas 6.3.4 and 6.3.6 there exists a corresponding greedy parallel forest schedule $S^* = (C_0, \ldots, C_f)$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq f$. Thus, f is an upper bound on the number of sequential rounds required to solve the given instance and so it suffices to upper bound f.

By Lemma 6.3.10, S^* coats layer 1 with contracted amoebots in $\mathcal{O}(n)$ parallel rounds, w.h.p. Lemma 6.3.12 then shows that once layer $i-1$ has been coated by contracted amoebots, S^* coats layer i in an additional $\mathcal{O}(|L_i|)$ parallel rounds. Let ℓ be the index of the final layer that n amoebots can coat; i.e., $\sum_{j=1}^{\ell-1} |L_j| < n \leq \sum_{j=1}^{\ell} |L_j|$. Then, in total, the worst-case number of parallel rounds needed to coat all ℓ possible layers is:

$$f \leq \mathcal{O}(n) + \sum_{i=1}^{\ell} \mathcal{O}(|L_i|) = c\left(n + \sum_{i=1}^{\ell} |L_i|\right) = \mathcal{O}(n)$$

w.h.p., where $c > 0$ is some sufficiently large constant. Therefore, since f is an upper bound on the sequential runtime, so is $\mathcal{O}(n)$. □

6.4 Lower Bounds

We now turn to lower bounds on the runtime of algorithms for object coating. Let $T_{\mathcal{A}}(\mathcal{S}, O)$ denote the worst-case number of sequential rounds over all possible fair activation sequences required by an algorithm $\mathcal{A}$ to solve an instance $(\mathcal{S}, O)$ of the object coating problem, and let $T_{\mathcal{A}} = \max_{(\mathcal{S},O)}\{T_{\mathcal{A}}(\mathcal{S}, O)\}$ denote the worst-case sequential runtime over all valid problem instances. We begin by proving a linear lower bound on $T_{\mathcal{A}}$ for any local-control algorithm $\mathcal{A}$.

Theorem 6.4.1. *For any local-control algorithm $\mathcal{A}$ for the object coating problem, $T_{\mathcal{A}} = \Omega(n)$, where n is the number of amoebots in $\mathcal{S}$.*

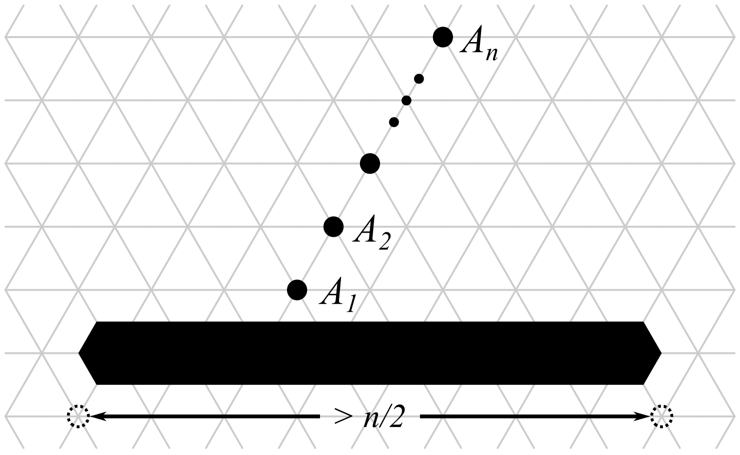


Figure 22. Object Coating Instance for Linear Lower Bound. All n amoebots (black dots) must occupy nodes in the first layer of object O (black polygon) to solve this instance, but amoebot A_n is n nodes away and thus must move at least $\Theta(n)$ times.

Proof. Consider the instance $(\mathcal{S}, O)$ of the object coating problem depicted in Figure 22 where the amoebot system $\mathcal{S}$ forms a single line of amoebots $A_1, \ldots, A_n$ such that only A_1 is connected to the object O. Since $|L_1| > n$, all n amoebots must enter layer 1 to solve this instance. So A_n must move $\Theta(n)$ times since its distance to the object is $d(A_n, O) = n$. Since each amoebot is only guaranteed to be activated once per sequential round, A_n may in the worst-case move at most once per sequential round. Therefore, regardless of algorithm $\mathcal{A}$, we have $T_{\mathcal{A}} \geq T_{\mathcal{A}}(\mathcal{S}, O) = \Omega(n)$. □

A linear lower bound is fairly large and therefore not very helpful to distinguish between different algorithms for object coating, so we instead compare an algorithm's runtime to the best possible runtime. Formally, an algorithm $\mathcal{A}$ is *c-competitive* for some constant $c > 1$ if for any valid instance $(\mathcal{S}, O)$ of the object coating problem,

$$\mathrm{E}\left[T_{\mathcal{A}}(\mathcal{S}, O)\right] \leq c \cdot \mathrm{OPT}(\mathcal{S}, O) + k,$$

where the expectation is taken over an algorithm's random decisions if it is randomized, $\mathrm{OPT}(\mathcal{S}, O)$ is the minimum worst-case number of sequential rounds over all fair

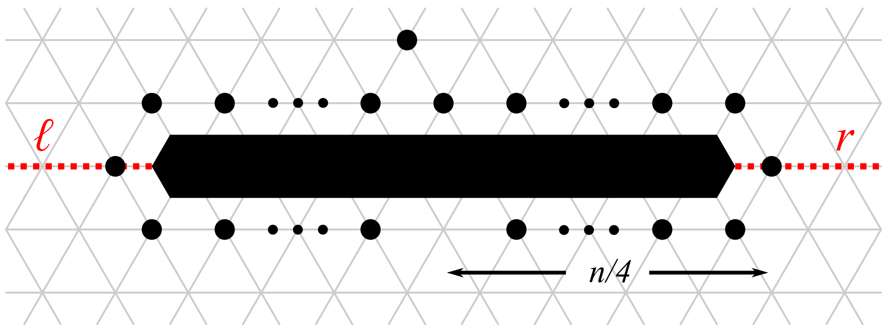


Figure 23. Object Coating Instance for Linear Competitive Ratio. The object O (black polygon) occupies a straight line of nodes in G_Δ while the amoebots (black dots) almost entirely coat its first layer. However, one node of the first layer below O and equidistant from its left and right points is unoccupied while one node of the second layer above O and (nearly) equidistant from its left and right points is occupied.

activation sequences required by any algorithm to solve $(\mathcal{S}, O)$, and $k > 0$ is a constant independent of $(\mathcal{S}, O)$. Unfortunately, a large lower bound also holds for the competitive ratio of any local-control algorithm, even with randomization.

Theorem 6.4.2. *Any local-control algorithm $\mathcal{A}$ for the object coating problem has a competitive ratio of $\Omega(n)$, where n is the number of amoebots in the system.*

Proof. We show that there exists a valid instance $(\mathcal{S}, O)$ of the object coating problem that can be solved by an optimal algorithm in $\mathcal{O}(1)$ sequential rounds but requires any local-control algorithm $\Omega(n)$ times longer, establishing that any local-control algorithm is at best $\Omega(n)$-competitive. Let the object O be a horizontal line of nodes of any finite length and let $\mathcal{S}$ be a system of $|L_1| - 1$ contracted amoebots occupying all of layer 1 except for one node below O equidistant from its sides and one amoebot above O in layer 2 equidistant from its sides (see Figure 23). An optimal algorithm could move the amoebot system as in Figure 24 to solve this instance in $\mathcal{O}(1)$ sequential rounds. Note that this particular optimal algorithm even maintains system-object connectivity throughout its execution, something we do not require in general.

Now consider any local-control algorithm $\mathcal{A}$ for object coating. Define the *left-imbalance at round* i, denoted $\phi_\ell(i)$, as the net number of amoebots that have crossed from the top of O to the bottom on its left side by the start of sequential round i. Analogously, define the *right-imbalance at round* i, denoted $\phi_r(i)$, as the net number of amoebots that have crossed from the bottom of O to the top on its right side by the start of sequential round i. In the worst-case, information and amoebots may only travel a distance of i nodes in i rounds. Thus, for any round $i \leq n/4$, the probability distributions of $\phi_\ell(i)$ and $\phi_r(i)$ are independent. Moreover, no amoebot up to round $n/4$ can determine which side (i.e., ℓ or r) it is closer to since the positions of the gap and layer 2 amoebot are each equidistant from the sides. So for any round $i \leq n/4$, we have $\Pr[\phi_\ell(i) = k] = \Pr[\phi_r(i) = k]$ for any integer k. By round $n/4$, we have the following cases.

Case 1. $\phi_\ell(n/4) = \phi_r(n/4)$. Then the net change in numbers of amoebots on the top and bottom of O is zero, and thus there are more amoebots on the top of O than on the bottom. So there must exist an unoccupied node on the bottom of O, as there was initially, so O has not yet been coated.

Case 2. $\phi_\ell(n/4) \neq \phi_r(n/4)$. From our insights above, we have that for any two integers k_1 and k_2, $\Pr[\phi_\ell(n/4) = k_1 \wedge \phi_r(n/4) = k_2] = \Pr[\phi_\ell(n/4) = k_2 \wedge \phi_r(n/4) = k_1]$. Hence, the cumulative probability of all outcomes where $\phi_\ell(n/4) < \phi_r(n/4)$ is equal to the cumulative probability of all outcomes where $\phi_\ell(n/4) > \phi_r(n/4)$. If $\phi_\ell(n/4) < \phi_r(n/4)$, then there are again more amoebot on the top of O than on the bottom, so O has not yet been coated.

Therefore, the probability that $\mathcal{A}$ has not yet solved the coating problem in $n/4$ rounds is at least $1/2$, and thus $\mathrm{E}[T_{\mathcal{A}}(\mathcal{S}, O)] \geq 1/2 \cdot n/4 = n/8$. Since $\mathrm{OPT}(\mathcal{S}, O) = \mathcal{O}(1)$, this establishes a linear competitive ratio. □

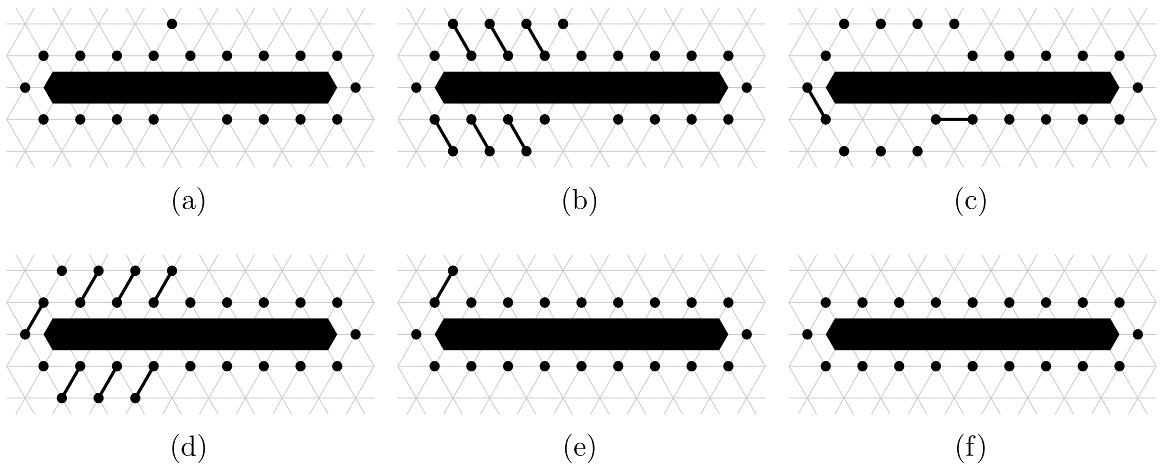


Figure 24. Optimal Algorithm for Object Coating. (a)–(f) show the amoebot system configuration at the start of sequential round $i = 1, \ldots, 6$, respectively. After five sequential rounds, the object is coated. Note that this algorithm solves the given instance in at most five sequential rounds regardless of the fair activation sequence used or the object/system size.

Together, Theorems 6.3.13, 6.4.1, and 6.4.2 yield the following corollary.

Corollary 6.4.3. *Algorithm* Universal-Coating *is worst-case asymptotically optimal and has an optimal competitive ratio, w.h.p.*

6.5 Simulations

We conclude with simulation results that demonstrate that in practice, the Universal-Coating algorithm achieves a sublinear average competitive ratio. Since $\mathrm{OPT}(\mathcal{S}, O)$ is difficult to compute for general instances, we first obtain an appropriate lower bound. Recall that the distance between nodes u and v, denoted $d(u, v)$ is the number of edges in the shortest (u, v)-path in G_Δ; the distance between a node v and a subset of nodes U is $d(v, U) = \min_{u \in U} d(u, v)$. In a slight abuse of notation, we use $d(A, v)$ to denote the distance between the node occupied by A and the node v.

For any valid instance $(\mathcal{S}, O)$ of the object coating problem, let $\mathcal{C}(\mathcal{S}, O)$ denote the set of all coatings that would solve this instance; i.e., $\mathcal{C}(\mathcal{S}, O)$ contains subsets of nodes C such that:

$$\min_{v \in V \setminus (O \cup U)} d(v, O) \geq \max_{v \in U} d(v, O)$$

Given a coating $C \in \mathcal{C}(\mathcal{S}, O)$, let $B(\mathcal{S}, C)$ denote the complete bipartite graph on partitions $\mathcal{S}$ and C. Set the weight of each edge $e = (A, v) \in \mathcal{S} \times C$ to $w(e) = d(A, v)$. Every perfect matching in $B(\mathcal{S}, C)$ corresponds to an assignment of amoebots to positions in a coating of object O. The maximum edge weight in a matching corresponds to the maximum distance an amoebot has to traverse to reach its final (retired) position. Let $M(\mathcal{S}, C)$ be the set of all perfect matchings in $B(\mathcal{S}, C)$. We define the *matching dilation* of instance $(\mathcal{S}, O)$ as:

$$\mathrm{MD}(\mathcal{S}, O) = \min_{C \in \mathcal{C}(\mathcal{S}, O)} \left\{ \min_{M \in M(\mathcal{S}, C)} \left\{ \max_{e \in M} w(e) \right\} \right\}$$

The matching dilation of an instance lower bounds all distances amoebots must traverse to reach their retired positions in a coating. In the worst-case, each amoebot may move at most once per sequential round, so we have $\mathrm{OPT}(\mathcal{S}, O) \geq \mathrm{MD}(\mathcal{S}, O)$. We note that $\mathrm{MD}(\mathcal{S}, O)$ is not a tight bound on $\mathrm{OPT}(\mathcal{S}, O)$. In particular, it only considers the distances between amoebots and their final destinations in the coating but ignores the congestion that may arise as amoebots move there.

Observe that any coating in $\mathcal{C}(\mathcal{S}, O)$ includes nodes from some fixed number of layers ℓ; specifically, it must include all nodes of the first $\ell - 1$ layers (i.e., they are completely coated) and some subset of nodes from the final ℓ-th layer (i.e., the final layer may be partially coated). Depending on the number of amoebots in $\mathcal{S}$ and the length of layer ℓ, there may be prohibitively many possible coatings to iterate through individually in calculating the matching dilation explicitly.[7] We instead

[7]Given an amoebot system $\mathcal{S}$ of n amoebots and an object O, we can determine the final number

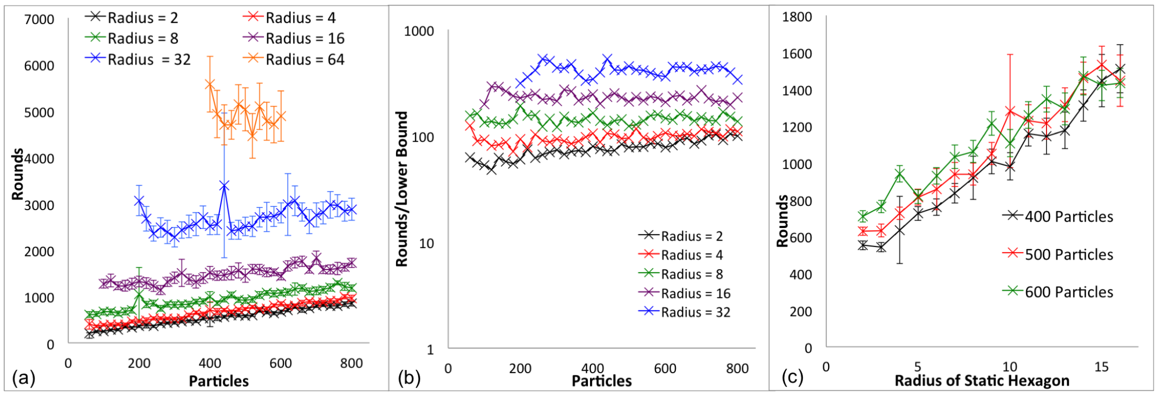


Figure 25. Simulations of the Universal-Coating Algorithm. Statistics are shown for 20 repeated iterations per run, with error bars shown for 95% confidence intervals. (a) The runtime of Universal-Coating in sequential rounds for varying system and object sizes. (b) The ratio of runtime in sequential rounds to the lower bound w^* in log-scale. (c) The relationship between runtime and object size for different system sizes.

efficiently compute a lower bound on the matching dilation by considering assignments of amoebots to *any* position in the first ℓ layers. Formally, we consider a complete bipartite graph with partitions $\mathcal{S}$ and $\bigcup_{i=1}^{\ell} L_i$ where the edge weights are set as described above. We then search for a minimum edge weight w^* such that there exists a bipartite matching saturating $\mathcal{S}$ and $\bigcup_{i=1}^{\ell-1} L_i$ that only uses edges of weight at most w^*. The search for a satisfactory matching given a fixed maximum edge weight can be achieved in polynomial time with standard algorithms for minimum cost matchings, and w^* can be found in a logarithmic number of iterations using binary search. Therefore, since w^* is the smallest value such that there exists a valid coating in $\mathcal{C}(\mathcal{S}, O)$ where no amoebot is further than w^* from its final position, we have that $\text{MD}(\mathcal{S}, O) \geq w^*$.

Using AmoebotSim (https://github.com/SOPSLab/AmoebotSim), we simulated

of layers ℓ. Moreover, we know that $k = n - \sum_{i=1}^{\ell-1} |L_i|$ amoebots are left to coat layer ℓ since the problem definition requires the first $\ell - 1$ layers to be completely filled. Therefore, there are $\binom{|L_\ell|}{k}$ valid coatings in $\mathcal{C}(\mathcal{S}, O)$ that cannot be iterated efficiently unless $k \approx 1$ or $k \approx |L_\ell|$.

Universal-Coating where amoebot systems of n are initialized randomly around regular hexagonal objects of varying radii. Figure 25a shows the runtime of Universal-Coating in sequential rounds as a function of system size n for different object sizes. We observe the runtime increasing linearly with the system size, matching the proven runtime in Theorem 6.3.13. Figure 25b shows the ratio of runtime to the lower bound w^* on the matching dilation of the system. While Theorem 6.4.2 and Corollary 6.4.3 show that Universal-Coating achieves the worst-case optimal competitive ratio of $\Theta(n)$, we observe in practice that the average competitive ratio may be closer to logarithmic. Finally, Figure 25c shows the runtime in sequential rounds as a function of object radius for different system sizes. Overall, the runtime increases linearly with object radius, though there is significantly higher variability in the runtime for larger object radii. This is likely due to the node-based leader election phase, which depends on the length of the object's boundary.

Chapter 7

CONVEX HULL FORMATION

Determining the *convex hull* of a set of points is a well-studied and classical problem in computational geometry and combinatorial optimization. While this problem is usually treated abstractly, here we consider it as a collective task for programmable matter where a system must form a physical seal around a static object using as little resources as possible [53]. This is an attractive behavior for programmable matter as it would enable systems to, for example, isolate and contain oil spills [198], mimic the collective transport capabilities seen in ant colonies [125, 142], or even surround and engulf malignant entities in the human body as phagocytes do [1]. Though our algorithm is certainly not the first distributed approach taken to computing convex hulls, to our knowledge it is the first to do so with distributed computational entities that have *no sense of global orientation nor of their coordinates* and are limited to only *local sensing and constant-size memory.* Moreover, to our knowledge ours is the first distributed approach to computing *restricted-orientation convex hulls*, a generalization of usual convex hulls (see definitions in Section 7.1). Finally, our algorithm is *gracefully degrading*: when the number of amoebots is insufficient to form an object's convex hull, a maximal partial convex hull is still formed.

The convex hull problem is arguably one of the best-studied problems in computational geometry. Many parallel algorithms have been proposed to solve it (see, e.g., [3, 76, 85]), as have several distributed algorithms (see [69, 146, 164]). However, conventional models of parallel and distributed computation assume that the computational and communication capabilities of the individual processors far exceed those of individual modules of programmable matter. Most commonly, for example, the nodes are assumed to know their global coordinates and to can communicate non-locally. Amoebots in the amoebot model have only constant-size memory and can communicate only with their immediate neighbors. Furthermore, the object's boundary may be much larger than the number of amoebots, making it impossible for the amoebot system to store all the geographic locations. Finally, to our knowledge, there only exist centralized algorithms to compute (strong) restricted-orientation convex hulls (see, e.g., [119] and the references therein); ours is the first to do so in a distributed setting.

The related problems of shape formation (Section 2.5, [52, 58, 62, 66, 95]) and object coating (Chapter 6, [55, 61]) have both been considered under the amoebot model. Like shape formation, convex hull formation is a reconfiguration problem involving a manipulation of the amoebot system's shape; however, the desired convex hull is a function of the object and thus is not known to the amoebots a priori. Object coating also depends on an object, but may not form a convex seal around the object using the minimum number of amoebots.

Organization. We begin by defining two variants of the convex hull formation problem in the context of our triangular lattice G_Δ based on two interpretations of convex hulls in restricted orientation geometry: the *strong* $\mathcal{O}_\Delta$*-hull* and the *(weak)* $\mathcal{O}_\Delta$*-hull* (Section 7.1). Our algorithm for convex hull formation has two phases: the amoebot system first explores the object to learn the convex hull's dimensions, and then uses this knowledge to form the convex hull. In Section 7.2, we introduce the main ideas behind the learning phase as a novel local algorithm run by a single amoebot with unbounded memory. We then give new results on organizing a system of amoebots each with $\mathcal{O}(1)$ memory into binary counters in Section 7.3. Combining the results of these two sections, we present the full distributed algorithm for learning and forming the strong $\mathcal{O}_\Delta$-hull in Section 7.4. We conclude by presenting an extension of our algorithm to solve the $\mathcal{O}_\Delta$-hull formation problem in Section 7.5.

7.1 The Convex Hull Formation Problem

For this chapter, we assume the simplified sequential amoebot model in which at most one amoebot is active at a time and the adversary activates every amoebot infinitely often (Section 2.2.4). We further assume geometric space, common chirality, and constant-size memory (see Table 1). In addition to the standard model, we define some terminology specific to our application of convex hull formation, all of which are illustrated in Figure 26. Recall from Section 2.4 that an *object* is a finite, static, simply connected set of nodes that does not perform computation. The *boundary* $B(O)$ of an object O is the set of all nodes in $V \setminus O$ that are adjacent to O. Object O contains a *tunnel of width* i if the subgraph induced by $V \setminus O$ is i-connected but not $(i+1)$-connected. We assume that amoebots can differentiate between adjacent object nodes and neighboring amoebots.

We now formally define the notions of convexity and convex hulls for our model. We start by introducing the concepts of *restricted-orientation convexity* (also known as *$\mathcal{O}$-convexity*) and *strong restricted-orientation convexity* (or *strong $\mathcal{O}$-convexity*) which are well established in computational geometry [84, 167]. We then apply these generalized notions of convexity to our discrete setting on the triangular lattice G_Δ.

In the continuous setting, given a set of orientations $\mathcal{O}$ in $\mathbb{R}^2$, a geometric object is said to be *$\mathcal{O}$-convex* if its intersection with every line with an orientation from

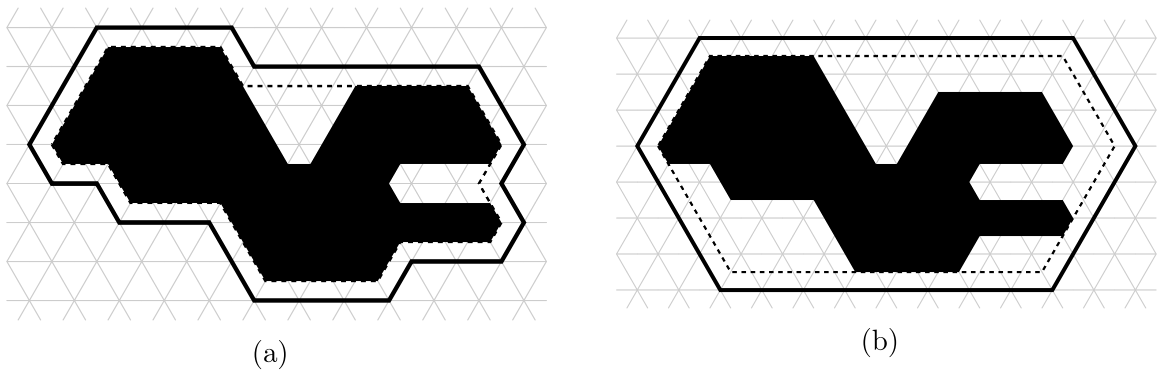


Figure 26. Restricted-Orientation Convex Hulls. An object O (black) with a tunnel of width 1 on its right side and its (a) $\mathcal{O}$-hull (dashed line) and $\mathcal{O}_\Delta$-hull $H'(O)$ (solid black line), and (b) strong $\mathcal{O}$-hull (dashed line) and strong $\mathcal{O}_\Delta$-hull $H(O)$ (solid black line).

$\mathcal{O}$ is either empty or connected. The *$\mathcal{O}$-hull* of a geometric object O is defined as the intersection of all $\mathcal{O}$-convex sets containing O, or, equivalently, as the minimal $\mathcal{O}$-convex set containing O. An *$\mathcal{O}$-block* of two points in $\mathbb{R}^2$ is the intersection of all half-planes defined by lines with orientations in $\mathcal{O}$ and containing both points. The *strong $\mathcal{O}$-hull* of a geometric object O is defined as the minimal $\mathcal{O}$-block containing O.

We now apply the definitions of *$\mathcal{O}$-hull* and *strong $\mathcal{O}$-hull* to the discrete setting of a lattice. Let $\mathcal{O}$ be the *orientation set* of G_Δ, i.e., the three orientations of axes of the triangular lattice. The *(weak) $\mathcal{O}_\Delta$-hull* of object O, denoted $H'(O)$, is the set of nodes in $V \setminus O$ adjacent to the $\mathcal{O}$-hull of O in $\mathbb{R}^2$ (see Figure 26a).[8] Analogously, the *strong $\mathcal{O}_\Delta$-hull* of object O, denoted $H(O)$, is the set of nodes in $V \setminus O$ adjacent to the strong $\mathcal{O}$-hull of O in $\mathbb{R}^2$ (see Figure 26b). For simplicity, unless there is a risk of ambiguity, we will use the terms "strong $\mathcal{O}_\Delta$-hull" and "convex hull" interchangeably throughout this work.

An instance of the *strong $\mathcal{O}_\Delta$-hull formation problem* (or the *convex hull formation*

[8]We offset the convex hull from its traditional definition by one layer of nodes since the amoebots cannot occupy nodes already occupied by O.

problem) has the form $(\mathcal{S}, O)$ where $\mathcal{S}$ is a finite, connected system of initially contracted amoebots and $O \subset V$ is an object. We assume that (i) $\mathcal{S}$ contains a unique leader amoebot ℓ initially adjacent to O,[9] (ii) there are at least $|\mathcal{S}| > \log_2(|H(O)|)$ amoebots in the system, and (iii) O does not have any tunnels of width 1.[10] A local, distributed algorithm $\mathcal{A}$ *solves* an instance $(\mathcal{S}, O)$ of the convex hull formation problem if, when each amoebot executes $\mathcal{A}$ individually, $\mathcal{S}$ is reconfigured so that every node of $H(O)$ is occupied by a contracted amoebot. The $\mathcal{O}_\Delta$-*hull formation problem* can be stated analogously.

7.2 The Single-Amoebot Algorithm

We first consider an amoebot system composed of a single amoebot A with unbounded memory and present a local algorithm for learning the strong $\mathcal{O}_\Delta$-hull of object O. As will be the case in the distributed algorithm, amoebot A does not know its global coordinates or orientation. We assume A is initially on $B(O)$, the boundary of O. The main idea of this algorithm is to let A perform a clockwise traversal of $B(O)$, updating its knowledge of the convex hull as it goes.

In particular, the convex hull can be represented as the intersection of six half-planes $\mathcal{H} = \{N, NE, SE, S, SW, NW\}$, which A can label using its local compass (see Figure 27). Amoebot A estimates the location of these half-planes by maintaining six

[9]One could use the Improved-Leader-Election algorithm described in Chapter 5 to obtain such a leader in $\mathcal{O}(|\mathcal{S}|)$ rounds, with high probability. Removing this assumption would simply change all the deterministic guarantees given in this work to guarantees with high probability.

[10]We believe our algorithm could be extended to handle tunnels of width 1 in object O, but this would require technical details beyond the scope of this work.

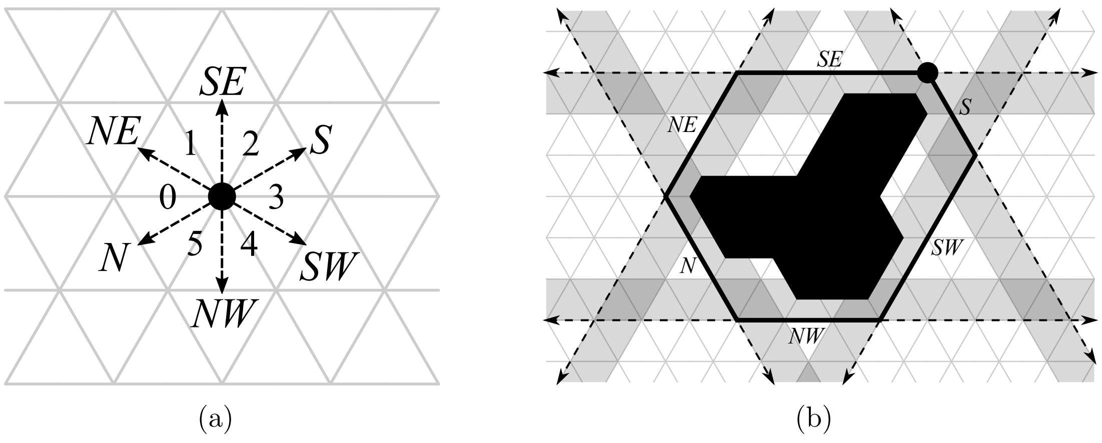


Figure 27. The Half-Planes of a Convex Hull. (a) An amoebot's local labeling of the six half-planes composing the convex hull: the half-plane between its local 0 and 5-labeled edges is N, and the remaining half-planes are labeled accordingly. (b) An object (black) and the six half-planes (dashed lines with shading on included side) whose intersection forms its convex hull (black line). As an example, the node depicted in the upper-right is distance 0 from the S and SE half-planes and distance 7 from N.

counters $\{d_h : h \in \mathcal{H}\}$, where each counter d_h represents the L_1-distance[11] from the position of A to half-plane h. If at least one of these counters is equal to 0, A is on its current estimate of the convex hull.

Each counter is initially set to 0, and A updates them as it moves. Let $[6] = \{0, \ldots, 5\}$ denote the six directions A can move in, corresponding to its contracted port labels. In each step, A first computes the direction $i \in [6]$ to move toward using the right-hand rule, yielding a clockwise traversal of $B(O)$. Since O was assumed to not have tunnels of width 1, direction i is unique. Amoebot A then updates its distance counters by setting $d_h \leftarrow \max\{0, d_h + \delta_{i,h}\}$ for all $h \in \mathcal{H}$, where $\delta_i = (\delta_{i,N}, \delta_{i,NE}, \delta_{i,SE}, \delta_{i,S}, \delta_{i,SW}, \delta_{i,NW})$ is defined as follows:

$$\delta_0 = (1, 1, 0, -1, -1, 0) \qquad \delta_1 = (0, 1, 1, 0, -1, -1) \qquad \delta_2 = (-1, 0, 1, 1, 0, -1)$$
$$\delta_3 = (-1, -1, 0, 1, 1, 0) \qquad \delta_4 = (0, -1, -1, 0, 1, 1) \qquad \delta_5 = (1, 0, -1, -1, 0, 1)$$

[11]The distance from a node to a half-plane is the number of edges in a shortest path between the node and any node on the line defining the half-plane.

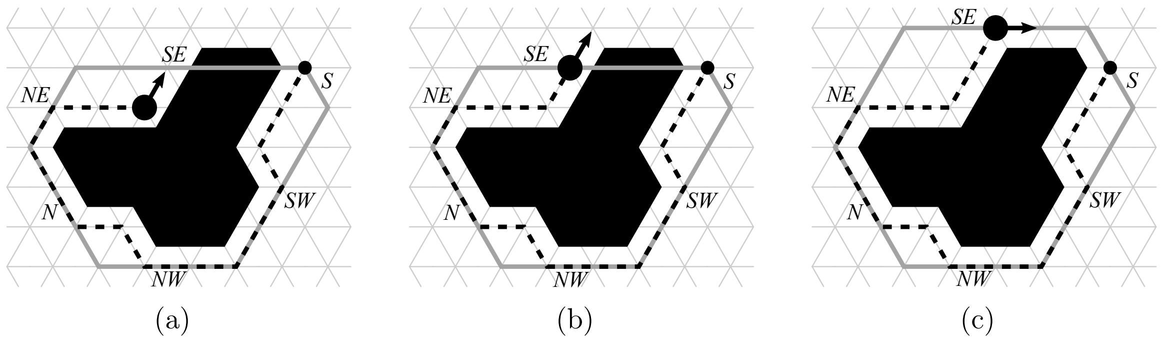


Figure 28. The Single-Amoebot Algorithm for Convex Hull Formation. The amoebot A with its convex hull estimate (gray line) after traversing the path (dashed line) from its starting point (small black dot). (a) $d_h \geq 1$ for all $h \in \mathcal{H}$, so its next move does not push a half-plane. (b) Its next move is toward the SE half-plane and $d_{SE} = 0$, so (c) SE is pushed.

Thus, every movement decrements the distance counters of the two half-planes to which A gets closer, and increments the distance counters of the two half-planes from which A gets farther away. Whenever A moves toward a half-plane to which its distance is already 0, the value stays 0, essentially "pushing" the estimation of the half-plane one step further. An example of such a movement is given in Figure 28.

Finally, A needs to detect when it has learned the complete convex hull. To do so, it stores six terminating bits $\{b_h : h \in \mathcal{H}\}$, where b_h is equal to 1 if A has visited half-plane h (i.e., if d_h has been 0) since A last pushed any half-plane, and 0 otherwise. Whenever A moves without pushing a half-plane (e.g., Figure 28a–28b), it sets $b_h = 1$ for all h such that $d_h = 0$ after the move. If its move pushed a half-plane (e.g., Figure 28b–28c), it resets all its terminating bits to 0. Once all six terminating bits are 1, A contracts and terminates.

We now analyze the correctness and runtime of this single-amoebot algorithm. Note that, since the amoebot system contains only one amoebot A, each activation of A is also a (fair, sequential) round. For a given round i, let $H_i(O) \subset V$ be the set of all nodes enclosed by A's estimate of the convex hull of O after round i, i.e., all nodes in the closed intersection of the six half-planes. We first show that A's estimate of the convex hull represents the correct convex hull $H(O)$ after at most one traversal of the object's boundary, and does not change afterwards.

Lemma 7.2.1. *If amoebot A completes its traversal of $B(O)$ in round i^*, then $H_i(O) = H(O)$ for all $i \geq i^*$.*

Proof. Since A exclusively traverses $B(O)$, $H_i(O) \subseteq H(O)$ for all rounds i. Furthermore, $H_i(O) \subseteq H_{i+1}(O)$ for any round i. Once A has traversed the whole boundary, it has visited a node of each half-plane corresponding to $H(O)$, and thus $H_{i^*}(O) = H(O)$. $\square$

We now show amoebot A terminates if and only if it has learned the complete convex hull.

Lemma 7.2.2. *If $H_i(O) \subset H(O)$ after some round i, then $b_h = 0$ for some half-plane $h \in \mathcal{H}$.*

Proof. Suppose to the contrary that after round i, $H_i(O) \subset H(O)$ but $b_h = 1$ for all $h \in \mathcal{H}$; let i be the first such round. Then after round $i-1$, there was exactly one half-plane $h_1 \in \mathcal{H}$ such that $b_{h_1} = 0$; all other half-planes $h \in \mathcal{H} \setminus \{h_1\}$ have $b_h = 1$. Let $h_2, \ldots, h_6$ be the remaining half-planes in clockwise order, and let round $t_j < i-1$ be the one in which b_{h_j} was most recently flipped from 0 to 1, for $2 \leq j \leq 6$. Amoebot A could only set $b_{h_j} = 1$ in round t_j if its move in round t_j did not push any half-planes and $d_{h_j} = 0$ after the move. There are two ways this could have occurred.

First, A may have already had $d_{h_j} = 0$ in round $t_j - 1$ and simply moved along h_j in round t_j, leaving $d_{h_j} = 0$. But for this to hold and for A to have had $b_{h_j} = 0$ after round $t_j - 1$, A must have just pushed h_j, resetting all its terminating bits to 0.

Amoebot A could not have pushed any half-plane during rounds t_2 up to $i-1$, since $b_{h_2} = \cdots = b_{h_6} = 1$, so this case only could have occurred with half-plane h_2.

For the remaining half-planes h_j, for $3 \leq j \leq 6$, A must have had $d_{h_j} = 1$ in round $t_j - 1$ and moved into h_j in round t_j. But this is only possible if A pushed h_j in some round prior to $t_j - 1$, implying that A has already visited $h_3, \ldots, h_6$. Therefore, A has completed at least one traversal of $B(O)$ by round i, but $H_i(O) \subset H(O)$, contradicting Lemma 7.2.1. □

Lemma 7.2.3. *Suppose $H_i(O) = H(O)$ for the first time after some round i. Then amoebot A terminates at some node of $H(O)$ after at most one additional traversal of $B(O)$.*

Proof. Since i is the first round in which $H_i(O) = H(O)$, amoebot A must have just pushed some half-plane h — resetting all its terminating bits to 0 — and now occupies a node u with distance 0 to h. Due to the geometry of the triangular lattice, the next node in a clockwise traversal of $B(O)$ from u must also have distance 0 to h, so A will set b_h to 1 after its next move. As A continues its traversal, it will no longer push any half-planes because its convex hull estimation is complete. Thus, A will visit every other half-plane h' without pushing it, causing A to set each $b_{h'}$ to 1 before reaching u again. Amoebot A sets its last terminating bit b_{h^*} to 1 when it next visits a node v with distance 0 to h^*. Therefore, A terminates at $v \in B(O) \cap H(O)$. □

The previous lemmas immediately imply the following theorem. Let $B = |B(O)|$.

Theorem 7.2.4. *The single-amoebot algorithm terminates after $t^* = \mathcal{O}(B)$ rounds with amoebot A at a node $u \in B(O) \cap H(O)$ and $H_{t^*}(O) = H(O)$.*

Algorithm 11 BINARY-COUNTER for Amoebot A_i

```
1: if the next non-final token in $A_i$.toks is a $c^+$ then
2:     if $A_i$.bit $= 0$ then
3:         Dequeue $c^+$ from $A_i$.toks, delete $c^+$, and set $A_i$.bit $\leftarrow 1$.
4:     else if ($A_i$.bit $= 1$) $\wedge$ ($A_{i+1}$.toks is not full, i.e., has less than two tokens) then
5:         Dequeue $c^+$ from $A_i$.toks, enqueue $c^+$ into $A_{i+1}$.toks, and set $A_i$.bit $\leftarrow 0$.
6:     else if $A_i$.bit $= \perp$ then
7:         Dequeue $f$ from $A_i$.toks and enqueue $f$ into $A_{i+1}$.toks.
8:         Dequeue $c^+$ from $A_i$.toks, delete $c^+$, and set $A_i$.bit $\leftarrow 1$.
9: else if the next non-final token in $A_i$.toks is a $c^-$ then
10:     if $A_i$.bit $= 1$ then
11:         if $\neg(A_{i+1}$.bit $= 1 \wedge A_{i+1}$.toks $= [c^-])$ then
12:             Dequeue $c^-$ from $A_i$.toks, delete $c^-$, and set $A_i$.bit $\leftarrow 0$.
13:             if ($A_{i+1}$.toks $= [f]$) $\wedge$ ($A_i \neq A_0$) then
14:                 Dequeue $f$ from $A_{i+1}$.toks, enqueue $f$ into $A_i$.toks, and set $A_i$.bit $\leftarrow \perp$.
15:     else if ($A_i$.bit $= 0$) $\wedge$ ($A_{i+1}$.toks is not full) then
16:         Dequeue $c^-$ from $A_i$.toks, enqueue $c^-$ into $A_{i+1}$.toks, and set $A_i$.bit $\leftarrow 1$.

   Functions available only to the leader, $\ell = A_0$:
17: function GENERATE($op$)
18:     if $A_0$.toks $= []$, i.e., $A_0$.toks is empty then
19:         if $op$ is increment then generate $c^+$ and enqueue it into $A_0$.toks.
20:         else if $op$ is decrement then generate $c^-$ and enqueue it into $A_0$.toks.
21: function ZEROTEST($A_0$, $A_1$)
22:     if ($A_1$.bit $= 1$) $\wedge$ ($A_1$.toks $= [c^-]$) then return unavailable.
23:     else return ($A_1$.toks $= [f]$) $\wedge$ (($A_0$.bit $= 0 \wedge A_0$.toks $= []$) $\vee$ ($A_0$.bit $= 1 \wedge A_0$.toks $= [c^-]$))
```

7.3 A Binary Counter of Amoebots

For a system of amoebots each with constant-size memory to emulate the single-amoebot algorithm of Section 7.2, the amoebots need a mechanism to distributively store the distances to each of the strong $\mathcal{O}_\Delta$-hull's six half-planes. To that end, we now describe how to coordinate an amoebot system as a binary counter that supports increments and decrements by one as well as zero-testing. This BINARY-COUNTER algorithm subsumes previous work on collaborative computation under the amoebot model that detailed an increment-only binary counter [162]. This algorithm uses *tokens*, or constant-size messages that can be passed between amoebots (see Section 2.4). Accompanying pseudocode is given in Algorithm 11.

Suppose that the participating amoebots are organized as a simple path with the leader amoebot at its start: $\ell = A_0, A_1, A_2, \ldots, A_k$. Each amoebot A_i stores a value $A_i.\text{bit} \in \{\perp, 0, 1\}$, where $A_i.\text{bit} = \perp$ implies A_i is not part of the counter; i.e., it is beyond the most significant bit. Each amoebot A_i also stores tokens in a queue $A_i.\text{toks}$; the leader ℓ can only store one token, while all other amoebots can store up to two. These tokens can be increments c^+, decrements c^-, or the unique *final token* f that represents the end of the counter. If an amoebot A_i (for $0 < i \leq k$) holds f — i.e., $A_i.\text{toks}$ contains f — then the counter value is represented by the bits of each amoebot from the leader ℓ (storing the least significant bit) up to and including A_{i-1} (storing the most significant bit).

The leader ℓ is responsible for initiating counter operations, while the rest of the amoebots use only local information and communication to carry these operations out. To increment the counter, the leader ℓ generates an increment token c^+ (assuming it was not already holding a token). Now consider this operation from the perspective of any amoebot A_i holding a c^+ token, where $0 \leq i \leq k$. If $A_i.\text{bit} = 0$, A_i consumes c^+ and sets $A_i.\text{bit} \leftarrow 1$. Otherwise, if $A_i.\text{bit} = 1$, this increment needs to be carried over to the next most significant bit. As long as $A_{i+1}.\text{toks}$ is not full (i.e., A_{i+1} holds at most one token), A_i passes c^+ to A_{i+1} and sets $A_i.\text{bit} \leftarrow 0$. Finally, if $A_i.\text{bit} = \perp$, this increment has been carried over past the counter's end, so A_i must also be holding the final token f. In this case, A_i forwards f to A_{i+1}, consumes c^+, and sets $A_i.\text{bit} \leftarrow 1$.

To decrement the counter, the leader ℓ generates a decrement token c^- (if it was not holding a token). From the perspective of any amoebot A_i holding a c^- token, where $0 \leq i < k$, the cases for $A_i.\text{bit} \in \{0, 1\}$ are nearly anti-symmetric to those for the increment. If $A_i.\text{bit} = 0$ and $A_{i+1}.\text{toks}$ is not full, A_i carries this decrement over by passing c^- to A_{i+1} and setting $A_i.\text{bit} \leftarrow 1$. However, if $A_i.\text{bit} = 1$, we only allow A_i to consume c^- and set $A_i.\text{bit} \leftarrow 0$ if $A_{i+1}.\text{bit} \neq 1$ or A_{i+1} is not only holding a c^-. While not necessary for the correctness of the decrement operation, this will enable conclusive zero-testing. Additionally, if A_{i+1} is holding f, then A_i is the most significant bit. So this decrement shrinks the counter by one bit; thus, as long as $A_i \neq A_0$, A_i additionally takes f from A_{i+1}, consumes c^-, and sets $A_i.\text{bit} \leftarrow \perp$.

Finally, the zero-test operation: if $A_1.\text{bit} = 1$ and A_1 only holds a decrement token c^-, ℓ cannot perform the zero-test conclusively (i.e., zero-testing is "unavailable"). Otherwise, the counter value is 0 if and only if A_1 is only holding the final token f and (i) $\ell.\text{bit} = 0$ and $\ell.\text{toks}$ is empty or (ii) $\ell.\text{bit} = 1$ and ℓ is only holding a decrement token c^-.

7.3.1 Correctness Analysis

We now show the *safety* of our increment, decrement, and zero-test operations for the distributed counter. More formally, we show that given any sequence of these operations, our distributed binary counter will eventually yield the same values as a centralized counter, assuming the counter's value remains nonnegative.

If our distributed counter was fully synchronized, meaning at most one increment or decrement token is in the counter at a time, the distributed counter would exactly mimic a centralized counter but with a linear slowdown in its length. Our counter instead allows for many increments and decrements to be processed in a pipelined fashion. Since the c^+ and c^- tokens are prohibited from overtaking one another, thereby altering the order the operations were initiated in, it is easy to see that the counter will correctly process as many tokens as there is capacity for.

So it remains to prove the correctness of the zero-test operation. We will prove this in two parts: first, we show the zero-test operation is always eventually available. We then show that if the zero-test operation is available, it is always reliable; i.e., it always returns an accurate indication of whether or not the counter's value is 0.

Lemma 7.3.1. *If at time t zero-testing is unavailable (i.e., amoebot A_1 is holding a decrement token c^- and $A_1.bit = 1$) then there exists a time $t' > t$ when zero-testing is available.*

Proof. We argue by induction on i — the number of consecutive amoebots A_j, for $x \leq j < x + i$, such that A_j.bit $= 1$ and A_j only holds a c^- token — that there exists a time $t^* > t$ where A_x can consume c^- and set A_x.bit $\leftarrow 0$. If $i = 1$, then either A_{x+1}.bit $\neq 1$ or A_{i+1} is not only holding a c^-, so A_x can process its c^- at its next activation (say, at $t^* > t$).

Now suppose $i > 1$ and the induction hypothesis holds up to $i - 1$. Then at time t, every amoebot A_j with $x \leq j < x + i - 1$ is holding a c^- token and has A_j.bit $= 1$. By the induction hypothesis, there exists a time $t_1 > t$ at which A_{x+1} is activated and can consume its c^- token, setting A_{x+1}.bit $\leftarrow 0$. So the next time A_x is activated (say, at $t^* > t_1$) it can do the same, consuming its c^- token and setting A_x.bit $\leftarrow 0$. This concludes our induction.

Suppose A_1 is holding a decrement token c^- and A_1.bit $= 1$ at time t, leaving the zero-test unavailable. Applying the above argument to A_1, there must exist a time $t^* > t$ such that A_1 can process its c^- and set A_1.bit $\leftarrow 0$. Since the increment and decrement tokens remain in order, A_1 will not be holding a c^- token when ℓ is next activated (say, at $t' > t^*$) allowing ℓ to perform a zero-test. □

Lemma 7.3.2. *If the zero-test operation is available, then it reliably decides whether the counter's value is* 0*.*

Proof. The statement of the lemma can be rephrased as follows: assuming the zero-test operation is available, the value of the counter $v = 0$ if and only if A_1 only holds the final token f and either (i) ℓ.bit $= 0$ and ℓ.toks is empty or (ii) ℓ.bit $= 1$ and ℓ only holds a decrement token c^-. Let $(*)$ represent the right hand side of this iff. Note that v is defined only in terms of the operations the leader has initiated, not in terms of what the amoebots have processed.

We first prove the reverse direction: if $(*)$ holds, then $v = 0$. By $(*)$, we know that A_1.toks only holds f. Thus, ℓ.bit is both the least significant bit (LSB) and the most significant bit (MSB). Also by $(*)$ we know that either ℓ.bit $= 0$ and ℓ.toks is empty, or ℓ.bit $= 1$ and ℓ.toks $= [c^-]$. In either case, it is easy to see that $v = 0$.

To prove that if $v = 0$, then $(*)$ holds, we argue by induction on the number of operations i initiated by the leader (i.e., the total number of c^+ and c^- tokens generated by ℓ). Initially, no operations have been initiated, so $v = 0$. The counter is thus in its initial configuration: A_1.toks only contains f, ℓ.bit $= 0$, and ℓ.toks is empty. So $(*)$ holds. Now suppose that the induction hypothesis holds for the first $i - 1$ operations initiated, and consider the time t_{i-1} just before ℓ generates the i-th operation at time t_i. There are two cases to consider: at time t_{i-1}, either $v = 0$ or $v > 0$.

Suppose $v = 0$ at time t_{i-1}. Since ℓ can only hold one token, ℓ.toks must have been empty at time t_{i-1} in order for ℓ to initiate another operation at time t_i. This operation must have been an increment, since a decrement on $v = 0$ violates the counter's nonnegativity. So at time t_i, $v = 1 > 0$ and thus "if $v = 0$, then $(*)$ holds" is vacuously true.

So suppose $v > 0$ at time t_{i-1}. The only nontrivial case is when $v = 1$ at time t_{i-1} and the i-th operation is a decrement; otherwise, v remains greater than 0 and "if

$v = 0$, then $(*)$ holds" is vacuously true. In this nontrivial case, $v = 0$ and ℓ.toks $= [c^-]$ at time t_i. To show $(*)$ holds, we must establish that ℓ.bit $= 1$ and A_1 only holds f at time t_i. Suppose to the contrary that ℓ.bit $= 0$ at time t_i. Then the c^- token in ℓ.toks must eventually be carried over to some amoebot A_j with $j \geq 1$ that will process it. But this implies that $v > 2^j - 1 \geq 1$ at time t_i, a contradiction that $v = 0$.

Finally, suppose to the contrary that A_1.toks $\neq [f]$ at time t_i. If A_1.bit $= \perp$, we reach a contradiction because ℓ.bit $= 0$ is the LSB and ℓ.toks $= [c^-]$, implying that $v < 0$. If A_1.bit $= 0$, we reach a contradiction because ℓ.bit $= A_1$.bit $= 0$ and thus there must exist an amoebot A_j with $j \geq 2$ that will consume the c^- token held by ℓ, implying that $v > 2^j - 1 \geq 3$. So we have that A_1.bit $= 1$ at time t_i. If A_1.toks $= [c^-]$, we reach a contradiction because the zero-test operation is available. If A_1.toks is empty or contains a c^+ token, we reach a contradiction because A_1.bit $= 1$, implying that $v > 1$. But since A_1 cannot hold two c^- tokens (as ℓ would had to have consumed a previous c^- token while A_1.bit $= 1$ and A_1.toks $= [c^-]$) and cannot hold both f and a c^- token (as this implies $v < 0$), the only remaining case is that A_1.toks $= [f]$, a contradiction. □

7.3.2 Runtime Analysis

To analyze the runtime of our distributed binary counters, we use a *dominance argument* (Remark 1) comparing the progress of the counter operations between a sequential and parallel execution, building upon the analysis of [162] that bounded the running time of an increment-only distributed counter. We first prove that the counter operations are, in the worst case, at least as fast in a sequential execution as they are in a simplified parallel execution. We then give an upper bound on the number of parallel rounds required to process these operations; combining these two results also gives a worst case upper bound on the running time in terms of sequential rounds.

Let a configuration C of the distributed counter encode each amoebot's bit value and any increment or decrement tokens it might be holding. A configuration is *valid* if (*i*) there is exactly one amoebot (say, A_i) holding the final token f, (*ii*) $A_j.\text{bit} = \bot$ if $j \geq i$ and $A_j.\text{bit} \in \{0, 1\}$ otherwise, and (*iii*) if an amoebot A_j is holding a c^+ or c^- token, then $j \leq i$. A *schedule* is a sequence of configurations $(C_0, \ldots, C_t)$. Let X be a *nonnegative* sequence of m increment and decrement operations; i.e., for all $0 \leq i \leq m$, the first i operations have at least as many increments as decrements.

Definition 7.3.3. A *parallel counter schedule* $(X, (C_0, \ldots, C_t))$ is a schedule $(C_0, \ldots, C_t)$ such that each configuration C_i is valid, each amoebot holds at most one token, and, for every $1 \leq i \leq t$, C_i is reached from C_{i-1} by satisfying the following for each amoebot A_j:

1. If $j = 0$, then $A_j = \ell$ generates the next operation according to X.
2. A_j is holding c^+ in C_i and either $A_j.\text{bit} = 0$, causing A_j to consume c^+ and set $A_j.\text{bit} \leftarrow 1$, or $A_j.\text{bit} = \bot$, causing A_j to additionally pass the final token f to A_{j+1}.
3. A_j is holding c^- and $A_j.\text{bit} = 1$ in C_i, so A_j consumes c^-. If A_{j+1} is holding f in C_i, A_j takes f from A_{j+1} and sets $A_j.\text{bit} \leftarrow \bot$; otherwise it simply sets $A_j.\text{bit} \leftarrow 0$.
4. A_j is holding c^+ and $A_j.\text{bit} = 1$ in C_i, so A_j passes c^+ to A_{j+1} and sets $A_j.\text{bit} \leftarrow 0$.
5. A_j is holding c^- and $A_j.\text{bit} = 0$ in C_i, so A_j passes c^- to A_{j+1} and sets $A_j.\text{bit} \leftarrow 1$.

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible.

Using the same sequence of operations X and any fair sequential activation sequence S, we compare a greedy parallel counter schedule to a *sequential counter schedule* $(X, (C_0^S, \ldots, C_t^S))$, where C_i^S is the amoebot system configuration at the completion of the i-th sequential round in S. Recall that in the sequential setting, each amoebot (except the leader ℓ) is allowed to hold up to two counter tokens at once while the parallel schedule is restricted to at most one token per amoebot (Definition 7.3.3). For a given (increment or decrement) token c, let $I_C(c)$ be the index of the amoebot holding c in configuration C if such an amoebot exists, or ∞ if c has already been consumed. For any two configurations C and C' and any token c, we say C *dominates* C' *with respect to* c — denoted $C(c) \succeq C'(c)$ — if and only if $I_C(c) \geq I_{C'}(c)$. We say C *dominates* C' — denoted $C \succeq C'$ — if and only if $C(c) \succeq C'(c)$ for every token c.

Lemma 7.3.4. *Given any nonnegative sequence of operations X and any fair sequential activation sequence S beginning at a valid configuration C_0^S in which each amoebot holds at most one token, there exists a greedy parallel counter schedule $(X, (C_0, \ldots, C_t))$ with $C_0 = C_0^S$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq t$.*

Proof. With a nonnegative sequence of operations X, a fair activation sequence S, and a valid starting configuration C_0^S, we obtain a unique sequential counter schedule $(X, (C_0^S, \ldots, C_t^S))$. We construct a greedy parallel counter schedule $(X, (C_0, \ldots, C_t))$ using the same sequence of operations X as follows. Let $C_0 = C_0^S$, and note that since each amoebot in C_0^S was assumed to hold at most one token, C_0 is a valid parallel configuration. Next, for $0 \leq i < t$, let C_{i+1} be obtained from C_i by performing one *parallel round*: each amoebot greedily performs one of Actions 2–5 of Definition 7.3.3 if possible; the leader ℓ additionally performs Action 1 if possible.

To show $C_i^S \succeq C_i$ for all $0 \leq i \leq t$, argue by induction on i. Clearly, since $C_0 = C_0^S$, we have $I_{C_0}(c) = I_{C_0^S}(c)$ for any token c in the counter. Thus, $C_0^S \succeq C_0$. So suppose that for all rounds $0 \leq r < i$, we have $C_r^S \succeq C_r$. Consider any counter token c in C_i. Since both the sequential and parallel schedules follow the same sequence of operations X, it suffices to show that $I_{C_i}(c) \leq I_{C_i^S}(c)$. By the induction hypothesis, we have that $I_{C_{i-1}}(c) \leq I_{C_{i-1}^S}(c)$, but there are two cases to distinguish between:

Case 1. Token c has made strictly more progress in the sequential setting than in the parallel setting by round $i-1$, i.e., $I_{C_{i-1}}(c) < I_{C_{i-1}^S}(c)$. If c is consumed in parallel round i, then c must have been consumed at some time before sequential round i. Otherwise, since c is carried over at most once per parallel round, $I_{C_i}(c) \leq I_{C_{i-1}}(c) + 1 \leq I_{C_{i-1}^S}(c) \leq I_{C_i^S}(c)$.

Case 2. Token c has made the same amount of progress in the sequential and parallel settings by round $i-1$, i.e., $I_{C_{i-1}}(c) = I_{C_{i-1}^S}(c)$. Inspection of Definition 7.3.3 shows that nothing can block c from making progress in the next parallel round, a fact we will formalize in Lemma 7.3.5. So if c is consumed in parallel round i, we must show it is also consumed in sequential round i; otherwise, c will be carried over in parallel round i, and we must show it is also carried over in sequential round i.

Suppose to the contrary that amoebot A_j consumes c in parallel round i but not in sequential round i. Then c must be a decrement token, and whenever A_j was activated in sequential round i, it must have been that $A_{j+1}.\text{bit} = 1$ and $A_{j+1}.\text{toks}$ contained a decrement token c', blocking the consumption of c. By the induction hypothesis, we have that $I_{C_{i-1}}(c') \leq I_{C_{i-1}^S}(c') = j+1$, and since the order of tokens is maintained, we have that $j = I_{C_{i-1}}(c) < I_{C_{i-1}}(c')$. Combining these expressions, we have $I_{C_{i-1}}(c') = j+1$; i.e., A_{j+1} holds c' just before parallel round i. We will show this situation is impossible: it cannot occur in the parallel execution that A_j is holding a decrement token c it will consume while A_{j+1} is also holding a decrement token c' in the same round. For c' to have reached A_{j+1}, it must have been carried over from A_j in a previous round when $A_j.\text{bit} = 0$. Since the parallel counter schedule is greedy, the only way c' is still at A_{j+1} in parallel round i is if this carry over occurred in the preceding round, $i-1$. This carry over would have left $A_j.\text{bit} = 0$ in parallel round i, but for A_j to be able to consume c in round i, as supposed, we must have that $A_j.\text{bit} = 1$, a contradiction.

Now suppose to the contrary that c is carried over from A_j to A_{j+1} in parallel round i but not in sequential round i. Then whenever A_j was last activated in sequential round i, A_{j+1} must have been holding two counter tokens, say c' and c'', where c' is buffered and c'' is the token A_{j+1} is currently processing. Thus, since counter tokens cannot overtake one another (i.e., their order is maintained), A_{j+1} must have been holding c' and c'' before sequential round i began, i.e., $I_{C_{i-1}^S}(c') = I_{C_{i-1}^S}(c'') = j+1$. But amoebots in the parallel setting cannot hold two tokens at once, and since the order of the tokens is maintained, we must have $I_{C_{i-1}}(c'') > I_{C_{i-1}}(c') \geq I_{C_{i-1}}(c) + 1 = j + 1$. Combining these expressions, we have $I_{C_{i-1}}(c'') > I_{C_{i-1}}(c') \geq j + 1 = I_{C_{i-1}^S}(c'')$, contradicting $C_{i-1}^S \succeq C_{i-1}$.

Therefore, $I_{C_i}(c) \leq I_{C_i^S}(c)$ in both cases. Since the choice of c was arbitrary, we conclude that $C_i^S \succeq C_i$. □

So it suffices to bound the number of parallel rounds required to process all counter operations. The following lemma shows that the counter can always process a new increment or decrement operation at the start of a parallel round.

Lemma 7.3.5. *Consider any counter token c in any configuration C_i of a greedy parallel counter schedule $(X, (C_0, \ldots, C_t))$. In C_{i+1}, c either has been carried over once ($I_{C_{i+1}}(c) = I_{C_i}(c) + 1$) or has been consumed ($I_{C_{i+1}}(c) = \infty$).*

Proof. This follows directly from Definition 7.3.3. If counter token c is held by the unique amoebot A that will consume it in configuration C_i, then by Actions 2 or 3 (if c is an increment or decrement token, respectively), nothing prohibits A from consuming c in parallel round $i+1$. Since the parallel counter schedule is greedy, this must occur, so $I_{C_{i+1}}(c) = \infty$.

Otherwise, c needs to be carried over from, say, A_j to A_{j+1} where $j = I_{C_i}(c)$. In the parallel setting, each amoebot can only store one token at a time. So the only reason c would not be carried over to A_{j+1} in parallel round $i+1$ is if A_{j+1} was also holding a counter token that needed to but couldn't be carried over in parallel round $i+1$. But this is impossible, since tokens can always be carried over past the end of the counter, and thus all tokens can be carried over in parallel. So $I_{C_{i+1}}(c) = I_{C_i}(c) + 1$. □

Unlike in the sequential setting, zero-testing is always available in the parallel setting.

Lemma 7.3.6. *The zero-test operation is available at every configuration of a greedy parallel counter schedule.*

Proof. Recall that zero-testing is unavailable whenever $A_1.\text{bit} = 1$ and $A_1.\text{toks} = [c^-]$. This issue stems from ambiguity about where the most significant bit is in the sequential setting, since it is possible for an adversarial activation sequence to flood the counter with decrements while temporarily stalling the amoebot holding the final token f. This results in a configuration where the counter's value is effectively 0 (with many decrements waiting to be processed), but the counter has not yet shrunk appropriately, bringing f to amoebot A_1.

This is not a concern of the parallel setting; by Lemma 7.3.5, we have that each counter token is either carried over or consumed in the next parallel round. So if

A_1 is holding a decrement token c^- and A_1.bit $= 1$, it must be because $A_0 = \ell$ just generated that c^- and forwarded it to A_1 in the previous parallel round. Thus, a conclusive zero-test can be performed at the end of each parallel round. □

We can synthesize these results in the following theorem.

Theorem 7.3.7. *Given any nonnegative sequence X of m operations and any fair sequential activation sequence S, the* Binary-Counter *algorithm processes all operations in $\mathcal{O}(m)$ sequential rounds.*

Proof. Let $(X, (C_0, \ldots, C_t))$ be the greedy parallel counter schedule corresponding to the sequential counter schedule defined by S and X in Lemma 7.3.4. By Lemma 7.3.5, the leader ℓ can generate one new operation from S in every parallel round. Since we have m such operations, the corresponding parallel execution requires m parallel rounds to generate all operations in X. Also by Lemma 7.3.5, assuming in the worst case that all m operations are increments, the parallel execution requires an additional $\lceil \log_2 m \rceil$ parallel rounds to process the last operation. If ever the counter needed to perform a zero-test, we have by Lemmas 7.3.6 and 7.3.2 that this can be done immediately and reliably. So all together, processing all operations in S requires $\mathcal{O}(m + \log_2 m) = \mathcal{O}(m)$ parallel rounds in the worst case, which by Lemma 7.3.4 is also an upper bound on the worst case number of sequential rounds. □

7.4 The Convex-Hull-Formation Algorithm

We now show how a system of n amoebots each with only constant-size memory can emulate the single-amoebot algorithm of Section 7.2. Recall that we assume there are sufficient amoebots to maintain the binary counters and that the system contains a unique leader amoebot ℓ initially adjacent to the object. In our Convex-Hull-Formation algorithm, this leader ℓ is primarily responsible for emulating the amoebot with unbounded memory in the single-amoebot algorithm. To do so, it organizes the other amoebots in the system as distributed memory, updating its distances d_h to half-plane h as it moves along the object's boundary. This is our algorithm's *learning phase*. In the *formation phase*, ℓ uses these complete measurements to lead the other amoebots in forming the convex hull. There is no synchronization among the various (sub)phases of our algorithm; for example, some amoebots may still be finishing the learning phase after the leader has begun the formation phase.

7.4.1 Learning the Convex Hull

The *learning phase* combines the movement rules of the single-amoebot algorithm (Section 7.2) with the distributed binary counters (Section 7.3) to enable the leader to measure the convex hull $H(O)$. We note that there are some nuances in adapting the general-purpose Binary-Counter algorithm for use in our Convex-Hull-Formation algorithm. For clarity, we will return to these issues in Section 7.4.2 after describing this phase.

In the learning phase, each amoebot A can be in one of three states, denoted $A.\text{state} \in \{\text{LEADER}, \text{FOLLOWER}, \text{IDLE}\}$. All non-leader amoebots are assumed to be initially idle and contracted. To coordinate the system's movement, the leader ℓ orients the amoebot system as a spanning tree rooted at itself. This is achieved using the well-established *spanning forest primitive* (see, e.g., Chapter 6 and [52]). If an idle amoebot A is activated and has a non-idle neighbor, then A becomes a follower and sets A.parent to this neighbor. This primitive continues until all idle amoebots become followers.

Imitating the single-amoebot algorithm of Section 7.2, ℓ performs a clockwise traversal of the boundary of the object O using the right-hand rule, updating its distance counters along the way. It terminates once it has visited all six half-planes without pushing any of them, which it detects using its terminating bits b_h. In this multi-amoebot setting, we need to carefully consider both how ℓ updates its counters and how it interacts with its followers as it moves.

Rules for Leader Computation and Movement. If ℓ is expanded and it has a contracted follower child A in the spanning tree that is keeping counter bits, ℓ pulls A in a handover.

Now suppose ℓ is contracted. If all its terminating bits b_h are equal to 1, then ℓ has learned the convex hull, completing this phase. Otherwise, it must continue its traversal of the object's boundary. If the zero-test operation is unavailable or if it is holding increment/decrement tokens for any of its d_h counters, it does not move. Otherwise, let $i \in [6]$ be its next move direction according to the right-hand rule, and let v be the node in direction i. There are two cases: either v is unoccupied, or ℓ is blocked by another amoebot occupying v.

In the case ℓ is blocked by a contracted amoebot A, ℓ can *role-swap* with A, exchanging its memory with the memory of A. In particular, ℓ gives A its counter bits, its counter tokens, and its terminating bits; promotes A to become the new leader by setting $A.\text{state} \leftarrow \textsc{leader}$ and clearing A.parent; and demotes itself by setting $\ell.\text{state} \leftarrow \textsc{follower}$ and $\ell.\text{parent} \leftarrow A$. This effectively advances the leader's position one node further along the object's boundary.

If either v is unoccupied or ℓ can perform a role-swap with the amoebot blocking it, ℓ first calculates whether the resulting move would push one or more half-planes using update vector δ_i. Let $\mathcal{H}' = \{h \in \mathcal{H} : \delta_{i,h} = -1 \text{ and } d_h = 0\}$ be the set of half-planes being pushed, and recall that since zero-testing is currently available, ℓ can locally check if $d_h = 0$. It then generates the appropriate increment and decrement tokens according to δ_i. Next, it updates its terminating bits: if it is about to push a half-plane (i.e., $\mathcal{H}' \neq \emptyset$), then it sets $b_h \leftarrow 0$ for all $h \in \mathcal{H}$; otherwise, it can again use zero-testing to set $b_h \leftarrow 1$ for all $h \in \mathcal{H}$ such that $d_h + \delta_{i,h} = 0$. Finally, ℓ performs its move: if v is unoccupied, ℓ expands into v; otherwise, ℓ performs a role-swap with the contracted amoebot blocking it.

Rules for Follower Movement. Consider any follower A. If A is expanded and has no children in the spanning tree nor any idle neighbor, it simply contracts. If A is contracted and is following the tail of its expanded parent $B = A.\text{parent}$, it is possible for A to push B in a handover. Similarly, if B is expanded and has a contracted child A, it is possible for B to pull A in a handover. However, if A is not emulating counter bits but B is, then it is possible that a handover between A and B could disconnect the counters (see Figure 29). So we only allow these handovers if either (i) both keep counter bits, like A_3 and A_4 in Figure 29; (ii) neither keep counter bits, like B_2 and B_3 in Figure 29; or (iii) one does not keep counter bits while the other holds the final token, like A_6 and C_1 in Figure 29.

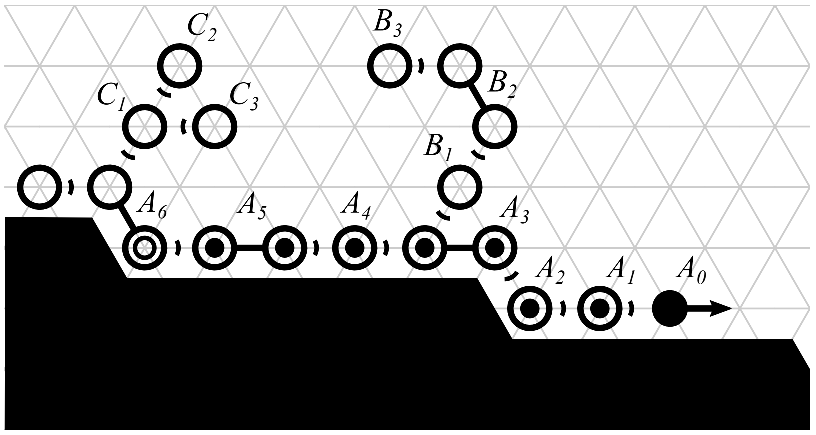


Figure 29. Safe Handovers in the Learning Phase. The leader A_0 (black dot) and its followers (black circles). Followers with dots keep counter bits, and A_6 holds the final token. Allowing B_1 to handover with A_3 would disconnect the counter, while all other potential handovers are safe.

7.4.2 Adapting Binary-Counter for Convex-Hull-Formation

Both the learning phase (Section 7.4.1) and the formation phase (Section 7.4.3) use the six distance counters d_h, for $h \in \mathcal{H}$. As alluded to in the previous section, we now describe how to adapt the general-purpose Binary-Counter algorithm described in Section 7.3 for the Convex-Hull-Formation algorithm.

First, since the amoebot system is organized as a spanning tree instead of a simple path, an amoebot A must unambiguously decide which neighboring amoebot keeps the next most significant bit. Amoebot A first prefers a child in the spanning tree already holding bits of a counter. If none exist, a child “hull”, “marker”, or “pre-marker” amoebot (see Section 7.4.3) is used. Finally, if none exist, a child on the object's boundary is chosen. (We prove that at least one of these cases is satisfied in Lemma 7.4.6).

Second, each amoebot may participate in up to six d_h counters instead of just one. Since the different counters never interact with one another, this modification is easily handled by indexing the counter variables by the counter they belong to. For each half-plane $h \in \mathcal{H}$, the final token f_h denotes the end of the counter d_h, increment and decrement tokens are tagged c_h^+ and c_h^-, respectively, and an amoebot A keeps bits $A.\text{bit}_h$ and holds tokens $A.\text{toks}_h$.

Third, the amoebot system is moving instead of remaining static, which affects the binary counters in two ways. As described in Section 7.4.1, certain handovers must be prohibited to protect the connectivity of the counters. Role-swaps would also disconnect the counters, since the leader transfers its counter information (bits, tokens, etc.) into the memory of the amoebot blocking it. To circumvent this issue, we allow each amoebot to keep up to two bits of each counter instead of one. Then, during a role-swap, the leader only transfers its less significant bits/tokens for each counter d_h, retaining the information related to the more significant bits and thus keeping the counters connected.

The fourth and final modification to the binary counters is called *bit forwarding*. As described above, both amoebots involved in the role-swap are left keeping only one bit instead of two. Thus, if ever an amoebot A only has one bit of a counter d_h while the amoebot B keeping the next most significant bit(s) has two, A can take the less significant bit and tokens from B. This ensures that all amoebots eventually hold two bits again.

Other than these four adaptations, the mechanics of the counter operations remain exactly as in Section 7.3. These adaptations increase the memory load per amoebot by only a constant factor (i.e., by one additional bit per half-plane), so the constant-size memory constraint remains satisfied.

7.4.3 Forming the Convex Hull

The *formation phase* brings as many amoebots as possible into the nodes of the convex hull $H(O)$. It is divided into two subphases. In the *hull closing subphase*, the leader amoebot ℓ uses its binary counters to lead the rest of the amoebot system along a clockwise traversal of $H(O)$. If ℓ completes its traversal, leaving every node of the convex hull occupied by (possibly expanded) amoebots, the *hull filling subphase* fills the convex hull with as many contracted amoebots as possible.

We begin with the hull closing subphase. When the learning phase ends, the leader amoebot ℓ occupies a position $s \in H(O)$ (by Lemma 7.2.3) and its distributed binary counters contain accurate distances to each of the six half-planes $h \in \mathcal{H}$. The leader's main role during the hull closing subphase is to perform a clockwise traversal of $H(O)$, leading the rest of the amoebot system into the convex hull. In particular, ℓ uses its binary counters to detect when it reaches one of the six vertices of $H(O)$, at which point it turns $60°$ clockwise to follow the next half-plane, and so on.

The amoebot system tracks the position s that ℓ started its traversal from by ensuring a unique MARKER amoebot occupies it. The marker amoebot is prohibited from contracting out of s except as part of a handover, at which point the marker role is transferred so that the marker amoebot always occupies s. Thus, when ℓ encounters the marker amoebot occupying the next node of the convex hull, it can locally determine that it has completed its traversal and this subphase.

However, there may not be enough amoebots to close the hull. Let n be the number of amoebots in the system and $H = |H(O)|$ be the number of nodes in the convex hull. If $n < \lceil H/2 \rceil$, eventually all amoebots enter the convex hull and follow the leader as far as possible without disconnecting from the marker amoebot, which is prohibited from moving from position s. With every hull amoebot expanded and unable to move any farther, a token passing scheme is used to inform the leader that there are insufficient amoebots for closing the hull and advancing to the next subphase. Upon receiving this message, the leader terminates, with the rest of the amoebots following suit.

In the following, we give a detailed implementation of this subphase from the perspective of an individual amoebot A.

Rules for Leader Computation and Movement. If the leader ℓ is holding the "all expanded" token and does not have the marker amoebot in its neighborhood — indicating that there are insufficient amoebots to complete this subphase — it generates a "termination" token and passes it to its child in the spanning tree. It then terminates by setting ℓ.state $\leftarrow$ FINISHED.

Otherwise, if ℓ is expanded, there are two cases. If ℓ has a contracted hull child A (i.e., a child A with A.state = HULL), ℓ performs a pull handover with A. If ℓ does not have any hull children but does have a contracted follower child A keeping counter bits, then this is its first expansion of the hull closing subphase and the marker should occupy its current tail position. So ℓ sets A.state $\leftarrow$ PRE-MARKER and performs a pull handover with A (see Figure 30a–30b).

During its hull traversal, ℓ keeps a variable ℓ.plane $\in \mathcal{H}$ indicating which half-plane boundary it is currently following. It checks if it has reached the next half-plane by zero-testing: if the distance to the next half-plane is 0, ℓ updates ℓ.plane accordingly. It then inspects the next node of its traversal along ℓ.plane, say v. If v is occupied by the marker amoebot A, then ℓ has completed the hull closing subphase; it updates A.state $\leftarrow$ FINISHED and then advances to the hull filling subphase. Otherwise, if ℓ is

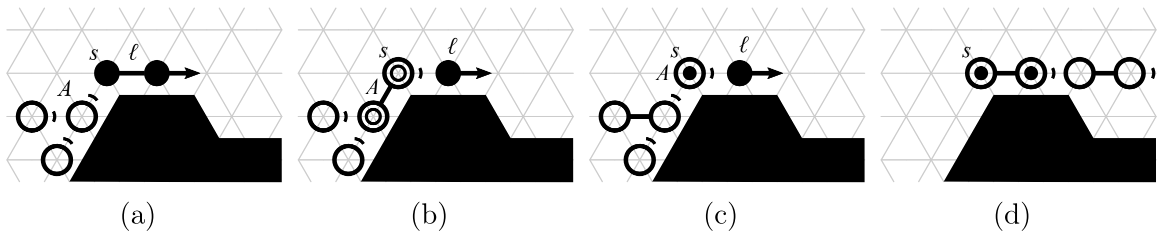


Figure 30. The Hull Closing Subphase. (a) After expanding for the first time, the leader ℓ occupies the starting position s with its tail. (b) After performing a handover with ℓ, follower child Q becomes the pre-marker (inner circles). (c) When Q contracts, it becomes the marker (inner dot). (d) If there are insufficient amoebots to close the hull, the marker amoebot will eventually become expanded and unable to contract without vacating position s.

contracted, it continues its traversal of the convex hull by either expanding into node v if v is unoccupied or by role-swapping with the amoebot blocking it, just as it did in the learning phase.

Rules for the Marker Amoebot Logic. The marker role must be passed between amoebots so that the marker amoebot always occupies the position at which the leader started its hull traversal. Whenever a contracted marker amoebot A expands in a handover with its parent, it remains a marker amoebot. When A subsequently contracts as a part of a handover with a contracted child B, A becomes a hull amoebot and B becomes a PRE-MARKER. Finally, when the pre-marker B contracts — either on its own or as part of a handover with a contracted child — B becomes the marker amoebot (see Figure 30c).

Importantly, the marker amoebot A never contracts outside of a handover, as this would vacate the leader's starting position (see Figure 30d). If A is ever expanded but has no children or idle neighbors, it generates the "all expanded" token and passes it forward along expanded amoebots only. If this ultimately causes the leader to learn there are insufficient amoebots to close the hull (as described above) and the "termination" token is passed all the way back to A, A terminates by consuming the termination token and becoming FINISHED.

Rules for Follower and Hull Amoebot Behavior. Follower amoebots move just as they did in the learning phase, with two additional conditions. First, if ever a follower is involved in a handover with the pre-marker or marker amoebot, their states are updated as described above. Second, follower amoebots never perform handovers with hull amoebots.

Hull amoebots are simply follower amoebots that have joined the convex hull. They only perform handovers with the leader and other hull amoebots. Additionally, they're responsible for passing the "all expanded" and "termination" tokens: if an expanded hull amoebot A holds the "all expanded" token and its parent is also expanded, A passes this token to its parent. If a hull amoebot A is holding the "termination" token, it terminates by passing this token to its hull or marker child and becoming FINISHED.

We now turn to the hull filling subphase, which is the final phase of the algorithm. It begins when the leader ℓ encounters the marker amoebot in the hull closing subphase, completing its traversal of the hull. At this point, the hull is entirely filled with amoebots, though some may be expanded. The remaining followers are either outside the hull or are trapped between the hull and the object. The goal of this subphase is to (i) allow trapped amoebots to escape outside the hull, and (ii) use the followers outside the hull to "fill in" behind any expanded hull amoebots, filling the hull with as many contracted amoebots as possible.

At a high level, this subphase works as follows. The leader ℓ first becomes finished. Each hull amoebot then also becomes finished when its parent is finished. A finished amoebot A labels a neighboring follower B as either TRAPPED or FILLER depending on whether B is inside or outside the hull, respectively. This can be determined locally using the relative position of B to the parent of A, which is the next amoebot on the hull in a clockwise direction. A trapped amoebot performs a coordinated series of movements with a neighboring finished amoebot to effectively take its place, "pushing" the finished amoebot outside the hull as a filler amoebot. Filler amoebots perform a clockwise traversal of the surface of the hull (i.e., the finished amoebots) searching for an expanded finished amoebot to handover with. Doing so effectively replaces a single expanded finished amoebot on the hull with two contracted ones.

There are two ways the hull filling subphase can terminate. Recall that n is the number of amoebots in the system and $H = |H(O)|$ is the number of nodes in the convex hull. If $n \geq H$, the entire hull can be filled with contracted amoebots. To detect this event, a token is used that is only passed along contracted finished amoebots. If it is passed around the entire hull, termination is broadcast so that all amoebots (including the extra ones outside the hull) become finished. However, it may be that $\lceil H/2 \rceil \leq n < H$; that is, there are enough amoebots to close the hull but not enough to fill it with all contracted amoebots. In this case, all amoebots will still eventually join the hull and become finished.

In the following, we describe the local rules underlying the three important primitives for this subphase.

Freeing Trapped Amoebots. Suppose a finished amoebot A has labeled a neighboring contracted amoebot B as trapped (see Figure 31a). In doing so, A sets itself as the parent of B. When B is next activated, it sets A.state $\leftarrow$ PRE-FILLER (see Figure 31b). This indicates to A that it should expand towards the outside of the hull as soon as

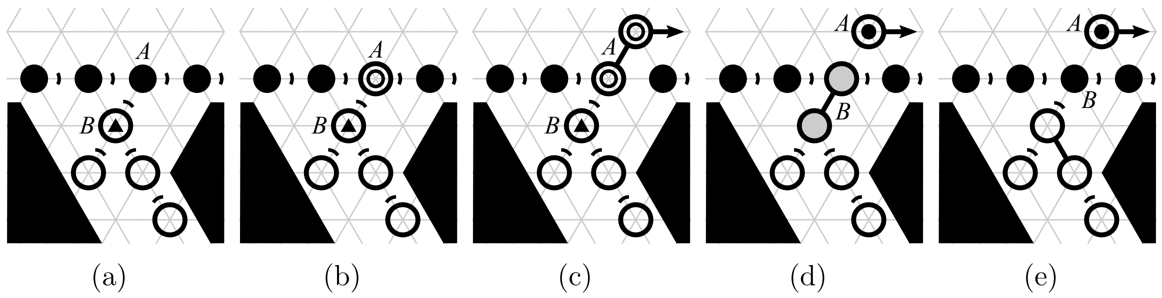


Figure 31. Freeing a Trapped Amoebot. (a) A finished amoebot A marks a neighboring follower B on the interior of the hull as trapped (inner triangle). (b) B marks its parent A as a pre-filler (inner circle). (c) A expands outside the hull. (d) In a handover between A and B, A becomes a filler (inner dot) and B becomes pre-finished (gray). (e) B contracts and becomes finished.

possible (Figure 31c). Once A has expanded, A and B perform a handover (Figure 31d). This effectively pushes A out of the hull, where it becomes a filler amoebot, and expands B into the hull, where it becomes pre-finished. Finally, whenever B contracts — either on its own or during a handover — it becomes finished, taking the original position and role of A (Figure 31e).

Filling the Hull. An amoebot A becomes a filler either by being labeled so by a neighboring finished amoebot or by being ejected from the hull while freeing a trapped amoebot, as described above. If A is expanded, it simply contracts if it has no children or idle neighbors, or performs a pull handover with a contracted follower child if it has one. If A is contracted, it finds the next node v on its clockwise traversal of the hull. Amoebot A simply expands into v unless the first occupied node clockwise from v is occupied by the tail of an expanded finished amoebot B. In this case, A performs a push handover with B, sets B to be its parent, and becomes pre-finished. Whenever A next contracts — either on its own or during a handover — it becomes finished. An example of a some movements of filler amoebots can be found in Figure 32.

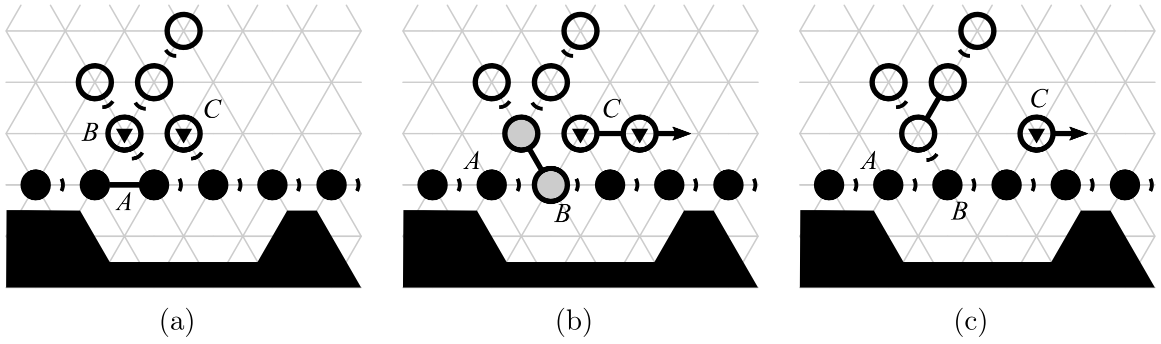


Figure 32. Movements of Filler Amoebots. (a) A finished amoebot A marks neighboring followers B and C on the exterior of the hull as fillers (inner triangle). (b) B performs a handover with A to fill the hull, becoming pre-finished (gray), while C expands along a clockwise traversal of the hull. (c) B contracts and becomes finished.

Detecting Termination. Before ℓ finishes at the start of this subphase, it generates an "all contracted" token containing a counter t initially set to 0. This token is passed backwards along the hull to contracted finished amoebots only. Whenever the token is passed through a vertex of the convex hull, the counter t is incremented. Thus, if a contracted finished amoebot is ever holding the "all contracted" token and its counter t is equal to 7, it terminates by consuming the "all contracted" token and broadcasting "termination" tokens to all its neighbors. Whenever an amoebot receives a termination token, it also terminates by becoming finished.

7.4.4 Correctness Analysis

Correctness of the Counters. We first build on the correctness proofs of Section 7.3 to show that the adapted distributed binary counters described in Section 7.4.2 remain correct. Recall that there are six d_h counters maintained by a spanning tree of follower amoebots rooted at the leader ℓ. Because the d_h counters never interact with one another, we can analyze the correctness of each counter independently. Also recall that we allow each amoebot to keep up to two bits of each counter instead of one. Since the order of the bits is maintained, this does not affect correctness. We begin by proving several general results. Throughout this section, recall that $B = |B(O)|$ denotes the length of the object's boundary, and $H = |H(O)|$ denotes the length of the object's convex hull.

Lemma 7.4.1. *The distributed binary counters never disconnect.*

Proof. By the spanning forest primitive, the amoebot system cannot physically become disconnected. So the only way to disconnect a counter d_h is to insert a follower that is not keeping bits of d_h between two amoebots that are. There are two ways this could occur. A contracted follower not keeping bits of d_h could perform a handover with an expanded follower that is (as in Figure 29), separating the counter from its more significant bits. Alternatively, the leader ℓ could role-swap without leaving behind a bit to keep d_h connected. Both of these movements were explicitly forbidden in Section 7.4.1, so the counters remain connected. □

Next, we prove two useful results regarding the lengths of the distributed binary counters.

Lemma 7.4.2. *Let L be the path of nodes traversed by leader ℓ from the start of the algorithm to its current position. Then there are at most $\lfloor \log_2 \min\{|L|, H\} \rfloor + 1$ amoebots holding bits of a distributed binary counter d_h.*

Proof. It is easy to see that the value of d_h is at most $\min\{|L|, H\}$: ℓ cannot be further from its current estimation of half-plane h than the number of moves it has made, and its distance from the true half-plane h is trivially upper bounded by the length of the convex hull. Since exactly $\lfloor \log_2 b \rfloor + 1$ bits are needed to store a binary value b, we have that $\lfloor \log_2 \min\{|L|, H\} \rfloor + 1$ bits suffice to store d_h. Each amoebot maintaining d_h holds at least one bit, so there are at most $\lfloor \log_2 \min\{|L|, H\} \rfloor + 1$ such amoebots. $\square$

Lemma 7.4.3. *Let L be the path of nodes traversed by leader ℓ from the start of the algorithm to its current position. Then there are at least $\min\{n, \lceil |L|/2 \rceil\}$ amoebots including ℓ along L.*

Proof. Argue by induction on $|L|$. If $|L| = 1$, then $\min\{n, \lceil |L|/2 \rceil\} = 1$ and ℓ is the only amoebot on its traversal path. So consider any $|L| > 1$, and suppose that the lemma holds for all $|L'| < |L|$. In particular, consider the subpath $L' \subseteq L$ containing all nodes of L except the one ℓ most recently moved into; thus, $|L'| = |L| - 1$. By the induction hypothesis, there were at least $\min\{n, \lceil (|L| - 1)/2 \rceil\}$ amoebots including ℓ on L'. We show that after ℓ moves into the $|L|$-th node of its traversal, there are at least $\min\{n, \lceil |L|/2 \rceil\}$ amoebots along L.

If $n \leq \lceil (|L| - 1)/2 \rceil$, then all amoebots (including ℓ) were on L'. Regardless of how ℓ moves into the $|L|$-th node of its traversal — i.e., either by an expansion or a role-swap — it cannot remove an amoebot as its follower. So there remain $n \geq \min\{n, \lceil |L|/2 \rceil\}$ amoebots along L.

Otherwise, if $n > \lceil (|L| - 1)/2 \rceil$, there are two cases to consider. If $|L| - 1$ is odd, then there were at least $|L|/2$ amoebots on L', a path of $|L| - 1$ nodes. Thus, at least one amoebot on L' was contracted. Via successive handovers, ℓ could eventually become contracted and perform its expansion or role-swap into the $|L|$-th node of its traversal, which again could not remove any of its followers. So there are at least $|L|/2 \geq \min\{n, \lceil |L|/2 \rceil\}$ amoebots along L.

The second case is if $|L| - 1$ is even, implying that there were at least $(|L| - 1)/2$ amoebots on L', a path of $|L| - 1$ nodes. If there were strictly more than $(|L| - 1)/2$ amoebots on L', at least one of them must have been contracted, and an argument similar to the odd case applies here as well. However, if there were exactly $(|L|-1)/2$ amoebots on L', then every amoebot along L' was expanded, including ℓ. Thus, some new follower must have joined L' in order to enable successive handovers that allowed ℓ to contract and then move into the $|L|$-th node of its traversal. So there are $(|L|-1)/2+1=(|L|+1)/2 \geq \min\{n, \lceil |L|/2 \rceil\}$ amoebots along L. □

These two lemmas are the key to proving the safety of our Convex-Hull-Formation algorithm's use of Binary-Counter. In particular, we now show that the distributed binary counters never intersect themselves — corrupting the order of the bits — and that there are always enough amoebots to maintain the counters.

Corollary 7.4.4. *The distributed binary counters never intersect.*

Proof. Suppose to the contrary that ℓ forms a cycle $\ell = A_1, \ldots, A_k, A_{k+1} = A_1$ in the spanning tree such that every amoebot A_i on the cycle is keeping bits of a counter d_h. Recall that ℓ first traverses $B(O)$ in the learning phase until it accurately measures the convex hull, at which point it traverses $H(O)$ in the hull closing subphase. The amoebots maintaining counters only exist on this traversal path. Thus, any cycle ℓ could create has length $k \geq H$. But by Lemma 7.4.2, there are at most $\lfloor \log_2 \min\{|L|, H\} \rfloor + 1$ amoebots holding bits of a given counter, and this value is maximized when $|L| \geq H$. So the cycle must have length at least H but at most $\lfloor \log_2 H \rfloor + 1$, which is impossible because $H \geq 6$ due to the geometry of the triangular lattice, a contradiction. □

Corollary 7.4.5. *There are always enough amoebots to maintain the distributed binary counters.*

Proof. We prove that the number of amoebots holding bits of a given counter never exceeds the number of amoebots following leader ℓ along its traversal path. By

Lemmas 7.4.2 and 7.4.3, it suffices to show $\lfloor \log_2 \min\{m, H\} \rfloor + 1 \leq \min\{n, \lceil m/2 \rceil\}$ for any number of nodes $m \geq 1$ traversed by ℓ. Using the assumption that $n > \log_2 H$, careful case analysis shows that this inequality holds. □

The following lemma shows that each amoebot can unambiguously decide which amoebot holds the next most significant bit of a counter when the amoebot system is structured as a spanning tree instead of a simple path.

Lemma 7.4.6. *Suppose a distributed binary counter d_h is maintained by amoebots $\ell = A_1, \ldots, A_k$, where $k \leq \lfloor \log_2 H \rfloor$. Then for every $i \in \{1, \ldots, k\}$, A_i can identify the amoebot responsible for the next most significant bit of d_h unambiguously.*

Proof. Recall from Section 7.4.2 that A_i identifies the amoebot responsible for the next most significant bit of d_h by preferring, in this order, a child already holding counter bits, a child hull or (pre-)marker amoebot, or a child on $B(O)$. We show such an amoebot exists and is unambiguous by induction on k.

If $k = 1$, then $\ell = A_1$ is the only amoebot keeping bits of d_h and thus has no children keeping counter bits. If ℓ is only holding one bit of d_h, then ℓ itself could hold the next most significant bit. So suppose ℓ is holding two bits of d_h, implying that ℓ has expanded or role-swapped at least twice. In the learning phase, no hull or (pre-)marker amoebots exist. Since ℓ only traverses $B(O)$ in this phase, it always has a follower child on $B(O)$. In the hull closing subphase, ℓ only traverses $H(O)$, and all amoebots on $H(O)$ are either hull amoebots or the (pre-)marker amoebot. The hull filling subphase does not use counters. Thus, in all phases, ℓ can unambiguously identify the amoebot responsible for the next most significant bit.

Now consider any $1 < k \leq \lfloor \log_2 H \rfloor$, and suppose the lemma holds for all $k' < k$. For all $1 \leq i < k$, A_{i+1} is the unambiguous child of A_i already holding bits of d_h.

So consider A_k. If A_k is only holding one bit of d_h, then A_k itself could hold the next most significant bit. So suppose A_k is holding two bits of d_h. If A_k is a hull amoebot, it has exactly one child also on the convex hull and this child must be a hull amoebot or the (pre-)marker amoebot. Otherwise (i.e., if A_k is not a hull amoebot), we know by the induction hypothesis that A_k is either the (pre-)marker amoebot or a follower on $B(O)$. In order for A_k to be holding two bits of d_h, the value of d_h must be at least 2^k since d_h is connected by Lemma 7.4.1. This implies ℓ has expanded or role-swapped at least 2^k times, so by Lemma 7.4.3 there are at least $\min\{n, \lceil (2^k+1)/2 \rceil\}$ amoebots following ℓ along its traversal path. To identify a unique child follower of A_k on $B(O)$, it suffices to show $\min\{n, \lceil (2^k+1)/2 \rceil\} \geq k$, i.e., that there are more followers extending along $B(O)$ than are currently holding bits of d_h. By our assumption that $n > \log_2 H$ and our supposition that $k \leq \lfloor \log_2 H \rfloor$, we have $n > \log_2 H \geq \lfloor \log_2 H \rfloor + 1 > k$. Since $2^k + 1$ is always odd whenever $k > 1$, we have $\lceil (2^k+1)/2 \rceil = 2^{k-1} + 1$, which is strictly greater than k for all $k > 1$. □

Thus, the counters are all extended along the same, unambiguous path of amoebots. To conclude our results on the distributed binary counters, we show that bit forwarding moves the bits of all six counters towards the leader as far as possible.

Lemma 7.4.7. *If ℓ only has one bit of a distributed binary counter d_h and is not holding the final token f_h at time t, then there exists a time $t' > t$ when ℓ either has two bits of d_h or is holding f_h.*

Proof. Suppose ℓ is only emulating one bit of a counter d_h and is not holding f_h at time t. Argue by induction on i, the number of consecutive amoebots starting at $\ell = A_1$ that are only emulating one bit of d_h and are not holding f_h. If $i = 1$, then A_2 must either be (i) emulating two bits of d_h, (ii) emulating the most significant bit

(MSB) of d_h and holding f_h, or (*iii*) only holding f_h. In cases (*i*) and (*ii*), ℓ can take the less significant bit from A_2 during its next activation (say, at time $t' > t$) while in case (*iii*) ℓ can take f_h instead.

Now suppose $i > 1$ and the induction hypothesis holds up to $i - 1$. Then A_{i-1} is only emulating one bit of d_h and is not holding f_h while A_i satisfies one of the three cases above. As in the base case, after the next activation of A_{i-1} (say, at $t_1 > t$), A_{i-1} is either emulating two bits of d_h or is holding f_h. Therefore, by the induction hypothesis, there exists a time $t' > t_1$ when ℓ is also either emulating two bits of d_h or holding f_h. □

Correctness of the Learning Phase. To prove the learning phase is correct, we must show that the leader ℓ obtains an accurate measurement of the convex hull by moving and performing zero-tests, emulating the single-amoebot algorithm of Section 7.2. We already proved in Lemmas 7.3.1 and 7.3.2 that ℓ will always eventually be able to perform a reliable zero-test. So we now prove the correctness of the amoebot system's movements. This relies in part on previous work on the spanning forest primitive (see Section 9, [52]), where movement for a spanning tree following a leader amoebot was shown to be correct. In fact, the correctness of our algorithm's follower movements follows directly from this previous analysis, so it remains to show the leader's movements are correct.

Lemma 7.4.8. *If ℓ is contracted, it can always eventually expand or role-swap along its clockwise traversal of $B(O)$. If ℓ is expanded, it can always eventually perform a handover with a follower.*

Proof. First suppose ℓ is contracted. Leader ℓ can only move if its zero-test operation is available for all of its d_h counters, which must eventually be the case by Lemma 7.3.1. Let v be the next clockwise node on $B(O)$. If v is unoccupied, ℓ can simply expand into node v. Otherwise, ℓ needs to perform a role-swap with the amoebot occupying v. This is only allowed when, for each counter d_h, ℓ holds two bits or the final token f_h. Lemma 7.4.7 shows this is always eventually true, implying ℓ can perform the role-swap. In either case, ℓ moves into v.

Now suppose ℓ is expanded. By the spanning forest primitive, some follower child A of ℓ will eventually contract. Thus, ℓ can perform a pull handover with A in its next activation to become contracted. □

By Lemma 7.4.8, we have that the leader ℓ can exactly emulate the movements of the single amoebot in Section 7.2. Thus, as a direct result of Theorem 7.2.4, ℓ completes the learning phase with an accurate measurement of the convex hull of O.

Correctness of the Formation Phase. The formation phase begins with the leader ℓ occupying its starting position $s \in H(O) \cap B(O)$. Recall that in the hull closing subphase, ℓ uses its binary counters to perform a clockwise traversal of the convex hull $H(O)$, leading the rest of the amoebot system into the convex hull. The amoebot system tracks the starting position s by ensuring a marker amoebot always occupies it, as we now prove.

Lemma 7.4.9. *The starting position s is always occupied by the leader, pre-marker, or marker amoebot.*

Proof. Initially, the leader ℓ occupies s. When it expands into the first node of $H(O)$, its tail still occupies s. When it contracts out of s as part of a handover with a contracted follower child A, it sets A as the pre-marker amoebot; at this point, the head of A occupies s. Whenever a pre-marker amoebot contracts to occupy s only — either on its own or as part of a handover with a contracted child — it becomes the marker amoebot. A marker amoebot A may expand so that its tail still occupies s, but can only contract out of s as part of a handover with a contracted child, which A then sets as the pre-marker amoebot. Thus, in all cases, s is either occupied by the leader, the pre-marker, or the marker amoebot. □

If there are insufficient amoebots to close the hull, we must show that the amoebot system fills as much of the hull as possible and then terminates.

Lemma 7.4.10. *If there are fewer than $\lceil H/2 \rceil$ amoebots in the system, each amoebot will eventually terminate, expanded over two nodes of $H(O)$.*

Proof. By nearly the same argument as for Lemma 7.4.8, ℓ will always eventually move along its traversal of $H(O)$, guided by its counters that continuously update the distances to each half-plane. However, by Lemma 7.4.9, the starting position s cannot be vacated by the marker amoebot unless another amoebot replaces it in a handover. Thus, ℓ will be able to traverse at most $2n$ nodes of $H(O)$ before all amoebots in the system are expanded, unable to move any further. By supposition, $n < \lceil H/2 \rceil$: if H is even, then $\lceil H/2 \rceil = H/2$ and thus $2n \leq H - 1$; if H is odd, then $\lceil H/2 \rceil = (H+1)/2$ and thus $2n \leq 2((H+1)/2 - 1) = H - 1$. Thus, there are insufficient amoebots to close the hull, even if all amoebots expand.

When the marker amoebot is expanded and has no children, which must occur by the above argument, it generates the "all expanded" token all_{exp}. Because the all_{exp} token is only passed towards the leader by expanded amoebots, we are guaranteed that every amoebot from the marker up to the amoebot currently holding all_{exp} is expanded. Thus, if ℓ ever receives the all_{exp} token but does not have the marker amoebot in its neighborhood, ℓ can locally decide that there are insufficient amoebots to close the hull. Termination is then broadcast from ℓ. □

Assuming there are sufficient amoebots to close the hull, we must show that the leader successfully completes its traversal of $H(O)$ and advances to the hull filling subphase.

Lemma 7.4.11. *If there are at least $\lceil H/2 \rceil$ amoebots in the system, then the leader ℓ will complete its traversal of $H(O)$, closing the hull.*

Proof. Once again, by nearly the same argument as for Lemma 7.4.8, ℓ will always eventually move along its traversal of $H(O)$. As in Lemma 7.4.10, ℓ will be able to traverse at most $2n$ nodes of $H(O)$. By supposition, since $n \geq \lceil H/2 \rceil$, we have that $2n \geq H$. Thus, there are enough amoebots for ℓ to close the hull.

So it remains to show that the "all expanded" token all_{exp} does not cause ℓ to terminate incorrectly when there are sufficient amoebots to close the hull. By Lemma 7.4.8, ℓ has completed at least one traversal of $B(O)$. Combining Lemma 7.4.3 with our supposition that $n \geq \lceil H/2 \rceil$ and the fact that $B \geq H$, we have that ℓ has at least

$$\min\left\{n, \left\lceil \frac{B}{2} \right\rceil\right\} - 1 \geq \min\left\{\left\lceil \frac{H}{2} \right\rceil, \left\lceil \frac{B}{2} \right\rceil\right\} - 1 \geq \left\lceil \frac{H}{2} \right\rceil - 1$$

amoebots following it. Thus, in order for the marker amoebot to be expanded and have no children — allowing it to generate the all_{exp} token — there must be at least $\lceil H/2 \rceil$ amoebots from the marker amoebot to ℓ all on the hull. If the all_{exp} token is eventually passed to ℓ, then all of these amoebots from the marker to ℓ must be expanded. So there must be exactly $H/2$ of them, since they are all expanded but must fit in the H nodes of the convex hull. Therefore, either ℓ receives the all_{exp} token but has already closed the hull or ℓ never receives the all_{exp} token. In either case, ℓ closes the hull and can advance to the hull filling subphase. □

The hull filling subphase begins when ℓ encounters the marker amoebot, closing the hull and finishing. At this point, the hull may be occupied by both expanded and contracted amoebots. We must show that this subphase fills the hull with as many contracted amoebots as possible.

Lemma 7.4.12. *If $H(O)$ is closed but not filled with all contracted amoebots and there exists an amoebot occupying a node not in $H(O)$, then at least one amoebot can make progress towards filling another hull position with a contracted amoebot.*

Proof. The main idea of this argument is to categorize all types of amoebots that occupy nodes outside $H(O)$ and then order these categories such that if no amoebots in the first i categories exist, then an amoebot in the $(i+1)$-th category must be able to make progress.

The first category contains all types of amoebots that are able to make progress without needing changes in their neighborhoods. Any idle amoebot adjacent to a non-idle amoebot can become a follower in its next activation. Similarly, any hull amoebot adjacent to a finished amoebot can become finished in its next activation.

If no amoebots from the first category exist to make progress independently, we show a amoebot from this second category can make progress. Any expanded amoebot with no children can contract in its next activation, since there are no idle amoebots adjacent to non-idle amoebots. Since no hull amoebots are adjacent to finished amoebots, all hull amoebots must be finished. Thus, any contracted follower adjacent to a node of $H(O)$ will be labeled as either trapped or filler by a neighboring finished amoebot in its next activation. Moreover, any trapped amoebot must have a finished parent: if this parent is expanded, the trapped amoebot can perform a handover with it to become pre-finished; otherwise, the trapped amoebot can mark this parent as a pre-filler.

Now consider a third category, assuming no amoebots from the first two categories exist. Among expanded followers waiting to contract in a handover, at least one expanded follower must have a contracted follower child to handover with because all followers with no children are contracted. Any expanded pre-filler must have a contracted trapped amoebot it is freeing, so these amoebots can perform a handover in their next activation, causing the trapped amoebot to become pre-finished and the pre-filler to become a filler.

The fourth category follows from the third. Any expanded filler or pre-finished amoebot must be waiting to perform a handover with a contracted follower child since all expanded amoebots with no children have already contracted. But all followers are now contracted and not adjacent to nodes of $H(O)$. Thus, any expanded filler or pre-finished amoebot waiting to perform a handover with a contracted follower child can do so in its next activation.

The fifth category follows from the third and fourth. From the fourth category, we can now assume all filler amoebots are contracted. So any contracted filler amoebots that can handover with a neighboring expanded finished amoebot do so, becoming pre-finished. But from the third category, we know there are no expanded pre-fillers protruding onto the surface of the convex hull. So there must exist a contracted filler whose next node on its clockwise traversal of the surface of $H(O)$ is unoccupied, and this contracted filler can expand in its next activation.

The final category contains contracted pre-fillers needing to expand outwards, onto the exterior of the convex hull. From the previous categories, we can assume that there are no longer any followers or fillers on the surface of the convex hull. Thus, nothing is blocking a contracted pre-filler from expanding outwards in its next activation. Therefore, as these six categories are exhaustive, we conclude that as long as there exists a amoebot occupying a node outside $H(O)$, at least one amoebot can make progress. □

Applying Lemma 7.4.12 iteratively, we can immediately conclude that the convex hull is eventually filled with all contracted amoebots if there are enough amoebots to do so, i.e., if there are at least H amoebots. However, if there are $\lceil H/2 \rceil \leq n < H$ amoebots (i.e., there are enough amoebots to close the hull but not enough to fill it with all contracted amoebots), applying Lemma 7.4.12 iteratively shows that the hull is filled with as many contracted amoebots as possible. The following lemma shows that the system terminates correctly in either case.

Lemma 7.4.13. *If there are at least $\lceil H/2 \rceil$ amoebots in the system, all amoebots eventually terminate, filling $H(O)$ with as many contracted amoebots as possible.*

Proof. Since there are at least $\lceil H/2 \rceil$ amoebots, the hull will be closed by Lemma 7.4.11. If the system contains $n \geq H$ amoebots, then applying Lemma 7.4.12 iteratively shows that the hull is eventually entirely filled with contracted finished amoebots. However, the $n - H$ extra amoebots must also terminate. Recall that the leader ℓ generates the "all contracted" token all_{con} before it finishes at the start of the phase, and that this token is passed backwards along the hull over contracted finished amoebots only. Thus, we are guaranteed that every amoebot from the amoebot currently holding the all_{con} token up to the finished amoebot that was the leader is contracted and finished. Since the hull is eventually filled with all contracted finished amoebots, all_{con}

eventually completes its traversal of the hull, triggering termination that is broadcast to all amoebots in the system.

If the system contains $\lceil H/2 \rceil \leq n < H$ amoebots, then there are too few amoebots to fill $H(O)$ with only contracted amoebots. Thus, applying Lemma 7.4.12 iteratively shows that eventually all amoebots join the hull and become finished. Therefore, in all cases, the hull is filled with as many contracted amoebots as possible and all amoebots eventually finish. $\square$

We summarize our correctness results in the following theorem.

Theorem 7.4.14. *The* Convex-Hull-Formation *algorithm correctly solves instance $(\mathcal{S}, O)$ of the convex hull formation problem if $|\mathcal{S}| \geq |H(O)|$, and otherwise forms a maximal partial strong $\mathcal{O}_\Delta$-hull of O.*

7.4.5 Runtime Analysis

We now bound the worst-case number of fair, sequential rounds for the leader ℓ to learn and form the convex hull. As in Section 7.3, we use dominance arguments (Remark 1) to show that the worst-case number of parallel rounds required by a carefully defined parallel schedule is no less than the runtime of our algorithm. The first dominance argument will show that the counters bits are forwarded quickly enough to avoid blocking leader expansions and role-intos. The second will relate the time required for ℓ to traverse the object's boundary and convex hull to the running time of our algorithm. Both build upon the dominance arguments of Section 6.3, which analyzed spanning trees of amoebots led by their root amoebots. Several nontrivial extensions are needed here to address the interactions between the counters and amoebot movements as well as traversal paths that can be temporarily blocked.

We first analyze the performance of bit forwarding (Section 7.4.2). Note that this is independent of the actual counter operations analyzed in Section 7.3; here, we analyze how the amoebots forward their counter bits towards the leader ℓ. Suppose a counter d_h is maintained by amoebots $\ell = A_0, A_1, \ldots A_k$, that is, each amoebot A_i holds one or two bits of d_h and amoebot A_k holds the final token f_h. A *bit forwarding configuration* C of counter d_h encodes the number of counter elements (i.e., bits of d_h or the final token f_h) each amoebot holds as $C = [C(0), \ldots, C(k)]$, where $C(i) \in \{1, 2\}$ is the number of elements held by amoebot A_i. A bit forwarding configuration C *dominates* another configuration C' — denoted $C \succeq C'$ — if and only if the first i amoebots of C hold at least as many bits of d_h as the first i amoebots of C' do, i.e., if $\sum_{j=0}^{i} C(j) \geq \sum_{j=0}^{i} C'(j)$ for all $i \in \{0, \ldots, k\}$.

Definition 7.4.15. A *parallel bit forwarding schedule* $(C_0, \ldots, C_t)$ is a sequence of bit forwarding configurations such that for every $1 \leq i \leq t$, C_i is reached from C_{i-1} using one of the following for each amoebot A_j, for $0 \leq j \leq k$:

1. The leader $\ell = A_0$ performs a role-swap with an amoebot in front of it, say A_{-1}, so $C_i(0) = C_{i-1}(0) - 1 = 1$ and $C_i(-1) = 1$, shifting the indexes forward.
2. A_k holding f_h either forwards f_h to A_{k-1} or A_{k-1} takes f_h from A_k, so $C_i(k) = C_{i-1}(k) - 1 = 0$ and $C_i(k-1) = C_{i-1}(k-1) + 1 = 2$.
3. A_j forwards a counter element to A_{j-1} and takes a counter element from A_{j+1}, so $C_i(j+1) = C_{i-1}(j+1) - 1$, $C_i(j) = C_{i-1}(j) = 1$, and $C_i(j-1) = C_{i-1}(j-1) + 1 = 2$.
4. A_j does not forward or receive any bits, so $C_i(j) = C_{i-1}(j)$.

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible without disconnecting the counters (i.e., leaving some $C(j) = 0$ for $j < k$) or giving any amoebot more than two counter elements.

Greedy parallel bit forwarding schedules can be directly mapped onto the *greedy parallel forest schedules* of Definition 6.3.2. An amoebot keeping two counter elements in a bit forwarding configuration C corresponds to a contracted amoebot in an amoebot system configuration M; two adjacent amoebots each keeping a single counter element in C correspond to a single expanded amoebot in M. Further, the way counter tokens are passed can be exactly mapped onto expansions, contractions, and handovers of amoebots. So the next result follows immediately from Lemmas 6.3.3 and 6.3.4 and the fact that ℓ can only role-swap if it has two counter elements.

Lemma 7.4.16. *Suppose leader ℓ only has one bit of a counter d_h and is not holding the final token f_h in round $0 \leq i \leq t-2$ of greedy parallel bit forwarding schedule $(C_0, \ldots, C_t)$. Then within the next two parallel rounds, ℓ will either have a second bit of d_h or will be holding f_h.*

Next, we combine the parallel counter schedule of Definition 7.3.3, the parallel bit forwarding schedule of Definition 7.4.15, and the movements of amoebots following leader ℓ to define a more general *parallel tree-path schedule.* We use these parallel tree-path schedules to bound the runtime of a spanning tree of amoebots led by a leader traversing some path $\mathcal{P}$. This bound will be the cornerstone of our runtime proofs for the learning and formation phases. Here, we consider amoebot system configurations C that encode each amoebot's position, state, whether it is expanded or contracted, and any counter bits and tokens it may be holding. Note that C contains all the information encoded by the counter configurations of Definition 7.3.3 and by the bit forwarding configurations of Definition 7.4.15. Thus, for an amoebot system configuration C, let C^{count} (resp., C^{bit}) be the counter configuration (resp., bit forwarding configuration) based on C.

Definition 7.4.17. A *parallel tree-path schedule* $((C_0, \ldots, C_t), \mathcal{P})$ is a schedule $(C_0, \ldots, C_t)$ such that the amoebot system in C_0 forms a tree of contracted amoebots rooted at the leader ℓ, $\mathcal{P}$ is a (not necessarily simple) path in $G_\Delta \setminus O$ starting at the position of ℓ in C_0 and, for every $1 \leq i \leq t$, C_i is reached from C_{i-1} such that (i) any counter operations are processed according to the parallel counter schedule $(\Delta, (C_0^{count}, \ldots, C_i^{count}))$ where Δ is the sequence of counter operations induced by the change vectors δ_i associated with $\mathcal{P}$, (ii) any bit forwarding operations are processed according to the parallel bit forwarding schedule $(C_0^{bit}, \ldots, C_i^{bit})$, and (iii) one of the following hold for each amoebot A:

1. The next position in path $\mathcal{P}$ is occupied by an amoebot and amoebot $A = \ell$ role-swaps with it.
2. The next position in path $\mathcal{P}$ is unoccupied and amoebot $A = \ell$ expands into it.
3. A contracts, leaving the node occupied by its tail empty in C_i.
4. A performs a handover with a neighbor B.
5. A does not move, occupying the same nodes in C_{i-1} and C_i.

Such a schedule is *greedy* if the parallel counter and bit forwarding schedules are greedy and the above actions are taken in parallel whenever possible without disconnecting the amoebot system or the counters.

Even when $\mathcal{P}$ is not a simple path, we know the distributed binary counters never disconnect or intersect by Lemma 7.4.1 and Corollary 7.4.4. Thus, for any greedy parallel counter schedule, its greedy parallel counter and bit forwarding schedules are characterized by Theorem 7.3.7 and Lemma 7.4.16, respectively. Property 1 of

Definition 7.4.17 handles role-swaps. Recall that if the leader ℓ is contracted, it must either hold two bits or the final token of each of its counters in order to role-swap with an amoebot blocking its traversal path without disconnecting the counters. So by Lemma 7.4.16, ℓ is never waiting to perform a role-swap for longer than a constant number of rounds in the parallel execution. The remaining properties are exactly those of a parallel forest schedule (Definition 6.3.2). Thus, by Lemmas 6.3.4 and 6.3.11, we have the following result:

Lemma 7.4.18. *If $\mathcal{P}$ is the (not necessarily simple) path of the leader's traversal, the leader traverses this path in $\mathcal{O}(|\mathcal{P}|)$ sequential rounds in the worst case.*

Using Lemma 7.4.18, we can directly relate the distance the leader ℓ has traversed to the system's progress towards learning and forming the convex hull. Once again, recall that $B = |B(O)|$ is the length of the object's boundary and $H = |H(O)|$ is the length of the object's convex hull. By Lemma 6.3.1, B amoebots self-organize as a spanning tree rooted at ℓ in at most B sequential rounds. By Lemmas 7.2.1 and 7.2.3, ℓ traverses $B(O)$ at most twice before completing the learning phase. Thus, by Lemma 7.4.18:

Lemma 7.4.19. *The learning phase completes in at most $\mathcal{O}(B)$ sequential rounds.*

The analysis of the hull closing subphase is similar, but contains an additional technical detail: the condition that a marker amoebot can only contract as part of a handover is not represented in a greedy parallel tree-path schedule. In particular, Property 3 of Definition 7.4.17 says that a marker amoebot with no children should contract in a greedy parallel tree-path schedule since doing so does not disconnect the amoebot system or the counters. But doing so would vacate the leader's starting position, which is explicitly prohibited by the algorithm. Instead of defining yet another type of parallel schedule capturing this condition and then relating it to parallel forest schedules (Definition 6.3.2), we instead simply observe that this cannot prohibit the leader from progressing according to Lemma 7.4.18 until all amoebots are expanded, at which point no amoebots can move any further. With this observation, we can prove the following runtime bound for the hull closing subphase.

Lemma 7.4.20. *In at most $\mathcal{O}(H)$ sequential rounds from when the leader ℓ completes the learning phase, either ℓ completes its traversal of $H(O)$ and closes the hull or every amoebot in the system terminates, expanded over two nodes of $H(O)$.*

Proof. If there are sufficiently many amoebots to close the hull (i.e., $n \geq \lceil H/2 \rceil$), then ℓ will complete its traversal of $H(O)$ by Lemma 7.4.11. The length of this traversal is H, so by Lemma 7.4.18, ℓ closes the hull in at most $\mathcal{O}(H)$ sequential rounds.

If instead there are insufficient amoebots to close the hull (i.e., $n < \lceil H/2 \rceil$), then Lemma 7.4.10 shows that every amoebot will eventually be expanded, occupying nodes of $H(O)$. Until all amoebots become expanded, there must exist at least one contracted amoebot in the system. The length of the leader's traversal path in this case is $2n < H$, so by Lemma 7.4.18, all amoebots become expanded in at most $\mathcal{O}(H)$ sequential rounds. Whenever it was that the marker amoebot first became expanded and had no children, it generated the "all expanded" token all_{exp}. Once all amoebots are expanded, the all_{exp} token must be passed forward at least once per sequential round, so the all_{exp} token reaches ℓ in at most another $\mathcal{O}(n) = \mathcal{O}(H)$ sequential rounds. In a similar fashion, it takes at most another $\mathcal{O}(n) = \mathcal{O}(H)$ sequential rounds for ℓ to broadcast termination to all amoebots in the system. Thus, every amoebot in the system is terminated and expanded over two nodes of $H(O)$ in at most $\mathcal{O}(H)$ sequential rounds. □

The final runtime lemma analyzes the hull filling subphase.

Lemma 7.4.21. *In at most $\mathcal{O}(H)$ sequential rounds from when the leader ℓ closes the hull,* $\min\{n, H\}$ *nodes of $H(O)$ will be filled with contracted finished amoebots.*

Proof Sketch. The main idea of this argument is to define a potential function representing the system's progress towards filling the hull with contracted finished amoebots, and then argue that this potential function reaches its maximum — representing the system filling the hull with as many contracted finished amoebots as possible — in at most $\mathcal{O}(H)$ sequential rounds. We consider *parallel filling schedules* that work similarly to parallel path-tree schedules, but take into consideration all the state transitions and movement rules of the hull filling subphase. Define a *filler segment* F as a connected sequence of filler amoebots on the surface of the hull, and the *head* $h(F)$ of a filler segment as the filler amoebot furthest clockwise in the segment. At time step t, let $U_t \subseteq H(O)$ be the set of nodes not yet occupied by contracted finished amoebots, f_t be the number of (pre-)filler amoebots in the system, and $\mathcal{F}_t$ be the set of distinct filler segments on the surface of the hull. We define our potential function as $\Phi(t) = -|U_t| + f_t - d(\mathcal{F}_t)$, where d is a function that sums the length of each traversal path from the head of a filler segment $F \in \mathcal{F}_t$ to the node in $U(t)$ it eventually fills. We then argue that $\Phi(t)$ strictly increases every constant number of rounds until either $U_t = 0$, meaning all hull nodes are occupied by contracted finished amoebots, or $U_t = 2(H - n)$, meaning there were insufficient amoebots to fill every hull node with a contracted finished amoebot. □

Putting it all together, we know the algorithm is correct by Theorem 7.4.14, the learning phase terminates in $\mathcal{O}(B)$ sequential rounds by Lemma 7.4.19, the hull closing subphase terminates in an additional $\mathcal{O}(H)$ sequential rounds by Lemma 7.4.20, and the hull filling subphase fills the convex hull with as many contracted amoebots as possible in another $\mathcal{O}(H)$ sequential rounds by Lemma 7.4.21. Thus, since $B \geq H$, we complete our analysis with the following theorem.

Theorem 7.4.22. *In at most $\mathcal{O}(B)$ sequential rounds, the* Convex-Hull-Formation *algorithm either solves instance $(\mathcal{S}, O)$ of the convex hull formation problem if $|\mathcal{S}| \geq |H(O)|$ or forms a maximal partial strong $\mathcal{O}_\Delta$-hull of O otherwise.*

The time required for all amoebots in the system to terminate may be longer than the bound given in Theorem 7.4.22, depending on the number of amoebots. As termination is further broadcast to the rest of the system, we know that at least one non-finished amoebot receives a termination signal and becomes finished in each sequential round. So,

Corollary 7.4.23. *The* Convex-Hull-Formation *algorithm terminates for all amoebots in system $\mathcal{S}$ in $\mathcal{O}(n)$ sequential rounds in the worst case.*

7.5 Forming the (Weak) $\mathcal{O}_\Delta$-Hull

To conclude, we show how the Convex-Hull-Formation algorithm can be extended to form the (weak) $\mathcal{O}_\Delta$-hull of object O, solving the $\mathcal{O}_\Delta$-hull formation problem. Our $\mathcal{O}_\Delta$-Hull-Formation algorithm extends the Convex-Hull-Formation algorithm at the point when a finished amoebot (say, A_{first}) first holds the "all contracted" token with counter value 7 and would usually broadcast termination. Note that by Theorem 7.4.14 this only happens if $n \geq H$, which we assume to be true for the $\mathcal{O}_\Delta$-Hull-Formation algorithm. Instead of terminating, A_{first} initiates the $\mathcal{O}_\Delta$-hull formation phase of our algorithm by becoming tightening. Every finished

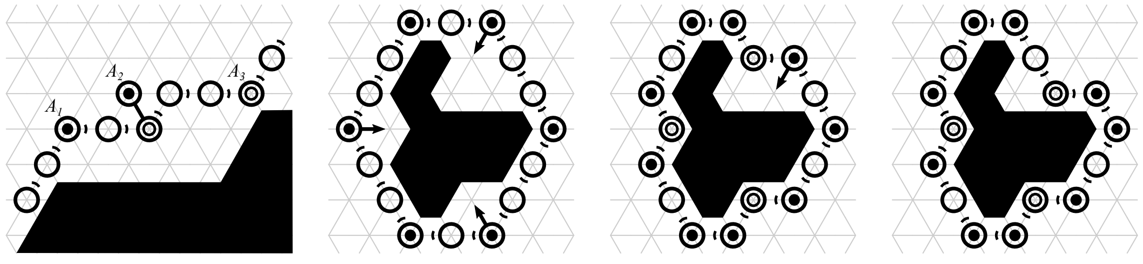


Figure 33. The $\mathcal{O}_\Delta$-HULL-FORMATION Algorithm. (a) A_1 is convex (black dot) and can perform a movement into the node between its successor and predecessor, A_2 has just made such a move, and A_3 is reflex (black circle). (b)–(d) Transforming the cycle of tightening amoebots that initially form $H(O)$ into $H'(O)$ by repeatedly moving convex amoebots towards the object.

contracted amoebot whose parent A.parent is tightening becomes tightening as well, declaring A.parent, which must be the next amoebot clockwise from A on $H(O)$, as its *successor*; analogously, the *predecessor* of A will be the amoebot B on $H(O)$ such that $B.\text{parent} = A$. Any amoebot A that is not finished becomes NON-TIGHTENING if it has a tightening or non-tightening amoebot B in its neighborhood and sets $A.\text{parent} \leftarrow B$. As the outcome, the amoebots on $H(O)$ form a bi-directed cycle of contracted tightening amoebots and all other amoebots are non-tightening, their parent pointers forming a spanning forest in which each root is a tightening amoebot.

Throughout the algorithm, we say a tightening amoebot A is *convex* (resp., *reflex*) if A and its successor and predecessor are tightening and contracted, its successor lies in direction d, and its predecessor lies in direction $(d+2) \bmod 6$ (resp., $(d+4) \bmod 6$); see Figure 33a. The idea of our algorithm is to progressively transform the structure of contracted finished amoebots initially forming the convex hull $H(O)$ into the object's $\mathcal{O}_\Delta$-hull $H'(O)$ by repeatedly moving convex amoebots towards the object (see Figure 33b–33d).

Moving Convex Amoebots. Since the cycle of finished amoebots initially occupies the convex hull $H(O)$, the algorithm begins with exactly six convex amoebots and no reflex amoebots. Whenever a contracted convex amoebot A becomes activated, it moves into the node "between" its successor and predecessor, i.e., into the node v in direction $(d + 1) \bmod 6$, where d is the direction to its successor (see amoebot A_2 in Figure 33a). Note that v must be a node contained within the strong $\mathcal{O}$-hull of O, i.e., moving A "shrinks" the cycle of tightening amoebots towards $H'(O)$. More specifically, if v is unoccupied, A simply expands into v. Otherwise, v must be occupied by a non-tightening amoebot B. If B is expanded, A simply pushes B. Otherwise, it role-swaps with B by declaring B to be a tightening amoebot, demoting itself to a non-tightening amoebot, setting B as its parent, and updating the predecessor and successor relationships for B while erasing its own.

If A is expanded, it pulls a contracted non-tightening child in a handover, if one exists, and otherwise contracts. Note that, as in the hull filling subphase of Convex-Hull-Formation, the distributed binary counters are no longer in use. Thus, any potential handovers can be performed without regard for the connectivity of the counters.

Termination Detection. Finally, we describe how to detect when the $\mathcal{O}_\Delta$-hull has been formed. When for the first time A_{first} occupies a node adjacent to O, it sends a *tight-termination* token with value 1 and forwards it to its successor. If a convex amoebot A has this token and can perform a movement, it sets the token value to 0 before forwarding it to its successor; by all other amoebots, the token is forwarded without any value change. If A_{first} receives the token with value 0, it deduces that there are still movements being made and resets the token value to 1, once again forwarding it to its successor. But if A_{first} receives the token with value 1, it knows the $\mathcal{O}_\Delta$-hull has been constructed. So A_{first} terminates by becoming TIGHT-FINISHED and any contracted amoebot with a tight-finished neighbor also becomes tight-finished.

We now show the correctness and runtime of the $\mathcal{O}_\Delta$-HULL-FORMATION algorithm. Recall that the $\mathcal{O}_\Delta$-hull $H'(O)$ has been formed if all nodes of $H'(O)$ are occupied by contracted amoebots.

Lemma 7.5.1. *Amoebot A_{first} does not become tight-finished before $H'(O)$ has been formed.*

Proof. Let $A_0 = A_{\text{first}}$ and $C = (A_0, A_1, \ldots, A_m = A_0)$ be the cycle of tightening amoebots, where A_{i+1} is the successor of A_i. Note that C never changes during the execution of the algorithm (by relabeling the amoebots involved in a role-swap). Also observe that if a contracted amoebot A_i cannot perform a movement at time t but can perform a movement at time $t' > t$, then A_{i-1} or A_{i+1} must perform a movement at some time between t and t'.

Suppose to the contrary that $A_0 = A_{\text{first}}$ becomes tight-finished at time t^* although a movement of some convex amoebot is still possible. Then the tight-termination token must have traversed the whole cycle, returning to A_{first} with value 1 at time t^*. Let time $t \leq t^*$ be the earliest time at which some amoebot A_i with $0 < i < k$ holds the tight-termination token and a amoebot A_j with $j < i$ can perform a movement; informally, t is the first time that a movement appears "behind" the tight-termination token's sweep of cycle C as it searches for movements. Amoebot $A_0 = A_{\text{first}}$ can never perform a movement, as it is already adjacent to the object at the time it creates the tight-termination token, and no tightening amoebot adjacent to the object can ever perform a movement. By the minimality of t, we know that for any $0 \leq t' < t$, all amoebots from A_1 up to the amoebot holding the tight-termination token also cannot perform a movement; in particular, this is true at time $t-1$. Thus, every amoebot A_k with $0 \leq k \leq i$, including A_{j-1} and A_{j+1}, cannot perform a movement at time $t-1$. But by the observations made above, this yields a contradiction to the claim that A_j could perform a movement at time t. $\square$

Lemma 7.5.2. *The tightening amoebots eventually form $H'(O)$, after which no convex amoebot can move anymore.*

Proof. Let $U_i \subset V$ be the set of nodes enclosed by the cycle of tightening amoebots after the i-th movement (not containing nodes occupied by tightening amoebots). To show that the tightening amoebots eventually form $H'(O)$, we show the following claims:

1. U_i is $\mathcal{O}$-convex and contains the object O for all i.
2. If U_i is an $\mathcal{O}$-convex set containing O, but is not minimal, then a movement is possible.

Together with the fact that $U_{i+1} \subset U_i$, this proves that $H'(O)$ is eventually formed. Clearly, once U_i is minimal, no movement is possible anymore, as otherwise it could not have been minimal.

To prove the first claim, argue by induction on i. Initially, the amoebots form $H(O)$, so U_0 is the strong $\mathcal{O}$-hull of O, and, by definition, is $\mathcal{O}$-convex and contains O. Now suppose U_i is $\mathcal{O}$-convex and contains O by the induction hypothesis, and let A be the amoebot that performs the next movement into a node v in direction $(d+1) \bmod 6$. Clearly, U_{i+1} still contains O. Since U_i is $\mathcal{O}$-convex, the intersection of U_i with any straight line of nodes containing v is connected. These intersections remain connected in U_{i+1} since the neighbors of v in directions $(d+3) \bmod 6$, $(d+4) \bmod 6$, and $(d+5) \bmod 6$ are not in U_i and $U_{i+1} = U_i \setminus \{v\}$.

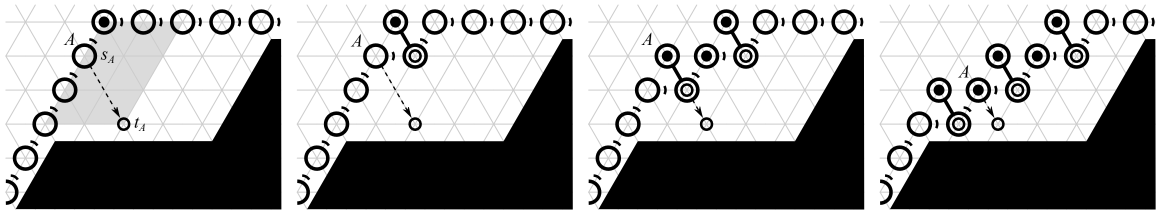
To prove the second claim, suppose that no movement of a convex amoebot is possible anymore. Therefore, every convex amoebot must be adjacent to the object, as only then it is incapable of moving any further. Every node $v \in U_i \setminus O$ on the boundary of U_i (i.e., that is adjacent to a node of $V \setminus U_i$) therefore lies on a straight line connecting two nodes of O (which are adjacent to convex amoebots). Therefore, the set that results from removing v from U_i cannot be $\mathcal{O}$-convex. Thus, U_i is minimal. □

Taken together, these lemmas prove the correctness of the $\mathcal{O}_\Delta$-Hull-Formation algorithm. By Lemma 7.5.1, we have that A_{first} will not terminate prematurely, stopping the remaining amoebots from correctly forming $H'(O)$. So by Lemma 7.5.2, $H'(O)$ is eventually formed and there are no remaining movements. Thus, the tight-termination token will never be set to value 0 again, resulting in the following concluding lemma.

Lemma 7.5.3. *Once $H'(O)$ has been formed, the tight-termination token traverses the cycle at most twice before A_{first} terminates.*

We now turn to the runtime analysis. Recall that $H = |H(O)| = |H'(O)|$ and $n = |\mathcal{S}|$. As in Section 7.4.5, we first bound the runtime for a parallel execution of the algorithm and then argue that the execution is dominated by a sequential execution. As before, we consider amoebot system configurations C that encode each amoebot's position, state, whether it is expanded or contracted, and tokens.

Definition 7.5.4. A *parallel cycle schedule* is a schedule $(C_0, \ldots, C_t)$ such that in C_0 all nodes of $H(O)$ are occupied by contracted tightening amoebots forming a directed cycle in clockwise direction and all other amoebots are contracted and connected to



(a) (b) (c) (d)

Figure 34. The Execution of a Greedy Parallel Cycle Schedule. (a) An amoebot A and its starting and ending positions. (b)–(d) Convex amoebots greedily moving towards the object whenever possible.

a tightening amoebot via a sequence of parent pointers. For every $1 \leq i \leq t$, C_i is reached from C_{i-1} such that the following holds for every amoebot A:

1. A is tightening, has a successor in direction d, and moves into the node u in direction $(d + 1)$ mod 6 by performing an expansion, if u is unoccupied, or by performing a role-swap, otherwise.
2. A has no children and contracts, leaving the node occupied by its tail empty in C_i.
3. A pulls in a child, or, if A is non-tightening, pushes its parent.
4. A occupies the same nodes in C_{i-1} and C_i.

Such a schedule is *greedy* if the above actions are taken in parallel whenever possible.

Given any greedy parallel cycle schedule, we can now show the following lemma.

Lemma 7.5.5. *Any greedy parallel cycle schedule reaches a configuration in which the amoebots form $H'(O)$ within $\mathcal{O}(H)$ parallel rounds.*

Proof. Consider the structure of tightening amoebots forming $H(O)$ at the beginning of the algorithm. Note that for any two convex amoebots A_1 and A_2 that are connected by a straight line of hull amoebots (i.e., that are visited consecutively in a traversal of the cycle), there exists at least one amoebot B between A_1 and A_2 on that line that is adjacent to the object O. Any movement that is performed by any amoebot between A_1 and B on that line can only be a direct or indirect consequence of A_1's first movement, but must be fully independent of any movement of A_2. Therefore, it suffices to analyze the execution of the algorithm on each of the six vertices of $H(O)$, i.e., on all tightening amoebots that initially lie between a convex amoebot A and the first amoebot adjacent to the object in any of the two directions in which A has adjacent tightening amoebots.

Consider a convex amoebot A occupying node u at the beginning of the algorithm. Let v and w be the first nodes in directions d and $(d+2) \bmod 6$, respectively, in which A has adjacent tightening amoebots such that v and w are adjacent to O. For any hull amoebot A occupying a node between u and v or between u and w, let i_A be the distance from A's initial position s_A to u, and let d_A be the distance from s_A to its final position t_A adjacent to O in direction $(d+1) \bmod 6$. We will show that A reaches t_A after at most $i_A + 2d_A$ parallel rounds, which, as both d_i and i are bounded above by H and the tightening amoebots of the six corners of $H(O)$ form $H'(O)$ independently, immediately implies the claim.

First, note that our algorithm ensures that there is never an expanded non-tightening amoebot whose parent is expanded, i.e., handovers can always be performed. Now fix some amoebot A occupying a node between u and v or between u and w and consider the parallelogram formed by diagonal vertices u and t_A, with edges extending in directions d and $(d+2) \bmod 6$ (see Figure 34a). By the definition of the $\mathcal{O}_\Delta$-hull, no node of this parallelogram can be a node of O. Therefore, the following can easily be shown by induction on the number of parallel rounds $t \leq i_A$: Every amoebot B such that $i_B \leq t-1$ is contracted and occupies the node that lies $(t - i_B)/2$ steps in direction $(d+1) \bmod 6$ of t_B, if i_B and t are both even or both odd; otherwise, B is expanded, its head occupying the node that lies $(t - i_B + 1)/2$ steps in direction $(d+1) \bmod 6$ of t_B (see Figure 34b–34d). Therefore, after i_A rounds, the successor of A, if s_A lies between u and v, or its predecessor, if it lies between u and w, is expanded for the first time (as in Figure 34b). In the next round, A will perform its first movement. It can easily be seen that A will not be hindered in its movement until it reaches t_A, which therefore takes $2d_A$ additional rounds. □

Similarly to the proofs of Sections 7.3.2 and 7.4.5, we compare a greedy parallel cycle schedule with an *sequential cycle schedule* $(C_0^S, \ldots, C_t^S)$ based on a fair sequential activation sequence S. For any two configuration C and C' and a tightening amoebot A, we say that C *dominates* C' *w.r.t.* A, if and only if A has performed at least as many movements in C as in C', and say that C *dominates* C' if and only if C dominates C' w.r.t. every amoebot. Note that if a movement is possible, it can never be hindered, and therefore we obtain the following lemma.

Lemma 7.5.6. *Given any fair sequential activation sequence S and some initial configuration C_0^S for the $\mathcal{O}_\Delta$-Hull-Formation algorithm, there exists a greedy parallel cycle schedule $(C_0, \ldots, C_t)$ with $C_0 = C_0^A$ such that $C_i^S \succeq C_i$ for all $0 \leq i \leq t$.*

After $H'(O)$ is formed, the tight-termination token is passed over the entire cycle at most twice by Lemma 7.5.3, which takes at most $\mathcal{O}(H)$ sequential rounds. Finally, once A_{first} terminates by becoming tight-finished, in the worst case only one additional amoebot becomes tight-finished in each subsequent round. Thus, it may take an additional $\mathcal{O}(n)$ sequential rounds in the worst case before all amoebots terminate. We conclude the following theorem.

Theorem 7.5.7. *In at most $\mathcal{O}(H)$ sequential rounds, the $\mathcal{O}_\Delta$-*Hull-Formation *algorithm solves instance $(\mathcal{S}, O)$ of the $\mathcal{O}_\Delta$-hull formation problem if $|\mathcal{S}| \geq |H(O)|$. After an additional $\mathcal{O}(|\mathcal{S}|)$ rounds, all amoebots have terminated.*

Chapter 8

COMPRESSION

Our first stochastic algorithm that replaces amoebot reliance on memory and communication with biased random decisions is for *compression*, gathering an amoebot system as compactly as possible [32]. More formally, an amoebot system must reconfigure to reach and remain among configurations with small boundaries, where we refer to the total length of a boundary as the *perimeter* of the configuration. Nature offers a fascinating variety of compression phenomena: fire ants form floating rafts [149], cockroach larvae perform self-organizing aggregation [113, 172], honey bees choose hive locations based on a decentralized process of swarming and recruitment [29], and the slime mold *Dictyostelium* undergoes a phase in its life cycle where 100,000 single-celled organisms gather into a cluster known as a "slug" [64]. No individual in these systems can view the group as a whole when making decisions; instead, cooperation is achieved by taking cues from nearby neighbors.

Our work on compression was originally inspired by the widely studied *Ising model* of ferromagnetism from statistical physics [111]. In this model, all nodes of some graph are assigned a positive or negative spin and a *temperature* parameter governs how likely it is for neighboring nodes to have the same spin. For certain temperatures, the system *clusters*, where large regions of the graph have the same spin. In an analogy to the Ising model, we consider a node of our triangular lattice G_Δ as having positive spin if it is occupied by an amoebot and having negative spin otherwise. Our algorithm's bias parameter λ is closely related to inverse temperature in the Ising model and thus governs the likelihood of having adjacent amoebots. Achieving compression corresponds to forming a cluster of positive spins in the Ising model with *fixed magnetization*, where the total number of nodes with each spin does not change. However, our work requires that amoebots only move to adjacent nodes and that the system remains connected, constraints not typically considered for Ising models but necessary for implementation as a distributed algorithm for amoebot systems.

Works in a variety of areas of computer science have considered compression-type problems. In swarm robotics, different variations of shape formation and aggregation problems have been studied (e.g., [20, 80, 93, 155, 160, 175]), but usually with robots that have more complex capabilities than our amoebots do. Similarly, pattern formation and creation of convex structures has been studied in the cellular automata domain (e.g., [38, 63]), but differs from our model by assuming more powerful computational capabilities. Distributed computing theory for autonomous mobile robots has treated the rendezvous (or gathering) problem which seeks to gather mobile agents at some node of a graph (see [13, 44, 86, 158] and the references therein). In comparison, amoebots follow the exclusion principle and hence are unable to gather at a single node, requiring different techniques. Lastly, algorithms for hexagon shape formation in the amoebot model were presented in [58, 62]. Although a hexagon satisfies our definition of a compressed configuration, these algorithms critically rely on a leader amoebot that coordinates the rest of the system. In comparison, the Markov chain-based algorithm we present takes a fully decentralized and local approach — forgoing the need for a leader — and is naturally robust and self-stabilizing.

8.1 Preliminaries

8.1.1 A Stochastic Approach to Self-Organizing Particle Systems

At a high level, in our stochastic approach we first define an energy function that captures our objectives for the particle system. We then design a Markov chain that, in the long run, favors particle system configurations with desirable energy values, and we prove these desirable configurations occur with high probability. Care is taken to ensure this Markov chain can be translated to a local, distributed algorithm run by each amoebot individually. (This chapter uses *particles* and *amoebots* interchangeably, preferring "particles" when using the centralized perspective of Markov chains and "amoebots" when referring to the corresponding distributed algorithms.)

The motivation underlying the design of this Markov chain is from statistical physics, where ensembles of particles similar to those we consider represent physical systems and demonstrate that local micro-behavior can induce global, macro-scale changes to the system [19, 25, 171]. Like a spring relaxing, physical systems favor configurations that minimize energy. Each system configuration σ has energy determined by a *Hamiltonian* $H(\sigma)$ and a corresponding weight $w(\sigma) = e^{-\beta \cdot H(\sigma)}$, where $\beta = 1/T$ is inverse temperature. Markov chains have been well-studied as a tool for sampling system configurations with probabilities proportional to their weight $w(\sigma)$, where configurations with the least energy $H(\sigma)$ have the highest weight and are thus most likely to be sampled [166].

Using a Metropolis filter [102], we can design a Markov chain $\mathcal{M}$ such that the eventual probability of the particle system being in configuration σ is $w(\sigma)/Z$, where $Z = \sum_{\sigma \in \Omega} w(\sigma)$ is a normalizing constant known as the *partition function*. In this *stationary distribution* of $\mathcal{M}$, configurations with larger weight $w(\sigma)$ — and smaller energy $H(\sigma)$ — occur with higher probability. We then use tools from Markov chain analysis to prove that configurations with large energy values occupy only an exponentially small fraction of this stationary distribution. By carefully designing $\mathcal{M}$ to use only local moves, we can adapt it as a local, distributed algorithm for self-organizing particle systems with the same guarantees on long-run behavior.

This stochastic approach to developing distributed algorithms for programmable matter is applicable to any problem where the objective can be described as minimizing some energy function, provided changes in that energy function can be calculated with only local information. However, a main challenge encountered in the applications considered so far has been proving the energy function is biased enough that desirable (low energy) configurations occur with sufficiently large probability. More formally, if there are many more configurations with high energy than with low energy, a situation known as *high entropy*, then the probability that a configuration drawn from the stationary distribution accomplishes the desired objective may not be very high. For this reason, biases must be large enough to guarantee low energy configurations — those that accomplish the objectives — dominate the state space, even if there are many undesirable high energy configurations.

8.1.2 Terminology for the Stochastic Approach

An amoebot system *arrangement* is the set of nodes in G_Δ that are occupied by tails of amoebots;[12] note that an arrangement does not distinguish which amoebot occupies which node within the arrangement. Two arrangements are equivalent if one is a translation of the other, and an equivalence class of arrangements is called an

[12]Lattice nodes occupied by heads of expanded amoebots are not considered part of a configuration, since the states of our Markov chain consider only contracted amoebots. This is for technical reasons that will be explained in Section 8.2.2.

amoebot system *configuration.* Note that two configurations differing by rotation are distinct from a global perspective, even though each individual amoebot has no sense of global orientation.

An *edge* of a configuration is an edge of G_Δ where both endpoints are occupied by tails of amoebots. Similarly, a *triangle* of a configuration is a triangular face of G_Δ with all three vertices occupied by tails of amoebots. The number of edges (resp., triangles) of a configuration σ is denoted $e(\sigma)$ (resp., $t(\sigma)$). When referring to a *path*, we mean a path of configuration edges. Two amoebots are *connected* if there is a path between them, and a configuration is *connected* if all pairs of amoebots are.

A *boundary* of a configuration σ is a minimal closed walk $\mathcal{W}$ on edges of σ that separates all amoebots of σ from a connected, unoccupied subgraph of G_Δ that has at least one node; see Figure 35. For each boundary $\mathcal{W}$, let $S_\mathcal{W}$ be the maximal such subgraph. If $S_\mathcal{W}$ is finite, we say it is a *hole.* If $S_\mathcal{W}$ is infinite, then $\mathcal{W}$ is the unique external boundary of σ. The *perimeter* $p(\sigma)$ of a configuration σ is the sum of the lengths of all boundaries of σ. Note that an edge may appear twice in the same boundary (if it is a cut-edge of σ) or in two different boundaries (e.g., if it separates two different connected components of unoccupied nodes). In these cases, the edge is counted twice in $p(\sigma)$.

We specifically consider connected amoebot system configurations. Starting from a connected configuration (possibly with holes), our algorithm will keep the system connected, eliminate all holes, and prohibit any new holes from forming, a fact we prove in Section 8.2.4. We eliminate holes because our current proof techniques require hole-free configurations. We maintain connectivity because — in addition to simplifying our proofs — allowing an amoebot system to disconnect is generally undesirable in the amoebot model. Because amoebots can only communicate with

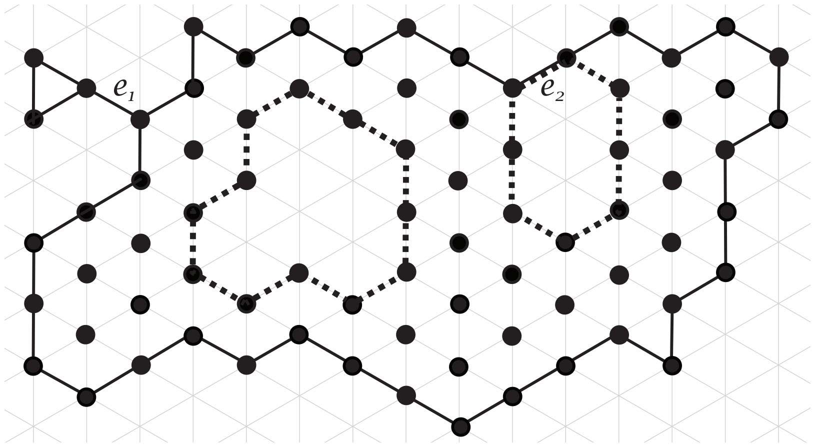


Figure 35. An Amoebot System Configuration and its Boundaries. The unique external boundary is shown as solid black lines and the other two boundaries surrounding holes are shown as dashed black lines. Edge e_1 appears on the external boundary twice, while edge e_2 appears in two different boundaries.

their immediate neighbors and do not have any global orientation or coordinates, disconnected components have no way of knowing their relative positions and thus cannot intentionally move toward one another to reconnect. One exception is [67], where amoebots were allowed to disconnect by no more than a small constant distance, with extremely careful constraints on amoebot movements ensuring eventual reconnection. In general, because the underlying lattice is infinite, it is extremely unlikely disconnected components will find each other by random search. Techniques exist for handling disconnected components constrained to remain in finite regions (see Section 11.2), but this requires modifying the underlying model, which we choose not to do here.

8.1.3 Compression of Amoebot Systems

Our objective is to solve the compression problem. There are many ways to formalize what it means for an amoebot system to be *compressed*. For example, one could try to minimize the diameter of the system, maximize the number of edges, or maximize the number of triangles. We choose to define compression in terms of minimizing perimeter. Here, we prove that for connected configurations with no holes (the states we eventually reach), minimizing perimeter, maximizing the number of edges, and maximizing the number of triangles are all equivalent and are stronger notions of compression than minimizing diameter.

The perimeter of a connected, hole-free configuration of n particles ranges from a maximum value $p_{max}(n) = 2n - 2$ when the system is in its least compressed state (a tree with no induced triangles) to some minimum value $p_{min}(n) = \Theta(\sqrt{n})$ when the system is in its most compressed state. It is easy to see $p_{min}(n) \leq 4\sqrt{n}$, and we now prove any configuration σ of n particles has $p(\sigma) \geq \sqrt{n}$; this bound is not tight but suffices for our proofs.

Lemma 8.1.1. *A connected configuration with $n \geq 2$ particles has perimeter at least $\sqrt{n}$.*

Proof. We argue by induction on n. A connected particle system with two particles necessarily has perimeter $2 \geq \sqrt{2}$. Let σ be any connected particle system configuration with $n > 2$ particles, and suppose the lemma holds for all connected configurations with fewer than n particles.

First, suppose there is a particle $P \in \sigma$ not incident to any triangles of σ. This implies P has one, two, or three neighbors, none of which are adjacent. If P has one neighbor, removing P from σ yields a configuration σ' with $n-1$ particles and, by induction, perimeter at least $\sqrt{n-1}$. Thus,

$$p(\sigma) = p(\sigma') + 2 \geq \sqrt{n-1} + 2 \geq \sqrt{n}.$$

If P has two neighbors, removing P from σ produces two connected configurations σ_1 and σ_2, where σ_1 has n_1 particles, σ_2 has n_2 particles, and $n_1 + n_2 = n - 1$. Thus,

$$p(\sigma) \geq \sqrt{n_1} + \sqrt{n_2} + 4 > \sqrt{n-1} + 4 > \sqrt{n}.$$

Similarly, if P has three neighbors, its removal produces three connected configurations with n_1, n_2, and n_3 particles, respectively, where $n_1 + n_2 + n_3 = n - 1$. We conclude:

$$p(\sigma) \geq \sqrt{n_1} + \sqrt{n_2} + \sqrt{n_3} + 6 > \sqrt{n-1} + 6 > \sqrt{n}.$$

Now, suppose every particle in σ is incident to some triangle of σ, implying there are at least $\lceil n/3 \rceil$ triangles in σ. An equilateral triangle with side length 1 has area $\sqrt{3}/4$, so the external boundary of σ encloses an area of at least $A = \lceil n/3 \rceil (\sqrt{3}/4) \geq \sqrt{3}n/12$. By the isoperimetric inequality, the minimum perimeter shape enclosing this area (without regard to lattice constraints) is a circle of radius r and perimeter p, where:

$$r = \sqrt{\frac{A}{\pi}} = \sqrt{\frac{n\sqrt{3}}{12\pi}}, \qquad p = 2\pi r = \sqrt{\frac{\pi n}{\sqrt{3}}} > \sqrt{n}.$$

As the external boundary of σ also encloses an area of at least $\sqrt{3}n/12$, we have $p(\sigma) > \sqrt{n}$. □

When n is clear from context, we omit it and refer to $p_{min} := p_{min}(n)$ and $p_{max} := p_{max}(n)$. We now formalize what it means for an amoebot system to be compressed.

Definition 8.1.2. For any $\alpha > 1$, a connected configuration σ is *α-compressed* if $p(\sigma) \leq \alpha \cdot p_{min}$.

We prove in Section 8.3 that our algorithm, when executed for a sufficiently long time, can achieve α-compression for any constant $\alpha > 1$ with probability at least

$1 - \zeta^{\sqrt{n}}$ where $\zeta < 1$ is a constant, provided n is sufficiently large. We note that α-compression implies the diameter of the amoebot system is also $\mathcal{O}(\sqrt{n})$, so our notion of α-compression is stronger than defining compression in terms of diameter.

In order to minimize perimeter using simple local moves, we exploit the following relationship. Because our algorithm eventually reaches and remains in the set of connected configurations with no holes (Section 8.2.4), we only consider this case.

Lemma 8.1.3. *For a connected configuration σ of n particles with no holes, $e(\sigma) = 3n - p(\sigma) - 3$.*

Proof. We count particle-edge incidences, of which there are $2e(\sigma)$. Counting another way, every particle has six incident edges, except for those on the (unique) external boundary $\mathcal{W}$. At each particle on $\mathcal{W}$, the exterior angle is 120, 180, 240, 300, or 360 degrees. These correspond to the particle "missing" 1, 2, 3, 4, or 5 of its six possible incident edges, respectively, or $degree/60 - 1$ missing edges. If $\mathcal{W}$ visits the same particle multiple times, we count the appropriate exterior angle and corresponding missing edges each time. From a well-known result about simple polygons with $p(\sigma)$ sides, the sum of exterior angles along $\mathcal{W}$ is $180p(\sigma) + 360$ degrees. Summing the number of "missing" edges $degree/60 - 1$ over all particles on $\mathcal{W}$, we find the total number of missing edges to be:

$$(180p(\sigma) + 360)/60 - p(\sigma) = 2p(\sigma) + 6.$$

This implies there are $6n - (2p(\sigma) + 6)$ total particle-edge incidences, so $2e(\sigma) = 6n - 2p(\sigma) - 6$. □

We briefly note that minimizing perimeter is also equivalent to maximizing triangles.

Lemma 8.1.4. *For a connected configuration σ of n particles with no holes, $t(\sigma) = 2n - p(\sigma) - 2$.*

Proof. The proof is nearly identical to that of Lemma 8.1.3 but counts particle-triangle incidences, of which there are $3t(\sigma)$. Counting another way, every particle has six incident triangles, except for those on the external boundary $\mathcal{W}$. Consider any traversal of $\mathcal{W}$; at each particle, the exterior angle is 120, 180, 240, 300, or 360 degrees. These correspond to the particle "missing" 2, 3, 4, 5, or 6 of its six possible incident triangles, respectively, or $degree/60$ missing triangles. If $\mathcal{W}$ visits the same particle multiple times, we count the appropriate exterior angle at each visit. The sum of exterior angles along $\mathcal{W}$ is $180p(\sigma) + 360$, so in total particles on the perimeter are missing $3p(\sigma) + 6$ triangles. This implies there are $6n - 3p(\sigma) - 6$ particle-triangle incidences, so $3t(\sigma) = 6n - 3p(\sigma) - 6$. □

The above lemmas imply the following corollary.

Corollary 8.1.5. *A connected configuration with no holes and minimum perimeter is also a configuration with the maximum number of edges and the maximum number of triangles.*

Because these three notions of compression are equivalent, we will state our algorithm in terms of maximizing the number of edges but prove our compression results in terms of minimizing perimeter, for ease of presentation. The original presentation of these results in [32] stated the algorithm in terms of maximizing the number of triangles.

8.1.4 Background on Markov Chains

A thorough treatment of Markov chains can be found in the standard textbook [127]; here, we present the necessary terminology relevant to our results. A *Markov chain* is a memoryless random process on a state space Ω. For compression, Ω will be all connected configurations of n particles; in particular, it will always be finite and discrete. A Markov chain randomly transitions between the states of Ω in a *stochastic* (i.e., time-independent) fashion, where the probability with which the chain transitions to its next state depends only on its current state. We focus on discrete time Markov chains, where one transition occurs per step. Because of this stochasticity, we can completely describe a Markov chain by its transition matrix M, which is an $|\Omega| \times |\Omega|$ matrix indexed by the states of Ω, defined such that for any pair $x, y \in \Omega$, $M(x, y)$ is the probability, if in state x, of transitioning to state y in one step of the Markov chain. The t-step transition probability $M^t(x, y)$ is the probability of transitioning from x to y in exactly t steps.

A Markov chain is *irreducible* if there is a sequence of valid transitions from any state to any other state, that is, if for all $x, y \in \Omega$ there is a t such that $M^t(x, y) > 0$. A Markov chain is *aperiodic* if for all $x \in \Omega$, $\gcd\{t : M^t(x, x) > 0\} = 1$. A Markov chain is *ergodic* if it is both irreducible and aperiodic, or equivalently, if there exists t such that for all $x, y \in \Omega$, $M^t(x, y) > 0$.

A *stationary distribution* of a Markov chain is a probability distribution π over Ω such that $\pi M = \pi$. Any finite, ergodic Markov chain converges to a unique stationary distribution given by $\pi(y) = \lim_{t \to \infty} M^t(x, y)$ for any $x, y \in \Omega$; importantly, for such chains this stationary distribution is completely independent of the starting state x. To verify a distribution π' is the unique stationary distribution of a finite ergodic Markov chain, it suffices to check that $\pi'(x)M(x, y) = \pi'(y)M(y, x)$ for all $x, y \in \Omega$ (the *detailed balance condition*; see, e.g., [83]). Detailed balance will be the key to connecting our global objective, captured in the stationary distribution π, to the local moves executed by our particles, which occur with probabilities described by the transition matrix M.

Given a state space Ω, a set of allowable transitions between states, and a desired stationary distribution π on Ω, the celebrated Metropolis–Hastings algorithm [102] gives a Markov chain on Ω that uses only allowable transitions and has stationary distribution π. This is accomplished by carefully setting the probabilities of the state transitions as follows. For a state $x \in \Omega$, its *neighbors* $N(x)$ are the states it can transition to and its *degree* is its number of neighbors. Starting at $x \in \Omega$, the Metropolis–Hastings algorithm picks $y \in N(x)$ uniformly with probability $1/(2\Delta)$, where Δ is the maximum degree of any state, and moves to y with probability $\min\{1, \pi(y)/\pi(x)\}$; with all the remaining probability, it stays at x and repeats. Using this probability calculation to decide whether or not to make a transition is known as a *Metropolis filter.* If the allowable transitions connect Ω (i.e., if the chain is irreducible), then π must be the stationary distribution by detailed balance.

While calculating $\pi(y)/\pi(x)$ seems to require global knowledge, this ratio can often be calculated easily using only local information when many terms cancel out, as will be the case for us. The states of the Markov chain $\mathcal{M}$ we consider are particle system configurations, its transitions correspond to individual particle moves, and our desired stationary distribution is $\pi(\sigma) = \lambda^{e(\sigma)}/Z$, where $Z = \sum_{\sigma \in \Omega} \lambda^{e(\sigma)}$ is the normalizing constant also known as the *partition function.* Each particle will calculate the Metropolis probabilities of $\mathcal{M}$ using only the difference in the number of neighbors (incident edges) it has before it moves (configuration x) and after it moves (configuration y), as $\pi(y)/\pi(x) = \lambda^{e(y)-e(x)}$. The key to our approach is that the value $e(y) - e(x)$ can be observed locally in the neighborhood of the moving particle without using global information. The resulting stationary distribution of $\mathcal{M}$ will favor configurations with more edges and thus, by Corollary 8.1.5, smaller perimeter.

8.2 Algorithms for Compression

Our objective is to give a stochastic algorithm enabling a self-organizing particle system on the triangular lattice G_Δ to provably solve the compression problem. Our algorithm relies only on local information and requires minimal communication: each particle only needs to know which of its adjacent locations are occupied by neighboring particles and which neighbors, if any, are expanded. Our solution is simple, robust, and oblivious (see Section 8.2.3).

In order to leverage powerful tools from Markov chain analysis to prove our algorithm's correctness, our algorithm is designed to maintain several properties. First, assuming the particle system is initially connected, our algorithm will ensure it stays connected, eventually eliminates any holes it may contain, and prohibits any new holes from forming — all using only local information. Second, any moves allowed by our algorithm after all holes have been eliminated are ensured to be *reversible*: if a particle moves from its current location to a new location in one step, then in the next step there is a nonzero probability that it moves back to its original location. Finally, the moves allowed by our algorithm suffice to transform any connected, hole-free particle system configuration into any other connected, hole-free configuration.

Our algorithm achieves compression by biasing particles towards moves that gain them more neighbors; i.e., where more edges with neighboring particles are formed. Specifically, a bias parameter λ controls how strongly the particles favor having more neighbors: $\lambda > 1$ corresponds to favoring neighbors, while $\lambda < 1$ corresponds to disfavoring neighbors. As Lemma 8.1.3 shows, locally favoring more neighbors is equivalent to globally favoring a shorter perimeter; this is the relationship we exploit to obtain particle compression.

8.2.1 The Markov Chain $\mathcal{M}_C$ for Compression

We begin by presenting two key properties that enable a particle to move from location ℓ to adjacent location ℓ' without disconnecting the particle system or forming a hole. We will let capital letters refer to particles and lower case letters refer to locations on the triangular lattice G_Δ, e.g., "particle P at location ℓ." For a particle P (resp., location ℓ), we use $N(P)$ (resp., $N(\ell)$) to denote the set of particles adjacent to P (resp., to ℓ), where by *adjacent* we mean connected by a lattice edge. For adjacent locations ℓ and ℓ', by $N(\ell \cup \ell')$ we mean $(N(\ell) \cup N(\ell')) \setminus \{\ell, \ell'\}$. Let $\mathbb{S} = N(\ell) \cap N(\ell')$ be the set of particles adjacent to both ℓ and ℓ'; note that $|\mathbb{S}| \in \{0, 1, 2\}$.

Property 8.2.1. $|\mathbb{S}| \in \{1, 2\}$ *and every particle in* $N(\ell \cup \ell')$ *is connected to a particle in* $\mathbb{S}$ *by a path through* $N(\ell \cup \ell')$.

Property 8.2.2. $|\mathbb{S}| = 0$, *ℓ and ℓ' each have at least one neighbor, all particles in* $N(\ell) \setminus \{\ell'\}$ *are connected by paths within this set, and all particles in* $N(\ell') \setminus \{\ell\}$ *are connected by paths within this set.*

These properties capture precisely the structure required to maintain system connectivity and prevent certain new holes from forming as a particle moves from location ℓ to ℓ'. Additionally, both are symmetric for ℓ and ℓ', necessary for particle moves to be reversible. However, they are not so restrictive as to limit the movement of particles and prevent compression from occurring. We will see that after a burn-in phase to eliminate any holes, moves satisfying these properties suffice to transform any configuration into any other.

We now define our Markov chain $\mathcal{M}_C$ for compression. The state space Ω of $\mathcal{M}_C$ is the set of all connected configurations of n contracted particles, and the rules and probabilities given in Algorithm $\mathcal{M}_C$ define the transitions between states. Later, in Section 8.2.2, we will show how to view this Markov chain as a local, distributed algorithm $\mathcal{A}_C$. Both $\mathcal{M}_C$ and $\mathcal{A}_C$ take as input a bias parameter $\lambda > 1$ and begin at an arbitrary connected starting configuration $\sigma_0 \in \Omega$.

Algorithm 12 Markov Chain $\mathcal{M}_C$ for Compression

```
   From any connected configuration $\sigma_0$ of $n$ contracted particles, repeat:
1: Select particle $P$ uniformly at random from among all particles; let $\ell$ be its location.
2: Choose neighboring location $\ell'$ and $q \in (0,1)$ each uniformly at random.
3: **if** $\ell'$ is unoccupied **then**
4:     $P$ expands to simultaneously occupy $\ell$ and $\ell'$.
5:     Let $e = |N(\ell)|$ be the number of neighbors $P$ had when it was contracted at $\ell$.
6:     Let $e' = |N(\ell')|$ be the number of neighbors $P$ would have if it contracts to $\ell'$.
7:     **if** $(i)$ $e \neq 5$, $(ii)$ $\ell$ and $\ell'$ satisfy Property 8.2.1 or Property 8.2.2, and $(iii)$ $q < \lambda^{e'-e}$ **then**
8:         $P$ contracts to $\ell'$.
9:     **else** $P$ contracts back to $\ell$.
```

In Markov chain $\mathcal{M}_C$, note that a constant number of random bits suffices to generate q in Step 2, as only a constant precision is required (given that $e' - e$ is an integer in $[-3, 3]$ and λ is a constant). In Step 7, Condition (i) ensures no new holes form, Condition (ii) ensures the particle system stays connected and $\mathcal{M}_C$ is eventually ergodic, and Condition (iii) ensures the particle moves happen with probabilities such that $\mathcal{M}_C$ converges to the desired stationary distribution.

In practice, Markov chain $\mathcal{M}_C$ yields good compression. We simulated $\mathcal{M}_C$ for $\lambda = 4$ on 100 particles that began in a line; the configurations after 1, 2, 3, 4, and 5 million iterations of $\mathcal{M}_C$ are shown in Figure 36. In Section 8.3, we will rigorously prove that Markov chain $\mathcal{M}_C$ achieves compression with all but exponentially small probability whenever $\lambda > 2 + \sqrt{2}$ (Theorem 8.3.5).

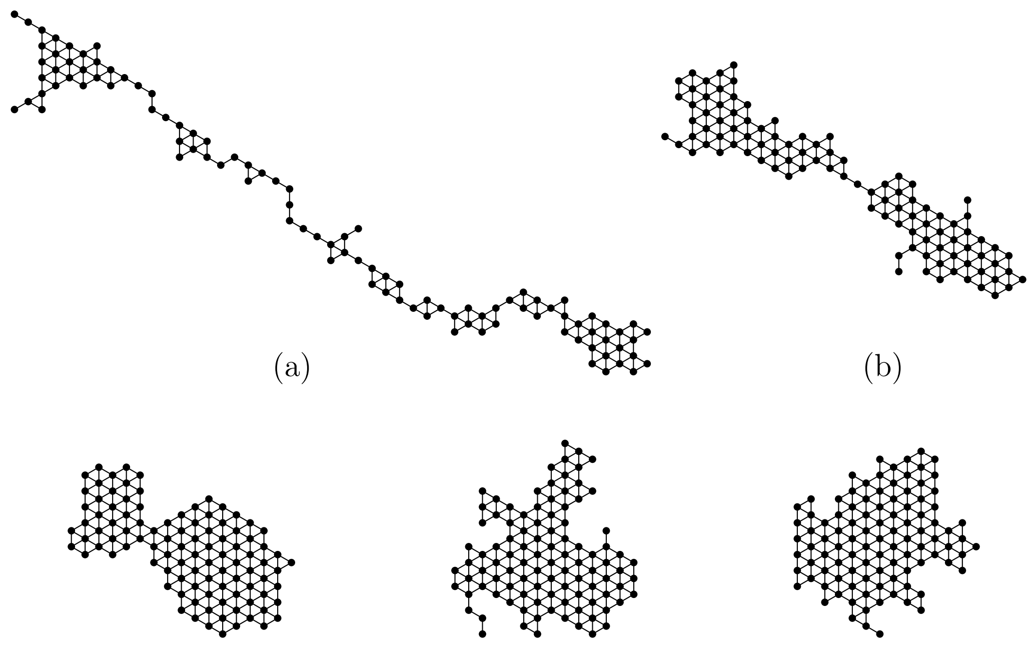


Figure 36. Simulation of Markov Chain $\mathcal{M}_C$ with $\lambda = 4$. A system of 100 particles initially in a line after (a) 1 million, (b) 2 million, (c) 3 million, (d) 4 million, and (e) 5 million iterations of $\mathcal{M}_C$ with bias $\lambda = 4$.

8.2.2 The Local, Distributed Algorithm $\mathcal{A}_C$ for Compression

In order for each particle to individually run $\mathcal{M}_C$, a Markov chain with centralized control, we show how $\mathcal{M}_C$ can be translated into a local, distributed algorithm $\mathcal{A}_C$ that satisfies the constraints of the amoebot model. There are two parts of this translation: (*i*) selecting particles uniformly at random as in Step 1 of $\mathcal{M}_C$ must be translated to a fair sequential adversary used to activate individual amoebots, and (*ii*) moving particles in a combined expansion and contraction as in Steps 4–9 of $\mathcal{M}_C$ must be decoupled into two separate activations since the amoebot model allows at most one movement per activation. All other steps of $\mathcal{M}_C$ can be directly implemented by an individual amoebot with constant-size memory using only information from its local neighborhood.

Choosing a particle at random in Step 1 of $\mathcal{M}_C$ enables us to explicitly calculate the stationary distribution of $\mathcal{M}_C$ so that we can provide rigorous guarantees about its structure. However, with a fair sequential adversary, one cannot assume that the next amoebot to be activated is equally likely to be any amoebot (see Section 2.2.4). To mimic this uniformly random activation sequence in a local way, we assume each amoebot has its own Poisson clock with mean 1 and activates after a delay t drawn with probability e^{-t}. After completing its activation (executing one iteration of Algorithm 13), a new delay is drawn to its next activation, and so on. The exponential distribution guarantees that, regardless of which amoebot has just activated, all amoebots are equally likely to be the next to activate (see, e.g., [83]). Moreover, amoebots proceed without requiring knowledge of any other amoebots' clocks. Similar Poisson clocks are commonly used to describe physical systems that perform concurrent updates in continuous time.

We can better approximate a fair sequential adversary by allowing each amoebot to have its own constant mean for its Poisson clock, allowing for some amoebots to activate more often than others in expectation. In this setting, the probability that a given amoebot A is the next of the n amoebots to activate is not $1/n$, but rather some probability p_A that depends on all amoebots' Poisson means.[13] This does not change the stationary distribution of $\mathcal{M}_C$ (i.e., Lemma 8.2.13 still holds with a nearly identical proof that replaces $1/n$ with p_A), and our main results (Theorem 8.3.5 and Corollary 8.3.6) still follow. Because the same results hold regardless of the rates of amoebots' Poisson clocks, we assume clocks with mean 1 for simplicity. We emphasize that although these Poisson activation sequences are necessary for rigorously proving

[13]Probability p_A would only play a role in the analysis of $\mathcal{A}_C$, not in its execution. Amoebot A does not need to know or calculate p_A.

the convergence of $\mathcal{M}_C$ — and, therefore, the correctness of $\mathcal{A}_C$ — we do not expect the system's behavior to be substantially different for non-Poisson activation sequences.

Algorithm 13 Local, Distributed Algorithm $\mathcal{A}_C$ for Compression

```
    If $A$ is contracted:
 1: Let $\ell$ denote the current location of $A$.
 2: Choose a neighboring location $\ell'$ uniformly at random from the six possible choices.
 3: **if** $\ell'$ is unoccupied and $A$ has no expanded neighbors **then**
 4:     $A$ expands to simultaneously occupy $\ell$ and $\ell'$.
 5:     **if** there are no expanded amoebots adjacent to $\ell$ or $\ell'$ **then**
 6:         $A$.flag $\leftarrow$ TRUE.
 7:     **else** $A$.flag $\leftarrow$ FALSE.

    If $A$ is expanded:
 8: Choose $q \in (0, 1)$ uniformly at random.
 9: Let $N^*(\cdot) \subseteq N(\cdot)$ be the set of neighboring amoebots excluding any heads of expanded amoebots.
10: Let $e = |N^*(\ell)|$ be the number of neighbors $A$ had when it was contracted at $\ell$.
11: Let $e' = |N^*(\ell')|$ be the number of neighbors $A$ would have if it contracts to $\ell'$.
12: **if** ($i$) $e \neq 5$, ($ii$) locations $\ell$ and $\ell'$ satisfy Property 8.2.1 or Property 8.2.2 with respect to $N^*(\cdot)$,
    ($iii$) $q < \lambda^{e'-e}$, and ($iv$) $A$.flag = TRUE **then**
13:     $A$ contracts to $\ell'$.
14: **else** $A$ contracts back to $\ell$.
```

We now turn to decoupling the combined expansion and contraction movement in a single state transition of $\mathcal{M}_C$ into two (not necessarily consecutive) activations of a given amoebot running $\mathcal{A}_C$. We must carefully handle the way in which an amoebot's neighborhood may change between its two activations, ensuring that at most one amoebot per neighborhood moves at a time, mimicking the sequential nature of $\mathcal{M}_C$. Each amoebot A continuously runs $\mathcal{A}_C$, executing Steps 1–7 if contracted, and Steps 8–14 if expanded. Conditions (i)–(iii) in Step 12 of $\mathcal{A}_C$ are analogous to those in Step 12 of $\mathcal{M}_C$, but treat expanded amoebots as if they are still contracted at their tail location, rather than considering all occupied neighboring locations. We use the additional Condition (iv) to ensure A is the only amoebot in its neighborhood moving to a new position since it last expanded, as we now explain in more detail. For the purposes of this analysis, recall from Section 2.2.4 that a fair sequential adversary activates one amoebot at a time and must activate each amoebot infinitely often.

Suppose an amoebot A eventually moves from location ℓ to location ℓ' by expanding to occupy both positions at some time t and contracting to ℓ' at some time $t' > t$ according to an execution of $\mathcal{A}_C$. Since A eventually completes its movement to ℓ', there must have been no expanded amoebots adjacent to ℓ or ℓ' at time t (by Step 6 and Condition (iv) of Step 12 in $\mathcal{A}_C$). Any other amoebot B that expands into the neighborhood of A in the time interval (t, t') will see that A is expanded and set its flag to FALSE in Step 7 of $\mathcal{A}_C$. Since any such neighbor B with a FALSE flag must contract back to its original position during its next activation (by Condition (iv) of Step 12 and Step 14 of $\mathcal{A}_C$), amoebot A can safely ignore any expanded heads in its neighborhood, making decisions in Steps 8–12 of $\mathcal{A}_C$ as if B had never moved. Thus, the neighborhood of A remains effectively undisturbed in the interval (t, t'), allowing $\mathcal{A}_C$ to faithfully emulate $\mathcal{M}_C$.

Any objective that can be accomplished by $\mathcal{M}_C$ can be accomplished by $\mathcal{A}_C$ and vice versa. Consider an activation sequence of amoebots executing $\mathcal{A}_C$ that transforms the initial configuration σ_0 to a configuration σ' that potentially contains both expanded and contracted amoebots. Obtain configuration σ from σ' by preserving the locations of all contracted amoebots and considering every expanded amoebot to be contracted at its tail. Then there exists a sequence of transitions in $\mathcal{M}_C$ that reaches σ. The perimeter $p(\sigma')$ ignores heads of expanded amoebots (Section 8.1.2), so $p(\sigma) = p(\sigma')$. Conversely, every sequence of transitions in $\mathcal{M}_C$ that reaches a configuration σ directly corresponds to a sequence of activations (expansions followed immediately by contractions) of amoebots executing $\mathcal{A}_C$ also leading to $\sigma' = \sigma$, where again $p(\sigma) = p(\sigma')$. Thus, proving α-compression for σ also implies α-compression for σ', and vice-versa. Hence, we can use $\mathcal{M}_C$ and the respective Markov chain tools and techniques in order to analyze the correctness of $\mathcal{A}_C$. Because we show α-compression for $\mathcal{M}_C$ for all $\alpha > 1$ (Theorem 8.3.5), this also then implies α-compression for $\mathcal{A}_C$ for all $\alpha > 1$. In subsequent sections, we focus on analyzing $\mathcal{M}_C$.

We have shown our Markov chain $\mathcal{M}_C$ can be translated into a local, distributed algorithm $\mathcal{A}_C$ with the same behavior, but such implementations are not always possible in general. Any Markov chain for particle systems that relies on non-local particle moves or has transition probabilities that depend on non-local information cannot be executed by a local, distributed algorithm. Moreover, many algorithms under the amoebot model are not stochastic and thus cannot be meaningfully described as Markov chains (e.g., those described in Chapters 5–7).

8.2.3 Obliviousness and Robustness of $\mathcal{M}_C$ and $\mathcal{A}_C$

Our algorithm for compression has two key advantages over previous algorithms for self-organizing particle systems: inherent *obliviousness* and *robustness*. An algorithm is *oblivious* if it is stateless; i.e., a particle remembers no information from past activations and decides what to do based only on its observations of its current environment. In practical settings, such algorithms are desirable because they do not require persistent memory and are often self-stabilizing and fault-tolerant (see, e.g., obliviousness in autonomous mobile robots [87]); theoretically, they are of great interest because they are computationally weak at an individual level but can still collectively accomplish sophisticated goals. Algorithm $\mathcal{A}_C$ for compression is the first nearly oblivious algorithm for self-organizing particle systems, as each particle only needs to store its flag variable as a single bit of information between its expansion and contraction activations. Previous works under the amoebot model (see, e.g., Chapters 5–7), however, rely heavily on persistent particle memory for decision making and communication.

Our algorithm is also the first for self-organizing particle systems to meaningfully consider fault-tolerance.[14] A distributed algorithm's *fault-tolerance* has to do with its ability to achieve its goals despite possible *crash failures* or *Byzantine failures*. In a crash failure, an agent abruptly ceases functioning and may never be resuscitated. These failures are particularly problematic for systems with a single point of failure, as there is no guarantee the critical agent will remain non-faulty nor that its memory and role could be assumed by another agent if it crashes. In a Byzantine failure, some fraction of the agents are malicious and execute arbitrary behavior in an effort to stop the non-faulty portion of the system from achieving its task.

Before we introduced our compression algorithm, work on self-organizing particle systems had not addressed either type of possible fault, and many of the proposed algorithms were susceptible to complete failure if even a single particle crashed. If one or more particles were to crash in our algorithm for compression, they would cease moving and act as fixed points around which the remaining particles would simply continue to compress. For the more adversarial setting of Byzantine failures, since our algorithm is (nearly) oblivious and communication is limited to particles checking the flags of their neighbors, the malicious particles are unable to "lie" or otherwise corrupt healthy particles' behaviors. We speculate that the malicious particles could affect the overall compression of the system by expanding away from where the system is aggregating and refusing to contract, essentially acting as fixed points. However, if

[14]After our compression algorithm was first introduced in [32], fault-tolerance for self-organizing particle systems was also considered by Di Luna et al. for shape formation problems in [67].

the fraction of malicious particles is small, the non-faulty particles will still be able to compress around the malicious particles, as in crash failures.

8.2.4 Invariants for $\mathcal{M}_C$

We have established that algorithm $\mathcal{A}_C$ is a distributed implementation of Markov chain $\mathcal{M}_C$ (Section 8.2.2), so we will perform the rest of our analysis directly on $\mathcal{M}_C$. We begin by showing that $\mathcal{M}_C$ maintains certain invariants.

Lemma 8.2.1. *If the particle system is initially connected, during the execution of Markov chain $\mathcal{M}_C$ it remains connected.*

Proof. Consider one iteration of $\mathcal{M}_C$ where a particle P moves from location ℓ to location ℓ'. Let σ be the configuration before this move, and σ' the configuration after. We show that if σ is connected, then so is σ'.

A move of particle P from ℓ to ℓ' occurs only if ℓ and ℓ' are adjacent and satisfy Property 8.2.1 or Property 8.2.2. First, suppose they satisfy Property 8.2.1. If σ is connected, then for every particle Q there exists some path $\mathcal{P} = (P = P_1, P_2, \ldots, P_k = Q)$ from P to Q in σ. By Property 8.2.1, since $P_2 \in N(\ell)$, there exists a path from P_2 to a particle $S \in \mathbb{S}$ that is entirely contained in $N(\ell)$. After P moves to location ℓ', it remains connected to particle Q by a (not necessarily simple) walk that first travels to S, then travels through $N(\ell)$ to P_2, and finally follows $\mathcal{P}$ to Q. This implies P is connected to all particles from location ℓ', so σ' is connected via paths through P.

Next, assume locations ℓ and ℓ' satisfy Property 8.2.2. Let Q and $\overline{Q} \neq P$ be particles; we show that if σ is connected, then Q and $\overline{Q}$ must be connected by a path not containing P. If σ is connected, then Q and $\overline{Q}$ are connected by some path $\mathcal{P} = (Q = Q_1, Q_2, \ldots, Q_k = \overline{Q})$. If P is not in this path we are done, so suppose this path contains P, that is, $Q_i = P$ for some $i \in \{2, \ldots, k-1\}$. Both Q_{i-1} and Q_{i+1} are neighbors of ℓ, and by Property 8.2.2 all neighbors of ℓ are connected by a path in $N(\ell)$. Thus $\mathcal{P}$ can be augmented to form a (not necessarily simple) walk $\mathcal{W}$ by replacing P with a path from Q_{i-1} to Q_{i+1} in $N(\ell)$. As $P \notin \mathcal{W}$, this walk connects Q and $\overline{Q}$ in σ' without going through P, as desired. Because any two particles $Q, \overline{Q} \neq P$ are connected by a path not containing P, they remain connected after P moves from ℓ to ℓ'. Additionally, because ℓ' has at least one neighbor by Property 8.2.2, P at location ℓ' is connected to at least one particle, and via that particle to all other particles in σ'. Thus σ' is connected. $\square$

We prove in the next subsection that $\mathcal{M}_C$ will eventually reach a configuration with no holes (Lemma 8.2.8). After that point, the following lemma will apply. While it is true more broadly that $\mathcal{M}_C$ will never create new holes, we prove only what we will need, that new holes are never created in a hole-free configuration.

Lemma 8.2.2. *All configurations reachable by Markov chain $\mathcal{M}_C$ from a connected configuration with no holes do not have holes.*

Proof. Consider one iteration of $\mathcal{M}_C$ where a particle P moves from location ℓ to location ℓ'. Let σ be the configuration before this move, and σ' the configuration after. We show if σ is hole-free, then so is σ'.

By a *cycle* in a configuration σ we will mean a cycle in G_Δ that surrounds at least one unoccupied location and whose vertices are occupied by particles of σ. Note a configuration has a hole if and only if it has a cycle. Throughout this proof, we will argue about the existence of cycles rather than the existence of holes.

We first show that if σ' has a cycle then that cycle must contain P. Suppose, for the sake of contradiction, this is not the case and σ' has a cycle $\mathcal{C}$ with $P \notin \mathcal{C}$. If P is removed from location ℓ', then cycle $\mathcal{C}$ still exists in $\sigma' - P$. If P is then placed at ℓ, yielding σ, then $\mathcal{C}$ still exists unless it had enclosed exactly one unoccupied location, ℓ. However, this is not possible as any cycle in $\sigma' - P$ encircling ℓ would also necessarily encircle neighboring unoccupied location ℓ'. This implies cycle $\mathcal{C}$ exists in cycle-free configuration σ, a contradiction. We conclude any cycle in σ' must contain P.

Because particle P moved from location ℓ to location ℓ' in a valid step of Markov chain $\mathcal{M}_C$, it must be true (by the conditions checked in Step 7 of $\mathcal{M}_C$) that ℓ has fewer than five neighbors and locations ℓ and ℓ' satisfy Property 8.2.1 or Property 8.2.2.

First, suppose they satisfy Property 8.2.2. While P might momentarily create a cycle when it expands to occupy both locations ℓ and ℓ', it will then contract to location ℓ'. Suppose P is part of some cycle $\mathcal{C} = (P = P_1, P_2, \ldots, P_{k-1}, P_k = P)$ in σ'. By Property 8.2.2, P_2 and P_{k-1} are connected by a path in $N(\ell')$ that doesn't contain P. Replacing path (P_{k-1}, P, P_2) in cycle $\mathcal{C}$ by this path in $N(\ell')$ yields a (not necessarily simple) cycle $\mathcal{C}'$ in σ' not containing P, a contradiction.

Next, suppose ℓ and ℓ' satisfy Property 8.2.1. Because particle P moved from ℓ to ℓ' in a valid step of $\mathcal{M}_C$, location ℓ must have at most four neighbors in σ. This means that in σ', location ℓ has at most five neighbors — its original neighbors plus P at location ℓ' — and thus is adjacent to at least one unoccupied location. Suppose there exists some cycle $\mathcal{C} = (P = P_1, P_2, \ldots, P_{k-1}, P_k = P)$ in σ'. This cycle encircles at least one unoccupied location $\ell'' \neq \ell$: since ℓ is adjacent to another unoccupied location in σ', it cannot be the case that ℓ is the only unoccupied location inside $\mathcal{C}$. If there exists a path between P_2 and P_{k-1} in $N(\ell')$, the argument from the previous case applies and we are done. Otherwise, w.l.o.g., it must be that $|\mathbb{S}| = 2$ and there exist paths in $N(\ell \cup \ell')$ from P_{k-1} to $S_1 \in \mathbb{S}$ and from P_2 to $S_2 \in \mathbb{S}$, with $S_1 \neq S_2$. There then exists a (not necessarily simple) cycle $\mathcal{C}^*$ in σ obtained from $\mathcal{C}$ by replacing path

(P_{k-1}, P, P_2), where P is in location ℓ', with path $(P_{k-1}, \ldots, S_1, P, S_2, \ldots, P_2)$, where P is in location ℓ. $\mathcal{C}^*$ is a valid cycle in σ because it encircles unoccupied location $\ell'' \neq \ell$. This is a contradiction because σ has no cycles. We conclude by contradiction that, in all cases, σ' must have no cycles, and thus must have no holes. $\square$

8.2.5 Eventual Ergodicity of $\mathcal{M}_C$

The state space Ω of our Markov chain $\mathcal{M}_C$ is the set of all connected configurations of n contracted particles, and Lemma 8.2.1 ensures that we always stay within this state space. The initial configuration σ_0 of $\mathcal{M}_C$ may or may not have holes. By Lemma 8.2.2, once a hole-free configuration is reached, $\mathcal{M}_C$ remains in the part of the state space consisting of all hole-free connected configurations, which we call Ω^*. In this section, we prove that from any starting state $\mathcal{M}_C$ always reaches Ω^*. Furthermore, we prove that $\mathcal{M}_C$ is irreducible on Ω^*, that is, for any two configurations in Ω^* there exists some sequence of moves between them that has positive probability. Stated another way, what we show is that all states in Ω^* are *recurrent*, meaning once $\mathcal{M}_C$ reaches a state $\sigma \in \Omega^*$ it returns to σ with probability 1, while all states in $\Omega \setminus \Omega^*$ are *transient*, meaning they are not recurrent. As $\mathcal{M}_C$ is also aperiodic, we can conclude it is eventually ergodic on Ω^*, a necessary precondition for all of the Markov chain analysis to follow.

We note the details of these proofs have been substantially simplified and clarified from the originally published conference version of these results [32], where the proof of ergodicity required over 10 pages of detailed analysis. Figure 37 illustrates one difficulty. It depicts a hole-free configuration for which there exist no valid moves

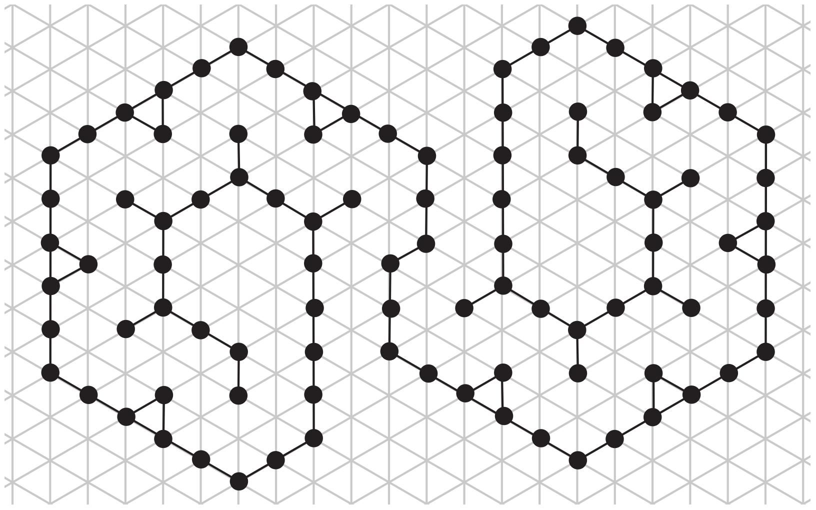


Figure 37. The Necessity of Jump Moves in $\mathcal{M}_C$. A connected configuration for which all valid moves of Markov chain $\mathcal{M}_C$ satisfy Property 8.2.2; no particle has a valid move satisfying Property 8.2.1. This demonstrates the subtlety of the Markov chain rules we have defined.

satisfying Property 8.2.1; the only valid moves satisfy Property 8.2.2. Thus if moves satisfying Property 8.2.2 are not included, neither Ω nor Ω^* is connected.

At a high level, we prove that for any configuration σ there exists a sequence of valid particle moves transforming σ into a straight line. Since a straight line is hole-free, this shows that from any initial configuration in Ω, there exists a sequence of moves with non-zero probability reaching Ω^*, as desired. We then prove any moves of $\mathcal{M}_C$ among states of Ω^* are reversible, which implies that for any $\tau \in \Omega^*$ there exists a sequence of valid particle moves transforming a straight line into τ. Altogether, this shows for any $\sigma, \tau \in \Omega^*$ there exists a sequence of valid moves (within Ω^*) transforming σ into τ, as required for ergodicity.

We will let m_1 be the vertical lattice line containing the leftmost particle(s) in σ. We label the subsequent vertical lattice lines as m_2, m_3, m_4, and so on. The process for moving the particles into one straight line is a sweep line algorithm, an approach often used in computational geometry [89, 181]. We first consider the particles in

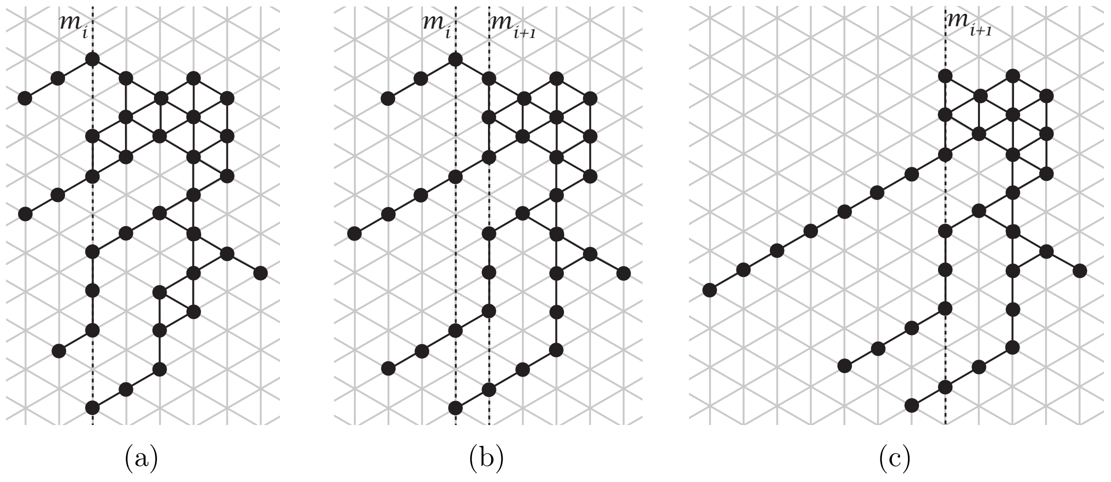


Figure 38. The Line Invariants for Ergodicity. (a) An example of a configuration and a line m_i that satisfies both invariants. (b) After a sequence of moves described in Lemma 8.2.4, m_{i+1} satisfies Invariant 8.2.1. (c) After a sequence of moves described in Lemma 8.2.5, m_{i+1} also satisfies Invariant 8.2.2.

leftmost vertical line m_1, then the particles in m_2, and so on; when considering line m_i, we maintain the following invariants:

Invariant 8.2.1. *All particles left of m_i form lines stretching down and left.*

Invariant 8.2.2. *Each such line stretches down and left from a particle in m_i that has an empty location directly below it.*

Line m_1 trivially satisfies these properties as there are no particles to its left. Figure 38a gives an example of an intermediate configuration and a line m_i satisfying these invariants. Starting from a configuration in which the invariants are satisfied for m_i, we describe how to find a sequence of valid particle moves after which m_{i+1} also satisfies the invariants. For the configuration in Figure 38a, the configuration obtained after first ensuring m_{i+1} satisfies Invariant 8.2.1 is shown in Figure 38b, and the configuration obtained after ensuring m_{i+1} also satisfies Invariant 8.2.2 is shown in Figure 38c.

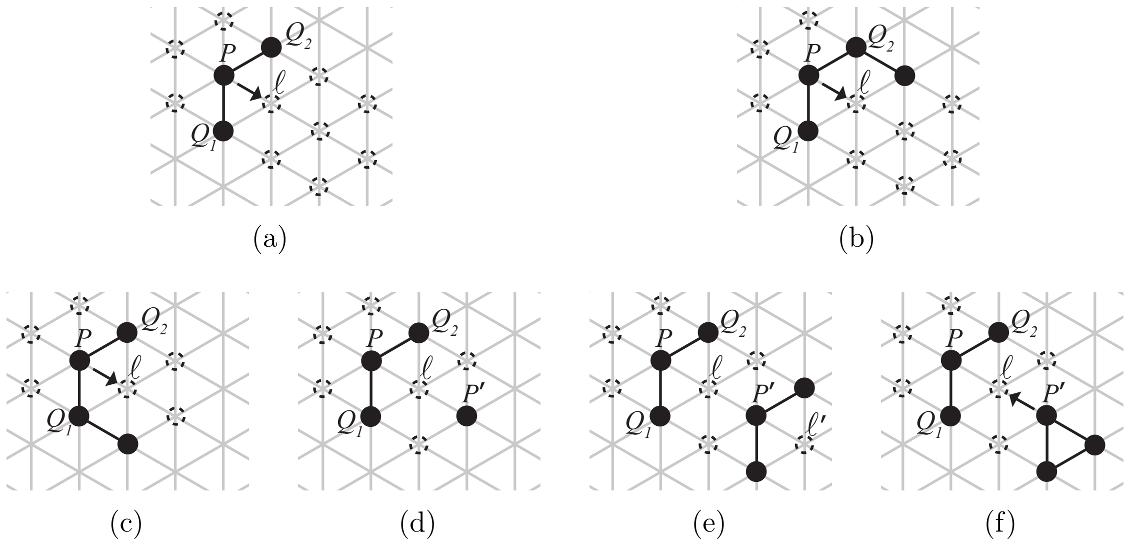


Figure 39. Local Moves for Line Setup. Particle positions from the base case (top row) and inductive step (bottom row) of the proof of Lemma 8.2.3. Particles are represented by black dots, and unoccupied locations are represented by dashed circles. Neighboring particles have a black line drawn between them.

Throughout this subsection, a *component of line* m_i will refer to a maximal collection of particles in m_i that are connected via paths in m_i. For example, in Figure 38a, m_i has four components (from top to bottom: of one, two, three, and one particles, respectively). We begin with a lemma about particle movements that will play a key role.

Lemma 8.2.3. *Suppose particle P has exactly two neighbors, Q_1 below it and Q_2 above-right of it, and let ℓ be the unoccupied location below-right of P. There exists a sequence of valid moves, occurring strictly below and right of P, after which either it is valid for P to move to ℓ or some other particle has already moved to ℓ.*

Proof. We induct on the number of particles strictly below and right of P. If there are no such particles, then it is valid (satisfying Property 8.2.1) for P to move from its current location ℓ_0 to ℓ. This is because $N(\ell_0) \cap N(\ell) = \{Q_1, Q_2\}$, and either these are the only two particles in $N(\ell_0 \cup \ell)$ (Figure 39a) or there is exactly one other particle in $N(\ell_0 \cup \ell)$ and it is adjacent to Q_2 (Figure 39b). Thus the conclusions of the lemma are already satisfied with an empty set of moves.

Suppose there are $k > 0$ particles strictly below and right of P, and for all $0 \leq k' < k$ the lemma holds. If it is already valid for P to move to ℓ, we are done; an example is given in Figure 39c. Otherwise, since P has fewer than five neighbors, it must be that neither Property 8.2.1 nor Property 8.2.2 is satisfied. Note $\mathbb{S} = N(P) \cap N(\ell)$ contains two particles, Q_1 and Q_2. Because Property 8.2.1 doesn't hold, and $N(P)$ doesn't contain any particles other than those of $\mathbb{S}$, it must be that there is a particle P' in $N(\ell)$ that is not connected to a particle in $\mathbb{S}$ by a path within $N(\ell)$. Then P' must occupy the location below-right of ℓ, and the locations adjacent to both ℓ and P' must be unoccupied; see Figure 39d. We now consider $N(P')$, which is of size at least one and at most three.

First, we suppose $N(Q')$ is not connected; see Figure 39e. In this case, P' must have exactly two neighbors, one below P' and the other above-right of P', while location ℓ' below-right of P' is unoccupied. There are fewer than k particles below and right of P' because this is a proper subset of the k particles below and right of P. By the induction hypothesis, we conclude there is a sequence of moves occurring entirely below and right of P' after which either it is valid for P' to move to ℓ' or another particle has moved to ℓ'. In the first case, we let P' move to ℓ' and afterwards it is valid (satisfying Property 8.2.1) for P to move to ℓ, because $N(\ell)$ now contains only Q_1 and Q_2. In the second case, a particle has moved to ℓ' but $N(P')$ otherwise remains unchanged, causing $N(P')$ to now be connected, the case we consider next.

Otherwise, suppose $N(P')$, which is of size at least one and at most three, is connected; see Figure 39f. Note the current location of P' and location ℓ satisfy

Property 8.2.2, so particle P' can move to ℓ. As P' and ℓ are below and right of P, this move satisfies the conclusions of the lemma. □

If m_i satisfies the invariants, we want to give a sequence of moves after which m_{i+1} also satisfies the invariants. The following lemma will be used towards that goal.

Lemma 8.2.4. *If m_i satisfies Invariants 8.2.1 and 8.2.2 and has a component of size at least two, there exists a sequence of valid moves that decreases the number of particles in m_i, after which m_i still satisfies the invariants.*

Proof. Consider any component of m_i of size at least two, and let P be the topmost particle in this component. Then P has a particle below it, no particle above it, and (by Invariants 8.2.1 and 8.2.2) no particle above-left or below-left of it. The two locations right of P may or may not be occupied. We consider two cases: when $N(P)$ is connected, and when it is not.

When $N(P)$ is disconnected, we invoke Lemma 8.2.3. It must be that P has two neighbors that satisfy the conditions of the lemma, and so there exists a sequence of valid moves after which either location ℓ below-right of P is occupied by another particle or it is valid for P to move to ℓ. All moves in this sequence occur right of P, and thus don't affect the invariants for m_i. If it is now valid for P to move to ℓ, we make this move and the number of particles in m_i has decreased, as desired. If another particle has moved to ℓ, then $N(P)$ is now connected, the next case we consider.

When $N(P)$ is connected, it must look as in Figures 40a, 40b, or 40c. In all cases, particle P moving down-left is a valid move that decreases the number of particles in m_i. However, Invariant 8.2.1 no longer holds for m_i after this move, so we continue to move particle P down until it is adjacent to the bottom particle Q in this component

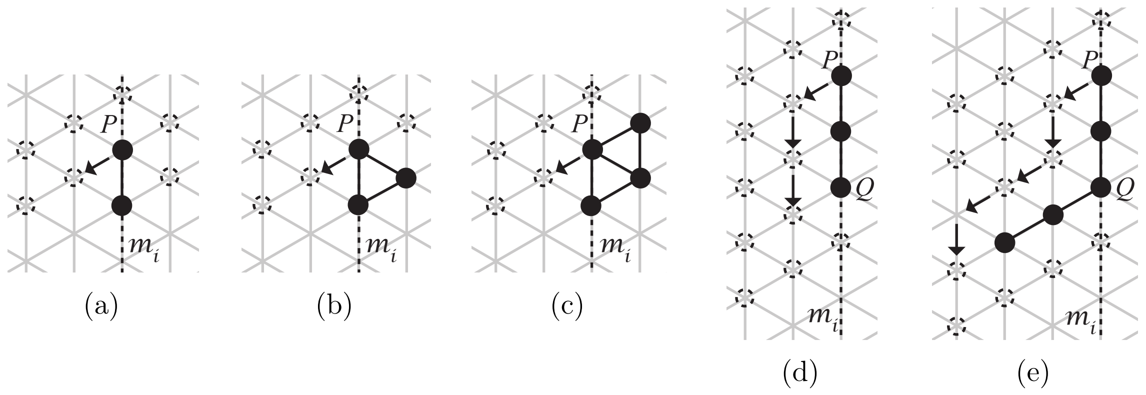


Figure 40. Local Moves for Line Formation. If P is the topmost particle in a component of m_i of size at least two and its neighborhood is connected, then (a)–(c) are the three possibilities for $N(P)$. In all three of these cases, moving P down-left satisfies Property 8.2.1. (d) and (e) show the two cases for subsequently moving P to a new position such that the invariants still hold for m_i.

of particles in m_i. If there is not already a line stretching down and left from Q, then P moves down once more to start such a line (Figure 40e), which is valid because of the invariants for m_i. If this line stretching down and left from Q already exists, we note the locations at distances one and two above this line must all be unoccupied. This follows from Invariants 8.2.1 and 8.2.2 for m_i: all particles left of m_i must extend down and left from the bottom particle of some component in m_i, and the first such particle above Q is at least two units above P's original location and thus at least three units above Q. Thus, it is valid (satisfying Property 8.2.1) to move P along this line and add it to the end (Figure 40d). In all cases, the number of particles in m_i decreases while the invariants for m_i remain satisfied, as desired. $\square$

Lemma 8.2.4 can be applied iteratively until all components of m_i are of size one and all particles left of m_i form lines stretching down-left from these components of size one. Thus, all particles left of m_{i+1} form lines stretching down-left, satisfying Invariant 8.2.1 for m_{i+1}. We now consider how to also satisfy Invariant 8.2.2 for m_{i+1}.

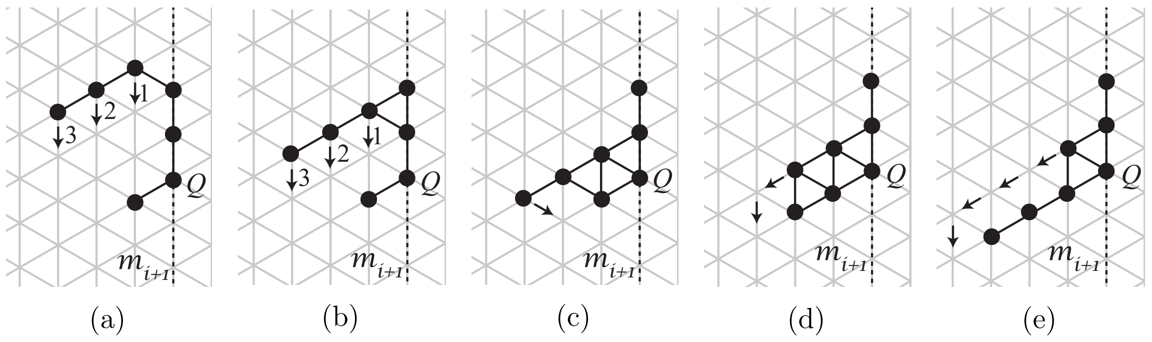


Figure 41. Local Moves for Merging Lines. (a)–(b) Each line can be moved down by iteratively moving the line's particles down one space from right to left. (c)–(e) Once flush with the bottommost line, particles from this line can move left and down to merge with the bottommost line, satisfying Invariant 8.2.2.

Lemma 8.2.5. *If m_i satisfies both invariants and m_{i+1} satisfies Invariant 8.2.1, then there exists a sequence of valid moves after which m_{i+1} satisfies both invariants.*

Proof. Because the configuration is connected, each line left of m_{i+1} is connected to some particle in m_{i+1}. However, the line may not stretch down and left from this particle or this particle may not have an empty location below it, as is required by Invariant 8.2.2. Consider any component of m_{i+1} which is adjacent to at least one line left of m_{i+1} stretching down-left. To satisfy Invariant 8.2.2, we merge all such lines into one, stretching down-left from the bottom particle Q in this component. First, we move the lowest line so that it is stretching down-left from Q. An entire line can be moved down one unit by first moving the rightmost particle in this line (the particle in line m_i) down once, which is necessarily a valid move, and then by subsequently moving the remaining particles down once from right to left (for an example of this downward movement of a line, see Figure 41a). This can be repeated until this lowest line is in the desired position, stretching down and left from Q.

Iteratively consider the next lowest line. As before, we move this line down one unit at a time by moving the particles each down once from right to left until the line is flush with the bottommost line (Figures 41a–41b). The particles in this line can then easily be added to the bottommost line one at a time, from left to right, as in Figures 41c–41e. We repeat this line merging process until all particles stretching down-left from this component of m_i have been reorganized into one line stretching down-left from Q. After repeating this process for all components in m_{i+1}, Invariant 8.2.2 is satisfied for m_{i+1}. Invariant 8.2.1 still holds for m_{i+1} as all particles are still in lines, so m_{i+1} now satisfies both invariants, as claimed. □

We now combine the previous two lemmas to get the main inductive step for our sweep-line procedure.

Lemma 8.2.6. *If m_i satisfies both invariants, then there exists a sequence of valid particle moves after which m_{i+1} also satisfies both invariants.*

Proof. Suppose m_i satisfies both invariants. If there are connected components of two or more particles contained in m_i, we can iteratively apply Lemma 8.2.4 to reduce the number of particles in m_i without affecting the invariants. After this, all components of m_i consist of one particle. Now all particles left of m_{i+1} are in lines (possibly consisting of just one particle) stretching down-left, satisfying Invariant 8.2.1. Next, we can apply Lemma 8.2.5 to ensure that m_{i+1} also satisfies Invariant 8.2.2, merging any lines stretching down-left from the same component of m_{i+1}. Thus, there exists a sequence of valid moves after which m_{i+1} satisfies both invariants, as claimed. □

Lemma 8.2.7. *There exists a valid sequence of moves transforming any configuration into a line.*

Proof. Initially, m_1 for σ trivially satisfies the invariants because there are no particles left of m_1. Repeatedly using Lemma 8.2.6, we obtain a sequence of moves after which the invariants hold for some line m_k which has no particles to its right.

All particles in m_k must be in a single component. If this was not the case, then the configuration would not be connected: particles left of m_k only form lines that are insufficient to connect multiple components of m_k, and there are no particles right of m_k. We know that the configuration must be connected because initial configuration σ_0 was connected and we have only made valid particle moves (Lemma 8.2.1), so this is a contradiction, and m_k must have a single component.

We repeatedly apply Lemma 8.2.4 until there is only one particle left in m_k and line m_k still satisfies the invariants. At this point the particles form a single line stretching down-left from the single particle in m_k, and we have given a sequence of valid moves transforming an arbitrary configuration into a line. □

In particular, this shows that for any connected configuration there exists a valid sequence of moves transforming it into a configuration with no holes.

Lemma 8.2.8. *Eventually $\mathcal{M}_C$ reaches a configuration with no holes, after which no holes are ever introduced again.*

Proof. Let $\sigma_0 \in \Omega$ be the initial (connected) configuration given as input to Markov chain $\mathcal{M}_C$. By Lemma 8.2.7 for σ_0, there is positive probability that $\mathcal{M}_C$ will reach $\Omega^* \subset \Omega$, the set of hole-free connected particle configurations. Lemma 8.2.7 holds for any configuration, so this is also true of each subsequent state σ_i. Since Ω is finite, $\mathcal{M}_C$ must eventually reach Ω^*, as desired. Finally, by Lemma 8.2.2, once Ω^* is reached, the particle system will remain hole-free for the rest of $\mathcal{M}_C$'s execution. □

Note that the previous lemma is equivalent to saying that any configuration with a hole is a transient state of Markov chain $\mathcal{M}_C$. We present one more lemma before proving $\mathcal{M}_C$ is irreducible on Ω^* once it reaches Ω^*. Let M be the transition matrix of $\mathcal{M}_C$, that is, $M(\sigma, \tau)$ is the probability of moving from state σ to state τ in one step of $\mathcal{M}_C$.

Lemma 8.2.9. *For any two configurations $\sigma, \tau \in \Omega^*$, if $M(\sigma, \tau) > 0$ then $M(\tau, \sigma) > 0$; that is, once $\mathcal{M}_C$ reaches Ω^*, all of its transitions are reversible on Ω^*.*

Proof. Let $\sigma, \tau \in \Omega^*$ be any two configurations such that $M(\sigma, \tau) > 0$. Then σ and τ differ by one particle P that is at location ℓ in σ and at an adjacent location ℓ' in τ.

In τ, particle P at location ℓ' has at most four neighbors. It cannot have six neighbors because location ℓ, which was previously occupied by P in σ, is now unoccupied. It cannot have five neighbors because otherwise ℓ' would have been a hole in σ when P was at ℓ, a contradiction to our assumption that $\sigma \in \Omega^*$. Because $M(\sigma, \tau) > 0$, Property 8.2.1 or Property 8.2.2 must hold for ℓ and ℓ'. Both properties are symmetric with regard to the role played by ℓ and ℓ'. Thus, if Markov chain $\mathcal{M}_C$ in state τ selects particle P, neighboring location ℓ, and sufficiently small probability q in Step 2, then because Conditions (i)–(iii) of Step 7 are satisfied, particle P moves to location ℓ. This proves $M(\tau, \sigma) > 0$. □

Lemma 8.2.10. *Once Markov chain $\mathcal{M}_C$ reaches Ω^*, it is irreducible on Ω^*, the state space of all connected configurations without holes.*

Proof. Let σ and τ be any two connected configurations of n particles with no holes. By Lemma 8.2.7, there exists a sequence of valid moves transforming σ into a line. By Lemmas 8.2.7 and 8.2.9, there exists a sequence of valid moves transforming this line into τ. □

Corollary 8.2.11. *Once $\mathcal{M}_C$ reaches Ω^*, it is ergodic on Ω^*.*

Proof. By Lemma 8.2.10, $\mathcal{M}_C$ is irreducible on Ω^*. As long as $n > 1$, then since the configuration is connected, every particle has at least one neighbor. Thus, at each step of $\mathcal{M}_C$ there is at least a 1/6 probability that no move is made because a particle proposes moving into an adjacent location that is already occupied, so $\mathcal{M}_C$ is aperiodic. Thus, once $\mathcal{M}_C$ reaches Ω^*, it is ergodic on Ω^*. □

We note that $\mathcal{M}_C$ is not irreducible on Ω, and thus not ergodic on Ω, because a configuration with a hole cannot be reached from a hole-free configuration. Ergodicity is necessary to apply tools from Markov chain analysis, as we do in the next subsection, which is why we focus on the behavior of $\mathcal{M}_C$ after it reaches Ω^*.

8.2.6 The Stationary Distribution π of $\mathcal{M}_C$

In this section we determine the stationary distribution of $\mathcal{M}_C$.

Lemma 8.2.12. *If π is a stationary distribution of $\mathcal{M}_C$, then for any $\sigma \in \Omega \setminus \Omega^*$, $\pi(\sigma) = 0$.*

Proof. For any configuration $\sigma \in \Omega \setminus \Omega^*$, there is a positive probability of moving into Ω^* in some later time step (Lemma 8.2.8). For any configuration $\tau \in \Omega^*$, there is zero probability of reaching a configuration with holes (Lemma 8.2.2). If a stationary distribution π were to put any positive probability mass on states in $\Omega \setminus \Omega^*$, over time the total probability mass within $\Omega \setminus \Omega^*$ would decrease as it leaks into Ω^* with no possibility of returning. Thus such a distribution could not be stationary, a contradiction. We conclude that any stationary distribution π of $\mathcal{M}_C$ has $\pi(\sigma) = 0$ for all $\sigma \in \Omega \setminus \Omega^*$, as claimed. □

Lemma 8.2.13. *Markov chain $\mathcal{M}_C$ has a unique stationary distribution π given by:*

$$\pi(\sigma) = \begin{cases} \frac{\lambda^{e(\sigma)}}{Z} & \sigma \in \Omega^* \\ 0 & \sigma \in \Omega \setminus \Omega^* \end{cases}$$

where $Z = \sum_{\sigma \in \Omega^*} \lambda^{e(\sigma)}$ *is the normalizing constant, also called the partition function.*

Proof. Lemma 8.2.12 guarantees that any stationary distribution of $\mathcal{M}_C$ has $\pi(\sigma) = 0$ for configurations $\sigma \notin \Omega^*$. Once $\mathcal{M}_C$ reaches Ω^* (which it is guaranteed to by Lemma 8.2.8), it is ergodic on Ω^* (Corollary 8.2.11). We conclude, because Ω^* is finite, that $\mathcal{M}_C$ on Ω^* has a unique stationary distribution, and thus $\mathcal{M}_C$ on Ω also has a unique stationary distribution.

We confirm that π as stated above is this unique stationary distribution by detailed balance. Let σ and τ be configurations in Ω^* with $\sigma \neq \tau$ such that $M(\sigma, \tau) > 0$. By Lemma 8.2.9, we have $M(\tau, \sigma) > 0$. Suppose particle P moves from location ℓ in σ to neighboring location ℓ' in τ. Let e be the number of edges formed by P has when it is in location ℓ, and let e' be that number when P is in location ℓ'. This implies $e(\sigma) - e(\tau) = e - e'$. If $\lambda^{e'} \leq \lambda^e$, then we see that

$$M(\sigma, \tau) = \frac{1}{n} \cdot \frac{1}{6} \cdot \lambda^{e'-e} \quad \text{and} \quad M(\tau, \sigma) = \frac{1}{n} \cdot \frac{1}{6} \cdot 1.$$

In this case we can verify that σ and τ satisfy the detailed balance condition:

$$\pi(\sigma) M(\sigma, \tau) = \frac{\lambda^{e(\sigma)}}{Z} \cdot \frac{\lambda^{e'-e}}{6n} = \frac{\lambda^{e(\tau)}}{Z \cdot 6n} = \pi(\tau) M(\tau, \sigma).$$

If $\lambda^{e'} > \lambda^e$, we can similarly calculate these probabilities to verify detailed balance:

$$M(\sigma, \tau) = \frac{1}{n} \cdot \frac{1}{6} \cdot 1 \quad \text{and} \quad M(\tau, \sigma) = \frac{1}{n} \cdot \frac{1}{6} \cdot \lambda^{e-e'},$$

$$\pi(\sigma) M(\sigma, \tau) = \frac{\lambda^{e(\sigma)}}{Z \cdot 6n} = \frac{\lambda^{e(\tau)}}{Z} \frac{\lambda^{e-e'}}{6n} = \pi(\tau) M(\tau, \sigma).$$

Since the detailed balance condition is satisfied for all $\sigma, \tau \in \Omega^*$, it only remains to verify that π is in fact a probability distribution:

$$\sum_{\sigma \in \Omega} \pi(\sigma) = \sum_{\sigma \in \Omega^*} \frac{\lambda^{e(\sigma)}}{Z} + \sum_{\sigma \in \Omega \setminus \Omega^*} 0 = \frac{\sum_{\sigma \in \Omega^*} \lambda^{e(\sigma)}}{\sum_{\sigma \in \Omega^*} \lambda^{e(\sigma)}} = 1.$$

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

While it is natural to assume maximizing the number of edges in a configuration results in more compression, here we formalize this. We prove π can also be expressed in terms of perimeter. This implies $\mathcal{M}_C$ converges to a distribution weighted by the perimeter of configurations, a global characteristic, even though the probability of any particle move is determined only by local information.

Corollary 8.2.14. *The stationary distribution π of $\mathcal{M}_C$ is also given by:*

$$\pi(\sigma) = \begin{cases} \frac{\lambda^{-p(\sigma)}}{Z} & \sigma \in \Omega^* \\ 0 & \sigma \in \Omega \setminus \Omega^* \end{cases}$$

where $Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)}$ is the normalizing constant, also called the partition function.

Proof. This expression is equal to the formulation given in Lemma 8.2.13 when $\sigma \notin \Omega^*$, so it suffices to verify the case $\sigma \in \Omega^*$. We use Lemma 8.1.3 and Lemma 8.2.13:

$$\pi(\sigma) = \frac{\lambda^{e(\sigma)}}{\sum_{\sigma \in \Omega^*} \lambda^{e(\sigma)}} = \frac{\lambda^{3n-p(\sigma)-3}}{\sum_{\sigma \in \Omega^*} \lambda^{3n-p(\sigma)-3}} = \frac{\lambda^{3n-3}}{\lambda^{3n-3}} \cdot \frac{\lambda^{-p(\sigma)}}{\sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)}} = \frac{\lambda^{-p(\sigma)}}{\sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)}},$$

yielding the desired result. □

The original version of these results [32] also expressed the stationary distribution in terms of the number of triangles in a configuration. Recall a *triangle* is a face of G_Δ that has all three of its vertices occupied by particles and $t(\sigma)$ is the number of triangles in configuration σ. We include the following corollary for completeness, but will not use it in subsequent sections.

Corollary 8.2.15. *The stationary distribution π of $\mathcal{M}_C$ is also given by*

$$\pi(\sigma) = \begin{cases} \frac{\lambda^{t(\sigma)}}{Z} & \sigma \in \Omega^* \\ 0 & \sigma \in \Omega \setminus \Omega^* \end{cases}$$

where $Z = \sum_{\sigma \in \Omega^*} \lambda^{t(\sigma)}$ is the normalizing constant, also called the partition function.

Proof. This follows from Lemma 8.1.4 and Corollary 8.2.14:

$$\pi(\sigma) = \frac{\lambda^{-p(\sigma)}}{\sum_{\sigma\in\Omega^*}\lambda^{-p(\sigma)}} = \frac{\lambda^{-(2n-t(\sigma)-2)}}{\sum_{\sigma\in\Omega^*}\lambda^{-(2n-t(\sigma)-2)}} = \frac{\lambda^{-2n+2}}{\lambda^{-2n+2}}\cdot\frac{\lambda^{t(\sigma)}}{\sum_{\sigma\in\Omega^*}\lambda^{t(\sigma)}} = \frac{\lambda^{t(\sigma)}}{\sum_{\sigma\in\Omega^*}\lambda^{t(\sigma)}},$$

yielding the desired result. □

8.2.7 Convergence Time of $\mathcal{M}_C$

We prove in Section 8.3 that when $\lambda > 2+\sqrt{2}$, the particle system will be compressed at stationarity with all but exponentially small probability. We do not give explicit bounds on the time required for this to occur, and moreover we believe proving rigorous bounds will be challenging. A common measure of convergence time of a Markov chain is its *mixing time*, the number of steps until the distribution is within total variation distance ε of the stationary distribution, starting from the worst initial configuration. Getting a polynomial bound on the mixing time of our Markov chain $\mathcal{M}_C$ is likely to be challenging because of its similarity to physical systems such as the ferromagnetic Ising and Potts models, where sites are assigned spins and nearest neighbors prefer to have the same spins. In our case, the particles can be thought of as one spin and the empty sites another, and the Markov chain $\mathcal{M}_C$ also favors configurations with more like spins adjacent. Local Markov chains on models like this, including the two-dimensional Ising model with constant boundary conditions, are believed to have polynomial mixing time; however, proving such a bound remains a difficult open problem despite breakthrough works showing subexponential [138] and subsequently quasipolynomial [135] upper bounds on the mixing time. The shrinkage over time of the boundary of the particle configuration under $\mathcal{M}_C$ is similar to the shrinkage over time in the Ising model of "droplets" of one state surrounded by the other state (see, e.g., [34] for work investigating such droplets in two and three dimensions). This shrinking of droplets is believed — but not proved — to be the salient feature determining the mixing time for the Ising model with constant boundary conditions. Because our algorithm has similarly hard-to-analyze features, we expect getting rigorous mixing time bounds will be challenging.

It is also worth noting that mixing time may not be the best measure of our algorithm's ability to achieve compression. While we prove in later sections that compression with high probability once $\mathcal{M}_C$ has reached its stationary distribution, compressed configurations could be reached much earlier. When starting from a line of n particles, preliminary simulations indicate that doubling the number of particles consistently results in about a ten-fold increase in iterations until compression is achieved. Based on this, we conjecture the number of iterations of $\mathcal{M}_C$ until compression occurs is $\Omega(n^3)$ and $\mathcal{O}(n^4)$, the equivalent of $\Omega(n^2)$ and $\mathcal{O}(n^3)$ sequential rounds of $\mathcal{A}_C$. Furthermore, we do not expect the presence of holes in the initial configuration to significantly delay compression, even though this may increase the mixing time.

8.3 Achieving Compression

We proved in Section 8.2.6 that Markov chain $\mathcal{M}_C$ converges to a unique stationary distribution, and we know that distribution exactly (Corollary 8.2.14). In this section, we show that when parameter λ is large enough, this stationary distribution exhibits compression with high probability. While compression could actually occur even earlier, before $\mathcal{M}_C$ is close to stationarity, our proofs rely on analyzing the stationary distribution of $\mathcal{M}_C$.

Recall for any $\alpha > 1$ we say a configuration σ with n particles is α-compressed if its perimeter $p(\sigma) < \alpha \cdot p_{min}$, where p_{min} is the minimum possible perimeter of a configuration with n particles. We prove that, for any $\alpha > 1$ and provided λ and n are large enough, a configuration chosen at random according to the stationary distribution of $\mathcal{M}_C$ is α-compressed with all but a probability that is exponentially small (in $\sqrt{n}$). Values of α closer to 1 simply require larger λ values. Conversely, we then prove (as a corollary) that for any $\lambda > 2 + \sqrt{2}$, there is a constant α such that with high probability α-compression occurs at stationarity.

8.3.1 Perimeter and Self-Avoiding Walks

We begin with some necessary results bounding the number of connected, hole-free particle system configurations with a certain perimeter. Let S_α be the set of all connected, hole-free configurations with perimeter at least $\alpha \cdot p_{min}$, for some constant $\alpha > 1$. We only consider hole-free configurations because we are concerned with behavior of $\mathcal{M}_C$ at stationarity and the stationary distribution π of $\mathcal{M}_C$ only gives positive probability to hole-free configurations in Ω^* (Corollary 8.2.14). We want an exponentially small upper bound on $\pi(S_\alpha) = \sum_{\sigma \in S_\alpha} \pi(\sigma)$, the probability of being in a configuration with large perimeter at stationarity.

Let c_k denote the number of connected, hole-free configurations with perimeter k. Recall that $p_{max} = 2n - 2$ is the maximum possible perimeter for a configuration of n particles; using the expression for π given in Corollary 8.2.14, we can write $\pi(S_\alpha)$ as:

$$\pi(S_\alpha) = \sum_{\sigma \in S_\alpha} \pi(\sigma) = \sum_{\sigma \in S_\alpha} \frac{\lambda^{-p(\sigma)}}{Z} = \frac{\sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{p_{max}} c_k \lambda^{-k}}{Z}.$$

Recall that Corollary 8.2.14 defined the partition function as $Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)}$, the summed weight of all connected, hole-free configurations. In order to give an upper bound on $\pi(S_\alpha)$, we establish a lower bound on Z and an upper bound on c_k. It

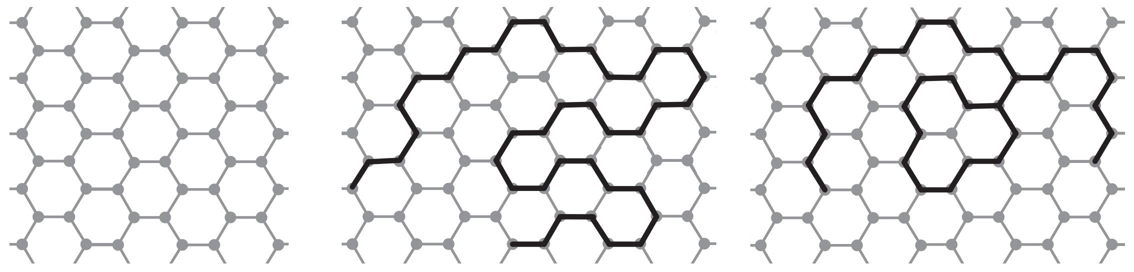


Figure 42. Self-Avoiding Walks in the Hexagonal Lattice. (a) The hexagonal lattice. (b) A self-avoiding walk in the hexagonal lattice. (c) A walk that is not self-avoiding.

suffices to use the trivial bound $Z \geq \lambda^{-p_{min}}$ for the former; to bound the latter, we turn to lattice duality and self-avoiding walks (for a more thorough treatment of self-avoiding walks, see, e.g., [18]).

Definition 8.3.1. A *self-avoiding walk* in a graph is a walk that never visits the same vertex twice.

Self-avoiding walks are most commonly studied for graphs that are planar lattices, and we will focus on self-avoiding walks in the hexagonal lattice, also called the honeycomb lattice (Figure 42a). Examples of self-avoiding walks and non-self-avoiding walks in this lattice are shown in Figures 42b and 42c, respectively. The hexagonal lattice is of interest because it is dual to the triangular lattice G_Δ that particles occupy in the amoebot model. That is, by creating a new vertex in every face of the triangular lattice and connecting two of these new vertices if their corresponding triangular faces have a common edge, we obtain the hexagonal lattice; see Figure 43a.

The number of self-avoiding walks of a certain length starting from a fixed vertex has been extensively studied for many planar lattices. This number is believed to grow exponentially with the length of the walk, and the base of this exponent is known as the *connective constant* of the lattice. More concretely, if N_l is the number of

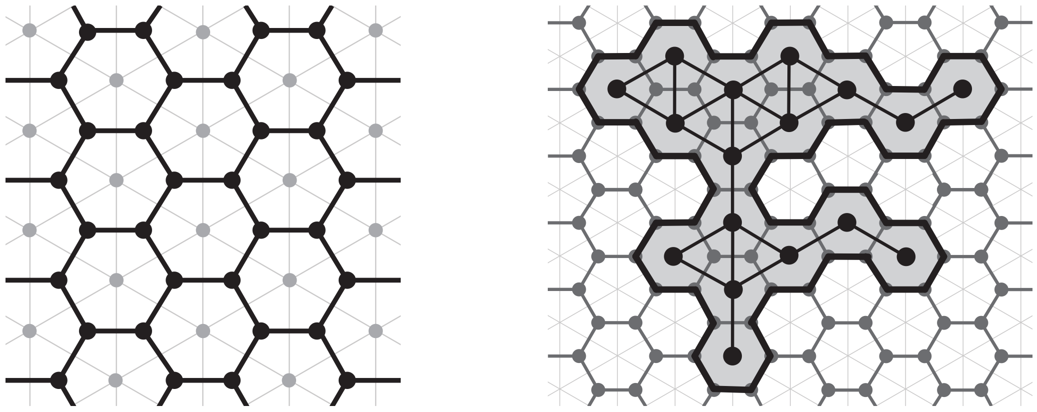


Figure 43. Self-Avoiding Walks and Particle System Configurations. (a) The duality between the triangular lattice G_Δ and the hexagonal lattice. (b) An example of a particle system configuration, its corresponding polygon in the hexagonal lattice (shaded), and the boundary of this region which is a self-avoiding polygon in the hexagonal lattice (bold).

self-avoiding walks of length l in some planar lattice L, then the connective constant of lattice L is defined as $\mu_L = \lim_{l\to\infty}(N_l)^{1/l}$. For example, the connective constant of the square lattice is $2.625622 \leq \mu_{sq} \leq 2.679193$, but an exact value has not been rigorously proved [115, 161]. The only lattice for which the connective constant is exactly known is our lattice of interest, the hexagonal lattice.

Theorem 8.3.2 (Duminil-Copin and Smirnov [75]). *The connective constant of the hexagonal lattice is $\mu_{hex} = \sqrt{2+\sqrt{2}}$.*

This theorem implies that the number of self-avoiding walks of length l in the hexagonal lattice is $f(l) \cdot \mu_{hex}^l$, for some subexponential function f (a function is *subexponential* if $\lim_{l\to\infty} f(l)^{1/l} = 1$).

To bound the number of connected, hole-free particle system configurations with some fixed perimeter, we turn from self-avoiding walks to the closely related notion of *self-avoiding polygons*, where a self-avoiding polygon is a self-avoiding walk that starts and ends at the same vertex (Figure 43b). The number of self-avoiding walks of length l is an upper bound on the number of self-avoiding polygons of perimeter l.

Lemma 8.3.3. *The number of connected, hole-free particle system configurations with n particles and perimeter k is at most $f(k) \cdot \left(2+\sqrt{2}\right)^k$ for some subexponential function f.*

Proof. Consider the dual to the triangular lattice G_Δ, the hexagonal lattice G_{hex} (Figure 43a). For any connected, hole-free particle system configuration σ with n particles, consider the union A_σ of all the faces of G_{hex} corresponding to vertices of G_Δ that are occupied in σ. Whenever two particles are adjacent in G_Δ, their corresponding faces in G_{hex} share an edge. This union A_σ is a simply connected polygon because σ is connected and has no holes; moreover, the boundary of A_σ forms a self-avoiding polygon in G_{hex} (bold in Figure 43b).

We first show that the number of connected, hole-free particle system configurations in G_Δ with perimeter k is at most the number of self-avoiding polygons in G_{hex} with perimeter $2k+6$. Let σ be a connected, hole-free configuration with perimeter k. A particle P is on the (unique external) boundary of σ if and only if its corresponding hexagon H_P in G_{hex} shares an edge with the perimeter of A_σ. That is, if a particle P appears once on the boundary of σ with exterior angle $\theta_P \in \{120°, 180°, 240°, 300°, 360°\}$, then H_P has $(\theta_P/60°) - 1$ of its edges contained in the perimeter of A_σ. More generally, if a particle P appears $m_P \geq 1$ times on the boundary of σ with exterior angles summing to θ_P, then H_P has $(\theta_P/60°) - m_P$ of its edges contained in the perimeter of A_σ. Thus, we conclude the number of edges on the perimeter of A_σ is:

$$p(A_\sigma) = \sum_{P \in p(\sigma)} \left(\frac{\theta_P}{60°} - m_P\right) = \frac{\sum_{P \in p(\sigma)} \theta_P}{60°} - k = \frac{180°k + 360°}{60°} - k = 2k + 6.$$

As noted before, the number of self-avoiding polygons in a lattice with perimeter l is at most the number of self-avoiding walks in that lattice with length l. So the number of self-avoiding polygons in G_{hex} with perimeter $2k+6$ is at most the number of self-avoiding walks in G_{hex} of length $2k+6$. By Theorem 8.3.2, the number of self-avoiding walks in G_{hex} of length $2k+6$ is $f_1(2k+6)\cdot\mu_{hex}^{2k+6}$, for some subexponential function f_1. Let $f(k) = f_1(2k+6)\cdot\mu_{hex}^{6}$; note that f is also subexponential. Then, all together, the number of connected, hole-free particle system configurations with perimeter k is at most $f(k)\cdot\mu_{hex}^{2k} = f(k)\cdot\left(2+\sqrt{2}\right)^k$, as desired. □

We can restate Lemma 8.3.3 in a slightly different way to make it easier to use in our later proofs.

Lemma 8.3.4. *For any $\nu > 2+\sqrt{2}$, there is an integer $n_1(\nu)$ such that for all $n \geq n_1(\nu)$, the number of connected, hole-free particle system configurations with n particles and perimeter k is at most ν^k.*

Proof. From Lemma 8.3.3 we know that the number of connected, hole-free configurations with n particles and perimeter k is at most at most $f(k)\cdot\left(2+\sqrt{2}\right)^k$, for some subexponential function f. Because $\nu > 2+\sqrt{2}$ and f is subexponential, it follows that:

$$\lim_{k\to\infty}\frac{f(k)\cdot(2+\sqrt{2})^k}{\nu^k} = 0.$$

Let $k_1(\nu)$ be such that for all $k > k_1(\nu)$, $f(k)\cdot\left(2+\sqrt{2}\right)^k \leq \nu^k$; $k_1(\nu)$ must exist because the above limit is less than one. Let $n_1(\nu) = k_1(\nu)^2$. For any $n \geq n_1(\nu)$, all connected configurations with n particles have perimeter at least $k_1(\nu)$ by Lemma 8.1.1. We conclude that for any $n \geq n_1(\nu)$ and for any k between $p_{min}(n)$ and $p_{max}(n)$, $f(k)\cdot\left(2+\sqrt{2}\right)^k \leq \nu^k$ and thus the number of connected, hole-free configurations with perimeter k is at most ν^k, as claimed. If k is not within these bounds, there are no configurations with n particles and perimeter k, so the lemma is trivially true. $\square$

We note that, in general, the closer ν is to $2+\sqrt{2}$ the larger $n_1(\nu)$ has to be.

8.3.2 Proof of Compression

To simplify notation, we define the *weight* of a configuration σ to be $w(\sigma) = \pi(\sigma)\cdot Z = \lambda^{-p(\sigma)}$. For a set $S \subseteq \Omega$, we define $w(S) = \sum_{\sigma \in S} w(\sigma)$ as the sum of the weights of all configurations in S. We now prove our main result.

Theorem 8.3.5. *For any $\alpha > 1$, let $\lambda^* = (2+\sqrt{2})^{\frac{\alpha}{\alpha-1}}$. There exists $n^* \geq 0$ and $\zeta < 1$ such that for all $\lambda > \lambda^*$ and $n > n^*$, the probability that a random sample σ drawn according to the stationary distribution π of $\mathcal{M}_C$ is not α-compressed is exponentially small in $\sqrt{n} = \Theta(p_{min})$:*

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

Proof. Recall that S_α is the set of connected, hole-free configurations with perimeter at least $\alpha \cdot p_{min}$. We wish to show that $\pi(S_\alpha)$ is smaller than some function that is exponentially small in $\sqrt{n}$.

We first consider the partition function Z of π; recall $Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)}$. If σ_{min} is a configuration of n particles achieving the minimum possible perimeter p_{min}, then $w(\sigma_{min}) = \lambda^{-p_{min}}$ is a lower bound on Z. It follows that:

$$\pi(S_\alpha) = \frac{w(S_\alpha)}{Z} < \frac{w(S_\alpha)}{w(\sigma_{min})}.$$

The remainder of this proof will be spent finding an upper bound on $w(S_\alpha)/w(\sigma_{min})$ that is exponentially small in $\sqrt{n}$. To begin, we stratify S_α into sets of configurations that have the same perimeter; recall every configuration has an integer perimeter because of lattice constraints. Let A_k be the set of all configurations with perimeter k; then $S_\alpha = \bigcup_{k=\lceil \alpha \cdot p_{min} \rceil}^{p_{max}} A_k$. Noting that $p_{max} = 2n-2$, we can then write:

$$\frac{w(S_\alpha)}{w(\sigma_{min})} = \frac{\sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} w(A_k)}{\lambda^{-p_{min}}}.$$

Since all configurations in A_k have the same perimeter k, they also have the same weight λ^{-k}; thus, $w(A_k) = |A_k|\lambda^{-k}$. Choose ν such that $\lambda^* < \nu^{\frac{\alpha}{\alpha-1}} < \lambda$, implying $2+\sqrt{2} < \nu < \lambda^{\frac{\alpha-1}{\alpha}}$; since $\lambda > \lambda^*$, such a ν must exist. By Lemma 8.3.4, provided n is large enough (i.e., larger than the value $n_1(\nu)$), we have $|A_k| \le \nu^k$. So, we have:

$$\frac{w(S_\alpha)}{w(\sigma_{min})} = \frac{\sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} |A_k|\lambda^{-k}}{\lambda^{-p_{min}}} \le \frac{\sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} \nu^k\lambda^{-k}}{\lambda^{-p_{min}}} = \sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} \nu^k \lambda^{-k}\lambda^{p_{min}}.$$

Because $k \ge \alpha \cdot p_{min}$, it follows that $k/\alpha \ge p_{min}$. As $\lambda > \lambda^* > 2+\sqrt{2} > 1$,

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \le \sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} \nu^k \lambda^{-k}\lambda^{k/\alpha} = \sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} \left(\frac{\nu}{\lambda^{\frac{\alpha-1}{\alpha}}}\right)^k.$$

We chose ν such that $\nu < \lambda^{\frac{\alpha-1}{\alpha}}$, so $(\nu/\lambda^{\frac{\alpha-1}{\alpha}}) < 1$. Because $k \ge \alpha \cdot p_{min} > \alpha\sqrt{n}$ (by Lemma 8.1.1), we have:

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \le \sum_{k=\lceil \alpha \cdot p_{min} \rceil}^{2n-2} \left(\frac{\nu}{\lambda^{\frac{\alpha-1}{\alpha}}}\right)^{\alpha\sqrt{n}} \le (2n-2)\left(\frac{\nu}{\lambda^{\frac{\alpha-1}{\alpha}}}\right)^{\alpha\sqrt{n}}.$$

Since $(\nu/\lambda^{\frac{\alpha-1}{\alpha}}) < 1$, we can find a constant $\zeta < 1$ such for all sufficiently large n, say $n \ge n_2$, it holds that:

$$\frac{w(S_\alpha)}{w(\sigma_{min})} \le (2n-2)\left(\frac{\nu}{\lambda^{\frac{\alpha-1}{\alpha}}}\right)^{\alpha\sqrt{n}} < \zeta^{\sqrt{n}}.$$

Setting $n^* = \max\{n_1(\nu), n_2\}$ completes the proof of the theorem. □

Although Theorem 8.3.5 is proved only in the case where the number of particles is sufficiently large, we expect and observe it to hold for much smaller n. We note that the closer the value $\nu^{\frac{\alpha}{\alpha-1}}$ used in the above proof is to λ^* the larger $n_1(\nu)$ is, and the closer ν is to λ, the larger n_2 is. In particular, when λ is close to λ^* then n^* must be large, while for $\lambda \gg \lambda^*$, smaller values of n^* suffice for the proof. Computing an exact value for n^* is difficult because the value $n_1(\nu)$ from Lemma 8.3.4 is not known explicitly; see Section 4 of [75] and the references therein.

While the above result shows that $\mathcal{M}_C$ (and, by extension, the local, distributed algorithm $\mathcal{A}_C$) accomplishes α-compression for any $\alpha > 1$, smaller values of α require larger values of λ. In practice, when λ is large, $\mathcal{M}_C$ takes a long time to reach any compressed configuration. Because of this, what happens when λ is small is also of interest. We now show that provided $\lambda > 2 + \sqrt{2}$, there is some constant α such that α-compression occurs. Of course, there is again a tradeoff: the smaller λ is, the larger α must be.

Corollary 8.3.6. *For any $\lambda > 2+\sqrt{2}$, for any constant $\alpha > \log_{2+\sqrt{2}}\lambda/(\log_{2+\sqrt{2}}\lambda - 1)$ there exists $n^* \geq 0$ and $\zeta < 1$ such that for all $n \geq n^*$, a random sample σ drawn according to the stationary distribution π of $\mathcal{M}_C$ satisfies*

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

Proof. If $\alpha > \frac{\log_{2+\sqrt{2}}\lambda}{\log_{2+\sqrt{2}}\lambda - 1}$, then solving for λ gives $\lambda > (2+\sqrt{2})^{\frac{\alpha}{\alpha-1}}$. Theorem 8.3.5 then gives the desired result. $\square$

8.4 Achieving Expansion

Now that we have proved Markov chain $\mathcal{M}_C$ (and distributed algorithm $\mathcal{A}_C$) yields compression whenever $\lambda > 2 + \sqrt{2}$, it is natural to ask about the behavior of $\mathcal{M}_C$ when $\lambda \leq 2 + \sqrt{2}$. As $\lambda > 1$ corresponds to particles favoring having more neighbors, one might conjecture that compression occurs whenever $\lambda > 1$. We show, counterintuitively, that this is not the case: for all $\lambda < 2.17$, compression does not occur. For example, consider the simulation depicted in Figure 44: even after 20 million iterations of $\mathcal{M}_C$ with bias $\lambda = 2$, the system has not compressed. This stands in stark contrast to the simulations depicted in Figure 36, which achieved good compression after only 5 million iterations of $\mathcal{M}_C$ using $\lambda = 4$.

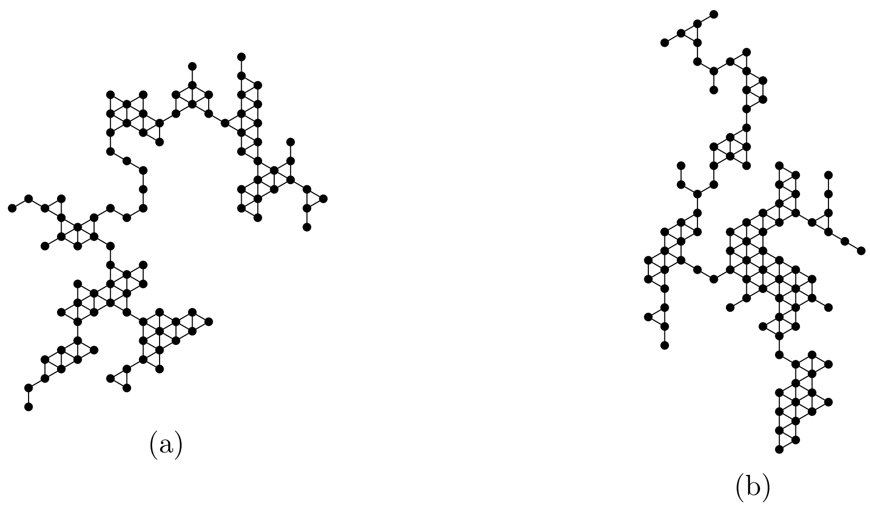


Figure 44. Simulation of Markov Chain $\mathcal{M}_C$ with $\lambda = 2$. A system of 100 particles initially in a line after (a) 10 million and (b) 20 million iterations of $\mathcal{M}_C$ with bias $\lambda = 2$.

Analogous to our definition of α-compression, we say a configuration σ is β-*expanded* for some $0 < \beta < 1$ if $p(\sigma) > \beta \cdot p_{max}$. For a configuration of n particles, $p_{max} = 2n - 2 = \Theta(n)$ and $p_{min} = \Theta(\sqrt{n})$, so β-expansion and α-compression for any constants β and α are mutually exclusive for sufficiently large n. We prove in this section that, for all $0 < \lambda < 2.17$ and provided n is large enough, there is a constant β such that a configuration chosen at random according to the stationary distribution of $\mathcal{M}_C$ is β-expanded with probability at least $1 - \zeta^{\sqrt{n}}$ for some constant

ζ. As mentioned above, this is notable because it implies that $\lambda > 1$ (i.e., favoring more neighbors) is not sufficient to guarantee compression as one might first guess.

We begin with some preliminaries about counting the number of particle system configurations with a certain perimeter, which will give us a lower bound on the partition function Z. We then use this bound to show that for all $\lambda < \sqrt{2}$, expansion occurs. By revisiting and improving this lower bound on Z, we can improve this result and show expansion occurs for all $\lambda < 2.17$.

8.4.1 A Non-trivial Lower Bound on the Partition Function

Let S^β be the set of all connected, hole-free particle system configurations with perimeter at most $\beta \cdot p_{max}$ for some constant $0 < \beta < 1$. Analogous to the approach for proving compression, we want to show $\pi(S^\beta) = \sum_{\sigma \in S^\beta} \lambda^{-p(\sigma)}/Z$, the stationary probability of being in a configuration with small perimeter, is exponentially small in $\sqrt{n}$. Recall that Corollary 8.2.14 defined the partition function as $Z = \sum_{\sigma\in\Omega^*} \lambda^{-p(\sigma)}$, the summed weight of all connected, hole-free configurations. The critical component of this result is an improved lower bound on Z; the trivial bound of $Z \geq \lambda^{-p_{min}}$ used for compression does not suffice for expansion. We give our first non-trivial lower bound on Z in Lemma 8.4.1, and this result is valid for all $\lambda > 0$. Later, we will obtain an improved lower bound on Z that is valid for all $\lambda \geq 1$.

Obtaining these lower bounds on Z for expansion requires a lower bound on the number of configurations with n particles and a given perimeter; this is the opposite of what we did for compression, where we upper bounded this quantity. To begin, we recall $p_{max} = 2n - 2$ and note:

$$Z = \sum_{\sigma\in\Omega^*} \lambda^{-p(\sigma)} \geq \sum_{\sigma\in\Omega^* \,:\, p(\sigma)=p_{max}} \lambda^{-p(\sigma)} = c_{2n-2}\lambda^{-(2n-2)},$$

where c_{2n-2} is the number of configurations with n particles and perimeter exactly $2n-2$. Note if a configuration σ with n particles has perimeter $2n-2$, then by Lemmas 8.1.3 and 8.1.4 it must be that σ has exactly $n-1$ edges and no triangles; that is, σ is an induced tree in G_Δ. We present a method for enumerating a subset of these trees, giving a lower bound on c_{2n-2}.

Lemma 8.4.1. *For any $\lambda > 0$, $Z \geq (\sqrt{2}/\lambda)^{p_{max}}$.*

Proof. We enumerate n-vertex paths in G_Δ where every step is either down-right or up-right; this is a subset of the trees contributing to c_{2n-2}. Starting from the first particle, there are 2^{n-1} ways to place rest of the particles to form such a path, where each one is either up-right or down-right from the previous one. This means there are at least 2^{n-1} such paths, giving:

$$c_{2n-2} \geq 2^{n-1} = \sqrt{2}^{2n-2} = \sqrt{2}^{p_{max}}.$$

From this, it follows that:

$$Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)} \geq \sum_{\sigma \in \Omega^* \ : \ p(\sigma) = p_{max}} \lambda^{-p_{max}} \geq \sqrt{2}^{p_{max}} \lambda^{-p_{max}} = (\sqrt{2}/\lambda)^{p_{max}},$$

the desired result. □

As the next result will show, Lemma 8.4.1 directly implies the particle system does not compress, even in the limit, for any $\lambda < \sqrt{2}$. This bound could be improved significantly with a better lower bound for c_{2n-2}, but this will be eclipsed by the lower bound for Z when $\lambda \geq 1$ given in Section 8.4.3.

8.4.2 Proof of Expansion

We now show, using Lemma 8.4.1, that for any value of $\beta \in (0,1)$ it is possible to achieve β-expansion by simply running $\mathcal{M}_C$ with input parameter λ sufficiently small. The closer β is to 1, the closer λ must be to 0 in order to achieve β-expansion.

Theorem 8.4.2. *For any* $0 < \beta < 1$*, let* $\lambda^* = \min\left(\sqrt{2}, \sqrt{2}^{\frac{1}{1-\beta}}(2+\sqrt{2})^{\frac{-\beta}{1-\beta}}\right)$. *There exists* $n^* \geq 0$ *and* $\zeta < 1$ *such that for all* $\lambda < \lambda^*$ *and* $n \geq n^*$*, the probability that a random sample* σ *drawn according to the stationary distribution* π *of* $\mathcal{M}_C$ *is not* β*-expanded is exponentially small in* $\sqrt{n}$*:*

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

Proof. Because $\lambda < \lambda^*$, we know $\lambda < \sqrt{2}^{\frac{1}{1-\beta}}(2+\sqrt{2})^{\frac{-\beta}{1-\beta}}$. Rearranging terms in this expression, we obtain $\lambda^{(\beta-1)/\beta}2^{1/2\beta} > 2+\sqrt{2}$. Let ν be any value satisfying $\lambda^{(\beta-1)/\beta}2^{1/2\beta} > \nu > 2+\sqrt{2}$. We will later use the fact that for any such choice of ν,

$$\nu\lambda^{\frac{1-\beta}{\beta}}2^{-\frac{1}{2\beta}} < 1.$$

Let S^β be the set of configurations of perimeter at most $\beta \cdot p_{max}$. We wish to show that $\pi(S^\beta)$ is smaller than some function that is exponentially small in $\sqrt{n}$. Using Lemma 8.4.1 to upper bound the partition function Z of the stationary distribution π, we have:

$$\pi\left(S^\beta\right) = \frac{w\left(S^\beta\right)}{Z} \leq \frac{w\left(S^\beta\right)}{\left(\sqrt{2}/\lambda\right)^{p_{max}}} = w\left(S^\beta\right)\left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}}.$$

The remainder of this proof will be spent finding an upper bound on the right hand side of the above equation that is exponentially small in $\sqrt{n}$. To begin, we stratify S^β into sets of configurations that have the same perimeter. Let B_k be the set of all configurations with perimeter k; then $S^\beta = \bigcup_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} B_k$. We can then write:

$$w(S^\beta) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}} = \sum_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} w(B_k) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}}.$$

The weight of each element in the set B_k is the same, λ^{-k}. By Lemma 8.3.4 and our careful choice of ν, above, the number of configurations in set B_k is at most ν^k provided k is sufficiently large. So,

$$w(S^\beta) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}} \leq \sum_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} \nu^k \lambda^{-k} \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}}.$$

Because $k \leq \beta \cdot p_{max}$, we have $p_{max} \geq k/\beta$. As $\lambda < \lambda^* \leq \sqrt{2}$, we have:

$$w(S^\beta) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}} \leq \sum_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} \nu^k \lambda^{-k} \left(\frac{\lambda}{\sqrt{2}}\right)^{\frac{k}{\beta}} = \sum_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} \left(\frac{\nu \lambda^{\frac{1-\beta}{\beta}}}{\sqrt{2}^{\frac{1}{\beta}}}\right)^k.$$

Because of our choice of ν, the rightmost term in parentheses is less than one. By applying the inequalities $p_{max} = 2n-2 \geq k \geq p_{min} > \sqrt{n}$ (by Lemma 8.1.1),

$$w(S^\beta) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}} \leq \sum_{k=p_{min}}^{\lfloor \beta \cdot p_{max} \rfloor} \left(\frac{\nu \lambda^{\frac{1-\beta}{\beta}}}{\sqrt{2}^{\frac{1}{\beta}}}\right)^{\sqrt{n}} \leq (2n-2) \left(\frac{\nu \lambda^{\frac{1-\beta}{\beta}}}{\sqrt{2}^{\frac{1}{\beta}}}\right)^{\sqrt{n}}.$$

Again using the fact that the rightmost term in parentheses is less than one, we can find a constant $\zeta < 1$ and an n^* such that for all $n \geq n^*$,

$$\pi(S^\beta) \leq w(S^\beta) \left(\frac{\lambda}{\sqrt{2}}\right)^{p_{max}} \leq (2n-2) \left(\frac{\nu \lambda^{\frac{1-\beta}{\beta}}}{\sqrt{2}^{\frac{1}{\beta}}}\right)^{\sqrt{n}} < \zeta^{\sqrt{n}},$$

proving the theorem. □

While the above result shows that $\mathcal{M}_C$ accomplishes β-expansion for any $\beta < 1$, larger values of β require smaller values of λ. However, larger values of λ are still of interest as we wish to characterize how the behavior of $\mathcal{M}_C$ and $\mathcal{A}_C$ depends on λ. We now show that provided $\lambda < \sqrt{2}$, there is some constant β such that β-expansion occurs. Of course, there is again a tradeoff: the larger λ is, the smaller β is.

Corollary 8.4.3. *For all $0 < \lambda < \sqrt{2}$, for any constant $\beta < \frac{\log\sqrt{2} - \log\lambda}{\log(2+\sqrt{2}) - \log\lambda}$, there exists $n^* \geq 0$ and $\zeta < 1$ such that for all $n \geq n^*$, a random sample σ drawn according to the stationary distribution π of $\mathcal{M}_C$ satisfies*

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

Proof. Theorem 8.4.2 applies whenever $\lambda < \sqrt{2}^{\frac{1}{1-\beta}}(2+\sqrt{2})^{\frac{-\beta}{1-\beta}}$. Solving for β, we see the theorem applies whenever $\beta < \frac{\log\sqrt{2} - \log\lambda}{\log(2+\sqrt{2}) - \log\lambda}$, as desired. $\square$

This proves the counterintuitive result that $\lambda > 1$ is not sufficient to guarantee compression. While $\lambda > 1$ guarantees that configurations with smaller perimeter have higher weight at stationarity, our work in this section shows that there are so many configurations with large perimeter that, for $\lambda < \sqrt{2}$, these large perimeter configurations dominate the stationary distribution. Raising λ above $2 + \sqrt{2}$, we observe an energy/entropy tradeoff. In this regime, the high energy (small perimeter) configurations dominate the state space as opposed to those with high entropy (large perimeter), yielding compression.

8.4.3 An Improved Lower Bound on the Partition Function

We can improve the bound of $\lambda < \sqrt{2}$ appearing in Corollary 8.4.3 by finding a better lower bound on the partition function Z. When we know $\lambda \geq 1$, the improved bounds in Lemma 8.4.6 below can be used to show β-expansion occurs for an even greater range of values for λ, $\lambda < 2.17$. Again, larger values of λ necessitate smaller, but still constant, values of β.

To get an improved bound on Z, the key observation is that when $\lambda \geq 1$, any

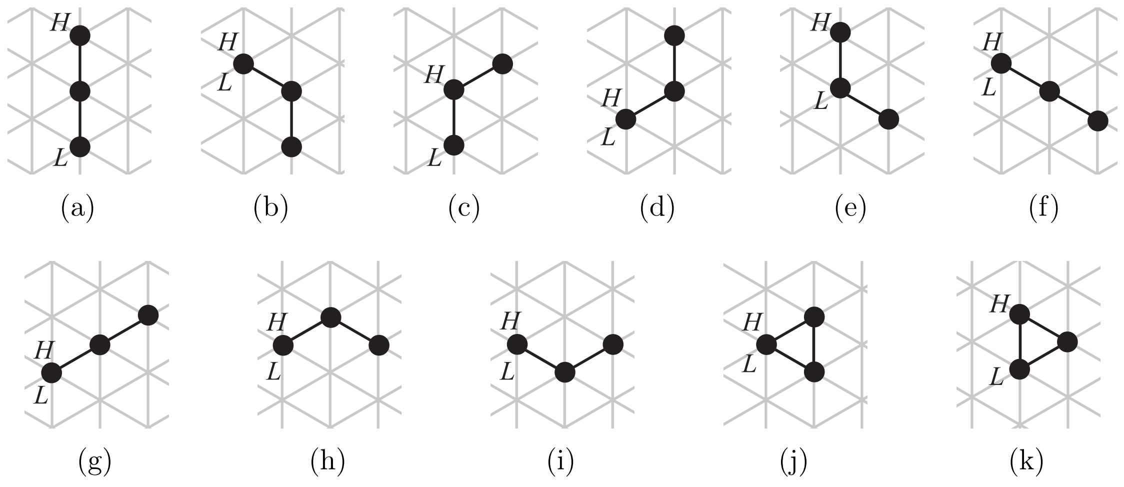


Figure 45. Three-Particle System Configurations. All 11 connected, hole-free configurations of three particles. In each, the highest leftmost particle is labeled H, and the lowest leftmost particle is labeled L; when there is only one leftmost particle $H = L$.

value $k < 2n - 2$ satisfies $\lambda^{-k} \geq \lambda^{-(2n-2)}$. Thus, as $p_{max} = 2n - 2$, it follows that

$$Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)} \geq \sum_{\sigma \in \Omega^*} \lambda^{-(2n-2)} = |\Omega^*| \cdot \lambda^{-(2n-2)},$$

where the sums are over all connected, hole-free particle system configurations with n particles. Thus, it suffices to find a lower bound on the total number of connected, hole-free configurations with n particles and any perimeter, instead of only counting the number of configurations with maximum perimeter as we did in Section 8.4.1. Leveraging this observation will yield a better lower bound on Z than in the previous case where λ was unrestricted.

Lemma 8.4.4. *For $\lambda \geq 1$, $Z \geq 0.12 \cdot (1.67/\lambda)^{p_{max}}$.*

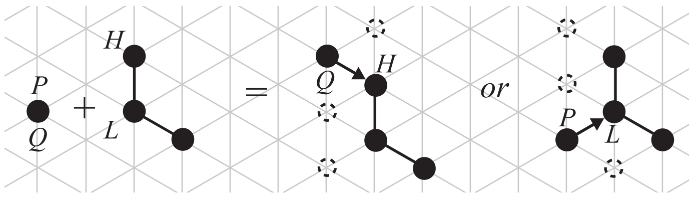
Proof. We first give a lower bound on the number of connected, hole-free configurations with n particles by iteratively enumerating a subset of them. Note there are 11 connected, hole-free configurations with exactly 3 particles; all 11 are shown in Figure 45.

Given some connected, hole-free configuration σ with $1 + 3j$ particles, for $j \geq 0$, we show how to enumerate 22 distinct hole-free configurations of $4 + 3j$ particles, each consisting of three particles added to the right of σ. Let P be the highest rightmost particle of σ and let Q be the lowest rightmost particle of σ; possibly $P = Q$. Choose any of the 11 hole-free configurations with 3 particles, and let L be its lowest leftmost particle and H be its highest leftmost particle as in Figure 45; possibly $H = L$. Attach this configuration to σ either by placing H below and right of Q or by placing L above and right of P; see Figure 46 for two such examples. Note even if $Q = P$ and $H = L$, this still results in two distinct attachments. In the first case, all locations directly below Q and all locations directly above H are unoccupied; this ensures the only adjacency between σ and the newly added three particles is between Q and H, meaning no holes have been created. Similarly in the second case, all locations above P or below L are unoccupied, again ensuring no holes form.

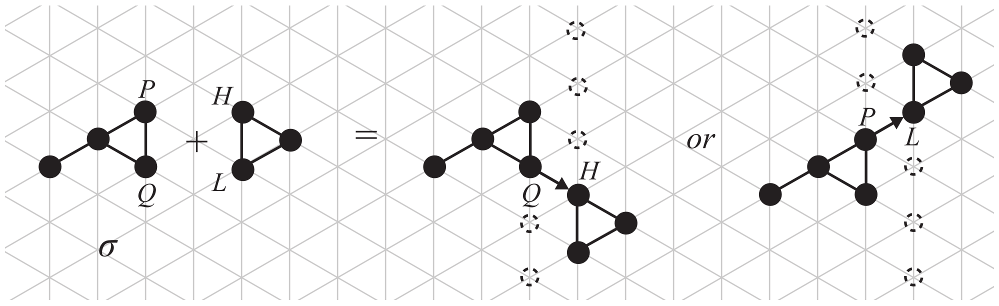
Using this process and beginning with a single particle (as in Figure 46a), we can enumerate 22^j distinct configurations with $1 + 3j$ particles for all $j \geq 0$. This does not enumerate all configurations on $1 + 3j$ particles: for example, there are 42 configurations on 4 particles and this process only enumerates 22 of them. However, this process iterates nicely and produces reasonable lower bounds as the number of particles gets large.

To get a lower bound on the number of configurations of n particles when $n \not\equiv 1 \bmod 3$), we can simply enumerate all configurations on $1 + 3j \leq n$ particles for $j = \left\lfloor \frac{n-1}{3} \right\rfloor$, and add one or two particles to each in some deterministic way. We conclude that for any n, the number of connected, hole-free configurations of n particles is at least:

$$22^{\left\lfloor \frac{n-1}{3} \right\rfloor} \geq 22^{\frac{n-1}{3}} \cdot 22^{-2/3} = 22^{-2/3}(22^{1/6})^{2n-2} > 0.12 \cdot 1.67^{2n-2}.$$



(a)



(b)

Figure 46. Enumerating Tree Configurations. The iterative process of Lemma 8.4.4. (a) One of the 11 connected, hole-free configurations of three particles, and the two ways it can attach to the single particle with which the iterative process begins. (b) Another of the 11 connected hole-free configurations of three particles, and the two ways it can attach to a configuration with four particles.

Using this bound, it follows that:

$$Z = \sum_{\sigma \in \Omega^*} \lambda^{-p(\sigma)} \geq \sum_{\sigma \in \Omega^*} \lambda^{-(2n-2)} = \frac{|\Omega^*|}{\lambda^{2n-2}} > \frac{0.12 \cdot 1.67^{2n-2}}{\lambda^{2n-2}} = 0.12 \cdot \left(\frac{1.67}{\lambda}\right)^{p_{max}},$$

proving the lemma. □

This bound can be improved even further by considering configurations of 50 particles instead of configurations of three particles. A result by Jensen [114] will be essential. In that paper, the author presents a parallel algorithm efficient enough to count the number of benzenoid hydrocarbons containing h hexagonal cells up to $h = 50$. A benzenoid hydrocarbon containing h hexagonal cells is exactly equivalent to a connected, hole-free particle system configuration with h particles, implying the following.

Lemma 8.4.5 (Jensen [114])**.** *The number of connected, hole-free particle system configurations with* 50 *particles is:*

$$N_{50} = 2{,}430{,}068{,}453{,}031{,}180{,}290{,}203{,}185{,}942{,}420{,}933.$$

Lemma 8.4.6. *For $\lambda \geq 1$, $Z \geq 0.13 \cdot (2.17/\lambda)^{p_{max}}$.*

Proof. We use the same approach as in Lemma 8.4.4, noting that $2.17 \approx (2N_{50})^{1/100}$. To get a lower bound on the number of configurations with n particles, first write n as $n = 1+50i+j$, where $i, j \in \mathbb{Z}_{\geq 0}$ and $j < 50$; subject to these requirements, i and j are unique. Iteratively construct one particle configuration σ with n particles by beginning with a single particle and repeatedly attaching one of the N_{50} configurations with 50 particles to the right as in the proof of Lemma 8.4.4: place its highest leftmost particle H below and right of the existing configuration's lowest rightmost particle Q, or place its lowest leftmost particle L above and right of the existing configuration's highest rightmost particle P. This process, applied i times, yields a connected, hole-free configuration of $1+50i = n-j$ particles. There are then $2N_j$ ways, following the same procedure, to attach the remaining j particles to form a configuration of n particles. In this way, we can enumerate $(2N_{50})^i \cdot 2N_j$ unique connected, hole-free configurations of n particles. It follows that the number of connected, hole-free configurations of n particles is at least:

$$(2N_{50})^i \cdot 2N_j = (2N_{50})^{\frac{n-1-j}{50}} \cdot 2N_j = (2N_{50})^{\frac{n-1}{50}} \cdot (2N_{50})^{-\frac{j}{50}} \cdot 2N_j.$$

Calculations show that for all $0 \leq j < 50$, we have $(2N_{50})^{-j/50} \cdot 2N_j \geq 0.13$. It follows that the number of connected, hole-free configurations of n particles is at least:

$$0.13 \cdot \left((2N_{50})^{1/100}\right)^{2n-2} = 0.13 \cdot \left((2N_{50})^{1/100}\right)^{p_{max}}.$$

Noting that $(2N_{50})^{1/100} > 2.17$, it follows that:

$$Z \geq \sum_{\sigma \in \Omega^*} \lambda^{-(2n-2)} = |\Omega^*| \cdot \lambda^{-(2n-2)} \geq 0.13 \cdot (2.17)^{p_{max}} \lambda^{-p_{max}} = 0.13 \cdot (2.17/\lambda)^{p_{max}},$$

the desired result. □

As we will see next, this will directly imply that the particle system will not exhibit compression for any $\lambda < 2.17$. We expect this bound will improve given accurate counts of the number of connected, hole-free configurations for even larger numbers of particles. Computationally this seems infeasible, and a careful analysis of the work done in [114] suggests the best bound achievable by this method would be expansion for all $\lambda < 2.27$, only a mild improvement and still far from the known lower bound for compression, $\lambda > 2 + \sqrt{2}$. Indeed, recent work using enumeration methods has only improved the $\lambda < 2.17$ bound to 2.21 [134].

8.4.4 Proof of Expansion for a Larger Range of Bias

We now show, using Lemma 8.4.6, that it is possible to achieve β-expansion (i.e., compression does not occur) using any value of λ up to 2.17.

Theorem 8.4.7. *For all $1 \leq \lambda < x := (2N_{50})^{1/100} \approx 2.17$, for any $\beta < \frac{\log x - \log \lambda}{\log(2+\sqrt{2}) - \log \lambda}$ there exists $n^* \geq 0$ and $\zeta < 1$ such that for all $n \geq n^*$, a random sample σ drawn according to the stationary distribution π of $\mathcal{M}_C$ satisfies*

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

Proof. First, note that the condition

$$\beta < \frac{\log x - \log \lambda}{\log\left(2+\sqrt{2}\right) - \log \lambda} = \frac{\log(x/\lambda)}{\log\left(\left(2+\sqrt{2}\right)/\lambda\right)}$$

can be equivalently expressed as $2+\sqrt{2} < \lambda^{(\beta-1)/\beta} x^{1/\beta}$. Pick ν to be between these two values, so that $2+\sqrt{2} < \nu < \lambda^{(\beta-1)/\beta} x^{1/\beta}$.

Let S^β be the set of configurations of perimeter at most $\beta \cdot p_{max}$. We wish to show that $\pi(S^\beta)$ is smaller than some function that is exponentially small in $\sqrt{n}$. Applying Lemma 8.4.6, which gives a lower bound on the partition function Z of stationary distribution π, we see that:

$$\pi\big(S^\beta\big) = \frac{w\big(S^\beta\big)}{Z} \le \frac{w\big(S^\beta\big)}{0.13\left(\frac{x}{\lambda}\right)^{p_{max}}} \le 8w\big(S^\beta\big)\left(\frac{\lambda}{x}\right)^{p_{max}}.$$

The remainder of this proof will be spent finding an upper bound on the right hand side of the above equation that is exponentially small in $\sqrt{n}$. To begin, we stratify S^β into sets of configurations that have the same perimeter. Let B_k be the set of all configurations with perimeter k; then $S^\beta = \bigcup_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor} B_k$. We can then write:

$$w\big(S^\beta\big)\left(\frac{\lambda}{x}\right)^{p_{max}} = \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor} w(B_k)\left(\frac{\lambda}{x}\right)^{p_{max}}.$$

The weight of each element in the set B_k is the same, λ^{-k}. By Lemma 8.3.4, the number of elements in set B_k is at most ν^k for k sufficiently large because $\nu > 2+\sqrt{2}$. So we have:

$$w\big(S^\beta\big)\left(\frac{\lambda}{x}\right)^{p_{max}} \le \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor} \nu^k\lambda^{-k}\left(\frac{\lambda}{x}\right)^{p_{max}}.$$

Because $k \le \beta\cdot p_{max}$, we have $p_{max} \ge k/\beta$. As $\lambda < x$, we have:

$$w\big(S^\beta\big)\left(\frac{\lambda}{x}\right)^{p_{max}} \le \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor} \nu^k\lambda^{-k}\left(\frac{\lambda}{x}\right)^{\frac{k}{\beta}} = \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor}\left(\frac{\nu\lambda^{\frac{1}{\beta}}}{\lambda x^{\frac{1}{\beta}}}\right)^k = \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor}\left(\frac{\nu}{\lambda^{\frac{\beta-1}{\beta}}x^{\frac{1}{\beta}}}\right)^k.$$

Because we picked ν so that $\nu < \lambda^{(\beta-1)/\beta}x^{1/\beta}$, the rightmost term in parentheses is less than one. By applying the inequalities $p_{max} = 2n-2 \ge k \ge p_{min} > \sqrt{n}$ (by Lemma 8.1.1), we see that

$$w\big(S^\beta\big)\left(\frac{\lambda}{x}\right)^{p_{max}} \le \sum_{k=p_{min}}^{\lfloor \beta\cdot p_{max}\rfloor}\left(\frac{\nu}{\lambda^{\frac{\beta-1}{\beta}}x^{\frac{1}{\beta}}}\right)^{\sqrt{n}} \le (2n-2)\left(\frac{\nu}{\lambda^{\frac{\beta-1}{\beta}}x^{\frac{1}{\beta}}}\right)^{\sqrt{n}}.$$

Again using the fact that the rightmost term in parentheses above is less than one, we can find a constant $\zeta < 1$ and an n^* such that for all $n \geq n^*$,

$$\pi(S^\beta) \leq 8w(S^\beta)\left(\frac{\lambda}{x}\right)^{p_{max}} \leq 8(2n-2)\left(\frac{\nu}{\lambda^{\frac{\beta-1}{\beta}}x^{\frac{1}{\beta}}}\right)^{\sqrt{n}} < \zeta^{\sqrt{n}},$$

proving the theorem. □

Combining Theorem 8.4.7 with Corollary 8.4.3 gives the following result.

Corollary 8.4.8. *For all* $0 < \lambda < (2N_{50})^{1/100} \approx 2.17$*, there exists a constant* $0 < \beta < 1$ *such that for sufficiently large* n *with all but exponentially small (in* $\sqrt{n}$*) probability a sample drawn according to stationary distribution* π *of* $\mathcal{M}_C$ *is* β*-expanded.*

Proof. If $0 < \lambda < 1$, then by Corollary 8.4.3, for any constant $\beta < \frac{\log\sqrt{2}-\log\lambda}{\log(2+\sqrt{2})-\log\lambda}$, for sufficiently large n with all but exponentially small probability β-expansion occurs at stationarity. Note that for $\lambda < 1$, we have $1 > \frac{\log\sqrt{2}-\log\lambda}{\log(2+\sqrt{2})-\log\lambda} > 0$, so there always exists such a positive constant β less than this bound for which β-expansion occurs.

If $1 \leq \lambda < x := (2N_{50})^{1/100} \approx 2.17$, then by Theorem 8.4.7, for any $\beta < \frac{\log x-\log\lambda}{\log(2+\sqrt{2})-\log\lambda}$, for sufficiently large n with all but exponentially small probability β-expansion occurs at stationarity. Because $1 > \frac{\log x-\log\lambda}{\log(2+\sqrt{2})-\log\lambda} > 0$, there always exists such a positive constant β less than this bound for which β-expansion occurs.

We conclude that for any $0 < \lambda < (2N_{50})^{1/100}$, there exists a constant $0 < \beta < 1$ such that for sufficiently large n with all but exponentially small probability a sample drawn according to stationary distribution π of $\mathcal{M}_C$ is β-expanded. □

Chapter 9

SHORTCUT BRIDGING

We next apply the stochastic approach to *shortcut bridging*, where an amoebot system must dynamically maintain bridge structures that balance a tradeoff between efficiency and cost [7, 8]. This behavior is inspired by the work of Reid et al. [168] who found that *Eciton* army ants continuously modify the shape and position of foraging bridges — constructed and maintained by their own bodies — across holes and uneven surfaces on the forest floor. These bridges appear to stabilize in a structural formation that balances the "benefit of increased foraging trail efficiency" with the "cost of removing workers from the foraging pool to form the structure" [168]. Shortcut bridging is an attractive goal for programmable matter, as many application domains involve surfaces with structural irregularities and dynamic topologies. For example, recent progress in the molecular programming domain has achieved a process of DNA nanotube nucleation, growth, and diffusion to maintain connections between DNA origami landmarks [150]. In the future, programmable matter could be employed to detect and span small cracks in infrastructure; dynamic bridging behavior would enable the system to remain connected and shift its position as cracks form and change.

9.1 Preliminaries

9.1.1 The Shortcut Bridging Problem

In addition to the amoebot model (Chapter 2) and the terminology for the stochastic approach (Section 8.1.2), we first introduce some terminology specific to shortcut bridging. Just as the uneven surfaces of the forest floor affect the foraging behavior of army ants, the collective behavior of particle systems should change when G_Δ is non-uniform. Here, we focus on system behaviors when the nodes of G_Δ are either *gap* (unsupported) or *land* (supported). A particle can tell whether the node(s) it occupies are gap or land nodes. We also make use of *objects* — i.e., static, occupied nodes that do not perform computation — to keep the particle system anchored to certain fixed sites. In order to analyze the strength of the bridges produced by our algorithm, we define the *weighted perimeter* $\overline{p}(\sigma, c)$ of a connected, hole-free configuration σ to be the summed weight of the edges on the boundary of σ, where edges between land locations have weight 1, edges between gap locations have weight $c > 1$, and edges with one endpoint on land and one endpoint in the gap have weight $(1 + c)/2$.

An instance of the *shortcut bridging problem* has the form $(L, O, \sigma_0, c, \alpha)$, where $L \subseteq V$ is the set of land locations, O is the set of (two) objects to bridge between, σ_0 is the initial configuration of the particle system, $c > 1$ is a fixed weight for edges between gap locations, and $\alpha > 1$ is a parameter capturing our error tolerance. An instance is *valid* if (*i*) the objects of O and particles of σ_0 all occupy locations in L, (*ii*) σ_0 connects the objects, and (*iii*) σ_0 is connected. A distributed algorithm *solves* a valid instance $(L, O, \sigma_0, c, \alpha)$ if, beginning from σ_0, it reaches and remains in a set of configurations Σ^* such that any $\sigma \in \Sigma^*$ has weighted perimeter $\overline{p}(\sigma, c)$ within an α-factor of its minimum possible value, with high probability.

In analogy to the apparatus used in [168] (Figure 48a), we are particularly interested

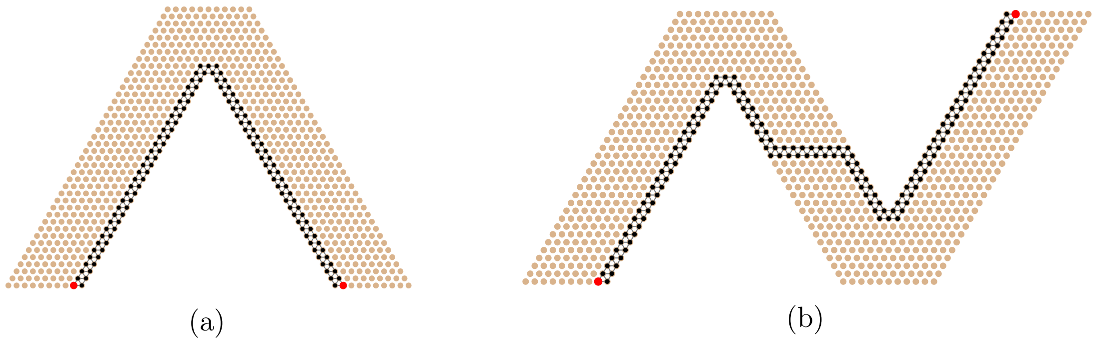


Figure 47. Example Initial Configurations for Shortcut Bridging. Initial configurations σ_0 of particles (black) connecting two objects O (red) on land masses L (brown and black) for two instances of the shortcut bridging problem for which we present simulation results (Section 9.5).

in instances where L forms a V-shape, O has two objects positioned at either base of L, and σ_0 lines the interior sides of L, as in Figure 47a. However, our algorithm is not limited to this setting; for example, we show simulation results for an N-shaped land mass (Figure 47b) in Section 9.5.

The weighted perimeter balances the trade-off observed in [168] between the competing objectives of establishing a short path between the fixed endpoints while not having too many particles in the gap. Although both metrics are amenable to our analysis, we focus on weighted perimeter instead of the number of particles in the gap for two reasons. First, the structure and thickness of bridges produced using weighted perimeter more closely resemble those of ant bridges, while using the number of particles in the gap results in consistently thin, jagged structures (compare Figures 48b vs. 48c). Second, only particles on the perimeter can move, and thus recognize the potential risk of being in the gap.

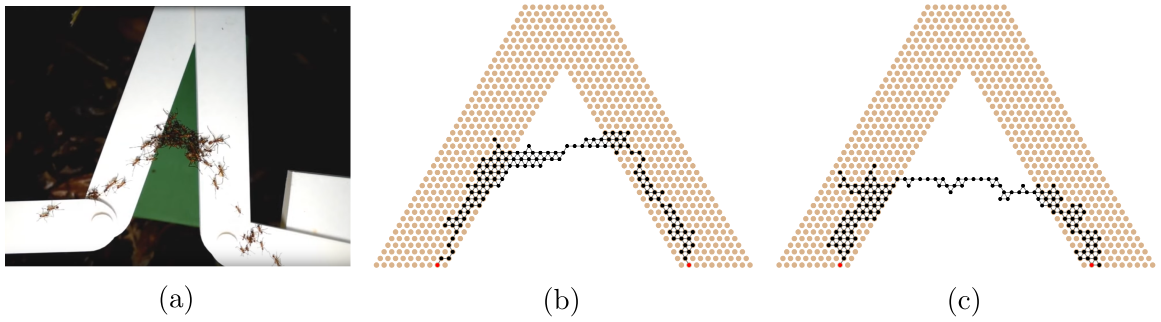


Figure 48. Choosing an Optimization Metric to Produce Ant-Like Bridges. (a) In this image from [168], army ants of the genus *Eciton* build a dynamic bridge which balances the benefit of a shortcut path with the cost of committing ants to the structure. (b) Our shortcut bridging algorithm also balances competing objectives and converges to similar configurations. (c) Minimizing the number of particles in the gap instead of the weighted perimeter results in thin bridges with large clusters of particles on land that do not resemble the ant bridges as closely.

9.1.2 Applying the Stochastic Approach

Recall from Section 8.1.1 that in the stochastic approach to self-organizing particle systems, we define an energy function that captures our objectives for the particle system and then design a Markov chain that favors configurations with desirable energy values in the long run. Care is taken to ensure this Markov chain can be executed in a distributed manner by each particle individually using only local information from their immediate neighborhoods.

For shortcut bridging, we introduce a Hamiltonian over particle system configurations that assigns the lowest energy values to configurations with desirable bridge structures; we then design our algorithm to favor these configurations with small Hamiltonians. We assign each configuration σ a Hamiltonian $H(\sigma) = \overline{p}(\sigma, c)$, its weighted perimeter. Employing a bias parameter λ, the weight of a configuration σ is $w(\sigma, c) = \lambda^{-\overline{p}(\sigma,c)}$, where $w(\sigma, c)$ is the likelihood with which we want our algorithm to yield σ. As λ gets larger, these weights increasingly favor configurations where

$H(\sigma) = \overline{p}(\sigma, c)$ is small and the desired bridging behavior is exhibited. Using a Markov chain, we will ensure that the eventual probability with which we are at state σ is $w(\sigma, c)/Z$, where $Z = \sum_{\sigma'} w(\sigma', c)$ is the normalizing constant.

As was the case for compression (Chapter 8), the corresponding distributed algorithm for shortcut bridging is nearly oblivious — requiring only two bits of persistent memory between particle activations — and inherently robust, forming the best bridge possible with respect to any crashed particles' locations.

9.2 Algorithms for Shortcut Bridging

Recall that for the shortcut bridging problem, we desire for our algorithm to achieve small weighted perimeter, where boundary edges in the gap cost a factor of $c > 1$ more than those on land. The algorithm must balance the competing objectives of having a short path between the two objects while not forming too large of a bridge. We capture these factors by preferring configurations σ that have both small *perimeter* $p(\sigma)$, the length of the walk around the boundary of the particle system, and small *gap perimeter* $g(\sigma)$, the number of perimeter edges that are in the gap, where edges with one endpoint in the gap and one endpoint on land count as half an edge in the gap. While these objectives may appear to be aligned rather than competing, decreasing the length of the overall perimeter increases the gap perimeter and vice versa in the problem instances we consider (e.g., Figure 47). We note that $\overline{p}(\sigma, c) = p(\sigma) + (c - 1)g(\sigma)$, and thus minimizing weighted perimeter is equivalent to simultaneously minimizing both perimeter and gap perimeter.

9.2.1 The Markov Chain $\mathcal{M}_B$ for Shortcut Bridging

Our Markov chain algorithm incorporates two bias parameters: λ and γ. The value of λ controls the preference for having small perimeter, while γ controls the preference for having small gap perimeter. In this paper, we only consider $\lambda > 1$ and $\gamma > 1$, which correspond to favoring small perimeter and small gap perimeter, respectively. Using a Metropolis filter, we ensure our algorithm converges to stationary distribution π given by $\pi(\sigma) = \lambda^{-p(\sigma)}\gamma^{-g(\sigma)}/Z$ where $Z = \sum_{\sigma'} \lambda^{-p(\sigma')}\gamma^{-g(\sigma')}$ is the normalizing constant necessary to make π a probability distribution. Arithmetic shows:

$$\lambda^{-\overline{p}(\sigma,c)} = \lambda^{-p(\sigma)-(c-1)g(\sigma)} = \lambda^{-p(\sigma)}\left(\lambda^{c-1}\right)^{-g(\sigma)},$$

so setting $\gamma = \lambda^{c-1}$ yields our desired stationary distribution.

We note that λ is the same parameter that controlled compression in Chapter 8, where Markov chain $\mathcal{M}_C$ converged to a distribution over connected, hole-free configurations σ proportional to $\lambda^{-p(\sigma)}$. That work showed that $\lambda > 1$ is not sufficient to ensure compression, so we restrict our attention to $\lambda > 2 + \sqrt{2}$, the regime where compression provably occurs.

To ensure our algorithm maintains some desired invariants throughout its execution, we introduce two properties every movement must satisfy. Specifically, these properties maintain system connectivity[15] and prevent holes from forming. Moreover, both properties are symmetric, which is necessary for the transitions of our Markov chain to be reversible.

We extend the notation for particle neighborhoods from Section 8.2.1 to include objects. For a location ℓ, let $N(\ell)$ denote the set of particles and objects adjacent to

[15]Unlike in compression where the particle system is always connected, here the particle system may disconnect into several components which remain connected through objects.

ℓ. For adjacent locations ℓ and ℓ', we use $N(\ell \cup \ell')$ to denote the set $N(\ell) \cup N(\ell')$, excluding particles or objects occupying ℓ or ℓ'. Let $\mathbb{S} = N(\ell) \cap N(\ell')$ be the particles and objects adjacent to both locations; we note $|\mathbb{S}| \in \{0, 1, 2\}$.

Property 9.2.1. $|\mathbb{S}| \in \{1, 2\}$ *and every particle or object in* $N(\ell \cup \ell')$ *is connected to a particle or object in* $\mathbb{S}$ *by a path through* $N(\ell \cup \ell')$.

Property 9.2.2. $|\mathbb{S}| = 0$, ℓ *and* ℓ' *each have at least one neighbor, all particles and objects in* $N(\ell) \setminus \{\ell'\}$ *are connected by paths within this set, and all particles and objects in* $N(\ell') \setminus \{\ell\}$ *are connected by paths within this set.*

We now present our Markov chain $\mathcal{M}_B$ for a valid instance $(L, O, \sigma_0, c, \alpha)$ of the shortcut bridging problem. For input parameter $\lambda > 2 + \sqrt{2}$, set $\gamma = \lambda^{c-1}$. For simplicity, we assume that the initial configuration σ_0 is not only connected — as required for the instance to be valid — but also hole-free. As was shown for compression in Lemma 8.2.8, our algorithm will eventually eliminate any holes that a connected configuration contains, so we focus on the behavior of the system after this occurs.

Algorithm 14 Markov Chain $\mathcal{M}_B$ for Shortcut Bridging

```
   From any connected, hole-free configuration $\sigma_0$ of $n$ contracted particles, repeat:
1: Choose a particle $P$ uniformly at random from among all $n$ particles; let $\ell$ be its location.
2: Choose a neighboring location $\ell'$ and $q \in (0,1)$ each uniformly at random.
3: if $\ell'$ is unoccupied then
4:     $P$ expands to occupy both $\ell$ and $\ell'$.
5:     Let $\sigma$ (resp., $\sigma'$) be the configuration with $P$ at $\ell$ (resp., at $\ell'$).
6:     if $(i)$ $|N(\ell)| \neq 5$, $(ii)$ $\ell$ and $\ell'$ satisfy Property 9.2.1 or Property 9.2.2, and
7:        $(iii)$ $q < \lambda^{p(\sigma)-p(\sigma')}\gamma^{g(\sigma)-g(\sigma')}$ then
8:         $P$ contracts to $\ell'$.
9:     else $P$ contracts back to $\ell$.
```

Conditions (i) and (ii) of Step 6 ensure that the particle system remains connected and no new holes are formed during the execution of $\mathcal{M}_B$. In particular, condition *(i)* explicitly disallows a particle with five neighbors from moving into the only unoccupied location in its neighborhood, as doing so would create a hole. Condition (*iii*) of Step 7 is the Metropolis filter discussed above; the proposed particle move, once confirmed to be valid, only occurs with probability:

$$\min\{1, \lambda^{p(\sigma)-p(\sigma')}\gamma^{g(\sigma)-g(\sigma')}\} = \min\{1, \lambda^{\overline{p}(\sigma,c)-\overline{p}(\sigma',c)}\},$$

where σ is the configuration with P at location ℓ and σ' is the configuration with P at location ℓ'. Although $p(\sigma) - p(\sigma')$ and $g(\sigma) - g(\sigma')$ are values defined at system-level scale, in the next section we show these differences can be calculated locally.

The state space Ω of $\mathcal{M}_B$ is the set of all configurations reachable from σ_0 via valid transitions of $\mathcal{M}_B$. We conjecture that this includes all connected, hole-free configurations of n particles connected to both objects, but proving all such configurations are reachable from σ_0 is not necessary for our results. Note that the corresponding proof of irreducibility for compression (Lemma 8.2.10) does not generalize here due to the presence of static objects.

9.2.2 The Local, Distributed Algorithm $\mathcal{A}_B$ for Shortcut Bridging

In order for individual particles to run $\mathcal{M}_B$, a Markov chain with centralized control, we must translate $\mathcal{M}_B$ into a local, distributed, algorithm $\mathcal{A}_B$ that fully respects the constraints of the (simplified, sequential) amoebot model (Chapter 2). As was the case for compression, two key issues are (*i*) translating the uniformly at random particle selection in Step 1 of $\mathcal{M}_B$ to particle activations determined by a fair sequential adversary and (*ii*) decoupling a particle's combined expansion and contraction in Steps 4–9 of $\mathcal{M}_B$ into two separate activations since a particle can perform at most one movement per activation. We address the former with Poisson clocks of constant means and the latter with one-bit flags in particle memory; see Section 8.2.2 for details.

An additional issue for this translation that is unique to shortcut bridging arises in the local calculation of the differences $p(\sigma) - p(\sigma')$ and $g(\sigma) - g(\sigma')$ used in Step 6 of $\mathcal{M}_B$. As will be made clear in the following lemma, it is necessary for a moving particle to know whether its neighbors occupy land or gap locations when calculating $g(\sigma) - g(\sigma')$. However, a node's information is only assumed to be accessible to the particle occupying it, so we introduce a P.land variable in particle memory that is equal to 1 if the particle's tail occupies a land location and 0 otherwise. As in the distributed algorithm $\mathcal{A}_C$ for compression, expanded amoebots are treated as if they are still contracted at their tail location when considering neighborhoods $N(\cdot)$.

Lemma 9.2.1. *An expanded particle P occupying adjacent locations ℓ and ℓ' in G_Δ can calculate the values of $p(\sigma) - p(\sigma')$ and $g(\sigma) - g(\sigma')$ in Condition (iii) of $\mathcal{M}_B$ and $\mathcal{A}_B$ using only local information involving ℓ, ℓ', and $N(\ell \cup \ell')$.*

Proof. Observe that $p(\sigma) - p(\sigma')$ and $g(\sigma) - g(\sigma')$ need only be calculated if Conditions (i) and (ii) hold. By Lemma 8.1.3 from compression,

$$p(\sigma) - p(\sigma') = |N(\ell')| - |N(\ell)|,$$

which can be calculated using only local information.

Recall that gap perimeter is defined as the number of boundary edges in the gap, counting edges between gap and land as half an edge; this is equal to the number of particles that are on the perimeter and in the gap, counted with appropriate multiplicity if a particle appears on the perimeter more than once. Given a particle Q and a configuration τ, let $G(Q, \tau)$ be equal to 1 if Q occupies a gap location in τ and 0 otherwise. Let $\delta(Q, \tau)$ be the number of times Q appears on the perimeter of τ.

Then the desired difference is:

$$g(\sigma) - g(\sigma') = \sum_{Q \in \mathcal{S}} \left(G(Q,\sigma)\delta(Q,\sigma) - G(Q,\sigma')\delta(Q,\sigma') \right).$$

Define $\Delta(Q) = \delta(Q,\sigma) - \delta(Q,\sigma')$. For particle P, since Conditions (i) and (ii) of Step 6 hold, $\Delta(P) = 0$. For any particle $Q \notin N(\ell \cup \ell') \cup \{P\}$, $\Delta(Q) = 0$ since its neighborhood is not affected by the movement of P. Moreover, for any particle $Q \neq P$, $G(Q,\sigma) = G(Q,\sigma')$ since it does not move. So:

$$\begin{aligned} g(\sigma) - g(\sigma') &= \sum_{Q \in \mathcal{S}} \left(G(Q,\sigma)\delta(Q,\sigma) - G(Q,\sigma')\delta(Q,\sigma') \right) \\ &= \delta(P,\sigma)\left(G(P,\sigma) - G(P,\sigma') \right) + \sum_{Q \in N(\ell \cup \ell')} G(Q,\sigma)\Delta(Q). \end{aligned}$$

The first term is easily calculated locally. Since σ is connected and hole-free, $\delta(P,\sigma)$ is the number of maximal, consecutive sets of unoccupied locations adjacent to ℓ. Since P is expanded, occupying both ℓ and ℓ', it can see ℓ and ℓ' as land or gap locations to determine $G(P,\sigma)$ and $G(P,\sigma')$. For the summation, P can read Q.land from any neighbor $Q \in N(\ell \cup \ell')$ to determine $G(Q,\sigma)$, so it remains to show that P can locally calculate $\Delta(Q)$ for any $Q \in N(\ell \cup \ell')$. First suppose that Q occupies a location adjacent to ℓ but not ℓ'. Then:

$$\Delta(Q) = \begin{cases} -1 & \text{if } Q \text{ has two neighbors in } N(\ell); \\ 1 & \text{if } Q \text{ has no neighbors in } N(\ell); \\ 0 & \text{otherwise.} \end{cases}$$

The opposite is true if Q occupies a location adjacent to ℓ' but not ℓ:

$$\Delta(Q) = \begin{cases} 1 & \text{if } Q \text{ has two neighbors in } N(\ell'); \\ -1 & \text{if } Q \text{ has no neighbors in } N(\ell'); \\ 0 & \text{otherwise.} \end{cases}$$

Lastly, if Q occupies a location adjacent to both ℓ and ℓ', then:

$$\Delta(Q) = \begin{cases} 0 & \text{if } Q \text{ has zero or two neighbors in } N(\ell \cup \ell'); \\ -1 & \text{if } Q \text{ shares a neighbor with } \ell \text{ but not } \ell'; \\ 1 & \text{if } Q \text{ shares a neighbor with } \ell' \text{ but not } \ell; \end{cases}$$

In all cases, P can calculate $\Delta(Q)$, and thus also $g(\sigma) - g(\sigma')$, using only local information. □

Algorithm 15 Local, Distributed Algorithm $\mathcal{A}_B$ for Shortcut Bridging

```
    If $A$ is contracted:
 1: Let $\ell$ denote the current location of $A$.
 2: Choose a neighboring location $\ell'$ uniformly at random from the six possible choices.
 3: if $\ell'$ is unoccupied and $A$ has no expanded neighbors then
 4:     $A$ expands to occupy both $\ell$ and $\ell'$.
 5:     if there are no expanded amoebots adjacent to $\ell$ or $\ell'$ then
 6:         $A$.flag ← TRUE.
 7:     else $A$.flag ← FALSE.

    If $A$ is expanded:
 8: Choose $q \in (0,1)$ uniformly at random.
 9: Let $N^*(\cdot) \subseteq N(\cdot)$ be the set of neighboring amoebots excluding any heads of expanded amoebots.
10: Let $\Delta p = p(\sigma) - p(\sigma')$ and $\Delta g = g(\sigma) - g(\sigma')$ be calculated according to Lemma 9.2.1.
11: if (i) $|N^*(\ell)| \neq 5$, (ii) $\ell$ and $\ell'$ satisfy Property 9.2.1 or Property 9.2.2 with respect to $N^*(\cdot)$,
    (iii) $q < \lambda^{\Delta p}\gamma^{\Delta g}$, and (iv) $A$.flag = TRUE then
12:     $A$ contracts to $\ell'$.
13:     if $\ell'$ is a land location then $A$.land ← 1.
14:     else $A$.land ← 0.
15: else $A$ contracts back to $\ell$.
```

With Lemma 9.2.1 in place, we can present the local, distributed algorithm $\mathcal{A}_B$ for shortcut bridging (Algorithm 15). The pseudocode is written from the perspective of an amoebot A that has been activated according to its Poisson clock. As was the case for compression, any objective that can be accomplished by Makrov chain $\mathcal{M}_B$ can be accomplished by the distributed algorithm $\mathcal{A}_B$ and vice versa. Thus, it suffices to analyze $\mathcal{M}_B$ using the respective tools of Markov chain analysis when characterizing the long-run behavior of algorithm $\mathcal{A}_B$, as we do in the next section.

9.3 Analysis of the Markov Chain $\mathcal{M}_B$

We begin our analysis with some useful properties of Markov chain $\mathcal{M}_B$. Our first two claims follow from the analysis of the Markov chain $\mathcal{M}_C$ for compression given in Section 8.2.4 and basic properties of Markov chains and our particle systems.

Lemma 9.3.1. *If σ_0 is connected and has no holes, then at every iteration of $\mathcal{M}_B$, the current configuration is connected and has no holes.*

Proof. Lemmas 8.2.1 and 8.2.2 proved that no transitions made by $\mathcal{M}_C$ could introduce holes or disconnect the particle system. Since the moves allowed by $\mathcal{M}_B$ are a subset of those allowed in $\mathcal{M}_C$ (since the local properties checked at each iteration are the same), $\mathcal{M}_B$ cannot introduce holes or disconnect the system either. □

Lemma 9.3.2. *If σ_0 has no holes, then $\mathcal{M}_B$ is ergodic.*

Proof. Markov chain $\mathcal{M}_B$ is irreducible because we defined Ω to be precisely those configurations reachable by valid transitions of $\mathcal{M}_B$ starting from σ_0. $\mathcal{M}_B$ is aperiodic because at each iteration there is a probability of at least $1/6$ that no move occurs, as each particle has at least one neighbor. Thus, $\mathcal{M}_B$ is ergodic. □

As $\mathcal{M}_B$ is finite and ergodic, it converges to a unique stationary distribution, and we can find that distribution using detailed balance.

Lemma 9.3.3. *The stationary distribution of $\mathcal{M}_B$ is*

$$\pi(\sigma) = \lambda^{-p(\sigma)}\gamma^{-g(\sigma)}/Z,$$

where $Z = \sum_{\sigma' \in \Omega} \lambda^{-p(\sigma')}\gamma^{-g(\sigma')}$.

Proof. Let M be the transition matrix for $\mathcal{M}_B$; i.e., $M(\sigma, \tau)$ denotes the probability that $\mathcal{M}_B$ transitions from configuration σ to configuration τ in a single step. Properties 9.2.1 and 9.2.2 ensure that particle P moving from location ℓ to location ℓ' is valid if and only if P moving from ℓ' to ℓ is. This implies that for any configurations σ and τ, $M(\sigma, \tau) > 0$ if and only if $M(\tau, \sigma) > 0$. Using this, we easily verify the lemma via detailed balance.

Let $\sigma, \tau \in \Omega$ be distinct configurations that differ by one valid move of a particle P from location ℓ to neighboring location ℓ', and let n be the number of particles in the system. Then,

$$M(\sigma, \tau) = \frac{1}{n} \cdot \frac{1}{6} \cdot \min\{\lambda^{p(\sigma)-p(\tau)}\gamma^{g(\sigma)-g(\tau)}, 1\}, \text{ and}$$
$$M(\tau, \sigma) = \frac{1}{n} \cdot \frac{1}{6} \cdot \min\{\lambda^{p(\tau)-p(\sigma)}\gamma^{g(\tau)-g(\sigma)}, 1\}.$$

W.l.o.g., assume that λ and γ satisfy $\lambda^{p(\sigma)-p(\tau)}\gamma^{g(\sigma)-g(\tau)} \leq 1$. Then,

$$\pi(\sigma)M(\sigma, \tau) = \frac{\lambda^{-p(\sigma)}\gamma^{-g(\sigma)}}{Z} \cdot \frac{\lambda^{p(\sigma)-p(\tau)}\gamma^{g(\sigma)-g(\tau)}}{6n} = \frac{\lambda^{-p(\tau)}\gamma^{-g(\tau)}}{Z} \cdot \frac{1}{6n} = \pi(\tau)M(\tau, \sigma)$$

The definition of Z implies π satisfies $\sum_{\sigma' \in \Omega} \pi(\sigma') = 1$, so π is a valid probability distribution and we conclude π is the unique stationary distribution of $\mathcal{M}_B$. $\square$

The stationary distribution can be alternatively expressed using weighted perimeter $\overline{p}(\sigma, c) = p(\sigma) + (c-1)g(\sigma)$.

Lemma 9.3.4. *For $c = 1 + \log_\lambda \gamma$, the stationary distribution of $\mathcal{M}$ is given by*

$$\pi(\sigma) = \lambda^{-\overline{p}(\sigma, c)}/Z,$$

where $Z = \sum_{\sigma' \in \Omega} \lambda^{-\overline{p}(\sigma', c)}$.

We now prove our main result.

Theorem 9.3.5. *Consider any $\lambda > 2+\sqrt{2} =: \nu$, any $\gamma > 1$, and any $\alpha > \frac{\log \lambda}{\log \lambda - \log \nu} > 1$. There exists $n^* \geq 0$ such that for all $n > n^*$, the probability that a random sample σ drawn according to the stationary distribution π of $\mathcal{M}_B$ over configurations Ω with n particles satisfies*

$$\overline{p}(\sigma, 1 + \log_\lambda \gamma) > \alpha \cdot \overline{p}_{min}$$

is exponentially small in n, where $\overline{p}_{min}$ is the minimum weighted perimeter of a configuration in Ω.

Proof. This proof mimics that of α-compression (Theorem 8.3.5), but additional insights are necessary to accommodate the difficulties introduced by considering weighted perimeter instead of perimeter. Throughout we consider weighted perimeter $\overline{p}(\sigma) = \overline{p}(\sigma, 1 + \log_\lambda \gamma)$.

Define the weight of a configuration $\sigma \in \Omega$ to be:

$$w(\sigma) := \pi(\sigma) \cdot Z = \lambda^{-p(\sigma)} \gamma^{-g(\sigma)} = \lambda^{-\overline{p}(\sigma)},$$

where $Z = \sum_{\sigma' \in \Omega} \lambda^{-p(\sigma')} \gamma^{-g(\sigma')}$. For a set of configurations $S \subseteq \Omega$, we define its weight $w(S) = \sum_{\sigma \in S} w(\sigma)$; analogously, let $\pi(S) = \sum_{\sigma \in S} \pi(\sigma) = w(S)/Z$. Let $\sigma_{min} \in \Omega$ be a configuration with minimal weighted perimeter $\overline{p}_{min}$, and let S_α be the set of configurations with weighted perimeter at least $\alpha \cdot \overline{p}_{min}$. We show that for sufficiently large n,

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

where $\zeta < 1$. The first equality and inequality follow directly from the definitions of Z, w, and σ_{min}. We focus on the last inequality.

Stratify S_α into sets of configurations that have the same weighted perimeter; there are at most $\mathcal{O}(n^2)$ such sets, as the total perimeter and gap perimeter can each take on at most $\mathcal{O}(n)$ values. Label these sets as $A_1, A_2, \ldots, A_m$ in order of increasing weighted perimeter, where m is the total number of distinct weighted perimeters of configurations in S_α. Let $\overline{p}_i$ be the weighted perimeter of all configurations in set A_i; since $A_i \subseteq S_\alpha$, then $\overline{p}_i \geq \alpha \cdot \overline{p}_{min}$.

Each configuration $\sigma \in A_i$ has the same weight, namely $w(\sigma) = \lambda^{-\overline{p}_i}$, so to bound $w(A_i) = |A_i|\lambda^{-\overline{p}_i}$ it suffices to bound $|A_i|$. A configuration with weighted perimeter $\overline{p}_i$ has perimeter $p \leq \overline{p}_i$, and Lemma 8.3.3 from the analysis of compression implies the number of connected, hole-free configurations with perimeter p is at most $f(p)\nu^p$, for some subexponential function f. Letting p_{min} denote the minimum possible (unweighted) perimeter of a configuration of n particles, we conclude that:

$$w(A_i) = \lambda^{-\overline{p}_i}|A_i| \leq \lambda^{-\overline{p}_i} \cdot \sum_{p=p_{min}}^{\overline{p}_i} f(p)\nu^p \leq \lambda^{-\overline{p}_i} f_1(\overline{p}_i)\nu^{\overline{p}_i},$$

where $f_1(\overline{p}_i) = \sum_{p=p_{min}}^{\overline{p}_i} f(p)$ is necessarily also a subexponential function because it is a sum of at most a linear number of subexponential terms. So,

$$w(S_\alpha) = \sum_{i=1}^{m} w(A_i) \leq \sum_{i=1}^{m} f_1(\overline{p}_i)\left(\frac{\nu}{\lambda}\right)^{\overline{p}_i} \leq f_2(n)\left(\frac{\nu}{\lambda}\right)^{\alpha \cdot \overline{p}_{min}},$$

where $f_2(n) = \sum_{i=1}^{m} f_1(\overline{p}_i)$ is a subexponential function because $\overline{p}_i = \mathcal{O}(n)$, $m = \mathcal{O}(n^2)$, and f_1 is subexponential. The last inequality above holds as $\lambda > \nu$ and $\overline{p}_i \geq \alpha \cdot \overline{p}_{min}$. Then, since $Z = \sum_{\sigma' \in \Omega} \lambda^{-\overline{p}(\sigma')} \geq \lambda^{-\overline{p}_{min}} = w(\sigma_{min})$,

$$\pi(S_\alpha) = \frac{w(S_\alpha)}{Z} \leq \frac{w(S_\alpha)}{w(\sigma_{min})} \leq f_2(n)\left(\frac{\nu}{\lambda}\right)^{\alpha \cdot \overline{p}_{min}} \lambda^{\overline{p}_{min}} = f_2(n)\left[\lambda\left(\frac{\nu}{\lambda}\right)^{\alpha}\right]^{\overline{p}_{min}}.$$

The constant $\lambda(\nu/\lambda)^\alpha$ is less than 1 whenever $\alpha > \frac{\log\lambda}{\log\lambda - \log\nu}$. Since the perimeter of any configuration of n particles is at least $\sqrt{n}$ (Lemma 8.1.1), $\overline{p}_{min} \geq \sqrt{n}$. Because $f_2(n)$ is subexponentially large but $(\lambda(\nu/\lambda)^\alpha)^{\sqrt{n}}$ is exponentially small, asymptotically the latter term dominates and we conclude there exists $\zeta < 1$ such that for all sufficiently large n,

$$\pi(S_\alpha) \leq f_2(n)(\lambda(\nu/\lambda)^\alpha)^{\sqrt{n}} < \zeta^{\sqrt{n}},$$

which proves the theorem. □

Though Theorem 9.3.5 is proved only in the case where the number of particles is sufficiently large, we expect and observe it to hold for much smaller n. However, we are unable to compute an explicit bound on how large n must be for these results to hold because the exact form of the subexponential function $f(p)$ in the above proof is unknown (see Section 4 of [75] and the references therein).

The following corollary shows that our algorithm solves any instance $(L, O, \sigma_0, c, \alpha)$ of the shortcut bridging problem when parameters λ and γ are chosen accordingly.

Corollary 9.3.6. *The local, distributed algorithm $\mathcal{A}_B$ corresponding to Markov chain $\mathcal{M}_B$ solves any valid instance of the shortcut bridging problem where the number of particles is sufficiently large.*

Proof. Given any valid instance $(L, O, \sigma_0, c, \alpha)$ of the shortcut bridging problem, it suffices to run $\mathcal{A}_B$ starting from configuration σ_0 with parameters $\lambda > (2+\sqrt{2})^{\frac{\alpha}{\alpha-1}}$ and $\gamma = \lambda^{c-1}$. Then $\alpha > \frac{\log(\lambda)}{\log(\lambda)-\log(2+\sqrt{2})} > 1$, so by Theorem 9.3.5 the system reaches and remains with all but exponentially small probability in a set of configurations with weighted perimeter $\overline{p}(\sigma, c) < \alpha \cdot \overline{p}_{min}$, where $\overline{p}_{min}$ is the minimum weighted perimeter of a configuration in Ω. Solving the shortcut bridging problem only requires the weaker condition that this occurs with all but a polynomially small probability, which our algorithm certainly achieves. □

9.4 Dependence on the Gap Angle

To understand the relationship between bridging and shape, we consider V-shaped land masses of various angles (e.g., Figure 47a). We prove our shortcut bridging algorithm has a dependence on the internal angle θ of the gap similar to that of the army ant bridges studied by Reid et al. [168]. We show that when θ is sufficiently small, with all but exponentially small probability the bridge constructed by the particles stays close to the bottom of the gap (away from the apex of angle θ). On the other hand, we show that for some large values of θ, when λ and γ satisfy certain conditions, with all but exponentially small probability the bridge stays close to the top of the gap. We prove these results with a Peierls argument and careful analysis of the geometry of the gap. Simulations of our shortcut bridging algorithm for varying angles can be found in Section 9.5.

We first give a formal construction for the V-shaped land mass L given any $\theta \in (0, \pi)$ and constant width $w \geq 2$. Let $e \in E$ be any edge of the triangular lattice and label its endpoints as v_1 and v_2. Extend line segment ℓ_1 from v_1 such that it forms an angle of $\pi/2 + \theta/2$ with e. Similarly extend line segment ℓ_2 from v_2, of the same length and on the same side of e as ℓ_1, also forming an angle of $\pi/2 + \theta/2$ with e. Segments ℓ_1 and ℓ_2 then differ in their orientation by angle θ. W.l.o.g., we assume ℓ_1 is clockwise from ℓ_2 around e. Let b be the line through ℓ_1 and ℓ_2's other endpoints (not v_1 and v_2). The land mass consists of v_1, v_2, and all vertices of G_Δ that are outside of ℓ_1 and ℓ_2 and from which there exists a lattice path of length at most w to a vertex strictly between ℓ_1 and ℓ_2. Vertices of G_Δ on the opposite side of b from e are not included in the land mass. For example, Figure 49a depicts a land mass with $\theta \sim \pi/6$ and Figure 49b shows another with $\theta \sim \pi/2$; both have width $w = 5$. This careful definition involving edge e is necessary to ensure there are no adjacent land locations on opposite sides of the gap, as could happen for small θ if the land mass is not constructed carefully.

From now on we will, in a slight abuse of notation, refer to the gap locations

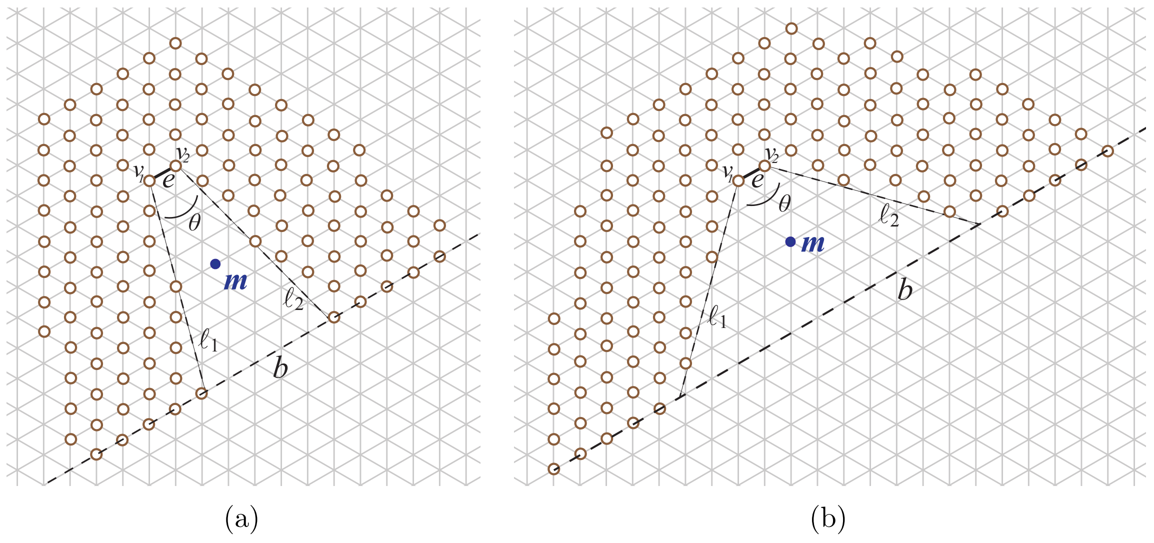


Figure 49. Construction of V-shaped Land Masses by Gap Angle. The land mass L of constant width 5 for (a) a small value of $\theta \approx \pi/6$ and height 8 and (b) a large value of $\theta \approx \pi/2$ and height 9. Point m is the midpoint of the segment between the midpoints of ℓ_1 and ℓ_2, and b is shown as a dashed line.

between ℓ_1 and ℓ_2 as *the gap*. By the *bottom of the gap*, we mean the line b through ℓ_1 and ℓ_2's other endpoints (not v_1 and v_2). We may assume b is a line of the triangular lattice by truncating ℓ_1 and ℓ_2 so that both end on a lattice line; this does not change the land mass L. We also assume $b \cap \ell_1$ and $b \cap \ell_2$ are not vertices of the triangular lattice G_Δ; if they are, we can perturb ℓ_1 and ℓ_2 slightly, without changing the land mass. Note b is always parallel to e.

The *height* of land mass L is the length of a shortest path in G_Δ from v_1 or v_2 to b that only visits land locations; the land mass in Figure 49a has height 8, while the land mass in Figure 49b has height 9. Let m be the midpoint of the segment connecting the midpoints of ℓ_1 and ℓ_2; m is in the center of the gap, halfway between e and b.

The initial configuration σ_0 we consider is a path of width 2 lining the interior sides of the land mass L (see Figure 50). We position the two fixed objects of O in

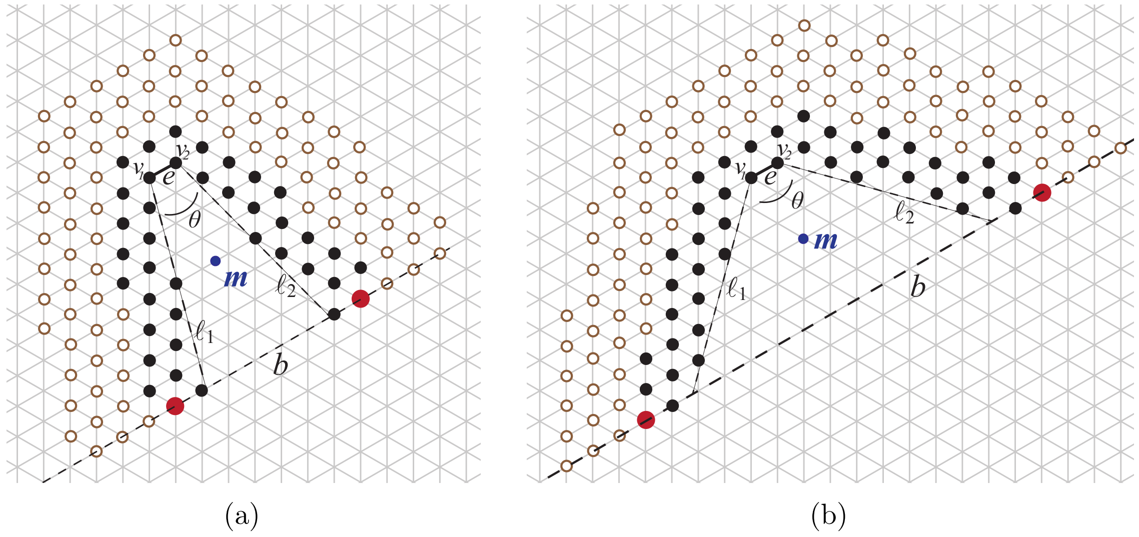


Figure 50. Initial System Configurations on V-shaped Land Masses by Gap Angle. The initial configuration σ_0, with particles shown in black and objects enlarged and red, for (a) a small value of $\theta \approx \pi/6$ and (b) a large value of $\theta \approx \pi/2$. Point m is the midpoint of the segment between the midpoints of ℓ_1 and ℓ_2, and b is shown as a dashed line.

line b at the second vertices outside ℓ_1 and ℓ_2, anchoring the particles on either side of the gap. Note the height of L is exactly the number of particles in σ_0 next to ℓ_1 (or ℓ_2), excluding v_1 and v_2.

Lemma 9.4.1. *Let L be a V-shaped land mass of height k and angle θ. The initial configuration σ_0 has $4k + 5$ particles and two objects.*

Proof. First, suppose $\theta \leq \pi/3$, as in Figure 50a. Each lattice line parallel to e and intersecting ℓ_1 and ℓ_2, up to but not including b, contains exactly four particles. There are k such lattice lines. Line b contains two particles. In the lattice line above and parallel to e, there are three particles. In total, this gives $4k + 2 + 3 = 4k + 5$ particles and two objects.

Now, suppose $\theta > \pi/3$, as in Figure 50b; a different counting approach is necessary. Consider the lattice line through v_1 and the gap location adjacent to v_1 and v_2; this line and all lines parallel to it intersecting ℓ_1 contain exactly two particles, and there are k such lines. The same is true for v_2 and ℓ_2. Uncounted by this approach are five additional particles: the two particles adjacent to each of the two objects, and the particle adjacent to v_1 and v_2. In total, this gives $2k + 2k + 4 + 1 = 4k + 5$ particles and two objects. □

For a given σ, let x be the particle or object contained in line b farthest outside of ℓ_1, and let y be the particle or object in line b farthest outside of ℓ_2. We will refer to the perimeter of σ traversed counterclockwise from x to y as the *inner perimeter* of σ. We say the inner perimeter is *above a point* p if p is to the right of the inner perimeter traversed from x to y; it is *below a point* p if p is to its left.

We can partition Ω into two sets S_1 and S_2, where S_1 contains all configurations whose inner perimeter is strictly above midpoint m of the gap and S_2 contains all configurations whose inner perimeter goes through or below m. We first prove that for $\lambda > 2+\sqrt{2}$ (i.e., in the range of compression) and $\gamma > 1$, there is an angle θ_1 such that for all $\theta < \theta_1$, $\pi(S_1)$ is exponentially small. We then prove that for $\lambda > 2 + \sqrt{2}$ and $\gamma > \lambda^4(2 + \sqrt{2})^4$, there is a θ_2 such that for all $\theta \in (\pi/3, \theta_2)$, $\pi(S_2)$ is exponentially small. We expect much better bounds θ_1 and θ_2 can be obtained with more effort, and that these results generalize to all $\lambda > 2 + \sqrt{2}$ and $\gamma > 1$, but here we simply demonstrate it is possible to give rigorous results about the dependence of the bridge structure on θ.

9.4.1 Proofs for Small θ

We begin with some structural lemmas.

Lemma 9.4.2. *Let L be a V-shaped land mass of height k and angle $\theta \leq \pi/3$. Then any path in G_Δ that starts and ends at the bottom of the gap and goes strictly above the midpoint m of the gap has length at least $k+1$.*

Proof. For $\theta \leq \pi/3$, there are $k-1$ lattice lines parallel to b strictly between b and e. Of these lines exactly $\lceil (k-1)/2 \rceil$ are below or contain m. Any path from b to a location above m and back to b must contain at least two vertices in each of these lattice lines, two vertices in b, and one vertex strictly above m, giving a total of

$$3 + 2\lceil (k-1)/2 \rceil \geq 3 + 2((k-1)/2) = k+2$$

vertices. As the length of a path is the number of edges it contains, the path must have length at least $k+1$. $\square$

Lemma 9.4.3. *The i-th lattice line below and parallel to e contains $h(i)$ gap locations between ℓ_1 and ℓ_2, where:*

$$i\sqrt{3}\tan\frac{\theta}{2} \leq h(i) \leq i\sqrt{3}\tan\frac{\theta}{2} + 2.$$

Proof. Let b_i be the i-th lattice line below and parallel to e. We use trigonometry to analyze the length of b_i between ℓ_1 and ℓ_2; see Figure 51a. Consider the triangle formed by b_i, ℓ_1, and the line perpendicular to e at v_1, which we call ℓ^*. Lines ℓ_1 and ℓ^* form an angle of $\theta/2$, and the distance between e and b_i along ℓ^* is $i\sqrt{3}/2$. It follows that the length of b_i between ℓ_1 and ℓ^* is $i\sqrt{3}\tan(\theta/2)/2$. Altogether, this implies b_i between ℓ_1 and ℓ_2 is of length $i\sqrt{3}\tan(\theta/2)+1$. As each edge of the triangular lattice has length 1, this means there are between $i\sqrt{3}\tan(\theta/2)$ and $i\sqrt{3}\tan(\theta/2)+2$ gap locations in b_i, as claimed. $\square$

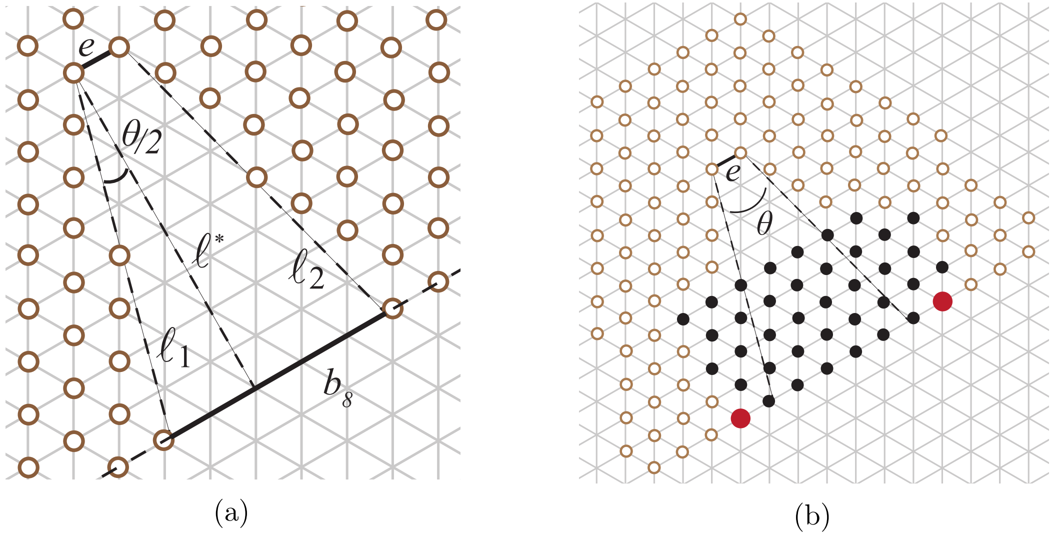


Figure 51. Supporting Figures for the Analysis of Small Gap Angles. (a) A depiction of the notation used in the proof of Lemma 9.4.3; the intersection of b_8 and the gap is depicted as a solid segment, which is of length $8\sqrt{3}\tan(\theta/2)+1$ and contains 4 gap locations. (b) The configuration σ^* used in Lemma 9.4.4 for $\theta=\pi/6$ and $k=8$.

Lemma 9.4.4. *Let L be a V-shaped land mass of height k and angle $\theta \leq \pi/3$. Then the normalizing constant Z of the stationary distribution π of $\mathcal{M}_B$ satisfies*

$$Z \geq C\left[(\lambda\gamma)^{-2\sqrt{3}\tan\frac{\theta}{2}}\right]^k,$$

for a constant C that depends on θ, λ, and γ but not on k.

Proof. Observe that $Z=\sum_{\sigma\in\Omega}\lambda^{-p(\sigma)}\gamma^{-g(\sigma)}$ satisfies $Z \geq \lambda^{-p(\sigma')}\gamma^{-g(\sigma')}$ for any $\sigma' \in \Omega$. We now construct a particular $\sigma^* \in \Omega$ (Figure 51b) and calculate its perimeter and gap perimeter. Let σ^* contain a straight line of particles along b connecting the two objects, and let u be the number of objects and particles in this line. By Lemma 9.4.3, since $b=b_k$ and u includes two particles on land as well as two objects,

$$k\sqrt{3}\tan\frac{\theta}{2}+4 \leq u \leq k\sqrt{3}\tan\frac{\theta}{2}+6.$$

Continue constructing σ^* by placing rows of u particles above this initial row such that the row starts and ends on opposite sides of the gap. By Lemma 9.4.1, there are

$4k+7$ total objects and particles, so there will be $v = \lceil (4k+7)/u \rceil$ such rows, with the last row possibly incomplete. We note that v satisfies:

$$v = \left\lceil \frac{4k+7}{u} \right\rceil \leq \frac{4k+7}{u} + 1 \leq \frac{4k+7}{k\sqrt{3}\tan\frac{\theta}{2}+4} + 1 \leq \frac{4}{\sqrt{3}\tan\frac{\theta}{2}} + \frac{7}{4} + 1 \leq \frac{4}{\sqrt{3}\tan\frac{\theta}{2}} + 3$$

and:

$$v = \left\lceil \frac{4k+7}{u} \right\rceil \geq \frac{4k+7}{u} \geq \frac{4k+7}{k\sqrt{3}\tan\frac{\theta}{2}+6} \geq \frac{4k}{k\sqrt{3}\tan\frac{\theta}{2}+6k} \geq \frac{4}{\sqrt{3}\tan\frac{\theta}{2}+6}$$

Configuration σ^* has perimeter at most $2u + 2v - 4$ and gap perimeter at most $u - 4 + z$, where z is the number of particles occupying gap locations in the upper perimeter of σ^*. These z remaining particles must be in either the $(k-v+1)$-th or $(k-v+2)$-th lattice lines below e, so we can bound z by again applying Lemma 9.4.3:

$$z \leq (k-v+1)\sqrt{3}\tan\frac{\theta}{2} + 2.$$

Altogether, this implies:

$$\begin{aligned} p(\sigma^*) &\leq 2u + 2v - 4 \\ &\leq 2k\sqrt{3}\tan\frac{\theta}{2} + 12 + \frac{8}{\sqrt{3}\tan\frac{\theta}{2}} + 6 - 4 \\ &\leq k\left(2\sqrt{3}\tan\frac{\theta}{2}\right) + \left(\frac{8}{\sqrt{3}\tan\frac{\theta}{2}} + 14\right), \end{aligned}$$

and:

$$\begin{aligned} g(\sigma^*) &\leq u - 4 + z \\ &\leq k\sqrt{3}\tan\frac{\theta}{2} + 6 - 4 + (k-v+1)\sqrt{3}\tan\frac{\theta}{2} + 2 \\ &\leq 2k\sqrt{3}\tan\frac{\theta}{2} + \left(-\frac{4}{\sqrt{3}\tan\frac{\theta}{2}+6} + 1\right)\sqrt{3}\tan\frac{\theta}{2} + 4 \\ &\leq k\left(2\sqrt{3}\tan\frac{\theta}{2}\right) + \left(\sqrt{3}\tan\frac{\theta}{2} - \frac{4\sqrt{3}\tan\frac{\theta}{2}}{\sqrt{3}\tan\frac{\theta}{2}+6} + 4\right) \end{aligned}$$

We note that the second parentheses in the final bounds above for $p(\sigma^*)$ and $g(\sigma^*)$ are constants that only depend on θ. This implies that there is a constant

$$C = \lambda^{-\left(14+\frac{8}{\sqrt{3}\tan\frac{\theta}{2}}\right)}\gamma^{-\left(\sqrt{3}\tan\frac{\theta}{2}-\frac{4\sqrt{3}\tan\frac{\theta}{2}}{\sqrt{3}\tan\frac{\theta}{2}+6}+4\right)}$$

such that:

$$Z \geq \lambda^{-p(\sigma^*)}\gamma^{-g(\sigma^*)} \geq C\left[(\lambda\gamma)^{-2\sqrt{3}\tan\frac{\theta}{2}}\right]^k$$

As claimed, C depends only on λ, γ, and θ, and is independent of k. $\square$

Theorem 9.4.5. *Let $\lambda > 2+\sqrt{2} =: \nu$ and $\gamma > 1$. Then there exists a constant θ_1 such that for all V-shaped land masses with angle $\theta < \theta_1$, the probability that the inner perimeter is above midpoint m is exponentially small in k, the height of the gap, provided k is sufficiently large. In particular,*

$$\theta_1 = 2\tan^{-1}\left(\frac{\log_{\lambda\gamma}(\lambda/\nu)}{\sqrt{3}}\right).$$

Proof. Recall that $S_1 \subseteq \Omega$ is the set of configurations for which the inner perimeter is strictly above m. We show that S_1 has exponentially small weight at stationarity; in particular, we show $\pi(S_1)$ is bounded above by $f_2(k)\xi^k$, where $f_2(k)$ is a subexponential function and $\xi < 1$ is a constant.

If $\sigma \in S_1$, then by Lemma 9.4.2 we have $p(\sigma) \geq 2k+2$, as its inner perimeter — and thus the rest of the perimeter as well — must be above m. Furthermore, because the perimeter by definition includes both objects and particles which number $4k+7$ by Lemma 9.4.1, any configuration $\sigma \in \Omega$ has $p(\sigma) \leq 2(4k+7)-2 = 8k+12$. In Section 8.3, we exploited a connection to self-avoiding walks in the hexagon lattice to show the number of connected, hole-free configurations with perimeter p is at most $f(p)\nu^p$ for some subexponential function f (Lemma 8.3.3). This is certainly also an upper bound on the number of configurations in S_1 with perimeter p. Because

$\gamma^{-g(\sigma)} < 1$, we have:

$$\pi(S_1) = \sum_{\sigma \in S_1} \frac{\lambda^{-p(\sigma)}\gamma^{-g(\sigma)}}{Z} < \sum_{p=2k+2}^{8k+12} \frac{f(p)\nu^p\lambda^{-p}}{Z}.$$

Let $f_1(k) = \sum_{p=2k+2}^{8k+12} f(p)$, and note that this function is subexponential in k because its number of summands is linear in k. Because $\lambda > \nu$ and $p \geq 2k+2$, we have that:

$$\pi(S_1) \leq \frac{f_1(k)\left(\frac{\nu}{\lambda}\right)^{2k+2}}{Z}.$$

By Lemma 9.4.4, there is a constant $C_1 = \nu^2/(\lambda^2 C)$ such that:

$$\pi(S_1) \leq \frac{f_1(k)\left(\frac{\nu}{\lambda}\right)^{2k+2}}{C\left[(\lambda\gamma)^{-2\sqrt{3}\tan\frac{\theta}{2}}\right]^k} = C_1 f_1(k)\left(\frac{\nu(\lambda\gamma)^{\sqrt{3}\tan\frac{\theta}{2}}}{\lambda}\right)^{2k}$$

For all $\theta < 2\tan^{-1}\left(\log_{\lambda\gamma}(\lambda/\nu)/\sqrt{3}\right)$, the term in parentheses above is less than one:

$$\frac{\nu(\lambda\gamma)^{\sqrt{3}\tan\frac{\theta}{2}}}{\lambda} < \frac{\nu(\lambda\gamma)^{\log_{\lambda\gamma}\left(\frac{\lambda}{2+\sqrt{2}}\right)}}{\lambda} = 1$$

Because $C_1 f_1(k)$ is a subexponential function but the term above, raised to the $2k$ power, is exponentially small, the latter eventually dominates and we conclude there is a constant $\xi < 1$ such that for sufficiently large k, $\pi(S_1) < \xi^k$, proving the theorem. $\square$

Since $n = 4k+5$ by Lemma 9.4.1, the probability that the inner perimeter is above point m is also exponentially small in n, the number of particles. As an example, for $\lambda = 4$ and $\gamma = 2$ (the parameters of the simulations in Figures 54 and 55), our methods give $\theta_1 = 0.0879 \approx 5.03°$. However, simulations suggest this bound is far from tight. In general, as λ increases, so does the angle θ_1: a stronger bias towards a shorter perimeter means the bridge forms closer to the bottom of the gap and at even larger angles the bridge remains below m. Similarly, as γ decreases the bridge moves down towards the bottom of the gap and at even larger angles remains below m.

As with Theorem 9.3.5, we are unable to give explicit bounds on the "sufficiently large k" required by the statement of Theorem 9.4.5 because determining the exact form of the subexponential function $f(p)$ in the above proof remains an open problem (see Section 4 of [75]). However, we expect and observe that the claims of this theorem hold even for the small k for which our proofs do not apply.

9.4.2 Proofs for Large θ

We now consider the set $S_2 = \Omega \setminus S_1$, which consists of all configurations where the inner perimeter goes through or below m. We will show that for some large angles θ, for all $\lambda > 2 + \sqrt{2}$ and $\gamma > (2 + \sqrt{2})^4 \lambda^4$, $\pi(S_2)$ is exponentially small. While a lower bound on γ is necessary for the proofs presented below, we believe this is an artifact of our proof rather than the problem itself and suspect this requirement can be loosened or removed altogether.

For $\theta \geq \pi/3$, it is no longer true that a V-shaped land mass of height k has exactly $k-1$ lattice lines between b and e. We define a new quantity q, the *gap depth*, as the length of a shortest path from e to b in G_Δ; unlike in the definition of the height k of a gap, this shortest path is not required to stay on land locations. The Euclidean distance between e and b is then $\sqrt{3}q/2$. Furthermore, q can be expressed as a function of k and θ.

Lemma 9.4.6. *For a V-shaped land mass of height k and angle $\theta \geq \pi/3$, the gap depth q satisfies*

$$k = \left\lceil \left(\frac{1}{2} + \frac{\sqrt{3}}{2} \tan \frac{\theta}{2} \right) q \right\rceil$$

Proof. Consider the path from v_1 to line b that leaves v_1 forming an angle of $2\pi/3$ with e, and then proceeds along b until it reaches a land location; see Figure 52, where

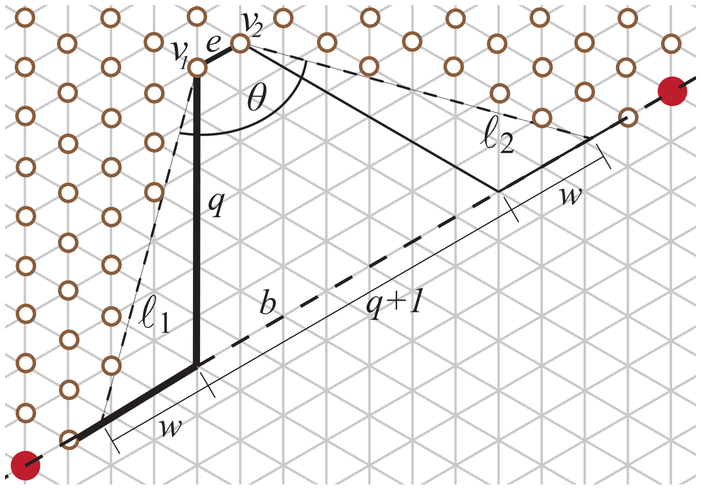


Figure 52. Calculating the Gap Depth. The path of length k (bold) from vertex v_1 to the first land location in line b considered in the proof of Lemma 9.4.6; this path is used to calculate the gap height k in terms of the gap depth q. By also considering the reflection of this path from v_2 (solid line), we can calculate the distance between the two objects to be $q + 2\lceil w \rceil + 3$ (Lemma 9.4.7).

this path is shown in bold. The total length of this path is k, and its first segment from v_1 to b is length q. Let w be the length of b between this path's turning point and ℓ_1; then $k = q + \lceil w \rceil$. This path and ℓ_1 form an obtuse triangle where two sides have lengths q and w, respectively. The angle opposite the side of length w is $\theta/2 - \pi/6$, while the angle opposite the side of length q is $\pi - 2\pi/3 - (\theta/2 - \pi/6) = \pi/2 - \theta/2$. Length w can be calculated in terms of length q with the law of sines:

$$w = \frac{\sin\left(\frac{\theta}{2} - \frac{\pi}{6}\right)}{\sin\left(\frac{\pi}{2} - \frac{\theta}{2}\right)} q = \frac{\sin\frac{\theta}{2}\cos\frac{\pi}{6} - \cos\frac{\theta}{2}\sin\frac{\pi}{6}}{\cos\frac{\theta}{2}} q = \frac{\frac{\sqrt{3}}{2}\sin\frac{\theta}{2} - \frac{1}{2}\cos\frac{\theta}{2}}{\cos\frac{\theta}{2}} q = \frac{q\sqrt{3}}{2}\tan\frac{\theta}{2} - \frac{q}{2}$$

Because q is an integer, it follows that:

$$k = q + \lceil w \rceil = \left\lceil q + \frac{q\sqrt{3}}{2}\tan\frac{\theta}{2} - \frac{q}{2} \right\rceil = \left\lceil \left(\frac{1}{2} + \frac{\sqrt{3}}{2}\tan\frac{\theta}{2}\right) q \right\rceil,$$

which is the desired result. □

For simplicity, we do the bulk of our analysis using the gap depth q instead of the

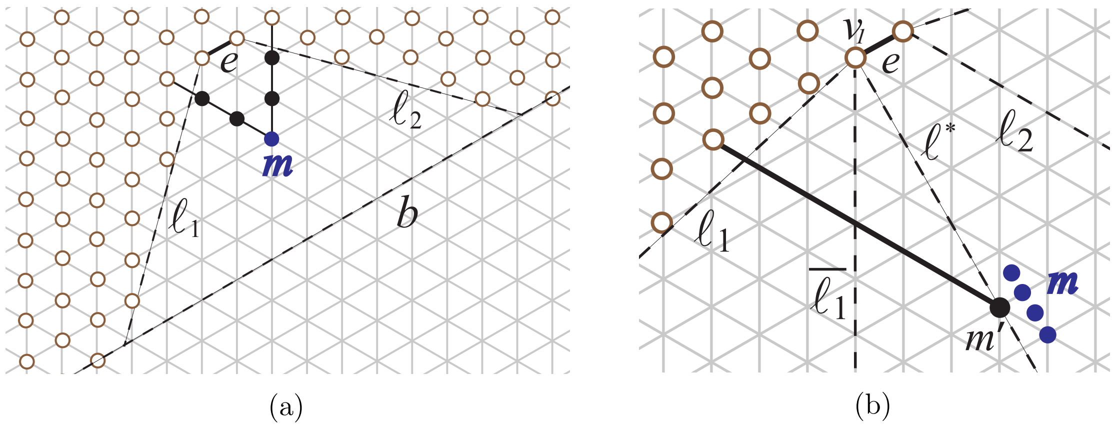


Figure 53. Supporting Figures for the Analysis of Large Gap Angles. From the proof of Lemma 9.4.8: (a) An example of a shortest path between land locations on opposite sides of the gap passing through midpoint m. (b) The four possible locations for midpoint m for which a shortest path passing through or below m contains m', and a shortest path from m' to a land location (solid line).

gap height k. The previous lemma shows that proving an expression is exponentially small in q implies it is also exponentially small in k.

Lemma 9.4.7. *For any V-shaped land mass of gap depth q and angle $\theta \geq \pi/3$, any configuration σ has perimeter at least*

$$p(\sigma) \geq \left(2\sqrt{3}\tan\frac{\theta}{2}\right)q + 6.$$

Proof. We first bound the distance between the two objects on either side of the gap. Using the length w from the proof of Lemma 9.4.6, the distance between the two objects in any configuration is $q + 2\lceil w \rceil + 3 \geq q + 2w + 3$ (see Figure 52). The perimeter of any particle configuration is at least twice this distance, so for any σ,

$$p(\sigma) \geq 2q + 4w + 6 = 2q + 4\left(\frac{q\sqrt{3}}{2}\tan\frac{\theta}{2} - \frac{q}{2}\right) + 6 = \left(2\sqrt{3}\tan\frac{\theta}{2}\right)q + 6,$$

which is the desired bound. □

Lemma 9.4.8. *For any V-shaped land mass of gap depth q and angle $\theta > \pi/3$, any configuration $\sigma \in S_2$ (passing below or through midpoint m of the gap) has gap perimeter $g(\sigma) \geq \frac{q}{2}$.*

Proof. If $\sigma \in S_2$, i.e., if its inner perimeter passes through or below m, then it must contain a path that starts and ends at land locations and also passes through or below m. We consider all such paths and give a lower bound on the number of gap locations they must contain. The shortest such paths start and end on opposite sides of the gap, so we focus on paths of this type.

If m is a vertex of G_Δ, one shortest path between land locations passing through m leaves m along the two lattice lines not parallel to e and follows them until reaching the land mass, as in Figure 53a. If m is on a lattice edge, a shortest path passing below m is constructed in the same way, beginning from each of the edge's endpoints. Otherwise, if m is neither a lattice point nor on a lattice edge, the same procedure is followed for the first lattice point or lattice edge below m. In all cases, let m' be the point of intersection between this path and ℓ^*, the line perpendicular to e through v_1. Figure 53b shows all the possible locations of m producing a particular m'. Inspection shows that in all of these cases, m' is contained in the $2\lfloor\frac{q+1}{4}\rfloor$-th lattice line below e.

Let $\overline{\ell_1}$ be the line from v_1 to b forming an angle of $2\pi/3$ with e (see Figure 53b). Because $\theta > \pi/3$, all vertices of G_Δ contained in $\overline{\ell_1}$ except v_1 are gap locations. Any shortest path from m' to a land location must share a vertex of G_Δ with line $\overline{\ell_1}$. Because m' is in the $2\lfloor\frac{q+1}{4}\rfloor$-th lattice line below e, any path from m' to $\overline{\ell_1}$ is of length at least $\lfloor\frac{q+1}{4}\rfloor$ and contains at least $\lfloor\frac{q+1}{4}\rfloor + 1$ gap locations, including both of its endpoints. By symmetry, this means any path between land locations passing below m, and thus any inner perimeter of a particle configuration passing below m, contains at least

$$2\left(\left\lfloor \frac{q+1}{4} \right\rfloor + 1\right) \geq 2\left(\frac{q-2}{4} + 1\right) \geq \frac{q}{2}$$

gap locations, as claimed. □

Theorem 9.4.9. *Let $\lambda > 2 + \sqrt{2} =: \nu$ and $\gamma > (\lambda\nu)^4$. Then there exists a constant $\theta_2 > \pi/3$ such that for all V-shaped land masses with angle $\theta \in (\pi/3, \theta_2)$, the probability that the inner perimeter goes through or below midpoint m is exponentially small in k, the height of the gap, provided k is sufficiently large.*

Proof. Recall S_2 is the set of all configurations whose inner perimeter goes through or below m. We show that $\pi(S_2)$ is exponentially small in k, the height of the gap. By definition,

$$\pi(S_2) = \frac{\sum_{\sigma \in S_2} \lambda^{-p(\sigma)}\gamma^{-g(\sigma)}}{Z}.$$

By Lemma 9.4.1, the number of particles and objects in σ_0 for a land mass of height k is $4k+7$. Since σ_0 is a path of width 2 and every particle occupies a land location, $p(\sigma_0) = 4k+7$ and $g(\sigma_0) = 0$. Thus,

$$Z = \sum_{\sigma \in \Omega} \lambda^{-p(\sigma)}\gamma^{-g(\sigma)} \geq \lambda^{-p(\sigma_0)}\gamma^{-g(\sigma_0)} = \lambda^{-4k-7}.$$

It is simpler to work with gap depth q instead of gap height k. By Lemma 9.4.6, k satisfies $k \leq \left(\frac{1}{2} + \frac{\sqrt{3}}{2}\tan\frac{\theta}{2}\right) q + 1$, so:

$$Z \geq \lambda^{-4k-7} \geq \lambda^{-4\left(\frac{1}{2}+\frac{\sqrt{3}}{2}\tan\frac{\theta}{2}\right)q-4-7} = \lambda^{-\left(2+2\sqrt{3}\tan\frac{\theta}{2}\right)q-11}.$$

Combining this with Lemma 9.4.8, we have:

$$\pi(S_2) = \sum_{\sigma \in S_2} \frac{\lambda^{-p(\sigma)}\gamma^{-g(\sigma)}}{Z} \leq \lambda^{\left(2+2\sqrt{3}\tan\frac{\theta}{2}\right)q+11} \sum_{\sigma \in S_2} \lambda^{-p(\sigma)}\gamma^{-\frac{q}{2}}$$

Let p_{min} (resp., p_{max}) be the minimum (resp., maximum) possible perimeter for a valid particle configuration in S_2. By Lemma 9.4.7, $p_{min} \geq 2\sqrt{3}\tan(\theta/2)q$. As shown in the proof of Theorem 9.4.5, $p_{max} = 8k + 12$; in terms of q, by Lemma 9.4.6,

$$p_{max} \leq 8\left(\frac{q}{2} + \frac{q\sqrt{3}}{2}\tan\frac{\theta}{2} + 1\right) + 12 = 4q + 4q\sqrt{3}\tan\frac{\theta}{2} + 20.$$

Once again using Lemma 8.3.3 to upper bound the number of configurations with perimeter p by the expression $f(p)\nu^p$, where f is some subexponential function, we have that:

$$\begin{aligned}
\pi(S_2) &\leq \lambda^{\left(2+2\sqrt{3}\tan\frac{\theta}{2}\right)q+11} \sum_{p=p_{min}}^{p_{max}} f(p)\nu^p\lambda^{-p}\gamma^{-\frac{q}{2}} \\
&\leq \lambda^{\left(2+2\sqrt{3}\tan\frac{\theta}{2}\right)q+11} \left(\sum_{p=p_{min}}^{p_{max}} f(p)\right)\left(\frac{\nu}{\lambda}\right)^{p_{min}}\gamma^{-\frac{q}{2}} \\
&\leq \left(\lambda^{11}\sum_{p=p_{min}}^{p_{max}} f(p)\right)\left(\lambda^{\left(2+2\sqrt{3}\tan\frac{\theta}{2}\right)}\left(\frac{\nu}{\lambda}\right)^{2\sqrt{3}\tan\frac{\theta}{2}}\gamma^{-\frac{1}{2}}\right)^q \\
&= \left(\lambda^{11}\sum_{p=p_{min}}^{p_{max}} f(p)\right)\left(\lambda^2\nu^{2\sqrt{3}\tan\frac{\theta}{2}}\gamma^{-\frac{1}{2}}\right)^q
\end{aligned}$$

The first parentheses is a function $f_1(q)$ that is subexponential in q, as it has a polynomial number of summands based on our calculations of p_{min} and p_{max} (which are expressions in terms of q), and each summand is subexponential. When the term in the second set of parentheses above is less than one, the second factor (this term raised to the q power) is exponentially small in q, the gap depth, and thus for sufficiently large q this term dominates and the entire expression is exponentially small in q. This holds whenever θ satisfies:

$$\theta < 2\tan^{-1}\left(\frac{1}{2\sqrt{3}}\log_\nu\left(\gamma^{1/2}\lambda^{-2}\right)\right) = 2\tan^{-1}\left(\frac{1}{\sqrt{3}}\log_\nu\left(\frac{\gamma^{1/4}}{\lambda}\right)\right) =: \theta_2.$$

Whenever $\gamma^{1/4}/\lambda > \nu$ — i.e., whenever $\gamma > (\lambda\nu)^4$ — the argument of $\tan^{-1}$ above is at least $1/\sqrt{3}$, and thus $\theta_2 > \pi/3$. It follows that whenever $\gamma > (\lambda\nu)^4$ and $\theta \in (\pi/3, \theta_2)$,

$$\pi(S_2) < f_1(q)\psi^q,$$

where $f_1(q)$ is subexponentially large in q and $\psi < 1$ so the second term is exponentially small in q. For sufficiently large q, the second term dominates, and we conclude the weight of set S_2 at stationarity is exponentially small in q. Because k and q differ only by additive and multiplicative constants, it is also exponentially small in k, the gap height, for sufficiently large k. □

As was the case for small angles, here also we have that by Lemma 9.4.1, there are $n = 4k + 5$ particles. Thus, we have that the probability the inner perimeter goes through or below midpoint m when θ is sufficiently large is also exponentially small in n. If we again use the example value of $\lambda = 4$ (as in the simulations depicted in Figures 54 and 55), Theorem 9.4.9 requires $\gamma > (\lambda\nu)^4 \approx 34{,}786$. This value is large, but importantly is constant; i.e., it does not depend on n. For example, when $\lambda = 4$ and $\gamma = 10^5$, our methods show that the resulting bridge remains above midpoint m with high probability for any angle between $\pi/3 = 60°$ and $\theta_2 \approx 1.2234 \approx 70.10°$. On the other hand, a simulation with $\lambda = 4$, $\gamma = 2$, and $\theta = 90°$ is shown in Figure 55c to remain well above the midpoint m, suggesting that this behavior is stable for much smaller values of γ and a much larger range of angles than we were able to prove.

As for Theorems 9.3.5 and 9.4.5, we are unable to give explicit bounds on the "sufficiently large k" required by the statement of Theorem 9.4.9 because the exact form of $f(p)$ in its proof is unknown, but we expect and observe that it holds even for the small k for which our proof does not apply.

9.5 Simulations

We can see the performance of our algorithm from simulation results on a variety of instances. Figure 54 shows snapshots over time for a bridge shortcutting a V-shaped

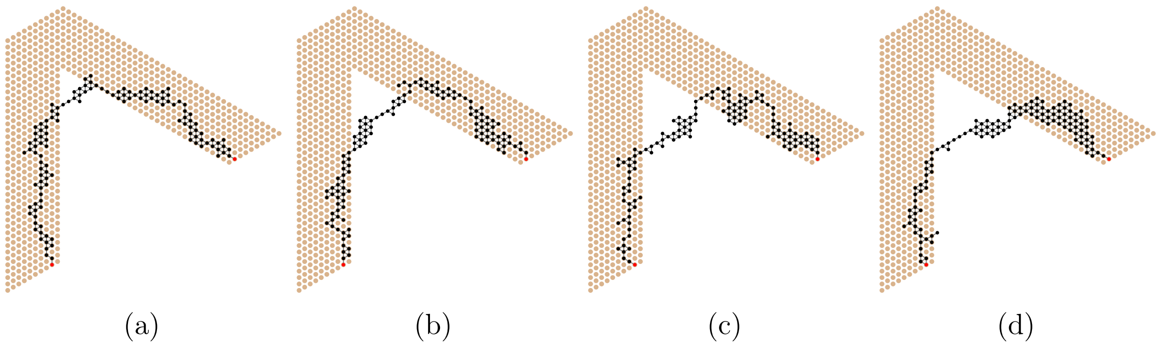


Figure 54. Simulation of Markov Chain $\mathcal{M}_B$ on a V-shaped Land Mass. A particle system using biases $\lambda = 4$ and $\gamma = 2$ to shortcut a V-shaped land mass with $\theta = \pi/3$ after (a) 2 million, (b) 4 million, (c) 6 million, and (d) 8 million iterations of Markov chain $\mathcal{M}_B$, beginning in configuration σ_0 shown in Figure 47a.

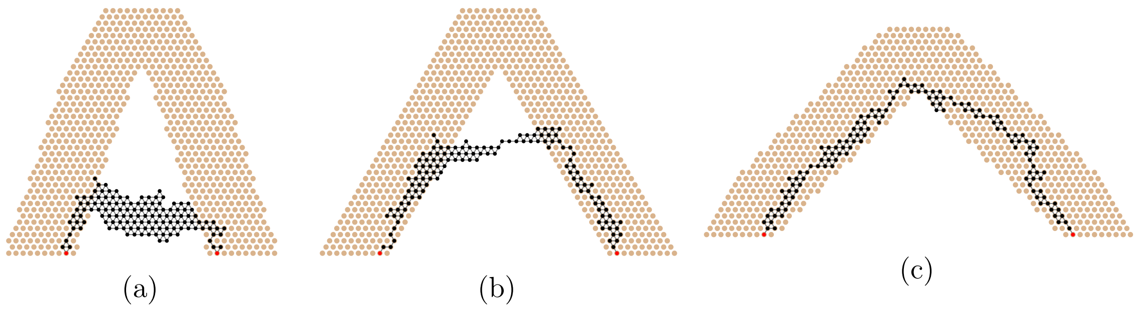


Figure 55. Simulations of Markov Chain $\mathcal{M}_B$ Varying Gap Angle. A particle system using biases $\lambda = 4$ and $\gamma = 2$ to shortcut a V-shaped land mass with angle (a) $\pi/6$, (b) $\pi/3$, and (c) $\pi/2$ after 20 million iterations of Markov chain $\mathcal{M}_B$. For a given angle, the land mass L and initial configuration σ_0 were constructed as described in Section 9.4.

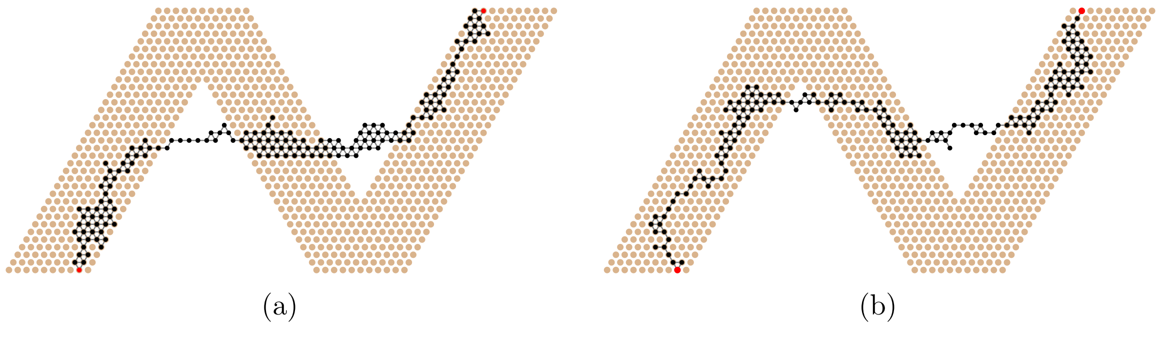


Figure 56. Simulation of Markov Chain $\mathcal{M}_B$ on an N-shaped Land Mass. A particle system using biases $\lambda = 4$ and $\gamma = 2$ to shortcut an N-shaped land mass after (a) 10 million and (b) 20 million iterations of Markov chain $\mathcal{M}_B$, beginning in configuration σ_0 shown Figure 47b.

gap with internal angle $\theta = \pi/3$ and biases $\lambda = 4, \gamma = 2$. Qualitatively, this bridge matches the shape and position of the army ant bridges in [168]. Figure 55 shows the results of an experiment that held λ, γ, and the number of iterations of $\mathcal{M}_B$ constant, varying only the internal angle of the V-shaped land mass. The particle system exhibits behavior consistent with the theoretical results in Section 9.4 and the army ant bridges in [168], shortcutting closer to the bottom of the gap when θ is small and staying almost entirely on land when θ is large. Lastly, Figure 56 shows the resulting bridge structure when the land mass is N-shaped, demonstrating that our algorithm can be generalized beyond the original inspiration of V-shaped land masses to shortcut multiple gaps in more complex structures.

These simulations demonstrate the successful application of our stochastic approach to shortcut bridging. Moreover, experimenting with variants suggests this approach may be useful for other related applications in the future. One related behavior of particular interest is "exploration bridging", where a particle system first explores its environment to discover sites of interest, and then converges to a bridge-like structure between them. We are also interested in formulating alternative local rules for shortcut bridging which yield bridges that appear more "structurally sound", though we suspect the information needed to do so may be difficult to encode in our particle systems due to the constant-size memory constraint of the amoebot model.

Chapter 10

SEPARATION

Across many disciplines spanning computational, physical, and social sciences, heterogeneous systems self-organize into both *separated* (or segregated) and *integrated* states. Examples include molecules exhibiting attractive and repulsive forces, distinct types of bacteria competing for resources while collaborating towards common goals (e.g., [184, 190]), social insects tolerating or aggressing towards those from other colonies (e.g., [117, 174]), and inherent human biases that influence how we form and maintain social groups (e.g., [36, 107, 187]). In each of these, individuals are of different "types": integration occurs when the ensemble gathers together without much preference about the type of their neighbors, while separation occurs when individuals cluster with others of the same type.

We apply the stochastic approach to separation and integration of *heterogeneous* amoebot systems in which amoebots have immutable *colors* [31]. Our inspiration comes from the classical Ising model in statistical physics [111, 188], where the vertices of a graph are assigned positive and negative "spins" and there are rules governing the probability that adjacent vertices have the same spin. Connected to the Ising model is classical work from stochastic processes on the Schelling model of segregation [179, 180], which explores how individuals' micro-motives can induce macro-level phenomena like racial segregation in residential neighborhoods. Recent variants of this model from computer science have investigated the degree of individual bias required to induce such segregation [23, 110], and a related distributed algorithm has been developed [154]. Our work differs from those on the Ising and Schelling models because of natural physical constraints on systems of self-organizing, active particles like ours. For example, interpreting particles of one color to be vertices with positive spin and particles of another color to be vertices with negative spin, this acts like an Ising model, but on a graph that evolves as particles move. Despite these obstacles, we apply tools for rigorously analyzing the Ising and similar models to prove our distributed algorithm for separation and integration accomplishes the desired goals.

While this dissertation focuses on distributed algorithms, it is worth noting that efficient stochastic algorithms for separation can be challenging even with centralized Markov chains. Separation of a region into equitably sized, compact districts has been widely explored recently in the context of gerrymandering, where the aim is to sample colorings of a weighted graph from an appropriately defined stationary distribution [57, 105]. Heuristics for random districting have been discussed in the media, but there are still no known rigorous, efficient algorithms.

Results. We present a distributed algorithm for self-organizing separation and integration that takes as input two bias parameters, λ and γ. Setting $\lambda > 1$ corresponds to particles favoring having more neighbors; this is known to cause compression in homogeneous systems when λ is large enough (Chapter 8, [32]). For separation in the heterogeneous setting, we introduce a second parameter γ, where $\gamma > 1$ corresponds to particles favoring having more neighbors *of their own color.* We then investigate for what values of λ and γ our algorithm yields compression and separation. Informally, a particle system is separated if there is a subset of particles such that (i) the boundary between this subset and the rest of the system is small, (ii) a large majority of particles in this subset are of the same color, say c, and (iii) very few particles with color c exist outside of this subset. This notion of separation (defined formally in Definition 10.1.3) captures what it means for a system to have large monochromatic regions of particles.

We prove that for any $\lambda > 1$ and $\gamma > 4^{5/4} \approx 5.66$ such that $\lambda\gamma > 2(2+\sqrt{2})e^{0.0003} \approx 6.83$, our algorithm accomplishes separation with high probability.[16] However, we prove the opposite for some values of γ close to one; counterintuitively, this even includes some values of $\gamma > 1$, the regime where particles favor having like-colored neighbors. Formally, we prove that for any $\lambda > 1$ and $\gamma \in (79/81, 81/79)$ such that $\lambda(\gamma + 1) > 2(2 + \sqrt{2})e^{0.00003} \approx 6.83$, our algorithm fails to achieve separation (i.e., it achieves integration) with high probability.

Proof Techniques. Because our distributed algorithm is based on a Markov chain, we can use standard tools such as detailed balance to understand its long-term behavior and prove its convergence to a unique probability distribution π over particle system configurations. This stationary distribution π depends on the input parameters λ and γ. Our main contribution is analyzing π for various ranges of λ and γ, showing that a configuration drawn from distribution π is either very likely (for large γ) or very unlikely (for γ close to one) to be separated.

To show separation occurs when λ and γ are both large, we modify the proof technique of *bridging* introduced by Miracle, Pascoe, and Randall [147]. To show separation does not occur when λ is large and γ is small (close to one), we use a probabilistic argument, a Chernoff-type bound, and a decomposition of configurations into different regions. These arguments — both for large and small γ — require that the particle system is compressed; i.e., that the system has perimeter $\Theta(\sqrt{n})$. However,

[16] For separation and integration, we say an event A occurs *with high probability (w.h.p.)* if $\Pr[A] \geq 1 - c^{n^\delta}$, where $0 < c < 1$ and $\delta > 0$ are constants and n is the number of particles. Our w.h.p. results all have $\delta \in \{1/2, 1/2 - \varepsilon\}$, for arbitrarily small $\varepsilon > 0$.

the arguments from [32] showing compression occurs for homogeneous systems when λ is large do not extend to the heterogeneous setting.

We instead turn to the *cluster expansion* from statistical physics to show our separation algorithm achieves compression for large enough γ. The cluster expansion was first introduced in 1937 by Mayer [139], though a more modern treatment can be found in the textbook [90] where it is used to derive several properties of statistical physics models including the Ising and hard-core models. In the past year, the cluster expansion has received renewed attention in the computer science community due to the recent work of Helmuth, Perkins, and Regts that uses the cluster expansion to develop approximate counting and sampling algorithms for low-temperature statistical physics models on lattices including the Potts and hard-core models [103]. Subsequent work has considered similar techniques on expander graphs [116] and random regular bipartite graphs [130]. Inspired by the interpolation method of Barvinok [16, 17], these works give algorithms for estimating partition functions that explicitly calculate the first $\log n$ coefficients of the cluster expansion. We use the cluster expansion differently, to separate the volume and surface contributions to a partition function.

The cluster expansion is a power series representation of $\ln Z$ where Z is a *polymer partition function*. We relate each of our quantities of interest to a particular polymer partition function, and then use a version of the Kotecký-Preiss condition [123] to show that the power series in the cluster expansion is convergent for the ranges of parameters we are interested in. We then use this convergent cluster expansion to split our polymer partition functions into a *volume term*, depending only on the size of the region of interest, and a *surface term*, depending only on its perimeter. This separation into volume and surface terms turns out to be the key to our compression argument, both for large γ and for γ close to one. While splitting partition functions into volume and surface terms is not a new idea in the statistical physics literature (for example, Section 5.7.1 of [90] uses it to derive an explicit expression for the infinite volume pressure of the Ising model on $\mathbb{Z}^d$ with large magnetic field), we are the first to bring this approach into the computer science literature. We are hopeful it will be useful beyond its specific applications presented here.

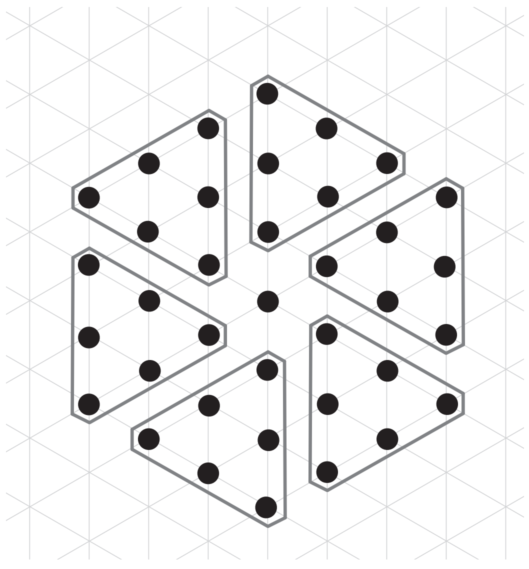
10.1 Preliminaries

10.1.1 Terminology and Results for Homogeneous Amoebot Systems

We first recall the relevant terminology and results from our work on compression (Chapter 8, [32]). A particle system *arrangement* is the set of nodes of the triangular lattice G_Δ occupied by particles. Two arrangements are equivalent if they are translations of each other; we define a particle system *configuration* to be an equivalence class of arrangements. An *edge* of a configuration is an edge of G_Δ where both endpoints are occupied by particles. A configuration is *connected* if for any two particles in the system, there is a path of such edges between them. A configuration has a *hole* if there is a maximal, finite, connected component of unoccupied nodes in G_Δ.

As we justify with Lemma 10.2.1, our analysis will focus on connected, hole-free configurations. The *boundary* of such a configuration σ is the closed walk $\mathcal{P}$ on edges of σ that encloses all particles of σ and no unoccupied nodes of G_Δ. The *perimeter* $p(\sigma)$ of configuration σ is the length of this walk, also denoted $|\mathcal{P}|$. The following bounds the number of configurations with a given perimeter.

Lemma 10.1.1 (Lemma 8.3.4)**.** *For any $\nu > 2 + \sqrt{2}$, there is an integer $n_1(\nu)$ such*


(a)

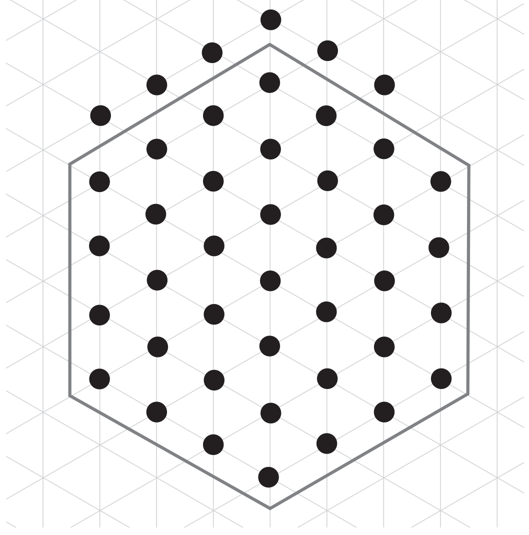

(b)

Figure 57. Bounding the Minimum Perimeter of a System Configuration. (a) The regular hexagon with side length $\ell = 3$, has $3\ell^2 + 3\ell + 1 = 37$ total particles. (b) A configuration with $\ell = 3$ and $k = 6$, totaling $n = 3\ell^2 + 3\ell + 1 + k = 43$ particles. It has perimeter $20 < 2\sqrt{3}\sqrt{n} \approx 22.716$.

that for all $n \geq n_1(\nu)$, the number of connected, hole-free particle system configurations with n particles and perimeter k is at most ν^k.

Let $p_{min}(n)$ be the minimum possible perimeter for a configuration of n particles; it is easy to see that $p_{min}(n) = \Theta(\sqrt{n})$. Recall that for any $\alpha > 1$, a configuration of n particles is said to be *α-compressed* if $p(\sigma) \leq \alpha \cdot p_{min}(n)$ (Definition 8.1.2). The following lemma establishes a concrete upper bound on $p_{min}(n)$.

Lemma 10.1.2. *For any $n \geq 1$, there is a connected, hole-free particle system configuration of n particles with perimeter at most $2\sqrt{3}\sqrt{n}$. That is, $p_{min}(n) \leq 2\sqrt{3}\sqrt{n}$.*

Proof. The lemma can easily be verified for $n \leq 6$. For $n \geq 7$, we begin with the case where $n = 3\ell^2 + 3\ell + 1$ for some integer $\ell \geq 1$. A regular hexagon with side length ℓ can be decomposed into a single center node and six triangles each with

$\ell(\ell+1)/2$ particles, totalling $3\ell^2+3\ell+1$ particles (see Figure 57a). Such a hexagon has perimeter 6ℓ. We see that:

$$p_{min}(3\ell^2+3\ell+1) \leq 6\ell \leq 2\sqrt{3}\sqrt{3\ell(\ell+1)} \leq 2\sqrt{3}\sqrt{n-1} \leq 2\sqrt{3}\sqrt{n}.$$

Now consider $n = 3\ell^2+3\ell+1+k$, where integers ℓ and k satisfy $k \in [1, 6\ell+6)$. As $(3\ell^2+3\ell+1)+6\ell+6 = 3(\ell+1)^2+3(\ell+1)+1$, this covers all possible values of n. Construct a configuration on $n = 3\ell^2+3\ell+1+k$ particles by first constructing a regular hexagon of side length ℓ and then adding the remaining k particles around the outside of this hexagon in a single layer, completing one side before beginning the next (see, e.g., Figure 57b where $\ell = 3$ and $k = 6$). For $k \leq \ell$, the perimeter of this configuration is $6\ell+1$. More generally, the perimeter increases by one when particles begin to be added to a new side of the hexagon, and so for $i \in \{2, \ldots, 6\}$ and $k \in ((i-1)\ell+(i-2), i\ell+(i-1)]$, the perimeter of this configuration is $6\ell+i$. Using $i \leq 6$ and $\ell \geq 1$, we have that for any $i \in \{1, \ldots, 6\}$,

$$\begin{aligned}
p_{min}(3\ell^2+3\ell+1+k) &\leq 6\ell+i \\
&\leq 2\sqrt{3}\sqrt{\left(\sqrt{3}\ell+\frac{i}{2\sqrt{3}}\right)^2} \\
&= 2\sqrt{3}\sqrt{3\ell^2+\frac{i^2}{12}+i} \\
&\leq 2\sqrt{3}\sqrt{3\ell^2+3+i} \\
&\leq 2\sqrt{3}\sqrt{3\ell^2+3\ell+1+i-1} \\
&\leq 2\sqrt{3}\sqrt{3\ell^2+3\ell+1+k} = 2\sqrt{3}\sqrt{n}.
\end{aligned}$$

This concludes our proof. □

10.1.2 Heterogeneous Systems and The Separation Problem

Generalizing previous work on the amoebot model in which all amoebots are homogeneous and indistinguishable, we assume that each amoebot A has a fixed *color* $c(A) \in \{c_1, \ldots, c_k\}$ that is visible to itself and its neighbors, where $k \ll n$ is a constant. We extend the definition of *configuration* given in Section 10.1.1 to include both the nodes of G_Δ occupied by amoebots as well as the colors of those amoebots. An edge of configuration σ with endpoints occupied by amoebots A and B is *homogeneous* if $c(A) = c(B)$ and *heterogeneous* otherwise.

We further extend the original model by allowing neighboring amoebots to exchange their positions in a *swap move*. Swap moves have no meaning in homogeneous systems as all amoebots are indistinguishable, but they grant heterogeneous systems flexibility in allowing amoebots trapped in the interior of the system to move freely.[17] These swap moves are not necessary for the correctness of our algorithm or our rigorous analysis, but enable faster convergence in practice.

In this paper, we study heterogeneous systems with $k = 2$ color classes. Our algorithm performs well in practice for larger values of k and we expect our proof techniques would generalize without needing significant new ideas. However, this generalization would be cumbersome; thus, for simplicity, we restrict our attention to systems with colors $\{c_1, c_2\}$. For 2-heterogeneous systems, we can formally define separation with respect to having large monochromatic regions.

Definition 10.1.3. For $\beta > 0$ and $\delta \in (0, 1/2)$, a 2-heterogeneous system configuration σ is said to be (β, δ)*-separated* if there is a subset of amoebots R such that:

[17]In domains where physical swap moves are unrealistic, colors could be treated as in-memory attributes that could be exchanged by neighboring amoebots to simulate a swap move.

1. There are at most $\beta\sqrt{n}$ edges of σ with exactly one endpoint in R;
2. The density of amoebots of color c_1 in R is at least $1 - \delta$; and
3. The density of amoebots of color c_1 not in R is at most δ.

A configuration is *integrated* if no such β, δ exist.

Unpacking this definition, β controls how small a boundary there is between the monochromatic region R and the rest of the system, with smaller β requiring smaller boundaries. The δ parameter expresses the tolerance for having amoebots of the wrong color within the monochromatic region R: small values of δ require stricter separation of the color classes, while larger values of δ allow for more integrated configurations. Notably, R does not need to be connected.

10.2 Algorithms for Separation

10.2.1 The Markov Chain $\mathcal{M}_S$ for Separation

Our Markov chain algorithm achieves separation by biasing particles towards moves that both gain them more neighbors overall and more like-colored neighbors. We use two bias parameters to control this preference: $\lambda > 1$ corresponds to particles favoring having more neighbors, and $\gamma > 1$ corresponds to particles favoring having more neighbors of their own color.

In order to leverage powerful techniques from Markov chain analysis and statistical physics to prove the correctness of our algorithm, we design our algorithm to follow certain invariants. First, assuming the initial system configuration is connected, our algorithm ensures it remains connected; this is necessary because particles are strictly limited to local interactions, so a disconnected particle cannot communicate with or even find the others. Second, our algorithm eventually eliminates all holes in the configuration, and no new holes are ever formed. This is necessary because our proof techniques only apply to hole-free configurations. Third, once all holes have been eliminated, all moves allowed by our algorithm are *reversible*: if a particle moves from node u to an adjacent node v in one step, there is a nonzero probability that it moves back to u in the next step. Finally, the moves allowed by our algorithm suffice to transform any connected, hole-free configuration into any other connected, hole-free configuration.

Our algorithm uses two locally-checkable properties that ensure particles do not disconnect the system or form a hole when moving (our first two invariants). We use the following notation. For a location ℓ — i.e., a node of the triangular lattice G_Δ — let $N_i(\ell)$ denote the particles of color c_i occupying locations adjacent to ℓ. For neighboring locations ℓ and ℓ', let $N_i(\ell \cup \ell')$ denote the set $N_i(\ell) \cup N_i(\ell')$, excluding particles occupying ℓ and ℓ'. When ignoring color, let $N(\ell) = \bigcup_i N_i(\ell)$; define $N(\ell \cup \ell')$ analogously. Let $\mathbb{S} = N(\ell) \cap N(\ell')$ denote the set of particles adjacent to both locations. A particle can move from location ℓ to ℓ' if one of the following are satisfied:

Property 10.2.1. $|\mathbb{S}| \in \{1, 2\}$ *and every particle in* $N(\ell \cup \ell')$ *is connected to exactly one particle in* $\mathbb{S}$ *by a path through* $N(\ell \cup \ell')$.

Property 10.2.2. $|\mathbb{S}| = 0$*, and both* $N(\ell) \setminus \{\ell'\}$ *and* $N(\ell') \setminus \{\ell\}$ *are nonempty and connected.*

Note that these properties do not need to be verified for swap moves, since swap moves do not change the set of occupied locations and thus cannot disconnect the system or create a hole.

We now define the Markov chain $\mathcal{M}_S$ for separation. The state space Ω of $\mathcal{M}_S$ is the set of all connected heterogeneous particle system configurations of n contracted particles, and Algorithm 16 defines its transition probabilities. We note that $\mathcal{M}_S$, a centralized Markov chain, can be directly translated to a local, distributed, algorithm $\mathcal{A}_S$ that can be run by each particle independently and concurrently to achieve the same system behavior. This translation is much the same as for previous algorithms developed using the stochastic approach; we refer the interested reader to Sections 8.2.2 and 9.2.2 for details. Importantly, this translation is only possible because all probability calculations and property checks in $\mathcal{M}_S$ use strictly local information available to the particles involved. Simulations of $\mathcal{M}_S$ can be found in Section 10.2.3.

Algorithm 16 Markov Chain $\mathcal{M}_S$ for Separation and Integration

```
    From any connected configuration $\sigma_0$ of $n$ contracted particles, repeat:
 1: Choose a particle $P$ uniformly at random; let $c_i$ be its color and $\ell$ its location.
 2: Choose a neighboring location $\ell'$ and $q \in (0,1)$ each uniformly at random.
 3: if $\ell'$ is unoccupied then
 4:     $P$ expands to occupy both $\ell$ and $\ell'$.
 5:     Let $e = |N(\ell)|$ be the number of neighbors $P$ had when it was contracted at $\ell$.
 6:     Let $e' = |N(\ell')|$ be the number of neighbors $P$ would have if it contracts to $\ell'$.
 7:     Let $e_i = |N_i(\ell)|$ and $e'_i = |N_i(\ell')|$ be defined analogously for neighbors of color $c_i$.
 8:     if $(i)$ $e \neq 5$, $(ii)$ $\ell$ and $\ell'$ satisfy Property 10.2.1 or 10.2.2, and $(iii)$ $q < \lambda^{e'-e} \cdot \gamma^{e'_i - e_i}$ then
 9:         $P$ contracts to $\ell'$.
10:     else $P$ contracts back to $\ell$.
11: else if $\ell'$ is occupied by particle $Q$ of color $c_j$ then
12:     if $q < \gamma^{|N_i(\ell')\setminus\{P\}|-|N_i(\ell)|+|N_j(\ell)\setminus\{Q\}|-|N_j(\ell')|}$ then $P$ and $Q$ perform a swap move.
```

10.2.2 The Stationary Distribution of Markov Chain $\mathcal{M}_S$

In this section, we prove that Markov chain $\mathcal{M}_S$ maintains the four invariants described previously and then characterize its stationary distribution.

Lemma 10.2.1. *If the particle system is initially connected, it remains connected throughout the execution of $\mathcal{M}_S$. Moreover, $\mathcal{M}_S$ eventually eliminates any holes in the initial configuration, after which no holes are ever introduced again.*

Proof. This follows directly from analogous results shown for the Markov chain $\mathcal{M}_C$ for compression (Section 8.2.4). Although the separation and compression algorithms assign different probabilities to particle moves, the set of allowed movements is exactly the same, excluding swap moves. However, swap moves do not change the set of occupied nodes of G_Δ, so they cannot disconnect the system or introduce a hole. Thus, because $\mathcal{M}_C$ keeps the system connected, eventually eliminates all holes, and prohibits new holes from forming, so does $\mathcal{M}_S$. □

Lemma 10.2.2. *Once all holes have been eliminated, every possible particle move is reversible; that is, if there is a positive probability of moving from configuration σ to configuration τ, then there is a positive probability of moving from τ to σ.*

Proof. Suppose that a particle P moves from location ℓ to ℓ'. In the next time step, it is possible for P to be chosen again (Step 1) and for ℓ to be chosen as the position to explore (Step 2). Because Properties 10.2.1 and 10.2.2 are symmetric with respect to ℓ and ℓ', whichever was satisfied in the forward move will also be satisfied in this reverse move. Finally, the probability checked in Condition (*iii*) of Step 9 is always nonzero, so all together there is a nonzero probability that P moves back to ℓ in this reverse move. Swap moves can be shown to be reversible in a similar way. □

Lemma 10.2.3. *Markov chain $\mathcal{M}_S$ is ergodic on the state space of connected, hole-free configurations.*

Proof Sketch. One can show that $\mathcal{M}_S$ is irreducible (i.e., the moves of $\mathcal{M}_S$ suffice to transform any configuration to any other configuration) similarly to the proof of the same fact for compression (see Lemma 8.2.10): it is first shown that any configuration can be reconfigured into a straight line; then, the line can be sorted by the color of the particles; finally, by reversibility (Lemma 10.2.2), the line can be reconfigured into any configuration. Additionally, it is easy to see that $\mathcal{M}_S$ is aperiodic: at each iteration of $\mathcal{M}_S$, there is a nonzero probability that the configuration does not change. Thus, because $\mathcal{M}_S$ is irreducible and aperiodic, we conclude it is ergodic. □

Because $\mathcal{M}_S$ is finite and ergodic, it converges to a unique stationary distribution π that we now characterize. For a configuration σ, recall that $p(\sigma)$ denotes its perimeter and $e(\sigma)$ denotes its number of edges, or nearest-neighbor pairs. Let $a(\sigma)$ be the number of homogeneous edges in σ and let $h(\sigma)$ be the number of heterogeneous edges in σ.

Lemma 10.2.4. *The stationary distribution of $\mathcal{M}_S$ is:*

$$\pi(\sigma) = \begin{cases} (\lambda\gamma)^{-p(\sigma)} \cdot \gamma^{-h(\sigma)}/Z & \text{if } \sigma \text{ is connected and hole-free;} \\ 0 & \text{otherwise.} \end{cases}$$

where $Z = \sum_\sigma (\lambda\gamma)^{-p(\sigma)} \cdot \gamma^{-h(\sigma)}$ is the normalizing constant also known as the partition function.

Proof. By Lemma 10.2.1, when $\mathcal{M}_S$ starts at a connected configuration it eventually reaches and remains in the set of configurations that are connected and hole-free. Thus, disconnected configurations and configurations with holes have zero weight at stationarity. We first verify that $\pi(\sigma) = \lambda^{e(\sigma)} \cdot \gamma^{a(\sigma)}/Z_e$, where $Z_e = \sum_\sigma \lambda^{e(\sigma)} \cdot \gamma^{a(\sigma)}$, is the stationary distribution of $\mathcal{M}_S$ by detailed balance. We then show that this form of π can be rewritten as in the lemma.

Consider any two connected, hole-free configurations σ, τ that differ by one move of some particle from location ℓ in σ to a neighboring location ℓ' in τ. By examining

$\mathcal{M}_S$, we see that the probability of transitioning from σ to τ is:

$$M(\sigma, \tau) = \min\left\{1, \lambda^{|N(\ell')|-|N(\ell)|} \cdot \gamma^{|N_i(\ell')|-|N_i(\ell)|}\right\} /6n.$$

A similar analysis shows:

$$M(\tau, \sigma) = \min\left\{1, \lambda^{|N(\ell)|-|N(\ell')|} \cdot \gamma^{|N_i(\ell)|-|N_i(\ell')|}\right\} /6n.$$

Without loss of generality, suppose $\lambda^{|N(\ell')|-|N(\ell)|} \cdot \gamma^{|N_i(\ell')|-|N_i(\ell)|} < 1$, meaning $M(\sigma, \tau)$ is this value over $6n$ and $M(\tau, \sigma) = 1/6n$. Because the only edges that differ in σ and τ are incident to ℓ or ℓ',

$$\begin{aligned}
\pi(\sigma)M(\sigma, \tau) &= \frac{\lambda^{e(\sigma)} \cdot \gamma^{a(\sigma)}}{Z_e} \cdot \frac{1}{n} \cdot \frac{1}{6} \cdot \lambda^{|N(\ell')|-|N(\ell)|} \cdot \gamma^{|N_i(\ell')|-|N_i(\ell)|} \\
&= \frac{\lambda^{e(\sigma)} \cdot \gamma^{a(\sigma)}}{Z_e} \cdot \frac{1}{n} \cdot \frac{1}{6} \cdot \lambda^{e(\tau)-e(\sigma)} \cdot \gamma^{a(\tau)-a(\sigma)} \\
&= \frac{\lambda^{e(\tau)} \cdot \gamma^{a(\tau)}}{Z_e} \cdot \frac{1}{n} \cdot \frac{1}{6} \cdot 1 = \pi(\tau)M(\tau, \sigma)
\end{aligned}$$

Thus, detailed balance is satisfied for particle moves that are not swaps.

Suppose instead that σ and τ differ by a swap move of particle P with color c_i at location ℓ in σ and particle Q with color c_j at neighboring location ℓ' in σ. This move could occur if P or Q is chosen in Step 1 of $\mathcal{M}_S$, so:

$$M(\sigma, \tau) = \min\left\{1, \gamma^{|N_i(\ell')\setminus\{P\}|-|N_i(\ell)|+|N_j(\ell)\setminus\{Q\}|-|N_j(\ell')|}\right\} /3n.$$

Similarly, because τ has P at location ℓ' and Q at location ℓ, we have:

$$M(\tau, \sigma) = \min\left\{1, \gamma^{|N_i(\ell)\setminus\{P\}|-|N_i(\ell')|+|N_j(\ell')\setminus\{Q\}|-|N_j(\ell)|}\right\} /3n.$$

Without loss of generality, suppose that $\gamma^{|N_i(\ell')\setminus\{P\}|-|N_i(\ell)|+|N_j(\ell)\setminus\{Q\}|-|N_j(\ell')|} < 1$, so

$M(\sigma,\tau)$ is this value over $3n$ and $M(\tau,\sigma) = 1/3n$. Then,

$$\begin{aligned}
\pi(\sigma)M(\sigma,\tau) &= \frac{\lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}{Z_e}\cdot\frac{2}{n}\cdot\frac{1}{6}\cdot\gamma^{|N_i(\ell')\backslash\{P\}|-|N_i(\ell)|+|N_j(\ell)\backslash\{Q\}|-|N_j(\ell')|}\\
&= \frac{\lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}{Z_e}\cdot\frac{2}{n}\cdot\frac{1}{6}\cdot\gamma^{(|N_i(\ell')\backslash\{P\}|+|N_j(\ell)\backslash\{Q\}|)-(|N_i(\ell)|+|N_j(\ell')|)}\\
&= \frac{\lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}{Z_e}\cdot\frac{2}{n}\cdot\frac{1}{6}\cdot\gamma^{a(\tau)-a(\sigma)}\\
&= \frac{\lambda^{e(\tau)}\cdot\gamma^{a(\tau)}}{Z_e}\cdot\frac{2}{n}\cdot\frac{1}{6}\cdot 1 = \pi(\tau)M(\tau,\sigma)
\end{aligned}$$

Thus, detailed balance is satisfied in both cases, so we conclude the stationary distribution π over connected, hole-free configurations is given by $\pi(\sigma) = \lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}/Z_e$.

Since every edge of σ is either homogeneous or heterogeneous, we have $e(\sigma) = a(\sigma) + h(\sigma)$. By Lemma 8.1.3, we have $e(\sigma) = 3n - p(\sigma) - 3$ for any connected, hole-free configuration σ, where n is the number of particles in the system. Thus, we can rewrite this unique stationary distribution as follows:

$$\begin{aligned}
\pi(\sigma) &= \frac{\lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}{Z_e}\\
&= \frac{\lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}{\sum_\sigma \lambda^{e(\sigma)}\cdot\gamma^{a(\sigma)}}\\
&= \frac{(\lambda\gamma)^{-3n+3}\cdot(\lambda\gamma)^{e(\sigma)}\cdot\gamma^{a(\sigma)-e(\sigma)}}{(\lambda\gamma)^{-3n+3}\cdot\sum_\sigma(\lambda\gamma)^{e(\sigma)}\cdot\gamma^{a(\sigma)-e(\sigma)}}\\
&= \frac{(\lambda\gamma)^{e(\sigma)-3n+3}\cdot\gamma^{a(\sigma)-e(\sigma)}}{\sum_\sigma(\lambda\gamma)^{e(\sigma)-3n+3}\cdot\gamma^{a(\sigma)-e(\sigma)}}\\
&= \frac{(\lambda\gamma)^{-p(\sigma)}\cdot\gamma^{-h(\sigma)}}{\sum_\sigma(\lambda\gamma)^{-p(\sigma)}\cdot\gamma^{-h(\sigma)}}.
\end{aligned}$$

This concludes our proof. □

The remainder of this paper will be spent analyzing this stationary distribution.

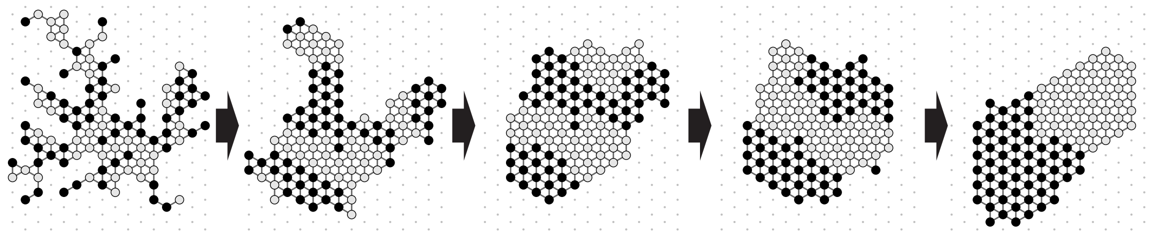


Figure 58. Simulation of Markov Chain $\mathcal{M}_S$ with $\lambda = 4$ and $\gamma = 4$. A 2-heterogeneous particle system of 100 particles starting from an arbitrary initial configuration after (from left to right) 0; 50,000; 1,050,000; 17,050,000; and 68,250,000 iterations.

10.2.3 Simulations

We supplement our rigorous results with simulations that show separation occurs for even better values of λ and γ than our proofs guarantee, indicating that our proven bounds are likely not tight. We simulated $\mathcal{M}_S$ on heterogeneous particle systems with two colors, using 50 particles of each color. Figure 58 shows the progression of $\mathcal{M}_S$ over time with bias parameters $\lambda = 4$ and $\gamma = 4$, the regime in which particles prefer to have more neighbors, especially those of their own color. The simulation ran for nearly 70 million iterations, but much of the system's compression and separation occurs in the first million iterations. Separation still occurs even when swap moves are disallowed, but takes much longer to achieve.

Figure 59 compares the resulting system configurations after running $\mathcal{M}_S$ from the same initial configuration for the same number of iterations, varying only the values of λ and γ. We observe four distinct phases: compressed-separated, compressed-integrated, expanded-separated, and expanded-integrated. We rigorously verify the compressed-separated behavior (i.e., when λ and γ are large), and do the same for the compressed-integrated behavior (i.e., when λ is large and γ is small). We do not give proofs for expanded configurations; in fact, our current definition of separation may not accurately capture what occurs in expanded configurations.

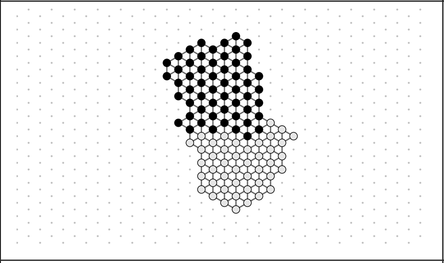
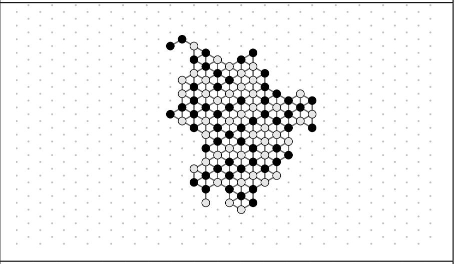
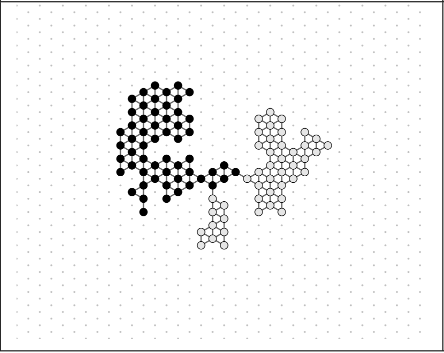
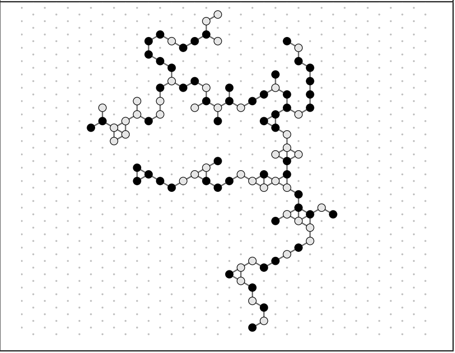
	$\gamma = 5.20$ (Separation)	$\gamma = 0.58$ (Integration)
$\lambda = 5.20$ (Compression)		
$\lambda = 0.58$ (Expansion)		

Figure 59. The Four Phases of Markov Chain $\mathcal{M}_S$. A 2-heterogeneous particle system of 100 particles starting in the leftmost configuration of Figure 58 after 50,000,000 iterations of $\mathcal{M}_S$ for various values of the parameters λ and γ.

10.3 Technical Overview

We consider large λ and want to know for which values of γ separation occurs. Our proof techniques apply to α-compressed configurations (i.e., those whose boundaries have length $p \leq \alpha \cdot p_{min}$, where p_{min} is the smallest possible perimeter for the given number of particles).

In Section 10.4, we show that separation provably occurs among α-compressed configurations, when γ is large enough. We use a technique known as *bridging* that was developed to analyze molecular mixtures called *colloids* [147]. Adapting the bridging approach to our setting requires several new innovations to overcome obstacles such as the irregular shapes of particle system configurations, the non-self-duality of the triangular lattice, the interchangeability between color classes, and other technicalities related to interfaces between particles of different colors. The main result of this section is Theorem 10.4.10, which states that if $\mathcal{P}$ is a boundary of an α-compressed configuration and if α, β, and δ satisfy a technical condition, then configurations with boundary $\mathcal{P}$ and weights proportional to π are (β, δ)-separated with high probability when γ is large enough. We also show that for any $\lambda > 1$ and $\gamma > 4^{5/4} \approx 5.66$, there exist $\beta > 0$ and $\delta \in (0, 1/2)$ such that $\mathcal{M}_S$ achieves (β, δ)-separation among α-compressed configurations at stationarity with high probability (Corollary 10.4.12).

In Section 10.5, we show that there are some values of γ close to one for which separation does not occur among α-compressed configurations. This counterintuitively includes some $\gamma > 1$, where particles have a preference for having like-colored neighbors. We prove this using a probabilistic argument in which we find a set of polynomially many events such that if separation occurs, then at least one of these events occurs. We then show that each event occurs with probability at most $\zeta^{n^{1/2-\varepsilon}}$ for some $\zeta < 1$ and arbitrarily small $\varepsilon > 0$, which via a union bound over the polynomial number of events implies separation is very unlikely. The main result of this section is Theorem 10.5.5, which states that if $\mathcal{P}$ is a boundary of an α-compressed configuration, then if γ and δ satisfy some technical conditions, configurations with boundary $\mathcal{P}$ and weights proportional to π are (β, δ)-separated with probability at most $\zeta^{n^{1/2-\varepsilon}}$ where $\varepsilon > 0$ can be arbitrarily close to zero and $\zeta < 1$ when γ is close to one. We also show that for any $\lambda > 1$ and $\gamma \in (79/81, 81/79)$, there exist $\beta > 0$ and $\delta \in (0, 1/2)$ such that $\mathcal{M}_S$ achieves (β, δ)-separation among α-compressed configurations at stationarity with probability at most $\zeta^{n^{1/2-\varepsilon}}$, where $\varepsilon > 0$ and $\zeta < 1$ (Corollary 10.5.6).

With these results in place, it remains to show that Markov chain $\mathcal{M}_S$ achieves α-compression for the ranges of λ and γ of interest. However, the proofs of compression used in Chapter 8 do not generalize for the heterogeneous setting, so we use the *cluster* *expansion* to overcome this obstacle. We omit the full details of these arguments from this dissertation and instead refer the interested reader to the full paper [31]. We choose instead to emphasize the arguments of separation and integration when α-compression is assumed as we will use these results again in Section 11.2.

10.4 Proof of Separation

We prove that, among configurations with small perimeter, those that are separated are exponentially more likely than those that are not in the stationary distribution of Markov chain $\mathcal{M}_S$, which is also the stationary distribution of our distributed algorithm $\mathcal{A}_S$.

Recall $\Omega_{\mathcal{P}} \subseteq \Omega$ is all configurations with no holes and boundary $\mathcal{P}$. Let $\pi_{\mathcal{P}}$ be the stationary distribution conditioned on being in $\Omega_{\mathcal{P}}$, $\pi_{\mathcal{P}}(\sigma) = \pi(\sigma)/\pi(\Omega_{\mathcal{P}})$. Because all configurations in $\Omega_{\mathcal{P}}$ have the same perimeter, using the definition of π given in Lemma 10.2.4 we see that all terms of the form $(\lambda\gamma)^{-p(\sigma)}$ cancel, yielding $\pi_{\mathcal{P}}(\sigma) = \gamma^{-h(\sigma)}/Z_{\mathcal{P}}$, where $Z_{\mathcal{P}} = \sum_{\sigma \in \Omega_{\mathcal{P}}} \gamma^{-h(\sigma)}$.

Recall a configuration σ is *α-compressed* if its perimeter is at most $\alpha \cdot p_{min}$, where p_{min} is the minimum possible perimeter for the particles in σ. Our main result in this section is that, for all $\mathcal{P}$ that determine α-compressed configurations, non-separated configurations have exponentially small weight according to $\pi_{\mathcal{P}}$.

We formally define separation in terms of the existence of a monochromatic region R as follows. If R is some subset of the particles in a configuration σ, then we say that $bd_{int}(R)$ is all edges of G_Δ with both endpoints occupied by particles in σ and exactly one endpoint in R. For later use, we also define $bd_{out}(R)$ is all edges of G_Δ where one point is occupied by a particle in R and the other endpoint is unoccupied and $bd(R) = bd_{int}(R) \cup bd_{out}(R)$. The following is the definition of separation we will use throughout this section; it is equivalent to Definition 10.1.3 but stated in a more formal way.

Definition 10.4.1. For $\beta > 0$ and $\delta \in (0, 1/2)$, a configuration $\sigma \in \Omega_{\mathcal{P}}$ is (β, δ)-*separated* if there is a subset R of particles such that:

1. $|bd_{int}(R)| \leq \beta\sqrt{n}$;
2. The density of particles of color c_1 in R is at least $1 - \delta$; and
3. The density of particles of color c_1 not in R is at most δ.

Here, β is a measure of how small the boundary between R and its complement $\overline{R}$ must be and δ is a tolerance of having particles of the wrong color within R. We note that requiring $|bd_{int}(R)| \leq \beta\sqrt{n}$ is equivalent to having at most $\beta\sqrt{n}$ edges with one endpoint in R and one endpoint in $\overline{R}$. We note that this definition is symmetric with respect to the role played by c_1 in R and the role played by c_2 in $\overline{R}$. R does not need to be connected or hole-free.

We let $S_{\beta,\delta} \subseteq \Omega_{\mathcal{P}}$ be the configurations in $\Omega_{\mathcal{P}}$ that are (β, δ)-separated for some $\beta > 0$ and $\delta < 1/2$. We prove (Theorem 10.4.10) that for γ sufficiently large, as long as $\mathcal{P}$ is α-compressed, $\beta > 2\alpha\sqrt{3}$, and $\delta < 1/2$, with all but exponentially small probability a sample drawn from $\pi_{\mathcal{P}}$ is in $S_{\beta,\delta}$:

$$\pi_{\mathcal{P}}(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) \leq \zeta^{\sqrt{n}},$$

where $\zeta < 1$ is a constant. In the remainder of this section, we prove this result.

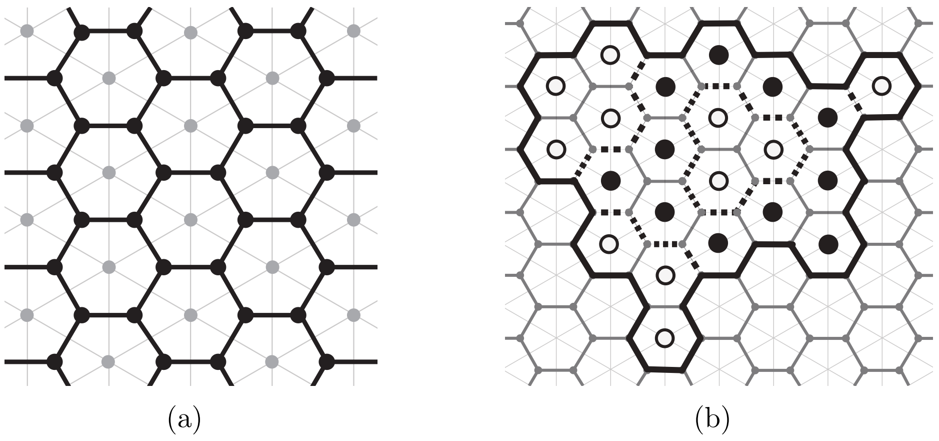


Figure 60. Lattice Duality and Contours. (a) The duality between the triangular lattice G_Δ (gray) and the hexagonal lattice G_{hex} (black). (b) A configuration σ with 11 black particles and 11 white particles. The boundary contour in the dual lattice G_{hex} is thick and black, while all four heterogeneous contours of σ are shown by dashed lines.

10.4.1 Lattice Duality and Contours

We begin with some background on lattice duality that will simplify our proofs in the remainder of this section. The dual to the triangular lattice G_Δ, obtained by creating a new vertex in every face of G_Δ and connecting two of these vertices if their corresponding triangular faces have a common edge, is the hexagonal lattice G_{hex}; see Figure 60a. There is a bijection between edges of G_Δ and edges of G_{hex}, associating an edge of G_Δ with the unique edge of G_{hex} that crosses it and vice versa. Throughout, we refer to a *contour* as a walk in G_{hex} that never visits the same vertex twice, except possibly to start and end at the same place; these are also known as *self-avoiding walks* or, when starting and ending at the same place, *self-avoiding polygons*.

Each edge $e \in G_{hex}$ crosses a unique edge $f \in G_\Delta$, and we say an e *separates* the two locations connected by f. For a configuration σ, we say $e \in G_{hex}$ is a *boundary edge* if it separates a particle of σ from an unoccupied location, and is a *heterogeneous* *edge* if it separates two particles of different colors. A contour is a *boundary contour* if all of its edges are boundary edges and is a *heterogeneous contour* if all of its edges are heterogeneous. See Figure 60b for an example of a configuration σ with particles of two different colors and its boundary and heterogeneous contours.

For a configuration σ without holes, its boundary $\mathcal{P}$ can be completely described by taking the union of all boundary edges in G_{hex}, which yields a boundary contour in G_{hex} which we will call $\mathcal{P}_{hex}$. Recall from our analysis of the Markov chain $\mathcal{M}_C$ for compression that if $|\mathcal{P}| = k$, then $|\mathcal{P}_{hex}| = 2k + 6$ (Lemma 8.3.3). For a configuration with no holes and boundary $\mathcal{P}$ or boundary contour $\mathcal{P}_{hex}$, we can completely describe the colors of its particles (up to swapping the colors) by giving all heterogeneous edges. Because there are only two colors, the heterogeneous edges in G_{hex} form non-intersecting contours because by parity every vertex of G_{hex} has either zero or two incident heterogeneous edges. Each (maximal) heterogeneous contour either starts and ends at different places on the boundary contour or is a closed loop; we call the former a *crossing contour* and the latter an *isolated contour*. The configuration in Figure 60b has three crossing contours and one isolated contour.

The crossing contours of a configuration σ separate the particles into simply connected components whose boundary particles all have the same color. A *face* of a configuration σ is a maximal simply connected subset F where all particles in F incident on an edge of $bd(F)$ have the same color, which we call the *color* of F. For any face F, its maximality implies all edges in $bd_{int}(F)$ are heterogeneous in σ. A face is *outer* if it includes at least on particle on $\mathcal{P}$.

10.4.2 Bridging Systems

Let (B, I) be a collection of contours in G_{hex} within a face F, where B contains *bridge contours* connecting each isolated contour in set I (a subset of the isolated contours within F) to the boundary of F. For a given (B, I), we say particle P is *bridged* in face F if there exists a path through particles of the same color as P to $bd(F)$ or to a bridged isolated contour in I. A particle is *unbridged* if such a path does not exist. We say that (B, I) is a *δ-bridge system* for face F if:

1. $|B| \leq |I|(1-\delta)/2\delta$, where $|B|$ is the total number of edges in all the bridge contours in B and $|I|$ is the total number of edges in all the bridged isolated contours in I.
2. The number of unbridged particles in F is $\leq \delta|F|$, where $|F|$ is the number of particles in F.

Note the δ in this definition is the same δ as in the set $S_{\beta,\delta}$ that we are trying to show has exponentially small weight. We now show how to find a δ-bridge system for any face F.

Lemma 10.4.2. *For any face F, there exists a δ-bridge system for F.*

Proof. Figure 61 gives one example of a face F and a δ-bridge constructed for F. W.l.o.g., suppose F is of color c_1. If F has only one particle, then $(\emptyset, \emptyset)$ is a δ-bridge system for F. We now suppose F has more than one particle and there exists a δ-bridge system for all regions with a smaller number of particles than F. We will iteratively construct a δ-bridge system (B, I) for F.

To start, let $(B, I) = (\emptyset, \emptyset)$, which satisfies $|B| \leq |I|(1-\delta)/2\delta$. Let $u(F)$ be the unbridged particles for (B, I) in F. If $|u(F)| \leq \delta|F|$, where $|F|$ is the number of

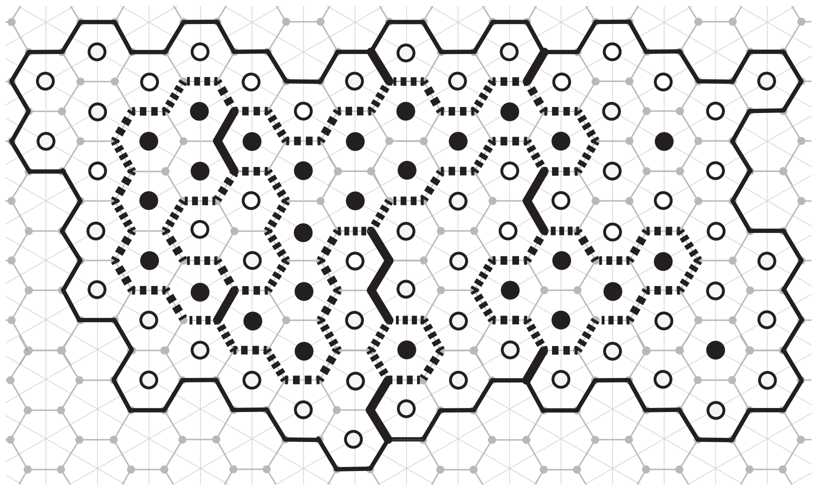


Figure 61. Bridging Systems. A face F and one potential δ-bridge (B, I) for F, where B consists of thick black edges and I consists of all dashed edges. The boundary of F in G_{hex} is shown with thin black lines.

particles in face F, then (B, I) is a valid δ-bridge system for F. If not, we give a procedure for adding to (B, I) that reduces the number of unbridged particles in F and maintains two invariants: (i) $|B| \leq |I|(1-\delta)/2\delta$ and (ii) for any $\mathcal{I} \in I$ not surrounded by another contour in I, the face $F_{\mathcal{I}}$ consisting of all particles inside $\mathcal{I}$ contains at most $\delta|F_{\mathcal{I}}|$ unbridged particles. Both invariants are true for initial configuration $(\emptyset, \emptyset)$. Repeating this process until $u(F) \leq \delta|F|$ gives a valid δ-bridge for F.

Suppose we are given a bridge system (B, I) for F that satisfies both invariants but leaves $u > \delta|F|$ unbridged particles. Let F_{ext} be the particles in F that are not inside any bridged isolated contours in (B, I). We will consider contours $\mathcal{V}$ in G_{hex} that stretch vertically across F, from one part of its boundary to another, consisting of alternating down-left and down-right edges. We call such contours *vertical contours*. We include in set $\mathbb{V}_F$ all (infinite) vertical contours that contain at least one edge inside F_{ext}; we will only be interested in their intersection with F_{ext}, which need not be contiguous. For any $\mathcal{V} \in \mathbb{V}_F$, let $\mathcal{V} \cap F_{ext}$ be all particles in F_{ext} directly right of $\mathcal{V}$ and let $\mathcal{V} \cap u(F_{ext})$ be the unbridged ones. Because $u(F) > \delta|F|$, by applying

Invariant (ii) we conclude that $u(F_{ext}) > \delta|F_{ext}|$. It follows that there exists $\overline{\mathcal{V}} \in \mathbb{V}_F$ such that $|\mathcal{V} \cap u(F_{ext})| > \delta|\mathcal{V} \cap F_{ext}|$.

Any particle $P \in \overline{\mathcal{V}} \cap u(F_{ext})$ must be surrounded by an unbridged isolated contour, as otherwise it would have a monochromatic path to the boundary of F; if there are multiple isolated contours surrounding P, one must be the *outermost*, encircling all the others. Enumerate all outermost isolated contours surrounding particles in $u(F_{ext}) \cap \overline{\mathcal{V}}$ as $\mathcal{I}_j$ for $j = 1, \ldots, k$. Let F_j be the face surrounded by $\mathcal{I}_j$, which is of color c_2. By our induction hypothesis, because $|F| > |F_j|$ there exists a δ-bridge system (B_j, I_j) for F_j. We add to bridge system (B, I) for F the set of bridges $\bigcup_j B_j$ and the set of bridged isolated contours $\bigcup_j I_j$. Furthermore, we add to B all the segments of $\overline{\mathcal{V}}$ that are left of bridged particles in $\overline{\mathcal{V}} \cap F_{ext}$, a set we call B_0, and we add to I all $\mathcal{I}_j$. Because the number of particles that are newly bridged by this construction is at least $|u(F) \cap \overline{\mathcal{V}}|$, we have reduced the number of unbridged particles in F. It only remains to show that this new bridge system satisfies the necessary invariants.

To see that (B, I) satisfies Invariant (ii), note that the only new contours $\mathcal{I} \in I$ not surrounded by other contours in I are the $\mathcal{I}_j$. All particles that were bridged in any $F_{\mathcal{I}_j} = F_j$ are now bridged in F, since both the boundary of F_j and the bridged contours in I_j are now bridged contours in I. Because (B_j, I_j) is a valid δ-bridge system for $F_{\mathcal{I}_j} = F_j$, F_j contains at most $\delta|F_j|$ unbridged particles, as desired.

It remains to show that (B, I) satisfies Invariant (i). Because (B_j, I_j) is a δ-bridge for F_j, $|B_j| \leq |I_j|(1-\delta)/2\delta$ for all j. Next, we see that $\sum_j |\mathcal{I}_j| \geq 4 \cdot |u(F_{ext}) \cap \mathcal{V}|$, as the $\mathcal{I}_j$ collectively contain at least two contour edges left of and two contour edges right of each particle in $u(F_{ext}) \cap \overline{\mathcal{V}}$. Because $\overline{\mathcal{V}}$ satisfies $|\overline{\mathcal{V}} \cap u(F_{ext})| > \delta|\overline{\mathcal{V}} \cap F_{ext}|$, then $\sum_j |\mathcal{I}_j| \geq 4\delta|\overline{\mathcal{V}} \cap F_{ext}|$. Bridge B_0 added to B contains two contour edges for each bridged particle in $F_{ext} \cap \overline{\mathcal{V}}$ and at most a $(1-\delta)$-fraction of the particles in $F_{ext} \cap \overline{\mathcal{V}}$

are bridged, so $|B_0|/2 \leq (1-\delta)|\overline{\mathcal{V}} \cap F_{ext}|$. Combining the previous two equations,

$$\sum_j |\mathcal{I}_j| \geq 4\delta|\overline{\mathcal{V}} \cap F_{ext}| \geq 4\delta\left(\frac{1}{2(1-\delta)}|B_0|\right) = \frac{2\delta}{1-\delta}|B_0|$$

We conclude that the additions B_0 and B_j to B and the additions $\mathcal{I}_j$ and I_j to I satisfy:

$$|B_0| + \sum_{j=1}^{k}|B_j| \leq \frac{1-\delta}{2\delta}\sum_{j=1}^{k}|\mathcal{I}_j| + \frac{1-\delta}{2\delta}\sum_{j=1}^{k}|I_j| = \frac{1-\delta}{2\delta}\sum_{j=1}^{k}(|\mathcal{I}_j| + |I_j|)$$

Thus, Invariant (i) is satisfied.

We have added to (B, I) while maintaining both invariants and reducing the number of unbridged particles in F. We can continue this process until there are at most $\delta|F|$ unbridged particles in F; then, Invariant (i) implies (B, I) is a δ-bridge system for F. □

Lemma 10.4.3. *For each $\sigma \in \Omega_{\mathcal{P}}$ with n particles, there exists a δ-bridge system (B, I) for σ where B contains bridge contours connecting each isolated contour in set I (a subset of σ's isolated contours) to σ's boundary contour or to a crossing contour, such that:*

- *$|B| \leq |I|(1-\delta)/2\delta$, and*
- *The number of unbridged particles in σ is at most δn.*

Proof. The crossing contours of σ partition σ into faces. Construct a δ-bridge system for each of these faces and take their union. □

We now connect δ-bridges to configurations that are (β, δ)-separated.

Lemma 10.4.4. *Let $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ and let (B, I) be the δ-bridge system for σ constructed in Lemma 10.4.3. Let x be the total length of crossing contours in σ and let y be the total length of bridged isolated contours in I. Then $x + y > \beta\sqrt{n}$.*

Proof. Let $\mathcal{F}$ be the set of outermost faces of σ, that is, those faces of σ that contain a particle on σ's perimeter. For each $F \in \mathcal{F}$ of color c_i, if particle $P \in F$ is surrounded by b bridged isolated contours then put P in set R if and only if $i + b \equiv 1 (\text{mod } 2)$. Because of how we have carefully defined R, inspection shows $bd_{int}(R) = x + y$. Using the properties of δ-bridge system (B, I), one can show the density of particles of color c_1 is at least $1 - \delta$ in R and at most δ outside of R. If it were true that $x + y \leq \beta\sqrt{n}$, then σ would be (β, δ)-separated, a contradiction as $\sigma \notin S_{\beta,\delta}$. Thus, it must hold that $x + y > \beta\sqrt{n}$. □

10.4.3 Information Theoretic Argument for Separation

To show the set $\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ of configurations with boundary contour $\mathcal{P}$ that are not (β, δ)-separated has exponentially small weight under distribution $\pi_{\mathcal{P}}$, we will define a map $f = f_3 \circ f_2 \circ f_1$ from this set into $\Omega_{\mathcal{P}}$ and examine how this map changes weights of configurations. If the number of particles of one color is less than or equal to δn, then all configurations in $\Omega_{\mathcal{P}}$ are (β, δ)-separated with $R = \emptyset$ or $\overline{R} = \emptyset$, so we assume each color class has more than δn particles.

For $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$, let (B, I) be the δ-bridge system constructed for σ according to Lemma 10.4.3. Let $f_1(\sigma)$ be the (unique) particle configuration that has the same boundary contour $\mathcal{P}$ as σ and particle P that has color c_i in σ and is surrounded by b bridged isolated contours in I is given color $c_{(i+b)(\text{mod } 2)}$ in $f_1(\sigma)$. We let $Im(f_1(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}))$ be the set of configurations that f_1 maps to.

We define f_2 with domain $Im(f_1(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}))$ to complement all faces of color c_2 that touch the boundary of the configuration (i.e. that include particles on $\mathcal{P}$). The next lemmas explore the composition of these maps f_1 and f_2 as applied to configurations $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$.

Lemma 10.4.5. *For any $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$, $f_2(f_1(\sigma))$ has boundary contour $\mathcal{P}$, all particles adjacent to $\mathcal{P}$ have color c_1, and there are at most δn particles of color c_2.*

Proof. The first two claims follow easily from the definitions of f_1 and f_2. To see that the last claim holds, we note that any particles of color c_2 in $f_2(f_1(\sigma))$ must have been unbridged by the bridge system (B, I) for σ, and there are at most δn such unbridged particles by the definition of a δ-bridge system. □

Lemma 10.4.6. *Let $\tau \in Im((f_2 \circ f_1)(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}))$. The number of configurations $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ with crossing contours of total length x and bridged isolated contours (bridged by a bridging system (B, I) from Lemma 10.4.3) of total length y that have $f_2(f_1(\sigma)) = \tau$ is, for $p = |\mathcal{P}|$ the perimeter of any configuration in $\Omega_{\mathcal{P}}$, at most $3^p 4^{(x+y)\left(\frac{1+3\delta}{4\delta}\right)}$.*

Proof. Any configuration $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ has boundary $\mathcal{P}$ of length p and boundary contour $\mathcal{P}_{hex}$ of length $2p + 6$. One can verify from first principles that $\mathcal{P}_{hex}$ makes p left turns and $p + 6$ right turns when traversed clockwise. Any bridges or crossing contours that meet $\mathcal{P}$ do so at distinct left turns of $\mathcal{P}$. We can mark each left turn of $\mathcal{P}$ as the start of a bridge, the start of a crossing contour, or neither; the number of ways to do so is 3^p.

Next, we can trace out all crossing contours of σ, beginning at the starting points marked along $\mathcal{P}$. In tracing these contours, which do not intersect, at each vertex in G_{hex} we make either a left turn or a right turn. Additionally, each vertex along these contours can either be the beginning of a bridge in B, branching in the opposite direction from the contour, or not. Because x is the total length of σ's crossing contours, the number of valid ways to do this is at most $2^x \times 2^x = 4^x$.

Finally, we trace out the bridges and isolated contours of each face of σ in a depth-first way, beginning at the starting points marked along $\mathcal{P}$ and the crossing contours. Bridges as constructed in Lemma 10.4.3 always move in the vertical direction, so the direction of the next edge of a bridge, if it exists, is known; at each step we only need to know if the bridge continues or if a bridged isolated contour begins. When tracing out isolated contours, just like with heterogeneous crossing contours, there are four choices for the next step: the direction in which the contour continues (two choices) and whether or not a bridge branches off (two choices). Isolated contours end when they reach an already-constructed bridge, and bridges end when they reach a crossing contour, an already-constructed isolated contour, or $\mathcal{P}$. The number of possibilities for this depth-first traversal of the bridges and isolated contours of σ is at most $2^{|B|}4^{|I|} \leq 2^{\frac{1-\delta}{2\delta}y}4^y$.

Altogether, any configuration $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ with crossing contours of total length x and bridged isolated contours of total length y that have $f_2(f_1(\sigma)) = \tau$ can be uniquely identified by marking $\mathcal{P}$, tracing crossing contours, and tracing bridges and bridged isolated contours. The number of valid ways to do this is at most:

$$3^p 4^x 2^{\frac{1-\delta}{2\delta}y} 4^y = 3^p 4^{x+y+\frac{1-\delta}{4\delta}y} \leq 3^p 4^{(x+y)\left(\frac{1+3\delta}{4\delta}\right)}$$

This is an upper bound on the number of preimages of τ under $f_2 \circ f_1$ with correct x and y. □

Any $\tau \in Im((f_2 \circ f_1)(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta})$ will not be in $\Omega_{\mathcal{P}}$ because it has too few particles of color c_2. We will define f_3 such that $f_3(\tau)$ is similar to τ and has the correct number of particles of each color, but we first need the following lemma. Let $\Omega_{\mathcal{P}}^{c_1}$ be the set of configurations with all particles on boundary $\mathcal{P}$ having color c_1, and recall that $w_{\mathcal{P}}(\sigma) = \pi_{\mathcal{P}}(\sigma)/Z_{\mathcal{P}} = \gamma^{-h(\sigma)}$.

Lemma 10.4.7. *For a configuration $\sigma \in \Omega_{\mathcal{P}}^{c_1}$, it is possible to flip the colors of some particles to yield a configuration $g(\sigma) \in \Omega_{\mathcal{P}}$ with $n/2$ particles of each color such that $\gamma^{-|\mathcal{P}|}w(\sigma) \leq w(g(\sigma))$. Furthermore, for any $\tau \in \Omega_{\mathcal{P}}$, there are at most n different $\sigma \in \Omega_{\mathcal{P}}^{c_1}$ such that $f(\sigma) = \tau$.*

Proof. If $\sigma \in \Omega_{\mathcal{P}}$, let $g(\sigma) = \sigma$ and the lemma holds. For $\sigma \in \Omega_{\mathcal{P}}^{c_1} \setminus \Omega_{\mathcal{P}}$, label the particles of σ in order from left to right and, within each column, from top to bottom. Flip the colors of particles in this order, until there are the correct number of particles of each color. If σ has more than $n/2$ particles of color c_i, then after flipping the colors of all particles it has fewer than $n/2$ particles of color c_i. At some intermediate step, there must have been exactly $n/2$ particles of color c_i and the configuration is in $\Omega_{\mathcal{P}}$, as desired. We let the first such configuration be $g(\sigma)$.

Because we flip all particles in one column before flipping any particles in the next column, all heterogeneous edges introduced by this process are in two adjacent columns. If h is the total height of σ — the vertical difference between its lowest and highest particles — then the number of adjacencies between particles whose color was flipped and particles whose color was not flipped is at most $2h$. This is an upper bound on the number of heterogeneous edges introduced by the flips. The height of a configuration is less than half its perimeter, so we conclude the number of new heterogeneous edges is at most $2h(\sigma) \leq p(\sigma) = |\mathcal{P}|$. Thus, $w_{\mathcal{P}}(\sigma) \leq w_{\mathcal{P}}(g(\sigma))$.

Given $\tau \in \Omega_{\mathcal{P}}$ and a number $k \in \{0, 1, \ldots, n-1\}$, complementing the colors of the first k elements (according to the canonical ordering from above) of τ yields a configuration that maps to τ under g. These n configurations, which may or may not be in $\Omega_{\mathcal{P}}^{c_1}$, are the only ones that could map to τ under g. $\square$

Since $Im((f_2 \circ f_1)(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta})) \subseteq \Omega_{\mathcal{P}}^{c_1}$, Lemma 10.4.7 immediately implies the following.

Lemma 10.4.8. *For a configuration $\tau \in Im((f_2 \circ f_1)(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}))$, it is possible to flip the colors of some particles to yield a configuration $f_3(\tau)$ with the correct number of particles of each color such that at most $|\mathcal{P}|$ additional heterogeneous edges are introduced. Furthermore, for any $\nu \in \Omega_{\mathcal{P}}$, there are at most n different $\tau \in Im((f_2 \circ f_1)(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}))$ such that $f_3(\tau) = \nu$.*

Let $f = f_3 \circ f_2 \circ f_1$ be a map from $\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ to $\Omega_{\mathcal{P}}$. For $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$, let x be the total length of crossing heterogeneous contours in σ and y be the total length of all isolated contours in σ that are bridged when constructing a δ-bridge system according to the process of Lemma 10.4.2.

Lemma 10.4.9. *For $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ where $\mathcal{P}$ is α-compressed, $h(\sigma) - h(f(\sigma)) \geq (x+y)\left(1 - \frac{2\alpha\sqrt{3}}{\beta}\right)$.*

Proof. Configuration $f_1(\sigma)$ has y fewer heterogeneous edges than σ, and configuration $f_2(f_1(\sigma))$ has x fewer heterogeneous edges than $f_1(\sigma)$. When going from $f_2(f_1(\sigma))$ to $f(\sigma) = f_3(f_2(f_1(\sigma)))$, at most $2\alpha\sqrt{3}\sqrt{n}$ heterogeneous edges are added (Lemma 10.4.8). Using Lemma 10.4.4, we conclude that:

$$\begin{aligned} h(\sigma) - h(f(\sigma)) &\geq x + y - |\mathcal{P}| \geq x + y - 2\alpha\sqrt{3}\sqrt{n} \\ &\geq x + y - 2\alpha\sqrt{3}\left(\frac{x+y}{\beta}\right) \geq (x+y)\left(1 - \frac{2\alpha\sqrt{3}}{\beta}\right) \end{aligned}$$

This proves the claim. □

We are now ready to prove our main result. Recall that for a fixed boundary $\mathcal{P}$, the probability distribution $\pi_{\mathcal{P}}$ is over colored particle configurations with this boundary where $\pi_{\mathcal{P}}(\sigma)$ is proportional to $\gamma^{-h(\sigma)}$.

Theorem 10.4.10. *Let $\mathcal{P}$ be the boundary of n particles with $|\mathcal{P}| \leq \alpha p_{min}$. For any $\beta > 2\sqrt{3}\alpha$ and any $\delta < 1/2$, if γ is large enough that*

$$3^{\frac{2\alpha\sqrt{3}}{\beta}} 4^{\frac{1+3\delta}{4\delta}} \gamma^{-1+\frac{2\alpha\sqrt{3}}{\beta}} < 1,$$

then for sufficiently large n the probability that a configuration drawn from $\pi_{\mathcal{P}}$ is not (β, δ)-separated is exponentially small:

$$\pi_{\mathcal{P}}(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) < \zeta^{\sqrt{n}}$$

where $\zeta < 1$.

Proof. For any $\nu \in \Omega_{\mathcal{P}}$, we count the number of configurations in $\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ such that $f(\sigma) = \nu$. By Lemma 10.4.6, the number of such preimages with crossing contours of total length x and bridged isolated contours of total length y is at most $n3^p 4^{(x+y)(1+3\delta)/4\delta}$, where $p = |\mathcal{P}|$. As $p < \alpha p_{min} < 2\alpha\sqrt{3}\sqrt{n}$, by Lemma 10.4.4 we have $p < 2\alpha\sqrt{3}(x+y)/\beta$. We can rewrite the number of preimages in $f^{-1}(\nu)$ with given values of x and y as:

$$n3^p 4^{(x+y)\left(\frac{1+3\delta}{4\delta}\right)} < n3^{2\alpha\sqrt{3}\left(\frac{x+y}{\beta}\right)} 4^{(x+y)\left(\frac{1+3\delta}{4\delta}\right)} = n\left(3^{\frac{2\alpha\sqrt{3}}{\beta}} 4^{\frac{1+3\delta}{4\delta}}\right)^{x+y}$$

We now sum over all possible values of $x+y$. For each possible value of $x+y$, there are at most $x+y+1$ ways in which each of x and y could have contributed to this sum. By Lemma 10.4.4, $x+y > \beta\sqrt{n}$, and because the edges counted in $x+y$ are a subset of all edges in the configuration, $x+y < 3n$. We conclude, for $z = x+y$,

$$|f^{-1}(\nu)| \leq n \sum_{z=\lceil\beta\sqrt{n}\rceil}^{3n} (z+1)\left(3^{\frac{2\alpha\sqrt{3}}{\beta}} 4^{\frac{1+3\delta}{4\delta}}\right)^z$$

Finally, we see that for any $\nu \in \Omega_{\mathcal{P}}$, using Lemma 10.4.9,

$$\begin{aligned}\frac{\sum_{\sigma\in f^{-1}(\nu)}\pi_{\mathcal{P}}(\sigma)}{\pi_{\mathcal{P}}(\nu)} &= \sum_{\sigma\in f^{-1}(\nu)}\left(\frac{1}{\gamma}\right)^{h(\sigma)-h(f(\sigma))}\\ &\le n\sum_{z=\lceil\beta\sqrt{n}\rceil}^{3n}(z+1)\left(\frac{1}{\gamma}\right)^{z\left(1-\frac{2\alpha\sqrt{3}}{\beta}\right)}\\ &\le n\sum_{z=\lceil\beta\sqrt{n}\rceil}^{3n}(z+1)\left(3^{\frac{2\alpha\sqrt{3}}{\beta}}4^{\frac{1+3\delta}{4\delta}}\gamma^{-1+\frac{2\alpha\sqrt{3}}{\beta}}\right)^{z}\end{aligned}$$

This sum is exponentially small whenever the number of particles n is sufficiently large and the base of the exponent satisfies $3^{\frac{2\alpha\sqrt{3}}{\beta}}4^{\frac{1+3\delta}{4\delta}}\gamma^{-1+\frac{2\alpha\sqrt{3}}{\beta}} < 1$. Whenever $\beta > 2\alpha\sqrt{3}$, $\delta < 1/2$, and γ is large enough this is true, so we can find a constant $\zeta < 1$ such that for sufficiently large n,

$$\frac{\sum_{\sigma\in f^{-1}(\nu)}\pi_{\mathcal{P}}(\sigma)}{\pi_{\mathcal{P}}(\nu)} < \zeta^{\lceil\beta\sqrt{n}\rceil} < \zeta^{\sqrt{n}}$$

Because each $\sigma \in \Omega_{\mathcal{P}} \setminus S_{\beta,\delta}$ has some image $f(\sigma) \in \Omega_{\mathcal{P}}$, we use this fact to see that:

$$\pi_{\mathcal{P}}(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) = \sum_{\sigma\in\Omega_{\mathcal{P}}\setminus S_{\beta,\delta}}\pi_{\mathcal{P}}(\sigma) \le \sum_{\nu\in\Omega_{\mathcal{P}}}\sum_{\sigma\in f^{-1}(\nu)}\pi_{\mathcal{P}}(\sigma) \le \sum_{\nu\in\Omega_{\mathcal{P}}}\pi_{\mathcal{P}}(\nu)\zeta^{\sqrt{n}} = \zeta^{\sqrt{n}}$$

We conclude that when n is sufficiently large, $\beta > 2\alpha\sqrt{3}$, $\delta < 1/2$, and γ is large enough, the probability a particle configuration drawn from $\pi_{\mathcal{P}}$ is not (β,δ)-separated is exponentially small. $\square$

We now extend this result about the occurrence of separation when fixing an α-compressed boundary $\mathcal{P}$ to a statement about the occurrence of separation among all α-compressed boundaries. Let π_α be the probability distribution over all configurations that are α-compressed obtained by restricting π to this set, so that $\pi_\alpha(\sigma)$ is proportional to $(\lambda\gamma)^{-p(\sigma)}\gamma^{-h(\sigma)}$. We obtain the following result.

Theorem 10.4.11. *For any $\alpha > 1$, $\beta > 2\sqrt{3}\alpha$, and $\delta < 1/2$, if γ is large enough that*

$$3^{\frac{2\alpha\sqrt{3}}{\beta}} 4^{\frac{1+3\delta}{4\delta}} \gamma^{-1+\frac{2\alpha\sqrt{3}}{\beta}} < 1$$

then for n sufficiently large the probability that a configuration drawn from π_α is not (β, δ)-separated is exponentially small:

$$\pi_\alpha(\Omega_\alpha \setminus S_{\beta,\delta}) < \zeta^{\sqrt{n}}$$

where $\zeta < 1$.

Proof. This result follows directly from Theorem 10.4.10. Let $\zeta < 1$ be a constant such that for any $\mathcal{P}$ with $|\mathcal{P}| < \alpha p_{min}$, $\pi_{\mathcal{P}}(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) < \zeta^{\sqrt{n}}$. We then see that:

$$\begin{aligned}
\pi_\alpha(\Omega_\alpha \setminus S_{\beta,\delta}) &= \sum_{\mathcal{P}:|\mathcal{P}|<\alpha p_{min}} \pi_\alpha(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) \\
&= \sum_{\mathcal{P}:|\mathcal{P}|<\alpha p_{min}} \pi_\alpha(\Omega_{\mathcal{P}}) \pi_{\mathcal{P}}(\Omega_{\mathcal{P}} \setminus S_{\beta,\delta}) \\
&\leq \sum_{\mathcal{P}:|\mathcal{P}|<\alpha p_{min}} \pi_\alpha(\Omega_{\mathcal{P}}) \zeta^{\sqrt{n}} = \zeta^{\sqrt{n}}.
\end{aligned}$$

This proves the claim. □

Corollary 10.4.12. *For Markov chain $\mathcal{M}_S$ with parameters $\lambda > 1$ and $\gamma > 4^{5/4} \approx 5.66$ and any $\alpha > 1$, there exist constants β and δ such that for sufficiently large n, $\mathcal{M}_S$ provably accomplishes (β, δ)-separation among α-compressed configurations at stationarity with high probability.*

Proof. Among α-compressed configurations, Theorem 10.4.11 shows that if β, δ, and γ satisfy $3^{\frac{2\alpha\sqrt{3}}{\beta}} 4^{\frac{1+3\delta}{4\delta}} \gamma^{-1+\frac{2\alpha\sqrt{3}}{\beta}} < 1$, then the probability that a configuration sampled from π_α is not (β, δ)-separated is at most $\zeta^{\sqrt{n}}$ for a constant $\zeta < 1$. For $\gamma > 4^{5/4}$, one can always find a $\delta < 1/2$ such that $\gamma > 4^{(1+3\delta)/4\delta}$. For $\alpha > 1$, one can always find a constant $\beta > 2\sqrt{3}\alpha$ such that the exponent $2\alpha\sqrt{3}/\beta$ is sufficiently close to zero that the above expression is less than one, as desired. We conclude that when n is sufficiently large, the probability (β, δ)-separation occurs at stationarity is at least $1 - \zeta^{\sqrt{n}}$ for some $\zeta < 1$. $\square$

This concludes our proofs that $\mathcal{M}_S$ accomplishes separation.

10.5 Proof of Integration

We next prove that, among configurations with small perimeter, then for γ close to 1 there exist $\beta > 0$ and $\delta \in (0, 1/2)$ such that (β, δ)-separation does not occur at stationarity when n is sufficiently large.

Recall from Definition 10.4.1 that a two-color configuration is (β, δ)-separated if there is a set R of particles with small boundary (i.e., $|bd_{int}(R)| < \beta\sqrt{n}$) such that the density of particles of color c_1 is at least $1 - \delta$ in R and at most δ outside of R. In this section, we assume for the sake of simplicity that there are n total particles with $n/2$ of each color, though we expect our results to generalize with little effort whenever there are a constant fraction of particles of each color. Note that if there is not a constant fraction of particles of each color, the configuration is always (β, δ)-separated for any β and δ by treating the set of all particles as R.

Consider any configuration σ that is α-compressed and (β, δ)-separated, and let R be a set witnessing this separation; i.e., $|bd_{int}(R)| < \beta\sqrt{n}$, at most a δ-fraction of particles in R are color c_2, and at most a δ-fraction of the particles not in R are color c_1. Recall $\overline{R}$ is all particles not in R. We will use the following lemma bounding the size of R.

Lemma 10.5.1. *For a set R witnessing (β, δ)-separation in a configuration σ of n particles with $n/2$ of each color,*

$$\frac{1-2\delta}{1-\delta} \cdot \frac{n}{2} < |R| \leq \frac{1}{1-\delta} \cdot \frac{n}{2}$$

Proof. Suppose $|R| > \frac{n}{2-2\delta}$. By the definition of (β, δ)-separation, at most a δ-fraction of these particles can be of color c_2, which means at most $\delta n/(2-2\delta)$ of them. The remaining particles of R must be of color c_1, so the number of particles of color c_1 in R satisfies

$$|R| - \frac{\delta n}{2-2\delta} > \frac{n}{2-2\delta} - \frac{\delta n}{2-2\delta} = \frac{(1-\delta)n}{2(1-\delta)} = \frac{n}{2}$$

So R has strictly more than $n/2$ particles of color c_1, contradicting the fact that there are only $n/2$ particles of color c_1 in the entire configuration. We conclude that $|R| \leq \frac{1}{1-\delta} \cdot \frac{n}{2}$. By the symmetry between R and $\overline{R}$ in the definition of separation, $|\overline{R}| \leq \frac{1}{1-\delta} \cdot \frac{n}{2}$ and thus:

$$|R| \geq n - \frac{1}{1-\delta} \cdot \frac{n}{2} = \frac{1-2\delta}{1-\delta} \cdot \frac{n}{2}$$

□

Recall that $bd_{out}(R)$ is the set of edges of G_Δ with one endpoint in R and the other endpoint unoccupied and $bd(R) = bd_{out}(R) \cup bd_{int}(R)$. Furthermore, because we assume our configuration of interest is α-compressed,

$$|bd(R)| = |bd_{out}(R)| + |bd_{int}(R)| \leq \alpha p_{min} + \beta\sqrt{n} \leq (2\sqrt{3}\alpha + \beta)\sqrt{n},$$

which holds since $p_{min} \leq 2\sqrt{3}\sqrt{n}$ by Lemma 10.1.2.

To show that (β, δ)-separation does not occur, we will define polynomially many events such that if (β, δ)-separation occurs then at least one of these events occurs; we then show each of these events has a superpolynomially small probability of occurring.

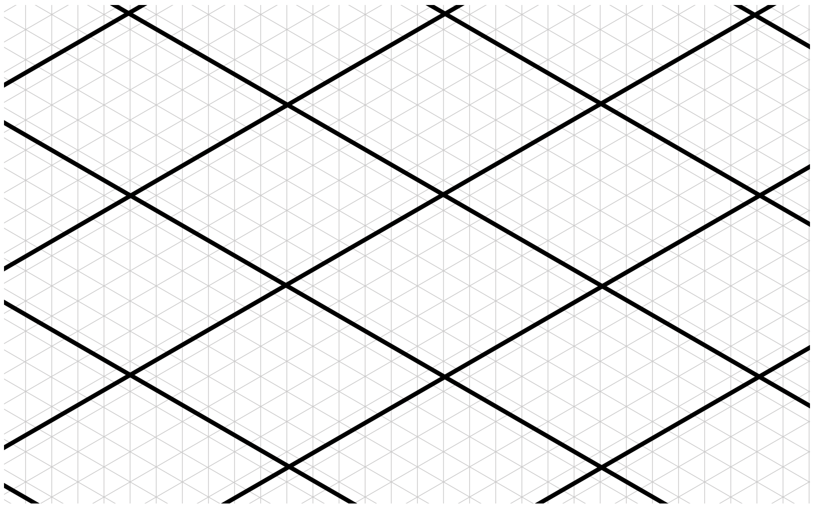


Figure 62. Diamond Partition of G_Δ. In this example, the diamonds have side length 6 and each contain $6^2 = 36$ total vertices.

To this end, we consider a partition of the lattice G_Δ into diamonds with side length n^c for some $c < 1/4$, where c is chosen so that n^c is an integer; see Figure 62 for an example of a partition of G_Δ into diamonds with side length six. Each diamond in this partition contains n^{2c} vertices of G_Δ. The events we consider will be that one diamond in the partition is fully occupied by particles and at most $\delta' < 1/2$ of these particles are of color c_2.

Lemma 10.5.2. *Let σ be an α-compressed configuration. If σ is (β, δ)-separated, then for any $\delta' > \delta/(1-2\delta)$, there exists a diamond in our partition that is fully occupied by particles and has at most $\delta' n^{2c}$ particles of color c_2.*

Proof. Let R witness the (β, δ)-separation of σ. Because each diamond in our partition contains n^{2c} vertices of G_Δ, by Lemma 10.5.1 the total number of diamonds intersecting R is at least $|R|/n^{2c} \geq \frac{n^{1-2c}(1-2\delta)}{1-\delta}$. Meanwhile, we see that at most $|bd(R)| \leq (\beta + 2\sqrt{3}\alpha)\sqrt{n}$ diamonds intersect the boundary of R. The number of diamonds in our partition comprised entirely of particles in R (and thus fully occupied by particles) must be at least $n^{1-2c}(1-2\delta)/(2-2\delta) - (\beta + 2\sqrt{3}\alpha)\sqrt{n}$. Suppose to the contrary that each of these diamonds has at least $\delta' > \delta/(1-2\delta)$ particles of color c_2. Then the total number n_2 of particles of color c_2 in R would be at least:

$$\delta' n^{2c}\left(\frac{(1-2\delta)n^{1-2c}}{2-2\delta} - (\beta + 2\sqrt{3}\alpha)\sqrt{n}\right) = \left(\delta'\frac{1-2\delta}{2-2\delta}n - \delta'(2\sqrt{3}\alpha+\beta)n^{1/2+2c}\right)$$

Provided $c < 1/4$, asymptotically the first term above dominates and for n sufficiency large we have that:

$$n_2 > \frac{\delta'(1-2\delta)}{2-2\delta}n = \left(\frac{\delta}{1-2\delta}\right)\frac{1-2\delta}{2-2\delta}n = \delta\left(\frac{1}{2-2\delta}n\right) \geq \delta|R|$$

This shows that greater than a δ-fraction of the particles in R have color c_2, contradicting that σ is (β,δ)-separated with cluster R. We conclude that there must exist a diamond in the partition that is fully occupied by particles in R with at most a δ' fraction of particles of color c_2. □

We will be interested in values of δ that give $\delta' < 1/2$, that is, in $\delta < 1/4$. Let $\mathcal{P}$ be the boundary of an α-compressed configuration. We examine the set $\Omega_{\mathcal{P}}$ of configurations with this boundary. Recall $\pi_{\mathcal{P}}$ is the stationary distribution conditioned on boundary $\mathcal{P}$. For a coloring σ of the particles inside $\mathcal{P}$, we say that the weight of this configuration is $w_{\mathcal{P}}(\sigma) = \gamma^{-h(\sigma)}$. We can then write $\pi_{\mathcal{P}}(\sigma) = \frac{w_{\mathcal{P}}(\sigma)}{Z_{\mathcal{P}}}$, where $Z_{\mathcal{P}} = \sum_{\sigma\in\Omega_{\mathcal{P}}} w_{\mathcal{P}(\sigma)}$.

For any $n^c \times n^c$ diamond $\mathcal{D}$ with every vertex on or inside $\mathcal{P}$ occupied by a particle, we consider the configurations $\sigma \in \Omega_{\mathcal{P}}$ that have fewer than a δ'-fraction of the particles in $\mathcal{D}$ with color c_2. We want to show that the set of all such configurations has exponentially small weight at stationarity.

For such σ, we will break the term $w_{\mathcal{P}}(\sigma)$ up according to contributions within $\mathcal{D}$, contributions between $\mathcal{D}$ and $\overline{\mathcal{D}}$, and contributions within $\overline{\mathcal{D}}$, where $\overline{\mathcal{D}}$ is the set of all particles on or inside $\mathcal{P}$ not in $\mathcal{D}$. There are at most $8n^c + 6$ edges between $\mathcal{D}$ and $\overline{\mathcal{D}}$, so these edges can contribute at most $\max\{1, \gamma^{-8n^c-6}\}$ and at least $\min\{1, \gamma^{-8n^c-6}\}$ to the weight of a configuration, where the values achieved by these maximum and minimums depend on whether $\gamma > 1$ or $\gamma < 1$. Instead of looking at contributions from $\mathcal{D}$ or $\overline{\mathcal{D}}$ for particular configurations, we look at the sum of contributions within these regions over many possible configurations.

For any set Λ of vertices of G_Δ, let Ω_Λ^ℓ be all colorings of vertices in Λ with exactly ℓ particles assigned color c_2; we will consider $\Lambda = \mathcal{D}$ and $\Lambda = \overline{\mathcal{D}}$. For $\sigma \in \Omega_\Lambda^\ell$, we say that $h_\Lambda(\sigma)$ is the number of edges of G_Δ where both endpoints are in Λ and are assigned different colors in σ. The corresponding partition functions are:

$$Z_\Lambda^\ell = \sum_{\sigma \in \Omega_\Lambda^\ell} \gamma^{-h_\Lambda(\sigma)}$$

The following lemma will play an important role. Note that Λ need not be connected or hole-free.

Lemma 10.5.3. *For any* $\Lambda \subseteq V(G_\Delta)$*,*

$$\frac{Z_\Lambda^\ell}{Z_\Lambda^{|\Lambda|/2}} \leq \max\{\gamma^{6(\ell-|\Lambda|/2)}, \gamma^{-6(|\Lambda|/2-\ell)}\}$$

Proof. We first note that, because of the symmetry between colors, $Z_\Lambda^\ell = Z_\Lambda^{|\Lambda|-\ell}$. W.l.o.g., we assume $\ell \geq |\Lambda|/2$. For any $\sigma \in \Omega_\Lambda^\ell$, we can map it to a configuration in $\Omega_\Lambda^{|\Lambda|/2}$ by choosing $\ell - |\Lambda|/2$ of the ℓ particles of color c_2 and changing their color to c_1. There are $\binom{\ell}{\ell-|\Lambda|/2}$ way to do this, and doing so changes the number of heterogeneous edges within Λ by at most $6\,(\ell - |\Lambda|/2)$. When doing this, each configuration $\tau \in \Omega_\Lambda^{|\Lambda|/2}$ is obtained from $\binom{|\Lambda|/2}{\ell-|\Lambda|/2}$ different $\sigma \in \Omega_\Lambda^\ell$. This implies:

$$Z_\Lambda^\ell \cdot \binom{\ell}{\ell - |\Lambda|/2} \cdot \min\{\gamma^{6(\ell-|\Lambda|/2)}, \gamma^{-6(\ell-|\Lambda|/2)}\} \leq Z_\Lambda^{|\Lambda|/2} \cdot \binom{|\Lambda|/2}{\ell - |\Lambda|/2}$$

Because $\ell \geq |\Lambda|/2$, we have that $\binom{\ell}{\ell-|\Lambda|/2} \geq \binom{|\Lambda|/2}{\ell-|\Lambda|/2}$. Rearranging terms, we obtain the desired result. $\square$

Lemma 10.5.4. *Let $\delta' < 1/2$ and γ be close enough to one such that there exists an $\varepsilon \in (\delta', 1/2)$ where:*

$$\left(\frac{\varepsilon}{1-\varepsilon}\right)^{(\varepsilon-\delta')/11} < \gamma < \left(\frac{1-\varepsilon}{\varepsilon}\right)^{(\varepsilon-\delta')/11} \qquad (*)$$

Let $\mathcal{P}$ be the boundary of an α-compressed configuration with n particles and let $\mathcal{D}$ be an $n^c \times n^c$ diamond inside $\mathcal{P}$. The probability that a configuration drawn from $\pi_{\mathcal{P}}$ has at most a δ'-fraction of particles with color c_2 in $\mathcal{D}$ is at most $\zeta^{n^{2c}}$ for some $\zeta < 1$, provided n is sufficiently large.

Proof. We first note that $Z_{\mathcal{P}}$ satisfies:

$$Z_{\mathcal{P}} \geq Z_{\mathcal{D}}^{n^{2c}/2} Z_{\overline{\mathcal{D}}}^{(n-n^{2c})/2} \min\{\gamma^{-8n^c-6}, 1\}$$

For any $k \leq \delta' n^{2c}$, let $S_{\mathcal{D}}^k \subseteq \Omega_{\mathcal{P}}$ be the set of configurations with exactly k particles of color c_2 in $\mathcal{D}$. Then,

$$\pi(S_{\mathcal{D}}^k) \leq \frac{Z_{\mathcal{D}}^k Z_{\overline{\mathcal{D}}}^{n/2-k} \max\{\gamma^{-8n^c-6}, 1\}}{Z_{\mathcal{D}}^{n^{2c}/2} Z_{\overline{\mathcal{D}}}^{(n-n^{2c})/2} \min\{\gamma^{-8n^c-6}, 1\}} \leq \max\{\gamma^{-8n^c-6}, \gamma^{8n^c+6}\} \frac{Z_{\mathcal{D}}^k}{Z_{\mathcal{D}}^{n^{2c}/2}}$$

We note that there are fewer than $3n^{2c}$ edges within $\mathcal{D}$, so

$$\binom{n^{2c}}{k} \min\{\gamma^{-3n^{2c}}, 1\} \leq Z_{\mathcal{D}}^k \leq \binom{n^{2c}}{k} \max\{\gamma^{-3n^{2c}}, 1\}$$

This yields:

$$\frac{Z_{\mathcal{D}}^k}{Z_{\mathcal{D}}^{n^{2c}/2}} \leq \frac{\binom{n^{2c}}{k} \max\{\gamma^{-3n^{2c}}, 1\}}{\binom{n^{2c}}{n^{2c}/2} \min\{\gamma^{-3n^{2c}}, 1\}} = \max\{\gamma^{-3n^{2c}}, \gamma^{3n^{2c}}\} \prod_{i=k+1}^{n^{2c}/2} \frac{i}{n^{2c}-i+1}$$

Let $\varepsilon \in (\delta', 1/2)$ be a constant satisfying $(*)$; by supposition we know such a ε exists. Because $k < \delta' n^{2c}$ and each term in the product above is less than one, we see that

$$\begin{aligned}\frac{Z_{\mathcal{D}}^{k}}{Z_{\mathcal{D}}^{n^{2c}/2}} &\leq \max\{\gamma^{-3n^{2c}}, \gamma^{3n^{2c}}\} \prod_{i=\delta' n^{2c}}^{\varepsilon n^{2c}} \frac{i}{n^{2c} - i + 1} \\ &\leq \max\{\gamma^{-3n^{2c}}, \gamma^{3n^{2c}}\} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{(\varepsilon-\delta')n^{2c}}\end{aligned}$$

Therefore,

$$\begin{aligned}\pi(S_{\mathcal{D}}^{k}) &\leq \max\{\gamma^{-8n^{c}-6}, \gamma^{8n^{c}+6}\} \frac{Z_{\mathcal{D}}^{k}}{Z_{\mathcal{D}}^{n^{2c}/2}} \\ &\leq \max\{\gamma^{-8n^{c}-6}, \gamma^{8n^{c}+6}\} \max\{\gamma^{-3n^{2c}}, \gamma^{3n^{2c}}\} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{(\varepsilon-\delta')n^{2c}} \\ &\leq \max\{\gamma^{-6}, \gamma^{6}\} \left(\max\{\gamma^{-11}, \gamma^{11}\} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{\varepsilon-\delta'}\right)^{n^{2c}}\end{aligned}$$

By supposition, the term in parentheses above is strictly less than one, meaning that for sufficiently large n we have that $\pi(S_{\mathcal{D}}^{k}) \leq (\zeta_k)^{n^{2c}}$ for some $\zeta_k < 1$. We now see that the probability that $\mathcal{D}$ has at most a δ'-fraction of particles with color c_2 is:

$$\pi\left(\bigcup_{k=0}^{\delta' n^{2c}} S_{\mathcal{D}}^{k}\right) \leq \sum_{k=0}^{\delta' n^{2c}} (\zeta_k)^{n^{2c}} \leq \delta' n^{2c} \max_{k} (\zeta_k)^{n^{2c}}.$$

For sufficiently large n, this is at most $\zeta^{n^{2c}}$ for some $\zeta < 1$. This concludes our proof. □

We could alternatively have chosen any $\varepsilon \in (\delta', 1/2)$ and instead obtained a range of γ for which the same result holds. Instead of finding an optimal value of ε as a function of δ' to obtain the largest range, we note that this optimum value is achieved near $\delta'/2 + 1/4$, halfway between δ' and $1/2$. Making this assumption allows us to get some concrete bounds on γ and δ', as we do in the corollaries below. First, we show this result implies the absence of separation.

Theorem 10.5.5. *Let $\mathcal{P}$ be any α-compressed boundary. Consider any $\delta < 1/4$ and γ close enough to one such that there exists an $\varepsilon \in (\delta/(1-2\delta), 1/2)$ where:*

$$\left(\frac{\varepsilon}{1-\varepsilon}\right)^{(\varepsilon-\delta/(1-2\delta))/11} < \gamma < \left(\frac{1-\varepsilon}{\varepsilon}\right)^{(\varepsilon-\delta/(1-2\delta))/11} \qquad (*)$$

For any $\beta > 0$, the probability that a configuration sampled from $\pi_{\mathcal{P}}$ is (β, δ)-separated is at most $\overline{\zeta}^{\sqrt{n}}$ for some constant $\overline{\zeta} < 1$, provided n is sufficiently large.

Proof. Since $\delta < 1/4$, it is possible to choose a $\delta' < \delta/(1-2\delta)$ satisfying $\delta' < 1/2$ and:

$$\left(\frac{\varepsilon}{1-\varepsilon}\right)^{(\varepsilon-\delta')/11} < \gamma < \left(\frac{1-\varepsilon}{\varepsilon}\right)^{(\varepsilon-\delta')/11}$$

which is possible because $(*)$ is satisfied with strict inequalities.

If a configuration $\sigma \in \Omega_{\mathcal{P}}$ is (β, δ)-separated, then by Lemma 10.5.2 for the δ' we have chosen there is an $n^c \times n^c$ diamond $\mathcal{D}$ that contains at most $\delta' n^{2c}$ particles of color c_2, where $c < 1/4$ such that n^c is an integer (for larger and larger n we can pick c closer and closer to $1/4$). The interior of $\mathcal{P}$ can be covered by at most n diamonds, so by a union bound and Lemma 10.5.4 the probability that σ is (β, δ)-separated is less than $n\zeta^{n^{2c}}$, for a constant $\zeta < 1$ and sufficiently large n. Therefore, exists a constant $\overline{\zeta} < 1$ such that $n\zeta^{n^{2c}} < \overline{\zeta}^{n}$, proving the theorem. $\square$

We wish to understand the range of γ for which there exists an ε satisfying the $(*)$ equation in Theorem 10.5.5; we focus on the upper bound on γ, as the lower bound is its reciprocal. If $\gamma < \left(\frac{1-\varepsilon}{\varepsilon}\right)^{\varepsilon/11}$, then we can always choose a δ small enough so that this equation is satisfied. Maximizing this expression exactly with respect to ε is challenging to do exactly, so we note that numerically this is achieved when $\varepsilon \approx 0.217812$, corresponding to an upper bound on γ of about 1.02564, which for simplicity we round down to the more explicit bound of $\gamma < 81/79 \approx 1.02532$.

Corollary 10.5.6. *For Markov chain $\mathcal{M}_S$ with parameters $\lambda > 1$ and $\gamma \in (79/81, 81/79)$ and any $\alpha > 1$, there exist constants β and δ such that for sufficiently large n, $\mathcal{M}_S$ does not accomplish (β, δ)-separation among α-compressed configurations at stationarity with high probability.*

Proof. Because $\gamma < 81/79$, then for $\varepsilon = 0.22$ it holds that $\gamma < \left(\frac{1-\varepsilon}{\varepsilon}\right)^{\varepsilon/11}$. Similarly, as $\gamma > 79/81$, for the same ε it holds that $\gamma > \left(\frac{\varepsilon}{1-\varepsilon}\right)^{\varepsilon/11}$. It is always possible to find a δ small enough so that $(*)$ of Theorem 10.5.5 holds. Thus, among α-compressed configurations, Theorem 10.5.5 guarantees that for any $\beta > 0$, the probability that the particles are not (β, δ)-separated is at least $1 - \zeta^{\sqrt{n}}$, where $\zeta < 1$ is a constant and n is sufficiently large. $\square$

Chapter 11

APPLICATIONS TO SWARM ROBOTICS AND GRANULAR ACTIVE MATTER

Swarm robotics and active matter physics provide powerful tools for the study and control of ensembles driven by internal sources. At the macroscale, controlling swarms typically utilizes significant memory, processing power, and coordination unavailable at the microscale (e.g., for colloidal robots that could be useful for fighting disease, fabricating intelligent textiles, and designing nanocomputers). In this chapter, we focus on a two-pronged approach to "programming" ensembles across scales by *leveraging the physics of interactions*: we pair theoretical, algorithmic abstractions of self-organizing particle systems with experimental robot systems of active granular matter that intentionally lack digital computation and communication, using minimal (or no) sensing and control to test theoretical predictions. The distributed, stochastic algorithms of Chapters 8–10 form the theoretical foundations of this approach. We first investigate an analogy between compression (Chapter 8) and a *phototaxing* behavior — i.e., directed locomotion towards or away from a light source — that arises in an ensemble of analog "smart, active particles", or *supersmarticles* (Section 11.1, [178]). We then establish a tight feedback loop between the theoretical predictions of separation and integration (Chapter 10) and the robot ensemble behaviors of *aggregation*, *dispersion*, and *collective transport* (Section 11.2, [128]). These results were obtained in collaboration with the active matter physics and robotics research group of Daniel I. Goldman at the Georgia Institute of Technology; this dissertation will primarily emphasize the algorithmic contributions.

11.1 Phototactic Supersmarticles

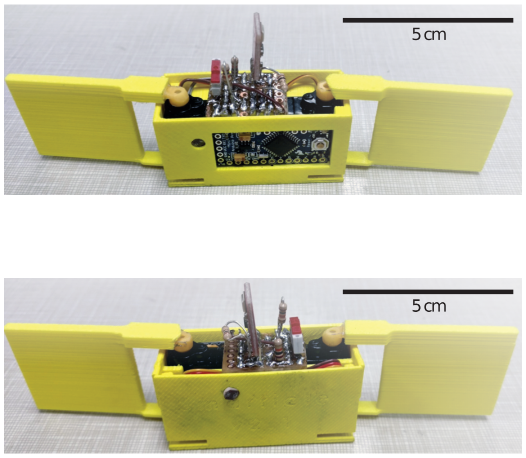
We investigate how a system of active granular matter can achieve *directed locomotion*, where the interactions of individual particles cause the system to move together as a collective in a desired direction. Specifically, we consider ensembles of particles that are individually incapable of locomotion. When constrained to remain in close proximity to other particles, we show that the ensemble can generate collective motion. Moreover, we show how external stimuli in the form of a light source introduce asymmetries in particle activity levels, yielding collective directed displacement towards or away from the light — a behavior known as *phototaxing*.

We first study phototaxing in an experimental testbed of "smart, active particles" (or *smarticles*) developed by the Goldman research group. Each smarticle is a small, three-link, planar robot that can sense the presence of light but is incapable of individual rotation or displacement. We refer to a collection of smarticles enclosed by an unanchored rigid ring as a *supersmarticle*. This "robot made of robots" is capable of behaviors more sophisticated than that of any individual smarticle; phototaxing is one such behavior, as we demonstrate in experiments (Section 11.1.1). Extensions of the present work on phototaxing in supersmarticles have employed a control theoretic and reinforcement learning approach [177]. Recent work has used supersmarticles a model testbed for investigating *rattling*, a unifying framework for characterizing and controlling emergent self-organization in complex systems [43].

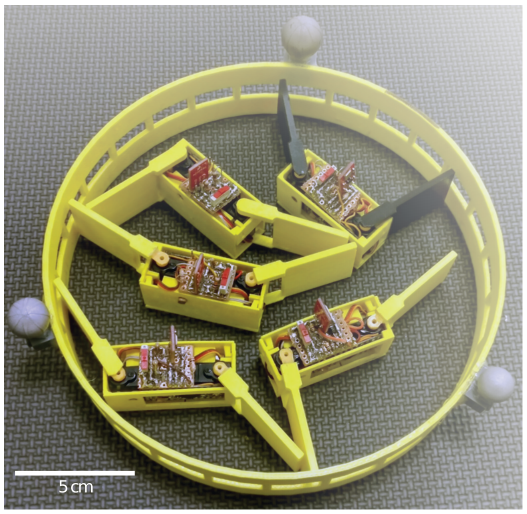
Phototaxing in supersmarticles is achieved by individual smarticles becoming inactive when they sense light. We posit that these inactive smarticles can be approximated as an extension of the boundary whose collision model is softer than that of the rigid ring. This is consistent with studies of randomly diffusing self-propelled particles enclosed in a boundary that is partitioned into two sections, one with a softer potential and the other with a more rigid potential [118, 182]. In simulations, it was shown that the pressure applied by the particles' collisions with the softer boundary is larger than that of the collisions with the more rigid boundary. We utilize this phenomena emerging from physical interactions in our experiments to achieve directed locomotion in our ensembles of individually non-motile robots.

To characterize phototaxing behaviors from a theoretical perspective, we utilize our results on *compression*, where a system of self-actuating computational particles gathers together as tightly as possible (Chapter 8, [32]). Remarkably, phototaxing can be achieved by just one subtle modification to the compression algorithm: particles change their likelihood of activating when they sense light. In Section 11.1.2, we prove that phototaxing occurs for systems of two and three particles. We then complement these rigorous results with simulations for much larger systems that demonstrate the same behavior.

Both the physical and theoretical systems we consider have three key properties: (*i*) individual particles move regularly with no sense of direction, (*ii*) there is a constraint ensuring particles remain in close proximity to each other, and (*iii*) particles' activity levels change in response to light. In both systems, these basic properties suffice to produce phototaxing. Perhaps the most surprising result is that this direction motion towards (or away from) the light source is achieved without all particles knowing where the light source is; the occlusion of light by other particles suffices for the system to move accordingly, using strictly local interactions.



(a)



(b)

Figure 63. Smarticles and Supersmarticle Collectives. (a) An individual smarticle from the front (top) and back (bottom). (b) A supersmarticle composed of five individual smarticles.

11.1.1 Supersmarticle Design and Experimental Results

Each *smarticle* is a small ($14 \times 2.5 \times 3$ cm), three-link, planar robot with two revolute joints where only the center link is in contact with the ground (Figure 63a). A smarticle can change its shape in place by changing the angle of its outer links (or "arms"), but cannot rotate or displace individually. Smarticles can be programmed with predefined, shape changes in their joint-angle space; we call a closed, periodic trajectory in joint-angle space a *gait*. A *supersmarticle* is a collection of smarticles enclosed in an unanchored, rigid ring (Figure 63b). Details of the smarticle hardware and design can be found in [178].

Each smarticle individually performs one of two behaviors: one where the smarticle is *active*, changing its shape according to the square gait shown in Figure 64, and another where the smarticle is *inactive*, holding its three links fixed and parallel to

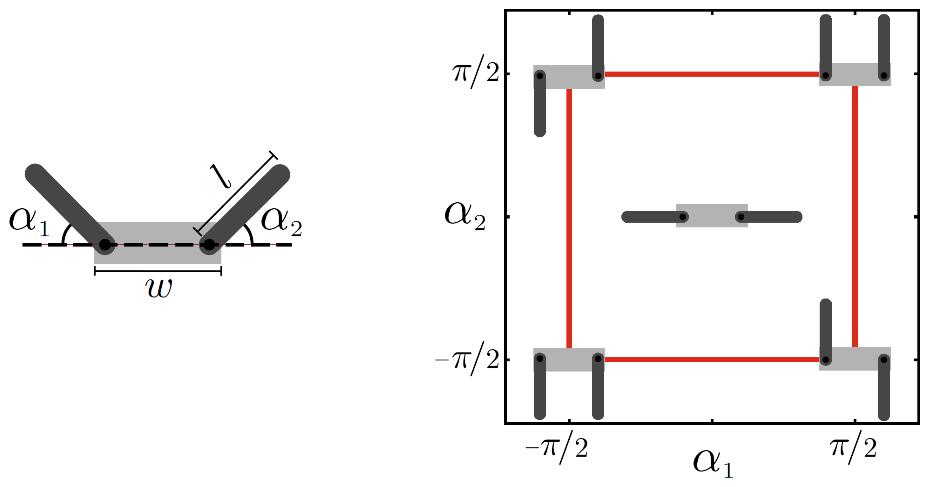


Figure 64. The Smarticle Square Gait. Left: The configuration space of a single smarticle defined by the angles α_1 and α_2 between the outer and inner links. Right: The square gait used in the phototaxing experiments.

each other. A smarticle persists in the active state until either of its photoresistors (found on the front and back of the smarticle) detect light above a fixed threshold. It then transitions to the inactive state, where it persists until the light sensed by both of its photoresistors once again drops below the threshold, allowing it to transition back to the active state.

Given the light sensor locations on the smarticles' bodies and the geometry of the planar experimental setup, typically only one smarticle is inactive at a time. The inactive smarticle occludes light from reaching other smarticles behind it, keeping their photoresistors below the threshold necessary to become inactive. This occlusion effectively produces a light gradient across the supersmarticle which provides a decentralized, stigmergic communication method. Each smarticle's behavior is a response to its local environment, which in turn affects the local environment of its neighbors.

For each experiment, we placed the supersmarticle at the center of a 60×60 cm level test plate with its composing smarticles randomly oriented. All experiments were performed in a dark room so that, by default, smarticles remain in the active

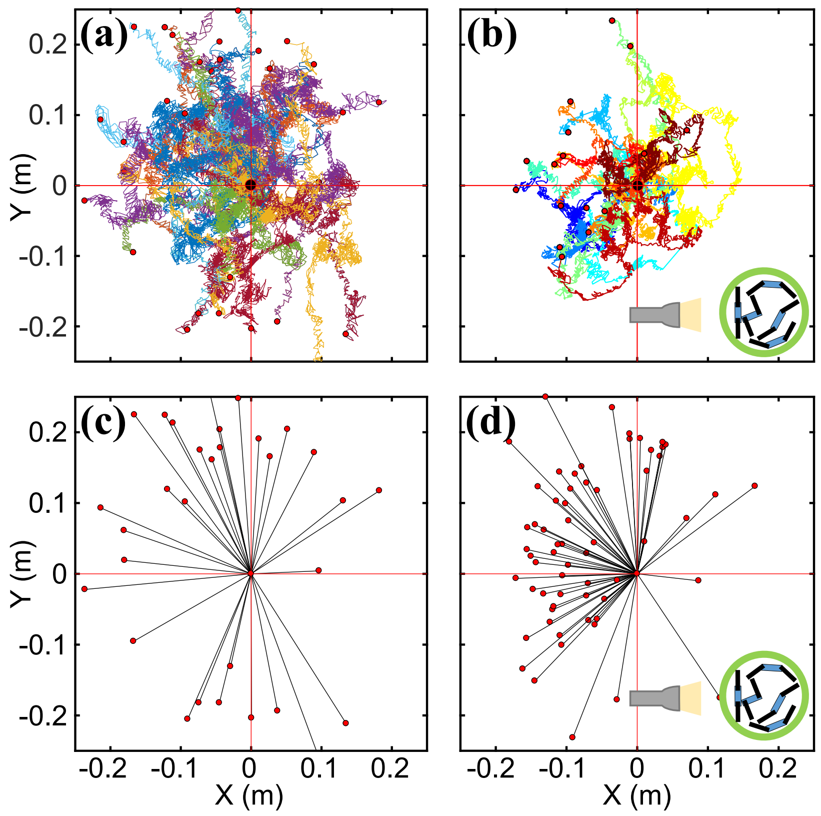


Figure 65. Supersmarticle Trajectories. The top row shows raw trajectories of the supersmarticle's center of geometry for (a) unbiased and (b) light-biased motion. Each colored trajectory represents a separate experimental trial beginning at $(0, 0)$ and ending at a red circle. The bottom row shows the initial and final positions of a superset of trials shown in the top row for (c) unbiased and (d) light-biased motion. In all plots, trials where the light was not directed in the $+x$ direction were rotated for the sake of comparison, as indicated by the flashlight legends.

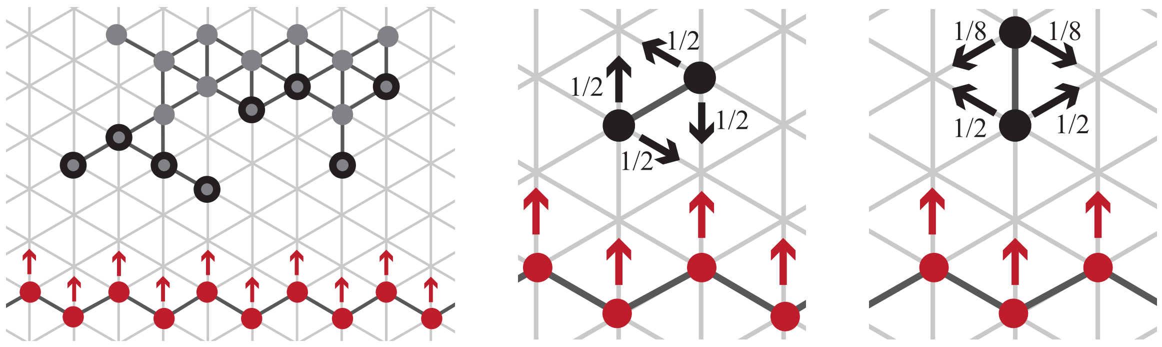
state. In control experiments, no light source is introduced and all smarticles remain active; in phototaxing experiments, a light source is placed at the center of one of the test plate's edges and is continuously directed towards the nearest exposed photoresistor, rendering a single smarticle inactive. An experiment run is ended when the supersmarticle translates to an edge of the test plate or after 10 minutes, whichever happens first. The phototaxing experiments were repeated with the light source at each of the four possible locations to avoid any systematic error.

Diffusive behavior was observed in both the control (Figures 65a and 65c) and phototaxing experiments (Figures 65b and 65d), but the presence of inactive smarticles near the light source introduces a biased drift towards the light. Supersmarticles in the phototaxing experiments consistently diffused in the direction of the light source with an average success rate of $82.3 \pm 6.0\%$ across all trials. We can further examine the supersmarticles' locomotion as mean-squared displacement over time $\langle r^2(t) \rangle = vt^\alpha$, where v is the collective's characteristic speed and α characterizes the movement as either subdiffusive ($\alpha < 1$), diffusive ($\alpha = 1$), or superdiffusive ($\alpha > 1$). By fitting a line to the log-log plot of our supersmarticles' mean-squared displacement curves, we find that the control experiments have a mean slope of $\alpha = 0.99$ m^2/s and the phototaxing experiments have a mean slope of $\alpha = 1.04 \pm 0.02$ m^2/s. This indicates that the presence of a light source shifts the supersmarticles' locomotion from diffusive to superdiffusive, where the active transport phenomenon causes the system to propagate towards the light source.

11.1.2 A Distributed, Stochastic Algorithm for Phototaxing

To complement the physical experiments, we give a local, distributed algorithm for phototaxing under the geometric amoebot model. We first prove that this algorithm causes directed locomotion in response to light for systems of two and three amoebots and then give simulations that demonstrate this same behavior for larger systems.

We assume that the system initially forms a connected configuration of contracted amoebots. We model light as a collection of point sources that each broadcast light along lattice lines in the same direction (see Figure 66a). The first amoebot in each lattice line senses the light, while all others behind it do not. We assume these light



(a) (b) $\mathrm{E}[\Delta h] = 0$ (c) $\mathrm{E}[\Delta h] = 3/32$

Figure 66. Modeling Phototaxing in the Amoebot Model. (a) An amoebot system configuration (gray dots) with point sources broadcasting light upwards along the lattice edges. Amoebots outlined in black can sense the light while all others are occluded. (b)–(c) The two types of configurations for systems of two amoebots analyzed in Theorem 11.1.1 and the probabilities of each amoebot's movement if it is activated next.

sources are positioned sufficiently far from the system so as to not interfere with its motion. To quantify the amoebot system's directed locomotion, we define the *height* of a amoebot system to be the y-coordinate of its center of mass, where each edge of G_Δ is assumed to have length 1 and the light sources have y-coordinate 0 or $-1/2$. A distributed algorithm $\mathcal{A}$ is said to solve the phototaxing problem if, when each amoebot independently executes algorithm $\mathcal{A}$, the height of the amoebot system strictly increases or decreases in expectation.

Our distributed algorithm $\mathcal{A}_P$ for phototaxing (Algorithm 17) is a remarkably simple extension of the stochastic, distributed algorithm for compression (Algorithm 13): if an amoebot senses light, it executes an activation of compression; otherwise, it executes an activation of compression with probability $1/4$. We reproduce the necessary details of compression in the pseudocode; however, for simplicity, we obfuscate the usual details of the distributed translation and instead assume particles expand and contract to their new positions in a single activation (see Section 8.2.2 for details).

The 1/4 probability used in $\mathcal{A}_P$ was chosen because it works well in practice, causing amoebots that sense the light to be four times as active as those that do not in expectation. Smaller probabilities cause different structural configurations to emerge while larger values correspond to slower phototaxing.

Algorithm 17 Local, Distributed Algorithm $\mathcal{A}_P$ for Phototaxing

```
1: if amoebot $A$ senses light then Compression( ).
2: else with probability 1/4 Compression( ).
3: function Compression( )
4:     Let $\ell$ denote the current location of $A$.
5:     Choose a neighboring location $\ell'$ and $q \in (0,1)$ each uniformly at random.
6:     if $\ell'$ is unoccupied then
7:         $A$ expands to simultaneously occupy $\ell$ and $\ell'$.
8:         Let $e = |N(\ell)|$ be the number of neighbors $A$ had when it was contracted at $\ell$.
9:         Let $e' = |N(\ell')|$ be the number of neighbors $A$ would have if it contracts to $\ell'$.
10:        if $e \neq 5$, locations $\ell$ and $\ell'$ satisfy Property 8.2.1 or 8.2.2, and $q < \lambda^{e'-e}$ then
11:            $A$ contracts to $\ell'$.
12:        else $A$ contracts back to $\ell$.
```

In a system of two amoebots, algorithm $\mathcal{A}_P$ simplifies to the following: if amoebot A senses light, then move to one of the two unoccupied locations adjacent to both amoebots chosen uniformly at random; otherwise, do so with probability 1/4. We prove that this simplified algorithm solves the phototaxing problem.

Theorem 11.1.1. *For an amoebot system of $n = 2$ amoebots, algorithm $\mathcal{A}_P$ solves the phototaxing problem.*

Proof. We show that after two amoebot activations, the expected height of the system has increased by at least 3/64, implying that the amoebot system is moving away from the light source. Up to translation and reflection, there are two possible types of configurations a system of two amoebots can be in: either (i) both amoebots can sense the light, or (ii) one amoebot occludes the other; see Figure 66b and 66c, respectively. Regardless of configuration type, each amoebot is equally likely to activate next. In a type (i) configuration, case analysis shows the expected change in height after one activation is 0. Furthermore, with probability 1/2 the system remains in type (i) and with probability 1/2 it enters type (ii). In a type (ii) configuration, there are two cases. If the occluded amoebot activates next, then with probability 1/4 it moves a distance of $-1/2$ in the y-direction, decreasing the height of the system by 1/4. Otherwise, if the amoebot exposed to the light activates next, then it moves a distance of 1/2 in the y-direction, increasing the height of the system by 1/4. Altogether, the expected change in the height h of the system is:

$$\mathrm{E}\left[\Delta h\right] = \frac{1}{2} \cdot \frac{1}{4} \cdot \left(-\frac{1}{4}\right) + \frac{1}{2} \cdot 1 \cdot \frac{1}{4} = \frac{3}{32}$$

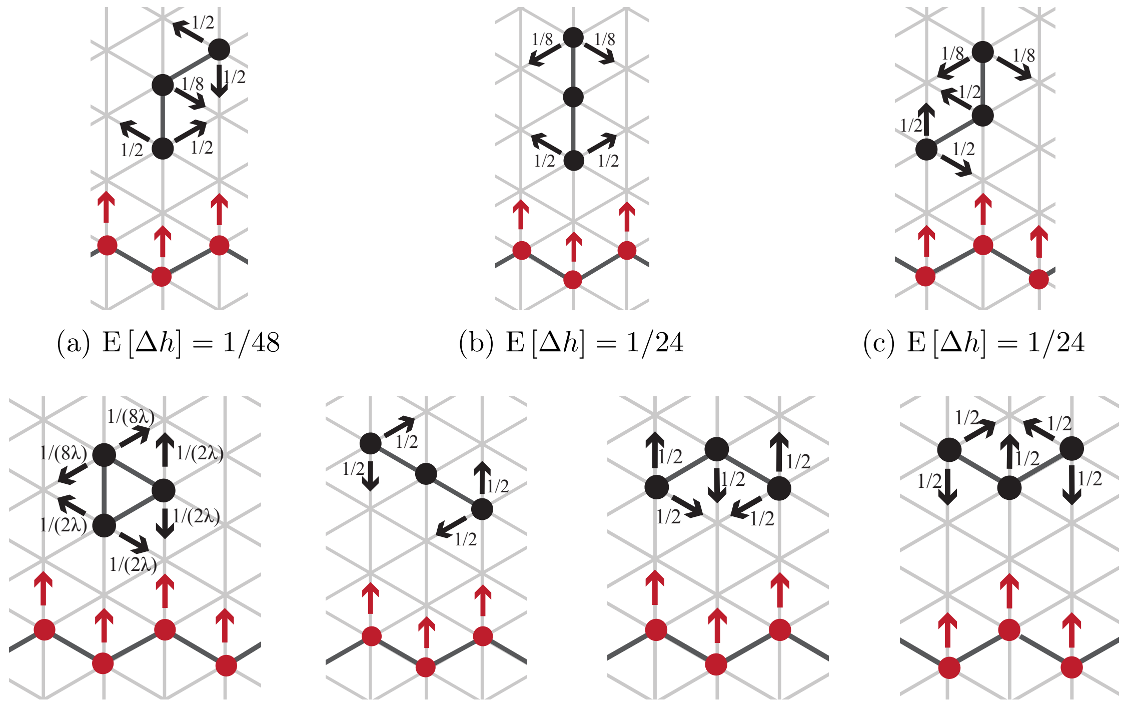
If the system starts in a type (i) configuration, then conditioning on the configuration type after one activation shows that the expected height of the system has increased by at least 3/64 in two activations. Otherwise, if the system starts in a type (ii) configuration, the height of the system increases by at least $3/32 > 3/64$ in two activations. □

The same results holds for systems of three amoebots, albeit with a slightly slower drift. In systems of exactly three amoebots, algorithm $\mathcal{A}_P$ simplifies to Algorithm 18. Observe that the compression bias parameter λ and the Metropolis filter based on the number of neighbors now play a role in the probability calculations.

Algorithm 18 Algorithm $\mathcal{A}_P$ for Phototaxing on $n = 3$ Amoebots

```
1: Let $L$ be the set of (at most two) valid locations to move to.
2: for each location $\ell' \in L$ do
3:     Set probability $p_{\ell'} \leftarrow 1/2$.
4:     if moving to $\ell'$ decreases the number of neighbors then $p_{\ell'} \leftarrow p_{\ell'}/\lambda$.
5:     if amoebot $A$ does not sense light then $p_{\ell'} \leftarrow p_{\ell'}/4$.
6: Move to a location $\ell' \in L$ with probability $p_{\ell'}$; otherwise, do not move.
```

Theorem 11.1.2. *For an amoebot system of $n = 3$ amoebots, algorithm $\mathcal{A}_P$ solves the phototaxing problem with any bias parameter $\lambda > 2 + \sqrt{2}$.*



(a) $\mathrm{E}[\Delta h] = 1/48$ (b) $\mathrm{E}[\Delta h] = 1/24$ (c) $\mathrm{E}[\Delta h] = 1/24$

(d) $\mathrm{E}[\Delta h] = 0$ (e) $\mathrm{E}[\Delta h] = 0$ (f) $\mathrm{E}[\Delta h] = 0$ (g) $\mathrm{E}[\Delta h] = 0$

Figure 67. Phototaxing in a System of Three Amoebots. The seven types of configurations for systems of three amoebots analyzed in Theorem 11.1.2 and the probabilities of each amoebot's movement if it is activated next.

Proof. We show that after three amoebot activations, the expected height of the system has increased by $1/(64\lambda)$. Up to translation and reflection, there are seven possible types of configurations a system of three amoebots could be in; these are shown in Figure 67. Case analysis shows that the expected change in height is nonnegative in all seven types. For types (a)–(c), the expected increase in height after one amoebot activation is at least $1/(64\lambda)$; since expected height is nondecreasing, the same holds after three consecutive amoebot activations. For types (d)–(g), the expected change in height after a single amoebot activation is 0, so we consider multiple consecutive activations. There is a positive probability that a type (d) configuration will transition to a type (a) or (c) configuration in one activation; thus, using conditional expectation, we have that the expected increase in height after two activations is at least $1/(64\lambda)$.

Similarly, the expected increase in height after two activations starting from a type (e) configuration is 1/96; because $\lambda > 2 + \sqrt{2}$, we have $1/96 > 1/(64\lambda)$.

Starting from a type (f) or (g) configuration, it takes at least two amoebot moves to reach a configuration where the system height should increase in expectation after an additional activation. A type (f) or (g) configuration reaches a type (a) configuration after two activations with probability $1/18 + 1/(9\lambda)$ and reaches a type (c) configuration after two activations with probability $1/18 + 5/(72\lambda)$. Thus, the total expected increase in height after three activations starting from a type (f) or (g) configuration is:

$$\left(\frac{1}{18} + \frac{1}{9\lambda}\right)\frac{1}{48} + \left(\frac{1}{18} + \frac{5}{72\lambda}\right)\frac{1}{24} = \frac{1}{288} + \frac{1}{192\lambda} > \frac{1}{64\lambda}$$

where the final inequality follows because $\lambda > 2 + \sqrt{2}$. Therefore, starting from any configuration of three amoebots, the expected height of the system will increase by at least $1/(64\lambda)$ in three activations. $\square$

While our proofs only hold for small systems, algorithm $\mathcal{A}_P$ exhibits phototaxing behavior for arbitrarily large amoebot systems. A simulation for a system of 91 amoebots is shown in Figure 68. Though the motion is largely random, there is an evident drift away from the light sources. This drift was consistent across all simulation runs.

11.2 Aggregation, Dispersion, and Collective Transport in BOBbots

Self-organizing collective behaviors are found throughout nature, including shoals of fish aggregating to intimidate predators [137], fire ants forming rafts to survive floods [149], and bacteria forming biofilms to share nutrients when they are metabolically stressed [133]. Inspired by such systems, researchers in swarm robotics and

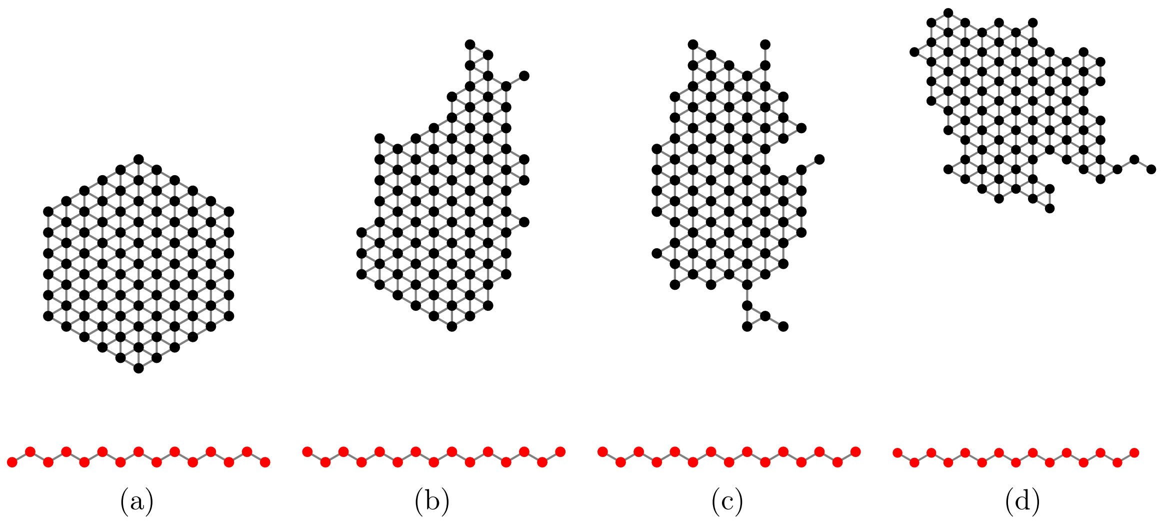


Figure 68. Simulation of Algorithm $\mathcal{A}_P$ for Phototaxing. An system of 91 amoebots running algorithm $\mathcal{A}_P$ with $\lambda = 4$ after (a) 0, (b) 10 million, (c) 20 million, and (d) 30 million amoebot activations.

programmable active matter have used many approaches towards enabling ensembles of simple, independent units to cooperatively accomplish complex tasks [20, 27, 73]. Both control theoretic and distributed computing approaches have achieved some success, but often rely critically on robots computing and communicating complex state information, requiring relatively sophisticated hardware that can be prohibitive at small scales [77, 87]. Alternatively, statistical physics approaches model swarms as systems being driven away from thermal equilibrium by robot interactions and movements (see, e.g., [140, 153]). Tools from statistical physics such as the Langevin and Fokker-Planck equations can then be used to analyze the mesoscopic and macroscopic system behaviors [101]. Current approaches present inherent tradeoffs, especially as individual robots become smaller and have limited functional capabilities [106, 195] or approach the thermodynamic limits of computing and power [193].

To apply to a general class of micro- or nano-scale devices with limited capabilities, we focus on systems of autonomous, self-actuated entities that utilize strictly local interactions to induce macroscale behaviors. Two behaviors of interest are *dynamic free aggregation*, where agents gather together without preference for a specific aggregation site (see Section 3.2.1 of [20]), and *dispersion*, its inverse. These problems are widely studied, but most work either considers robots or models with relatively powerful capabilities — e.g., persistent memory for complex state information [160, 175] or long-range communication and sensing [81, 93, 156] — or lack rigorous mathematical foundations explaining the generality and limitations of their results as sizes scale [45, 91, 129]. Recent studies on active interacting particles [2] and inertial, self-organizing robots [56] employ physical models to treat aggregation and clustering behaviors, but neither prove behavior guarantees that scale with system size and volume. Supersmarticle ensembles [43, 177] are significantly more complex, exhibiting many transient behavioral patterns stemming from their many degrees of freedom and chaotic interactions, making them less amenable to rigorous algorithmic analysis.

Here we take a two-pronged approach to understanding the fundamental principles of programming task-oriented matter that can be implemented across scales without requiring sophisticated hardware or traditional computation that leverages the physics of local interactions. We use a theoretical abstraction of self-organizing particle systems (SOPS), where we can design and rigorously analyze simple distributed algorithms to accomplish specific goals that are flexible and robust to errors. We then build a new system of deliberately rudimentary active "cohesive granular robots" (which, to honor granular physics pioneer Robert Behringer, we call "BOBbots" for Behaving, Organizing, Buzzing robots) to test whether the theoretical predictions can be realized in a real-world damped driven system. Remarkably, the lattice based equilibrium model quantitatively captures the aggregation dynamics of the robots. With a provable algorithmic model and even simpler BOBbots capturing the algorithm's essential rules, we next explore how the BOBbot aggregation dynamics can be used for the task of *object transport*, clearing non-robot impurities from the environment. This complementary approach demonstrates a new integration of the fields of distributed algorithms, active matter, and granular physics that navigates a translation from theoretical abstraction to practice, utilizing methodologies inherent to each field.

For the purposes of this dissertation, we will primarily emphasize the algorithmic contributions connecting the stochastic algorithm for separation and integration (Chapter 10, [31]) to the present goals of aggregation and dispersion. In particular, we omit the details of the BOBbot design and manufacturing, the stress sensing experiments, and the analysis of the continuum dynamics modeled by the Cahn–Hilliard equation. These details can be found in the full paper [128].

11.2.1 The Markov Chain $\mathcal{M}_A$ for Aggregation and Dispersion

While many systems use interparticle attraction and sterical exclusion to achieve system-wide aggregation and interparticle repulsion to achieve dispersion, these methods typically use some long-range sensing and tend to be nonrigorous, lacking formal proofs guaranteeing desirable system behavior. We instead leverage our study of stochastic processes for *self-organizing particle systems* (SOPS) that allows us to define formal distributed algorithms and rigorously quantify long-term behavior (Chapters 8–10).

In Chapter 8, we analyzed the Markov chain $\mathcal{M}_C$ for compression and expansion, which are analogous to aggregation and dispersion under the assumption that the particle system remains simply connected (i.e., the system forms a single connected cluster with no holes). This Markov chain is based on local moves that connect the state space of all simply connected configurations of particles. Moves are defined so that each particle, when activated by its own Poisson clock (i.e., after a delay chosen at random from a Poisson distribution with constant mean), chooses a random neighboring node and moves there with a probability that is a function of the number of neighbors in the current and new positions provided the node is unoccupied and the move satisfies local conditions that guarantee the configuration stays simply connected. In particular, for configurations σ and τ differing by the move of a single particle P along a lattice edge, the transition probability is defined as $M(\sigma, \tau) \propto \min(1, \lambda^{e'-e})$, where $\lambda > 0$ is a bias parameter that is an input to the algorithm, e is the number of neighbors of P in σ, and e' is the number of neighbors of P in τ. These probabilities arise from the celebrated Metropolis–Hastings algorithm [102] and are defined so that the Markov chain converges to a unique Boltzmann distribution π such that $\pi(\sigma)$ is proportional to $\lambda^{e(\sigma)}$, where $e(\sigma)$ is the number of nearest neighbor pairs in σ (i.e., those pairs that are adjacent on the lattice).

We showed that under $\mathcal{M}_C$, the simply connected particle system provably aggregates into a compact conformation when $\lambda > 3.42$ and expands into a conformation with nearly maximal (linear) perimeter when $\lambda < 2.17$ with high probability (Theorems 8.3.5 and 8.4.7). However, despite rigorously achieving both aggregation and dispersion, $\mathcal{M}_C$ has two notable drawbacks that make it infeasible for direct implementation in a physical system of simple robots: the connectivity requirement that tethers the particles together and the "look ahead" requirement used to calculate transition probabilities ensuring convergence to the desired Boltzmann distribution.

To address these issues, we define a modified aggregation and dispersion algorithm $\mathcal{M}_A$ (Algorithm 19) where particles can disconnect and moves rely only on the current state. Here, particles occupy nodes of a finite region of the triangular lattice, again

Algorithm 19 Markov Chain $\mathcal{M}_A$ for Aggregation and Dispersion

```
   From any configuration $\sigma_0$ of $n$ contracted particles in an $\alpha$-compressed region, repeat:
1: Choose a particle $P$ uniformly at random from among all $n$ particles.
2: Let $\ell$ be the location of $P$ and $e$ be the number of neighbors it has.
3: Choose a neighboring location $\ell'$ and $q \in (0,1)$ each uniformly at random.
4: if $\ell'$ is unoccupied and $q < \lambda^{-e}$ then $P$ moves to $\ell'$.
5: else $P$ remains at $\ell$.
```

moving stochastically and favoring configurations with more pairs of neighboring particles. Each particle has its own Poisson clock and, when activated, chooses a random adjacent lattice node. If that node is unoccupied, the particle moves there with probability λ^{-e}, where e is the number of current neighbors of the particle, for bias parameter $\lambda > 0$. Thus, rather than biasing particles towards nodes with more neighbors, we instead discourage moves away from nodes with more neighbors, with larger λ corresponding to a stronger ferromagnetic attraction between particles (Figure 69a).

We first prove that this new chain $\mathcal{M}_A$ converges to the same Boltzmann distribution $\pi(\sigma) \propto \lambda^{e(\sigma)}$ over system configurations σ as the original algorithm $\mathcal{M}_C$.

Lemma 11.2.1. *The unique stationary distribution of $\mathcal{M}_A$ is $\pi(\sigma) = \lambda^{e(\sigma)}/Z$, where $Z = \sum_\tau \lambda^{e(\tau)}$ is a normalizing constant.*

Proof. Let σ and τ be any two SOPS configurations with $\sigma \neq \tau$ such that $M(\sigma,\tau) > 0$, implying that τ can be reached from σ by a single move of some particle P. Suppose P has e neighbors in σ and has e' in τ. We must show the *detailed balance condition* holds with respect to the transition probabilities:

$$M(\sigma,\tau)\pi(\sigma) = M(\tau,\sigma)\pi(\tau)$$

The Markov chains in Chapters 8–10 were designed using the Metropolis–Hastings algorithm [102] which specifies transition probabilities $M(\sigma,\tau) = \min\{\pi(\tau)/\pi(\sigma), 1\}$ to

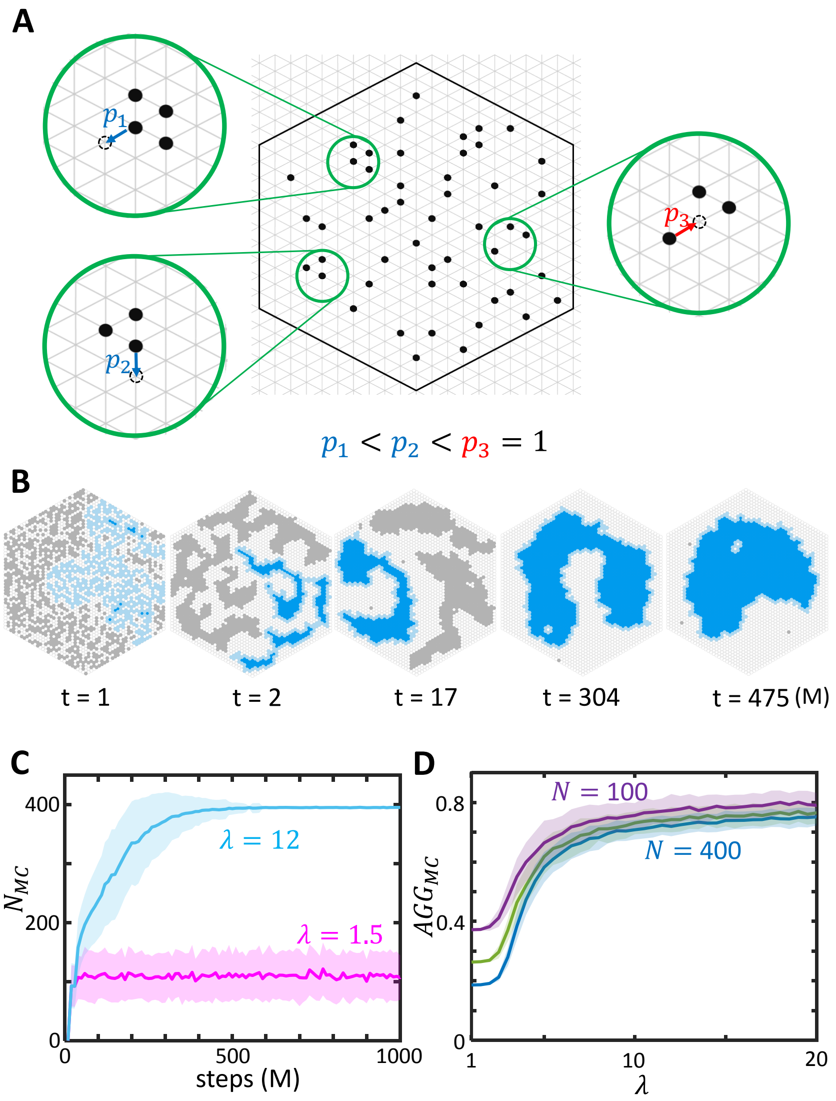


Figure 69. The Markov Chain $\mathcal{M}_A$ for Aggregation. (a) A particle moves away from a node where it has e neighbors with probability λ^{-e}, where $\lambda > 0$. Thus, moves from locations with more neighbors are made with smaller probability than those with fewer (e.g., in the insets, $p_1 = \lambda^{-3} < p_2 = \lambda^{-2} < p_3 = 1$). (b) A simulation of $\mathcal{M}_A$ with 1377 particles for $\lambda = 7.5$ showing progressive aggregation. The bulk of the largest connected component is shown in blue and its periphery is shown in light blue. (c) Time evolution of N_{MC}, the size of the largest connected component, showing dispersion for $\lambda = 1.5$ and aggregation for $\lambda = 12$. The simulations use 400 particles. (d) Phase change in λ-space for the aggregation metric $AGG_{MC} = N_{MC}/(k_0 P_{MC}\sqrt{n})$, where k_0 is a scaling constant, P_{MC} is the number of particles on the periphery of the largest component, and n is the total number of particles. This phase change is qualitatively invariant to the system's size.

capture the ratio between stationary weights of the current and proposed configurations. So we have that $\pi(\tau)/\pi(\sigma) = \lambda^{e'-e}$. It is then easy to see that this ratio is unchanged by the modified transition probabilities where $M(\sigma, \tau) = \lambda^{-e}$ and $M(\tau, \sigma) = \lambda^{-e'}$, and thus detailed balance is satisfied:

$$\frac{M(\sigma, \tau)}{M(\tau, \sigma)} = \frac{\lambda^{-e}}{\lambda^{-e'}} = \lambda^{e'-e} = \frac{\pi(\tau)}{\pi(\sigma)}$$

Therefore, since π satisfies detailed balance and $\mathcal{M}_A$ is an ergodic finite Markov chain, we conclude that π is the unique stationary distribution of $\mathcal{M}_A$. $\square$

This lemma shows that Markov chain $\mathcal{M}_A$ will converge to the same stationary distribution as $\mathcal{M}_C$ did, but can do so without "look ahead" information. Algorithm $\mathcal{M}_C$ further lifts the connectivity constraint by allowing particles to disconnect, instead restricting their possible locations from the infinite triangular lattice to a bounded region. However, our analysis of the stationary distribution π for $\mathcal{M}_C$ focused on simply connected configurations and do not generalize to the disconnected setting. To overcome this obstacle, we instead turn to our analysis of the Markov chain $\mathcal{M}_S$ for separation and integration (Chapter 10). We proved that among particle systems of two colors that were α-compressed (i.e., that have boundary length at most α times the minimum possible perimeter), $\mathcal{M}_S$ would provably achieve separation whenever $\gamma > 4^{5/4} \approx 5.66$ and would provably achieve integration whenever $0.98 \approx 79/81 < \gamma < 81/79 \approx 1.02$ (Corollaries 10.4.12 and 10.5.6). This separation algorithm can be applied to the setting where an α-compressed, bounded region of the lattice is completely filled with particles that move by "swapping" places with their neighbors. By viewing particles of one color as "empty space" and particles of the other color as our particles of interest, the swap moves in the separation algorithm correspond to particle moves within a bounded area. These are precisely the moves used in our aggregation algorithm, where separation corresponds to aggregation and integration corresponds to dispersion. Thus, our results for separation (Corollaries 10.4.12 and 10.5.6) combined with the following formal definition for aggregation and dispersion that is modeled after Definition 10.4.1 of separation immediately implies the subsequent theorem.

Definition 11.2.2. For $\beta > 0$ and $\delta \in (0, 1/2)$, a configuration σ is (β, δ)-*aggregated* if there is a subset R of lattice nodes such that:

1. At most $\beta\sqrt{n}$ edges have exactly one endpoint in R;
2. The density of particles in R is at least $1 - \delta$; and
3. The density of particles not in R is at most δ.

A configuration is *dispersed* if no such (β, δ) exist.

Theorem 11.2.3. *Let configuration σ be drawn from the stationary distribution of $\mathcal{M}_A$ on a bounded, α-compressed region of the triangular lattice with a sufficiently large number of particles n. If $\lambda > 5.66$, then with high probability there exist $\beta > 0$ and $0 < \delta < 1/2$ such that σ will be (β, δ)-aggregated. However, when $0.98 < \lambda < 1.02$, the configuration σ will be dispersed with high probability.*

Varying values of λ in simulation gives strong indication that dispersion persists for larger values of λ and the aggregation algorithm undergoes a phase transition whereby the macroscopic behavior of the system suddenly changes from dispersion to aggregation (Figure 69c–d), mimicking the fixed magnetization ferromagnetic Ising model which motivated our Markov chain algorithm. Nonetheless, our proofs demonstrate that our system has two distinct phases of behavior for different ranges of λ for any system with a sufficiently large number of interacting particles, which is enough for our purposes.

11.2.2 BOBbots: A Model Active Cohesive Granular Matter System

Next, to test whether the lattice-based equilibrium system can be used to control a real-world swarm in which there are no guarantees of detailed balance or Boltzmann distributions, we introduce a collective of active cohesive granular robots which we name *BOBbots* (Figure 70a–70c) — Behaving, Organizing, Buzzing robots — whose design *physically embodies* the aggregation algorithm. Driven granular media provide a useful soft matter system to integrate features of the physical world into the toolkit for programming collectives. This builds upon three decades of work understanding how forced collections of simple particles interacting locally can lead to remarkably complex and diverse phenomena, not only mimicking solids, fluids, and gasses [6, 131] — e.g., in pattern formation [79, 143], supercooled and glassy phenomena [98, 121], and shock waves [170] — but also displaying phenomena characteristic of soft matter systems such as stress chains [108] and jamming transitions [24, 46]. While cohesive granular materials are typically generated in situations where particles are small (powders, with interactions dominated by electrostatic or even van der Waals interactions) or wet (with interactions dominated by formation of liquid bridges between particles) [104, 148], we generate our cohesive granular robots using loose magnets which can rotate to always achieve attraction.

The movement and interactions between BOBbots were designed to capture the salient features of the abstract stochastic algorithm while replacing all sensing, communication, and probabilistic computation with physical morphology and interactions. Each BOBbot has a cylindrical chassis with a base of elastic "brushes" that are physically coupled to an off-center eccentric rotating mass vibration motor (ERM). The vibrations caused by the rotation of the ERM are converted into locomotion by

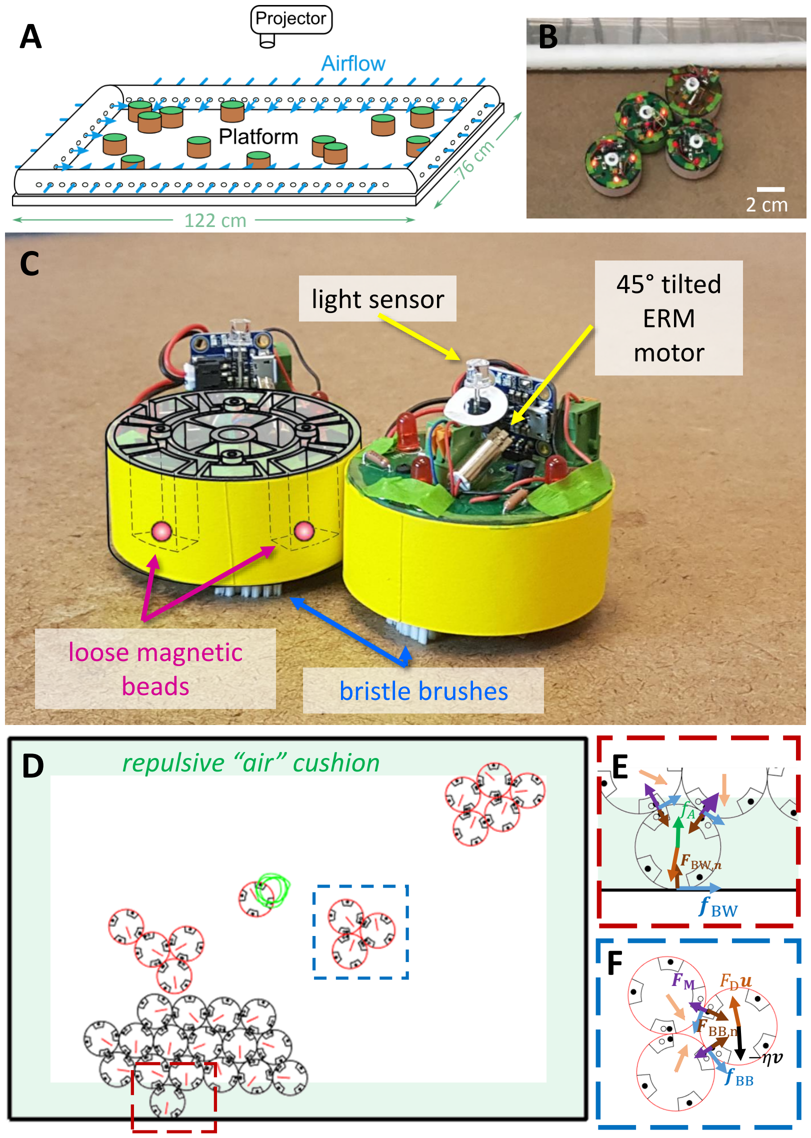


Figure 70. BOBbots and their collective motion. (a) Schematic of experimental setup. BOBbots are placed in a level arena with airflow gently repelling them from the boundaries. (b) A closeup of the experimental platform. (c) Mechanics of the BOBbots. Loose magnetic beads housed in the BOBbots' peripheries can reorient so BOBbots always attract each other. The vibration of the ERM motor and the asymmetry of bristles lead to the directed motion. The light sensor activates the motion. (d) Discrete element method simulation setup. (e) The BOBbot-boundary interactions: airflow repulsion f_A, BOBbot-boundary friction f_{BW}, and normal force $F_{\text{BW,n}}$. (f) The inter-BOBbot interactions: attraction between magnetic beads F_M, inter-BOBbot friction f_{BB}, and sterical exclusion $F_{\text{BB,n}}$.

the brushes (Figure 70c). Due to asymmetry in our construction of this propulsion mechanism, the BOBbots traverse predominantly circular trajectories [126] that are randomized through their initial conditions but — unlike the SOPS particles — are inherently deterministic with some noise and occur at a constant speed per robot distributed as $v_0 = 4.8 \pm 2.0$ cm/s.

Analogous to the modified transition probabilities in the aggregation algorithm that discourage particles from moving away from positions where they have many neighbors, each BOBbot has loose magnets housed in shells around its periphery that always reorient to be attractive to nearby BOBbots (Figure 70c). The probability that a BOBbot detaches from its neighbors is negatively correlated with the attractive force from the number of engaged magnets, approximating the movement probabilities given by the algorithm which scale inversely and geometrically with the number of neighbors. The strength of the magnets F_{M0} determines whether the system aggregates or disperses in the long run, analogous to λ in the algorithm.

To allow for study of larger BOBbot ensembles and more comprehensive sweeps of parameter space, we also performed Discrete-Element Method (DEM) simulations of the BOBbots (see Figure 70d–f). The motion of an individual BOBbot is modeled as a set of overdamped Langevin-type equations governing both its translation and rotation subject to its diffusion, drift [112], magnetic attraction, and sterical exclusion with other BOBbots. The translational drift corresponds to the speed from the equilibrium of the drive and drag forces while the rotational drift corresponds to the circular rotation. Similar methods have been used to understand macroscale phenomena emerging from collectives of microscopic elements [101] and to model particle motion in active matter [165].

Mitigating the effects of the arena's fixed boundaries in both experiments and simulations presented a significant design challenge. BOBbots can persist along the boundary or in corners, affecting system dynamics by, for example, enabling aggregates to form where they would not have otherwise or hindering multiple aggregates from integrating. To address these issues, uniform airflow was employed to gently repel BOBbots away from the boundary and similar effects were implemented in simulation.

11.2.3 Clustering Dynamics in BOBbots and Theory

Since the critical elements of the Markov chain $\mathcal{M}_A$ can be physically embodied by robots as simple as our BOBbots, to test if $\mathcal{M}_A$ could quantitatively capture collective dynamics, we next investigated the degree to which collectives of BOBbots aggregate as a function of their peripheral magnet strength F_{M0} in both robotic experiments and DEM simulations. (For convenience, F_{M0} is normalized by the gravity of Earth $g = 9.81$ m/s^2 when using the unit of gram.) The experimental protocol begins with placing magnets of a particular strength F_{M0} into the BOBbots' peripheral slots. The BOBbots are positioned and oriented randomly in a rectangular arena and are then actuated uniformly for a fixed time during which the BOBbots' positions and the size of the largest connected component are tracked (Figure 71a–c). These trials are conducted for several F_{M0} values with repetition. We followed the same protocol in simulations.

In experiment and DEM simulation, we observe an abrupt, rapid rise and then saturation in the size N_{MC} of the largest connected component as the magnetic attraction F_{M0} increases (Figure 71d). These curves resemble those in Figure 69d, with the magnetization F_{M0} playing a role analogous to the bias parameter λ. Given this correspondence, we explored whether the equilibrium SOPS model could be used

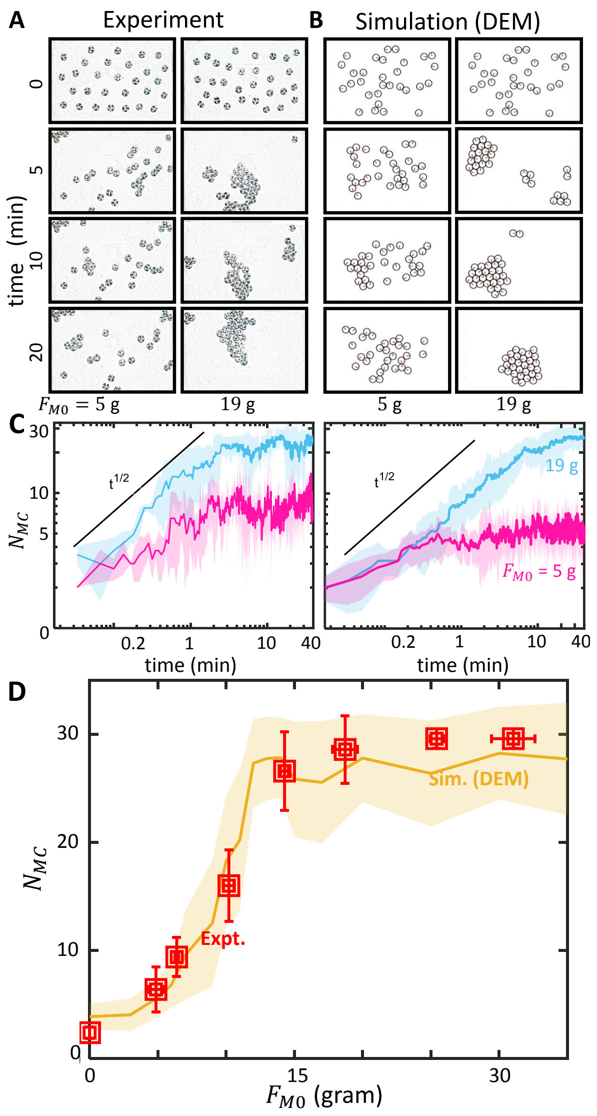


Figure 71. Evolution of BOBbot Clusters. (a) Time evolution snapshots of both experiment and (b) simulation for a system of 30 BOBbots with different magnet strengths: $F_{M0} = 5$ g (left) where we observe dispersion, and $F_{M0} = 19$ g (right) where we observe aggregation. (c) Time evolutions of the size of the largest component N_{MC} in experiment and simulation for a system of 30 BOBbots with $F_{M0} = 5$ g (magenta) and $F_{M0} = 19$ g (blue). (d) Scaling of cluster size vs. magnetic strength for a system of 30 BOBbots showing an increase in N_{MC} as the magnet strength F_{M0} increases. The yellow plot line shows the mean and standard deviation of N_{MC} in the 150 simulation runs for each magnetic strength F_{M0} between 1–35 g, with a step size of 1 g. Experimental data is shown in red with error bars showing the standard deviation of largest cluster size N_{MC} and the uncertainty of F_{M0} due to empirical measurement.

to make quantifiable predictions in the robot experiments. First, we designed a test to examine how force and λ scale. Recall that in the SOPS algorithm, the force acting on each particle is proportional to λ^e, where e is the particle's current number of neighbors. In the experiments, BOBbots cannot count their neighbors, but the magnets are expected to provide a similar force that also increases geometrically when more magnets are engaged.

To estimate the relationship between force and λ, we investigate the rate at which a BOBbot loses or gains neighbors over a fixed amount of time. Viewing a BOBbot's completion of half its circular motion as analogous to a particle moving to a new lattice node in the SOPS algorithm and using this time interval to evaluate the transition, simulation data shows that a BOBbot's transition probability from having a higher number of neighbors e to a lower number e' closely follows the algorithm's $P(\sigma, \tau) \propto \min(1, \lambda^{e'-e})$ transition probabilities (Figure 72a). Further, we evaluated the BOBbots' effective bias parameter λ_{eff} as a function of F_{M0} and found an exponential relation $\lambda_{\text{eff}} = \exp(\beta F_{M0})$, where β is a constant representing inverse temperature (Figure 72b). The BOBbots' transition probabilities can then be approximated as $P(\sigma, \tau) = \exp(-\beta(\epsilon_e - \epsilon_{e'}))$, where β is the inverse temperature of the system and $\epsilon_e = e \cdot F_{M0}$ can be interpreted as the energy contributed by a BOBbot's e neighbors.

With the relation between F_{M0} and λ_{eff} established, we next compare the aggregation behaviors exhibited by the SOPS algorithm and the BOBbot ensembles. Figure 72c shows the fraction of particles/BOBbots in the largest component N_{MC}/n observed in both the SOPS algorithm $\mathcal{M}_A$ and BOBbot simulations after converting with respect to λ_{eff}; the algorithm does indeed capture the maximum cluster fraction observed in the simulations. Notably, the aggregated and dispersed regimes in λ-space established by Theorem 11.2.3 provide a rigorous understanding of these BOBbot

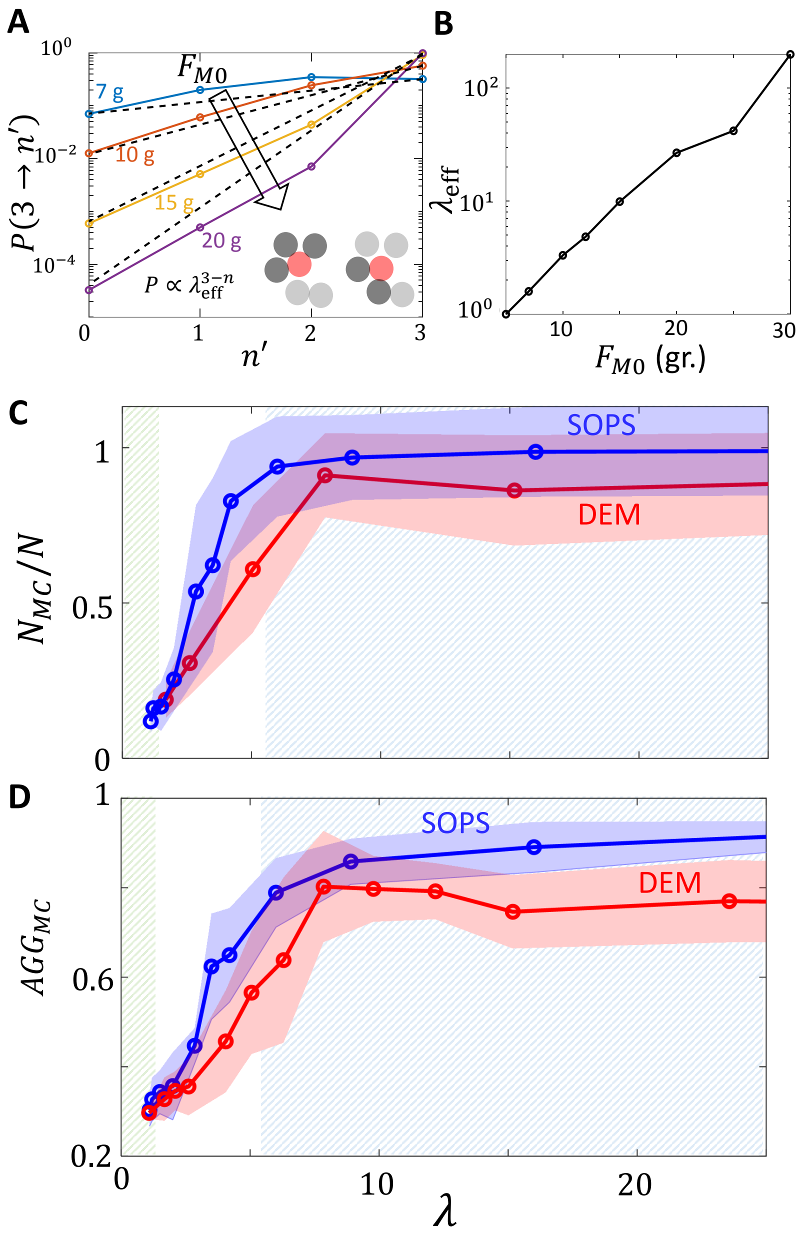


Figure 72. Algorithmic Interpretation of BOBbot Clustering. (a) A diagram showing how the effective bias parameter λ_{eff} is evaluated from the DEM simulation. (b) The dependence of λ_{eff} on the magnetic attraction force F_{M0}. The (c) maximum cluster fraction N_{MC}/n and (d) aggregation metric AGG_{MC} for different values of λ in both the SOPS algorithm $\mathcal{M}_A$ (blue) and physical simulations (red). The green and blue shaded regions show the dispersed and aggregated regimes proved from theory, respectively.

collective behaviors. For instance, the proven dispersed regime $0.98 < \lambda < 1.02$ gives a clear explanation for why agents will not aggregate even in the presence of mutual attraction. Further, it also helps establish the magnitude of attraction needed to saturate the aggregation.

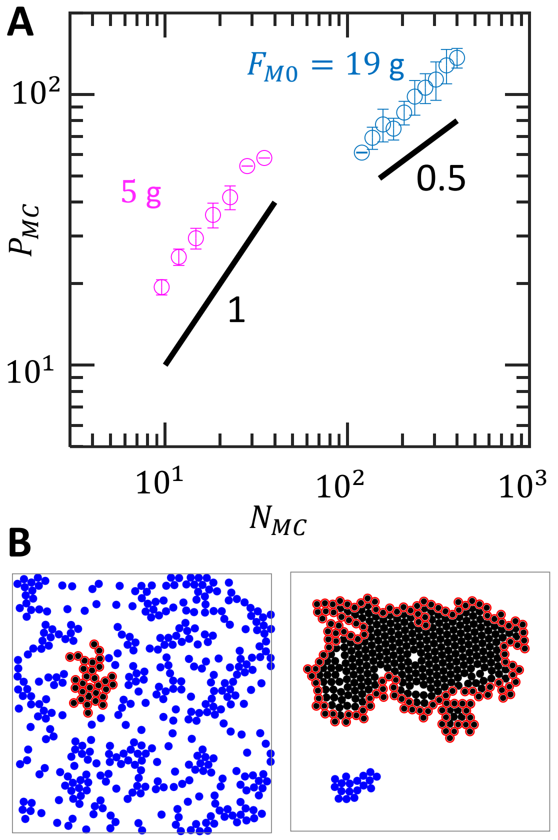


Figure 73. Perimeter Scaling of BOBbot Clusters. (a) Log-log plot showing the scaling relationship between the largest component's size N_{MC} and perimeter P_{MC} in number of BOBbots for simulated systems of 400 BOBbots with $F_{M0} = 5$ g (magenta) and 19 g (cyan) for fixed boundary conditions. Each data point is the average of 20 simulations. While the SOPS predicts a scaling power of 0.5 for the aggregated case (cyan), the data shows a slightly larger — but still sublinear — power of 0.66 ± 0.07. (b) Final snapshot of the collective motion of 400 BOBbots with $F_{M0} = 5$ g (left) and 19 g (right). BOBbots shown in black belong to the largest connected component; those outlined in red are on its periphery.

We additionally test the SOPS prediction that the maximum cluster should not only be large but also compact, occupying a densely packed region. The results from separation [31] that we apply here for aggregation suggest the following relationship between the size of the largest component N_{MC} and its perimeter P_{MC}. In dispersed configurations, P_{MC} should scale linearly with N_{MC}, meaning that most BOBbots lie on the periphery of their components. In aggregated configurations, however, P_{MC} should scale as $N_{MC}^{1/2}$, approximating the minimal perimeter for the same number of BOBbots by at most a constant factor. We test these scaling relationships in simulations with 400 BOBbots (Figure 73) and find that the theory's predictions hold in the dispersed regime; however, the 0.66 ± 0.07 sublinear scaling power for the aggregated case is slightly higher than the theory's prediction of 0.5. This discrepancy may in part be due to boundary and finite-size effects — in fact, DEM simulations with periodic boundaries show a scaling power of 0.59 ± 0.18 that is closer to the SOPS theory — but is also affected by non-reversibility inherent in the BOBbots' circular trajectories. To make a quantitative comparison that captures when components are both large and compact, we track $AGG_{MC} = N_{MC}/(k_0 P_{MC}\sqrt{n})$, where k_0 is a scaling constant defined such that $AGG_{MC} = 1$ when the system is optimally aggregated, achieving the minimum possible perimeter. Physically, AGG_{MC} is reminiscent of surface tension for which energy minimization leads to a smaller interface (in our setting, smaller perimeter P_{MC}), yielding an AGG_{MC} closer to 1. We obtain agreement between the SOPS and DEM simulations with respect to this metric as well (Figure 72d), further validating the theory's prediction, though the DEM simulations yield slightly smaller AGG_{MC} than the SOPS algorithm for large λ.

11.2.4 Object Transport in the Aggregated Phase

Encouraged by the close connections between the physical system and the underlying theoretical model, we sought to test whether aggregated BOBbots could collectively accomplish a task. In particular, could an aggregated BOBbot collective "recognize" the presence of a non-robot impurity in its environment and cooperatively expel it from the system? Typically, such collective transport tasks — e.g., the cooperative transport of food by ants [82, 192] — either manifest from an order-disorder transition or rely heavily on conformism between agents for concerted effort and alignment of forces. With our BOBbot collectives, we instead aim to accomplish transport via simple mechanics and physical interactions emergently controlling global behavior without any complex control, communication, or computation.

By maintaining a high magnetic attraction F_{M0}, we remain in the aggregated regime where most BOBbots connect physically and can cumulatively push against untethered impurities (e.g., a box or disk) introduced in the system (Figure 74a). The BOBbot collective's constant stochastic reconfiguration grants it the ability to envelop, grasp, and dislodge impurities as their individual forces additively overcome the impurities' friction, leading to large displacement in the aggregated regime (Figure 74b, right) with a median displacement of 7.9 cm over 12 minutes. On the contrary, we find that systems with weak magnetic attraction (i.e., those in the dispersed regime) can typically only achieve small impurity displacement (Figure 74b, left) with a median displacement of 0.9 cm over 12 minutes. We observe infrequent anomalies in which dispersed collectives achieve larger displacement than aggregated ones, but these outliers arise from idiosyncrasies of our rudimentary robots (e.g., an aggregated cluster

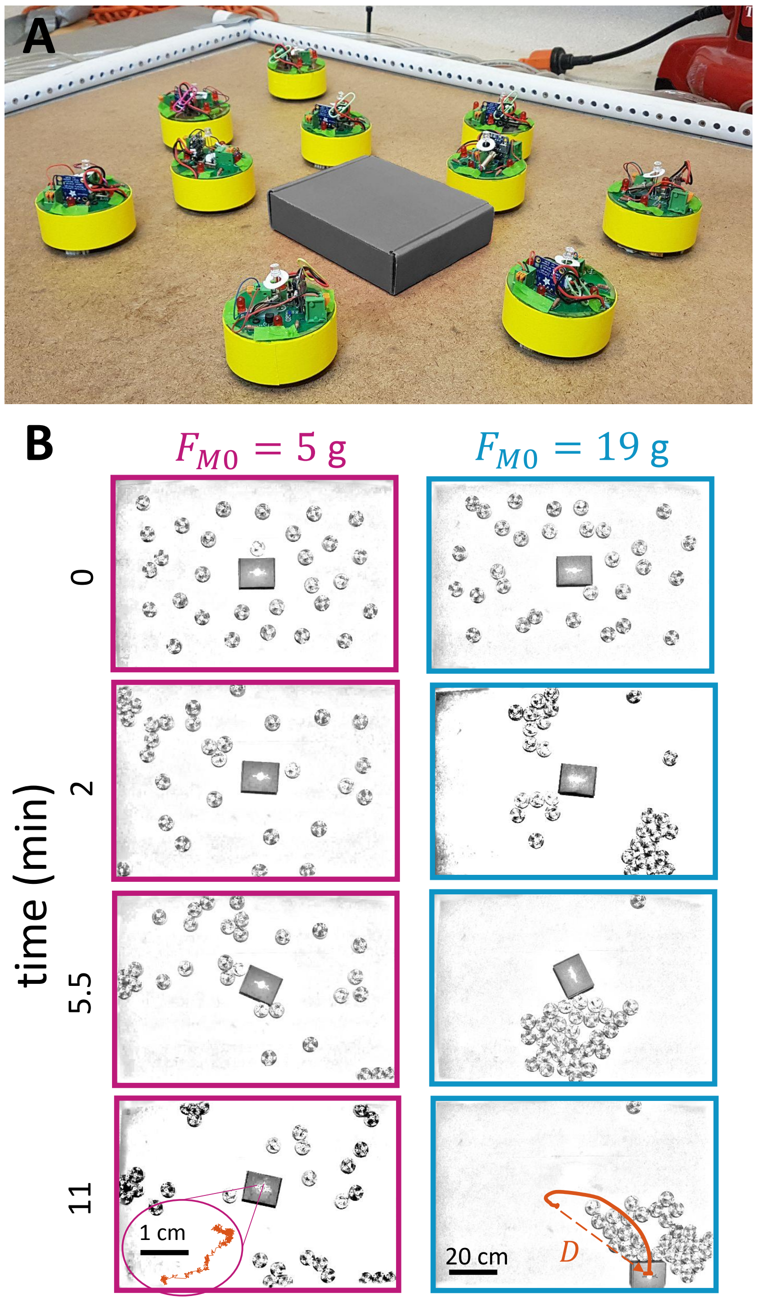


Figure 74. Object Transport Using Aggregation. (a) Schematic of the experimental setup. (b) Time evolution snapshots of box transport by a system of 30 BOBbots with magnet strength $F_{M0} = 5$ g and 19 g. The box has a mass of 60 g. The final panel shows the object's complete trajectory, where D denotes the Euclidean distance of the final displacement.

of BOBbots may continuously rotate in place without coming in contact with an impurity due to the BOBbots' individual orientations in the aggregate).

Characterizing the impurity's transport dynamics as mean-squared displacement over time $\langle r^2(\tau)\rangle = v\tau^\alpha$ reveals further disparities between the aggregated and dispersed BOBbot collectives (Figure 75a). On a log-log plot, the intercept indicates $\log(v)$, where v is the characteristic speed of the impurity's transport; we observe that in all but one fringe case the strongly attractive collectives achieve transport that is orders of magnitude faster than those of the weakly attractive ones (Figure 75b). The slope of each trajectory indicates the exponent α that characterizes transport as subdiffusive ($\alpha < 1$), diffusive ($\alpha = 1$), or superdiffusive ($\alpha > 1$). While all the strongly attractive collectives immediately achieve nearly ballistic transport (with $\alpha = 1.85 \pm 0.11$ for $\tau < 20$ s) indicating rapid onset of cluster formation and pushing, the weakly attractive collectives initially exhibit mostly subdiffusive transport (with $\alpha = 0.89 \pm 0.56$ for $\tau < 20$ s) caused by intermittent collisions from the dispersed BOBbots (Figure 75c). When the slight heterogeneous distribution of the dispersed BOBbots remains unchanged for a sufficiently long time, the accumulation of displacement in a persistent direction can cause a small drift, leading to ballistic transport at a longer time scale. Nonetheless, the transport speeds achieved by the dispersed collectives are two orders of magnitude smaller than those of the strongly attractive ones.

Simulations of impurity transport reproduce the experimental results (Figure 75b, inset), including the rare anomalies. Seven of the 100 simulations of weakly attractive collectives succeeded in transporting the impurity to the arena boundary at slow speeds while 76 of the 100 simulations of strongly attractive collectives did so ballistically. The remaining 24 simulations of attractive collectives that did not achieve ballistic

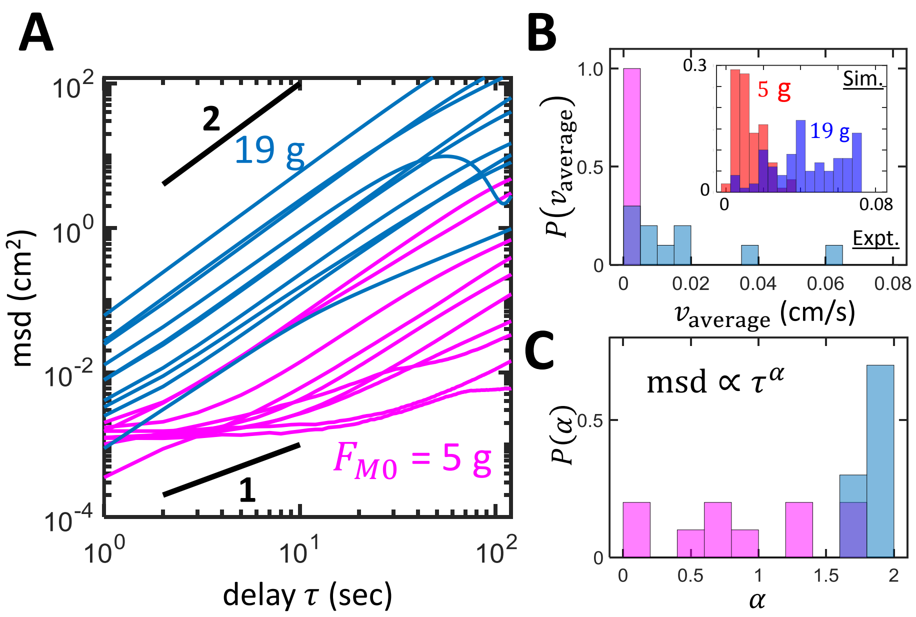


Figure 75. Mean-Squared Displacement of Object Transport. (a) Mean-squared displacement of the box over time in log-log scale for collectives with $F_{M0} = 5$ g (magenta) and 19 g (blue). (b) Distribution of the average speed, calculated as the final displacement D (as shown in Figure 74b) divided by total time. Inset: Simulation results for the overall transport speed. The two peaks for $F_{M0} = 19$ g correspond to pushing to the edges and corners. (c) Distributions of the mean-squared displacement exponent α at short time scale $\tau < 20$ s.

transport consistently formed an aggregate that never came into contact with the impurity. We found that disaggregating established aggregates by introducing time periods with no attraction enabled them to dissolve and reform for another attempt at transport. Using different disaggregating sequences, the attractive collectives achieved ballistic transport in 15–20% more simulations than without disaggregating.

11.2.5 Discussion

In this section, we use mathematical ideas from distributed computing and statistical physics to create task-oriented cohesive granular media composed of simple interacting robots called BOBbots. As predicted by the theory, the BOBbots aggregate compactly with stronger magnets (corresponding to large bias parameter λ) and disperse with weaker magnets (or small λ). Simulations capturing the physics governing the BOBbots' motions and interactions further confirm the predicted phase change with larger numbers of BOBbots. The collective transport task then demonstrates the utility of the aggregation algorithm.

There are several noteworthy aspects of these findings. First, the *theoretical framework* of the underlying SOPS model can be generalized to allow many types of relaxations to its assumptions, provided its dynamics remain reversible and model a system at thermal equilibrium. For example, noting that the probability that a robot with e neighbors detaches may not scale precisely as λ^{-e} as suggested by the Boltzmann weights, we can generalize algorithm $\mathcal{M}_A$ to be more sensitive to small variations in these weights: the proofs establishing the two distinct phases can be shown to extend to this setting, provided the probabilities p_e of detaching from e neighbors satisfy $c_1\lambda^{-e} \leq p_e \leq c_2\lambda^{-e}$, for constants $c_1, c_2 > 0$.

The *robustness* of the local, stochastic algorithms makes the macro-scale behavior of the collective resistant to many types of idiosyncrasies inherent in the BOBbots, including bias in the directions of their movements, the continuous nature of their trajectories, and nonuniformity in their speeds and magnet strengths. Moreover, our algorithms are inherently self-stabilizing due to their memoryless, stateless nature, always converging to a desired system configuration — overcoming faults and other perturbations in the system — without the need for external intervention. In our context, the algorithm will naturally continue to aggregate, even as some robots may fail or the environment is perturbed.

Moreover, we find that the *nonequilibrium dynamics* of the BOBbots are largely captured by the theoretical models that we analyze at thermal equilibrium, which is in agreement with the findings of Stenhammar et al. [183]. For example, in addition to visually observing the phase change as the magnetic strengths increase, we are able to test precise predictions about the size and perimeter of the largest connected components based on the formal definitions of aggregation and dispersion from the SOPS model. We additionally use simulations to study the transition probability of a BOBbot from having e neighbors to having e' neighbors to see if the magnetic interactions conform to the theory, and indeed we see a geometric relation decrease in the probability of moving as we increase the number of neighbors, as predicted. The resultant correspondence between the magnetic attraction and effective bias in the algorithm confirms a quantitative connection between the physical world and the abstract algorithm.

In summary, the framework presented here using provable distributed, stochastic algorithms to inspire the design of robust, simple systems of robots with limited computational capabilities seems quite general. It also allows one to leverage the extensive amount of work on distributed and stochastic algorithms, and equilibrium models and proofs in guiding the tasks of inherently out of equilibrium robot swarms. Preliminary results show that we likely can achieve other basic tasks such as alignment, separation (or speciation), and flocking through a similar principled approach. We note that exploiting physical embodiment with minimal computation seems a critical step in scaling collective behavior to encompass many cutting edge settings, including micro-sized devices that can be used in medical applications and cheap, scalable devices for space and terrestrial exploration. Additionally, we plan to further study the important interplay between equilibrium and nonequilibrium dynamics to better solidify these connections and to understand which relaxations remain in the same universality classes.

Chapter 12

CONCLUSION

This dissertation advances programmable matter's modeling, algorithm design and analysis, and applications to swarm robotic and active matter systems. In Chapter 2, we presented the canonical amoebot model as a complete reformulation of the amoebot model of programmable matter, emphasizing its handling of concurrency and hierarchy of assumption variants. We then presented two black box enhancements of existing amoebot algorithms, extending sequential, energy-agnostic algorithms that satisfy certain conventions to the concurrent, energy-constrained setting without sacrificing correct behavior (Chapters 3 and 4). Chapters 5–10 detailed two complementary approaches to algorithm design and analysis: one that relies on amoebot memory and communication to solve problems like leader election, object coating, and convex hull formation; and another that uses only biased random decisions to solve problems like compression, shortcut bridging, and separation. Compared to the stateful algorithms of Chapters 5–7, the stochastic algorithms of Chapters 8–10 are inherently self-stabilizing and robust to errors. Chapter 11 detailed how these stochastic algorithms can be readily adapted to two experimental systems, supersmarticles and BOBbots, to provably characterize collective behaviors of phototaxing, aggregation, dispersion, and collective transport. We conclude with several new and exciting directions for future research.

Fault Tolerant Programmable Matter. Chapters 2–3 focused on bridging the gap between existing models of active programmable matter and the constraints of programmable matter hardware with respect to concurrency and energy usage. Another key area of improvement is the modeling and mitigation of *faulty behavior*, i.e., when individual modules of programmable matter may *crash* or exhibit *malicious (Byzantine)* behavior. While crash failures have been considered under the amoebot model in specific cases — e.g., for energy distribution in Chapter 4, for shape formation in [67], and for the stochastic algorithms of Chapters 8–10 — the amoebot model does not yet have a formal fault model for crash failures. Faults also pose a major problem for our lock-based approach to concurrency control (Chapter 3), as the canonical amoebot model's Lock operation is no longer deadlock-free in the presence of crash failures. Finally, nearly all existing design and analysis of algorithms for programmable matter assume the *cooperative* nature of the participating modules. Byzantine failures thus force a paradigm shift to *competitive* dynamics, which are significantly more complex.

Asynchronous Algorithms for Programmable Matter. The canonical amoebot model of Chapter 2 introduced a unifying and formal framework for algorithm design and adversarial activation models. The majority of the results in this dissertation assumed a simplified fair sequential setting where the adversary could activate at most one amoebot per time and was forced to activate every amoebot infinitely often. Shifting towards the concurrent setting, the asynchronous hexagon formation algorithm of Section 2.5 and the concurrency control protocol of Chapter 3 gave two complementary sets of sufficient conditions for designing correct algorithms under unfair asynchronous adversaries, the most general of all adversarial activation models. The first approach stems from the analysis of the hexagon formation algorithm in Section 2.5: if an algorithm is correct under any unfair sequential adversary, its enabled actions remain enabled despite concurrent action executions, and its enabled actions execute successfully and invariant from their sequential executions, then the algorithm can immediately be shown to be correct under any unfair asynchronous adversary. However, these conditions are relatively strict. The second approach requires a compliant algorithm to be correct under any unfair sequential adversary and to satisfy the conventions of the concurrency control protocol. However, as was discussed in Section 3.4, it is an open question at this time to find any algorithm involving movement that satisfies the monotonicity convention. Thus, while we are hopeful that these two approaches can be applied to the analysis of existing and future algorithms under asynchronous adversaries in the canonical amoebot model, it remains an open problem to find less restrictive sufficient conditions or to develop other general approaches to the design and analysis of asynchronous algorithms.

Bridging Algorithmic Theory to the Mechanics of Physical Systems. The physics of local interactions provide a powerful toolkit for programming task-oriented collectives across scales without requiring sophisticated hardware or traditional computation. Chapter 11 described a two-pronged approach to programming physical ensembles by bridging the rigorous analysis of algorithmic theory to the physics of damped driven systems. Remarkably, in the case of the BOBbots of Section 11.2, the lattice-based distributed algorithms leveraging *equilibrium statistical physics* quantitatively captured the *nonequilibrium dynamics* of an analog robot swarm. This approach opens many exciting directions for interdisciplinary research, especially in the areas of social insects and colloidal robots [132, 196].

Distributed Algorithms for the Dynamics of Movement. Programmable matter is just one example of a system of cooperating agents whose communication links change over time as a result of active (self-directed) motion; practical examples include social insect colonies, swarm robotics, and autonomous vehicular networks. Dynamic networks have received some attention from the distributed computing community over the last decade (e.g., under the *time-varying graph model* [35] and for *peer-to-peer networks* [11]), especially in the area of self-stabilization when topological changes are sparse in both time and space [4]. However, unlike dynamics caused by rare transient failures — as in existing works on self-stabilization — moving systems shift their topologies rapidly but locally. How do these *dynamics of movement* compare with existing treatments of dynamic networks? What fundamental behaviors (such as *broadcast* and *leader election*) can be achieved in these models, especially in the setting where individual processes are very limited (e.g., in terms of memory)? As one concrete example problem, is it possible to adapt a stateless form of broadcast known as *amnesiac flooding* [109] under dynamics of movement? Studying systems of moving, computationally limited agents is a natural generalization of programmable matter research, and has the potential to form rich connections between these distributed computing subfields.

REFERENCES

[1] Alan Aderem and David M. Underhill. “Mechanisms of phagocytosis in macrophages”. In: *Annual Review of Immunology* 17.1 (1999), pp. 593–623.

[2] Mayank Agrawal, Isaac R. Bruss, and Sharon C. Glotzer. “Tunable emergent structures and traveling waves in mixtures of passive and contact-triggered-active particles”. In: *Soft Matter* 13.37 (2017), pp. 6332–6339.

[3] Selim G. Akl and Kelly A. Lyons. *Parallel Computational Geometry.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[4] Karine Altisen et al. “Brief Announcement: Self-Stabilizing Systems in Spite of High Dynamics”. In: *Symposium on Principles of Distributed Computing.* PODC 2020. 2020, pp. 227–229.

[5] Karine Altisen et al. *Introduction to Distributed Self-Stabilizing Algorithms.* Ed. by Michel Raynal. Vol. 8. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019.

[6] Bruno Andreotti, Yoël Forterre, and Olivier Pouliquen. *Granular media: between fluid and solid.* Cambridge University Press, 2013.

[7] Marta Andrés Arroyo et al. “A Stochastic Approach to Shortcut Bridging in Programmable Matter”. In: *DNA Computing and Molecular Programming.* DNA 2017. 2017, pp. 122–138.

[8] Marta Andrés Arroyo et al. “A Stochastic Approach to Shortcut Bridging in Programmable Matter”. In: *Natural Computing* 17.4 (2018), pp. 723–741.

[9] Dana Angluin, James Aspnes, and David Eisenstat. “Fast computation by population protocols with a leader”. In: *Distributed Computing* 21.3 (2008), pp. 183–199.

[10] Dana Angluin et al. “Computation in networks of passively mobile finite-state sensors”. In: *Distributed Computing* 18.4 (2006), pp. 235–253.

[11] John Augustine, Gopal Pandurangan, and Peter Robinson. “Distributed Algorithmic Foundations of Dynamic Networks”. In: *ACM SIGACT News* 47 (1 2016), pp. 69–98.

[12] Baruch Awerbuch. “Complexity of Network Synchronization”. In: *Journal of the ACM* 32.4 (1985), pp. 804–823.

[13] Evangelos Bampas et al. “Almost Optimal Asynchronous Rendezvous in Infinite Multidimensional Grids”. In: *Distributed Computing.* DISC 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 297–311.

[14] Eduardo Mesa Barrameda, Shantanu Das, and Nicola Santoro. “Deployment of Asynchronous Robotic Sensors in Unknown Orthogonal Environments”. In: *Algorithmic Aspects of Wireless Sensor Networks.* ALGOSENSORS 2008. 2008, pp. 125–140.

[15] Palina Bartashevich, Doreen Koerte, and Sanaz Mostaghim. “Energy-saving decision making for aerial swarms: PSO-based navigation in vector fields”. In: *2017 IEEE Symposium Series on Computational Intelligence.* SSCI 2017. 2017, pp. 1–8.

[16] Alexander I. Barvinok. *Combinatorics and complexity of partition functions.* Vol. 30. Algorithms and Combinatorics. Springer International Publishing, 2016.

[17] Alexander I. Barvinok and Pablo Soberón. “Computing the partition function for graph homomorphisms with multiplicities”. In: *Journal of Combinatorial Theory, Series A* 137 (2016), pp. 1–26.

[18] Roland Bauerschmidt et al. “Lectures on self-avoiding walks”. In: *Probability and Statistical Physics in Two and More Dimensions* 15 (2012), pp. 395–476.

[19] R. J. Baxter, I. G. Enting, and S. K. Tsang. “Hard-Square Lattice Gas”. In: *Journal of Statistical Physics* 22 (1980), pp. 465–489.

[20] Levent Bayindir. “A review of swarm robotics tasks”. In: *Neurocomputing* 172 (2016), pp. 292–321.

[21] Rida A. Bazzi and Joseph L. Briones. “Stationary and Deterministic Leader Election in Self-organizing Particle Systems”. In: *Stabilization, Safety, and Security of Distributed Systems.* SSS 2019. 2019, pp. 22–37.

[22] Petra Berenbrink, George Giakkoupis, and Peter Kling. “Optimal Time and Space Leader Election in Population Protocols”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing.* STOC 2020. 2020, pp. 119–129.

[23] Prateek Bhakta, Sarah Miracle, and Dana Randall. “Clustering and mixing times for segregation models on $\mathbb{Z}^2$”. In: *Proceedings of the Twenty-fifth Annual*

ACM-SIAM Symposium on Discrete Algorithms. SODA 2014. 2014, pp. 327–340.

[24] Dapeng Bi et al. "Jamming by shear". In: *Nature* 480.7377 (2011), pp. 355–358.

[25] Antonio Blanca et al. "Phase coexistence for the hard-core model on $\mathbb{Z}^2$". In: *Combinatorics, Probability and Computing* 28.1 (2019), pp. 1–22.

[26] Vincenzo Bonifaci, Kurt Mehlhorn, and Girish Varma. "Physarum can compute shortest paths". In: *Journal of Theoretical Biology* 309 (2012), pp. 121–133.

[27] Manuele Brambilla et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1 (2013), pp. 1–41.

[28] Federico Cali, Marco Conti, and Enrico Gregori. "IEEE 802.11 protocol: design and performance evaluation of an adaptive backoff mechanism". In: *IEEE Journal on Selected Areas in Communications* 18.9 (2000), pp. 1774–1786.

[29] Scott Camazine et al. "House-Hunting by Honey Bee Swarms: Collective Decisions and Individual Behaviors". In: *Insectes Sociaux* 46.4 (1999), pp. 348–360.

[30] Jason Campbell, Padmanabhan Pillai, and Seth Copen Goldstein. "The Robot is the Tether: Active, Adaptive Power Routing for Modular Robots With Unary Inter-robot Connectors". In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IROS 2005. IEEE, 2005, pp. 4108–4115.

[31] Sarah Cannon et al. "A Local Stochastic Algorithm for Separation in Heterogeneous Self-Organizing Particle Systems". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. APPROX/RANDOM 2019. 2019, 54:1–54:22.

[32] Sarah Cannon et al. "A Markov Chain Algorithm for Compression in Self-Organizing Particle Systems". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC 2016. 2016, pp. 279–288.

[33] John I. Capetanakis. "Tree algorithms for packet broadcast channels". In: *IEEE Transactions on Information Theory* 25.5 (1979), pp. 505–515.

[34] Pietro Caputo et al. ""Zero" temperature stochastic 3D Ising model and dimer covering fluctuations: a first step towards interface mean curvature motion". In: *Communications of Pure and Applied Mathematics* 64 (2011), pp. 778–831.

[35] Arnaud Casteigts et al. “Time-Varying Graphs and Dynamic Networks”. In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (2012), pp. 387–408.

[36] Claudio Castellano, Santo Fortunato, and Vittorio Loreto. “Statistical physics of social dynamics”. In: *Reviews of Modern Physics* 81.2 (2009), pp. 591–646.

[37] Cameron Chalk et al. “Freezing Simulates Non-freezing Tile Automata”. In: *DNA Computing and Molecular Programming.* DNA 2018. 2018, pp. 155–172.

[38] Arturo Chavoya and Yves Duthen. “Using a Genetic Algorithm to Evolve Cellular Automata for 2D/3D Computational Development”. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation.* GECCO 2006. New York, NY, USA: ACM, 2006, pp. 231–232.

[39] Ho-Lin Chen et al. “Fast Algorithmic Self-assembly of Simple Shapes Using Random Agitation”. In: *DNA Computing and Molecular Programming.* 2014, pp. 20–36.

[40] Moya Chen, Doris Xin, and Damien Woods. “Parallel computation using active self-assembly”. In: *Natural Computing* 14 (2015), pp. 225–250.

[41] Yen-Ru Chin, Jui-Ting Tsai, and Ho-Lin Chen. “A minimal requirement for self-assembly of lines in polylogarithmic time”. In: *Natural Computing* 17.4 (2018), pp. 743–757.

[42] Gregory S. Chirikjian. “Kinematics of a metamorphic robotic system”. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation.* Vol. 1. ICRA 1994. 1994, pp. 449–455.

[43] Pavel Chvykov et al. “Low rattling: A predictive principle for self-organization in active collectives”. In: *Science* 371.6524 (2021), pp. 90–95.

[44] Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. “Asynchronous Robots on Graphs: Gathering”. In: *Distributed Computing by Mobile Entities: Current Research in Moving and Computing.* Cham: Springer, 2019, pp. 184–217.

[45] Nikolaus Correll and Alcherio Martinoli. “Modeling and designing self-organized aggregation in a swarm of miniature robots”. In: *The International Journal of Robotics Research* 30.5 (2011), pp. 615–626.

[46] Eric I. Corwin, Heinrich M. Jaeger, and Sidney R. Nagel. “Structural signature of jamming in granular media”. In: *Nature* 435.7045 (2005), pp. 1075–1078.

[47] Shantanu Das et al. “Autonomous Mobile Robots with Lights”. In: *Theoretical Computer Science* 609 (2016), pp. 171–184.

[48] Shantanu Das et al. “The Power of Lights: Synchronizing Asynchronous Robots Using Visible Bits”. In: *IEEE 32nd International Conference on Distributed Computing Systems*. ICDCS 2012. 2012, pp. 506–515.

[49] Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. “Mutual Exclusion for Asynchronous, Anonymous, Dynamic, Constant-Size Memory Message Passing Systems”. Manuscript in preparation. 2021.

[50] Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. “The Canonical Amoebot Model: Algorithms and Concurrency Control”. Manuscript in preparation. 2021.

[51] Joshua J. Daymude, Andréa W. Richa, and Jamison W. Weber. “Bio-Inspired Energy Distribution for Programmable Matter”. In: *International Conference on Distributed Computing and Networking 2021*. ICDCN 2021. 2021, pp. 86–95.

[52] Joshua J. Daymude et al. “Computing by Programmable Particles”. In: *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*. Cham: Springer, 2019, pp. 615–681.

[53] Joshua J. Daymude et al. “Convex Hull Formation for Programmable Matter”. In: *Proceedings of the 21st International Conference on Distributed Computing and Networking*. ICDCN 2020. 2020, 2:1–2:10.

[54] Joshua J. Daymude et al. “Improved Leader Election for Self-Organizing Programmable Matter”. In: *Algorithms for Sensor Systems*. ALGOSENSORS 2017. 2017, pp. 127–140.

[55] Joshua J. Daymude et al. “On the Runtime of Universal Coating for Programmable Matter”. In: *Natural Computing* 17.1 (2018), pp. 81–96.

[56] A. Deblais et al. “Boundaries Control Collective Dynamics of Inertial Self-Propelled Robots”. In: *Physical Review Letters* 120.18 (2018), p. 188002.

[57] Daryl DeFord, Moon Duchin, and Justin Solomon. “Recombination: A family of Markov chains for redistricting”. Preprint available online at https://arxiv.org/abs/1911.05725. 2019.

[58] Zahra Derakhshandeh et al. “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*. 2015, 21:1–21:2.

[59] Zahra Derakhshandeh et al. “Brief Announcement: Amoebot - A New Model for Programmable Matter”. In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA 2014. 2014, pp. 220–222.

[60] Zahra Derakhshandeh et al. “Leader Election and Shape Formation with Self-Organizing Programmable Matter”. In: *DNA Computing and Molecular Programming*. 2015, pp. 117–132.

[61] Zahra Derakhshandeh et al. “Universal Coating for Programmable Matter”. In: *Theoretical Computer Science* 671 (2017), pp. 56–68.

[62] Zahra Derakhshandeh et al. “Universal Shape Formation for Programmable Matter”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 2016, pp. 289–299.

[63] Andreas Deutsch and Sabine Dormann. *Cellular Automaton Modeling of Biological Pattern Formation*. 2nd. Basel, Switzerland: Birkhäuser Basel, 2017.

[64] Peter Devreotes. “Dictyostelium discoideum: A Model System for Cell-Cell Interactions in Development”. In: *Science* 245.4922 (1989), pp. 1054–1058.

[65] Giuseppe A. Di Luna et al. “Shape Formation by Programmable Particles”. In: *21st International Conference on Principles of Distributed Systems*. Vol. 95. OPODIS 2017. 2018, 31:1–31:16.

[66] Giuseppe A. Di Luna et al. “Shape Formation by Programmable Particles”. In: *Distributed Computing* 33 (2020), pp. 69–101.

[67] Giuseppe Antonio Di Luna et al. “Line Recovery by Programmable Particles”. In: *Proceedings of the 19th International Conference on Distributed Computing and Networking*. ICDCN 2018. New York, NY, USA: ACM, 2018, 4:1–4:10.

[68] Giuseppe Antonio Di Luna et al. “Mutual Visibility by Luminous Robots Without Collisions”. In: *Information and Computation* 254.3 (2017), pp. 392–418.

[69] Mohamadou Diallo et al. “Scalable 2D Convex Hull and Triangulation Algorithms for Coarse Grained Multicomputers”. In: *Journal of Parallel and Distributed Computing* 56.1 (1999), pp. 47–70.

[70] Edsger W. Dijkstra. “Self-Stabilizing Systems in Spite of Distributed Control”. In: *Communications of the ACM* 17.11 (1974), pp. 643–644.

[71] Shlomi Dolev et al. “Ameba-Inspired Self-Organizing Particle Systems”. Workshop paper at Biological Distributed Algorithms 2013. Available online at https://arxiv.org/abs/1307.4259. 2013.

[72] Shlomi Dolev et al. “In-vivo energy harvesting nano robots”. In: *2016 IEEE International Conference on the Science of Electrical Engineering*. 2016, pp. 1–5.

[73] Marco Dorigo, Guy Theraulaz, and Vito Trianni. “Reflections on the future of swarm robotics”. In: *Science Robotics* 5.49 (2020), eabe4385.

[74] David Doty. “Theory of Algorithmic Self-Assembly”. In: *Communications of the ACM* 55.12 (2012), pp. 78–88.

[75] Hugo Duminil-Copin and Stanislav Smirnov. “The connective constant of the honeycomb lattice equals $\sqrt{2+\sqrt{2}}$”. In: *Annals of Mathematics* 175.3 (2012), pp. 1653–1665.

[76] Patrick Dymond, Jieliang Zhou, and Xiaotie Deng. “A 2-D parallel convex hull algorithm with optimal communication phases”. In: *Parallel Computing* 27.3 (2001), pp. 243–255.

[77] Karthik Elamvazhuthi and Spring Berman. “Mean-field models in swarm robotics: a survey”. In: *Bioinspiration & Biomimetics* 15.1 (2019), p. 015001.

[78] Yuval Emek et al. “Deterministic Leader Election in Programmable Matter”. In: *46th International Colloquium on Automata, Languages, and Programming*. ICALP 2019. 2019, 140:1–140:14.

[79] Peter Eshuis et al. “Phase diagram of vertically shaken granular matter”. In: *Physics of Fluids* 19.12 (2007), p. 123301.

[80] Nazim Fatès. “Solving the decentralised gathering problem with a reaction–diffusion–chemotaxis scheme”. In: *Swarm Intelligence* 4.2 (2010), pp. 91–115.

[81] Nazim Fatès and Nikolaos Vlassopoulos. “A Robust Scheme for Aggregating Quasi-Blind Robots in an Active Environment”. In: *International Journal of Swarm Intelligence Research* 3.3 (2012), pp. 66–80.

[82] Ofer Feinerman et al. “The physics of cooperative transport in groups of ants”. In: *Nature Physics* 14 (2018), pp. 683–693.

[83] William Feller. *An Introduction to Probability Theory and Its Applications*. Vol. 1. New York: Wiley, 1968.

[84] Eugene Fink and Derick Wood. *Restricted-Orientation Convexity*. Monographs in Theoretical Computer Science. An EATCS Series. Berlin, Germany: Springer-Verlag Berlin Heidelberg, 2004.

[85] Per-Olof Fjällström et al. “A sublogarithmic convex hull algorithm”. In: *BIT Numerical Mathematics* 30.3 (1990), pp. 378–384.

[86] Paola Flocchini. “Gathering”. In: *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*. Cham: Springer, 2019, pp. 63–82.

[87] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, eds. *Distributed Computing by Mobile Entities*. Switzerland: Springer International Publishing, 2019.

[88] Paola Flocchini et al. “Rendezvous with Constant Memory”. In: *Theoretical Computer Science* 621 (2016), pp. 57–72.

[89] Steven Fortune. “A sweepline algorithm for Voronoi diagrams”. In: *Algorithmica* 2.1 (1987), p. 153.

[90] Sacha Friedli and Yvan Velenik. *Statistical Mechanics of Lattice Systems: A Concrete Mathematical Introduction*. Cambridge: Cambridge University Press, 2018.

[91] Simon Garnier et al. “Self-Organized Aggregation Triggers Collective Decision Making in a Group of Cockroach-Like Robots”. In: *Adaptive Behavior* 17.2 (2009), pp. 109–133.

[92] Nicolas Gastineau et al. “Distributed Leader Election and Computation of Local Identifiers for Programmable Matter”. In: *Algorithms for Sensor Systems*. ALGOSENSORS 2018. 2019, pp. 159–179.

[93] Melvin Gauci et al. “Self-organized aggregation without computation”. In: *International Journal of Robotics Research* 33.8 (2014), pp. 1145–1161.

[94] Kyle Gilpin, Ara Knaian, and Daniela Rus. “Robot pebbles: One centimeter modules for programmable matter through self-disassembly”. In: *2010 IEEE International Conference on Robotics and Automation*. ICRA 2010. 2010, pp. 2485–2492.

[95] Robert Gmyr. “Distributed Algorithms for Overlay Networks and Programmable Matter”. PhD thesis. Paderborn University, 2017.

[96] Robert Gmyr et al. “Forming tile shapes with simple robots”. In: *Natural Computing* 19 (2020), pp. 375–390.

[97] Robert Gmyr et al. “Shape Recognition by a Finite Automaton Robot”. In: *43rd International Symposium on Mathematical Foundations of Computer Science*. MFCS 2018. 2018, 52:1–52:15.

[98] Daniel I. Goldman and Harry L. Swinney. “Signatures of glass formation in a fluidized bed of hard spheres”. In: *Physical Review Letters* 96.14 (2006), p. 145702.

[99] Seth C. Goldstein et al. “Beyond Audio and Video: Using Claytronics to Enable Pario”. In: *AI Magazine* 30.2 (2009), pp. 29–45.

[100] Seth Copen Goldstein, Jason D. Campbell, and Todd C. Mowry. “Programmable Matter”. In: *Computer* 38.6 (2005), pp. 99–101.

[101] Heiko Hamann. *Swarm Robotics: A Formal Approach*. Springer, 2018.

[102] Wilfred K. Hastings. “Monte Carlo Sampling Methods Using Markov Chains and Their Applications”. In: *Biometrika* 57.1 (1970), pp. 97–109.

[103] Tyler Helmuth, Will Perkins, and Guus Regts. “Algorithmic Pirogov-Sinai Theory”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. 2019, pp. 1009–1020.

[104] Arnaud Hemmerle, Matthias Schröter, and Lucas Goehring. “A cohesive granular material with tunable elasticity”. In: *Scientific Reports* 6.1 (2016), p. 35630.

[105] Gregory Herschlag et al. “Quantifying Gerrymandering in North Carolina”. In: *Statistics and Public Policy* 7.1 (2020), pp. 30–38.

[106] Lindsey Hines et al. “Soft Actuators for Small-Scale Robotics”. In: *Advanced Materials* 29.13 (2017), p. 1603483.

[107] Michael A. Hogg and John C. Turner. “Interpersonal attraction, social identification and psychological group formation”. In: *European Journal of Social Psychology* 15.1 (1985), pp. 51–66.

[108] Daniel Howell, R.P. Behringer, and Christian Veje. “Stress fluctuations in a 2D granular Couette experiment: a continuous transition”. In: *Physical Review Letters* 82.26 (1999), p. 5241.

[109] Walter Hussak and Amitabh Trehan. “Brief Announcement: On Termination of a Flooding Process”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing.* PODC 2019. 2019, pp. 153–155.

[110] Nicole Immorlica et al. “Exponential Segregation in a Two-dimensional Schelling Model with Tolerant Individuals”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA 2017. 2017, pp. 984–993.

[111] Ernst Ising. “Beitrag zur theorie des ferromagnetismus [Contribution to the Theory of Ferromagnetism]”. In: *Zeitschrift für Physik* 31.1 (1925), pp. 253–258.

[112] Soudeh Jahanshahi, Hartmut Löwen, and Borge Ten Hagen. “Brownian motion of a circle swimmer in a harmonic trap”. In: *Physical Review E* 95.2 (2017), p. 022606.

[113] Raphael Jeanson et al. “Self-organized aggregation in cockroaches”. In: *Animal Behaviour* 69.1 (2005), pp. 169–180.

[114] Iwan Jensen. “A parallel algorithm for the enumeration of benzenoid hydrocarbons”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2009.2 (2009), P02065.

[115] Iwan Jensen. “Improved lower bounds on the connective constants for two-dimensional self-avoiding walks”. In: *Journal of Physics A: Mathematical and General* 37.48 (2004), pp. 11521–11529.

[116] Matthew Jenssen, Peter Keevash, and Will Perkins. “Algorithms for #BIS-hard problems on expander graphs”. In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA 2019. 2019, pp. 2235–2247.

[117] Brian R. Johnson, Ellen van Wilgenburg, and Neil D. Tsutsui. “Nestmate recognition in social insects: overcoming physiological constraints with collective

decision making". In: *Behavioral Ecology and Sociobiology* 65.5 (2011), pp. 935–944.

[118] G. Junot et al. "Active versus Passive Hard Disks against a Membrane: Mechanical Pressure and Instability". In: *Physical Review Letters* 119.2 (2017), p. 028002.

[119] Rolf G. Karlsson and Mark H. Overmars. "Scanline algorithms on a grid". In: *BIT Numerical Mathematics* 28.2 (1988), pp. 227–241.

[120] Serge Kernbach. *Handbook of Collective Robotics: Fundamentals and Challenges*. Jenny Stanford Publishing, 2013.

[121] Aaron S. Keys et al. "Measurement of growing dynamical length scales and prediction of the jamming transition in a granular material". In: *Nature Physics* 3.4 (2007), pp. 260–264.

[122] Jon Kleinberg and Éva Tardos. *Algorithm Design and Analysis*. Pearson Education, Inc., 2006.

[123] Roman Kotecký and David Preiss. "Cluster Expansion for Abstract Polymer Models". In: *Communications in Mathematical Physics* 103 (1986), pp. 491–498.

[124] Sam Kriegman et al. "A scalable pipeline for designing reconfigurable organisms". In: *Proceedings of the National Academy of Sciences* 117.4 (2020), pp. 1853–1859.

[125] C. Ronald Kube and Eric Bonabeau. "Cooperative transport by ants and robots". In: *Robotics and Autonomous Systems* 30.1 (2000), pp. 85–101.

[126] Felix Kümmel et al. "Circular motion of asymmetric self-propelling particles". In: *Physical Review Letters* 110.19 (2013), p. 198302.

[127] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov chains and mixing times*. Providence, RI, USA: American Mathematical Society, 2009.

[128] Shengkai Li et al. "Programming Active Granular Matter with Mechanically Induced Phase Changes". To appear in Science Advances; pre-print available online at https://arxiv.org/abs/2009.05710. 2021.

[129] Shuguang Li et al. "Particle robotics based on statistical mechanics of loosely coupled components". In: *Nature* 567 (2019), pp. 361–365.

[130] Chao Liao et al. “Counting Independent Sets and Colorings on Random Regular Bipartite Graphs”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques.* APPROX/RANDOM 2019. 2019, 34:1–34:12.

[131] Melody X. Lim et al. “Cluster formation by acoustic forces and active fluctuations in levitated granular matter”. In: *Nature Physics* 15.5 (2019), pp. 460–464.

[132] Albert Tianxiang Liu et al. “Autoperforation of two-dimensional materials to generate colloidal state machines capable of locomotion”. In: *Faraday Discussions* (2021).

[133] Jintao Liu et al. “Metabolic co-dependence gives rise to collective oscillations within biofilms”. In: *Nature* 523 (2015), pp. 550–554.

[134] Kevin Lough. *Enumeration Methods and Series Analysis of Self-Avoiding Polygons on the Hexagonal Lattice, with Applications to Self-organizing Particle Systems.* Undergraduate Honors Thesis, Arizona State University. 2019.

[135] Eyal Lubetzky et al. “Quasi-polynomial mixing of the 2D stochastic Ising model with “plus" boundary up to criticality”. In: *Journal of the European Mathematical Society (JEMS)* 15.2 (2013), pp. 339–386.

[136] Bruce J. MacLennan. “The Morphogenetic Path to Programmable Matter”. In: *Proceedings of the IEEE* 103.7 (2015), pp. 1226–1232.

[137] Anne E. Magurran. “The adaptive significance of schooling as an anti-predator defence in fish”. In: *Annales Zoologici Fennici* 27.2 (1990), pp. 51–66.

[138] Fabio Martinelli and Fabio Lucio Toninelli. “On the Mixing Time of the 2D Stochastic Ising Model with “Plus" Boundary Conditions at Low Temperature”. In: *Communications in Mathematical Physics* 296.1 (2010), pp. 175–213.

[139] Joseph E. Mayer. “The Statistical Mechanics of Condensing Systems. I”. In: *The Journal of Chemical Physics* 5 (1937), pp. 67–73.

[140] Siddharth Mayya et al. “Non-Uniform Robot Densities in Vibration Driven Swarms Using Phase Separation Theory”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IROS 2019. 2019, pp. 4106–4112.

[141] R. Preston McAfee. “A dominant strategy double auction”. In: *Journal of Economic Theory* 56.2 (1992), pp. 434–450.

[142] Helen F. McCreery and Michael D. Breed. “Cooperative transport in ants: a review of proximate mechanisms”. In: *Insectes Sociaux* 61.2 (2014), pp. 99–110.

[143] Francisco Melo, Paul B. Umbanhowar, and Harry L. Swinney. “Hexagons, kinks, and disorder in oscillated granular layers”. In: *Physical Review Letters* 75.21 (1995), pp. 3838–3841.

[144] Othon Michael, George Skretas, and Paul G. Spirakis. “On the transformation capability of feasible mechanisms for programmable matter”. In: *Journal of Computer and System Sciences* 102 (2019), pp. 18–39.

[145] Othon Michail and Paul G. Spirakis. “Simple and efficient local codes for distributed stable network construction”. In: *Distributed Computing* 29 (2016), pp. 207–237.

[146] Russ Miller and Quentin F. Stout. “Efficient parallel convex hull algorithms”. In: *IEEE Transactions on Computers* 37.12 (1988), pp. 1605–1618.

[147] Sarah Miracle, Dana Randall, and Amanda Pascoe Streib. “Clustering in Interfering Binary Mixtures”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques.* APPROX/RANDOM 2011. 2011, pp. 652–663.

[148] Namiko Mitarai and Franco Nori. “Wet granular materials”. In: *Advances in Physics* 55.1–2 (2006), pp. 1–45.

[149] Nathan J. Mlot, Craig A. Tovey, and David L. Hu. “Fire ants self-assemble into waterproof rafts to survive floods”. In: *Proceedings of the National Academy of Sciences* 108.19 (2011), pp. 7669–7673.

[150] A.M. Mohammed et al. “Self-assembling DNA nanotubes to connect molecular landmarks”. In: *Nature Nanotechnology* 12 (2017), pp. 312–316.

[151] Sanaz Mostaghim, Christoph Steup, and Fabian Witt. “Energy Aware Particle Swarm Optimization as search mechanism for aerial micro-robots”. In: *2016 IEEE Symposium Series on Computational Intelligence*. SSCI 2016. 2016, pp. 1–7.

[152] Nils Napp, Samuel Burden, and Eric Klavins. “Setpoint regulation for stochastically interacting robots”. In: *Autonomous Robots* 30.1 (2011), pp. 57–71.

[153] Gennaro Notomista et al. “A Study of a Class of Vibration-Driven Robots: Modeling, Analysis, Control and Design of the Brushbot”. In: *2019 IEEE/RSJ*

International Conference on Intelligent Robots and Systems. IROS 2019. 2019, pp. 5101–5106.

[154] Hamed Omidvar and Massimo Franceschetti. “Self-organized Segregation on the Grid”. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. PODC 2017. 2017, pp. 401–410.

[155] Anil Özdemir et al. “Finding Consensus Without Computation”. In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1346–1353.

[156] Anil Özdemir et al. “Spatial Coverage Without Computation”. In: *2019 IEEE International Conference on Robotics and Automation*. ICRA 2019. 2019, pp. 1346–1353.

[157] Matthew J. Patitz. “An Introduction to Tile-Based Self-Assembly and a Survey of Recent Results”. In: *Natural Computing* 13.2 (2014), pp. 195–224.

[158] Andrzej Pelc. “Deterministic Rendezvous Algorithms”. In: *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*. Cham: Springer, 2019, pp. 423–454.

[159] Daniel Pickem et al. “The Robotarium: A remotely accessible swarm robotics research testbed”. In: *2017 IEEE International Conference on Robotics and Automation*. ICRA 2017. 2017, pp. 1699–1706.

[160] Benoit Piranda and Julien Bourgeois. “Designing a quasi-spherical module for a huge modular robot to create programmable matter”. In: *Autonomous Robots* 42 (2018), pp. 1619–1633.

[161] André Pönitz and Peter Tittman. “Improved Upper Bounds for Self-Avoiding Walks in $\mathbb{Z}^d$”. In: *The Electronic Journal of Combinatorics* 7.R21 (2000), pp. 1–10.

[162] Alexandra Porter and Andréa W. Richa. “Collaborative Computation in Self-organizing Particle Systems”. In: *Unconventional Computation and Natural Computation*. UCNC 2018. 2018, pp. 188–203.

[163] Arthur Prindle et al. “Ion channels enable electrical communication in bacterial colonies”. In: *Nature* 527 (2015), pp. 59–63.

[164] Sergio Rajsbaum and Jorge Urrutia. “Some problems in distributed computational geometry”. In: *Theoretical Computer Science* 412.41 (2011), pp. 5760–5770.

[165] Sriram Ramaswamy. “Active matter”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2017 (2017), p. 054002.

[166] Dana Randall. “Rapidly mixing Markov chains with applications in computer science and physics”. In: *Computing in Science & Engineering* 8.2 (2006), pp. 30–41.

[167] Gregory J. E. Rawlins. “Explorations in Restricted Orientation Geometry”. PhD thesis. Ontario: University of Waterloo, 1987.

[168] C. R. Reid et al. “Army ants dynamically adjust living bridges in response to a cost–benefit trade-off”. In: *Proceedings of the National Academy of Sciences* 112.49 (2015), pp. 15113–15118.

[169] Chris R. Reid and Tanya Latty. “Collective behaviour and swarm intelligence in slime moulds”. In: *FEMS Microbiology Reviews* 40.6 (2016), pp. 798–806.

[170] Erin C. Rericha et al. “Shocks in supersonic sand”. In: *Physical Review Letters* 88.1 (2001), p. 014302.

[171] Ricardo Restrepo et al. “Improving mixing conditions on the grid for counting and sampling independent sets”. In: *Probability Theory and Related Fields* 156 (2013), pp. 75–99.

[172] Colette Rivault and Ann Cloarec. “Cockroach aggregation: discrimination between strain odours in Blattella germanica”. In: *Animal Behaviour* 55.1 (1998), pp. 177–184.

[173] Paul W. K. Rothemund and Erik Winfree. “The Program-Size Complexity of Self-Assembled Squares (Extended Abstract)”. In: *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. STOC 2000. 2000, pp. 459–468.

[174] T’ai H. Roulston, Grzegorz Buczkowski, and Jules Silverman. “Nestmate discrimination in ants: effect of bioassay on aggressive behavior”. In: *Insectes Sociaux* 50.2 (2003), pp. 151–159.

[175] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. “Programmable Self-Assembly in a Thousand-Robot Swarm”. In: *Science* 345.6198 (2014).

[176] Erol Şahin. “Swarm Robotics: From Sources of Inspiration to Domains of Application”. In: *Swarm Robotics*. 2005, pp. 10–20.

[177] William Savoie et al. “A robot made of robots: Emergent transport and control of a smarticle ensemble”. In: *Science Robotics* 4.34 (2019), eaax4316.

[178] William Savoie et al. “Phototactic Supersmarticles”. In: *Artificial Life and Robotics* 23.4 (2018), pp. 459–468.

[179] Thomas C. Schelling. “Dynamic models of segregation”. In: *The Journal of Mathematical Sociology* 1.2 (1971), pp. 143–186.

[180] Thomas C. Schelling. “Models of Segregation”. In: *The American Economic Review* 59.2 (1969), pp. 488–493.

[181] Michael I. Shamos and Dan Hoey. “Geometric intersection problems”. In: *17th Annual Symposium on Foundations of Computer Science*. SFCS 1976. Washington, DC, USA: IEEE Computer Society, 1976, pp. 208–215.

[182] Alexandre P. Solon et al. “Pressure and Phase Equilibria in Interacting Active Brownian Spheres”. In: *Physical Review Letters* 114.19 (2015), p. 198301.

[183] Joakim Stenhammar et al. “Continuum Theory of Phase Separation Kinetics for Active Brownian Particles”. In: *Physical Review Letters* 111.14 (2013), p. 145702.

[184] Philip S. Stewart and Michael J. Franklin. “Physiological heterogeneity in biofilms”. In: *Nature Reviews Microbiology* 6 (2008), pp. 199–210.

[185] Jun Tanimoto and Hiroki Sagara. “Relationship between dilemma occurrence and the existence of a weakly dominant strategy in a two-player symmetric game”. In: *Biosystems* 90.1 (2007), pp. 105–114.

[186] Tommaso Toffoli and Norman Margolus. “Programmable matter: Concepts and realization”. In: *Physica D: Nonlinear Phenomena* 47.1 (1991), pp. 263–272.

[187] John C. Turner. “Towards a cognitive redefinition of the social group”. In: *Cahiers de Psychologie Cognitive/Current Psychology of Cognition* 1.2 (1981), pp. 93–118.

[188] Dejan Vinković and Alan Kirman. “A physical analogue of the Schelling model”. In: *Proceedings of the National Academy of Sciences* 103.51 (2006), pp. 19261–19265.

[189] Jennifer E. Walter, Elizabeth M. Tsai, and Nancy M. Amato. “Algorithms for Fast Concurrent Reconfiguration of Hexagonal Metamorphic Robots”. In: *IEEE Transactions on Robotics* 21.4 (2005), pp. 621–631.

[190] Guopeng Wei et al. "Molecular Tweeting: Unveiling the Social Network Behind Heterogeneous Bacteria Populations". In: *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*. BCB 2015. New York, NY, USA: ACM, 2015, pp. 366–375.

[191] Hongxing Wei et al. "Staying-alive path planning with energy optimization for mobile robots". In: *Expert Systems with Applications* 39.3 (2012), pp. 3559–3571.

[192] Sean Wilson et al. "Design of ant-inspired stochastic control policies for collective transport by robotic swarms". In: *Swarm Intelligence* 8 (2014), pp. 303–327.

[193] David H. Wolpert. "The stochastic thermodynamics of computation". In: *Journal of Physics A: Mathematical and Theoretical* 52.19 (2019), p. 193001.

[194] Damien Woods et al. "Active Self-Assembly of Algorithmic Shapes and Patterns in Polylogarithmic Time". In: *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*. ITCS 2013. 2013, pp. 353–354.

[195] Hui Xie et al. "Reconfigurable magnetic microrobot swarm: Multimode transformation, locomotion, and manipulation". In: *Science Robotics* 4.28 (2019), eaav8006.

[196] Jing Fan Yang et al. "Synthetic Cells: Colloidal-sized state machines". In: *Robotic Systems and Autonomous Platforms*. Ed. by Shawn M. Walsh and Michael S. Strano. Woodhead Publishing in Materials. Woodhead Publishing, 2019, pp. 361–386.

[197] Mark Yim et al. "Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]". In: *IEEE Robotics Automation Magazine* 14.1 (2007), pp. 43–52.

[198] Guoxian Zhang, Gregory K. Fricke, and Devendra P. Garg. "Spill Detection and Perimeter Surveillance via Distributed Swarming Agents". In: *IEEE/ASME Transactions on Mechatronics* 18.1 (2013), pp. 121–129.

APPENDIX A

FULL ACKNOWLEDGEMENTS

To Annie Carson Daymude, my wife, love, and partner in all things — what words can I say? You are with me in the moments of success and elation and in the times where stress and brokenness cause me to forget myself. My life is infinitely more beautiful for having you in it.

To my parents Susan and Andrew Daymude who gave me my beginning, I will forever be thankful for the love, faith, and tools you equipped me with to enter the world confidently and gently, modeling service to others and humility in accomplishments. To my sister Dani Daymude, your companionship and love and belief in me even when I could not love or believe in myself is a gift that I will never be done thanking you for. To my brother AJ Daymude, you are joy and delight personified; thank you for your love, humor, and permission to take life less seriously. To Big Imo (Janney Suh), Little Imo (Ann Losquadro), and Frank Losquadro, thank you for teaching me fun, adventure, competition, and love that does not know distance. Memories with you will always be Southern California beaches, backyard hockey in Tejon Ranch, and the laughter and joy of Christmas. To 할머니 (Grace Suh) and 할아버지 (David Suh), your lives are an inspiration to me and your pride in who I've become gives me strength. 사랑해요. To Mimi (JoAnn Richards) and Poppi (Carl Richards), thank you for your love that extends beyond distance and time apart.

To my second parents Julie and Monty Carson, thank you for being my home away from home, instilling an enjoyment of travel in me, and caring both Annie and me so well. To my new sisters Catie, Emma, and Mila Carson, you are the best friends I never knew I was missing and now can't imagine my life without. And to the rest of my new family — especially Andrea, Rod, and Mattea Pauls as well as Chad, Jana, Isaac, Ira, and Willem Sundin — thank you for so graciously welcoming me as one of your own and allowing me to be a part of your lives.

To Andréa Richa, who believed in me from the start, welcomed me into her research group as an undergraduate, celebrated me in becoming her PhD student, guided me through the twists and turns of academia, gave me the flexibility to be near Annie during my PhD, advocated for me to teach, and supported me in innumerable other ways, thank you. I am proud to be your student and to carry your training forward into my career. To Dana Randall, thank you for all the ways you complement Andréa's mentoring, pushing me when you know it's what I need and celebrating with me when I come out stronger on the other side. To Christian Scheideler, thank you for years of fascinating discussions and working alongside me on challenging problems, going all the way back to my first summer in Paderborn. Thank you also to Ted Pavlic, whose lab visits were some of the most thrilling and inspiring moments of my PhD; Stephanie Gil, whose teaching helped me see new connections between distributed computing and swarm robotic systems; Dan Goldman, whose generosity and collaboration opened novel and rich interpretations of our algorithmic theory; Michael Strano, Todd Murphey, and Jeremy England who always instigated fascinating and inspiring discussions; Hal Kierstead, whose lectures and office hours made me fall in love with theory; and Susanna Fishel, whose support of my early attempts at teaching gave me confidence to continue on.

Thank you to my grade school teachers who nurtured my love of mathematics, science, writing, and stories, especially Mrs. Core, Mrs. Chamberlain, Mrs. Wood, Mr. Zameroski, Mr. Figueroa, Mr. Bristol, Mr. Rosenast, and Mr. Duncan. To the teachers who told me that I would eventually hit a wall where material would be too difficult for me to understand, I want you to know that I have found no such walls. Instead, I have been met by brilliant and generous people that are as willing to teach and support me as I am willing to be taught and supported.

To Robert Gmyr and Thim Strothmann, your mentoring gave me the confidence that I needed to pursue a PhD. I will always remember your advice that while not all research is meaningful, we have the privilege and responsibility to search for problems that matter. To Sarah Cannon, your brilliance, patience, and kindness guided me through many critical points of my PhD; I am deeply thankful to know you. To Jamison Weber and Anya Chaturvedi, your friendship has been invaluable as we navigated our graduate degrees together; I am excited to see where your research and work take you. To Shengkai Li, your seemingly boundless energy, creativity, and generosity have been the backbone of our collaboration; thank you. To Kristian Hinnenthal, though we did not work together for long, I hope that our friendship will continue; may the Lord bless you and keep you as you, Mimi, and Linus explore what life holds outside the university. To my students Joseph Briones, Ryan Yiu, Cem Gökmen, Noble Harasha, Ziad Abdelkarim, Chris Boor, and Kevin Lough, thank you for helping me become a more effective and compassionate mentor.

To Daniel Herder and Anna Rischitelli, you have been my closest friends since long before this PhD endeavor and I am so thankful that we get to continue to do life together. To Amie Pierone, it was uniquely wonderful to discover the beauty of mathematics with you and to have someone who shared the faith to learn alongside. To Siv Cheruvu, Cynthia Guo, Tim Lee, Anna Shao, Mwangala Akamandisa, Miriam Kilimo, Sarah Brister, and Abena Twumasi, I have never in my life felt so immediately welcomed and loved by a group of new friends as I did in my fall in Atlanta. Thank you for showing me what vibrant, beautiful community in Christ and in academia can look like. To Geoff and Katie Gentry, Jean Duerbeck, Turner and Deborah Beazley, Noah Schumerth, Christian Ross, Levi Helm, and Charles and Holly Starr, thank you all for accompanying me on my journey to integrate our Christian faith with the intellectual pursuits of academia; I pray that we all can find a way to do meaningful work humbly and generously in the service of others. To Tom and Gayle Parker, Victoria and Danny King, and Cris and Katie Tietsort, thank you for your encouragement and prayers as I worked to finish this dissertation; may the Lord make his face shine upon you and be gracious to you. Finally, to Theo and Crystal Davis, you mean more to me and Annie than I know how to express; we love you and Lucia and are so thankful for your presence in our lives.