# Self-* Algorithms for Distributed Systems

*Programmable Matter & Overlay Networks*

Thim Frederik Strothmann

Paderborn University

**Reviewers:**

- Prof. Dr. Christian Scheideler, Paderborn University
- Prof. Dr. Andréa W. Richa, Arizona State University
- Prof. Dr. Friedhelm Meyer auf der Heide, Paderborn University

*To Katharina, my wonderful wife.*

# Abstract

This thesis considers two scenarios for *self-** algorithms in distributed computing: *self-organizing programmable matter* and *monotonic searchability for self-stabilizing overlay topologies*.

The former topic considers programmable matter that consists of tiny computationally limited units called *particles*, which can move in two-dimensional space, bond and communicate with each other. This kind of matter is studied in the recently introduced *amoebot* model and we investigate the feasibility of solving fundamental problems for programmable matter in that model. More precisely, the focus is on two major problems: *coating* and *shape formation*. In coating, the particles are connected to an unknown object (e.g., it can be convex or concave) and the ultimate goal is to coat the object as evenly as possible. We present an algorithmic framework that solves the coating problem in a worst-case runtime that is linear in the number of particles, which is shown to be worst-case optimal. In shape formation, we focus on building basic shapes out of programmable matter where the size of the constructed shape scales with the number of particles. We introduce an algorithmic framework to construct various simple geometric shapes, which again has a linear worst-case runtime. Supplementary to these two central problems we investigate the ability of constant-size programmable matter that is connected to an unknown object.

The latter topic focuses on the problem of maintaining *searchability* in an overlay topology while that topology is stabilizing. More specifically, we investigate *self-stabilizing* protocols for the line topology: i.e., protocols that are guaranteed to *converge* from any possible initial state to a desired state in which the overlay constitutes a line. In addition to the convergence process, the protocols should also monotonically maintain a property called searchability: i.e., once a node $a$ can send a search message to another node $b$ in the topology, it is always able to do so in the future. We study this problem in two variants: the *strict line* topology and the *super-line* topology. In the first variant the

ultimate goal topology is a line over all nodes: i.e., each node is only connected to its predecessor and successor (according to their ID). In the second variant, we allow the goal topology to have more edges, but the line has to be a subgraph of it. In both scenarios we present: (i) a protocol that stabilizes to the desired protocol, (ii) a routing protocol that is able to route search messages, (iii) a self-stabilization proof and (iv) a monotonic searchability proof.

# Zusammenfassung

In dieser Doktorarbeit werden zwei Szenarien für *Self-** Algorithmen in verteilten Systemen betrachtet: *selbst-organisierende programmierbare Materie* und *monotone Suchbarkeit für selbst-stabilisierende Overlaytopologien.*

Das erste Thema betrachtet programmierbare Materie, welche aus kleinen, in ihren rechnerischen Fähigkeiten beschränkten Einheiten besteht, die *Partikel* genannt werden. Diese können sich im zweidimensionalen Raum bewegen, sich verbinden und miteinander kommunizieren. Programmierbare Materie solcher Art kann im erst kürzlich eingeführten *amoebot* Model betrachtet werden. Es wird eruiert ob programmierbare Materie grundlegende Probleme in diesem Model lösen kann. Genauer gesagt liegt der Fokus auf zwei Hauptproblemstellungen: *Coating* und *Shape Formation.* Im Coating sind die Partikel mit einem statischen, unbekannten Objekt (welches beispielsweise konvex oder konkav sein kann) verbunden. Hier ist das ultimative Ziel das Objekt gleichmäßig zu ummanteln. Es wird ein algorithmisches Framework präsentiert, welches das Coating Problem löst. Die worst-case Laufzeit ist dabei linear in der Partikelanzahl, was worst-case optimal ist. Bei der Shape Formation soll die Materie einfache Formen konstruieren, wobei die Größe der Form mit der Anzahl der Partikel skaliert. Auch hier wird ein algorithmisches Framework präsentiert, welches mehrere einfache geometrische Figuren bauen kann. Die worst-case Laufzeit ist ebenfalls linear in der Partikelanzahl. Ergänzend zu diesen zwei Problemschwerpunkten wird die Fähigkeit von programmierbarer Materie konstanter Größe betrachtet, die zu einem unbekannten Objekt verbunden ist.

Das zweite Thema fokussiert sich auf das Problem die *Suchbarkeit* monton in einer Overlaytopologie aufrecht zu erhalten während sich diese stabilisiert. Konkret werden *selbst-stabilisierende* Protokolle für die Linientopologie betrachtet, also Protokolle die garantiert von einem beliebigen initialen Zustand zu einem gewünschten Zustand, in dem die Topologie eine Linie ist, *konvergieren.* Zusätzlich zu der Konvergenz sollen die Protokolle auch die Eigenschaft der

Suchbarkeit monoton aufrechterhalten. Dies bedeutet, dass sobald ein Knoten $a$ einen anderen Knoten $b$ mit einer Suchnachricht erreicht, Knoten $a$ dies auch zu allen zukünftigen Zeitpunkten schafft. Das Problem wird in zwei Varianten betrachtet: der *strikten Linientopologie* und der *Super-Linientopologie*. In der ersten Variante soll in der Zieltopologie jeder Knoten nur zwei Nachbarn haben, seinen Vorgänger und seinen Nachfolger (sortiert nach der ID). In der zweiten Variante werden mehrere Nachbarn in der Zieltopologie erlaubt, aber die Linie muss ein Subgraph sein. Für beide Szenarien wird: (i) ein selbst-stabilisierendes Protokoll für die entsprechende Topologie präsentiert, (ii) ein Routing Protokoll für Suchnachrichten angegeben, (iii) die Selbst-Stabilisierung bewiesen und (iv) der Erhalt der monotonen Suchbarkeit bewiesen.

# Contents

**List of Theorems** **xi**

**List of Figures** **xv**

**Foreword** **xvii**

**1. Introduction** **1**
1.1. Thesis Overview . . . . . 3
1.2. List of Own Publications . . . . . 7

**I. Self-Organizing Programmable Matter** **11**

**2. Prologue** **13**
2.1. Model . . . . . 16
2.2. Related Literature . . . . . 19

**3. Universal Coating** **29**
3.1. Problem Statement . . . . . 30
3.2. Universal Coating Algorithm . . . . . 32
3.3. Correctness . . . . . 39
3.4. Runtime Analysis . . . . . 50

**4. Basic Shape Formation** **67**
4.1. Problem Statements . . . . . 68
4.2. Shape Formation Algorithm . . . . . 69
4.3. Line Shape Formation . . . . . 72
4.4. Hexagon Shape Formation . . . . . 76
4.5. Triangle Shape Formation . . . . . 79

**5. Constant Size Particle Systems** **83**
5.1. Problem Statements . . . . . 84
5.2. One Particle . . . . . 85
5.3. Two Particles . . . . . 87
5.4. Three Particles . . . . . 91

**6. Conclusion of Part I** **93**
6.1. Further Results . . . . . 94
6.2. Future Work . . . . . 96

**II. Monotonic Searchability for Self-Stabilizing Topologies** **99**

**7. Prologue** **101**
7.1. Model . . . . . 104
7.2. Related Literature . . . . . 107

**8. Problem Statement, Preliminaries & Primitives** **111**
8.1. Problem Statement . . . . . 112
8.2. Restrictions and Preliminary Results . . . . . 114
8.3. Primitives for Overlay Networks . . . . . 119

**9. Monotonic Searchability** **127**
9.1. Monotonic Searchability for the Line Topology . . . . . 128
9.2. Monotonic Searchability for the Super-Line Topology . . . . . 150
9.3. Comparing the Strict Line and the Super-Line . . . . . 166

**10. Conclusion of Part II** **177**
10.1. Further Results . . . . . 178
10.2. Future Work . . . . . 179

**Bibliography** **181**

# List of Theorems


3.1. Lemma (No Hole Property) . . . . . 40
3.2. Lemma (Ring of Trees Structure) . . . . . 40
3.3. Lemma (Forest Subgraph) . . . . . 40
3.4. Lemma (Ring Subgraph) . . . . . 41
3.5. Lemma (Idle-Active Connectedness) . . . . . 42
3.6. Corollary (Connectedness of $P$ and $O$) . . . . . 42
3.7. Lemma (Complaint Flags Equality) . . . . . 42
3.8. Lemma (Idle to Active) . . . . . 43
3.9. Lemma (Boundary Followers Become Roots) . . . . . 44
3.10. Corollary (Super-roots Become Roots) . . . . . 44
3.11. Lemma (Complaint Flag Leads to Movement) . . . . . 44
3.12. Lemma (Expanded Particles Contract) . . . . . 44
3.13. Lemma (Occupation of Surface Layer) . . . . . 45
3.14. Lemma (Leader on Surface Layer) . . . . . 46
3.15. Lemma (Progress of Coating I) . . . . . 46
3.16. Lemma (Progress of Coating II) . . . . . 47
3.17. Lemma (Bound on Activation and Retirement) . . . . . 47
3.18. Lemma (Bound on Movements) . . . . . 48
3.19. Observation (Super-root Movements) . . . . . 48
3.20. Observation (Layer Size) . . . . . 49
3.21. Theorem (Correctness of Universal Coating Algorithm) . . . 49
3.22. Lemma (Runtime Lower Bound (Worst-case)) . . . . . 50
3.23. Theorem (Competitive Analysis) . . . . . 51
3.24. Definition (Parallel Schedule) . . . . . 54
3.25. Lemma (Parent Child Relation in Forest Schedules) . . . . . 55
3.26. Lemma (Dominance Argument) . . . . . 56
3.27. Lemma (Greedy Schedule Construction) . . . . . 57
3.28. Definition (Complaint-based Parallel Schedule) . . . . . 59
3.29. Lemma (Dominance Argument with Complaints) . . . . . 60

3.30. Lemma (Runtime: Forming a Forest) . . . . . 61
3.31. Lemma (Progress on Surface Layer) . . . . . 62
3.32. Lemma (Runtime: Contracted Particles on the Surface Layer) 63
Claim (Runtime: Particles on the Surface Layer) . . . . . 63
3.33. Lemma (Runtime: Leader Election) . . . . . 64
3.34. Lemma (Runtime: From a Tree to a Path) . . . . . 64
3.35. Lemma (Runtime: Layer $i$) . . . . . 65
3.36. Lemma (Runtime: Higher layers) . . . . . 65
3.37. Theorem (Runtime: Universal Coating Algorithm) . . . . . 66

4.1. Lemma (Progress of Roots for $\mathcal{LSF}$) . . . . . 73
4.2. Theorem (Correctness of $\mathcal{LSF}$ Algorithm) . . . . . 73
4.3. Lemma (Greedy Forest Schedule for Shape Formation) . . . . 74
4.4. Theorem (Runtime of $\mathcal{LSF}$ Algorithm) . . . . . 76
4.5. Theorem (Correctness of $\mathcal{HSF}$ Algorithm) . . . . . 78
4.6. Theorem (Runtime of $\mathcal{HSF}$ Algorithm) . . . . . 79
4.7. Theorem (Correctness of $\mathcal{TSF}$ Algorithm) . . . . . 80
4.8. Theorem (Runtime of $\mathcal{TSF}$ Algorithm) . . . . . 81

5.1. Lemma (Impossibility: Convexity Test for One Particle) . . . 86
5.2. Theorem (Convexity Test for Two Particles) . . . . . 87
5.3. Corollary (Simple Shape Tests for Two Particles) . . . . . 88
5.4. Lemma (Impossibility: Line Length Comparison for One Particle) 88
5.5. Lemma (Impossibility: Line Length Comparison for Two Particles) . . . . . 89
5.6. Theorem (Impossibility: Remaining Shape Tests for Two Particles) . . . . . 91
5.7. Theorem (Rhombus Test for Three Particles) . . . . . 92
5.8. Corollary (Regular Hexagon Test for Three Particles) . . . . 92

8.1. Lemma (Unconditional Satisfaction vs. Monotonic Searchability) 115
8.2. Lemma (Searchability for Build-Line?) . . . . . 117
8.3. Lemma (Weak Connectivity Preservation) . . . . . 120
8.4. Theorem (Universality of Primitives) . . . . . 121
8.5. Corollary (Weak Universality of Primitives) . . . . . 122
8.6. Lemma (Necessity of each Primitive) . . . . . 122
8.7. Theorem (Decomposition into Primitives) . . . . . 123
8.8. Lemma (Deletion of References) . . . . . 123

9.1. Lemma (Neighbors are Sorted Correctly) . . . . . 133
9.2. Theorem (Build-Line+ Stabilizes to the Line) . . . . . 133
9.3. Lemma (Build-Line+ Preserves Weak Connectivity) . . . . 134
9.4. Lemma ($G_{NB}$ Becomes Bidirected and Strongly Connected) . 135

9.5. Lemma ($G_{NB}$ Becomes Bidirected) . . . . . 135
9.6. Corollary (Connected Components in $G_{NB}$ Become Lines) . . 136
9.7. Lemma (Properties of Non-temp. Edges in $G_{NB}$) . . . . . 136
9.8. Lemma (Properties of Temp. Edges in $G_{NB}$) . . . . . 138
9.9. Corollary (BUILD-LINE+ Stabilizes to a Supergraph of the Line) 139
9.10. Lemma (A Supergraph of the Line Becomes the Line) . . . . 139
9.11. Theorem (Monotonic Searchability for BUILD-LINE+) . . . . 140
9.12. Lemma (Admissible State Leads to Admissible Suffix) . . . . 142
9.13. Lemma (Suffix of Invariants 1 & 2 of of BUILD-LINE+) . . . 142
9.14. Lemma ($R(v)$ and $L(v)$ Grow Monotonically) . . . . . 143
9.15. Lemma (Suffix of Invariants 1 to 3 of of BUILD-LINE+) . . . 144
9.16. Lemma (Suffix of Invariants 1 to 5 of BUILD-LINE+) . . . . . 145
9.17. Lemma (BUILD-LINE+ Contains an Admissible State) . . . . 146
9.18. Corollary (Admissible Suffix of BUILD-LINE+) . . . . . 148
9.19. Lemma (Message Success for SEARCH+) . . . . . 148
9.20. Lemma (Successful Message Forwarding for SEARCH+) . . . . 148
9.21. Theorem (BUILD-SUPERLINE Stabilizes to the Super-line) . . 153
9.22. Lemma (Convergence of BUILD-SUPERLINE) . . . . . 154
9.23. Corollary (Closure of BUILD-SUPERLINE) . . . . . 156
9.24. Theorem (Monotonic Searchability for BUILD-SUPERLINE) . 157
9.25. Lemma (Admissible State Leads to Admissible Suffix (2)) . . 158
9.26. Corollary ($R(v)$ and $L(v)$ Grow Monotonically (2)) . . . . . 158
9.27. Lemma (Suffix of Invariant 1 of BUILD-SUPERLINE) . . . . . 158
9.28. Lemma (Suffix of Invariants 1 to 3 of BUILD-SUPERLINE) . . 159
9.29. Lemma (BUILD-SUPERLINE Contains an Admissible State) . 160
9.30. Corollary (Admissible Suffix of BUILD-SUPERLINE) . . . . . 162
9.31. Lemma (Message Success for Greedy-Search) . . . . . 162
9.32. Theorem (Short Routing Paths for Greedy-Search) . . . . . 165

# List of Figures

2.1. The Amoebot Model . . . . . 16

3.1. An example object with a tunnel of width 1. . . . . 31
3.2. Lower Bound . . . . . 51
3.3. Competitiveness . . . . . 52
3.4. Optimal Algorithm for Coating . . . . . 53

4.1. $\mathcal{HSF}$ Algorithm Execution . . . . . 77
4.2. $\mathcal{TSF}$ Algorithm Execution . . . . . 82

5.1. Example: Convexity Test for One Particle . . . . . 86
5.2. Example: Line Length Comparison Test . . . . . 88

8.1. Graph Instance for the Proof of Lemma 8.1 . . . . . 115
8.2. Graph Instance for the Proof of Lemma 8.2 . . . . . 118
8.3. The Four Primitives . . . . . 120
8.4. Graph Instances for the Proof of Lemma 8.8. . . . . 124

9.1. Degree Growth Comparison . . . . . 168
9.2. Distances Between Nodes (Build-Line+) . . . . . 170
9.3. Distances Between Nodes (Build-SuperLine) . . . . . 171
9.4. Stabilization Time Comparison . . . . . 172
9.5. Required Messages Comparison . . . . . 173
9.6. Influence of Search-Rates on Stabilization Time . . . . . 174
9.7. Successful Search Comparison with $\frac{5 \text{ Searches}}{200ms}$ . . . . . 175

# Foreword

The process of writing a PhD thesis is inherently coupled with a process of reflection concerning my PhD time. I am able to look back on the almost five years I spent in the *Theory of Distributed Systems* group and all the interesting and rewarding experiences I had. Before starting with the technical parts of my thesis I want to spare some lines for words of gratitude.

First and foremost, I want to thank my mentor and adviser Christian Scheideler, who was not only a knowledgeable academic supervisor who could provide information about age-old papers out of the top of his head, but was moreover a research colleague with a genuine interest in my ideas. Without you and your advice, this thesis would literally not have been possible. He also supported all my "adventures" outside of the academic realm – from university didactic certifications to my memberships in the university senate and faculty council. Thank you.

Additionally, I thank all my coauthors from the research group: Alexander, Andreas and Robert. I appreciate the research discussions we had (no matter how fruitful they were in the end) and fondly look back on all the papers we coauthored. Also the other (ex-)members of the group deserve to be mentioned here: Christina, Jonas, Kristian, Martina, Michael, Sebastian and Ulf-Peter. You had to put up with my humor and my character flaws and you all deserve a PhD solely for that. It was great to have so many nice persons around me in the naturally occurring ups and downs of the PhD phase.

Also, I want to thank Andrea Richa and her students Zahra Derakhshandeh

and Joshua Daymude from the Arizona State University with whom I had the honor of collaborating with in the programmable matter project. I think that the collaboration really strengthened the results that we achieved and is, thus, to some degree also responsible for this thesis.

A special thanks goes to my student assistant (and also mentee) Linghui Luo, whose work for the network simulator is more than appreciated. It was an absolute fun to have an assistant that is on a par and supervising your thesis and seeing your progress was much more fulfilling than anticipated.

Apart from all the people that I worked with, I also want to thank my family, especially my parents who supported me in my plan to study computer science back in 2007. I was delighted that you never cornered me to explain my research in layman's terms – the primal fear of every PhD student.

I additional want to thank the guys of *Easy Allies* - Daniel, Kyle, Don, Michael D., Bradley, Michael H., Brandon and Ben. Your work has been a constant source of amusement, which eased the (sometimes monotone) days of writing this thesis (and probably added a week or two to the writing process). Your positivity and "Love & Respect" attitudes have had a positive influence on my view on many things.

Most of all I want to thank the person who had to endure me throughout this academic journey and who suffered most under my working hours, conference visits and moods - my wife Katharina. Your presence in my life dampened my frustration when reviews were bad and proofs were falling apart. You were always eager to provide a place of retreat and beautifully force me to do and enjoy things outside of problem solving and analytical thinking. There is no more appropriate person who I could dedicate this thesis to.

*Thank you!*

# CHAPTER 1

# Introduction

> *The science of today is the technology of tomorrow.*
>
> Edward Teller, Theoretical Physicist

Ever since computers have been invented, they have been utilized to simplify tasks for humans. During (and after) World War II computers had the primary use of cracking military encryption codes (like *Colossus*, which was used by the British to crack the German Lorenz SZ 40/42 encryption machine) or to perform ballistics trajectory calculations (like the American ENIAC) — both tasks that are almost impossible to do by hand. In the following 70 years computers have become more versatile, powerful and interconnected. Yet, their main purpose is still to ease work for humans or to facilitate projects that would be impossible without them. Due to this increased complexity, the maintenance of computers, their programs and networks has become a specialized task in itself. For certain programs only experts with years of experience and accumulated (arcane) knowledge are able to maintain them. Moreover, the last years have shown that the increasing digitalization of everyday life also increases the complexity of computer systems. Thus, it is

not a bold prediction that one day computers and their programs will be too complex to be maintained by humans only.

One solution to this overarching problem of computer science is to build programs (and hardware systems) that take care of themselves, i.e., that they are able to handle failure states without human intervention and are inherently designed to recover. In the algorithmic community one common buzzword for such algorithms is *self-* algorithms*. Here the * is a placeholder for a multitude of different words: e.g., adjusting, configurating, healing, managing, optimizing, organizing, protecting and stabilizing. The common theme of these algorithms is that they are automatically adapting to unpredictable changes in their environment without external intervention. Especially in the area of distributed computing these algorithms are highly desirable, since a distributed scenario is by nature prone to external errors and changes of environmental parameters that cannot be anticipated beforehand. On a broad and intuitive level, one can distinguish between self-* algorithms that start from a correct baseline state and try to improve their behavior according to some target function (e.g., self-adjusting and self-optimizing algorithms) and algorithms that can recover from faulty or undesired system states to correct/desired ones (e.g., self-healing and self-stabilizing algorithms). However, this distinction is diffuse for some algorithms and thus cannot be treated as a proper categorization.

In this PhD thesis I investigate two areas of self-* algorithms: *self-organizing programmable matter* (see Part I) and *self-stabilizing overlay topologies* with a focus on the maintenance of *monotonic searchability* (see Part II). The basic premise of both topics is vastly different: The first topic investigates the abilities of a matter of computationally limited devices, whereas the second topic considers an overlay network of computers which exchange messages (see Chapters 2 and 7 for a thorough introduction on both topics). Despite their differences in vision and model, as well as the differences in the established results, both research areas are excellent examples for self-* algorithms. The algorithms put a heavy emphasize on the absence of user input and autonomously adapt to changes in their respective execution environments. From an algorithmic perspective, easy algorithmic primitive are a major building block in both topics.

Naturally, the two chosen topics are only a selection of possible scenarios for

self-* algorithms. Nevertheless, I hope this exemplifies that the field provides a broad research area which bears a lot of potential for the future. During my PhD phase I had the opportunity to work in both topics for an almost equal amount of time and even though I was frustrated with each topic at times, I would not want to trade off one for the other. Both have their inherent advantages and disadvantages (from a theoretical, practical and visionary perspective) and both have a reason to be investigated.

## 1.1. Thesis Overview

As already mentioned, my thesis consists of two different topics, each of which has a dedicated part. Since each topic from my work is vastly different from the other, each part of this thesis is absolutely self-contained and can be read without the other part. In the following, I describe the structure of each part and briefly explain the content of the chapters within it. Additionally, I mention the publications which serve as the main basis for the content of each chapter.

**Self-Organizing Programmable Matter** Chapter 2 introduces the topic of self-organizing programmable matter. It discusses the relevant related literature and establishes the *amoebot model* which is used in every technical chapter in Part I. The model assumes that the matter consists of tiny computationally limited units called particles, which can move in two-dimensional space, bond and communicate with each other. The model was first established in the following publication:

> **2014** (with Z. Derakhshandeh, S. Dolev, R. Gmyr, A. W. Richa and C. Scheideler). "Brief announcement: amoebot - a new model for programmable matter". In: *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, cf. [Der+14].

Chapter 3 considers the problem of universal coating, in which the programmable matter has to coat an unknown object uniformly. Throughout the chapter, I first introduce the universal coating problem and present an algorithm which aims at solving the problem efficiently. I prove the correctness of

the algorithm and show that the required worst-case time to solve the problem is linear in size of the programmable matter: i.e., in the number of particles. This runtime is provably worst-case optimal. Another major advantage of the algorithm is its simplicity – it is a combination of several algorithmic primitives that are integrated seamlessly without any explicit synchronization. The results of the chapter are based on the following two publications:

> **2017** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "Universal Coating for Programmable Matter". In: *Theoretical Computer Science*, cf. [Der+17].

> **2016** (with Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa and C. Scheideler). "On the Runtime of Universal Coating for Programmable Matter". In: *DNA Computing and Molecular Programming - 22nd International Conference, DNA 22, Munich, Germany, September 4-8, 2016, Proceedings*, cf. [Der+16a].

Note that an enhanced version of the conference paper, whose results are already included in this thesis, is submitted to the *Natural Computing* journal, see:

> **to appear** (with J. J. Daymude, R. Gmyr, A. W. Richa and C. Scheideler). "On the Runtime of Universal Coating for Programmable Matter". In: *Natural Computing*, cf. [Day+ar].

In Chapter 4, I investigate the problem of shape formation: i.e., the programmable matter has to change its shape to a predetermined one. More precisely, I explore three different basic shape formation problems: the construction of a line, a triangle and a hexagon. The main contribution of the chapter is an algorithmic framework which is able to build these three shapes (and potentially many more shapes) by just varying two rules in the framework. I prove the correctness of the algorithmic framework for all three problems and show that the time to construct the shape is again linear in the number of particles. This runtime analysis is original work that has not been published yet. The remaining results (i.e., the algorithm and its correctness) are based on the following publication:

**2015** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems". In: *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM' 15, Boston, MA, USA, September 21-22, 2015*, cf. [Der+15a].

Chapter 4 is the last technical chapter of Part I and considers the abilities of programmable matter of constant size. Naturally, the computational power of constantly many particles is very limited. I investigate the scenario in which the matter is connected to a static object and tries to gather information about the object. For example, the matter wants to evaluate whether the object is convex and if so, which kind of shape the object has. I can show that a single particle is not able to solve any of the problems introduced in the chapter, whereas two particles can determine whether the object is convex and has a simple shape (i.e., a line, a triangle, a parallelogram or a hexagon). With three particles one can determine whether the object has two sides of the same length. All results within the chapter are unpublished. The basic ideas were discussed informally at the Dagstuhl Seminar 16721 "Algorithmic Foundations of Programmable Matter" together with Damien Woods of *INRIA Paris*.

Part I is completed by Chapter 6 , which contains a conclusion of the first part. Moreover, it highlights my further research in the area of programmable matter, which is not part of this thesis and points out directions for future work.

**Monotonic Searchability for Self-Stabilizing Topologies** Part II is structured differently than Part I: i.e., whereas the first part investigates many different problems in the area of programmable matter, the second part focuses exclusively on the problem of maintaining monotonic searchability in a self-stabilizing overlay topology. In simple terms, I investigate whether it is possible to search successfully (and to maintain the successfulness) in a certain topology, while the self-stabilizing process of constructing the topology is still in progress. Chapter 7 follows a similar structure to Chapter 2 and introduces the topic of monotonic searchability, establishes the standard model used for topological self-stabilization and discusses the relevant related literature.

Chapter 8 formally defines the problem statement of monotonic searchability and introduces some preliminary results that impose restrictions and prerequisites for the initial topology. These first results are based on a section of the following publication:

> **2015** (with C. Scheideler and A. Setzer). "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, cf. [SSS15].

Additionally, Chapter 8 presents general results concerning the construction of overlay networks. These results are independent of the topic of monotonic searchability. However, they provide a helpful tool for my later investigation. This analysis is based on the following publication.

> **2016** (with A. Koutsopoulos and C. Scheideler). "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Information and Computation*, cf. [KSS16].

The very last theorem is based on:

> **2016** (with C. Scheideler and A. Setzer). "Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, cf. [SSS16].

Chapter 9 is the main technical chapter of Part II. It investigates monotonic searchability for the line topology in two different scenarios. In the first scenario the goal topology is a strict line (i.e., all nodes in the network are ordered by their identifier) in which each node has at most two neighbors; its successor and its predecessor. In the second scenario each node is allowed to have more neighbors, but the successor and predecessor have to be among them. For both scenarios I introduce a self-stabilizing protocol for the respective topologies and a protocol to route search requests in that topology. It is shown that the respective protocols indeed stabilize to their desired network topologies and that monotonic searchability is preserved during the self-stabilization process if the given routing protocol is used. The chapter is concluded by a comparison

between the two topologies in simulation experiments. The results concerning the strict line are based on the following publication.

> **2015** (with C. Scheideler and A. Setzer). "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, cf. [SSS15].

The results of the second scenario have not been published so far. The simulations of the comparative analysis were conducted by my student assistant Linghui Luo.

Similar to Part I, Part II finishes with a conclusion chapter. As before, this chapter highlights my further research in the area of self-stabilizing overlays that is not included in this thesis and lists possibilities for future work.

In all chapters except for this introductory one, I will adhere to the convention of the research community that even single authored publications use the first person plural to refer to the author.

## 1.2. List of Own Publications

This chapter is concluded by a section that contains all publications that I co-authored during my PhD phase. Brief announcements are left out deliberately. The publications are listed in reverse chronological order.

### 1.2.1. Journal Articles

> **2016** (with A. Koutsopoulos and C. Scheideler). "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Information and Computation*, cf. [KSS16].

> **2017** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "Universal Coating for Programmable Matter". In: *Theoretical Computer Science*, cf. [Der+17].

> **2016**. "The Impact of Communication Patterns on Distributed Self-Adjusting Binary Search Tree". In: *Journal of Graph Algorithms and Applications*, cf. [Str16].

### 1.2.2. Conference Publications

**2016** (with C. Scheideler and A. Setzer). "Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, cf. [SSS16].

**2016** (with Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa and C. Scheideler). "On the Runtime of Universal Coating for Programmable Matter". In: *DNA Computing and Molecular Programming - 22nd International Conference, DNA 22, Munich, Germany, September 4-8, 2016, Proceedings*, cf. [Der+16a].

**2016** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "Universal Shape Formation for Programmable Matter". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/-Pacific Grove, CA, USA, July 11-13, 2016*, cf. [Der+16b].

**2015** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems". In: *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM' 15, Boston, MA, USA, September 21-22, 2015*, cf. [Der+15a].

**2015** (with Z. Derakhshandeh, R. Gmyr, R. A. Bazzi, A. W. Richa and C. Scheideler). "Leader Election and Shape Formation with Self-organizing Programmable Matter". In: *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*, cf. [Der+15b].

**2015** (with C. Scheideler and A. Setzer). "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, cf. [SSS15].

**2015** (with A. Koutsopoulos and C. Scheideler). "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, cf. [KSS15].

**2015**. "The Impact of Communication Patterns on Distributed Self-Adjusting Binary Search Trees". In: *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, cf. [Str15].

**2014** (with D. Foreback, A. Koutsopoulos, M. Nesterenko and C. Scheideler). "On Stabilizing Departures in Overlay Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, cf. [For+14].

# Part I.

# Self-Organizing Programmable Matter

# CHAPTER 2

## Prologue

> ”*The applications for this tech are limitless. Construction. What used to take teams of people working by hand for months or years, can now be accomplished by one person. And that's just the beginning. [...] If you can think it, the microbots can do it. The only limit is your imagination.*”
>
> Hiro Hamada, Main Character in the Movie ”Big Hero 6”

Although it is often not stated explicitly in research papers, science fiction has often been an inspiration for the scientific community, especially for computer science and engineering. Popular examples of (scientific) inventions that were inspired by science fiction are the *submarine* which was inspired by Jules Verne's 1870 novel *”Twenty Thousand Leagues Under the Sea”* [Ver70], *mobile phones* whose development (and looks) were likely inspired by the communicators of the popular TV series *Star Trek*, and *virtual reality* which can be related to *Star Trek*'s Holodeck. Lesser known examples are the *taser*, which was envisioned in the young adult novel *”Tom Swift and His Electric Rifle, or, Daring Adventures in Elephant Land”* in 1911 [App11] and *geostationary satellites* for communication which were introduced by Arthur C. Clarke in his essay *”Extra-terrestrial Relays – Can*

*Rocket Stations Give World-wide Radio Coverage?"* [Cla45].

In this part of the thesis we investigate a topic that is also imbued by science fiction: *self-organizing programmable matter*. The basic idea of programmable matter is, inter alia, inspired by the T-1000 Terminator in the movie *Terminator 2: Judgment Day*. In the movie, the T-1000 is described and visualized as being composed of liquid metal and thus able to assume various shapes and colors. This idea of a material that is able to change its physical properties is the main inspiration of programmable matter: i.e., we envision matter that can change its physical properties such as shape, color, conductivity, material hardness, etc. in a self-organizing fashion. More explicitly, we assume that the change is not guided by human intervention, but by the matter itself and its perception of the environment.

A wide variety of systems could be seen as aligning with the overall vision of programmable matter; we focus on systems similar to those depicted in *Big Hero 6* which consist of simple computational elements, called particles. These particles can actively move in a self-organized way and can establish and release bonds with each other. However, each particle is severely handicapped in its computational abilities: e.g., it has constant-size memory, can only utilize local interactions and does not have any global information. Moreover, particles are completely anonymous and cannot distinguish among each other. We imagine each particle to be minuscule in size compared to that of the overall matter, and thus imagine the matter as being composed of a multitude of particles. We refer to such a programmable matter system as a particle system. It needs to be mentioned that the particles in Big Hero 6 (called microbots) are controlled by a neurological sensor that a person has to wear: i.e., the microbots are not self-organizing.

From a practical point of view, there are numerous scenarios in which programmable matter can either simplify a task that is hard to perform without it or even enable an activity that was impossible before. Imagine using the programmable matter to *coat* a bridge. Since each particle individually gathers information about its environment and can perform computations, we can use the matter to gather structural information about the bridge (e.g., stress, fissures, pressure) or the weather (e.g., wind speed, temperature, wet conditions). This information could then be accumulated and monitored or, to go even one step further, the programmable matter could react to the

information and repair cracks in the bridge by filling them up. Furthermore, one could use programmable matter for minimally invasive surgery. Imagine injecting the programmable matter into the human body to remove cancer. Given the ability to identify cancerous cells, the matter could surround them, isolating them from the healthy, vulnerable cells. A similar approach could be used to stop internal bleeding by coating arteries from the inside. Finally, one could use programmable matter as an everyday life product. For example, imagine a "multi-tool" that is composed of programmable matter and thus not only ergonomically adapts to the user, but is also able to perform various household functions: i.e., a comb that can change into a eating utensil, a hammer or a screwdriver. By adding and removing particles, the tool could scale in size to become a broom or even a ladder.

In this part of the thesis we study the possibilities of solving fundamental problems for programmable matter from a theoretical point of view. In doing so, we use the *amoebot* model (see Section 2.1), which is a two-dimensional abstraction from the vision of programmable matter we just described. We focus on two problems that are not only inspired by the above mentioned application scenarios, but also elementary problems that every incorporation of programmable matter has to face: *coating* (see Chapter 3) and *shape formation* (see Chapter 4). In coating, the programmable matter is connected to an object and the ultimate goal is to coat the object as evenly as possible. The object's shape and size are unknown: i.e., it can be convex or concave and it is possible that the surface of the object is larger than amount of particles available. In shape formation, we focus on building basic geometric shapes: e.g., a triangle. The size of the constructed shape should of course scale with the number of particles in the system. Supplementary to these two central problems, we also investigate the ability of small-size programmable matter: i.e., a particle system that consists of a constant number of particles (see Chapter 5). For all scenarios we are interested not only in the feasibility of solving problems but also in the efficiency of our solutions in terms of runtime.

Throughout all scenarios, we imagine that our matter forms one connected particle system at all times. Thus, particles have a high interest to not disconnect from the rest of the particle system: i.e., the *connectivity* of the particle system is a major concern for all of our results.

## 2.1. Model

In this section we present the (geometric) amoebot model as introduced in our very first publication [Der+14]. We assume that any structure that a particle system can form can be represented as a subgraph of an infinite graph $G = (V, E)$, where $V$ represents all possible positions the particles can occupy relative to their structure, and $E$ represents all possible atomic transitions that a particle can perform as well as all places where neighboring particles can bond to each other. In the *amoebot model*, we assume that $G = G_{\text{eqt}}$, where $G_{\text{eqt}}$ is the infinite regular triangular grid graph. Figure 2.1 (a) illustrates the standard planar embedding of $G_{\text{eqt}}$.

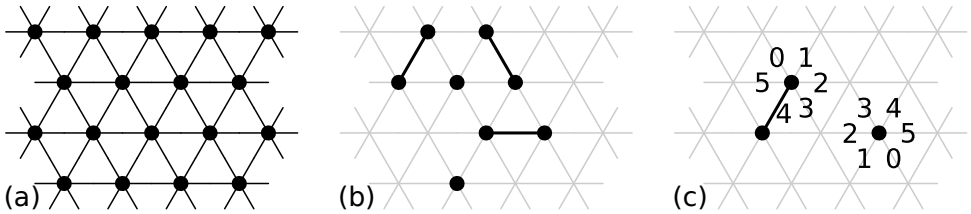



Figure 2.1.: (a) shows a section of $G_{\text{eqt}}$. Nodes of $G_{\text{eqt}}$ are shown as black circles. (b) shows five particles on $G_{\text{eqt}}$. The underlying graph $G_{\text{eqt}}$ is depicted as a gray mesh. A particle occupying a single node is depicted as a black circle, and a particle occupying two nodes is depicted as two black circles connected by an edge. (c) depicts two particles occupying two non-adjacent positions on $G_{\text{eqt}}$. The particles have different offsets for their head port labels.

Each particle occupies either a single node or a pair of adjacent nodes in $G_{\text{eqt}}$, and every node can be occupied by at most one particle. Two particles occupying adjacent nodes are *connected*, and we refer to such particles as *neighbors*. The connections between particles do not just ensure that the particles form a connected structure; they are also used for exchanging information as explained below.

Particles move through *expansions* and *contractions*: If a particle occupies one node (i.e., it is *contracted*), it can expand to an unoccupied adjacent node to occupy two nodes. If a particle occupies two nodes (i.e., it is *expanded*), it can contract to one of these nodes to occupy only a single node. Figure 2.1 (b) illustrates a set of particles (some contracted, some expanded) on the underlying

graph $G_{\text{eqt}}$. For an expanded particle, we denote the node the particle last expanded into as the *head* of the particle and call the other occupied node its *tail*. For a contracted particle, the single node occupied by the particle is both its head and its tail. A *handover* movement allows particles to stay connected as they move. Two scenarios are possible here: (1) a contracted particle $p$ can "push" a neighboring expanded particle $q$ and expand into the neighboring node previously occupied by $q$, forcing $q$ to contract, or (2) an expanded particle $p$ can "pull" a neighboring contracted particle $q$ to node $v$ it occupies thereby causing $q$ to expand into $v$, which allows $p$ to contract.

Particles are *anonymous*. Each particle has a collection of *ports*, one for each edge incident to the nodes occupied by it, and these ports have unique *port labels* from the local perspective of that particle. We assume that the particles have a common *chirality*, i.e., they all have the same notion of *clockwise direction*. This allows each particle to order the port labels of its head and tail in clockwise order. However, particles do *not* have a common sense of orientation since they can have different offsets of the labels (see Figure 2.1 (c)). Without loss of generality, we assume that each particle labels its head and tail ports from 0 to 5 in clockwise order. Whenever a particle $p$ is connected through some port to a particle $q$, we assume that $p$ knows the label of $q$'s port that lies opposite of the respective port of $p$. Furthermore, we assume that $p$ knows whether $q$'s port belongs to the head or the tail of $q$.

Each particle has a constant-size local memory that can be read and written to by any neighboring particle. This allows a particle to exchange information with a neighboring particle by simply writing it into the other particle's memory. A particle always knows whether it is contracted or expanded, and in the latter case it also knows along which head port label it is expanded. We assume that this information is also available to the neighboring particles (by publishing that label in its local memory). Due to the constant-size memory, particles cannot know the total number of particles, nor can they have any estimate on this number.

We assume the standard asynchronous model from distributed computing, where the particle system progresses through a sequence of *particle activations*, i.e., only one particle is active at a time. Whenever a particle is activated, it can perform an arbitrary bounded amount of computation (involving its own memory as well as the local memories of its neighbors) and at most one

movement. We define an *asynchronous round* to be over once each particle has been activated at least once.

The *configuration* $C$ of the particle system at the beginning of time $t$ consists of the nodes in $G_{\text{eqt}}$ occupied by the set of particles; in addition, for every particle $p$, $C$ contains the current state of $p$, including whether the particle is expanded or contracted, its port labels, and the contents of its local memory. If not mentioned otherwise, we refer to the particle system with $P$ and denote the *size* of $P$ (i.e., the number of particles in the system) with $n$.

## 2.2. Related Literature

Before discussing the related literature, we give a short overview of the structure of this section. Since our line of work in the area of programmable matter has only recently been established (e.g., the *amoebot* model was introduced in 2014 [Der+14]), we first give an overview of related models. Afterwards, we will specifically discuss related literature which focuses on the problems that we investigate in this thesis: *coating* and *shape formation*. Finally, we present the results that have been established in the amoebot model. The content of this section is a culmination of all related literature parts within our papers in the area of programmable matter [Der+14; Der+17; Der+16a; Der+15a].

**Related Models** Many approaches have already been proposed that could potentially be used for programmable matter or that share some similarities with the ultimate vision of programmable matter. Generally, one can distinguish between active and passive systems. In *passive systems* the particles either do not have any intelligence at all (i.e., they just move and bond based on their structural properties or due to interactions with the environment), or they have limited computational capabilities but cannot control their movements. In *active systems*, computational particles can control the way they act and move in order to solve a specific task. Note that we do not provide a full exegesis of the related models, and thus do not present all established results in detail. Most of them only share an underlying principle or idea with our amoebot model, but differ vastly in the purpose or the general research direction. We think that the chosen presentation is an appropriate trade-off between an in-depth discussion of each model (which is beyond the scope of this thesis) and simply name-dropping the models.

Prominent examples of research on **passive systems** are *DNA computing*, *tile self-assembly systems*, a variant of *population protocols*, and *slime molds*. These models are of little relevance to our concrete research since the particles cannot control the way they move and act by themselves. However, these models investigate problems that are similar to ours (especially tile assembly). Moreover, since some of these models have been investigated for decades, they have a solid body of theoretically founded algorithmic literature. In the following we will briefly present the four mentioned models.

The field of *DNA computing* has a more than 20 years old academic history. In 1994, Adleman [Adl94] was the first person to demonstrate that computation can be done with DNA on a molecular level. He was able to encode a graph into DNA molecules and solve the Hamiltonian path problem with enzymes on it. In the following years many practical results have been established (see the list of surveys below). On the theoretical side, [Bon+96] established the first results concerning the computational power of DNA: e.g., the authors show that DNA can efficiently compute satisfying assignments for general Boolean circuits and and that NP-hard problems like MAX-clique and Max-circuit-satisfiability can be solved. For more information about DNA computing, there is abundance of survey articles (e.g., [Pis97; DK02; WB08]) that provide an excellent overview.

Study of *tile self-assembly systems* was started by the seminal PhD thesis of Erik Winfree [Win98] in 1998 in which he introduced two theoretical models to investigate the self-assembly of DNA: the abstract Tile Assembly Model (aTAM) and the kinetic Tile Assembly Model (kTAM). The aTAM is the more abstract model which, e.g., ignores errors and thus provides a framework for studying the boundaries of such systems. On the other hand, the kTAM includes more physical restrictions and has helped to predict and shape the experimental direction of several laboratory experiments of DNA assembly. In its basic form, tile self-assembly consists of a set of square tiles which have labeled sides (called colors) and an initial assembly of tiles (called seed) which should grow into a predefined structure. This is done by assigning a positive integer strength value to each edge color and by demanding that when two tile edges are adjacent and their colors match then the edges bind with force equivalent to the strength of the color. An assembly then (usually) starts from the seed and additional tiles can attach, one at a time. A tile is allowed to attach to the already built assembly only if the sum of the bond strengths that it makes with the assembly meets a system-wide threshold value called the temperature. Tile-based self-assembly has proven to be a very rich area of research and researchers have created numerous submodels and variations of the aTAM and the kTAM (see the surveys [Dot12; Pat14; Woo13]).

In a distributed computing oriented context, Michail and Spirakis [MS16] recently proposed a model for network construction that is inspired by *population protocols* [Ang+06]. The population protocol model relates to self-organizing particle systems, but is also intrinsically different: agents (which would corre-

spond to our particles) freely move in space and interact in a pairwise fashion. Agents establish connections to any other agent in the system at any point in time, following the respective probabilistic distribution.

Finally, for *slime molds*, it has been shown that the slime mold *Physarum polycephalum* is able to solve the shortest paths problem in practice. There have been theoretical advances to model this phenomenon in the form of a coupled system of differential equations [TKN07] and a discretized (more computer science related) version [BMV12]. In fact, [BMV12] shows that the Physarum can indeed provably compute an approximation of the shortest path.

Prominent examples of **active systems** that share some similarities with our work are *robotic swarms* and *modular self-reconfigurable robotic systems*. Both fields have a seen a broad variety of practical research work as well as theoretical advances in the last years. Moreover, *cellular automata* are a well established model to study biologically inspired systems at a macro level. Finally, the *nubot model* is a theoretical framework that allows for algorithmic research of biomolecular-inspired systems. In the following we will again present the four models and shortly discuss their differences to the amoebot model.

In the area of *swarm robotics*, it is usually assumed that there is a collection of autonomous robots that can move freely in a given area and have limited sensing, vision, and communication ranges. The field is vast, including conferences that focus solely on swarm robotics and related fields: e.g. *ICSRSI* (International Conference on Swarm Robotics and Swarm Intelligence) and *SWARM* (Symposium on Swarm Behavior and Bio-Inspired Robotics). Swarm robots are used in a broad variety of contexts, including graph exploration (e.g., [Flo+13]), gathering problems (e.g., [AGM13; Cie+12]), shape formation problems (e.g., [Flo+08; RCN14]), and mimicking the collective behavior of natural systems to better understand the global effects of local behavior (e.g., [Cha09]). Surveys of recent results in swarm robotics can be found in [Ker12; Mcl08; BS13; Tan17]. The analytic techniques developed in swarm robotics and natural swarms are of some relevance to our work. However, the individual units in those systems have more powerful communication and/or processing capabilities than the particles we consider.

The field of *modular self-reconfigurable robotic systems* focuses on intra-robotic aspects such as design, fabrication, motion planning, and control of autonomous kinematic machines with variable morphology (e.g., [Fuk+88]).

[Yim+07] provides a survey of results that were established until 2007. *Metamorphic robots* form a subclass of self-reconfigurable robots that can dynamically self-reconfigure and share some of the characteristics of our amoebot model [Chi94] . They can be viewed as a large swarm of physically connected robotic modules which collectively act as a single entity and are thus to some relevance for our model. The hardware development in the field has been complemented by a number of algorithmic advances (e.g., [But+04; WWA04; Hur+15]), but mechanisms that automatically scale from a few to hundreds or thousands of individual units are still under investigation. From all mentioned algorithmic investigations, the most rigorous approach to analyze distributed and local algorithms for self-reconfigurable robotics has been done by Hurtado et al. in [Hur+15]. The model used in [Hur+15] shares some interesting similarities with our model: e.g., an underlying grid graph and a relative movement of robotic modules.

A *cellular automaton* is one of the classic models to study (biologically inspired) self-replicating systems. Established by Stanislaw Ulam and John von Neumann in the 1940s it is one the oldest models of computation and there is a wide variety of books on the power and applications of cellular automata (e.g., [Neu66; Wol86; Wol02; Sch11]). A cellular automaton is a (finite or infinite) collection of cells on a grid of specified shape which evolves through a number of discrete time steps. This evolution process is guided by a finite set of rules for each cell, which are based on the neighboring cells. The three most fundamental properties of a cellular automaton is the type of grid on which it is computed (e.g., a line in 1-dimensional space or square, triangular, and hexagonal grids in 2-dimensional space), the number of distinct states a cell can be in and the initial state of the automaton. Cellular automata differ vastly from our model since the cells can replicate or die at will and possess a global compass.

The *nubot* model (see e.g. [Che+14; CXW15; Woo+13]) by Woods et al. aims to provide the theoretical framework that would allow for rigorous algorithmic studies of biomolecular-inspired systems, specifically of self-assembly systems with active molecular components. Inspired by passive DNA-based tile self-assembly, molecular motors and molecular circuits, the model aims at capturing the interplay between molecular structure and dynamics. In it, simple molecular components form assemblies that can grow and shrink. Individual components

can undergo state changes and move relative to each other. There are quite a number of similarities between the nubot and the amoebot model: e.g., an underlying triangular grid graph, monomers (which correspond to our particles) with small internal memory and movement of monomers relative to another. Despite the similarities, key differences prohibit the translation of algorithms and other results under the nubot model to our systems and vice versa: e.g., there is always an arbitrarily large supply of surplus monomers that can be added to the nubot system as needed, and the nubot model includes a non-local notion of rigid-body movement. These key differences allow the nubots to form a line of size $n$ in logarithmic time.

**Coating** To the best of our knowledge, the problem of coating has not been considered in passive systems.

In active systems, coating has mostly been studied on the practical side of programmable matter, especially in the area of swarm robotics. In the field, it is commonly not studied as a stand-alone problem, but is part of *collective transport* (e.g., [Wil+14]) or *collective perception* (see the respective section of [Bra+13; NM12] for a summary of results). In collective transport a group of robots has to cooperate in order to transport an object. In general, the object is heavy and cannot be moved by a single robot, making cooperation necessary. In collective perception, a group of robots with a local perception each (i.e., only a local knowledge of the environment) aims at joining multiple individual perceptions to one big global picture (e.g., to collectively construct a map). Some research focuses on coating objects as an independent task under the name of *target surrounding* or *boundary coverage*. The techniques used in this context include stochastic robot behaviors (e.g., [KB14; Pav+13]), rule-based control mechanisms(e.g., [Blá+12]) and potential field-based approaches (e.g., [BLF12]).

**Shape Formation** Shape formation has been studied in almost any of the already presented models to some degree.

The most prominent examples of research on shape formation in passive systems appear in tile self-assembly systems, DNA computing and population protocols. Especially tile assembly, with its numerous submodels, has extensively studied the construction of finite, infinite, complex or aperiodic

shapes. Most interestingly, Soloveichik and Winfree [SW07] prove that any predetermined shape can be self-assembled with a number of tile types close to the Kolmogorov complexity of the target shape, if scaling of the shape is allowed. In general, this scale factor is at least linear in the size of the shape. An interesting recent result [SW15] combines the tile assembly model with *chemical reaction networks* (i.e., it uses chemical reaction networks to provide non-local control over a tile self-assembly process), which is able to produce many complex shapes with programs of low complexity. In fact, they can also bound the complexity of constructing a shape by the Kolmogorov complexity, but get rid of the dependence on a scale factor. A similar result is achieved in [Dem+11] in which the authors work in a submodel of tile assembly, which allows for a special kind of destroy operation. This reduces the scaling of the shape to only a logarithmic factor. For a general overview on the available shape formation results, we refer again to the three excellent surveys [Dot12; Pat14; Woo13]).

In DNA computing, shape formation has been considered in close relation to the already mentioned tile self-assembly model (see [Win+98] and the already mentioned tile assembly literature). Additionally, the field of *DNA Origami* also studies the formation shapes. Research was initiated by Paul Rothemund in 2006 [Rot06] in which he presents an approach to fold long single-stranded DNA molecules into arbitrary 2D shapes. This marks a pivotal point in DNA nanotechnology, since it enables control over designed molecular structures, thus opening up a new field which combines computer science, biochemistry and (bio)engineering as well as applications from medicine and (bio-)physics. For an overview of the established results in the last years we refer to one of the survey articles in the field (e.g., [KK10; Nan+10; Cas+11; Cha+16]).

In the population protocols model, network topology and shape construction problems are studied as well. In [MS16] the authors first consider algorithms for specific simple structures, such as a spanning line, a spanning star and a spanning ring. These algorithms do not terminate, but only converge. The expected time of convergence is analyzed under a uniform random scheduler. Moreover, the authors show some universal results by presenting a generic protocol that can simulate a Touring machine and is thus able to construct a large class of networks. Michail [Mic15] investigates algorithms that terminate with high probability for the construction of a line and a square. Similarly

to [MS16], he develops a universal approach that is able to construct a large class of 2D shapes. In order to achieve termination, Michail exploits the ability of nodes to self-assemble into larger structures that can then be used as distributed memories and the assumption that the system is well-mixed. This leads to a terminating protocol counting the size of the system w.h.p., which is the main building block for all constructions.

In active systems, shape formation (also called pattern formation) has been studied in almost every model. Especially in swarm robotics, the problem of pattern formation has a long history and has been investigated for around 20 years. There is a plethora of results and approaches which (mostly) focus on the practical side of the field. To the best of our knowledge there is no pattern formation survey. Thus, we will present some results that stood out to us and that can been seen as representatives of the achieved results. Most prominently, in [RCN14], the authors demonstrate that programmable self-assembly of complex two-dimensional shapes with hundreds or thousands of simple robots called *kilobots* is possible. Their algorithms are executed locally, but rely on a global pre-processing phase for shape formation that directly depends on the number of robots in the system. However, the dependence on the knowledge on the number of robots can be circumvented as it has been shown in [RS10]. In [AR10], the authors explore a method for a swarm of simple, physically identical, identically programmed robots that not only constructs polygonal approximations of arbitrary structures in the plane, but is also able to repair them systemetically and is tolerant to robot failures and externally-induced disturbances. Thus, their algorithm is not only self-assembling but also is self-healing. The practical work in the field has been complemented by quite a number of theoretic approaches of pattern formation. For example, in [Flo+08] the authors study the algorithmic limitations of building a predefined pattern by a set of asynchronous, anonymous, oblivious, autonomous mobile robots. The ability to build the shape depends strongly on common agreement about the environment. More precisely, without any agreement no pattern can be formed; with one compass needle that indicates north for all robots any odd number of robots can form an arbitrary pattern, and with two independent compass needles (e.g., north and east - which implies a common chirality) any set of robots can form any pattern. The authors of [Das+10] take a similar approach and investigate under which conditions oblivious robots can form a

*series* of geometric patterns even though the robots are oblivious. In particular, they study series of patterns which can be formed by robot systems under various restrictions such as anonymity, asynchronicity and a lack of common orientation.

In modular self-reconfigurable robotic systems the results of [Hur+15] are of high interest. They present the first distributed and local universal reconfiguration algorithm in the field: i.e., an algorithm that reconfigures any robot shape to any other robot shape with the same number of modules. Previous known theoretical approaches (e.g., [DP06; AK08]) focused on sequential and centralized algorithms.

In cellular automata the problem of pattern formation has been studied extensively. A survey of the early results can be found in [WP94]. Moreover, cellular automata have also been thoroughly investigated in biological pattern formation (see e.g. the books [DD05; YY05]). More recently, researchers have focused on using genetic algorithms to evolve cellular automata to produce predefined shapes [CD06] or to considered variants of the classic CA for pattern formation, such as conservative cellular automata [Ima+03] or reversible cellular automata [IHM02].

In the *nubot* model, one of the main results of [Woo+13] is to efficiently construct two-dimensional geometric shapes in polylogarithmic time in the size of the shape: i.e., they show how to build a computable shape of size $n \times n$ in time polylogarithmic in $n$, plus the time needed to simulate a Turing machine that computes whether or not a given pixel is in the final shape. In [Che+14] the authors show that this fast construction is, to some degree, even possible in a stricter model in which uncontrolled random movements, or agitations, are happening throughout the self-assembly process. More precisely they present a polylogarithmic expected time construction for squares and a sublinear expected time construction to build a line.

**The Amoebot Model** The amoebot model (see Section 2.2) was introduced in 2014 [Der+14] and is a model for self-organizing programmable matter that aims to provide a framework for rigorous algorithmic research for nanoscale systems. Since its introduction it has been utilized in a handful of publications that we will now briefly present. We will omit the publications [Der+17; Der+16a; Der+15a] since they are part of this thesis. In [Der+15b], the

authors describe a leader election algorithm for an abstract (synchronous) version of the amoebot model that decides the problem in expected linear time. The algorithm uses an elaborate token passing scheme to transfer information and makes use of the common chirality of particles. The particles are thus able to elect a leader particle on the unique outer boundary of the particle system (see Section 6.1 for a more detailed description). Recently, a universal shape formation algorithm [Der+16b] was introduced which takes an arbitrary input shape composed of a constant number of equilateral triangles of unit size (called faces) and lets the particles build that shape at a scale depending on the number of particles in the system. The algorithm runs in $O(\sqrt{n})$ asynchronous rounds, which is achieved by building an intermediate structure that allows for an efficient building process (again a more detailed description can be found in Section 6.1). Additionally, Cannon et al. [Can+16] introduced a Markov chain algorithm for the *compression problem*: i.e., the problem of compressing the particle system as much as possible. Due to its nature, the algorithm is stateless and oblivious. Together with the results in this thesis, this shows that there is potential to investigate a wide variety of problems in the model.

# CHAPTER 3

# Universal Coating

> ”*Anything one man can imagine, other men can make real.*”
>
> Jules Verne; Novelist, Poet, and Playwright

Self-organizing programmable matter has a variety of problems that originate from the vision of a malleable material that consists of tiny moving particles. One of those very natural problems for programmable matter is the problem of coating. Imagine a ship that should be coated by a new layer of paint. If that paint consists of programmable matter, the ship can be painted without ever leaving the water, since the paint itself uniformly coats the ship with a thin layer of material. There is a plethora of very similar scenarios in which a coating algorithm for programmable matter facilitates a heavy duty task or — in the case of coating an object contaminated by nuclear radiation — enables solving a task that otherwise would be impossible.

Naturally, a coating algorithm for programmable matter should be as general as possible. It is highly desirable that there is only one universal algorithm that allows for a coating of a broad class of objects, instead of having one specific algorithm that has to be adapted every time we want to coat a different object.

In this chapter we present a worst-case optimal algorithm for universal coating. Our *Universal Coating Algorithm* seamlessly adapts to a broad class of objects, uniformly coating the object by forming multiple coating layers if necessary.

**Chapter Outline** In Section 3.1 we formally define the problem of universal coating. Afterwards, we present our coating algorithm in Section 3.2. We separate the analysis of our algorithm into two parts. Section 3.3 focuses on the correctness of the algorithm: i.e., we show that the algorithm terminates and if it does so, the object is coated. Section 3.4 is concerned with analyzing the worst-case runtime of the algorithm and provides a runtime lower bound for the coating problem itself.

**Chapter Basis** The problem statement, the coating algorithm and its correctness analysis are based on the following journal article:

> **2017** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "Universal Coating for Programmable Matter". In: *Theoretical Computer Science*, cf. [Der+17].

The runtime analysis, as well as the evaluation on lower bounds, are based on the following conference publication:

> **2016** (with Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa and C. Scheideler). "On the Runtime of Universal Coating for Programmable Matter". In: *DNA Computing and Molecular Programming - 22nd International Conference, DNA 22, Munich, Germany, September 4-8, 2016, Proceedings*, cf. [Der+16a].

A full version of the paper is currently under submission (see[Day+ar]).

## 3.1. Problem Statement

In the *universal coating problem* we consider an instance $(P, O)$ where $P$ represents the particle system and $O$ represents the fixed object to be coated. Let $V(P)$ be the set of nodes occupied by $P$, and $V(O)$ be the set of nodes occupied by $O$ (when clear from the context, we may omit the $V(\cdot)$ notation). For any two nodes $v, w \in V_{\text{eqt}}$ (where $V_{\text{eqt}}$is a shorthand for all the nodes in

Figure 3.1.: An example object with a tunnel of width 1.

$G_{\text{eqt}}$), the *distance* $d(v, w)$ between $v$ and $w$ is the length of the shortest path in $G_{\text{eqt}}$ from $v$ to $w$. The distance $d(v, U)$ between a $v \in V_{\text{eqt}}$ and $U \subseteq V_{\text{eqt}}$ is defined as $\min_{w \in U} d(v, w)$. Define *layer* $i$ to be the set of nodes that have a distance $i$ to the object, and let $B_i$ be the number of nodes in layer $i$. An instance is *valid* if the following properties hold:

(a) All particles are initially contracted and are in an *idle* state.

(b) The subgraphs of $G_{\text{eqt}}$ induced by $V(O)$ and $V(P) \cup V(O)$, respectively, are connected: i.e., there is a single object and the particle system is connected to the object.

(c) The subgraph of $G_{\text{eqt}}$ induced by $V_{\text{eqt}} \backslash V(O)$ is connected: i.e., the object $O$ has no holes.

(d) $V_{\text{eqt}} \backslash V(O)$ is $2(\lceil \frac{n}{B_1} \rceil + 1)$-connected: i.e., $O$ does not have *tunnels* of width less than $2(\lceil \frac{n}{B_1} \rceil + 1)$ (see Figure 3.1 for an example of an object with a tunnel of width 1).

Concerning property (c), note that in case $O$ contains holes, we would consider the subset of particles in each connected region of $V_{\text{eqt}} \backslash V(O)$ separately. For property (d), we remark that a tunnel width of at least $2\lceil \frac{n}{B_1} \rceil$ is needed to guarantee that the object can be evenly coated. The coating of narrow tunnels requires specific technical mechanisms that complicate the protocol without contributing to the basic idea of coating, so we ignore such cases in favor of simplicity.

A configuration $C$ is *legal* if and only if all particles are contracted and

$$\min_{v \in V_{\text{eqt}} \backslash (V(P) \cup V(O))} d(v, V(O)) \geq \max_{v \in V(P)} d(v, V(O)),$$

i.e., all particles are as close to the object as possible. Figuratively speaking they *coat $O$ as evenly as possible*. A configuration $C$ is said to be *stable* if no particle in $C$ ever performs a state change or movement. An algorithm *solves* the universal coating problem if, starting from any valid instance, it reaches a *stable legal configuration* in a finite number of rounds.

## 3.2. Universal Coating Algorithm

In this section we present our *Universal Coating Algorithm*. This algorithm is constructed by combining a number of asynchronous primitives which are integrated seamlessly without any underlying synchronization. The *spanning forest* primitive organizes the particles into a spanning forest which determines the movement of particles while preserving system connectivity; the *complaint-based coating* primitive coats the *surface layer* (i.e., layer 1) by bringing particles not yet touching the object into layer 1 while there is still room; the *node-based leader election* primitive elects a node in the surface layer whose occupant becomes the leader particle used to trigger the general layering process for higher layers; and the *general layering* primitive allows each layer $i$ (for $i \geq 2$) to form once layer $i-1$ has been completed. All these primitives are described in detail in Section 3.2.2. However, before we characterize the coating algorithm, we introduce some preliminary notions in Section 3.2.1.

### 3.2.1. Notions

We define the set of *states* that a particle can be in as *idle*, *follower*, *root*, and *retired*. In addition to its state, a particle maintains a constant number of flags, which in our context are constant-size pieces of information visible to neighboring particles. A flag $f$ owned by some particle $p$ is denoted by $p.f$. In our algorithm, we assume that every time a particle contracts, it contracts out of its tail. Therefore, a node occupied by the head of a particle $p$ still is occupied by $p$ after a contraction.

A particle keeps track of its current layer number in the flag $p.layer$. In order to respect the constant-size memory constraint of particles, we take all layer numbers modulo 4. However, for ease of presentation, we omit the modulo 4 computations in the descriptions, except for the pseudocode. Additionally,

each root particle $p$ has a flag $p.down$ which stores a port label pointing to a node of the object if it is on the surface layer, and to an occupied node adjacent to its head in layer $p.layer - 1$ if it is not.

Moreover, $p$ has two additional flags, $p.CW$ and $p.CCW$, which are also port labels. Intuitively, a movement in direction $p.CW$ (resp., $p.CCW$) corresponds to a *clockwise* (resp. *counter-clockwise*) path around the connected structure consisting of the object and retired particles. A particle $p$ can compute $p.CW$ in the following way: Starting from port label $p.down$, $p.CW$ is the first label of $p$ in counter-clockwise order such that (i) the node that the label points is occupied by a particle with the same layer number or (ii) the node is unoccupied. Flag $p.CCW$ can be computed analogously. Note that since particles have a common chirality, they share the same notion of clockwise and counter-clockwise.

We say a layer is *filled* if all nodes in that layer are occupied with retired particles. To enhance readability and to avoid confusion with the modulo computation of particles, we use the term surface layer to refer to layer 1.

### 3.2.2. Coating Primitives

The **spanning forest primitive** (see Algorithm 1) organizes the particles into a spanning forest, which yields a straightforward mechanism for particles to move while preserving connectivity. As already stated, all particles are initially *idle*. A particle $p$ touching the object changes its state to *root*. Any other idle particle $p$ that is activated, evaluates whether it has a root or a follower in its neighborhood. If so, it stores the direction to one of them in $p.parent$ and changes its state to *follower*; otherwise, it remains idle. We say $p$ is the child of $p.parent$. A follower particle $p$ uses handovers to follow its parent and updates the direction $p.parent$ as it moves in order to maintain the same parent in the tree. Note that the particular particle at $p.parent$ may change since the particle occupying the node might perform a handover with another of its children. By using handovers as the only kind of movement, the trees formed by the parent relations stay connected, occupy only the nodes they occupied before, and do not mix with other trees. A root particle $p$ uses the flag $p.dir$ to determine its movement direction. As $p$ moves, it updates $p.dir$ such that it always points to the next node of a clockwise movement

around the object. For any particle $p$, we call the particle occupying the node that $p.parent$ resp. $p.dir$ points to the *predecessor* of $p$. If a root particle does not have a predecessor, we call it a *super-root*.

**Algorithm 1** Spanning Forest Primitive for Coating

A particle $p$ acts depending on its state as described below:

**idle**: If $p$ is adjacent to the object $O$, it becomes a *root* particle, makes the current node it occupies a *leader candidate node*, and starts running the leader election algorithm. If $p$ is adjacent to a *retired* particle, $p$ also becomes a *root* particle. If a neighbor $p'$ is a root or a follower, $p$ sets the flag $p.parent$ to the label of the port to $p'$, puts a *complaint flag* in its local memory, and becomes a *follower*. If none of the above applies, $p$ remains idle.

**follower**: If $p$ is contracted and adjacent to a retired particle or to $O$, then $p$ becomes a *root* particle. If $p$ is contracted and has an expanded parent, then $p$ initiates HANDOVER($p$) (Algorithm 2); otherwise, if $p$ is expanded, it considers the following two cases: (i) if $p$ has a contracted child particle $q$, then $p$ initiates HANDOVER($p$); (ii) if $p$ has no children and no idle neighbor, then $p$ contracts. Finally, if $p$ is contracted, it runs the function FORWARDCOMPLAINT($p, p.parent$) (Algorithm 3).

**root**: If particle $p$ is in the surface layer, $p$ participates in the leader election process. If $p$ is contracted, it first executes MARKINGANDRETIRING($p$) (Algorithm 5) and can become *retired* and possibly also a *marker*, accordingly. If $p$ does not become retired, then if it has an expanded root in $p.dir$, it initiates HANDOVER($p$); otherwise, $p$ calls LAYEREXTENSION($p$) (Algorithm 4). If $p$ is expanded, it considers the following two cases: (i) if $p$ has a contracted child, then $p$ initiates HANDOVER($p$); (ii) if $p$ has no children and no idle neighbor, then $p$ contracts. Finally, if $p$ is contracted, it runs FORWARDCOMPLAINT($p, p.dir$).

**retired**: $p$ clears a potential complaint flag from its memory and performs no further action.

The **complaint-based coating primitive** is used for the coating of the surface layer. This is accomplished by having each particle that becomes a follower generate a *complaint flag*. Complaint flags are forwarded by particles in a pipelined fashion from children to parents through the spanning forest. More precisely, every time a contracted particle $p$ holding at least one complaint flag is activated, it forwards one flag to its predecessor as long as that predecessor

holds less than two complaint flags (see Algorithm 3). We allow each particle to hold up to two complaint flags to ensure that flags quickly move from their origin to the super-roots while respecting the constant-size memory restriction of particles. A contracted super-root $p$ expands to $p.dir$ only if it holds at least one complaint flag. The super-root consumes one of the complaint flags it holds in an expansion. All other roots $p$ move towards $p.dir$ whenever possible (i.e., no complaint flags are required) by performing a handover with their predecessor (which has to be another root) or a successor (which is a root or follower of its tree). When performing a handover with a successor, preference is given to a follower in order to allow additional particles to enter the surface layer (see Algorithm 2). As we will show later, these movement rules ensure that whenever there are particles in the system that are not yet in the surface layer, eventually one of these particles moves to the surface layer, unless the surface layer is already completely filled with contracted particles.

**Algorithm 2** HANDOVER($p$)

```
1: if p is expanded then
2:     if p has at least one contracted child q such that q.parent points to the
   tail of p then
3:         if p.layer = 1 and one child is a follower then
4:             p performs a handover with a follower child
5:         else
6:             p performs a handover with any of its children
7: else
8:     if p is a follower and p.parent is expanded then
9:         p performs a handover with p.parent
10:    if p is a root and p.dir points to an expanded particle q then
11:        p performs a handover with q
```

**Algorithm 3** FORWARDCOMPLAINT($p, i$)

```
1: if p holds at least one complaint flag and the particle q adjacent to p in
   direction i holds less than two complaint flags then
2:     p forwards one complaint flag to q
```

The **leader election primitive** runs in parallel to the complaint-based coating primitive to elect a node in the surface layer as the leader node. This primitive is similar to the algorithm presented in [Day+17] with the difference

that leader candidates are nodes instead of static particles. This distinction is important because in our case particles are moving while leader election is in progress. The primitive terminates only once all nodes in the surface layer are occupied. We will explain the leader election primitive more in detail in Subsection 3.2.3. Once the leader node is determined and all nodes in the surface layer are filled by contracted particles the particle currently occupying that node becomes the *leader*. This leader becomes a special *marker particle*, marking a neighboring node in the next layer by a flag as a *marker node* which determines a starting point for layer 2, and becomes *retired*. Once a contracted root $p$ has a retired particle in the direction $p.dir$, it retires as well. This causes the particles in the surface layer to become retired in counter-clockwise order. At this point, the general layering primitive becomes active, which builds subsequent layers until there are no longer followers in the system. If the leader election primitive does not terminate (which happens only if $n < B_1$ and the surface layer is never completely filled), then the complaint flags ensure that the super-roots eventually stop, which eventually results in a stable legal coating.

**Algorithm 4** LayerExtension($p$)

**Calculating** $p.layer$**,** $p.down$ **and** $p.dir$

1: The layer number of any node occupied by the object is equal to 0.
2: Let $q$ be any neighbor of $p$ with smallest layer number (modulo 4).
3: $p.down \leftarrow p$'s label for port leading to $q$
4: $p.layer = (q.layer + 1) \mod 4$
5: Computes CW & CCW directions
6: **if** $p.layer$ is *odd* **then**
7: $\quad p.dir \leftarrow p.CW$
8: **else**
9: $\quad p.dir \leftarrow p.CCW$

**Extending layer** $p.layer$

10: **if** the node at $p.dir$ is unoccupied, and either $p$ is not on the surface layer or $p$ holds a complaint flag **then**
11: $\quad p$ expands in direction $p.dir$
12: $\quad p$ consumes a complaint flag, if it holds one

In the **general layering primitive** a follower becomes a root, whenever it is adjacent to a retired particle. Followers follow their parents as before.

Complaint flags are no longer needed to expand into empty nodes. Root particles continue to move along nodes of their layer in a clockwise direction (if the layer number is odd) or counter-clockwise direction (if the layer number is even) until they reach either the marker node of that layer, a retired particle in that layer, or an empty node of the lower layer. In the latter case the particles move in the lower layer, which causes them to change direction. A contracted root particle $p$ may retire if: (i) it occupies the marker node and the marker particle in the lower layer signals that all particles in that layer are retired (which it can determine locally), or (ii) it has a retired particle in the direction $p.dir$. Once a particle retires on a marker node, it becomes the marker particle for that layer and marks a neighboring node in the next layer as a marker node.

**Algorithm 5** MarkingAndRetiring($p$)

**First Marker Condition:**

**if** $p$ is the *leader* **then**
    $p$ becomes a *retired* particle
    $p$ sets the flag $p.marker$ to be the label of a port leading to a node guaranteed not to be in the surface layer — e.g., by taking the average direction of $p$'s two neighbors in the surface layer (by now complete)

**Extending Layer Markers:**

**if** $p$ is connected to a marker $q$ and the port $q.marker$ points towards $p$ **then**
    **if** both $q.CW$ and $q.CCW$ are retired **then**
        $p$ becomes a *retired* particle
        $p$ sets the flag $p.marker$ to the label of the port opposite the port connecting $p$ to $q$

**Retiring Condition:**

**if** the node in direction $p.dir$ is occupied by a retired particle **then**
    $p$ becomes a *retired* particle

### 3.2.3. Leader Election Primitive

We are now describing the leader election primitive in more detail. As already stated the primitive is similar to the algorithm presented in [Day+17] and is

used for electing a leader among the particles that touch the object: i.e., only particles in the surface layer participate in the leader election process. A leader only emerges if $B_1 \leq n$.

The leader election algorithm we use in this thesis is a slightly modified version of the leader election algorithm presented in [Day+17]. The tokens described in [Day+17] can be seen as flags in our algorithm. However, the algorithm of [Day+17] cannot be directly applied, since it is executed on a static particle system: i.e., the particles do not move. Consequently, for the purpose of universal coating, we abstract the leader election algorithm to conceptually run on the *nodes* of the surface layer since these are static. Thus, we elect a leader node and a contracted particle occupying that node becomes the leader. Particles on the surface layer provide the means for running the leader election process on the respective nodes: i.e., they store and transfer all flags that are needed for the leader algorithm. If a particle is expanded, it is responsible for both nodes it occupies: i.e., an expanded particle emulates the leader election process for two nodes simultaneously.

To be more precise, we are now going to specify how the information of the leader election is transferred if particles move. An expanded particle $p$ on the surface layer, whose tail also occupies a node $v$ on the surface layer and that is about to perform a handover with contracted particle $q$, passes all the leader information associated with node $v$ to $q$ in that handover. If there is no particle to perform a handover with, the expanded particle does not contract. If a particle $p$ occupying a node $v$ wants to forward some leader election information to an adjacent node $w$ (according to [Day+17]) that is currently unoccupied, it waits until either $p$ itself expands into $w$, or another particle occupies node $w$. Note that once a node $v$ in the surface layer is occupied at some time $t$, then $v$ is occupied at all timesteps $t' > t$.

The leader election can terminate only once all nodes in the surface layer are occupied. According to [Day+17] it does so w.h.p.[1] after $\mathcal{O}(n)$ rounds. Once the leader node of $B_1$ is elected, an contracted particle $p$ occupying this node checks whether the surface layer is completely filled with contracted particles. To do so, it generates a *check-flag* that is sent along the surface layer in CCW direction. The flag is forwarded to the next particle only if the neighbor in

[1] An event occurs *with high probability* (*w.h.p.*), if the probability of success is at least $1 - n^{-c}$, where $c > 1$ is a constant.

CCW direction is contracted. Additionally a particle occupying the leader node generates a *clear-flag* every time it expands. The clear-flag is forwarded in CW direction. Whenever a check-flag and a clear-flag meet they cancel each other out. Therefore, a check-flag can return back to the contracted particle $p$ occupying the leader node only if all particles on the surface layer are contracted. In that case $p$ declares itself the leader and the general layering primitive as described in Subsection 3.2.2 starts.

## 3.3. Correctness

In this section we show that our algorithm eventually solves the coating problem.

Let an *active* particle be a particle in either follower or root state. We call an active particle a *boundary particle* if it has the object or at least one retired particle in its neighborhood, otherwise it is a *non-boundary particle*. By definition, a boundary particle is either a root or a follower, whereas non-boundary particles are always followers.

Given a configuration $C$, we define a directed graph $A(C)$ over all nodes in $G_{\text{eqt}}$ occupied by *active* (follower or root) particles in $C$. For every expanded active particle $p$ in $C$, $A(C)$ contains a directed edge from the tail to the head node of $p$. For every follower $p$, $A(C)$ has a directed edge from the head of $p$ to $p.parent$. For the purposes of constructing $A(C)$, we also define parents for root particles: a root particle $p$ sets $p.parent$ to be the active particle $q$ occupying the node in direction $p.dir$ once $p$ has performed its first handover expansion with $q$. For every root particle $p$, $A(C)$ has a directed edge from the head of $p$ to $p.parent$, if it exists. The *ancestors* of a particle $p$ are all nodes reachable by a path from the head of $p$ in $A(C)$. The super-roots defined in Section 3.2.2 correspond to the roots of the trees in $A(C)$. Certainly, since every node has at most one outgoing edge in $A(C)$, the nodes of $A(C)$ can only form a collection of disjoint trees or a *ring of trees* (i.e., a connected graph consisting of a single directed cycle with trees rooted at it) as we show in Lemma 3.2.

First, we prove several safety conditions (see Subsection 3.3.1), and then we prove various liveness conditions (see Subsection 3.3.2) that together allow us to prove that our algorithm solves the coating problem (see Subsection 3.3.3).

### 3.3.1. Safety

Suppose that we start with a valid instance $(P, O)$: i.e., all particles in $P$ are initially contracted, are idle and $V(P) \cup V(O)$ forms a single connected component in $G_{\text{eqt}}$. In the following we show that $V(P) \cup V(O)$ stays connected at any time. We first start with the retired particles.

**Lemma 3.1.** *If $B_1 < n$ the set of retired particles always forms completely filled layers except for possibly the current topmost layer $\ell$, which is filled with a consecutive row of retired particles.*

*Proof.* From our algorithm and since $B_1 < n$, it follows that the first particle that retires is the leader particle, setting its marker flag in a direction to a node not in the surface layer. The particles in the surface layer then retire starting from the leader in CCW direction around the object. Once all particles in the surface layer are retired, the first particle that occupies the marker node of layer 2 retires and becomes the marker particle on layer 2, extending its marker flag in the same direction as the original flag of the leader. Starting from the marker particle in layer 2, boundary particles can retire in CW direction along layer 2. Once all particles in layer 2 are retired, the next layer starts forming. This process continues inductively layer by layer, thereby proving the lemma. □

Next we investigate the active particles, by characterizing the structure of $A(C)$.

**Lemma 3.2.** *At any time, $A(C)$ is a forest or a ring of trees. Each super-root is connected to the object or to a retired particle.*

*Proof.* An active particle can either be a follower or a root. First, we show the following lemma.

**Lemma 3.3.** *At any time, $A(C)$ restricted to non-boundary particles forms a forest.*

*Proof.* Let $A'(C)$ be the induced subgraph of $A(C)$ by the non-boundary particles only. Certainly, when all particles are still idle, the claim holds. So suppose that the claim holds up to time $t$. We show that it then also holds at time $t + 1$. Suppose that at time $t + 1$ an idle particle $p$ becomes active. If $p$

becomes a root, it is by definition not part of $A'(C)$. If it is a non-boundary particle (i.e., a follower), it sets $p.parent$ to a node occupied by a particle $q$ that is already active, so it extends the tree of $q$ by a new leaf, thereby maintaining the tree.

Edges of $A'(C)$ can change only if followers move. If a follower contracts without performing a handover (i.e., it is a leaf), $A'(C)$ trivially stays a forest. If a follower moves in a handover but stays a non-boundary particle, its incident may change, but $A'(C)$ also remains a forest. Finally the set of nodes of $A'(C)$ may change if a particle becomes a boundary particle. In that case the node and its incident edges are removed, but the remaining subgraph of $A'(C)$ stays a forest. □

Next we consider $A(C)$ restricted to boundary particles.

**Lemma 3.4.** *At any time, $A(C)$ restricted to boundary particles forms a forest or a ring.*

*Proof.* The boundary particles always occupy nodes adjacent to retired particles or the object. Therefore, boundary particles either all lie in a single layer or in two consecutive layers according to Lemma 3.1 if $B_1 < n$ or by a simple observation if $B_1 \geq n$. Since the layer numbers uniquely specify the movement direction of the particles, connected boundary particles within a layer can only form a directed line or a directed cycle in $A(C)$. Therefore, if all boundary particles are all in the same layer, the claim holds.

If boundary particles are in in two consecutive layers $\ell$ and $\ell - 1$, then layer $\ell - 1$ has to contain at least one retired particle: i.e., $A(C)$ restricted to boundary particles in layer $\ell - 1$ can only form one or more directed lines, but no ring. Note that by definition several lines are also a forest. Furthermore, all retired particles on layer $\ell - 1$ form a consecutive row according to Lemma 3.1. Therefore, $A(C)$ restricted to boundary particles in layer $\ell$ can also only form one or more directed lines. Finally, one boundary particle on layer $\ell$ can have an edge to a particle in layer $\ell - 1$, since $p.dir$ of a boundary particle $p$ can only point to the same or the next lower layer of $p$. This implies that in this case $A(C)$ restricted to the nodes occupied by all boundary particles forms a forest. □

Since a boundary particle $p$ never has an edge to a non-boundary particle the way $p.dir$ is defined, and a follower without an outgoing edge in $A(C)$ restricted to the non-boundary particles must have an outgoing edge to a boundary particle (otherwise it is a boundary particle itself), $A(C)$ is a forest or a ring of trees.

The second statement of the lemma follows from the fact that every boundary particle must be connected to the object or a retired particle. □

Finally, we investigate the structure formed by the idle particles.

**Lemma 3.5.** *At any time, every connected component of idle particles is connected to at least one non-idle particle or the object.*

*Proof.* Initially, the lemma holds by the definition of a valid instance. Suppose that the lemma holds at time $t$ and consider a connected component of idle particles. If one of the idle particles in the component is activated, it may either stay idle or change to an active particle. In both cases the lemma holds at time $t + 1$. If a retired particle that is connected to the component is activated, it does not move. If a follower or root particle that is connected to the component is activated, the particle can move. If it expands or does not move, it still occupies the node adjacent to the connected component. If it contracts, according to our algorithm that particle cannot contract outside of a handover with another follower or root particle. This implies that after the contraction the node adjacent to the connected component is still occupied. So in any of these cases, the connected component of idle particles remains connected to a non-idle particle. Therefore, the lemma holds at time $t + 1$. □

The following corollary is consequence of the previous three lemmas.

**Corollary 3.6.** *At any time, $V(P) \cup V(O)$ forms a single connected component.*

We close this section by proving an equality that relates the number of complaint flags, expanded boundary particles and number of non-boundary particles in a connected component of $A(C)$

**Lemma 3.7.** *Consider a connected component $G$ of $A(C)$. At any time before the first particle retires, the number of expanded boundary particles in $G$ plus the number of complaint flags in $G$ equals the number of non-boundary particles in $G$.*

*Proof.* Initially, the lemma holds since all particles are contracted and idle. Suppose the lemma holds at time $t$ and consider the next activation of a particle. In the following we only discuss relevant cases of the algorithm.

(a) If an idle particle becomes a non-boundary particle (i.e., it is not connected to the object but becomes a part of a connected component in $A(C)$), it also generates a complaint flag. So both the number of non-boundary particles and the number of complaint flags increase by one for the certain component.

(b) If a non-boundary particle expands as part of a handover with a boundary particle, both the number of expanded boundary particles and the number of non-boundary particles decrease by one for the component.

(c) If a boundary particle expands as part of a handover, that handover must be with another boundary particle, so the number of expanded boundary particles remains unchanged for that component.

(d) By our assumption there are no retired particles, so all boundary particles are in the surface layer. Hence, a boundary particle can only expand outside of a handover by consuming a complaint flag. This increases the number of expanded boundary particles by one and decreases the number of complaint flags by one.

(e) Finally, an expansion of a boundary particle outside of a handover can connect two components of $A(C)$. Since the equation given in the lemma holds for each of these components individually, it also holds for the newly formed component of $A(C)$. □

### 3.3.2. Liveness

We say that the particle system *makes progress* if (i) an idle particle becomes active, (ii) a movement (i.e., an expansion, handover, or contraction) is executed, or (iii) an active particle retires. Before we show under which circumstances our particle system eventually makes progress, we first show some propositions of how particles behave during the execution of our algorithm.

**Lemma 3.8.** *Eventually, every idle particle becomes active.*

*Proof.* As long as an idle particle exists, at least one idle particle $p$ is connected to a non-idle particle or the object according to Lemma 3.5. The next time $p$ is activated $p$ becomes active according to Algorithm 1. This proves the statement. □

The following statement shows that even though roots of $A(C)$ can temporarily be followers, they become root particles the next time they are activated.

**Lemma 3.9.** *In every tree of $A(C)$, every boundary particle in the follower state enters a root state the next time it is activated.*

*Proof.* Let $p$ be a boundary particle in the follower state. By definition $p$ must have a retired particle or the object in its neighborhood. Therefore, $p$ immediately becomes a root particle once it is activated according to Algorithm 1. □

The following is a direct consequence of Lemma 3.9.

**Corollary 3.10.** *Every super-root that is in the follower state enters the root state the next time it is activated.*

Furthermore, the following lemma provides a relation between the movement of super-roots and the availability of complaint flags.

**Lemma 3.11.** *For every tree of $A(C)$ with at least one complaint flag and a contracted super-root $p$, $p$ either eventually retires or expands in direction $p.dir$, thereby consuming a complaint flag, and $p$ may cease to be a super-root after the expansion.*

*Proof.* If $p$ is not a root, it becomes one the next time it is activated according to Corollary 3.10. Therefore, assume $p$ is a root. If there is a retired particle in $p.dir$, $p$ retires and the statements hold. If the node in $p.dir$ is unoccupied, $p$ can potentially expand. According to Algorithm 3, complaint flags are forwarded along the tree rooted at $p$ towards $p$ itself. Once a flag reaches $p$, it expands, thereby consuming the flag. If it does so, it might have an active particle in its movement direction and thus ceases to be a super-root. □

Next, we prove the statement that expanded particles do not starve: i.e., they eventually contract.

**Lemma 3.12.** *Eventually, every expanded particle contracts.*

*Proof.* Consider an expanded particle $p$ in a configuration $C$. By Lemma 3.8 we assume w.l.o.g. that all particles in $C$ are active or retired. If $p$ has no children in $A(C)$, then it can contract once it is activated. If there exists at least one child $q$ which is contracted, $p$ contracts in a handover (see Algorithm 2). If all children are expanded, we consider the tree of $A(C)$ that $p$ is part of. Consider one subpath of this tree that starts in $p$ $(v_1, v_2, \dots, v_k)$ such that $v_1, v_2$ are occupied by $p$ and $v_k$ is a node that does not have an incoming edge in $A(C)$. Let $v_i$ be the first node of this path that is occupied by a contracted particle. If all particles are expanded, then clearly the last particle occupying $v_{k-1}, v_k$ eventually contracts and $i = k-1$. Since $v_i$ is contracted it eventually performs a handover with the particle occupying $v_{i-2}, v_{i-1}$. Now we move inductively backwards along $(v_1, v_2, \dots, v_{i-1})$. It is guaranteed that a contracted particle eventually performs a handover with the expanded particle occupying the two nodes before it on the path. Therefore, a child of $p$ is eventually contracted and performs a handover with $p$. □

In the following two lemmas we specifically consider the case that $B \leq n$: i.e., the particles can coat at least the complete surface layer.

**Lemma 3.13.** *If $B \leq n$, the surface layer is eventually completely filled with contracted particles.*

*Proof.* Consider a configuration $C$ such that the surface layer is not completely filled with contracted particles. In this case the leader election cannot succeed, since it requires that all particles on the surface layer are contracted. Consequently, no particle is retired in $C$. By Lemma 3.8 we can assume w.l.o.g. that all particles in configuration $C$ are active. Since the surface layer is not completely filled by contracted particles, there is either at least one unoccupied node $v$ on the surface layer or all nodes are occupied, but there is at least one expanded particle on the surface layer. We show that in both cases a follower moves to the surface layer. Thereby, the layer is filled up until all particles on it are contracted.

In the first case, let $p$ be the super-root of a tree in $A(C)$ that still has non-boundary particles. Additionally, let $(p_0 = p, p_1, \dots, p_k)$ be the boundary particles of the tree, such that $p_{i-1}$ occupies the node in $p_i.dir$ and let $q$ be a non-boundary particle in the tree that is adjacent to some $p_j \in (p_0, \dots, p_k)$

such that $j$ is minimal. If a particle $p_i$ in $(p_0, \dots, p_j = q.parent)$ is expanded, it eventually contracts by a handover with $p_{i+1}$ (Lemma 3.12), and by consecutive handovers all particles in $(p_{i+1}, \dots, p_j)$ eventually expand and contract until the particle $p_j = q.parent$ expands. According to Algorithm 2, $p_j$ has to perform a handover with $q$. Therefore, the number of particles on the surface layer increases. If all particles in $(p_0, \dots, q.parent)$ are contracted, then by Lemma 3.7 a complaint flag still exists in the tree. Thus, $p$ eventually expands by Lemma 3.11. Consequently, we are back in the former case that a particle in $(p_0, \dots, q.parent)$ is expanded.

In the second case, let $p'$ be an expanded boundary particle and let $q'$ be the non-boundary particle with the shortest path in $A(C)$ to $p'$. By a similar argument as for the first case, particles on the surface layer perform handovers (starting with $p'$) until eventually the node in $q'.parent$ is occupied by a tail. Again, $q'$ eventually performs a handover and the number of particles on the surface layer has increased. □

As a direct consequence, we can show the following lemma.

**Lemma 3.14.** *If $B \leq n$, a leader particle is eventually elected in the surface layer.*

*Proof.* According to Lemma 3.13 the surface layer is eventually filled with contracted particles. Leader election successfully elects a leader node according to [Day+17]. The contracted particle $p$ occupying the leader node creates and forwards the check-flag and, since all particles on the surface layer are contracted, eventually receives it back. Consequently, $p$ becomes a leader. □

Now we are ready to prove the two major statements of this subsection that define two conditions for system progress.

**Lemma 3.15.** *If there are no retired particles and there is either a complaint flag or an expanded particle, the system eventually makes progress.*

*Proof.* If there is an idle particle, progress is ensured by Lemma 3.8. If an active particle is expanded Lemma 3.12 guarantees progress. Finally, in the last case all particles are active, none of them is expanded and there is a complaint flag. If the surface layer is completely filled, a leader is elected according to Lemma 3.14 and as a direct consequence the active particles on

the surface layer eventually retire, guaranteeing progress. If the surface layer is not completely filled, there exists at least one tree of $A(C)$ with a contracted super-root $p$ that has an unoccupied node in $p.dir$ and at least one complaint flag in the tree. Therefore, progress is ensured by Lemma 3.11. □

**Lemma 3.16.** *If there is at least one retired particle and one active particle, the system eventually makes progress.*

*Proof.* Again, if there is an idle particle, progress is ensured by Lemma 3.8. Moreover, note that since there is at least one retired particle, we can conclude that leader election has been successful (since the first particle that retires is the leader particle) if $B_1 < n$. If there is still a non-retired particle on the surface layer, it eventually retires according to the Algorithm, guaranteeing progress.

So suppose that all particles in the surface layer are retired. We distinguish between the following cases: (i) there exists at least one super-root, (ii) no super-root exists, but there is an expanded particle, and (iii) no super-root exists and all particles are contracted. In case (i), Corollary 3.10 guarantees that a super-root eventually enters the root state; therefore it eventually either expands (if $p.dir$ is unoccupied) or retires (if $p.dir$ is occupied by a retired particle or $p$ occupies a marked node). In case (ii), the expanded particle contracts according to Lemma 3.12. In case (iii), $A(C)$ forms a ring of trees, which can happen only if all boundary particles completely occupy a single layer, so there is an active particle that occupies the marker node on that layer. Since it is contracted by assumption, it retires upon activation.

Therefore, in all three cases the system eventually makes progress. □

### 3.3.3. Termination

Finally, we show that the algorithm eventually terminates in a legal configuration: i.e., a configuration in which the coating problem is solved. In order to show termination, we need the following two lemmas.

**Lemma 3.17.** *The number of times an idle particle turns active and an active particle becomes retired is bounded by $n$.*

*Proof.* From our algorithm it immediately follows that every idle particle is transformed only once into an active particle, and every active particle is

transformed only once into a retired particle. Moreover, a non-idle particle can never become idle again, and a retired particle can never become non-retired again, which proves the lemma. □

**Lemma 3.18.** *The overall number of expansions, handovers, and contractions in our algorithm is $\mathcal{O}(n^2)$.*

*Proof.* We prove the statement by showing that a single particle can only perform a linear amount of movements as a follower and also only a linear amount of movements as a root. We need the following observation, which immediately follows from our algorithm.

**Observation 3.19.** Only a super-root can expand to a non-occupied node, and every such expansion triggers a sequence of handovers, followed by a contraction in which every particle participates at most twice.

Consider any particle $p$. Note that only an active particle performs a movement. Let $C$ be the first configuration in which $p$ becomes active. If it is a non-boundary particle (i.e., a follower), then consider the directed path in $A(C)$ from the head of $p$ to the super-root $r$ of its tree or the first particle $r$ belonging to the ring in the ring of trees. Such a path must exist due to Lemma 3.2. Let $\mathcal{P} = (v_0, v_1, \dots, v_m)$ be the node sequence covered by this path where $v_0$ is the head of $p$ in $C$ and $v_m$ is the first node along that path with the object or a retired particle in its neighborhood. By Lemma 3.2 such a node sequence is well-defined since at least the node occupied by $r$ fulfills this condition. The length of $\mathcal{P}$: i.e., $|\mathcal{P}|$ is at most $2n$. According to Algorithm 1, $p$ attempts to follow $\mathcal{P}$ by sequentially expanding into the nodes $v_1, \dots, v_m$. In the worst case, $p$ becomes a boundary particle once it reaches $v_m$. Up to this point, $p$ has traveled along a path of length at most $2n$; therefore, the number of movements $p$ executes as a follower is $\mathcal{O}(n)$.

Now suppose $p$ is a boundary particle,; therefore in the root state. Let $C$ be the configuration in which $p$ becomes a boundary particle and let $\ell = p.layer$. Suppose that $\ell = 1$. From our algorithm we know that at most $n$ complaint flags are generated by the particles; therefore by Lemma 3.11 there are at most $n$ expansions on the surface layer. All other movements are handovers or contractions. Hence, it follows from Observation 3.19 that $p$ can only move $\mathcal{O}(n)$ times as a root on the surface layer.

Now consider the case that $\ell > 1$. Here we need the following observation, which is a direct consequence of the underlying triangular grid graph.

**Observation 3.20.** For every $i$ and every valid instance $(P, O)$ allowing $O$ to be coated by $i$ layers it holds that $B_i = B_0 + 6i$.

If $\ell = 2$, there must be a retired particle in the surface layer, and since the leader is the first particle that retires, Lemmas 3.13 and 3.14 imply that the surface layer is completely filled with contracted particles. So $p$ can only move along nodes of layer 2. Since $B_1 \leq n$, it follows from Observation 3.20 that $B_2 \leq n + 6$. If some particles on the surface layer are not retired, $p$ cannot move beyond the marker node in layer 2. So $p$ either becomes retired before reaching the marker node, or if it reaches the marker node, it has to wait there until all particles on the surface layer are retired, which causes its retirement. Therefore, $p$ can only move along at most $n + 6$ nodes. If $\ell > 2$, we know from Lemma 3.1 that layer $\ell - 2$ is completely filled with contracted particles. Since $B_{\ell-2} \leq n$ and $B_\ell = B_{\ell-2} + 12$, it follows that $B_\ell \leq n + 12$. Hence, $p$ moves along at most $n + 12$ nodes in layer $\ell$ before retiring. Alternatively, $p$ might move to layer $\ell - 1$, and $p$ moves along at most $n + 6$ further nodes in layer $\ell - 1$ before retiring. Thus, in any case, $p$ performs at most $\mathcal{O}(n)$ movements as a root particle.

Putting it all together, any particle makes at most a linear amount of movements as a follower, followed by at most a linear amount of movements as a root. Therefore, the number of movements any particle in the system performs is $\mathcal{O}(n)$. □

Lemmas 3.17 and 3.18 imply that the system can only make progress $\mathcal{O}(n^2)$ many times. Hence, eventually our system reaches a configuration in which it no longer makes progress and the algorithm terminates. It remains to show that when the algorithm terminates, the particle system is in a legal configuration: i.e., the algorithm solves the coating problem.

**Theorem 3.21.** *Our Universal Coating Algorithm terminates in a legal configuration.*

*Proof.* From the conditions of Lemmas 3.15 and 3.16 we know that the following statements hold when the algorithm terminates:

(a) Either all particles are retired or all particles are active (see Lemma 3.16).

(b) If all particles are active, there is neither a complaint flag nor an expanded particle in the system (see Lemma 3.15).

First suppose that all particle are retired. Then it follows directly from Lemma 3.1 that the configuration is legal. Next, suppose that all particles are active and contracted, and there is no complaint flag in the system. Then Lemma 3.7 implies that all active particles must be boundary particles. If there is at least one boundary particle in a layer $\ell > 1$, then there must be at least one retired particle, contradicting our assumption. Therefore, all boundary particles are in the surface layer, and since there are no more complaint flags and all boundary particles are contracted, the particle system is in a legal configuration, which proves our theorem. □

This concludes our correctness analysis of our Universal Coating Algorithm.

## 3.4. Runtime Analysis

Before we investigate the runtime of our specific coating algorithm, we first present a lower bound concerning the runtime of any coating algorithm in Subsection 3.4.1. Afterwards, we analyze the worst-case runtime of our Universal Coating Algorithm in Subsection 3.4.2.

### 3.4.1. Runtime Lower Bounds

Recall that a *round* is over once every particle in $P$ has been activated at least once. The *runtime* $T_{\mathcal{A}}(P, O)$ of a coating algorithm $\mathcal{A}$ is defined as the worst-case number of rounds (over all sequences of particle activations) required for $\mathcal{A}$ to solve the universal coating problem $(P, O)$. Certainly, there are instances $(P, O)$ where every coating algorithm has a runtime of $\Omega(n)$ (see Lemma 3.22), though there are also many other instances where the universal coating problem can be solved much faster.

**Lemma 3.22.** *The worst-case runtime required by any local-control algorithm to solve the universal coating problem is $\Omega(n)$.*

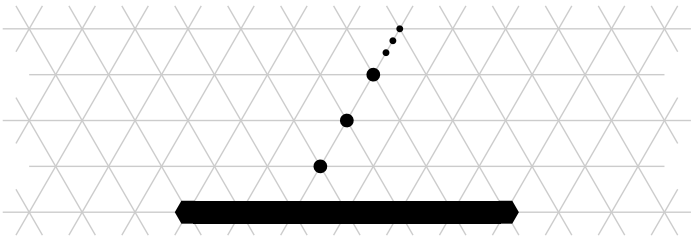


Figure 3.2.: Worst-case configuration concerning the number of rounds. There are $n$ particles (black dots) in a line connected to the object by a single particle.

*Proof.* Assume the particles $p_1, \ldots, p_n$ form a single line of $n$ particles connected to the object of the object via $p_1$ (Figure 3.2). Suppose $B_1 > n$. Since $d(p_n, O) = n$, it takes $\Omega(n)$ rounds in the worst-case (requiring $\Theta(n)$ movements) until $p_n$ touches the object's surface. This worst-case can happen, for example, if $p_n$ performs no more than one movement (either an expansion or a contraction) per round. □

Since a worst-case runtime of $\Omega(n)$ is fairly large, and therefore not very helpful to distinguish between different coating algorithms, one could study the runtime of coating algorithms relative to the best possible runtime. Unfortunately, a large lower bound also holds for the competitiveness of any local-control algorithm. A coating algorithm $\mathcal{A}$ is called *c-competitive* if for any valid instance $(P, O)$,

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

where $\mathrm{OPT}(P, O)$ is the minimum runtime needed to solve the universal coating problem $(P, O)$ and $k$ is a value independent of $(P, O)$.

**Theorem 3.23.** *Any local-control algorithm that solves the universal coating problem has a competitive ratio of $\Omega(n)$.*

*Proof.* We construct an instance of a coating problem $(P, O)$ which can be solved by an optimal algorithm in $\mathcal{O}(1)$ rounds, but requires any local-control algorithm to take $\Omega(n)$ times longer. Let $O$ be a straight line of arbitrary finite length, and let $P$ be a set of particles which entirely occupy the surface layer, with the exception of one unoccupied node below $O$ equidistant from its

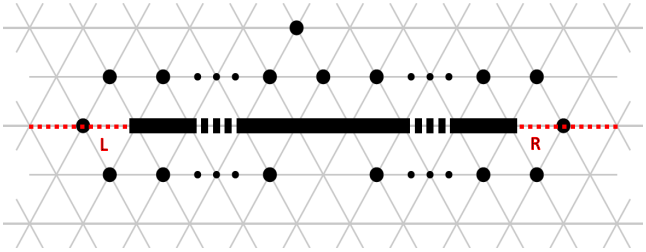



Figure 3.3.: The object occupies a straight line in $G_{\text{eqt}}$. The particles are all contracted and occupy the nodes around the object, with the exception that there is one unoccupied node below the object and one extra particle above the object. Borders $L$ and $R$ are shown as red lines.

endpoints and one additional particle above $O$ in layer 2 equidistant from its endpoints (see Figure 3.3).

An optimal algorithm could move the particles to solve the coating problem for the given example in $\mathcal{O}(1)$ rounds, as shown in Figure 3.4. Note that the optimal algorithm always maintains the connectivity of the particle system, so its runtime is valid even under the constraint that any connected component of particles must stay connected. However, for our local-control algorithms we allow particles to disconnect from the rest of the system.

Now consider an arbitrary local-control algorithm $A$ for the coating problem. Given a round $r$, we define the *imbalance* $\phi_L(r)$ at border $L$ as the net number of particles that have crossed $L$ from the top of $O$ to the bottom until round $r$; similarly, the imbalance $\phi_R(r)$ at border $R$ is defined to be the net number of particles that have crossed $R$ from the bottom of $O$ to the top until round $r$.

Certainly, there is an activation sequence in which information and particles can only travel a distance of up to $n/4$ nodes towards $L$ or $R$ within the first $n/4$ rounds. Hence, for any $r \leq n/4$, the probability distributions of $\phi_L(r)$ and $\phi_R(r)$ are independent of each other. Additionally, particles up to a distance of $n/4$ from $L$ and $R$ cannot distinguish between which border they are closer to, since the node of the gap is equidistant from the borders. This symmetry also implies that $\Pr[\phi_L(r) = k] = \Pr[\phi_R(r) = k]$ for any integer $k$. Let us focus on round $r = n/4$. We distinguish between the following cases.

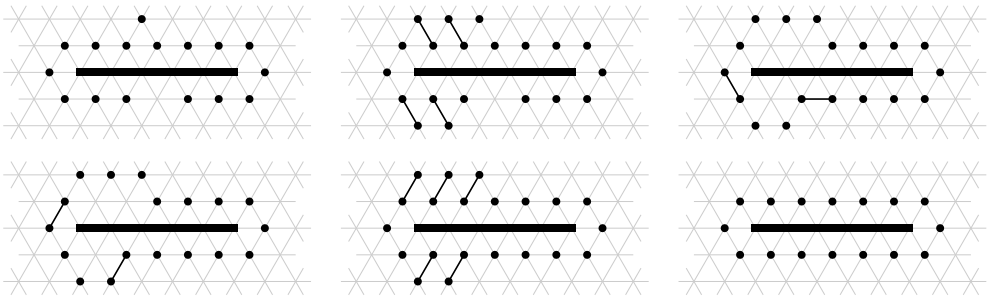


Figure 3.4.: Each subfigure represents the configuration of the system at the beginning of a round and are ordered from left to right, top to bottom. After 5 rounds (i.e., at the beginning of the sixth round) the object is coated. Note that the implied algorithm can be adapted to any length of the object and always requires only 5 rounds to solve the coating problem.

(a) $\phi_L(n/4) = \phi_R(n/4)$. Then there are more particles than nodes in the surface layer above $O$, so the coating problem cannot be solved yet.

(b) $\phi_L(n/4) \neq \phi_R(n/4)$. From our insights above we know that for any two values $k_1$ and $k_2$, $\Pr[\phi_L(n/4) = k_1 \text{ and } \phi_R(n/4) = k_2] = \Pr[\phi_L(n/4) = k_2 \text{ and } \phi_R(n/4) = k_1]$. Hence, the cumulative probability of all outcomes where $\phi_L(n/4) < \phi_R(n/4)$ is equal to the cumulative probability of all outcomes where $\phi_L(n/4) > \phi_R(n/4)$. If $\phi_L(n/4) < \phi_R(n/4)$, then there are again more particles than nodes in the surface layer above $O$, so the coating problem cannot be solved yet.

Thus, the probability that $\mathcal{A}$ has not solved the coating problem after $n/4$ rounds is at least $1/2$; therefore $\mathrm{E}[T_{\mathcal{A}}(P, O)] \geq 1/2 \cdot n/4 = n/8$. Since, on the other hand, $\mathrm{OPT} = \mathcal{O}(1)$, we have established a linear competitive ratio. □

Therefore, even the competitive ratio can be very high in the worst case. As a consequence, we will only study the worst-case runtime of our coating algorithm in the following section. As it turns out, our algorithm is worst-case optimal up to constant factors and is therefore as performant as possible

### 3.4.2. Worst-Case Number of Rounds

In this section, we show that our algorithm solves the coating problem within a linear number of rounds w.h.p. In doing so, we present a simpler synchronous parallel model for particle activations that we can use to analyze the worst-case number of rounds in Subsection 3.4.2.1. Subsection 3.4.2.2 presents the analysis of the number of rounds required to coat the surface layer. Finally, in Subsection 3.4.2.3, we analyze the number of rounds required to fill all other coating layers once the surface layer has been filled.

Recall that $B_i$ denotes the number of nodes in $G_{\text{eqt}}$ at distance $i$ from object $O$ (i.e., the number of nodes in layer $i$). Let $N$ be the layer number of the final layer for $n$ particles: i.e., $N$ satisfies $\sum_{j=1}^{N-1} B_j < n \leq \sum_{j=1}^{N} B_j$. Layer $i$ is said to be *complete* if every node in layer $i$ is occupied by a contracted retired particle (for $i < N$), or if all particles have reached a node such that they are contracted and never move again (for $i = N$). Throughout the analysis we revisit the notion of the graph $A(C)$ as defined in Section 3.3.

In order to precisely argue about the different kind of movements that a particle can perform, we make our notation of movements a bit more explicit. A movement executed by a particle $p$ can be either a *sole contraction* in which $p$ contracts and leaves a node unoccupied, a *sole expansion* in which $p$ expands into an adjacent unoccupied node, a *handover contraction with* $p'$ in which $p$ contracts and forces its contracted neighbor $p'$ to expand into the node it vacates, or a *handover expansion with* $p'$ in which $p$ expands into a node currently occupied by its expanded neighbor $p'$, forcing $p'$ to contract.

#### 3.4.2.1. From Asynchronous to Parallel Schedules

In this section, we show that instead of analyzing our algorithm for asynchronous activations of particles, it suffices to consider a much simpler model of parallel activations of particles. The idea we use here is an extension of the technique we developed in [Der+16b]. We define a *movement schedule* to be a sequence of particle system configurations $(C_0, \dots, C_t)$.

**Definition 3.24.** A movement schedule $(C_0, \dots, C_t)$ is called a *parallel schedule* if (i) in each $C_i$ every particle is either expanded or contracted, and every node of $G_{\text{eqt}}$ is occupied by at most one particle and (ii) for every $i \geq 0, C_{i+1}$ is

reached from $C_i$ such that for every particle $p$ one of the following properties holds:

(a) $p$ occupies the same node(s) in $C_i$ and $C_{i+1}$,

(b) $p$ expands into an adjacent node that was empty in $C_i$,

(c) $p$ contracts, leaving the node occupied by its tail empty in $C_{i+1}$, or

(d) $p$ is part of a handover with a neighboring particle $p'$.

While these properties allow at most one contraction or expansion per particle in moving from $C_i$ to $C_{i+1}$, multiple particles may move in this time.

Consider an arbitrary fair asynchronous activation sequence $A$ for a particle system and let $C_i^{(A)}$, for $0 \leq i \leq t$, be the particle system configuration at the end of asynchronous round $i$ in $A$ if each particle moves according to Algorithm 1. A *forest schedule* $\mathcal{S} = (A, (C_0, \dots, C_t))$ is a parallel schedule $(C_0, \dots, C_t)$ with the property that $A(C_0)$ is a forest of trees, and each particle $p$ follows the unique path $P_p$ which it would have followed according to $A$, starting from its node in $C_0$. This implies that $A(C_i)$ remains a forest of trees or a ring of trees for every $1 \leq i \leq t$. A forest schedule is said to be *greedy* if all particles perform movements according to Definition 3.24 of a parallel schedule in the direction of their unique paths whenever possible.

We begin our analysis with a result that is critical to both describing configurations of particles in greedy forest schedules and quantifying the amount of progress that greedy forest schedules make over time. Specifically, we show that if a forest's configuration is *well-behaved* at the start, then it remains so throughout its greedy forest schedule, guaranteeing that progress is made once every two configurations.

**Lemma 3.25.** *Given any fair asynchronous activation sequence $A$, consider any greedy forest schedule $(A, (C_0, \dots, C_t))$. 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 $1 \leq i \leq t$.*

*Proof.* Suppose to the contrary that $C_i$ is the first configuration that contains an expanded parent $p$ which has expanded children only. We consider all possible expanded and contracted states of $p$ and its children in $C_{i-1}$ and show

that none of them can result in $p$ and its children all being expanded in $C_i$. First suppose $p$ is expanded in $C_{i-1}$; then by supposition, $p$ has a contracted child $q$. By Definition 3.24, $q$ cannot perform any movements with its children (if they exist), so $p$ performs a handover contraction with $q$, yielding $p$ being contracted in $C_i$, which is a contradiction. So suppose $p$ is contracted in $C_{i-1}$. We know $p$ performs either a handover with its parent or a sole expansion in direction $p.dir$ since it is expanded in $C_i$ by supposition. Thus, any child of $p$ in $C_{i-1}$ — say $q$ — does not execute a movement with $p$ in moving from $C_{i-1}$ to $C_i$. Instead, if $q$ is contracted in $C_{i-1}$ then it remains contracted in $C_i$ since it is only permitted to perform a handover with its unique parent $p$; otherwise, if $q$ is expanded, it performs either a sole contraction if it has no children or a handover with one of its contracted children, which it must have by supposition. In either case, $p$ has a contracted child in $C_i$, which is also a contradiction.

As a final observation, two trees of the forest may "merge" when the super-root $s$ of one tree performs a sole expansion into an unoccupied node adjacent to a particle $q$ of another tree. However, $s$ is a root and thus defines $q$ as its parent only after performing a handover expansion with it; thus, the lemma holds in this case as well. □

For any particle $p$ in a configuration $C$ of a forest schedule, we define its *head distance* $d_h(p, C)$ (resp., *tail distance* $d_t(p, C)$) to be the number of edges along $P_p$ from the head (resp., tail) of $p$ to the end of $P_p$. Depending on whether $p$ is contracted or expanded, we have $d_h(p, C) \in \{d_t(p, C), d_t(p, C) - 1\}$. For any two configurations $C$ and $C'$ and any particle $p$, we say that $C$ *dominates* $C'$ *w.r.t.* $p$, denoted by $C(p) \succeq C'(p)$, if and only if $d_h(p, C) \leq d_h(p, C')$ and $d_t(p, C) \leq d_t(p, C')$. We say that $C$ *dominates* $C'$, denoted $C \succeq C'$, if and only if $C$ dominates $C'$ with respect to every particle. We now can show the following lemma.

**Lemma 3.26.** *Given any fair asynchronous activation sequence $A$ which begins at an initial configuration $C_0^{(A)}$ in which every expanded parent has at least one contracted child, there is a greedy forest schedule $\mathcal{S} = (A, (C_0, \ldots, C_t))$ with $C_0 = C_0^{(A)}$ such that $C_i^{(A)} \succeq C_i$ for all $0 \leq i \leq t$.*

*Proof.* We first introduce some supporting notation. Let $M(p) = p^{(1)}, p^{(2)}, \ldots$ be the sequence of movements $p$ executes according to $A$. Let $M_i(p)$ denote

the remaining subsequence of movements in $M(p)$ if the forest schedule reaches $C_i$, and let $m_i(p)$ denote the first movement in $M_i(p)$. Moreover, a movement $m_p$ of a particle $p$ is said to be *compatible* with a movement $m_q$ of particle $q$ if the execution of $m_p$ is still possible after $m_q$ is executed and vice versa.

**Lemma 3.27.** *A greedy forest schedule $\mathcal{S} = (A, (C_0, \dots, C_t))$ can be constructed from configuration $C_0 = C_0^{(A)}$ such that, for every $0 \leq i \leq t$, configuration $C_i$ is obtained from $C_{i-1}$ by executing only the movements of a greedily selected, mutually compatible subset of $\{m_{i-1}(p) : p \in P\}$.*

*Proof.* Argue by induction on $i$, the current configuration number. $C_0$ is trivially obtained, as it is the initial configuration. Assume by induction that the claim holds up to $C_{i-1}$. For $k \leq n$, let $M_{i-1} = \{m_{i-1}(p_1), \dots, m_{i-1}(p_k)\}$ be the greedily selected, mutually compatible subset of movements that $\mathcal{S}$ performs in moving from $C_{i-1}$ to $C_i$. Suppose to the contrary that a movement $m'(p) \notin M_{i-1}$ is executed by a particle $p \in P$. It can be easily seen that $m'(p)$ cannot be $m_{i-1}(p)$, since $m_{i-1}(p)$ was excluded when $M_{i-1}$ was greedily selected. Thus, it must be incompatible with one or more of the selected movements and cannot be executed at this time. So $m'(p) \neq m_{i-1}(p)$, and we consider the following cases:

(a) $m_{i-1}(p)$ is a sole contraction. Then $p$ is expanded and has no children in $C_{i-1}$, so we must have $m'(p) = m_{i-1}(p)$, since there are no other movements $p$ could execute, which is a contradiction.

(b) $m_{i-1}(p)$ is a sole expansion. Then $p$ is contracted and has no parent in $C_{i-1}$, so we must have $m'(p) = m_{i-1}(p)$, since there are no other movements $p$ could execute, which is a contradiction.

(c) $m_{i-1}(p)$ is a handover contraction with one of its children $q$. Then at some time in $\mathcal{S}$ before reaching $C_{i-1}$, $q$ became a descendant of $p$; thus, $q$ must also be a descendant of $p$ in $C_{i-1}$. If $q$ is not a child of $p$ in $C_{i-1}$, there exists a particle $z \notin \{p, q\}$ such that $q$ is a descendant of $z$, which is in turn a descendant of $p$. So in order for $m_{i-1}(p)$ to be a handover contraction with $q$, $M(z)$ must include actions which allow $z$ to "bypass" its ancestor $p$. However, this is impossible according to our algorithm, since particles follow their predetermined path in the tree. So

$q$ has to be a child of $p$ in $C_{i-1}$ and is contracted at the time $m_{i-1}(p)$ is performed. If $q$ is also contracted in $C_{i-1}$, then once again we must have $m'(p) = m_{i-1}(p)$. Otherwise, $q$ is expanded in $C_{i-1}$ and must have become so before $C_{i-1}$ was reached. But this yields a contradiction: since $\mathcal{S}$ is greedy, $q$ would have contracted prior to this point by executing either a sole contraction if it has no children, or a handover contraction with a contracted child whose existence is guaranteed by Lemma 3.25, since every expanded parent in $C_0$ has a contracted child.

(d) $m_{i-1}(p)$ is a handover expansion with $q$, its unique parent. Then we must have that $m_{i-1}(q)$ is a handover contraction with $p$, and an argument analogous to that of Case 3 follows. □

We conclude our proof by showing that each configuration of the greedy forest schedule $\mathcal{S}$ constructed according to Lemma 3.27 is dominated by its asynchronous counterpart. We argue by induction on $i$, the configuration number. Since $C_0 = C_0^{(A)}$, we have that $C_0^{(A)} \succeq C_0$. Assume by induction that for all rounds $0 \leq r \leq i-1$, we have $C_r^{(A)} \succeq C_r$. Consider any particle $p$. Since $\mathcal{S}$ is constructed using the exact set of movements $p$ executes according to $A$ and each time $p$ moves, it decreases either its head distance or tail distance by 1, it suffices to show that $p$ has performed at most as many movements in $\mathcal{S}$ up to $C_i$ as it has according to $A$ up to $C_i^{(A)}$.

If $p$ does not perform a movement between $C_{i-1}$ and $C_i$, we trivially have $C_i^{(A)}(p) \succeq C_i(p)$. Otherwise, $p$ performs movement $m_{i-1}(p)$ to obtain $C_i$ from $C_{i-1}$. If $p$ has already performed $m_{i-1}(p)$ according to $A$ before reaching $C_{i-1}^{(A)}$, then clearly $C_i^{(A)}(p) \succeq C_i(p)$. Otherwise, $m_{i-1}(p)$ must be the next movement $p$ is to perform according to $A$, since $p$ has performed the same sequence of movements in the asynchronous execution as it has in $\mathcal{S}$ up to the respective rounds $i-1$, and thus has equal head and tail distances in $C_{i-1}$ and $C_{i-1}^{(A)}$. It remains to show that $p$ can indeed perform $m_{i-1}(p)$ between $C_{i-1}^{(A)}$ and $C_i^{(A)}$. If $m_{i-1}(p)$ is a sole expansion, then $p$ is the super-root of its tree (in both $C_{i-1}$ and $C_{i-1}^{(A)}$) and must also be able to expand in $C_{i-1}^{(A)}$. Similarly, if $m_{i-1}(p)$ is a sole contraction, then $p$ has no children (in both $C_{i-1}$ and $C_{i-1}^{(A)}$) and must be able to contract in $C_{i-1}^{(A)}$. If $m_{i-1}(p)$ is a handover expansion with its parent $q$, then $q$ must be expanded in $C_{i-1}$. Parent $q$ must also be expanded in $C_{i-1}^{(A)}$; otherwise $d_h(q, C_{i-1}^{(A)}) > d_h(q, C_{i-1})$, contradicting the induction hypothesis.

An analogous argument holds if $m_{i-1}(p)$ is a handover contraction with one of its contracted children. Therefore, in any case we have $C_i^{(A)}(p) \succeq C_i(p)$ and, since the choice of $p$ was arbitrary, $C_i^{(A)} \succeq C_i$. □

We can show a similar dominance result when considering complaint flags.

**Definition 3.28.** A movement schedule $(C_0, \dots, C_t)$ is called a *complaint-based parallel schedule* if each $C_i$ is a valid configuration of a particle system in which every particle holds at most *one* complaint flag and for every $i \geq 0$, $C_{i+1}$ is reached from $C_i$ such that for every particle $p$ one of the following properties holds:

(a) $p$ does not hold a complaint flag and property (a), (c), or (d) of Definition 3.24 holds,

(b) $p$ holds a complaint flag $f$ and expands into an adjacent node that was empty in $C_i$, consuming $f$,

(c) $p$ forwards a complaint flag $f$ to a neighboring particle $p'$ which either does not hold a complaint flag in $C_i$ or is also forwarding its complaint flag.

A *complaint-based forest schedule* $\mathcal{S} = (A, (C_0, \dots, C_t))$ has the same properties as a forest schedule, with the exception that $(C_0, \dots, C_t)$ is a complaint-based parallel schedule as opposed to a parallel schedule. A complaint-based forest schedule is said to be *greedy* if all particles perform movements according to the Definition 3.28 in the direction of their unique paths whenever possible.

We can now extend the dominance argument to hold with respect to *complaint distance* in addition to head and tail distances. For any particle $p$ holding a complaint flag $f$ in configuration $C$, we define its complaint distance $d_c(f, C)$ to be the number of edges along $P_p$ from the node $p$ occupies to the end of $P_p$. For any two configurations $C$ and $C'$ and any complaint flag $f$, we say that $C$ *dominates* $C'$ *w.r.t.* $f$, denoted $C(f) \succeq C'(f)$, if and only if $d_c(f, C) \leq d_c(f, C')$. Extending the previous notion of dominance, we say that $C$ *dominates* $C'$, denoted $C \succeq C'$, if and only if $C$ dominates $C'$ with respect to every particle and with respect to every complaint flag.

It is also possible to construct a greedy complaint-based forest schedule whose configurations are dominated by their asynchronous counterparts, as we

did for greedy forest schedules in Lemma 3.26. Many of the details are the same, so to avoid redundancy we highlight the key differences here. The most obvious difference is the inclusion of complaint flags. Definition 3.28 restricts particles to hold at most one complaint flag at a time, where Algorithm 3 allows a capacity of two. This allows the asynchronous execution of our algorithm to not fall behind the parallel schedule in terms of forwarding complaint flags. Basically, Definition 3.28 allows a particle $p$ holding a complaint flag $f$ in a configuration $C_i$ to forward $f$ to its parent $q$ in $C_{i+1}$ even if $q$ also holds a own complaint flag in $C_i$, as long as $q$ is also forwarding its flag in $C_{i+1}$. The asynchronous execution does not have this luxury of synchronized actions. Thus, the mechanism of buffering up to two complaint flags at a time allows it to mimic the pipelining of forwarding complaint flags that is possible within two configurations of a complaint-based parallel schedule.

Another slight difference is that a contracted particle cannot expand into an empty adjacent node unless it holds a complaint flag to consume. However, this restriction reflects Algorithm 4, so once again the greedy complaint-based forest schedule can be constructed directly from the movements taken in the asynchronous execution. Moreover, since this restriction can only cause a contracted particle to remain contracted, the conditions of Lemma 3.25 are still upheld. Thus, we obtain the following lemma:

**Lemma 3.29.** *Given any fair asynchronous activation sequence $A$ which begins at an initial configuration $C_0^{(A)}$ in which every expanded parent has at least one contracted child, there is a greedy complaint-based forest schedule $\mathcal{S} = (A, (C_0, \dots, C_t))$ with $C_0 = C_0^{(A)}$ such that $C_i^{(A)} \succeq C_i$ for all $0 \leq i \leq t$.*

By Lemmas 3.26 and 3.29, we can easily deduce an upper bound for the number of rounds required by the asynchronous execution, once we have an upper bound for the time it takes a greedy forest schedule to reach a final configuration. Hence, the proofs in the remaining two subsections serve to upper bound the number of *parallel rounds* (i.e., number of configurations) any greedy forest schedule would require to solve the coating problem for a given valid instance $(P, O)$. Let $\mathcal{S}^* = (A, (C_0, \dots, C_k))$ be such a greedy forest schedule, where $C_0$ is the initial configuration of the particle system $P$ (of all contracted particles) and $C_k$ is the final coating configuration.

In Sections 3.4.2.2 and 3.4.2.3, we upper bound the number of parallel rounds

required by $\mathcal{S}^*$ to coat the surface layer and higher layers, respectively. More specifically, we bound the worst-case time it takes to complete a layer $i$ once layers $1, \dots, i-1$ have been completed. For convenience, we do not differentiate between complaint-based and regular forest schedules in the following sections, since the same dominance result holds whether or not complaint flags are considered. To prove these bounds, we need one last definition: a *forest–path schedule* (or short *fp* schedule) $\mathcal{S} = (A, (C_0, \dots, C_t), L)$ is a forest schedule $(A, (C_0, \dots, C_t))$ with the property that all the trees of $A(C_0)$ are rooted at a path $L = (v_1, v_2 \dots, v_\ell) \subseteq G_{\text{eqt}}$, and each particle $p$ must traverse $L$ in the same direction.

#### 3.4.2.2. Surface Layer: Complaint-based Coating and Leader Election

Our algorithm must first organize the particles using the spanning forest primitive, whose runtime is easily bounded. In the following we use the term *asynchronous round* to emphasize that we consider the runtime of our Universal Coating Algorithm, and not the runtime of $\mathcal{S}^*$.

**Lemma 3.30.** *Following the spanning forest primitive, the particles form a spanning forest within $\mathcal{O}(n)$ asynchronous rounds.*

*Proof.* Initially all particles are idle. In each round any idle particle adjacent to (i) the object, (ii) an active (follower or root) particle, or (iii) a retired particle becomes active. It then sets its parent flag if it is a follower, or becomes the root of a tree if it is adjacent to the object or a retired particle. In each round at least one particle becomes active, so it takes $\mathcal{O}(n)$ rounds in the worst case until all particles join the spanning forest. □

For ease of presentation, we assume that the particle system is of sufficient size to fill the surface layer: i.e., $B_1 \leq n$. The proofs can easily be extended to handle the case when $B_1 > n$. We also assume that the root of a tree also generates a complaint flag upon its activation (this assumption does not hurt our argument since it only increases the number of the flags generated in the system). Let $\mathcal{S}_1 = (A, (C_0, \dots, C_{t_1}), L_1)$ be the greedy *fp* schedule where $(A, (C_0, \dots, C_{t_1}))$ is a truncated version of $\mathcal{S}^*$, where $C_{t_1}$ is the configuration of $\mathcal{S}^*$ in which the surface layer becomes complete, and $L_1$ is the path of nodes in

the surface layer. The following lemma shows that the algorithm makes steady progress towards completing the surface layer.

**Lemma 3.31.** *Consider a configuration $i$ of the greedy $fp$ schedule $\mathcal{S}_1$, where $0 \leq i \leq t_1 - 2$. Then within the next two configurations of $\mathcal{S}_1$, (i) at least one complaint flag is consumed, (ii) at least one more complaint flag reaches a particle in the surface layer, (iii) all remaining complaint flags move one node closer to a super-root along $L_1$, or (iv) the surface layer is completely filled (possibly with some expanded particles).*

*Proof.* If the surface layer is filled, (*iv*) is satisfied; otherwise, there exists at least one super-root in $A(C_i)$. We consider several cases:

(a) There exists a super-root $s$ in $A(C_i)$ which holds a complaint flag. If $s$ is contracted, then it can expand and consume its flag in the next configuration. Otherwise, consider the case when $s$ is expanded. If it has no children, then within the next two configurations it can contract and expand again, consuming its complaint flag. Otherwise, by Lemma 3.25, $s$ must have a contracted child with which it can perform a handover to become contracted in $C_{i+1}$ and then expand and consume its complaint flag by $C_{i+2}$. In any case, (i) is satisfied.

(b) No super-root in $A(C_i)$ holds a complaint flag and not all complaint flags have been moved from follower particles to particles in the surface layer. Let $p_1, p_2, \ldots, p_z$ be a sequence of particles in the surface layer such that each particle holds a complaint flag, no follower child of any particle except $p_z$ holds a complaint flag, and no particles between the next super-root $s$ and $p_1$ hold complaint flags. Then, as each $p_i$ forwards its flag to $p_{i-1}$ according to Definition 3.28, the follower child of $p_z$ holding a flag is able to forward its flag to $p_z$, satisfying (ii).

(c) No super-root in $A(C_i)$ holds a complaint flag and all remaining complaint flags are held by particles in the surface layer. By Definition 3.28, since no preference needs to be given to flags entering the surface layer, all remaining flags move one node closer to a super-root in each configuration, satisfying (iii). □

We use Lemma 3.31 to show first that the surface layer is filled with particles (some possibly still expanded) in $\mathcal{O}(n)$ configurations of $\mathcal{S}^*$. From that point on, in another $\mathcal{O}(n)$ configurations, one can guarantee that expanded particles on the surface layer contract in a handover with a follower particle; hence all particles in the surface layer are contracted, as we see in the following lemma.

**Lemma 3.32.** *After $\mathcal{O}(n)$ configurations of $\mathcal{S}^*$, the surface layer must be filled with contracted particles.*

*Proof.* As an intermediate step we first prove the following claim.

*Claim.* After $8B_1 + 2$ configurations of $\mathcal{S}^*$, the surface layer must be filled with particles.

Suppose to the contrary that after $8B_1 + 2$ configurations, the surface layer is not completely filled with particles. Then none of these configurations satisfied case (iv) of Lemma 3.31. Thus, either case (i), case (ii), or case (iii) is satisfied every two configurations. Case (i) can be satisfied at most $B_1$ times (accounting for at most $2B_1$ configurations), since a super-root expands into an unoccupied node of the surface layer each time a complaint flag is consumed. Case (iii) can also be satisfied at most $B_1$ times (accounting again for at most $2B_1$ configurations), since once all remaining complaint flags are in the surface layer, every flag must reach a super-root in $B_1$ moves. Thus, the remaining $4B_2 + 2$ configurations have to satisfy case (ii) for $2B_1 + 1$ times, implying that $2B_1 + 1$ flags reached particles in the surface layer from follower children. But each particle can hold at most one complaint flag in $\mathcal{S}^*$, so at least $B_1 + 1$ flags must have been consumed, in order to provide enough memory space in particles already on $B_1$. Therefore the super-roots have collectively expanded into at least $B_1 + 1$ unoccupied nodes, which is a contradiction.

By the claim, it takes at most $8B_1 + 2$ configurations until the surface layer is completely filled with particles (some possibly expanded). In at most another $B_1$ configurations, every expanded particle in the surface layer contracts in a handover with a follower particle (since $B_1 \leq n$), and hence all particles in the surface layer are contracted after $\mathcal{O}(B_1) = \mathcal{O}(n)$ configurations. □

Once the surface layer is completely filled, the leader election primitive can proceed. From [Day+17] we can deduce the following runtime bound, which can be directly transferred since we assume that the surface layer is filled.

**Lemma 3.33.** *Within $\mathcal{O}(n)$ asynchronous rounds, a node of the surface layer has been elected as the leader node, w.h.p.*

Once a leader node has been elected and either no more followers exist (if $n \leq B_1$) or all nodes are completely filled by contracted particles (which can be checked in an additional $\mathcal{O}(B_1)$ asynchronous rounds), the particle currently occupying the leader node becomes the leader particle. Once a leader has emerged, the particles on the surface layer retire, which takes $\mathcal{O}(B_1)$ further asynchronous rounds.

#### 3.4.2.3. Higher Layers

We again use the dominance results we proved in Section 3.4.2.1 to focus on parallel schedules when proving an upper bound on the worst-case number of asynchronous rounds — denoted by $Layer(i)$ — for building layer $i$ once layer $i-1$ is complete, for $2 \leq i \leq N$. In doing so, we show the following lemma, which provides a more general result that can be used for this purpose.

**Lemma 3.34.** *Consider any greedy fp schedule $\mathcal{S} = (A, (C_0, \dots, C_t), L)$ with $L = (v_1, v_2, \dots, v_\ell)$ and any $k$ such that $1 \leq k \leq \ell$. If every expanded parent in $C_0$ has at least one contracted child, then in at most $2(\ell + k)$ configurations, nodes $v_{\ell-k+1} \dots v_\ell$ are occupied by contracted particles.*

*Proof.* Let $s$ be the super-root closest to $v_\ell$, and suppose $s$ initially occupies node $v_i$ in $C_0$. Additionally, suppose there are at least $k$ active particles in $C_0$ (otherwise, we do not have sufficient particles to occupy $k$ nodes of $L$). Argue by induction on $k$, the number of nodes in $L$ starting with $v_\ell$ which must be occupied by contracted particles. First suppose that $k = 1$. By Lemma 3.25, every expanded parent has at least one contracted child in any configuration $C_j$, so $s$ is always able to either expand forward into an unoccupied node of $L$ if it is contracted or contract as part of a handover with one of its children if it is expanded. Thus, in at most $2(\ell + k) = 2\ell + 2$ configurations, $s$ has moved forward $\ell$ nodes, is contracted, and occupies its final node $v_{\ell-k+1} = v_\ell$.

Now suppose that $k > 1$ and that each node $v_{\ell-x+1}$, for $1 \leq x \leq k-1$, becomes occupied by a contracted particle in at most $2(\ell+k-1) = 2(\ell+k)-2$ configurations. It suffices to show that $v_{\ell-k+1}$ also becomes occupied by a contracted particle in at most two additional configurations. Let $p$ be

the particle currently occupying $v_{\ell-k+1}$ (such a particle must exist since we supposed we had sufficient particles to occupy $k$ nodes and $\mathcal{S}$ ensures the particles follow this unique path). If $p$ is contracted in $C_{2(\ell+k)-2}$, then it remains contracted and occupying $v_{\ell-k+1}$, so we are done. Otherwise, if $p$ is expanded, it has a contracted child $q$ by Lemma 3.25. Particles $p$ and $q$ thus perform a handover in which $p$ contracts to occupy only $v_{\ell-k+1}$ at $C_{2(\ell+k)-1}$, proving the claim. □

For convenience, we introduce some additional notation. Let $n_i$ denote the number of particles of the system that cannot occupy to layers 1 through $i-1$ (i.e., $n_i = n - \sum_{j=1}^{i-1} B_j$) and let $t_i$ (resp., $C_{t_i}$) be the configuration in which layer $i$ becomes complete.

When coating some layer $i$, each root particle either moves either (i) through the nodes in layer $i$ in the set direction *dir* (CW or CCW) for layer $i$, or (ii) through the nodes in layer $i+1$ in the opposite direction over the already retired particles in layer $i$ until it finds an empty node in layer $i$. We bound the worst-case scenario for these two movements independently in order to get an upper bound on $Layer(i)$. Let $L_i = (v_1, \dots, v_{B_i})$ be the path of nodes in layer $i$ listed in the order that they appear from the marker node $v_1$ following direction *dir*, and let $\mathcal{S}_i = (A, (C_{t_{i-1}+1}, \dots, C_{t_i}), L_i)$ be a greedy $fp$ schedule where $(A, (C_{t_{i-1}+1}, \dots, C_{t_i}))$ is the section of $\mathcal{S}^*$ in which layer $i$ is coated. By Lemma 3.34, it would take $\mathcal{O}(B_i)$ configurations for all case (i) movements to complete; an analogous argument shows that all case (ii) movements complete in $\mathcal{O}(B_{i+1}) = \mathcal{O}(B_i)$ configurations. This implies the following lemma:

**Lemma 3.35.** *Starting from configuration $C_{t_{i-1}+1}$, the worst-case additional number of configurations for layer $i$ to become complete is $\mathcal{O}(B_i)$.*

Putting it all together, for layers 2 through $N$:

**Lemma 3.36.** *The worst-case number of configurations for $\mathcal{S}^*$ to coat layers 2 through $N$ is $\mathcal{O}(n)$.*

*Proof.* Starting from configuration $C_{t_1+1}$, it follows from Lemma 3.35 that the worst-case number of configurations for $\mathcal{S}^*$ to reach a legal coating of the object is upper bounded by

$$\sum_{i=2}^{N} Layer(i) \leq c \sum_{i=2}^{N} B_i = \Theta(n),$$

where $c > 0$ is constant. □

We can now combine all our results to prove the runtime of our algorithm.

**Theorem 3.37.** *The total number of asynchronous rounds required for the Universal Coating algorithm to reach a legal coating configuration, starting from an arbitrary valid instance $(P, O)$, is $\mathcal{O}(n)$ w.h.p.*

*Proof.* According to Lemma 3.30 it takes $\mathcal{O}(n)$ asynchronous rounds until the particles form a spanning forest. It then takes $\mathcal{O}(n)$ configurations of $\mathcal{S}^*$ to fill the surface layer with contracted particles according to Lemma 3.32 and by Lemmas 3.26 and 3.29 this is an upper bound for the runtime of our Universal Coating Algorithm. After additional $\mathcal{O}(n)$ asynchronous rounds w.h.p. (see Lemma 3.33) a leader has been elected and the surface is complete. We can then apply Lemma 3.36 to obtain that it takes an additional $\mathcal{O}(n)$ configurations to coat all higher layers, which is again an upper bound for our algorithm by Lemmas 3.26 and 3.29. □

# CHAPTER 4

# Basic Shape Formation

> ”*It can't form complex machines. [...] But it can form solid metal shapes.*”
>
> Scene from Terminator 2

YIELDING any imaginable shape, most commonly referred to as shape formation, is one of the most natural problems for self-organizing programmable matter. The matter should change its shape based on either user input or autonomous sensing and the constructed shape should also scale with the size of the matter: i.e., the number of particles in the system. In this chapter we focus on the problem of constructing basic geometric shapes. In doing so, we introduce a general algorithmic framework for basic shape formation problems. This framework constitutes of two algorithmic primitives: the *spanning forest primitive* and the *snake formation primitive*. In order to show the variability of this approach we present three concrete applications for specific shape formation problems, namely the formation of a *line*, a *hexagon* and a *triangle*.

**Chapter Outline** In Section 4.1 we formally define the three shape formation problems that we investigate in this chapter. Section 4.2 introduces the two

algorithmic primitives that constitute our framework. In the last three sections, we apply our framework to solve the three different problems, starting with the simplest shape – the line – in Section 4.3. The hexagon is considered in Section 4.4 and the triangle in Section 4.5.

**Chapter Basis** The problem statement of hexagon and triangle shape formation, as well as the corresponding algorithms and the correctness analysis are all based on the following publication:

> **2015** (with Z. Derakhshandeh, R. Gmyr, A. W. Richa and C. Scheideler). "An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems". In: *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM' 15, Boston, MA, USA, September 21-22, 2015*, cf. [Der+15a].

The line shape formation problem, together with its algorithm and correctness analysis, is based on:

> **2015** (with Z. Derakhshandeh, R. Gmyr, R. A. Bazzi, A. W. Richa and C. Scheideler). "Leader Election and Shape Formation with Self-organizing Programmable Matter". In: *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*, cf. [Der+15b].

All runtime proofs in this chapter have not been published before.

## 4.1. Problem Statements

In the *shape formation problem*, a particle system has to reconfigure into a given shape. We formally define a shape formation problem as a tuple $M = (I, G)$ where $I$ and $G$ are sets of connected configurations. We say $I$ is the set of initial configurations and $G$ is the set of goal configurations. For any initial configuration we assume that all particles are contracted and in an *idle* state.

Throughout this section we consider three different shape formation problems: *Line Shape Formation* ($\mathcal{LSF}$), *Hexagon Shape Formation* ($\mathcal{HSF}$) and *Triangle*

*Shape Formation* ($\mathcal{TSF}$). In the first problem, the desired goal shape is a straight line, in the second one it is a hexagon, and in the third it is a triangle. Accordingly, for the $\mathcal{LSF}$ problem, $G$ consists of all configurations such that the nodes occupied by the particles induce a straight line in $G_{\text{eqt}}$. Similarly, for the $\mathcal{HSF}$ and the $\mathcal{TSF}$ problem, $G$ consists of all configurations that constitute a hexagon in $G_{\text{eqt}}$ or a triangle in $G_{\text{eqt}}$, respectively. Note that depending on the number of particles the constructed shape may not necessarily be a perfect hexagon or triangle: i.e., the outer layer of the constructed shape may not be fully complete. We say an algorithm $\mathcal{A}$ *solves* a shape formation problem $M$ if for any execution of $\mathcal{A}$ started on an arbitrary configuration of $I$, $\mathcal{A}$ *terminates* (i.e., the execution eventually reaches a configuration in which each particle does not move anymore) in a configuration of $G$.

Throughout our investigation of shape formation we assume that any initial configuration contains one special particle which we call the *seed* particle. The seed provides the starting point for constructing the respective shape. Note that a seed particle could be established using a leader election algorithm (e.g., [Der+15b; Day+17]) that is executed before shape formation starts.

## 4.2. Shape Formation Algorithm

The shape formation algorithm we propose builds upon an algorithmic primitive that we introduced in Chapter 3. Before explaining the algorithm in detail, we are going to reiterate some notions that we use in this section.

A particle can be in four different *states*: *idle*, *follower*, *root*, and *retired*. Followers and roots are considered *active*. Initially, all particles are idle, except for the seed particle, which is always in the *retired* state. In addition to its state, each particle $p$ again maintains a constant number of *flags* (i.e., constant size pieces of information visible to neighboring particles), in its local memory. Again, we assume that every time a particle contracts, it contracts out of its tail. Thus, the node occupied by the head of a particle is still occupied by that particle after a contraction.

Generally speaking, the shape formation algorithms we propose progresses as follows. Particles organize themselves into multiple disjoint trees, in which the roots are non-retired particles adjacent to the partially constructed shape that consists of retired particles. Root particles lead the way of their tree by

moving in a predefined direction around the current structure.

The follower particles follow behind the leading root particles, hence the system flattens out towards the direction of movement. Roots follow the *snake formation primitive* to find the next node where the shape can be extended. Once such a node is reached, they stop moving and become retired. This process continues until all particles are retired. Note that the spanning forest component of this general approach is the same for all presented shape formation problems. The major difference is the rule which determines the next node to be occupied in the shape structure.

### 4.2.1. Spanning Forest Primitive

The spanning forest primitive, which we already used in Chapter 3, is a building block we use for all of our shape formation problems. The basic idea is the same as before: i.e., particles are organized into a spanning forest to get a straightforward mechanism for particles to move while preserving connectivity. Since space is not an issue, we again present the full pseudocode in Algorithm 6. Each particle continuously runs the spanning forest primitive until it becomes retired. The major differences to the spanning forest primitive of Chapter 3 is that root particles do not move around a given object and that there is no need to have distinct moving directions CW and CCW. Instead, root particles move around already retired particles in the direction given by RootDirection($p$) (see Algorithm 8), namely in clockwise order, until they find a valid node to retire on. Moreover, since there is no explicit notion of layers around an object, the spanning forest itself and the handover (see Algorithm 7) become easier to describe.

### 4.2.2. Snake Formation Primitive

Whereas the spanning forest primitive makes sure that particles organize themselves, the snake formation primitive is actually responsible for constructing the desired shape. The snake formation specifies how particles retire, which, by definition, is heavily dependent on the concrete problem that the shape formation algorithm should solve. Therefore, we do not give an in-depth description here, but instead describe the high level idea of the approach. The concrete versions of the snake formation primitive (including pseudocode) that

**Algorithm 6** Spanning Forest Primitive for Shape Formation

A particle $p$ acts depending on its state as described below:

**idle**: If $p$ is adjacent to a *retired* particle, $p$ becomes a *root* particle. If a neighbor $p'$ is a root or a follower, $p$ sets the flag $p.parent$ to the label of the port to $p'$, and becomes a *follower*. If none of the above applies, $p$ remains idle.

**follower**: If $p$ is contracted and adjacent to a retired particle, then $p$ becomes a *root* particle. If $p$ is contracted and has an expanded parent, then $p$ initiates HANDOVER($p$) (Algorithm 7). Otherwise, if $p$ is expanded, it considers the following two cases: (i) if $p$ has a contracted child particle $q$, then $p$ initiates HANDOVER($p$); (ii) if $p$ has no children and no idle neighbor, then $p$ contracts.

**root**: If $p$ is contracted, it first executes the corresponding snake formation algorithm (Algorithm 9, 10 or 11, for $\mathcal{LSF}$, $\mathcal{HSF}$ or $\mathcal{TSF}$ resp.) and can become *retired*, accordingly. If $p$ does not become retired and is contracted, it calls ROOTDIRECTION($p$) (Algorithm 8) and tries to expand in that direction or, in case the node is occupied by an expanded particle, initiates HANDOVER($p$). If $p$ is expanded, it considers the following two cases: (i) if $p$ has a contracted child, then $p$ initiates HANDOVER($p$); (ii) if $p$ has no children and no idle neighbor, then $p$ contracts.

**retired**: $p$ performs no further action.

are used for the $\mathcal{LSF}$, $\mathcal{HSF}$ and $\mathcal{TSF}$, respectively, are explained in detail in the corresponding sections.

The snake formation gets its name from the way the particles retire. The seed is already retired in an initial state. It specifies a direction by a flag on a port label, in which the structure of retired particles should grow. Throughout this section, we use the flag $p.retireDir$ for a retired particle $p$ to denote this specific port and we call the node that this port points to the next *valid* node to extend the shape. As soon as a contracted root occupies a valid node, it retires and specifies the next valid node to grow the structure. Thereby, we get an approach that scales naturally with the number of particles in the system. From a global point of view, it seems like the structure is grown as a snake of particles, since particles retire one-by-one and the *retireDir* pointers form a spanning line through the structure of retired particles. Different rules for snake formation realize different shapes.

**Algorithm 7** HANDOVER($p$)

```
1: if p is expanded then
2:     if p has at least one contracted child q such that q.parent points to the
   tail of p then
3:         p performs a handover with q
4: else
5:     if p is a follower and p.parent is expanded then
6:         p performs a handover with p.parent
7:     if p is a root and RootDirection(p) points to an expanded particle q
   then
8:         p performs a handover with q
```

**Algorithm 8** ROOTDIRECTION($p$)

```
1: Let i be the label of a port connected to a retired particle.
2: while i points to a retired particle do
3:     i ← label of next port in counter-clockwise direction
4: return i
```

## 4.3. Line Shape Formation

As the first shape formation problem we consider the $\mathcal{LSF}$ problem. The seed is used as the starting point for the line of particles and specifies the direction in which this line will grow.

In order to solve the $\mathcal{LSF}$ problem, we use the retiring condition presented in Algorithm 9 for the snake formation primitive. Particles retire at one endpoint of the line. Initially, the seed particle $s$ sets the flag $s.retireDir$ to an arbitrary port (e.g., the one labeled 0). Following the spanning forest primitive of Algorithm 6 any particle adjacent to a retired particle becomes a root. Each root $p$ moves in a clockwise fashion around the structure of retired particles (see Algorithm 8) until it finds the next node to extend the shape (i.e., a node adjacent to a retired particle $q$ by a port flagged with $q.retireDir$) and becomes retired. Once $p$ becomes retired, it sets the flag $p.retireDir$ to the port opposing the port that is connected to $q$ (see Algorithm 9). Thereby, the line grows in one direction starting from the seed. Note that the seed could also grow the line in two (opposing) directions from the get-go. However, this neither effects the correctness nor the asymptotic runtime of our approach.

We say that the particle system *makes progress* if (i) an idle particle becomes

**Algorithm 9** Retirement Condition for $\mathcal{LSF}$

```
if p is a contracted root then
    if p is adjacent to a retired particle p' such that p'.retireDir points to p
then
        Let i by the port label of p that points to p'
        p sets p.retireDir on port (i + 3) mod 6
        p becomes retired
```

active, (ii) a movement (i.e., an expansion, handover, or contraction) is executed, or (iii) an active particle retires. To prove the correctness of our algorithm we need the following lemma as an intermediate step. The lemma guarantees progress throughout the execution of our algorithm.

**Lemma 4.1.** *If a root particle exists, the system eventually makes progress.*

*Proof.* If there exists an expanded root it eventually contracts according to Lemma 3.12 of Subsection 3.3.2. If there is a contracted root $p$ that has an empty node in RootDirection($p$) it expands as soon as it is activated. So assume that all roots are contracted and that for every root RootDirection($p$) points to an occupied node. By construction of our algorithm this means that the whole structure of retired particles is surrounded by a layer of roots. Consequently, one root occupies the node that is specified to be the continuation of the snake formation (i.e., the node that $q.retireDir$ points to, where $q$ is the particle that retired last or the seed). This root eventually retires. □

**Theorem 4.2.** *The spanning forest primitive together with the snake formation for the line solve the $\mathcal{LSF}$ problem.*

*Proof.* We need to show that the algorithm terminates and that when it does, the formed shape is a straight line.

First, we show that the system eventually makes progress as long as non-retired particles exist. According to Lemma 3.8 in Subsection 3.3.2, every particle eventually becomes active. So assume that all particles are either active or retired. If there exists at least one root, progress is guaranteed by Lemma 4.1. Consider the case in which no root exists. According to the spanning forest primitive, there exists at least one follower that is either adjacent to the seed or to a retired particle. The next time this particle activates, it becomes either

a root (i.e., Lemma 4.1 can be applied) or retired, which guarantees progress. Thus the algorithm terminates.

Initially, the structure of retired particles contains only the seed particle and the claim holds. By induction, we assume that $C$ is the first configuration in which the current formed structure of retired particles contains $k$ retired particles. By induction hypothesis, assume that those particles form a line using $k$ particles. According to Algorithm 9, the only way a root $p$ can become the $(k+1)^{\text{th}}$ retired particle in or after $C$ is if it occupies the node that $q.retireDir$ points to, where $q$ is the $k$-th retired particle in the line shape. By construction the flag $q.retireDir$ points to the node that extends the straight line. Consequently, $p$ retires on node such that the retired structure is a line of $(k+1)^{\text{th}}$ particles. □

We now want to bound the worst-case runtime of our algorithm. In doing so we make use of the notions and results established in Chapter 3. Especially the results of Subsection 3.4.2.1 are applicable for the $\mathcal{LSF}$ and also for our shape formation approach in general. We make use of Lemma 3.26: i.e., each configuration of a greedy forest schedule $\mathcal{S}$ constructed according to Lemma 3.27 is dominated by its asynchronous counterpart. Therefore, we can use the dominance results and focus on parallel schedules when proving an upper bound on the worst-case number of rounds. We will again use an arbitrary fair asynchronous activation sequence $A$ for the particle system. Let $L_{\mathcal{LSF}} = (v_1, v_2, \dots, v_{n-1}) \subseteq G_{\text{eqt}}$ denote the unique path of nodes given by the snake formation primitive for the $\mathcal{LSF}$, such that $v_1$ is adjacent to the seed.

**Lemma 4.3.** *Consider any greedy forest schedule $\mathcal{S} = (A, (C_0, \dots, C_t))$ the path $L_{\mathcal{LSF}}$ and any $k$ with $1 \leq k \leq n$. If every expanded parent in $C_0$ has at least one contracted child, then in at most $4k + 14$ configurations, nodes $v_1 \cdots v_k$ are occupied by retired particles.*

*Proof.* Let $s$ be the root that retires in $v_1 \in L_{\mathcal{LSF}}$ according to $C_0$, which is well defined. Without loss of generality suppose there are at least $k$ active particles in $C_0$. We will prove by induction that after $2k + \Phi(\mathcal{S})$ configurations the first $k$ nodes of $L_{\mathcal{LSF}}$ are occupied by a retired particle, where $\Phi(\mathcal{S})$ is shorthand for a term to collect any additional configurations that are required. We later show how $\Phi(\mathcal{S})$ can be upper bounded.

First suppose that $k = 1$. By Lemma 3.25, every expanded parent has at least one contracted child in any configuration $C_j$, so $s$ is always able to either expand forward into an unoccupied node if it is contracted or contract as part of a handover with one of its children if it is expanded. If $s$ occupies $v_1$ or is adjacent to $v_1$, it retires on $v_1$ after at most 2 configurations. If this is not the case we put the number of additional configurations needed in $\Phi(\mathcal{S})$. Thus, in at most $2(k) + \Phi(\mathcal{S})$ configurations, $s$ is contracted, occupies $v_1$ and retires.

Now suppose that $k > 1$ and that each node $v_i$, for $1 \leq i \leq k-1$, is occupied by a retired particle in at most $2(k-1) + \Phi(\mathcal{S})$ configurations. Let $p$ be the particle that retires on $v_k$. Such a particle must exist since we supposed we had sufficient active particles to occupy $k$ nodes and $\mathcal{S}$ ensures that the particles follow the unique path to valid nodes. If $p$ either occupies $v_k$ or a node adjacent to $v_k$ and is contracted in $C_{2(k-1)+\Phi(\mathcal{S})}$ the induction follows immediately similarly to the proof of Lemma 3.34: i.e., $v_k$ also becomes occupied by a retired particle in at most two additional configurations. Otherwise, we again add the additional configurations needed to $\Phi(\mathcal{S})$.

Consequently, our induction is successful and we simply need to upper bound $\Phi(\mathcal{S})$. For a configuration $C_i$ we define the distance between two nodes $u, v$ adjacent to the structure of retired particles (short $d_i(u, v)$), or the particles occupying those nodes, to be the number of edges on the path around the structure of retired particles in clockwise order (i.e., their direction of travel). Note that in the induction base we need to use $\Phi(\mathcal{S})$ only if $s$ cannot occupy $v_1$ in two configurations: i.e., $d_0(s, v_1) > 1$. Additionally, $d_0(s, v_1) < 5$, since $s$ is connected to the seed in $C_0$. In the induction step, we also add configurations to $\Phi(\mathcal{S})$ only if the particle $p$ that retires on $v_k$ cannot occupy $v_k$ in two configurations. There are two scenarios in which this is possible, since $\mathcal{S}$ is a greedy schedule: (i) $p$ was not in the tree of $s$, but a root itself in $C_0$ with $d_0(p, s) > 1$ or (ii) $p$ was in the tree of $s$ or another root in $C_0$, but the tree was structured in such a way that once $p.parent$ retired, $p$ has to traverse the whole structure of retired particles. We investigate both scenarios separately.

Let us consider the first scenario. Since only 6 particles can be adjacent to the seed, one can easily show that the sum of $d_0(s, v_1)$ plus the distances of consecutive particles around the seed in $C_0$ is also upper bounded by 5. Since, root particles in $C_0$ can expand and contract until they merge with another tree, they need at most two configurations to travel a distance of 1.

Consequently, this part of $\Phi(\mathcal{S})$ is upper bounded by 10.

For the second scenario, assume w.l.o.g. that $p$ is the $k$-th particle that retires and that $C_j$ is the configuration in which $p.parent$ retired. By assumption of the scenario, $d_j(p, v_k)$ is at most the circumference of the retired structure: i.e., $2k + 4$ (since we construct a line) which we add to $\Phi(\mathcal{S})$. Moreover, there is no root between $p$ and $v_k$, since $p$ is the particle that retires next. Thus the tree rooted at $p$ is the only tree in $C_j$. In all configurations after $C_j$ the tree will flatten out along the structure of retired particles. Therefore, all followers in the tree also move along the structure of retired particles before retiring. As a consequence, the second scenario can occur only once, and a term linear in the length of the retired structure is added only once to $\mathcal{S}$.

Therefore, $\Phi(\mathcal{S})$ is upper bounded by $2k + 14$ and we get that in at most $4k + 14$ configurations, nodes $v_1 \cdots v_k$ are occupied by retired particles. □

Given any fair asynchronous activation sequence $A$, let $\mathcal{S}^* = (A, (C_0, \ldots, C_k))$ be a greedy forest schedule, where $C_0$ is the initial configuration of the particle system and $C_k$ is the final line configuration. We can now prove the following theorem.

**Theorem 4.4.** *The total number of asynchronous rounds required by our algorithm to solve the $\mathcal{LSF}$ problem is $\mathcal{O}(n)$.*

*Proof.* First note that Lemma 3.30 of Subsection 3.4.2.2 also holds for the $\mathcal{LSF}$: i.e., the particles form a spanning forest within $\mathcal{O}(n)$ rounds. Lemma 3.30 and Lemma 4.3 imply that $\mathcal{S}^*$ requires $\mathcal{O}(n)$ rounds to solve the $\mathcal{LSF}$. By Lemmas 3.26, the worst-case behavior of $\mathcal{S}^*$ is an upper bound for the runtime of our asynchronous algorithm. □

## 4.4. Hexagon Shape Formation

We now investigate the $\mathcal{HSF}$ problem. The hexagon is constructed around the seed particle. Note that a hexagon actually represents a disk in $G_{\text{eqt}}$, since it can be defined by the set of all nodes of $G_{\text{eqt}}$ within a certain distance $r$ from the seed. Remember that depending on the number of particles, the outmost layer of the final hexagon is not necessarily completely filled. Here, a layer refers to all the particles in the hexagon that have the same distance to the seed.

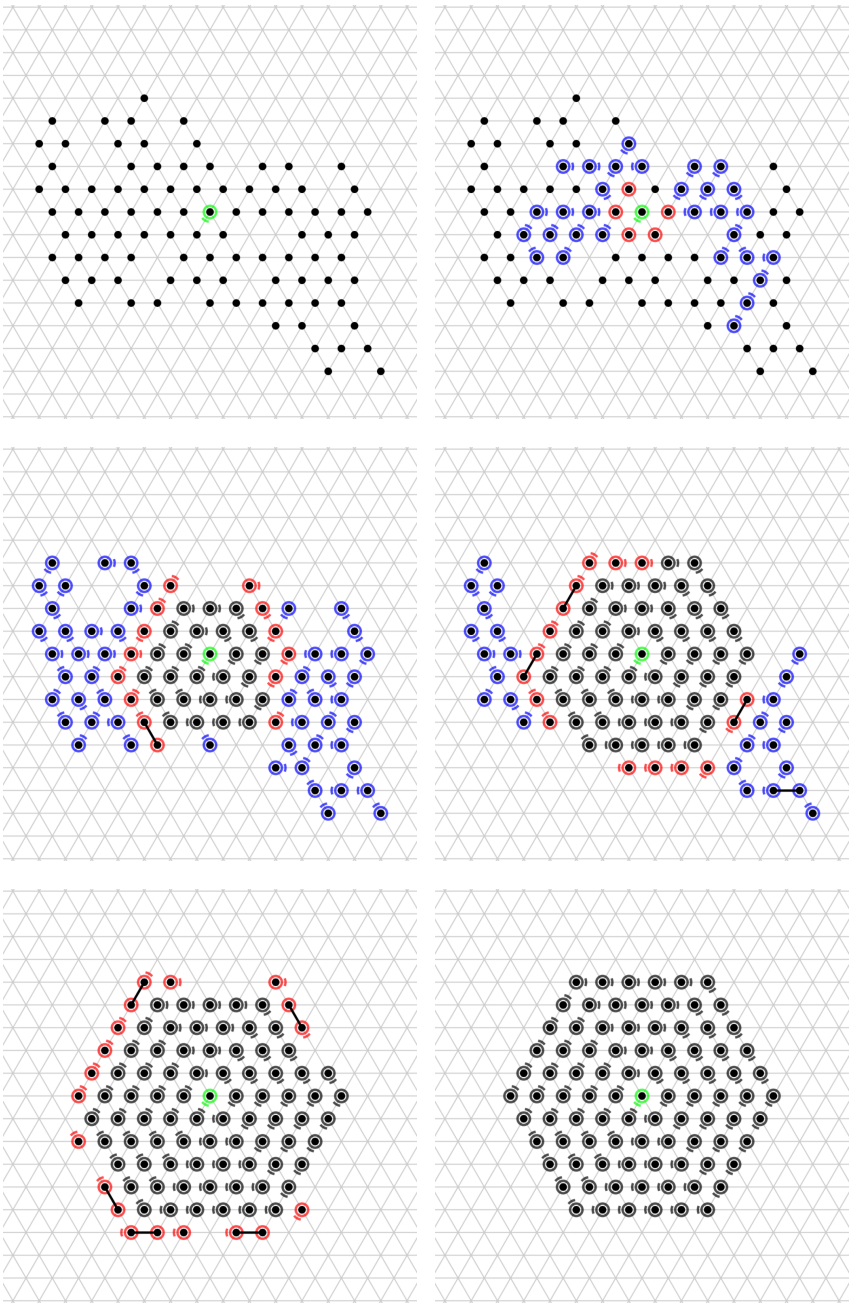


Figure 4.1.: Snapshots of an execution of the $\mathcal{HSF}$ algorithm. The seed is green, retired particles are black, roots are red and followers are blue.

For the $\mathcal{HSF}$, particles retire around the seed in an outwards growing spiral which incrementally adds new layers to the hexagon. Initially, the seed particle $p$ sets the flag $p.retireDir$ to an arbitrary port (e.g., the one labeled 0). Again, any particle adjacent to a retired particle becomes a root and a root $p$ moves in a clockwise fashion around the structure of retired particles until it finds the next node to extend the hexagon snake and becomes retired. Once $p$ becomes retired, it sets the flag $p.retireDir$ to the port that extends the hexagon by forming an outward spiral (see Algorithm 10). More precisely, $p$ computes $p.retireDir$ in the following way: Starting from the port which connects $p$ to the retired particle $p'$ (i.e. $p'.retireDir$ points to the node occupied by $p$), $p$ sets $p.retireDir$ to the port of the first unoccupied node in clockwise order. Thereby, the hexagon grows in a counter-clockwise spiral around the seed. Figure 4.1 depicts some snapshots of a run of the $\mathcal{HSF}$ algorithm on an example instance.

**Algorithm 10** Retirement Condition for $\mathcal{HSF}$

**if** $p$ is a contracted root **then**
  **if** $p$ is adjacent to a retired particle $p'$ such that $p'.retireDir$ points to $p$ **then**
    **while** port $i$ is connected to a retired particle **do**
      $i \leftarrow$ label of next port in clockwise direction
    $p$ sets the flag $p.retireDir$ for port $i$
    $p$ becomes *retired*.

**Theorem 4.5.** *The spanning forest primitive together with the snake formation for the hexagon solve the $\mathcal{HSF}$ problem.*

*Proof.* Again, we need to show that the algorithm terminates and that when it does, the system has the shape of a hexagon. The termination part of this theorem is analogous to the proof presented in Theorem 4.2 (we just have to use Lemma 4.1, which also holds for the $\mathcal{HSF}$). Hence it remains to prove that the structure of retired particles is indeed a hexagon.

Initially, the structure of retired particles contains only the seed particle, therefore the claim holds trivially. By induction, we assume that $C$ is the first configuration in which the current formed structure of retired particles contains $k$ retired particles. By induction hypothesis, assume that those particles form

a (possibly partial) hexagon using $k$ particles. According to Algorithm 10, the only way a root $p$ can become the $(k+1)^{\text{th}}$ retired particle in or after $C$ is if it occupies the node that $q.retireDir$ points to, where $q$ is the $k$-th retired particle in the hexagon. By construction the flag $q.retireDir$ points to the node that either extends the current outer layer of the hexagon or starts the next layer if the current layer is full. Consequently, $p$ retires on node such that the retired structure is a hexagon of $(k+1)^{\text{th}}$ particles. □

By an argument similar to the proof of Theorem 4.4 we can conclude the following theorem.

**Theorem 4.6.** *The total number of asynchronous rounds required by our algorithm to solve the $\mathcal{HSF}$ problem is $\mathcal{O}(n)$.*

## 4.5. Triangle Shape Formation

We finally investigate the $\mathcal{TSF}$ problem. The triangle is constructed using the seed particle as one vertex of the triangle. Again note that depending on the number of particles, the outmost layer of the final triangle is not necessarily completely filled. Again, a layer of the triangle refers to all particles in the triangle with the same distance to the seed vertex.

In order to solve the $\mathcal{TSF}$, one needs to set up the correct rules for retiring particles, which is accomplished by Algorithm 11. The retirement rules for $\mathcal{TSF}$ are more complex than the ones we established for $\mathcal{LSF}$ and $\mathcal{HSF}$. This is due to the fact that we need to explicitly take into account the formation of different *layers* of the triangle as we build it, whereas this was implicitly taken care of by the spiral formation in the $\mathcal{HSF}$ algorithm.

The construction again starts from the seed particle $p$, which occupies one of the triangle vertices. The seed marks two consecutive port labels as the directions along which two edges of the triangle are formed. It uses the flag $p.edge$ which can be of type $left$ and $right$ to mark the corresponding ports. We arbitrarily pick the ports with labels 0 and 1. These directions are propagated further by the particles that retire on the nodes that the ports point to. The seed starts the snake formation by setting the flag $p.retireDir$ on the port with the flag $p.edge[left]$: i.e., its port with label 0. From there on, Algorithm 11 constructs the triangle layer by layer. Layers are filled with retired particles in

an alternating fashion, going "from right to the left" on odd and "from left to the right" on even layers. Every time the snake of retired particles touches one of the edge markers (i.e., a layer is completely filled), it starts a new layer by setting the *retireDir* flag accordingly. Otherwise, particles simply retire in the current topmost layer extending the snake. Figure 4.2 illustrates this approach through some snapshots of the execution of the $\mathcal{TSF}$ algorithm.

**Algorithm 11** Retirement Condition for $\mathcal{TSF}$

```
1:  if p is a contracted root then
2:      if p is adjacent to a retired particle p' such that p'.retireDir points to
        p then
3:          Let i by the port label of p that points to p'
4:          if no adjacent edge is flagged as edge then
5:              p sets p.retireDir on port (i + 3) mod 6
6:          else
7:              Let j be the port label of p that contains the edge flag
8:              p sets the same edge flag on port (j + 3) mod 6
9:              if i ≠ j then                    ▷ p is the last particle of the layer
10:                 p sets p.retireDir on port (j + 3) mod 6
11:             else                             ▷ p is the first particle of the new layer
12:                 if edge.type = left then
13:                     p sets p.retireDir on port (i + 2) mod 6
14:                 else
15:                     p sets p.retireDir on port (i + 4) mod 6
16:         p becomes retired
```

We now show that our algorithm solves the $\mathcal{TSF}$ problem. Similarly to the result of the last sections, we can conclude the following theorem.

**Theorem 4.7.** *The spanning forest primitive together with the snake formation for the triangle solve the $\mathcal{TSF}$ problem.*

*Proof.* Again, we need to show that the algorithm terminates and that when it does, the system has the shape of a triangle. The termination part is again analogous to the proof presented in Theorem 4.2. Hence it remains to prove that the structure of retired particles is indeed a triangle.

Assume we have three particles as the base case (to build the smallest size perfect triangle on $G_{\text{eqt}}$). The seed $p^*$ sets the $p^*.retireDir$ flag and the $p^*.edge[left]$ flag on its port with label 0. A root particle $q$ might have to move

around the seed $p^*$ until it connects to the port 0 of the seed. Since $q$ sees both (edge and retirement) flags coming from the same particle, $p$ becomes retired and it starts constructing a new layer of the triangle by setting its $q.retireDir$ flag such that the next retiring particle continues filling this new layer. Particle $q$ also sets $p.edge[left]$ appropriately to propagate the inherited direction of the edge from the seed to next layer. The only node that the third particle can retire on is the one pointed to by $q.retireDir$ and it is easy to see that the resulting structure of the first three retired particles is a triangle.

Let $C$ be the first configuration in which the current formed structure of the retired particles contains $k$ retired particles, and let $q'$ denote the $(k)^{\text{th}}$ retired particle. By induction hypothesis, assume that those $k$ particles form a (possibly partial) triangle. According to Algorithm 11, the only way a root $r$ can become the $(k+1)^{\text{th}}$ retired particle in or after $C$ is if it occupies the node pointed to by flag $q'.retireDir$. Depending on the location of $q'$ in the triangle, three cases may arise: (i) $q'$ is part of the left edge, (ii) it is part of the right edge or (iii) it is neither of the two. First, consider the case when $q'$ is on the left edge (an analogous argument works if $q'$ is on the right edge). Since $q'$ is the last particle added to the current valid triangle, this results either in a perfect equilateral triangle or in a perfect equilateral triangle plus particle $r$ as the leftmost particle on a newly created layer. In the former case, $r$ retires on the leftmost valid node on the next layer of the triangle structure, pointed by $q'.edge[left]$. In the latter, $r$ simply fills another node of the current topmost layer next to $q'$. In both cases the resulting retired structure forms a valid triangle. Second, consider the situation where $q'$ is not an edge particle. Therefore, $q'$ is located on the topmost (partially filled) layer and $q'.retireDir$ is set to point to the next unoccupied node to continue the layer, which is then filled by $r$, correctly extending the triangle structure and proving the claim. □

Again, we can conclude the following theorem as done in the last two sections.

**Theorem 4.8.** *The total number of asynchronous rounds required by our algorithm to solve the $\mathcal{TSF}$ problem is $\mathcal{O}(n)$.*

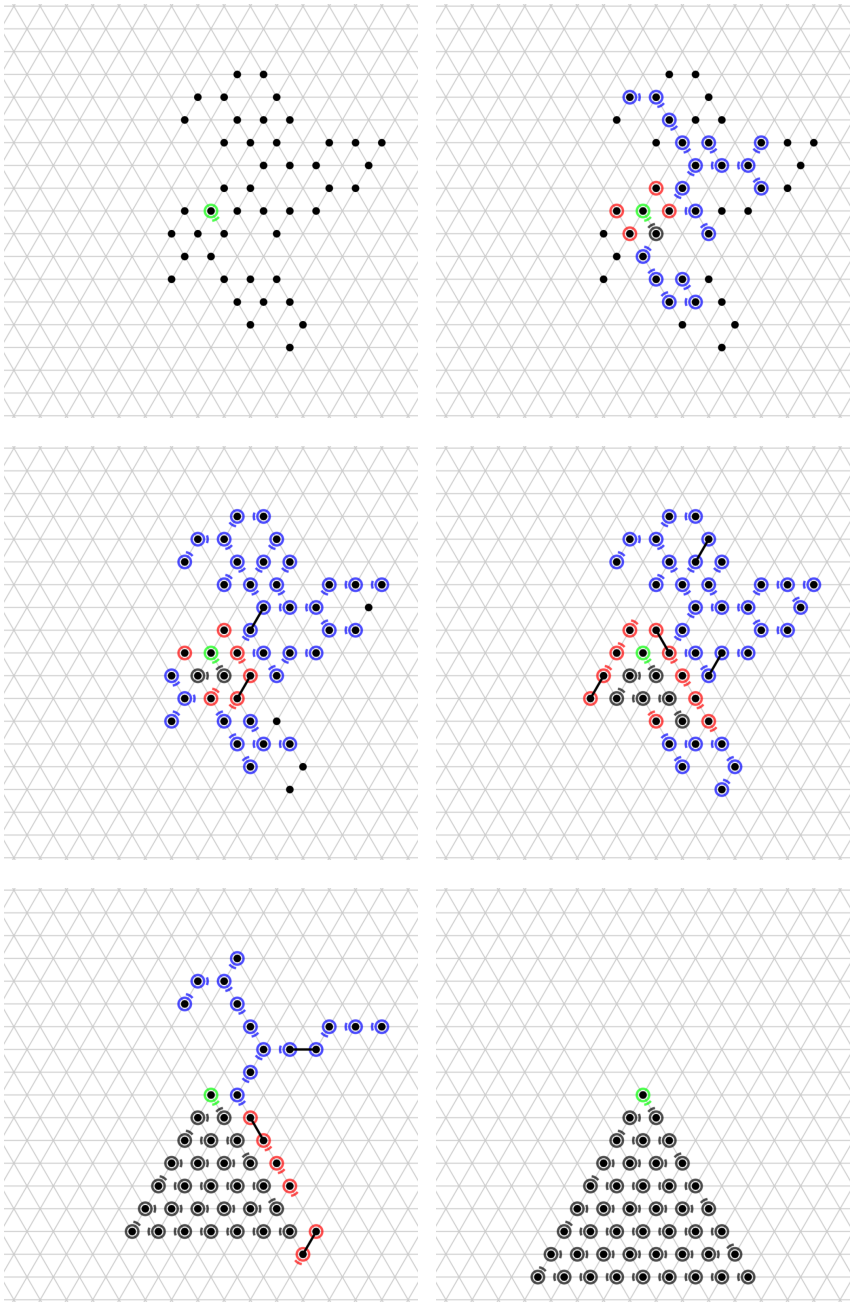


Figure 4.2.: Snapshots of an execution of the $\mathcal{TSF}$ algorithm. The seed is green, retired particles are black, roots are red and followers are blue.

# CHAPTER 5

## Constant Size Particle Systems

> ”*If size really mattered, the whale, not the shark, would rule the waters.*”
>
> Matshona Dhliwayo; Philosopher

An important basic characteristic of programmable matter is the ability of scaling with its size. For example, the more particles there are in the system, the larger the constructed shape can be. If we imagine that each particle is of microscale or nanoscale size, one clearly needs an abundance amount of particles to construct a shape of visible size. Moreover, aiming at an enormous quantity of particles is one of the reasons to create algorithms with linear or even sublinear runtime.

However, there is also a different perspective on particles. Due to their limited computational power and limited maneuverability a constant number of particles cannot perform complicated tasks. Yet, from a complexity point of view, it is interesting to investigate which tasks can be performed by constant size particle systems and whether there are negative results concerning certain tasks. In this chapter we investigate the power of a single particle, two particles and three particles. Since these particles themselves cannot do much except

construct all possible shapes that consist of one, two and three particles, we attach the particle system to a static object. The particles have to gather information about the object to decide whether the object fulfills certain geometric properties (e.g., a convexity test). In total we investigate seven geometric property tests of varying difficulty for the particle systems.

This chapter is not meant as a complete complexity theory analysis on the power of constant size particle systems. We merely want to provide an interesting different point of view on programmable matter outside of its original vision (which caters to large scales), inspired by the intellectual challenge of doing "something with almost nothing".

**Chapter Outline** In Section 5.1 we define the seven problems that we are going to investigate throughout this chapter. The following three sections investigate particle systems of increasing size. Section 5.2 focuses on the smallest possible particle system (i.e., a single particle) and shows that it cannot solve any of the problems. Section 5.3 doubles the number of particles in the system, which allows us to solve five out of the seven problems. Finally, Section 5.4 considers particles systems of size three for which we can devise algorithms for all seven problems.

**Chapter Basis** All the results of this chapter are unpublished. The problem statements and basic ideas were discussed informally at the Dagstuhl Seminar 16721 "Algorithmic Foundations of Programmable Matter". The proof of Theorem 5.1 is a straightforward adaption a proof in a different model and context in our short abstract [Gmy+17] submitted to the 33rd European Workshop on Computational Geometry (a workshop without any formal publications).

## 5.1. Problem Statements

Similarly to the universal coating problem we consider an instance $(P, O)$ where $P$ represents the particle system and $O$ represents a fixed object. Again, let $V(P)$ be the set of nodes occupied by $P$, and $V(O)$ be the set of nodes occupied by $O$. An instance is *valid* if the following properties hold:

(a) All particles are initially contracted and are in an *idle* state.

(b) The subgraphs of $G_{\text{eqt}}$ induced by $V(O)$, $V(P)$ and $V(P) \cup V(O)$, are connected: i.e., there is a single object, all particles are connected and the particle system is connected to the object.

(c) The subgraph of $G_{\text{eqt}}$ induced by $V_{\text{eqt}} \backslash V(O)$ is connected: i.e., the object $O$ has no holes.

Let $m$ be the length of the surface of $O$: i.e., the number of nodes in $V(O)$ that are adjacent to node not in $V(O)$. Throughout this chapter, we investigate problems in which the particle system has to decide whether $O$ fulfills certain geometric properties. A particle system $P$ decides such a problem if, starting from a valid instance, all particles agree whether the object fulfills the property. To be more precise, we consider the following problems.

**Convexity Test:** $P$ has to decide whether the object is convex.

**Line Test:** $P$ has to decide whether the object is a line.

**Triangle Test:** $P$ has to decide whether the object is a triangle.

**Parallelogram Test:** $P$ has to decide whether the object is a parallelogram.

**Rhombus Test:** $P$ has to decide whether the object is a rhombus.

**Hexagon Test:** $P$ has to decide whether the object is a hexagon.

**Regular Hexagon Test:** $P$ has has to decide whether the object is a regular hexagon.

We refer to the last six problems as *shape tests*. The convexity test is a subproblem of all shape tests. Moreover, the parallelogram test is a subproblem of the rhombus test and the hexagon test is a subproblem of the regular hexagon test. Note that there is no necessity for an equilateral triangle test, since every triangle is equilateral in $G_{\text{eqt}}$.

## 5.2. One Particle

In this section we consider a particle system that consists of one single particle. Obviously, the power of a single particle is very limited, in fact we can show the following result.

**Lemma 5.1.** *A single particle cannot decide the convexity test.*

The proof of the lemma is a straightforward adaptation of the proof of a similar statement done by Robert Gmyr in [Gmy+17]. For the sake of completeness, we briefly sketch the high level idea of the proof in the following.

The basic idea is that we can construct an object such that any algorithm $A$ that supposedly decides the convexity test has to fail. More specifically, one can show that the single particle is only able to distinguish between a finite number of solid hexagons (which have to pass the convexity test by definition), since $A$ only uses a constant amount of states. However, there is an infinite number of solid hexagons (since there is an infinite number of side lengths). Consequently, there is an infinite set of hexagon side lengths that are indistinguishable for the particle.

From this observation we can construct an object $O'$ that is a spiral of lines whose lengths are indistinguishable for the particle (see Figure 5.1 for a sketch of $O'$). The size of the spiral (i.e., the number of lines) and the starting position of the particle depends on the number of states that the particle requires to supposedly solve the convexity test. The particle traverses $O'$ similarly to solid hexagons with the corresponding side length. Since the number of sides and their lengths are chosen appropriately, the single particle has to decide that $O'$ is convex, which is clearly a contradiction to the assumption that it decides the convexity test.

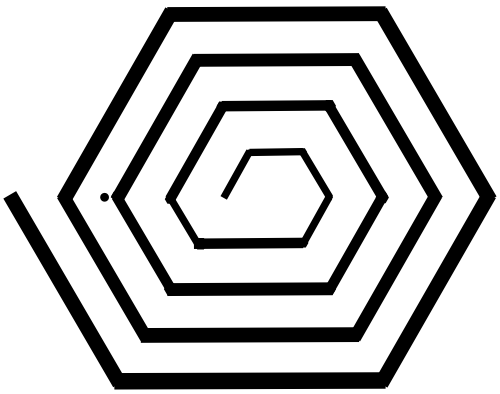


Figure 5.1.: Sketch of an example for $O'$. The black dot marks the initial position of the particle.

This implies that a single particle cannot solve any of the problems in this chapter.

## 5.3. Two Particles

Now consider a particle system of size two. Even though the computational power increased only marginally, two particles can decide quite a number of tests for the object.

The algorithm to solve the convexity test is straightforward. One particle remains idle at a position adjacent to the object. The other particle traverses the surface of the object in one direction, starting at a node adjacent to the other particle. During its traversal it counts the angles of the turns it makes at the vertices and (implicitly) the number of vertices of the object. Note that this counting is not conflicting with the constant-size memory of particles, as we will see shortly. For the algorithm, it is sufficient that the angles of the turns are computed according to the local orientation of the particle (i.e., its port labels). Without loss of generality assume that the particle considers a clockwise turn as a positive angle and a counter-clockwise turn as a negative angle.

Due to the geometric properties of $G_{\text{eqt}}$there are only four convex shapes: the line, the triangle, the parallelogram and the hexagon. Consequently, the moving particle only needs to store whether it sees at most 6 vertices of the object before it returns to the static particle (i.e., if there are more vertices, there is no need to store the exact amount of them). Additionally, it checks whether the angles of the object are either all positive or all negative. If so, the object is convex, otherwise it is not.

**Theorem 5.2.** *Two particles can decide the convexity test in time $\mathcal{O}(m)$.*

*Proof.* Since one particle remains static, the moving particle $p$ can determine that it has traversed the surface of the object. It is easy to see that an object is not convex if $p$ sees clockwise and counter-clockwise angles on its traversal. Since the algorithm exactly tests this property the correctness follows immediately.

The moving particle has to traverse the whole surface of $O$ before returning to the static one. Thus, the algorithm requires $\mathcal{O}(m)$ rounds. □

The algorithm to decide the convexity test can easily be adapted to decide the line test, the triangle test, the parallelogram test and the hexagon test. The major variation is the number of vertices that the moving particle $p$ counts and the test for the angles at each vertex of the object. For example, in the triangle test, $p$ needs to check whether the object has exactly three vertices and if it turns 120° at each vertex. Similar adaptions have to be done for the other three tests. In fact, we can combine all four shape test algorithms and the convexity test algorithm into one algorithm that decides all five problems at once. As a consequence, we can conclude the following corollary.

**Corollary 5.3.** *Two particles can decide the line test, the triangle test, the parallelogram test and the hexagon test in time $\mathcal{O}(m)$.*

We conclude this section by showing a negative result concerning the last two remaining shape tests. In order to show that two particles cannot decide the rhombus test and regular hexagon test, we investigate an essential subproblem: the ability to decide whether two adjacent line segments of a given object are of equal length. The length of a line in $G_{\text{eqt}}$ refers to the number of nodes the line occupies. Formally, in the *line length comparison* test, for a given object $O^*$ that consists of two straight lines of lengths $l_1, l_2$ which meet at one of their endpoints the particle system decides, whether $|V(l_1)| = |V(l_2)|$. See Figure 5.2 for an example instance. As an intermediate step we first show that it is impossible for one particle to decide the line length comparison test.

Figure 5.2.: An example instance of an object for the line length comparison test. The object fails the test.

**Lemma 5.4.** *A single particle cannot decide the line length comparison test.*

The proof of Lemma 5.4 is straightforward adaption of the proof of Lemma 5.1 as presented in [Gmy+17]. To enhance clarity and readability, we present a

full proof here, since some of the main techniques are used and enhanced in the upcoming proof for the two particle case.

*Proof.* Suppose that there is an algorithm that allows a single particle to decide the line length comparison test and let $k$ be the number of states used by the algorithm. Consider the execution of the algorithm on an instance where the object consists of two lines of the same length $\ell$ and the particle is initially adjacent to one endpoint of a line. We subdivide the execution of the algorithm into *phases*. A new phase of the execution starts whenever the particle is contracted and adjacent to one of the endpoints of the line or the node where the lines meet. Observe that the algorithm runs for at most $3k$ phases before the particle decides that the lines have the same length. Otherwise, the algorithm would again enter an infinite loop, which is a contradiction to the supposition.

To capture the path of the particle, we define the *traversal sequence* of the particle associated with line length $\ell$ as $(\,(v_1, q_1), (v_2, q_2), \ldots, (v_t, q_t)\,)$, where $t$ is the number of phases the algorithm takes until the decision is made, $v_i$ is the vertex occupied by the particle at the beginning of phase $i$, and $q_i$ is the state of the particle at the beginning of phase $i$. Since the algorithm takes at most $3k$ phases to make its decision (independently of $\ell$), there is a finite number of traversal sequences. Moreover, there is an infinite number of line lengths $\ell$. Thus, there exists an infinite set of line lengths $\mathcal{L}$ which have the same traversal sequence by the pigeonhole principle. Consequently, all objects with line lengths in $\mathcal{L}$ are indistinguishable for the particle.

Thus, we can define an object $O'$ for which the algorithm fails. The object consist of two lines, one of length $l'$ and one of length $l''$ such that $l' \neq l''$ and $l', l'' \in \mathcal{L}$. When our algorithm is executed on $O'$ we can again subdivide execution of the algorithm into phases (as defined above). Each phase of the execution corresponds to a traversal sequence for an object whose lines are of equal length and whose line length is in $\mathcal{L}$. Consequently, the algorithm has to decide that both lines are of equal length, which is a contradiction to the assumption that the algorithms works correctly. □

**Lemma 5.5.** *Two particles cannot decide the line length comparison test.*

*Proof.* For contradiction, suppose that there is an algorithm that allows two particles to decide the line length comparison test. For convenience we explicitly

name the particles $p_a$ and $p_b$. Let $k_a$ be the number of states used by $p_a$ and $k_b$ be the number of states used by $p_b$: i.e., we explicitly allow each particle to have its own algorithm. Without loss of generality let $p_a$ be the particle that makes the ultimate decision whether the object passes the line length comparison test. There are three general cases that are possible for our algorithm:

(a) Both particles always stay connected until $p_a$ makes its decision.

(b) One particle moves while the other one remains static.

(c) Both particles execute individual algorithms.

In the first case, we can interpret both particles as one super particle since they always stay connected. This super particle of course has more memory than just one single particle. Nevertheless, we can apply Lemma 5.4.

In the second case, only one of the particles moves. It can use the other static particle as a checkpoint and as an external memory. However, Lemma 5.4 is still applicable, since we can treat the static particle as just another starting point for a phase of the moving particle: i.e., the number of traversal sequences remains finite.

Therefore, it remains to consider the last case and an execution of the algorithm on the instance where the object consists of two lines of the same length $\ell$, and both particles are initially adjacent to one endpoint of a line and adjacent to each other. For the particle system we subdivide the execution of its algorithm into *combined phases* where a new phase of the execution starts whenever one of the particles is contracted and adjacent to one of the endpoints of the line, the node where the lines meet or the other particle. The algorithm runs for at most $4(k_a + k_b)$ combined phases before $p_a$ decides that the lines have the same length.

In order to show the desired result we need to extend the concept of traversal sequences and extend the proof of Lemma 5.4 . We define the *combined traversal sequence* of the particle system associated with $\ell$ as a sequence of quintuples $(\,(p, v_1^a, q_1^a, v_1^b, q_1^b), (p, v_2^a, q_2^a, v_2^b, q_2^b), \ldots, (p_a, v_t^a, q_t^a, v_t^b, q_t^b)\,)$, where $t$ is the number of combined phases the algorithm takes until the decision is made, $p$ is in $\{p_a, p_b\}$ and denotes the particle who entered a new phase, $v_i^a$ and $v_i^b$ are the vertices occupied by $p_a$ and $p_b$ at the beginning of the combined

phase $i$, and $q_i^a$, $q_i^b$ are the states of the particles at the beginning of phase $i$. Note that this notion clearly allows that the occupied vertex and the state of one of the particles stays the same for multiple consecutive combined phases: i.e., it is possible that only one particle moves while the other one remains static. Additionally, it is possible that $v_i^a$ or $v_i^b$ might contain two nodes, since one particle can be expanded while the other one starts a new phase. Again, the number of combined traversal sequences is finite, whereas the number of line lengths is infinite. Therefore, we can apply the pigeonhole principle again, which implies that there is an infinite set of line lengths $\mathcal{L}$ with the same combined traversal sequence. Similarly to the proof for the single particle case, we can construct an object for which the algorithm fails. □

From Lemma 5.5 we can easily conclude the following theorem, which concludes our analysis of the two particle scenario.

**Theorem 5.6.** *Two particles cannot decide the rhombus test and the regular hexagon test.*

## 5.4. Three Particles

We conclude our analysis of constant size particle systems, by showing that a particle system of size three can decide both the rhombus test and the regular hexagon test. In the following we describe the algorithm for the rhombus test. The algorithm for the regular hexagon is a straightforward adaption: i.e., we simply have to compare three side lengths instead of two.

The algorithm to decide the rhombus tests compares two adjacent side lengths of a parallelogram. Indeed, we assume that the particle system already decided that the object is a parallelogram and, thus, knows that it needs to compare two side lengths in order to decide the problem. Moreover, we assume that the three particles are all adjacent to one vertex of the parallelogram: i.e., they form a curved line that is bent around one of the vertices. For convenience, we name the particles $p_{left}$, $p_{center}$ and $p_{right}$. Note that it is not important whether $p_{left}$ is really to the left of $p_{center}$ from a global point of view, it is simply sufficient that $p_{center}$ is indeed the particle in the middle and it can distinguish between the two other particles.

Initially, $p_{center}$ aims at moving in the direction of $p_{left}$. Since it is adjacent to $p_{left}$, it makes $p_{left}$ move up one node along one side of the parallelogram. Then $p_{center}$ switches directions and moves to $p_{right}$. Once it is adjacent to $p_{right}$, it pushes $p_{right}$ one node along the other side of the parallelogram as well. Now $p_{center}$ again makes the switch and moves back to $p_{left}$. This process continues: i.e., $p_{center}$ always alternates directions and pushes the two other particles along their side of the parallelogram such that they always move up one node.

The algorithm can decide the test once one of the two outside particles is adjacent to another vertex of the parallelogram. If $p_{right}$ is the first particle to fulfill this property, the parallelogram is not a rhombus, since we initially started moving in the direction of $p_{left}$. If $p_{left}$ is the first particle adjacent to another parallelogram vertex, $p_{center}$ has to do one last run to $p_{right}$. In case $p_{right}$ is then also adjacent to a vertex once it has been pushed by $p_{center}$, the parallelogram is a rhombus; otherwise it is not. We can conclude the following theorem.

**Theorem 5.7.** *Three particles can decide the rhombus test in time $\mathcal{O}(m^2)$.*

*Proof.* The algorithm decides that the parallelogram is a rhombus only if $p_{right}$ and $p_{right}$ traveled the same distance to the respective vertices of the parallelogram. Consequently, this can happen only if both sides of the parallelogram indeed have the same length. If the parallelogram is not a rhombus, one of two adjacent sides has to be longer than the other. Since $p_{center}$ alternates its direction and pushes each of the outside particles one node at a time, it can detect that one side is longer.

The distance particle $p_{center}$ traverses can be upper bounded by the sum $\sum_{i=0}^{k} i + 1$, where $k$ is the length of the shorter side of the parallelogram. This side length is at most $\frac{m}{4}$. Therefore, algorithm requires $\mathcal{O}(m^2)$ rounds. □

As a corollary from the theorem, we get the following result which concludes our analysis in this section.

**Corollary 5.8.** *Three particles can decide the regular hexagon test in time $\mathcal{O}(m^2)$.*

This concludes our investigation on constant size particle systems.

# CHAPTER 6

# Conclusion of Part I

> *It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.*
>
> Isaac Asimov, Writer and Professor of Biochemistry

Let us conclude the first part of this thesis by subsuming all technical results of the previous chapters. As an addendum, we will also mention further additional results and future work directions.

Throughout the first part we investigated three different problem scenarios in the field of self-organizing programmable matter. In Chapter 3, we investigated the universal coating problem, in which the particle system has to uniformly cover a given static object. The Universal Coating Algorithm is a combination of different asynchronous primitives which are integrated seamlessly without any underlying synchronization. This algorithm provably solves the universal coating problem and requires $\mathcal{O}(n)$ asynchronous rounds in the worst case. This runtime is worst-case optimal.

Chapter 4 focused on building basic geometric shapes from programmable

matter. More specifically, we aimed at building a line, a triangle and a hexagon. We presented an algorithmic framework which can construct all three shapes in $\mathcal{O}(n)$ asynchronous rounds by just varying two rules in the algorithms.

In Chapter 5 we investigated the power of constant-size programmable matter. In doing so, we concentrated on the scenario in which the matter is attached to a static object and needs to evaluate various tests concerning the shape of the object. We provided some negative results concerning programmable matter that consists of only a single particle or two particles. This was contrasted by positive results (i.e., explicit algorithms) for different shape properties which are indeed decidable by two or three particles.

The remainder of this chapter is dedicated to further results and future work. In Section 6.1 we explain some of our related results in the area of programmable matter. These findings provide a nice contrast to the technical results of the first part and provide some insight into the algorithmic work we have done in the last years. Section 6.2 considers different directions for future results in the field.

## 6.1. Further Results

In this section we present two of our results in the area of programmable matter. Even though these results are not included in the technical parts of this thesis, they are closely related to the investigated problems. The high level algorithmic descriptions in this section are directly taken from their respective papers [Day+17] and [Der+16b].

The first major result concerns the problem of **leader election** [Day+17]: i.e., all particles start in the same state and an algorithm elects one particle as the leader while all others are non-leaders. The algorithm requires a linear number of asynchronous rounds with high probability. It operates on a *static* particle system – throughout the whole execution of the algorithm the particles do not move. Solving leader election is necessary for our universal coating algorithm in Chapter 3, since we need a specific leader particle on the surface layer. Moreover, the shape formation algorithms of Chapter 4 require a specific seed particle to form any kind of shape. Such a seed particle could be chosen by a leader election algorithm.

In the following we briefly describe the leader election algorithm. The

algorithm consists of six phases. These phases are not strictly synchronized among each other: i.e., at any point in time, different parts of the particle system may execute different phases. In the first phase each particle checks whether it is part of a boundary of the particle system (i.e., it is not completely surrounded by particles). Only boundary particles participate in leader election and the remaining five phases are executed on each boundary individually. In the second phase boundaries are subdivided into segments. In order to do so each particle flips a coin: particles that flip heads become leadership *candidates*, whereas particles that flip tails become non-candidates. A segment of a boundary consists of a candidate and all subsequent non-candidates up to the next candidate (in a pre-given direction of that boundary). In the third phase candidates are assigned a random identifier that is stored distributively in the particles of its segment. These identifiers are used in the fourth phase, in which candidates compete for leadership. To do so, we use an intricate token passing scheme that forwards all identifiers on a boundary along that boundary. If a candidate sees an identifier that is higher than its own, it revokes its candidacy. If it sees a lower identifier it stays a candidate. Whenever a candidate sees its own identifier the fifth phase is triggered in which it checks whether it is the last remaining candidate on its boundary. In that case, the remaining candidate initiates the final phase to determine whether it occupies the unique *outer boundary* of the system. If so, it becomes the leader. Otherwise, it revokes its candidacy.

As already mentioned this algorithm elects a leader in a linear number of rounds with high probability. In fact, the runtime is linear in the length of the outer boundary and we need a number of rounds that is linear in the diameter of the particle system to distribute the result of leader election to all particles. Most parts of the algorithm are deterministic, only phase two and phase three use randomness. The runtime is worst-case optimal.

The second result that we want to highlight is an algorithm to solve **universal shape formation** [Der+16b]. This algorithm takes an arbitrary input shape composed of a constant number of equilateral triangles of unit size (called faces) and lets the particles build that shape at a scale depending on the number of particles in the system. Our algorithm runs in $\mathcal{O}(\sqrt{n})$ asynchronous rounds, where $n$ is the number of particles in the system, provided we start from a well-initialized configuration of the particles (i.e., they form a giant equilateral

triangle). This is optimal in a sense that for any shape deviating from the initial configuration, any movement strategy would require $\Omega(\sqrt{n})$ rounds in the worst case. This result is an interesting contrast to the shape formation algorithms of Chapter 4. Our universal algorithm seems to be more powerful due to its universality. However, the universal shape formation algorithm cannot construct certain basic shapes, since they cannot be described by a constant number of equilateral triangles of unit size: e.g., it cannot build a thin line as done in Section 4.2.2.

Again, we are describing the general idea of the algorithm in the following. The underlying principle to achieve the desired runtime of shape formation is to move triangles of particles (which correspond to the unit size triangles in the input shape) en bloc in a parallel fashion. This can be achieved in $\mathcal{O}(\ell)$ asynchronous rounds, where $\ell$ is the side length of the triangle. Before the final shape is constructed, a preprocessing phase transforms the initial equilateral triangle of all particles into an intermediate structure. This structure is a giant line of smaller triangles whose size is small enough to have enough triangles for the latter shape formation but large enough to consume all particles in the final shape. This structure is not perfect and the number of smaller triangles does not exactly match the number of input faces. After the intermediate structure is built, one particle (i.e., one elected by leader election) computes a construction plan of the input shape: i.e., a permutation of the unit size input faces that dictates the chronological order in which the corresponding triangles are placed. This computation is far from trivial since one has to account for connectivity constraints, the possibility of walling in and imprecisions that are introduced in the construction of the intermediate structure. The process of building the shape itself is relatively straightforward: i.e., the triangles are moved in their position according to the construction plan.

## 6.2. Future Work

We conclude the first part of this thesis by pointing out interesting directions for future work. Since the area of programmable matter is relatively young, one can imagine a multitude of directions that allow for interesting research questions. Therefore, we focus on the future work possibilities that are geared towards the bigger picture of programmable matter, instead of focusing on

specific open algorithmic problems in the current state of the amoebot model.

One of the biggest challenges is to transfer the amoebot model from two-dimensional to three-dimensional space. This requires some adoption of the model, especially of the underlying grid graph, which has to allow for three-dimensional movement. Once this modeling step has been done, all algorithmic challenges that are solved in the current model are open in 3D: i.e., coating, shape formation and leader election need a thorough re-investigation. It seems plausible that some of our developed primitives do in fact work (e.g., the spanning forest), however most problems should indeed become harder in three dimensions.

Another interesting possibility is to make the amoebot model more realistic in terms of failures. In the current state of the model, everything works as intended — communication never fails, movements always succeed and particles always behave predictably. However, real-life programmable matter would not have the luxury of performing in a perfect world. One minor step to get to a more realistic model is to allow for particle failures that are detectable by other (correctly working) particles. Accordingly, the correctly working particles could try to solve their original problem and either ignore failed particles or exclude them from the system. Additionally, one could think about models in which movements are not perfect: i.e., instead of moving exactly to a node in $G_{\text{eqt}}$, particles can deviate by an $\varepsilon$ factor which is not under their control. Finally, one could envision a more adversarial model in which a failure leads to faulty particle behavior that is non-detectable.

Lastly, one could envision a model in which active particles and passive tiles are mixed: i.e., particles which are (similar to) amoebots that can carry passive tiles which cannot move, but be used as an external storage. An interesting problem for such a model is tile shape formation. The particles aim at restructuring a fixed tile set and one can study whether increasing the number of particles leads to a construction time speed-up. Problems similar to our investigation in Chapter 5 can also become interesting in this scenario.

# Part II.

# Monotonic Searchability for Self-Stabilizing Topologies

# CHAPTER 7

## Prologue

> ”*We are all now connected by the Internet, like neurons in a giant brain.*”
>
> Stephen William Hawking; Theoretical Physicist and Cosmologist

Life in the 21st century seems almost unimaginable without the Internet. In fact, the Internet is, without any doubt, one of the major accomplishments of computer science in the last decades, maybe even of the last century. It has grown from something that was coined an “information management system” by its inventor, the 2017 *ACM Turing Award* winner Tim Berners-Lee [Ber89] into a juggernaut of information, communication and (social) networks. Almost every part of modern life is directly or indirectly influenced by the Internet. And even though one might easily argue that not all (side-) effects of the Internet are beneficial for society (e.g., the recent “fake news” controversy of Donald Trump, various bugging scandals, illegal file sharing, etc.) on a larger scale the benefits seem to outweigh the detriments. From a scientific point of view, the Internet is an interesting phenomenon that can be investigated from different scientific disciplines and research angles.

Throughout the second part of this thesis we focus on a very technical part

and point of view of the Internet — *overlay networks*. An overlay network – or short overlay – is a virtual computer network that is built on top of another virtual or physical network. Abstractly speaking, nodes in the overlay network (which usually correspond to computers or users) can be thought of as being connected by virtual or logical links. These links might coincide to a path in the underlying network, perhaps through many physical links. Examples for overlays occur in different sizes and varieties throughout the Internet. Even the Internet itself can be seen as an overlay that is established on top of the physical network cables and routers. Any modern communication platform (e.g., WhatsApp, Skype, Discord) creates its own virtual overlay network to allow for communication between users. Early-day file sharing platforms (e.g., Gnutella, eDonkey, BitTorrent) heavily relied on peer-to-peer overlays to share files among their users. Even social networks like Facebook, Twitter, Instagram, etc. can be seen as overlays in which nodes of the overlay correspond to users and a link corresponds to a friendship.

For this thesis we are most interested in *peer-to-peer* overlays, i.e., networks in which each user has a bit of control over the overlay network, instead of having one server as the de facto ruler. In these networks nodes are allowed to change the links of the network and are therefore able to change the *topology* of the network. (In-)famous examples from the past for this kind of overlay are the already mentioned file sharing platforms, as well as Spotify and Skype, which have been using peer-to-peer overlays to enhance their streaming or phone call services. The most prominent current examples are digital cryptocurrencies like Bitcoin.

For large-scale peer-to-peer overlays unforeseen changes and faults are not an exception but the rule. Consequently, mechanisms are needed which ensure that whenever there are problems they are quickly repaired, and that all parts of the system, which are still functional are not negatively affected by the repair process. One class of protocols that is tailored to this task are *self-stabilizing* protocols. These protocols have the major advantage that they do not have to be well initialized in order to function properly. In fact, they are guaranteed to *converge* from any possible initial state to a desired state (called a *legitimate* state). Moreover, once a legitimate state has been reached, a self-stabilizing protocol always stays in that state as long as no faults occur. Self-stabilizing protocols for many different distributed computing problems have been studied

in the last four decades and there is a wide range of existing self-stabilizing overlay protocols (see Section 7.2 for a review on related literature).

From the presented practical examples for overlays one can clearly identify a common theme: an overlay does not serve as an end in itself, but as a means for purpose that is in close relation to the specific application. Throughout this thesis we want to focus on one specific purpose for an overlay that is highly inspired by one of the most popular online activities: *searching*. In the 2015 statistic "Most popular online activities of adult Internet users in the United States as of July 2015" of the NTIA (National Telecommunications and Information Administration) [TAB16] at least five of the ten most popular activities involve searching in some way or another. However, since searching is a very broad term that carries many different meanings, we want to concentrate on a very specific search – searching for other participants in the network, i.e., a participant $a$ of the overlay knows the name (or an identifier) of another possible participant $b$ and wants to know whether $b$ is present in the overlay. If $b$ is present and $a$ is able to find $b$ with a search message we say that $b$ is *searchable* for $a$.

In this part of the thesis we investigate self-stabilizing overlay protocols which maintain *searchability* in a *monotonic* fashion. Consequently, our protocols not only converge from any initial state that is *weakly connected* to the desired topology, but also have to make sure that once searchability between two nodes is established it is preserved during self-stabilization: i.e., once $b$ is searchable for $a$ it is always searchable in the future. This task is in general highly non-trivial and one has to reinvestigate existing self-stabilizing overlay protocols. More specifically, we concentrate on monotonic searchability for linear topologies (see Section 8.1 for the precise problem statement), since they yield a straightforward routing protocol and provide the perfect starting point to investigate the feasibility of monotonic searchability.

## 7.1. Model

We consider a distributed system consisting of a fixed set of nodes in which each node has a unique immutable numerical *identifier* (ID, for short) that serves as a reference for other nodes. The system is controlled by a protocol that specifies the *variables* and *actions* that are available in each node. In addition to the protocol-based variables there is a system-based variable for each node called *channel* whose value is a set of messages. We denote the channel of a node $u$ as $u.Ch$ and it contains all incoming messages for $u$. The message capacity of each channel is unbounded and we assume no message loss. A node $v$ can send a message to $u$ by adding a message to $u.Ch$. This is possible only if $v$ knows the reference of $u$. Besides these channels there are no further communication means, so only point-to-point communication is possible.

There are two types of actions that a protocol can execute. The first type has the form of a standard procedure $\langle label \rangle(\langle parameters \rangle) : \langle command \rangle$, where *label* is the unique name of that action, *parameters* specifies the parameter list of the action, and *command* specifies the statements to be executed when calling that action. Such actions can be called locally (which causes their immediate execution) and remotely. In fact, we assume that every message must be of the form $\langle label \rangle(\langle parameters \rangle)$, where *label* specifies the action to be called in the receiving node and *parameters* contains the parameters to be passed to that action call. All other messages are ignored by nodes. The second type has the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$, where *label* and *command* are defined as above and *guard* is a predicate over local variables. An action whose guard is simply **true** can be executed any time and is called a *timeout* action.

The *system state* is an assignment of values to every variable of each node and messages to each channel. An action in some node $u$ is *enabled* in some system state if its guard evaluates to **true**, or if there is a message in $u.Ch$ requesting to call it. In the latter case, when the corresponding action is executed, the message is processed (in which case it is removed from $u.Ch$). An action is *disabled* otherwise. Receiving and processing a message is considered an atomic step.

A *computation* is an infinite fair sequence of system states such that for each

state $s_i$, the next state $s_{i+1}$ is obtained by executing an action that is enabled in $s_i$. This disallows the overlap of action execution, i.e., action execution is *atomic*. We assume *weakly fair action execution* and *fair message receipt*. Weakly fair action execution means that if an action is enabled in all but finitely many states of a computation, then this action is executed infinitely often. Note that a timeout action of a node is executed infinitely often. Fair message receipt means that if a computation contains a state such that there is a message in a channel of a node which enables an action, then the corresponding action is eventually executed with the parameters of that message: i.e., the message is eventually processed. Besides these fairness assumptions, we place no bounds on message propagation delay or relative execution speeds of nodes: i.e., we allow fully asynchronous computations and non-FIFO message delivery. A *computation suffix* is a sequence of computation states past a particular state of this computation. In other words, any suffix of a computation is obtained by removing the initial state and finitely many subsequent states. Note that a computation suffix is also a computation. For a given computation we call the first state of the computation the *initial state*. For two states $s, s'$, we say $s'$ is *reachable* from $s$ if starting in $s$ there is a sequence of action executions such that we end up in state $s'$. Additionally, we use the notion $s < s'$ as a shorthand to indicate that $s$ happened chronologically before $s'$.

We consider protocols that do not manipulate the internals of node identifiers. Specifically, a protocol is *compare-store-send* if the only operations that it executes on identifiers is: (i) comparing them, (ii) storing them in local memory and (iii) sending them in a message. Therefore, operations on identifiers such as addition, radix computation, hashing, etc. are not used. In a compare-store-send protocol, a node may learn a new identifier of a node only by receiving it in a message. A compare-store-send protocol cannot create new identifiers and can only operate on the identifiers given to it.

The overlay network of a set of nodes is determined by their knowledge of each other. We say that there is a (directed) *edge* from node $u$ to node $v$, denoted by $(u, v)$, if node $u$ stores a reference (i.e., the ID) of $v$ in its local memory or has a message carrying the reference of $v$ in $u.Ch$. In the former case, the edge is called *explicit*; in the latter case, the edge is called *implicit*. As already stated, messages can only be sent via explicit edges. We denote the directed *network (multi-)graph* given by the explicit and implicit edges with

$G$. Additionally, $G_E$ is the subgraph of $G$ induced by only the explicit edges. A *weakly connected component* of a directed graph $G$ is a subgraph of $G$ of maximum size such that for any two nodes $u$ and $v$ in that subgraph there is a (not necessarily directed) path from $u$ to $v$. Two nodes that are not in the same weakly connected component are *disconnected*.

In this part of the thesis we are particularly concerned with search requests, i.e., SEARCH$(v, destID)$ messages that are routed along $G_E$ according to a given search protocol, where $v$ is the sender of the message and $destID$ is the identifier of a node we are looking for. We assume that SEARCH() requests are initiated locally by a (possibly user controlled) application operating on top of the network. Note that $destID$ does not need to be the identifier of an existing node $w$, since it is also possible that we are searching for a node that is not in the system. If a SEARCH$(v, destID)$ message reaches a node $w$ with $id(w) = destID$, the search request *succeeds*; if the message reaches some node $u$ with $id(u) \neq destID$ and cannot be forwarded anymore according to the given search protocol, the search request *fails*. For a given identifier $ID$, each node $u$ can decide for each neighbor $v$ whether $v$ is closer to the node $w$ with $id(w) = ID$ if such a node exists (we also say that $id(v)$ is closer to $ID$ than $id(u)$).

## 7.2. Related Literature

The idea of self-stabilization in distributed computing was introduced in the seminal paper by E.W. Dijkstra in 1974 [Dij74], in which he investigated the problem of a self-stabilizing token ring. Even though the paper is incredibly short and concise compared to scientific papers that appear today, it has created its own research field in the last four decades. Consequently, it is beyond the scope of this thesis to cover literature of the whole self-stabilization field. In fact, to the best of our knowledge there is no comprehensive up-to-date survey article that is able to cover the full spectrum of self-stabilization. Readers that want to familiarize themselves with basics of self-stabilization are referred to the book of Shlomi Dolev from 2000 [Dol00]. Additionally, the survey of Jerzy Brzezinski and Michal Szychowiak from the same year [BS00] provides one possible starting point for an in-depth literature review. In the last 15 years, surveys have only specialized on certain subfields of self-stabilization: e.g, algorithms for independent set, dominating set, coloring, and matchings [GK10] or scheduling hypotheses for self-stabilization [DT11]. As a consequence, we focus on the two subfields of self-stabilization that are most important for the results in this part of the thesis: *topological self-stabilization* and *safety property maintenance* during the convergence phase of self-stabilization. Note that we exclude the vast amount of P2P literature from this section, since it is our ultimate goal to create self-stabilizing protocols. Again. the content of this section is a culmination of all related literature parts within our papers in the area of self-stabilization [SSS15; KSS16; SSS16].

**Topological Self-Stabilization** The basic idea of topological self-stabilization, is to apply the concept of self-stabilization on graph topologies: i.e., an overlay network is initially in an arbitrary weakly connected state and the aim is a protocol that recovers to a certain predefined network topology. The first topologies that researchers investigated were rings [CF05; SR05] and lines [Ang+05; ORS07; Gal+14]. The basic idea for these topologies is that nodes always keep those edges that are needed for the list/ring (from its local point of view) and *linearizes* all other edges by sending them to its neighbors. It needs to be noted that some of these early protocols (e.g., [ORS07; Ang+05]) investigate a synchronous setting, instead of the asynchronous setting that is

commonly used today. Moreover, the protocol of [Ang+05] is not self-stabilizing in the classical sense. However, it was shown in a follow-up paper [AW07] that it can be made self-stabilizing if one assumes that nodes initially have an out-degree of 1.

Over the years more complex network topologies were investigated which are either inspired by the self-stabilizing list or use it as a subroutine. Richa et al. [RSS11] investigate a self-stabilizing De Bruijn network. They first present a transformation how a classical De Bruijn graph can be linearized such that all nodes of the network are sorted on a line and then use the techniques of [ORS07] to construct the line topology. In order to do so, each node in the network emulates two virtual nodes that are linearized as well. One consequence of their approach is the fact that the graph of the real nodes is not weakly connected even though the graph of real and virtual nodes is. They circumvent this problem by using a light-weight probing algorithm that checks whether each node is in a connected component with its virtual nodes. Nor et al. [NNS13] considered a self-stabilizing version of the skip list of Pugh [Pug90] peer-to-peer topology. As the name suggests, the techniques of [ORS07] can be used to stabilize the topology. In addition, the paper investigates the necessary conditions on initial states that hold in general for topological self-stabilization. Jacob et al. [Jac+] investigate self-stabilizing skip$^+$ graphs, which are an extension of the classic skip graphs of Aspnes and Shah [AS07]. This extension is necessary since the correctness of a skip graph cannot be checked locally. Intuitively, their protocol uses the linearization technique on each level of the skip$^+$ graph. Kniesburges et al. [KKS12] use the self-stabilizing line as the major building block to present a protocol that stabilizes to the 1-dimensional version of the small-world network of Chaintreau et al. [CFL08]. The key idea of the protocol is to establish a ring network that is enhanced with long-ranged links. These links are forwarded over time and can be forgotten by a node with a certain probability. Finally, there exists a self-stabilizing version of the popular Chord protocol called Re-Chord [KKS14]. The authors also use the linearization technique to achieve the desired protocol, which is in essence a careful reconsideration of the original Chord protocol that respects arbitrary initial states.

In [Jac+12] Jacob et al. present a self-stabilizing protocol for Delaunay graphs. In order to do so, each node computes a local Delaunay graph from its

knowledge and sends all non-Delaunay edges to its Delaunay neighbors. This is, in substance, the 2-dimensional interpretation of the linearization technique for the list.

Dolev et al. [DT13] give a self-stabilizing algorithm for spanders. A spander is a subgraph of an expander graph $G$ such that it is an expander using a subset of the edges of $G$. The key components of the protocol is an algorithm that chooses edges that should be in the expander, a monitoring technique that ensures the desired result and a self-stabilizing reset algorithm, which is used by the monitor.

Another direction of investigated topologies are tree-like structures: i.e., spanning trees [Hér+06; Clé+08], hypertrees [DK08] and a variant of radix trees called double-headed radix trees [AW07]. The main differences between the three topologies is the in-degree and out-degree of nodes in the final topology, the length of the longest path, and the construction time of the topology. More specifically, hypertrees have a fixed $\log_b(n)$ bound on the in- and out-degree (where $b$ is a parameter of the network) and also a longest path of length $\log_b(n)$, whereas the spanning trees also have a bound on the degree, but no bound on the path length. Double-headed radix trees are balanced search trees which support search, predecessor and successor operations in time proportional to the length of identifiers of the nodes. Conceptually, the tree can be thought of as a radix trees in which the leaves have been removed (i.e., their keys are propagated to some ancestor) and the root has been split into two (the "double head").

In contrast to these very specific topologies, Berns et al. [BGP13] introduced the general *transitive closure framework*, which is able to build a broad variety of overlay networks. One key idea of the framework is to build a supertopology of the end topology (e.g., the clique) and then remove the unnecessary edges. This protocol suffers from being very space inefficient:, i.e., during execution node degrees can grow to $\Omega(n)$ even though the target topology has a constant degree.

**Safety Property Maintenance** In the last 20 years many approaches have been investigated that focus on maintaining safety properties during the convergence phase of self-stabilization. The most prominent examples from the related literature are snap-stabilization [Bui+07; Del+10], safe convergence [KK07],

super-stabilization [DH97] and self-stabilization with service guarantee [JM14].

A protocol is *snap-stabilizing* if it always behaves according to its specification independent of its initial configuration. Snap-stabilization has a user-centric safety property (whereas the other approaches are system-centric): i.e., it ensures that the answer to a properly initiated user request by the protocol is correct. *Safe convergence* ensures that the system quickly converges from an initial state to a safe state (i.e., a configuration in which a predefined safety property is fulfilled). Afterwards it quickly converges to a legitimate state while maintaining the safety property during the stabilization. However, external disruptions are not handled in safe convergence. Self-stabilization with *service guarantee* fixes this drawback: i.e., such a protocol quickly converges to a safe state (in their words "the protocol quickly provides a minimal service"), and it maintains the safety properties during stabilization despite the occurrence of some disruptions. A *super-stabilizing* protocol guarantees that (i) starting from a legitimate configuration, a safety property is preserved after only one specific topology change, and (ii) the safety property is maintained during recovering to a legitimate configuration assuming that no more topology change occurs during the stabilization phase.

Close to our notion of monotonic searchability is the work by Yamauchi and Tixeuil [YT10] in the area of *monotonic convergence*. A self-stabilizing protocol is monotonically converging if every change done by a node $p$ makes the system approach a legitimate state and if every node changes its output only once. Consequently, the system makes monotonic progress towards a legitimate state. The authors investigate many monotonically converging protocols for different classical distributed problems (e.g., leader election and vertex coloring). However, their approach is more complexity analysis based than ours: i.e., they focus on the amount of non-local information that is needed to solve the above mentioned problems in a monotonic way.

Note that none of the concepts for safety property maintenance specifically investigated topological self-stabilization. However, in general the basic ideas and design principles could be used for protocols that stabilize topologies.

# CHAPTER 8

## Problem Statement, Preliminaries & Primitives

> ”*Everything should be made as simple as possible, but not simpler.*”
>
> Albert Einstein; Theoretical Physicist

In this chapter we first want to create a solid basis of understanding for the problem of monotonic searchability by introducing the problem statement and some preliminary results. These results impose some restrictions concerning the problem itself: i.e., in general, monotonic searchability is impossible to achieve. However, we can deduce some prerequisites for initial states that are necessary in order to achieve our desired goal and that provide a starting point for our results of Chapter 9. Moreover, we present some general results concerning the construction of overlay networks. We introduce four simple atomic actions (called primitives) and show that they are useful for maintaining an overlay. These results are independent of the idea of monotonic searchability. However, they provide a solid basis of understanding and can be taken advantage of in the upcoming Chapter 9.

**Chapter Outline** In Section 8.1 we formally define the problem statement for monotonic searchability in self-stabilizing overlay networks. Subsequently, in

Section 8.2 we introduce some preliminary results for monotonic searchability. Finally, in Section 8.3, we present our analysis of the aforementioned primitives.

**Chapter Basis** Section 8.1 and 8.2 are based on the following publication:

> **2015** (with C. Scheideler and A. Setzer). "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, cf. [SSS15].

The majority of Section 8.3 is based on the following journal article:

> **2016** (with A. Koutsopoulos and C. Scheideler). "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Information and Computation*, cf. [KSS16].

The very last theorem is based on:

> **2016** (with C. Scheideler and A. Setzer). "Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, cf. [SSS16].

## 8.1. Problem Statement

First, let us formally define what it means for a protocol to be self-stabilizing, and, moreover, what it means if a self-stabilizing protocol satisfies monotonic searchability.

A protocol is *self-stabilizing* if it satisfies the following two properties.

***Convergence:*** Starting from an *arbitrary system state*, the protocol is guaranteed to arrive at a *legitimate state*.

***Closure:*** Starting from a legitimate state the protocol remains in legitimate states thereafter.

The definition of a legitimate state is highly dependent on the goal of the protocol. In *topological self-stabilization* we allow self-stabilizing protocols to

perform changes to the overlay network. Thus, a legitimate state may then include a particular *graph topology* or a family of graph topologies. We say that a self-stabilizing protocol *stabilizes* to a certain graph topology, if in every legitimate state $G_E$ is a graph of the chosen topology . By definition a self-stabilizing protocol is able to recover from transient faults regardless of their nature. Moreover, a self-stabilizing protocol does not have to be initialized as it eventually starts to behave correctly regardless of its initial state.

One goal of this thesis is to design a self-stabilizing protocol for the *line graph* topology. That is, the nodes are sorted by identifier and each node has an edge to the two nodes with the closest preceding and succeeding identifier. We investigate two different scenarios. We first focus on the case in which each node stores only two references in a legitimate system state: its closest successor and its closest predecessor. Thereafter, we weaken this restriction and allow nodes to have multiple successor and predecessor references. However, the closest successor and the closest predecessor have to be among them, i.e., the resulting topology is a supergraph of the line. To clearly distinguish between the two topologies, we use the term *strict line* to refer to the first case and *super-line* for the second.

One major advantage of the line topology is that searching for identifiers is easy once a legitimate state has been reached. However, searching reliably while stabilization is still in progress is much more involved. We say a self-stabilizing protocol *satisfies monotonic searchability* according to some routing protocol $R$ if it fulfills the following two properties:

***Monotonicity:*** For any pair of nodes $v, w$ it holds that once a $\textsc{Search}(v, id(w))$ request initiated in state $s$ succeeds, any $\textsc{Search}(v, id(w))$ request initiated in a state $s' > s$ succeeds.

***Non-Triviality:*** In every computation of the protocol there is a suffix such that for each pair of nodes $v, w$ for which there is a path from $v$ to $w$ in the target topology $\textsc{Search}(v, id(w))$ requests succeed.

We do not mention $R$ if it is clear from the context.

## 8.2. Restrictions and Preliminary Results

We now introduce some restrictions from the related literature that are necessary for topological self-stabilization. Moreover, we present some general statements concerning monotonic searchability which are independent of the topology. These statements result in general restrictions concerning monotonic searchability.

The following propositions are restatements of results of [NNS13] and imply necessary conditions on initial system states.

**Initial Weak Connectivity:** If a self-stabilizing (compare-store-send) protocol stabilizes to a given connected graph topology, each computation starts in a weakly connected initial state.

**Validity of Identifiers:** If a self-stabilizing (compare-store-send) protocol stabilizes to a given connected graph topology, each computation starts in a state in which all identifiers stored in messages or on local memories of nodes belong to existing nodes.

Furthermore, we restrict the initial state to contain only a finite number of messages that can trigger actions specified by our protocol. An infinite number of messages could easily prevent the convergence of any self-stabilizing protocol. From now on, an initial system state satisfies all of these constraints.

A *message invariant* is a property of the following form: If there is a message $m$ in the incoming channel of a node, then a predicate $P$ must hold. A protocol may specify one or more message invariants. A message $m$ in some system state is called *corrupt* if its existence violates one or more message invariants. A state $s$ is called *admissible* if there are no corrupt messages in $s$. We say a protocol *admissibly satisfies* monotonic searchability if the following two conditions hold:

(a) In all computation suffixes of the protocol that start from admissible states monotonic searchability is satisfied.

(b) Every computation of the protocol contains at least one admissible state.

Otherwise, we say that the protocol *unconditionally satisfies* monotonic searchability.

With this notion, we can show that admissible satisfiability is necessary for monotonic searchability for any routing algorithm $R$.

**Lemma 8.1.** *If a (compare-store-send) self-stabilizing protocol satisfies monotonic searchability then this protocol cannot be unconditionally satisfying.*

*Proof.* Assume there is a compare-store-send self-stabilizing protocol $\mathcal{P}$ that unconditionally satisfies monotonic searchability. First of all, note that if $\mathcal{P}$ violates only the second condition of admissible satisfiability, then by definition computations exist in which monotonic searchability is never satisfied, implying that $\mathcal{P}$ cannot satisfy monotonic searchability in the first place. Thus, assume that the first condition is violated: i.e., the protocol satisfies monotonic searchability in computations with arbitrary messages, regardless of any invariants. Consider the network graph given in Figure 8.1: i.e., node $u$ and $v$ are connected by an explicit edge and there is an implicit edge $(v, w)$ (a message containing the reference of $w$ in $v.Ch$).

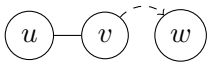



Figure 8.1.: Graph Instance for the Proof of Lemma 8.1

We carry out the proof as a game between the protocol and an adversary: On the basis of the decisions of the protocol, the adversary can decide on the delivery speed of messages, and may insert additional messages at each node. The latter is possible since nodes can not distinguish between these messages and messages that already existed in an initial state, which have not been received yet. Furthermore, the adversary may set the internal initial state of the nodes.

At first, we issue a $search(u, id(w))$ request in $u$ that we denote by $a$. We argue that the adversary can force $u$ to forward $a$ to $v$. Therefore note the following:

(a) As long as $u$ does not receive any further messages, $u$ does not know any other node, so $v$ is the only possible next hop for $a$.

(b) If $u$ tries to wait for a fixed amount of time before sending $a$, the adversary simply halts the system for that time: i.e., no messages are delivered in that time frame and the system state stays the same.

(c) If $u$ requires the receipt of another message in order to forward $a$, the adversary makes sure that this message is never received, which contradicts the assumption that monotonic searchability is satisfied. Therefore, $u$ does not rely on another message to forward $a$.

(d) If $u$ relies on its internal state to forward $a$, the adversary changes the initial internal state of $u$ such that it never forwards any message, which again contradicts the assumption that monotonic searchability is satisfied. Hence, $u$ does not rely on its state to forward $a$.

(e) There are no other conditions that $u$ can wait on.

Consequently, $u$ sends $a$ to $v$ eventually. At the point in time we issue a second $search(u, id(w))$ request in $u$, which we denote by $b$. For similar reasons as stated above, $b$ is eventually sent to $v$ as well.

Since both messages are now in $v.Ch$ and the adversary is allowed to decide on message speeds, it forces $v$ to receive $b$ first. Node $v$ has no explicit edge to $u$. Moreover, the adversary can enforce that the implicit edge $(v, w)$ is not received by $v$ until $v$ handles $b$. Therefore, $b$ must be answered with a failure at some point in time (since the $b$ cannot be forwarded anymore) and $u$ is informed about that.

Next, the adversary causes the implicit edge $(v, w)$ to arrive at $v$. Since the protocol must stabilize to the line topology, at some point in time, the edge $(v, w)$ is established. Until then, the adversary withholds message $a$ in $v.Ch$. Afterwards, when $a$ arrives at $v$, it can be forwarded to $w$ and the search request succeeds.

As a consequence $a$ succeeds, whereas $b$, which was initiated after $a$, fails. This is a contradiction to the assumption that the protocol satisfies monotonic searchability. □

Consequently, to prove that a protocol satisfies monotonic searchability we have to define invariants, which capture the validity of messages for searchability. Afterwards it is sufficient to show that: (i) the protocol guarantees monotonic searchability according to $R$ in admissible states and (ii) the protocol has a computation suffix in which every state is admissible

We want to conclude this section by showing that the original self-stabilizing protocol for the line topology [ORS07; Gal+14] cannot satisfy monotonic

searchability. We refer to the original protocol with the term Build-Line. In the following, we briefly sketch the protocol itself and present the corresponding pseudocode. The notation we use is not exactly congruent with the original literature, simply due to differences in the model. However, one can easily verify that the protocols do not differ in their essence.

In Build-Line, every node only maintains a single $left$ and $right$ neighbor. The protocol consists of two actions: a Timeout action and a Linearize() action. In its Timeout action a node periodically sends its own reference in a Linearize() message to its neighbors (see Algorithm 12). The Linearize() action is triggered by a Linearize() message which contains the reference of a single node. Once a node $u$ receives a Linearize($v$) message that contains the reference of a node $v$ with $id(u) < id(v)$ ($id(u) > id(v)$, respectively), $u$ (i) either saves $v$ as its new right (left) neighbor if $v$ is closer to $u$ than the current right (left) neighbor $w$ and delegates the reference of $w$ to $v$ or (ii), in case $v$ is not closer than $w$, $v$ is not saved and delegated to $w$ (see Algorithm 13). Here, *delegation* means that the reference of $v$ is sent in a Linearize($v$) message to $w$ and *not* kept in the local memory of $u$.

To enhance the readability, the pseudocode omits the cases in which a neighbor does not have a left or right neighbor. Note that this protocol is easily extendable to the case that a node might have more than just two references in an initial state. In its Timeout action a node simply has to check which two references are its closest left and right neighbors and linearize the other references accordingly.

A natural routing protocol for search requests for this topology is to always forward search requests to the neighbor in direction of the desired target ID, or to abort the search request in case no such neighbor exists.

We now show the following lemma.

**Lemma 8.2.** Build-Line *cannot satisfy monotonic searchability.*

*Proof.* Consider the topology given in Figure 8.2. Node $u$ has an explicit edge to node $w$ and an implicit edge to node $v$, i.e., a Linearize($v$) message in $u.Ch$. In this system state $s$ node $u$ can successfully search for node $w$, since there is a direct connection.

Now consider the state $s'$ in which $u$ receives the Linearize($v$) message. According to the Linearize() action, $u$ sets $v$ as its new right neighbor and

**Algorithm 12** TIMEOUT action of BUILD-LINE

```
1: if id(left) < id(self) then
2:     send LINEARIZE(self) to left
3: else
4:     send LINEARIZE(left) to self
5:     left ← ⊥
6: if id(right) > id(self) then
7:     send LINEARIZE(self) to right
8: else
9:     send LINEARIZE(right) to self
10:    right ← ⊥
```

**Algorithm 13** LINEARIZE($v$) action of BUILD-LINE

```
1: if id(v) < id(left) then
2:     send LINEARIZE(v) to left
3: if id(left) < id(v) < id(self) then
4:     send LINEARIZE(left) to v
5:     left ← v
6: if id(self) < id(v) < id(right) then
7:     send LINEARIZE(right) to v
8:     right ← v
9: if id(right) < id(v) then
10:    send LINEARIZE(v) to right
```

sends a LINEARIZE($w$) message to $v$. Similarly to the proof of Lemma 8.1 a SEARCH($u, id(w)$) request that is initiated in $s'$ is bound to fail, since there is no explicit path between $u$ and $w$ anymore. This proves the statement.

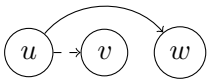



Figure 8.2.: Graph Instance for the Proof of Lemma 8.2

□

Note that Lemma 8.2 focuses on the line, since this part of the thesis specifically considers monotonic searchability for that topology. However, the general idea of the proof does in fact generalize to most self-stabilizing protocols that stabilize to a certain topology. The main problem is the use of

the aforementioned delegation, i.e., the sending a reference to another node without keeping it in the local memory. This delegation, which is one of the major obstacles for searchability, is more concretely defined in the upcoming section.

## 8.3. Primitives for Overlay Networks

An important property for any protocol that manages the topology of an overlay network is the fact that weak connectivity is never lost by its own actions. Therefore, it is highly desirable that every node only executes actions that preserve weak connectivity. In this section we introduce four *primitives*, i.e., simple atomic actions for manipulating edges in an overlay network that are safe in the sense that they preserve weak connectivity as long as there is no fault. This implies that *any* distributed protocol whose actions can be decomposed into these four primitives is guaranteed to preserve weak connectivity. Throughout this section we not only prove that the primitives preserve weak connectivity, but also that they are also *universal*: i.e., by using the primitives only we can in principle transform any weakly connected graph into any other weakly connected graph. We conclude this section by showing that we can even go beyond this abstract notion of universality and show that almost any existing (self-stabilizing) overlay protocol can be easily transformed to use the primitives only. It needs to be noted that the results from Lemma 8.3 to Lemma 8.6 have also been introduced in the PhD thesis of Andreas Koutsopoulos [Kou16]. However, both theses do not put the primitives and their properties in the focus of their investigation. Instead, they are used as a tool to show that overlay protocols maintain weak connectivity. Therefore, the results can be seen as supplementary material, to get a better understanding of the power of the primitives. Theorem 8.7 has not been part of the investigations in [Kou16].

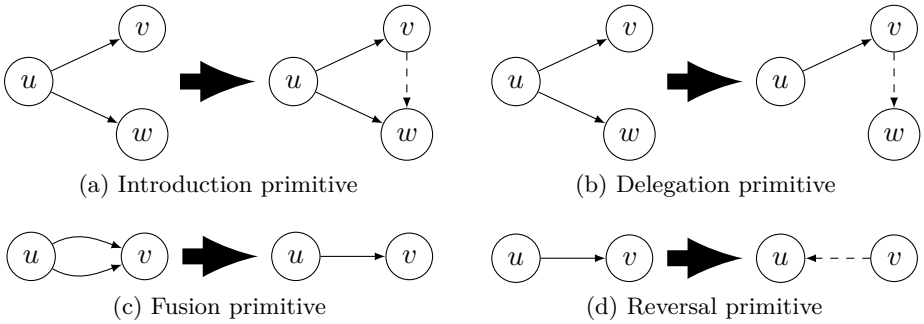
The four primitives are depicted in Figure 8.3. We define them in the following:

**Introduction** If a node $u$ has references of two nodes $v$ and $w$ with $v \neq w$, $u$ *introduces* $w$ to $v$ if $u$ sends a message to $v$ containing a reference of $w$ while keeping the reference.

**Delegation** If a node $u$ has a reference of two nodes $v$ and $w$, then $u$ *delegates* $w$'s reference to $v$ if $u$ sends a message to $v$ containing the reference of $w$ while deleting the reference of $w$ from the local memory.

**Fusion** If a node $u$ has two references of nodes $v$ and $w$ with $v = w$, then $u$ *fuses* the two references if it only keeps one of these references.

**Reversal** If a node $u$ has a reference of some other node $v$, then $u$ *reverses* the edge if it sends a reference of itself to $v$ while deleting its reference of $v$ from the local memory.




(a) Introduction primitive (b) Delegation primitive

(c) Fusion primitive (d) Reversal primitive

Figure 8.3.: The Four Primitives

Note that we assume that $u, v, w$ are pairwise distinct. The only exceptions are Fusion and *Self-Introduction*, a special case of the Introduction primitive in which $u$ sends a reference of itself to $v$ while keeping its reference to $v$. The four primitives have the advantage that they can be executed locally by every node in a wait-free fashion (as none of the primitives requires any feedback). Also, they just require the ability of nodes to check whether two references point to the same node (see Fusion) to be implementable. Other than that, access to the contents of the references is not needed: i.e., the primitives do not require numerical identifiers. Most importantly, all four primitives maintain weak connectivity, as we show in the following Lemma.

**Lemma 8.3.** *Introduction, Delegation, Fusion, and Reversal preserve weak connectivity.*

*Proof.* The statement obviously holds for Introduction since it only adds additional edges to $G$. In Delegation an edge $(u, w)$ is deleted, but there still

exists a path from $u$ to $w$ via $v$, so $u$ and $w$ are still in the same weakly connected component. Fusion deletes an edge only if it is superfluous in terms of connectivity. The Reversal rule deletes an edge $(u, v)$ but replaces it with an edge $(v, u)$, thereby preserving weak connectivity as well. □

Let $\mathcal{P}$ denote the set of all distributed protocols, such that all interactions between nodes can be decomposed into the four primitives. Lemma 8.3 implies that any protocol in $\mathcal{P}$ preserves weak connectivity, which was previously shown individually for each cited protocol of Section 7.2. Note that the first three primitives even preserve strong connectivity: i.e., if a protocol is restricted to use the first three primitives only, for any pair of nodes $u, v$ with a directed path in $G$ there is always a directed path from $u$ to $v$ in $G$. To the best of our knowledge, all self-stabilizing overlay protocols proposed so far (e.g., the line [SR05; ORS07; Gal+14], the Delaunay graph [Jac+12], etc.) are in $\mathcal{P}$. We say that a set of primitives is *universal* if the primitives allow one to get from any weakly connected graph $G = (V, E)$ to any other weakly connected graph $G' = (V, E')$. The set is *weakly universal* if $G'$ is strongly connected.

**Theorem 8.4.** *Introduction, Delegation, Fusion, and Reversal are universal.*

*Proof.* In order to prove the theorem, we give a general strategy how to transform an arbitrary weakly connected graph $G = (V, E)$ into any other weakly connected graph $G' = (V, E')$. At first, note that if every node continuously introduces all neighbors to each other, including self-introduction, then the topology of $G$ is eventually transformed into a clique.

Let $G'' = (V, E'')$ be the bidirected expansion of $G'$: i.e., for each edge $(u, v) \in E'$ there are edges $(u, v), (v, u) \in E''$. Next we show that by using Delegation and Fusion, one can transform the clique to $G''$. To do so, we make use of the fact that $G''$ is strongly connected by construction. Consider an arbitrary edge $(u, w)$ of the clique that is not in $E''$. Since $G''$ is strongly connected, there exists a shortest path from $u$ to $w$ in $G''$ that we maintain as we first want to keep all edges in $G''$. Let $(u, v_1, v_2, ..., v_k, w)$ be this path: i.e., $v_1$ is the first node on the path and adjacent to $u$ and $v_k$ is the last node on the path and therefore adjacent to $w$. In order to get rid of the edge $(u, w)$, $u$ uses the Delegation primitive and delegates the reference of $w$ to $v_1$. Now node $v_1$ (and all other nodes on the path) proceed similar to $u$ by forwarding

the reference of $w$ along the path up to the last node $v_k$. Node $v_k$ finally uses Fusion to merge the edge with the already existing edge $(v_k, w)$. By applying this procedure to all edges not in $E''$, all that remains is $G''$.

At last we can use Reversal and Fusion to transform $G''$ to $G'$. To do so, every edge $(u, v)$ that is in $E''$, but not in $E'$ is reversed by $u$. The newly created edge $(v, u)$ is fused with the already existing edge $(v, u) \in E'$. □

The following corollary follows directly from our proof of Theorem 8.4.

**Corollary 8.5.** *Introduction, Delegation and Fusion are weakly universal.*

Note that the results of Theorem 8.4 and Corollary 8.5 are not constructive, since we only show that *in principle* it is possible to get from any weakly connected graph topology to any other weakly connected graph topology. At the end of this section, we show how to give a more constructive result concerning the universality of the primitives. Furthermore, we can show that Introduction, Delegation, Fusion and Reversal are not only sufficient for universality but also necessary, in a sense that any proper subset of the primitives is not universal.

**Lemma 8.6.** *Any proper subset of the primitives Introduction, Delegation, Fusion and Reversal is not universal.*

*Proof.* To prove the statement, we show that each primitive has a unique function that cannot be replaced by the other primitives. Again let $G = (V, E)$ be the weakly connected graph that we want to transform into another weakly connected graph $G' = (V, E')$.

**Introduction:** It is the only primitive that can create new edges: i.e., any Graph $G'$ with $|E'| > |E|$ cannot be reached from $G$ without it.

**Fusion:** It is the only primitive that reduces the overall number of edges: i.e., any Graph $G'$ with $|E'| < |E|$ cannot be reached from $G$ without it.

**Delegation:** Consider two nodes that are connected by an edge in $G$ but not in $G'$. By using Introduction, Fusion and Reversal only a protocol can disconnect these two nodes.

**Fusion:** Consider that $G$ consists of only two nodes $u$ and $v$ and an edge $(u, v)$. Reversal is necessary to reach the goal topology $G'$ that consists solely of the edge $(v, u)$. □

We conclude the results of this chapter by strengthening our result concerning weak universality by showing that we can decompose an existing protocol for a strongly connected overlay topology into a new protocol that uses our primitives only.

**Theorem 8.7.** *Any compare-store-send protocol that stabilizes to a strongly-connected topology and preserves weak connectivity can be transformed such that all interactions between nodes can be decomposed into the primitives Introduction, Delegation and Fusion.*

We say that a node $u$ *deletes* a reference of another node $v$ if there exists an explicit non-multi-edge $(u, v)$ and $u$ executes an action such that $u$ removes the reference from its local memory without sending it to another node. Before we are able to prove Theorem 8.7, we need to show the following lemma.

**Lemma 8.8.** *Any compare-store-send protocol which stabilizes to a strongly-connected topology and contains an action such that a node $u$ deletes a reference cannot preserve weak connectivity*

*Proof.* Assume for contradiction that the protocol preserves weak connectivity and there is an action in which $u$ deletes a reference. Consider the left graph depicted in Figure 8.4 and assume that $u$ aims at deleting the edge $(u, v)$. Obviously, $u$ can delete the reference of $v$ without disconnecting the graph. However, since the protocol presumably preserves weak connectivity, $u$ may not delete the edge immediately, but could perform other actions. However, since the graph is still connected without $(u, v)$, $u$ eventually decides that it can delete $(u, v)$ safely and executes an action that ultimately deletes the reference of $v$. Let $s_u$ be the *internal state* of $u$ before it deletes the edge, i.e., the values of all variables of $u$ and messages in $u.Ch$. We construct a new system state by taking the graph depicted on the right in Figure 8.4 and setting the internal state of $u$ to $s_u$. Naturally, this is a valid initial state for a computation. Since the internal state of $u$ has not changed, it still makes the decision to delete its reference of $v$. This disconnects $u$ and $v$, which is a contradiction to the assumption that the protocol preserves weak connectivity. □


Figure 8.4.: Graph Instances for the Proof of Lemma 8.8.

Now we can prove Theorem 8.7.

*Proof of Theorem 8.7.* Let $A$ be a compare-store-send protocol that stabilizes to a strongly-connected topology and preserves weak connectivity. Moreover, let $u$ be a node that acts according to $A$. Since $A$ is a compare-store-send protocol we focus only on the actions of $A$ that specifically handle references of nodes.

At first consider all actions of $A$ that are executed because a local predicate becomes true: i.e., $u$ does not receive any message. Node $u$ has three options: (i) it does not interact with its references, (ii) it sends one or multiple references to one or multiple nodes, or (iii) it deletes one or multiple references. In case (i) there is no interaction between nodes. In case (ii) $u$ sends references and either keeps them in its memory or does not keep them. The first subcase can be transformed such that the Introduction primitive is used. For the second subcase we can use the Delegation primitive. Case (iii) is not allowed, due to Lemma 8.8.

Next consider all actions of $A$ that are triggered by a message that contains at least one reference: i.e., the message is received by $u$ and has to be processed. Node $u$ has multiple options for each reference in the received message: (i) keep the reference, and/or (ii) send the reference to one or multiple nodes, or (iii) delete the reference (i.e., neither save the reference nor send it to another node). Case (i) is either the Fusion primitive (in case $u$ already has a reference of that node) or does not need to be handled by a primitive, since keeping an edge replaces an implicit edge with an explicit edge. Case (ii) is again Introduction or Delegation, depending on whether $u$ keeps the reference. As before case (iii) is not allowed. Moreover, $u$ has the aforementioned option of sending one or multiple references of nodes that are not in the message to one or multiple nodes.

Finally, consider all actions of $A$ that are triggered by a message that does

not contain any references. This case basically reduces to the case in which a predicate triggers action execution because node $u$ has the same options for interaction with its references. □

This concludes this chapter and we now turn to the main topic of this part of the thesis: *monotonic searchability*.

# CHAPTER 9

# Monotonic Searchability

> ”*For there is nothing lost, that may be found, if sought.*”
>
> Edmund Spenser; Poet

ELABORATING on the topic of monotonic searchability (as introduced in the previous chapters), this chapter investigates concrete protocols that maintain monotonic searchability for the strict line and the super-line. The line topology was chosen deliberately as a starting point to investigate monotonic searchability, since it yields a straightforward routing protocol for searching. Moreover, a line is not only one of the simplest imaginable topologies, but has also proven a useful tool in the context of topological self-stabilization. Many of the results of the related literature (see Section 7.2) either use a self-stabilizing line as an underlying principle (e.g., the De-Bruijn graph and skip graphs) or are heavily inspired by the ideas of the self-stabilizing line (e.g., Delaunay graphs and small-world graphs).

The self-stabilizing strict line (Section 9.1) was our earliest result in the area of monotonic searchability. While we achieved our goal to design a protocol that maintains monotonic searchability for the line topology, the protocol itself

was suffering from (i) a lot of overhead (since it cannot use the Delegation primitive) and (ii) a slow and inefficient routing protocol. These drawbacks were the main driving factors for investigating the super-line topology (see Chapter 8 for the concrete problem statement), since we can circumvent both of these issues. With both protocols established it is the most natural question to ask how they compare to each other. We opted for a practical approach by implementing both protocols and comparing them in simulations. This is due to the fact that the asynchronous message passing model that we utilize in this thesis is not tailored for an in-depth theoretical efficiency comparison.

**Chapter Outline** In Section 9.1 we investigate a solution for monotonic searchability in the scenario where the topology is the strict line. Section 9.2 focuses on the super-line topology. We conclude this chapter in Section 9.3 by comparing the performance of both protocols in simulations.

**Chapter Basis** The results of Section 9.1 are based on the following publication:

> **2015** (with C. Scheideler and A. Setzer). "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, cf. [SSS15].

All results of Section 9.2 and Section 9.3 are unpublished. The basic algorithmic ideas of the super-line topology were first partially discussed in the lecture "Distributed Algorithms and Datastructures" by my supervisor Prof. Dr. Christian Scheideler. The simulations for the analysis in Section 9.3 were conducted by my student assistant Linghui Luo.

## 9.1. Monotonic Searchability for the Line Topology

In this section, we present the Build-Line+ protocol and the Search+ protocol. Build-Line+ stabilizes to the strict line topology and admissibly satisfies monotonic searchability according to Search+. This section is organized as follows: First, we describe Build-Line+ and Search+ in detail (Subsection 9.1.1). Then, we prove that the Build-Line+ protocol

stabilizes to the line topology (Subsection 9.1.2). Last, we show that the Build-Line+ protocol satisfies monotonic searchability according to Search+ (Subsection 9.1.3). For the remainder of this section, we drop the "according to Search+" clause, since we only consider searchability for Search+.

### 9.1.1. Description of Build-Line+ and Search+

Our solution to achieve monotonic searchability for the strict line builds upon the original solution for the line topology as introduced in [ORS07] (see Section 8.2 for the full description). As we have shown, this easy and elegant protocol cannot guarantee monotonic searchability. Throughout, this subsection we first introduce the Build-Line+ protocol that builds the topology and then introduce the Search+ protocol for searching. Note that each line of pseudocode that explicitly handles references of nodes is labeled with either $\mathfrak{I}, \mathfrak{D}, \mathfrak{F}, \mathfrak{R}$ or $\mathfrak{S}$. The first four letters refer to the four primitives for overlay networks of Section 8.3 (i.e., Introduction, Delegation, Fusion and Reversal), $\mathfrak{S}$ is meant as an abbreviation for *Storing* a reference. We use these annotations later to prove that Build-Line+ preserves weak connectivity.

The Build-Line+ protocol introduces the following changes to the original protocol [ORS07] in order to satisfy monotonic searchability: Instead of having a single left and right neighbor, a node $u$ has sets of neighbors $Left$ and $Right$ (which it sorts according to ID). In the following, we use the notation $Left(u)/Right(u)$ to refer to these sets of neighbors. The main design principle is that a node $u$ never delegates a reference (i.e., an edge to a node $v$) stored in $Left(u)$ or $Right(u)$ directly to another node $w$. Instead it first introduces this node to $w$ with an Introduce$(v, u)$ message (see Algorithm 15). It waits for an acknowledgement that the reference has been added to $Left(w)$ or $Right(w)$ (i.e., a Linearize$(v)$ message – see Algorithm 16), and afterwards delegates $v$ using a TempDelegate$(v)$ message (see Algorithm 17). More specifically, whenever a node $u$ has multiple neighbors to one side, it does not delegate edges to the closest neighbor directly, but does the following. W.l.o.g. assume that $u$ has multiple neighbors $v_1, \ldots, v_\ell$ to the right with $id(v_i) < id(v_{i+1})$. In the Timeout action (see Part 1 of Algorithm 14) $u$ introduces $v_i$ to $v_{i-1}$, with an Introduce$(v_i, u)$ message. Thereby, $v_{i-1}$ knows that it received the message from $u$. Node $v_{i-1}$ saves the reference to $v_i$ in its local memory, sends

a LINEARIZE($v_i$) message back to $u$ and a TEMPDELEGATE($u$) to itself (the latter is only necessary to preserve weak connectivity). Node $u$ can now react to that LINEARIZE($v_i$) message, by removing $v_i$ from its memory and sending the reference to the closest node to the left of $v_i$ in *Right* (which is not necessarily $v_{i-1}$ anymore). Thereby, $u$ preserves a path of explicit edges between $u$ and $v_i$. Additionally, $u$ sends its own reference to the closest neighbors with an INTRODUCE($u, \perp$) message in its TIMEOUT action.

In general, the TEMPDELEGATE($u$) action is used to delegate an implicit edge as long as it is not made explicit. Note that implicit edges are not used for searching, thus we do not have to apply the principle of introducing first and delegating afterwards for this kind of edges. However, we have to delegate them properly in order to preserve weak connectivity and to stabilize to the line. Note that a node temporarily stores more references than necessary for the final line. However, our protocol still eventually stabilizes to the strict line, as we show later. Throughout the pseudocode, we use the expression $self$ whenever a node refers to itself. Additionally, keep in mind that the timeout action is the only action that is not triggered as a result of another action, but is executed regularly.

**Algorithm 14** BUILD-LINE+: TIMEOUT

**Part 1: Self-Stabilization of Topology**

▷ Let $Left = \{v_1, v_2, ..., v_k\}$ with $id(v_1) < id(v_2) < \cdots < id(v_k)$

1: **for all** $v_i \in Left$ with $1 \leq i < k$ **do**
2: send INTRODUCE($v_i, self$) to $v_{i+1}$ ▷ $\mathfrak{I}$

▷ Let $Right = \{w_1, w_2, ..., w_l\}$ with $id(w_1) < id(w_2) < \cdots < id(w_l)$

3: **for all** $w_i \in Right$ with $1 < i \leq l$ **do**
4: send INTRODUCE($w_i, self$) to $w_{i-1}$ ▷ $\mathfrak{I}$
5: send INTRODUCE($self, \perp$) to $v_1$ ▷ $\mathfrak{I}$
6: send INTRODUCE($self, \perp$) to $w_1$ ▷ $\mathfrak{I}$

**Part 2: Monotonic Searchability**

7: **for all** $destID \in Waiting$ **do** ▷ Regularly send out probes
8: send $forwardProbe(self, destID, \{self\}, self.seq)$ to $self$ ▷ $\mathfrak{I}$

The SEARCH+ protocol works in the following way: Whenever a node $u$ wants to initiate a new search request, the INITIATENEWSEARCH($destID$) action of $u$ is called (see Algorithm 18). In this action, node $u$ creates and stores a new SEARCH($u, destID$) message and starts to periodically initi-

**Algorithm 15** BUILD-LINE+: INTRODUCE$(v, w)$

```
1: if id(v) < id(self) then
2:     if w ≠ ⊥ then
3:         Left ← Left ∪ {v}                                   ▷ 𝔖 / 𝔉
4:         send LINEARIZE(v) to w                              ▷ 𝔇
5:         send TEMPDELEGATE(w) to self                        ▷ 𝔇
6:     else                                                    ▷ w = ⊥
7:         send TEMPDELEGATE(v) to self                        ▷ 𝔇
8: else if id(v) > id(self) then          ▷ Analogous to the previous case.
```

**Algorithm 16** BUILD-LINE+: LINEARIZE$(v)$

```
1: send TEMPDELEGATE(v) to self                                 ▷ 𝔇
2: if id(v) < id(self) then
3:     if Left ≠ ∅ then
4:         x ← argmax{id(x')|x' ∈ Left}
5:         if v ≠ x then
6:             w ← argmin{id(w')|w' ∈ Left and id(w') > id(v)}
7:             Left ← Left\{v}
8:             send TEMPDELEGATE(v) to w                        ▷ 𝔇
9: else if id(v) > id(self) then          ▷ Analogous to the previous case.
```

ate FORWARDPROBE$(u, destID, \{u\}, self.seq)$ messages that it sends to itself (see Part 2 of Algorithm 14)). In the following, assume $id(u) < destID$ (the other case is analogous). Each FORWARDPROBE() message has a set of nodes $Next$ attached to it, which contains nodes that the message visits in the future. It also stores a sequence number counter $seq$, whose meaning we will explain later. Whenever a FORWARDPROBE$(u, destID, Next, seq)$ message is received by a node $v$ (see Algorithm 19), $v$ removes itself from $Next$ and adds all its right neighbors $x$ with $id(x) \leq destID$ to $Next$. Then the FORWARDPROBE$(u, destID, Next, seq)$ message is forwarded to the node with minimal ID in $Next$. If a FORWARDPROBE$(u, destID, Next, seq)$ message arrives at a node $v$ with $id(v) = destID$, it directly responds with a PROBESUCCESS$(destID, seq, v)$ message to $u$. If $Next$ is empty at a node $v$ with $id(v) \neq destID$ (after $v$ has added the aforementioned right neighbors), the FORWARDPROBE() message cannot reach its target and is answered with a PROBEFAIL$(destID, seq)$ message. In any case, as soon as $u$ receives a response, it acts accordingly. If a FORWARDPROBE$(u, destID, Next, seq)$ mes-

**Algorithm 17** BUILD-LINE+: TEMPDELEGATE($v$)

```
1: if id(v) < id(self) then
2:     if Left = ∅ then
3:         Left ← Left ∪ {v}                                   ▷ 𝔊
4:     else                                                    ▷ Left ≠ ∅
5:         x ← argmax{id(x′)|x′ ∈ Left}
6:         if id(x) < id(v) then
7:             Left ← Left ∪ {v}                               ▷ 𝔊
8:         else if id(x) > id(v) then
9:             TEMPDELEGATE(v) to x                            ▷ 𝔇
10: else if id(v) > id(self) then        ▷ Analogous to the previous case.
```

sage is answered by a PROBEFAIL($destID, seq$) message (see Algorithm 21), it drops the corresponding SEARCH($u, destID$) message completely. Otherwise (i.e., the probe is answered by a PROBESUCCESS($destID, v$) message), the SEARCH($u, destID$) message waiting at $u$ are directly sent to $v$ (see Algorithm 20).

Whenever additional SEARCH($u, destID$) messages are created at $u$ while $u$ is still waiting for an answer to an earlier initiated FORWARDPROBE($u, destID$), these messages simply wait together with the previous request (realized by simple $WaitingFor[destID]$ field) and are aborted or dispatched together as soon as the PROBEFAIL($destID$) or PROBESUCCESS($destID, v$) response arrives: i.e., search requests to the same destination are sent out in batches. Furthermore, note that nodes do not memorize whether they have already sent a FORWARDPROBE() message to a certain destination. Due to corrupt initial states, this knowledge could be wrong and nodes relying on this knowledge would wait forever. Therefore, nodes periodically send FORWARDPROBE() messages in the TIMEOUT action, instead of only once. Since we make no assumptions on the delivery speed of messages and since channels are not FIFO, it is possible that a PROBEFAIL() message arrives at a node $u$ that is an answer to a FORWARDPROBE() message that was initiated long ago. However, in the meantime, there might have been a successful response. To deal with this situation, each node $u$ stores a sequence number counter $seq$. Whenever INITIATENEWSEARCH($destID$) is executed by $u$ and there is no SEARCH($u, destID$) that waits for an answer to a FORWARDPROBE() message, $u$ increments $u.seq$ and stores the value with the SEARCH($u, destId$) request.

The current sequence number is always attached to each ForwardProbe() message $u$ sends. Responses to probes (success and failure) also contain this sequence number. Whenever a response is sent back to $u$, $u$ checks whether the sequence number of the response is at least the sequence number stored for $destID$. If that is not the case, it simply drops the message, since the answer belongs to a batch of Search$(u, destID)$ messages that have already been processed.

**Algorithm 18** Search+: InitiateNewSearch($destID$)

```
1: create new message m = Search(self, destID)
2: if WaitingFor[destID] = ∅ then
3:     WaitingFor[destID] ← {}
4:     self.seq ← self.seq + 1
5:     seq[destID] ← self.seq
                                   ▷ Store m in WaitingFor
6: WaitingFor[destID] ← WaitingFor[destID] ∪ {m}
```

In order to not blow up the pseudocode unnecessarily , we intentionally left out a sanity check for each node: i.e., before executing each action, each node $u$ makes sure that $Left$ contains only nodes $v$ with $id(v) < id(u)$ and that $Right$ contains only nodes $v$ with $id(u) < id(v)$. If this is not the case for some node $v$, $u$ rearranges the reference to $v$ accordingly. This way, in every computation, the following lemma holds:

**Lemma 9.1.** *For every node $v$ it holds: For all $x \in Left$, $id(x) < id(v)$, and for all $y \in Right$, $id(v) < id(y)$.*

### 9.1.2. Build-Line+ Stabilizes to the Line Topology

The main goal of this section is to prove the following theorem:

**Theorem 9.2.** Build-Line+ *stabilizes to the strict line topology.*

We prove the theorem in three steps: First, we show in Lemma 9.3 that starting from any initial state in which $G$ is weakly connected, $G$ is always weakly connected. Second, we show that starting from any initial state, there eventually is a state in which $G_E$ is a supergraph of the line graph and that the explicit edges corresponding to the line are never removed (see Lemma 9.4

**Algorithm 19** SEARCH+: FORWARDPROBE($source, destID, Next, seq$)

```
1:  if destID = id(self) then
2:      if Next ≠ ∅ then
3:          for all u ∈ Next do
4:              send TEMPDELEGATE(u) to self                          ▷ 𝔇
5:      send PROBESUCCESS(destID, seq, self) to source                ▷ 𝔍
6:      send TEMPDELEGATE(source) to self                             ▷ 𝔇
7:  else                                                  ▷ destID ≠ id(self)
8:      for all u ∈ Next with id(u) > destID do      ▷ Remove wrong nodes
9:          Next ← Next\{u}
10:         send TEMPDELEGATE(u) to self                              ▷ 𝔇
11:     if destID > id(self) then
12:         Next ← Next\{self} ∪ {w ∈ Right|id(w) ≤ destID}
13:         if Next = ∅ then
14:             send PROBEFAIL(destID, seq) to source
15:             send TEMPDELEGATE(source) to self                     ▷ 𝔇
16:         else                                              ▷ Next ≠ ∅
17:             u ← argmin{id(u)|u ∈ Next}
18:             if id(u) < id(self) then
19:                 send TEMPDELEGATE(u) to self                      ▷ 𝔇
20:             else if id(u) < id(argmin{id(v)|v ∈ Right}) then
21:                 Right ← Right ∪ {u}                               ▷ 𝔖
22:             send FORWARDPROBE(source, destID, Next, seq) to u     ▷ 𝔇
23:     else if destID < id(self) then      ▷ Analogous to the previous case.
```

to Corollary 9.9). Third, we prove in Lemma 9.10 that all superfluous explicit edges eventually vanish.

The first step of our proof is encapsulated by the following lemma:

**Lemma 9.3.** *If a computation of* BUILD-LINE+ *starts from a state where $G$ is weakly connected, then in every state $G$ remains weakly connected.*

*Proof.* The proof of the lemma relies on the fact that the BUILD-LINE+ protocol is, at its core, a composition of storing references and the four primitives presented in Section 8.3, which is illustrated by the letters $\mathfrak{I}, \mathfrak{D}, \mathfrak{F}, \mathfrak{R}$ and $\mathfrak{S}$ in the pseudocode. Every line that is not annotated by a letter, is not concerned with node references. Therefore, we can use the result of Lemma 8.3 which proves the lemma. □

**Algorithm 20** SEARCH+: PROBESUCCESS($destID, seq, dest$)

1: **if** $seq \geq seq[destID]$ **then** ▷ The message belongs to currently stored search requests to $dest$.
2: send all $m \in WaitingFor[destID]$ to $dest$
3: $WaitingFor[destID] \leftarrow \emptyset$
4: send TEMPDELEGATE($dest$) to $self$ ▷ $\mathfrak{D}$

**Algorithm 21** SEARCH+: PROBEFAIL($destID, seq$)

1: **if** $seq \geq seq[destID]$ **then** ▷ The message belongs to currently stored search requests to $dest$.
2: $WaitingFor[destID] \leftarrow \emptyset$

For the second step of the proof of the theorem, we introduce the notation $pred(u) := argmax\{id(v)|v \in Left(u)\}$ and $succ(u) := argmin\{id(v)|v \in Right(u)\}$. Furthermore, let $dist(u, v)$ for two nodes $u$ and $v$ denote the hop distance in the (ideal) line topology between $u$ and $v$. We define $rv(v)$ for a node $v$ as $dist(v, succ(v))$ if $Right(v) \neq \emptyset$ or as $n$ otherwise. We define $lv(v)$ analogously for $pred(v)$. Using this notion we define a potential function $\Phi := \sum_{i=1}^{n-1} rv(v_i) + \sum_{i=2}^{n} lv(v_i)$ where $v_1 < v_2 < \cdots < v_n$ are all nodes ordered by their ID increasingly. Notice that $\Phi$ is bounded from above by $2n(n-1)$ and from below by $2(n-1)$. Also notice that according to the protocol, $pred(v)$ ($succ(v)$) can change only if $v$ stores a new node that is closer to $v$ than $pred(v)$ ($succ(v)$) in $Left$ ($Right$). Thus, $\Phi$ never increases. We define the *closest neighbor graph* as the graph $G_{NB} = (V, E_{NB})$, where $V$ is the set of all nodes and $(x, y) \in E_{NB}$ if and only if $y = succ(x) \vee y = pred(x)$. Furthermore, we say an edge is *temporary* if it is an implicit edge due to a TEMPDELEGATE() message. All other types of implicit edges are called *non-temporary*. With this notion established we aim at showing the following lemma.

**Lemma 9.4.** *Consider a system state $s^*$ of the computation of* BUILD-LINE+ *such that $\Phi$ does not decrease in the computation suffix starting in $s^*$, then $G_{NB}$ is bidirected and strongly connected.*

We prove this lemma step-by-step with the Lemmas 9.5 to 9.8, starting with the following lemma:

**Lemma 9.5.** *Consider a system state $s^*$ of the computation of* BUILD-LINE+

*such that $\Phi$ does not decrease in the computation suffix starting in $s^*$, then $G_{NB}$ is bidirected.*

*Proof.* Assume for contradiction there exists an edge $(x, y) \in E_{NB}$ such that $(y, x) \notin E_{NB}$ and w.l.o.g. assume $id(x) < id(y)$. This implies $succ(x) = y$ and $x \neq pred(y)$. Since $\Phi$ does not change anymore, $y$ remains $succ(x)$ and eventually by the fair action execution assumption, TIMEOUT is executed in $x$ and it sends an INTRODUCE$(x, \perp)$ to $y$, which is eventually delivered to $y$ by the fair message receipt assumption. This implicit edge turns into a temporary edge $(y, x)$. Note that if $Left(y) = \emptyset$ or $id(pred(y)) < id(x)$, then $pred(y)$ is replaced by $x$ according to the protocol and because $id(x) < id(y)$. This causes $\Phi$ to decrease, which contradicts to the initial assumption. Therefore, $Left(y) \neq \emptyset$ and $id(x) < id(pred(y)) < id(y)$ must hold. According to the protocol, $(y, x)$ is delegated (first to $pred(y)$, then possibly further) until it reaches a node $z$ that either has no left neighbor or it holds that $id(pred(z)) < id(x) < id(z)$. Here similar arguments as above yield a contradiction. Thus, $G_{NB}$ is bidirected in $s^*$. □

The definition of a closest neighbor graph and Lemma 9.1 imply the following:

**Corollary 9.6.** *If $G_{NB}$ is bidirected and disconnected, every connected component forms a line.*

To show that $G_{NB}$ is also strongly connected if $\Phi$ is at its minimum value, we need two additional lemmas. First, we consider the non-temporary edges.

**Lemma 9.7.** *Consider a state $s^*$ of the computation of* BUILD-LINE+ *such that $G_{NB}$ is bidirected and disconnected. If there is a non-temporary edge $(w, v)$ with $w \in C_1, v \notin C_1$ for a connected component $C_1$, then eventually there is an explicit or a temporary edge $(x, y)$ with $x \in C_1$ and $y \notin C_1$ or $\Phi$ decreases.*

*Proof.* W.l.o.g., assume $id(w) < id(v)$. First of all, note that according to the protocol, if the graph $G_{NB}$ changes, $\Phi$ must decrease. In that case we are done, so in the following we assume that $G_{NB}$ does not change in the computation suffix starting in $s^*$. Furthermore, each connected component of $G_{NB}$ forms a line by Corollary 9.6. We now make a case distinction over all possible types for the edge $(w, v)$. We note that this analysis is an exhaustive and tedious

task, but since we aim for through proof, there is no way to circumvent this exercise in case distinctions.

(a) $(w, v)$ is an implicit edge from a ForwardProbe() message in which $v = source$ or $v \in Next$ and $id(w) = destID$. Then once the message is received, $(w, v)$ is turned into a temporary edge and the claim follows.

(b) $(w, v)$ is an implicit edge from a ForwardProbe() message in which $v = source$ and $destID > id(w)$. Consider the state in which this message is received and the corresponding action is executed. Then $Next$ is updated according to the protocol. If $Next$ is empty after this operation, a temporary edge $(w, v)$ is established and the claim holds. Otherwise, let $u := argmin\{id(u)|u \in Next\}$ after the update. If $id(u) > id(w)$ we have two sub-cases: $id(minRight(w)) > id(u)$ or $id(minRight(w)) \leq id(u)$. In the former case, $u$ is added to $Right(w)$, causing $\Phi$ to decrease, and the claim holds. In the latter case, due to the way $Next$ was updated, $minRight(w) = u$ must hold. Applying the previous arguments recursively yields that the message reaches a node $x \in C_1$ at some point in time where either $destID = id(x)$ or $Next = \emptyset$ after the update. In this case, a temporary edge $(x, v)$ is established.
Now, consider the case $id(u) < id(w)$. Again, we have two sub-cases: Either $u \notin C_1$ or $u \in C_1$. In the former case, since the protocol establishes the temporary edge $(w, u)$, the claim follows. In the latter case, the message is forwarded to $u \in C_1$. Let $u' := argmin\{id(u')|u' \in Next\}$. After the update of $Next$ according to the protocol, it holds $id(u') > id(u)$. Thus, this case reduces to the other case above.

(c) $(w, v)$ is an implicit edge from a ForwardProbe() message in which $v = source$ and $destID < id(w)$. This case is analogous to the previous one.

(d) $(w, v)$ is an implicit edge from a ForwardProbe() message in which $v \in Next$ and $destID > id(w)$. If $id(v) > destID$, $v$ is removed from $Next$ and a temporary edge $(w, v)$ is established. Otherwise, it holds that $id(w) < id(v) < destID$: i.e., the ForwardProbe() is forwarded and will visit $v$ on its path before it reaches its target (or fails to reach it). If the node with minimal ID in $Next$ is $v$ or another node $\notin C_1$ then

the statement holds because $w$ either saves that node locally or creates a temporary edge to that node. If the node with minimal ID in $Next$ is in $C_1$ we consider that node together with the ForwardProbe() message. Applying the previous arguments recursively yields that the message has to reach a node $x \in C_1$ at some point in time such that the node $y$ with minimal ID in $Next$ is $\notin C_1$, since $v$ obviously fulfills that condition. Consequently, $x$ saves $y$ locally or creates a temporary edge to $y$, which again shows the statement.

(e) $(w, v)$ is an implicit edge from a ForwardProbe() message in which $v \in Next$ and $destID > id(w)$. This case is analogous to the previous one.

(f) $(w, v)$ is an implicit edge from a ProbeSuccess() message (in which $id(v) = destID$) and a temporary edge $(w, v)$ is established.

(g) $(w, v)$ is an implicit edge from an Introduce() message. Note that according to the protocol, all edges in an Introduce() message are added either as explicit edges or as temporary edges.

(h) $(w, v)$ is an implicit edge from a Linearize() message and $(w, v)$ is turned into a temporary edge. □

Now we focus on the explicit edges and temporary edges.

**Lemma 9.8.** *Consider a state $s^*$ of the computation of* Build-Line+ *such that $G_{NB}$ is bidirected and disconnected. If there is an explicit or a temporary edge $(w, v)$ with $w \in C_1$ and $v \notin C_1$ for a connected component $C_1$, then eventually there is an explicit or temporary edge $(x, y)$ with $x \in C_1, y \notin C_1$ and $dist(x, y) < dist(w, v)$, or $\Phi$ decreases.*

*Proof.* W.l.o.g., assume $id(w) < id(v)$. First, assume $(w, v)$ is an explicit edge. If $v = succ(w)$, we have a contradiction to the assumption $w \in C_1$ and $v \notin C_1$. Thus $id(w) < id(succ(w)) < id(v)$ must hold. In this case, in Timeout a new edge $(x, v)$ with $id(w) < id(x) < id(v)$ is introduced and the claim holds. Second, assume that $(w, v)$ is an implicit edge from a TempDelegate() message. Then either $id(v) < id(succ(w))$ and $(w, v)$ turns into an explicit edge and $v$ becomes $succ(w)$ which causes $\Phi$ to decrease, or a TempDelegate($v$)

message is sent to $succ(w)$, resulting in a shorter edge $(succ(w), v)$. This completes the proof of the second claim. □

We are now ready to prove **Lemma 9.4**:

*Proof of Lemma 9.4.* Consider the state $s^*$ of the computation of Build-Line+ such that $\Phi$ does not decrease in the computation suffix starting in $s^*$. Furthermore, assume that the closest neighbor graph $G_{NB}$ is disconnected. First, Lemma 9.5 guarantees that $G_{NB}$ is bidirected. Furthermore, by Lemma 9.3, there must be at least one (implicit or explicit) edge $(w, v)$ between a connected component $C_1$ and another connected component. Together with Lemma 9.7 this implies that at some point there must be a temporary or explicit edge $(x, y)$ with $x \in C_1$ and $y \notin C_1$. However, then Lemma 9.8 can be applied. Since there is only a finite number of times that there can be a shorter edge, in some state $s' > s^*$ $\Phi$ has to decrease which yields a contradiction. Thus $G_{NB}$ must be weakly connected. Note that Lemma 9.5 implies that $G_{NB}$ is also strongly connected, yielding the claim of Lemma 9.4. □

Note that since $\Phi$ can never increase and since $\Phi$ is bounded from below, $\Phi$ can decrease for only a finite number of states: i.e., once we are in a suffix in which $\Phi$ remains constant, the conditions of Lemma 9.4 are fulfilled. This lemma and Corollary 9.6 imply the following corollary:

**Corollary 9.9.** *For any computation of* Build-Line+*, there is a state in which $G_E$ is a supergraph of the line topology.*

For the third step of the proof of the theorem, we have the following lemma:

**Lemma 9.10.** *If a computation of* Build-Line+ *contains a state in which $G_E$ is a supergraph of the line topology, then the computation contains a suffix in which $G_E$ is just the line topology and no new explicit edges are ever created again.*

*Proof.* For the proof, we introduce the following notation: We say an implicit edge $(u, v)$ is *right-relevant* if $id(u) < id(v)$ and the implicit edge $(u, v)$ is due to an Introduce$(v, w)$ message in $u.Ch$ for $w \neq \perp$. Accordingly, we say an edge $(u, v)$ is *left-relevant* if $id(v) < id(u)$ and the implicit edge $(u, v)$ is due

to an INTRODUCE$(v, w)$ message in $u.Ch$ for $w \neq \bot$. Additionally, we call an explicit edge $(u, v)$ *superfluous* if $v \neq succ(u) \wedge v \neq pred(u)$.

Consider a state in which the graph formed by the explicit edges is a supergraph of the line topology. First of all, notice that according to the protocol an explicit edge that belongs to the line topology is never removed, because this would require a node $u$ to get acquainted with a node $v$ that is closer than $minLeft(u)$ or $minRight(u)$, which is not possible. In addition, notice that according to the protocol, in every state (right-/left-)relevant edges are the sole implicit edges that can be turned into an explicit edge. Notice that a right-relevant edge $(u, v)$ can only be created by a node $id(w) < id(u)$ with a superfluous explicit edge to $v$. Thus, for every node $u$ it holds: if there is a state $s'$ such that no node $id(w) < id(u)$ with a relevant or superfluous edge $(w, u)$ exists, then there is no relevant or superfluous edge $(x, u)$ with $id(x) < id(u)$ in the computation suffix starting $s'$.

Consider the leftmost node $u$ that either has at least one right-relevant edge or at least one superfluous right neighbor. Note that once all right-relevant edges have been received by $u$, no node with $id(x) \leq id(u)$ ever adds a superfluous right neighbor again. Furthermore, notice that right-relevant edges are turned into explicit edges upon receipt. Now, for every superfluous right neighbor $v$ of $u$, $u$ sends an INTRODUCE$(v, u)$ to some node $w \in Right(u)$. Each of these messages is eventually received and is answered with by LINEARIZE$(v)$ message according to the protocol. This causes $u$ to delegate $v$ to a node $id(x) > id(u)$. After the last superfluous edge has been delegated, no node with $id(x) \leq id(u)$ will ever have a superfluous right neighbor again.

Continuing this approach, we can show that all superfluous right neighbors eventually vanish. Using analogous arguments, we can also show that all superfluous left neighbors eventually vanish. Thus, the lemma follows. □

Note that Corollary 9.9 and Lemma 9.10 imply that BUILD-LINE+ converges to the line topology. Moreover, the second part of Lemma 9.10 yields the closure property. This finishes the proof of Theorem 9.2.

### 9.1.3. Build-Line+ Satisfies Monotonic Searchability

In this subsection we prove the following theorem:

**Theorem 9.11.** BUILD-LINE+ *admissibly satisfies monotonic searchability according to* SEARCH+.

We start with some preliminaries. First we define $R(v)$ as the set of all nodes $x$ with $id(v) < id(x)$ for which there is a directed path from $v$ to $x$ consisting solely of explicit edges $(y, z)$ with $id(y) < id(z)$. Furthermore, we define $R(v, ID) := \{x \in R(v) | id(x) \leq ID\}$. For a set $U$, $R(U) := U \cup \bigcup_{u \in U} R(u)$ and $R(U, ID) := \{x \in R(U) | id(x) \leq ID\}$. Similarly, we define $L(v)$ as the set of all nodes $x$ with $id(x) < id(v)$ for which there is a directed path from $v$ to $x$ consisting solely of explicit edges $(y, z)$ with $id(z) < id(y)$. Accordingly, $L(U) := U \cup \bigcup_{u \in U} L(u)$ and $L(U, ID) := \{x \in L(U) | id(x) \geq ID\}$.

In order to have a clear definition of admissible states we define the following message invariants:

**Invariant 1** If there is an INTRODUCE$(v, w)$ message with $w \neq \bot$ in $u.Ch$, then $v \neq w$, and $u \in R(w)$ (or $u \in L(w)$).

**Invariant 2** If there is a LINEARIZE$(v)$ message in $w.Ch$, then there is a node $u \neq v$ with $u \in Right(w)$ and $v \in R(u)$ if $w < v$ (or $u \in Left(w)$ and $v \in L(u)$ if $v < w$).

**Invariant 3** If there is a FORWARDPROBE$(source, destID, Next, seq)$ message in $u.Ch$ with $id(source) < destID$ (respectively $destID < id(source)$), then

(a) $\forall x \in Next : id(x) \geq id(u)$ and $u = argmin_u\{id(u) | u \in Next\}$ (respectively $\forall x \in Next : id(x) \leq id(u)$ and $u = argmax_u\{id(u) | u \in Next\}$).

(b) $R(next) \subseteq R(source)$ (respectively $u \in L(source)$).

(c) if $v$ with $id(v) = destID$ exists and $v \notin R(Next, destID)$ (respectively $v \notin L(Next, destID)$) then for every state which this Invariant was true with $source.seq[destID] < seq$, $v \notin R(source, destID)$ (or analogously $v \notin L(source, destID)$).

**Invariant 4** If there is a PROBESUCCESS$(destID, seq, dest)$ message in $u.Ch$, then $id(dest) = destID$ and $dest \in R(u)$ if $destID > id(u)$ (or $dest \in L(u)$ if $destID < id(u)$).

**Invariant 5** If there is a ProbeFail($destID, seq$) message in $u.Ch$, then either there is no node with the ID $destID$, or for every state with $u.seq[destID] < seq$, there is no node $v$ with $id(v) = destID$ in $R(u)$ (and $v \notin L(u)$).

**Invariant 6** If there is a Search($v, destID$) message in $u.Ch$, then $id(u) = destID$ and $u \in R(v)$ if $id(v) < destID$ (or $u \in L(v)$ if $destID < id(v)$).

Intuitively, the message invariants aim at making sure that each message that exists in some state $s$ has either been sent out before $s$ or at least looks like it has been sent out before. Invariant 1 and 2 ensure that an Introduce($v, w$) message corresponds to an explicit edge $(v, w)$ and that a Linearize() message is an answer to a previously sent matching Introduce() message. The third Invariant is concerned with ForwardProbe() messages and guarantees that such a message $m$: (i) has a correct $Next$ set, (ii) has been following an existing explicit path in the past and (iii) is consistent with past ForwardProbe() messages (i.e., if $m$ fails to reach $destID$, all past valid ForwardProbe() messages failed to reach that target via an explicit path). Invariant 4 and 5 are concerned with ProbeSuccess() and ProbeFail() messages and basically state that a ProbeSuccess() message implies an explicit path to the target, whereas a ProbeFail() message implies either the non-existence of such an explicit path or the absence of a node with the corresponding $destID$. Finally, Invariant 6 ensures that a Search() message is sent only if there is indeed an explicit path to the node with $destID$. Before we can prove that our protocols satisfies monotonic searchability, we need to show that every computation contains a suffix that consists of admissible states only. The following Lemma 9.12 is a first step to show our desired result.

**Lemma 9.12.** *If a computation of* Build-Line+ *contains an admissible state, then all subsequent states are admissible.*

In order to prove Lemma 9.12, we need the following four lemmas:

**Lemma 9.13.** *If a computation of* Build-Line+ *contains a state such that the first two invariants hold, then the first two invariants hold in all subsequent states.*

*Proof.* Assume for contradiction there is a state $s_1$ in which the first two invariants hold and in the (directly) subsequent state $s_2$ one of the first two

invariants does not hold. This can only be due to one of the following three reasons:

(a) A new INTRODUCE$(v, w)$ message with $w \neq \perp$ was sent to a node $u$ with $u \notin R(w)$ (or $u \notin L(w)$) in $s_1$.

(b) A new LINEARIZE$(v)$ message was sent to a node $w$ in $s_1$, but there is no node $u \neq v$ with $u \in Right(w)$ and $v \in R(u)$ (or $u \in Left(w)$ and $v \in L(u)$).

(c) A node $y$ was removed from a set $Right(w)$ (or $Left(w)$).

We show that all three cases cannot happen.

For the first case, notice that according to the protocol, the only occasion where an INTRODUCE$(v, w)$ message with $w \neq \perp$ is sent is in the TIMEOUT action of a node $w$. However, it is only sent to nodes in $Right(w)$ (or $Left(w)$) and only with a first parameter $v \neq w$.

For the second case, notice that according to the protocol, the only occasion where a LINEARIZE$(v)$ message is sent to a node $w$ is in the INTRODUCE$(v, w)$ action of a node $u'$. This action must have been triggered by an INTRODUCE$(v, w)$ message with $w \neq \perp$. Thus, before the action was executed $u' \in R(w)$ (or $u' \in L(w)$) and $v \neq w$ were both fulfilled since the first invariant holds in $s_1$. This implies that there must be a node $u \in Right(w)$ – i.e., $w < u$ – such that $u' \in R(u)$ or $u' = u$ (or a node $u \in Left(w)$ – i.e., $u < w$ – such that $u' \in L(u)$ or $u' = u$). During the execution of the action, $v$ was added to $Right(u')$ (or $Left(u')$), which implies $v \in R(u)$ (or $v \in L(u)$).

For the third case, note that a node $y$ is removed from $Right(w)$ (or $Left(w)$) only if the LINEARIZE$(y)$ action has been executed by $w$ in $s_1$. However, by the second invariant, there must be a node $u \neq y$ with $u \in Right(w)$ and $y \in R(u)$ (or $u \in Left(w)$ and $y \in L(u)$). Thus, after the removal, $y \in R(w)$ still holds.

Therefore, in all three cases the first two invariants cannot be violated and have to hold in $s_2$, too. □

**Lemma 9.14.** *Consider a state in which the first two invariants hold, if for two nodes $x, v$ it holds that $x \in R(v)$ (respectively $x \in L(v)$), then in every subsequent state, $x \in R(v)$ ($x \in L(v)$).*

*Proof.* We only consider the case $x \in R(v)$, as $x \in L(v)$ is completely analogous.

Obviously, adding additional edges does not remove elements from $R(v)$. Let $y, z$ be two nodes in $R(v)$ such that $z \in Right(y)$ and $id(y) < id(z)$ The sole action that delegates the explicit edge $(y, z)$ and hence could remove nodes from $R(v)$ is the LINEARIZE(). Therefore, consider an arbitrary LINEARIZE($z$) action executed by $y$. Note that since we assumed that the first two invariants hold, right before LINEARIZE($z$) is executed it has to hold that there is a node $u \neq z$ with $u \in Right(y)$ and $z \in R(u)$, by the second invariant. Consequently, after $z$ is removed from $Right(y)$, $z \in R(y) \subset R(v)$ still holds. □

**Lemma 9.15.** *If a computation of* BUILD-LINE+ *contains a state such that the first three invariants hold, then the first three invariants hold in all subsequent states.*

*Proof.* Assume for contradiction that there is a state $s_1$ in which the first three invariants hold and in the (direct) subsequent state $s_2$ one of the first three invariants does not hold. Note that by Lemma 9.13 the first two invariants cannot be violated in $s_2$. Furthermore, by Lemma 9.14 and the fact that $u.seq[id]$ is monotonically increasing (according to the protocol), one can easily show that the sole reason why Invariant 3 is invalidated is if a new FORWARDPROBE() message is sent. Without loss of generality, we consider only the case $id(source) < destID$.

Assume a node $x$ sends a FORWARDPROBE($source, destID, Next, seq$) message to a node $y$. This may happen in two cases: Either in the TIMEOUT action of $x$, or if $x$ receives another FORWARDPROBE($source, destID, Next', seq$) message and executes the corresponding action. In the first case, $Next = \{x\}$ and it is easy to see that part a) and b) of the third invariant are trivially fulfilled. In the second case, both $\forall z \in Next' : id(z) \geq id(y)$ and $y = argmin_u\{id(u)|u \in Next'\}$ hold, since for state $s_1$ :(i) $\forall z \in Next' : id(z) \geq id(x)$ (by the third invariant) and $\forall z \in Right(x) : id(z) \geq id(x)$ (by Lemma 9.1), (ii) only nodes from $Right(x)$ are added to $Next$, (iii) $x$ was $argmin_u\{id(u)|u \in Next\}$ and is not added to $Next'$, and (iv) $y$ is selected as the minimum node from $Next'$. Since the third invariant holds in $s_1$, $x \in R(source)$, which implies $Right(x) \subseteq R(source)$. Now, since $R(Next') \subseteq R(source)$ (again by the third invariant) and $Next = Next' \backslash \{x\} \cup Right(x)$, $R(Next) \subseteq R(source)$. Thus Invariant 3b) holds afterwards.

For the third part of the third invariant, we again distinguish between the two cases that the message was either sent in the TIMEOUT action or in the FORWARDPROBE($source, dest, Next', seq$) action. In the former case, notice that $R(Next, destID) = R(source, destID)$. Assume there has been a state in which $source.seq[destID] < seq$ and $v \in R(source, destID)$ hold. Since $source.seq[destID]$ is monotonically increasing, this must have been a previous state. By Lemma 9.14, $v \in R(source, destID) = R(Next, destID)$ must still hold, yielding a contradiction. In the latter case, assume $v \in R(Next', destID)$ (otherwise, Invariant 3c) trivially holds). Notice that due to Invariant 3b), $x \in R(source)$. Since the only node that is in $R(Next', destID)$ but not in $R(Next, destID)$ is $x$, $v \in R(Next, destID)$ follows.

Thus, the first three invariants still hold in $s_2$. □

**Lemma 9.16.** *If a computation of* BUILD-LINE+ *contains a state such that the first five invariants hold, then the first five invariants hold in all subsequent states.*

*Proof.* Assume for contradiction that there is a state $s_1$ in which the first five invariants hold and in the (direct) subsequent state $s_2$ one of the first five invariants does not hold. Note that by Lemma 9.15 none of the first three invariants can be violated in $s_2$. Furthermore, by Lemma 9.14 and the fact that according to the protocol, $u.seq[id]$ is monotonically increasing, one can check that the sole reason why Invariant 4 or 5 are invalidated is that a new PROBESUCCESS() or PROBEFAIL() message is sent. Again, we only consider the case, $id(u) < destID$ as the other cases are completely analogous.

First, we consider PROBESUCCESS() messages. Assume that a node $x$ sends a PROBESUCCESS($destID, seq, dest$) message to a node $u$. According to the protocol, this may only be in a FORWARDPROBE() action, when a FORWARDPROBE($source, destID, Next, seq$) message arrives at $x$ with $id(x) = destID$ and $u = source$. By case b) of the third invariant, $dest \in R(u)$.

For PROBEFAIL() messages, assume that a node $x$ sends the message to a node $u$. This happens only if a FORWARDPROBE($source, destID, Next, seq$) message arrives at $x$ with $id(x) \neq destID$, $u = source$ and $Next = \{x\}$ and there is no $y$ in $Right(x)$ with $id(y) \leq destID$. If no node with the ID $destID$ exists, we are done. Otherwise, we have that $v \notin R(Next, w)$. By case c) of the third invariant, which implies the claim.

Therefore, the first five invariants have to hold in $s_2$, too. □

Using these lemmas, we can prove **Lemma 9.12**:

*Proof of Lemma 9.12.* Assume for contradiction there exists an admissible state $s_1$ such that in the (direct) subsequent state $s_2$ is not admissible. Note that by Lemma 9.16, none of the first five invariants can be violated in $s_2$. Furthermore, by Lemma 9.14 the sole reason why Invariant 6 can be invalidated is that a new SEARCH() message is sent. Again, without loss of generality $id(u) < destID$.

Assume a node $x$ sends a SEARCH$(v, destID)$ message to a node $u$. According to the protocol, $x = v$ and $v$ has received a PROBESUCCESS$(destID, seq, u)$ for which, by Invariant 4 $id(u) = destID$, and $u \in R(v)$ hold: i.e., the sixth invariant holds. Therefore, all invariants have to hold in $s_2$. □

It remains to show that each computation contains an admissible state.

**Lemma 9.17.** *In every computation of* BUILD-LINE+ *there is an admissible state.*

*Proof.* According to Theorem 9.2, each computation contains a state $s_1$ such that in the suffix starting in $s_1$ every node $x$ has at most one node in $Right(x)$ and at most one node in $Left(x)$. Note that according to the protocol, any INTRODUCE$(v, w)$ message with $v \neq w$ is sent only from a node $w$ with more than one node in $Right(w)$ or $Left(x)$. Thus, by the fair message receipt assumption, the suffix starting in $s_1$ contains a state $s_2$ in which all INTRODUCE$(v, w)$ messages have been received. Furthermore, notice that any LINEARIZE$(v)$ message is sent only from a node $u$ if $u$ received an INTRODUCE$(v, w)$ message. Thus, by the fair message receipt assumption, the suffix starting in $s_2$ contains a state $s_3$ in which all LINEARIZE() messages have been received. This implies that the first two invariants hold in $s_3$. By Lemma 9.13, they do so in every subsequent state.

Consider the computation suffix starting in $s_3$. We show that in this suffix every FORWARDPROBE$(source, destID, Next, seq)$ violating the third invariant vanishes eventually. Without loss of generality we consider only FORWARDPROBE$(source, destID, Next, seq)$ messages with the property that

$id(source) < destID$. First, notice that any ForwardProbe() message initiated in a Timeout action by a node $x$ cannot violate the third invariant. This is obvious for a) and b). For c), notice that if $v$ with $id(v) = destID$ exists and $v \notin R(Next, w)$ and there is an admissible state with $x.seq[destID] < seq$ and $v \in R(x)$, then according to the protocol this state must have been an earlier state and Lemma 9.14 implies that $v \in R(x)$ in the current state, yielding a contradiction.

Second, note that any existing ForwardProbe() message $m$ can trigger the creation of at most one other ForwardProbe() message $m'$ when $m$ is received by a node $x$. If $m$ does not violate the third invariant, $m'$ also does not violate the third invariant (for reasons similar to those in the proof of Lemma 9.15). Thus, we show that every ForwardProbe() message that violates the third invariant can only cause a finite number of ForwardProbe() messages that violate the third invariant (which are eventually received and thus disappear). First of all, note that every ForwardProbe() message $m$ violating Invariant 3a) cannot cause a ForwardProbe() message $m'$ violating Invariant 3a) according to the protocol. Thus, after all initial ForwardProbe() messages have been received, Invariant 3a) holds for every ForwardProbe() message. Now, observe that any such ForwardProbe() message which is received by a node $x$ can only initiate a new ForwardProbe() message to a node $y$ with $id(y) > id(x)$, according to the protocol. Since there is only a finite number of nodes, this implies that all ForwardProbe() messages violating Invariant 3 eventually disappear.

Now, consider a state of the computation $s_4$ in which all of the first three invariants hold. Note that by Lemma 9.15, they hold for all subsequent states, too. The sole action in which a new ProbeSuccess() or ProbeFail() message is sent is in the ForwardProbe() action of a node. Such an action requires the receipt of a ForwardProbe($source, destID, Next, seq$) message $m$ for which, by definition of $s_4$, the third invariant holds. Note that according to the protocol $m$ can only trigger a ProbeSuccess($destID, seq, dest$) message $m'$ that is sent to a node $x$ if $id(u) = destID$ (i.e., $dest = u$) and $x = source$. By Invariant 3b), $u \in R(source)$, implying $dest \in R(x)$: i.e., the fourth invariant holds regarding $m'$. Similarly, a ProbeFail($destID, seq, dest$) message $m'$ to a node $x$ can be caused by $m$ only if $id(u) < destID$ and $Next \backslash \{u\} \cup \{w \in Right | id(w) \leq destID\} = \emptyset$, implying that $v \notin R(Next, destID)$ for a node $v$

with $id(v) = destID$. By Invariant 3c), for every state in which Invariant 3 holds with $source.seq[destID] < seq$, $v \notin R(source, destID)$: i.e., the fifth invariant holds regarding $m'$. All in all, the computation suffix starting in $s_4$ contains a state $s_5$ such that all ProbeSuccess() and ProbeFail() messages that were in the incoming channel of nodes in $s_4$ have been received and consequently, for all ProbeSuccess() and ProbeFail() messages the fourth and fifth invariant holds. By Lemma 9.16, they hold for all subsequent states, too.

Consider the suffix starting in state $s_5$. Notice that a Search$(v, destID)$ message is sent only to a node $u$ in the ProbeSuccess$(destID, seq, u)$ action of $v$, which requires the receipt of a ProbeSuccess$(destID, seq, u)$ message for which, by definition of $s_5$, the fourth invariant holds. This implies $destID = id(u)$ and $u \in R(v)$, yielding Invariant 6 for the new message. Thus, after all Search() messages that were in the incoming channel of nodes in $s_5$ are received, the computation contains a state $s_6$ such that all invariants hold: i.e., $s_6$ is an admissible state. □

Note that Lemma 9.12 and Lemma 9.17 imply the following corollary:

**Corollary 9.18.** *Every computation of* Build-Line+ *contains a suffix in which every state is admissible.*

For the rest of this subsection, we assume that every computation starts in an admissible state, since we want to show that monotonic searchability is satisfied in admissible computation suffixes. Furthermore, without loss of generality, we only consider Search$(u, destID)$ messages with $id(u) < destID$.

Before we can prove Theorem 9.11, we need to prove the following lemma.

**Lemma 9.19.** *For any* ForwardProbe$(v, destID, Next, seq)$ *message* $m$ *with* $id(u) < destID$ *in* $u.Ch$*, it holds that if there exists a node* $w$ *with* $id(w) = destID$ *and* $w \in R(u)$*, then computation contains a state such that a message* $m' =$ ForwardProbe$(v, destID, Next', seq)$ *is in* $w.Ch$*.*

The following Lemma is an intermediate step to prove Lemma 9.19.

**Lemma 9.20.** *Consider a* ForwardProbe$(v, destID, Next, seq)$ *message* $m \in x.Ch$ *and a node* $u \in R(Next, destID)$*. Either* $u = x$ *or the computation contains a state in which a* ForwardProbe$(v, destID, Next', seq)$ *message is in* $y.Ch$ *for some node* $y$ *with* $id(y) > id(x)$ *and* $u \in R(Next', destID)$*.*

*Proof.* Note that when $m$ is received by $x$, a new message with $Next' = Next \backslash \{x\} \cup Right(x)$ is sent. According to the third invariant, for all nodes $z$ in $Next$, $id(z) \geq id(x)$ holds: i.e., $x$ is the node with minimum ID among all nodes in $Next$. By Lemma 9.1, $id(z) \geq id(x)$ also holds for the nodes $z$ in $Right(x)$. Thus, $x$ is the node with minimum ID among all nodes in $R(Next, destID)$ and the new ForwardProbe$(v, destID, Next', seq)$ message is sent to a node $y$ with $id(y) > id(x)$. Furthermore, $R(Next, destID) \backslash \{x\} \subseteq R(Next', destID)$. Thus, also $u \in R(Next', destID)$ and the claim follows. □

Using this, we can prove **Lemma 9.19**:

*Proof of Lemma 9.19.* Note that when $m$ arrives at a node $u$, $Next$ is changed such that $R(Next, w) = R(u, w)$. If $w \in R(u)$, then $w \in R(Next, w)$ afterwards. Thus, we can apply Lemma 9.20 recursively. This results in a state in which a ForwardProbe$(v, destID, Next', seq)$ is in $w.Ch$, which is eventually received according to the fair message receipt assumption. □

We are now ready to prove Theorem 9.11:

*Proof of Theorem 9.11.* Let $m, m'$ be two Search$(u, destID)$ messages initiated by node $u$ in admissible states, such that $m$ was initiated before $m'$ and assume for contradiction that $m$ is delivered successfully, whereas $m'$ is not. Let $v$ be the node with $id(v) = destID$. Note that if $m'$ is added to the set $WaitingFor[destID]$ while $m$ is in the set, then the protocol handles both messages identically: i.e., if $m$ is successfully delivered to $v$ due to an ProbeSuccess() message, $m'$ is delivered as well. Therefore, $m'$ is added to $WaitingFor[destID]$ when $m \notin WaitingFor[destID]$, which implies $u.seq[destID]$ has increased since the successful delivery of $m$ according to the protocol. Since we assume that $m'$ is not delivered successfully, either a ProbeFail$(dest, seq)$ message eventually arrives at $u$ with $seq \geq u.seq[destID]$, or no ProbeSuccess$(destID, seq, dest)$ with $seq \geq u.seq[destID]$, $dest = destID$ ever arrives at $u$. We consider both cases individually. In the first case, by the fifth invariant, $v \notin R(u)$ has to hold even though $m$ was already successfully delivered. By the sixth invariant, $v \in R(u)$ when $m$ was delivered, which is a contradiction to Lemma 9.14. In the second case, note that ForwardProbe$(u, destID, \{u\}, seq)$ messages are

regularly initiated by $u$ with $seq \geq u.seq[destID]$ (since $u.seq$ is monotonically increasing). Again, due to the successful delivery of $m$, the sixth invariant and Lemma 9.14, $v \in R(u)$ when $m'$ was initiated, and therefore, by Lemma 9.19, a ForwardProbe$(u, destID, Next', seq)$ message with $seq \geq u.seq[destID]$ is eventually in $v.Ch$, which is answered by a ProbeSuccess$(destID, seq, v)$ message, causing $m'$ to be delivered to $v$. This contradicts our initial assumption. □

## 9.2. Monotonic Searchability for the Super-Line Topology

In this section, we present the Build-SuperLine protocol and the Greedy-Search protocol. Build-SuperLine stabilizes to the super-line topology and it admissibly satisfies monotonic searchability according to Greedy-Search.

This section has a similar structure to Section 9.1: First, we describe Build-SuperLine and Greedy-Search in detail in Subsection 9.2.1. Then, in Subsection 9.2.2 we prove that the Build-SuperLine protocol stabilizes to the line topology and in Subsection 9.2.3 that the Build-SuperLine protocol satisfies monotonic searchability according to Greedy-Search. We conclude in Subsection 9.2.4 by presenting an extension of Build-SuperLine that allows for short routing paths of search messages . Again, we drop the "according to Greedy-Search" clause for the remainder of this section, since we only consider searchability for Greedy-Search.

### 9.2.1. Description of Build-SuperLine and Greedy-Search

Similarly to the Build-Line+ protocol, the Build-SuperLine protocol uses the original solution for the line topology [ORS07] as its basis. The major difference between Build-Line+ and Build-SuperLine is the way multiple neighbors are handled. In Build-Line+ a node can have multiple neighbors to the left and to the right, but ultimately the node aims at having only one neighbor to each side: i.e., it eventually delegates all explicit edges with the exception of the closest neighbors. In our Build-SuperLine protocol nodes do not delegate explicit edges, but keep the corresponding neighbors in their memory. Consequently, the protocol solely relies on Linearize() messages.

In the TIMEOUT action (see Part 1 of Algorithm 22) a node introduces itself to its closest left and right neighbor by a LINEARIZE() message. Moreover, it introduces the neighbors among each other in a linear fashion: i.e., the closest neighbor to the left/right is introduced to the second closest neighbor, which is introduced to the third closest, and so on. Whenever a node $u$ gets a LINEARIZE($v$) message with $u < v$ (see Algorithm 23), it first tests whether $v$ is closer to itself than its current right neighbor, if so, $v$ is stored in $Right(u)$. Otherwise, $u$ sends a LINEARIZE($v$) message to two nodes in $Right(u)$: (i) the node whose identifier is closest to $v$ from below and (ii) the node whose identifier is closest from above.

Note that this simple protocol has the major disadvantage that a node may send a numerous LINEARIZE() messages in a single TIMEOUT action. One can easily circumvent this by using two self-stabilizing counters, one each for $Left(u)$ and $Right(u)$. Whenever TIMEOUT is executed, only the $i$-th neighbor to the left is introduced to the $i+1$-th neighbor to the left, where $i$ is the value of the counter for the left neighbors. Afterwards, the counter is incremented or, in case the increment would yield a higher value than the number of nodes in $Left(u)$, reset to 1. The same is done with the counter for the right neighbors. It is not hard to see that this change essentially leads to the same behavior as the original protocol, with a drastic decrease of message overhead. In this Section we stick to our original protocol, since it simplifies the correctness analysis.

**Algorithm 22** BUILD-SUPERLINE: TIMEOUT

**Part 1: Self-Stabilization of Topology**

▷ Let $Left = \{v_1, v_2, \dots, v_k\}$ with $id(v_1) < id(v_2) < \dots < id(v_k)$

1: **for all** $v_i \in Left$ with $1 \leq i < k$ **do**

2: send LINEARIZE($v_i$) to $v_{i+1}$ ▷ $\mathfrak{I}$

▷ Let $Right = \{w_1, w_2, \dots, w_l\}$ with $id(w_1) < id(w_2) < \dots < id(w_l)$

3: **for all** $w_i \in Right$ with $1 < i \leq l$ **do**

4: send LINEARIZE($w_i$) to $w_{i-1}$ ▷ $\mathfrak{I}$

5: send LINEARIZE($self$) to $v_1$ ▷ $\mathfrak{I}$

6: send LINEARIZE($self$) to $w_1$ ▷ $\mathfrak{I}$

**Part 2: Monotonic Searchability**

7: **for** $\forall\ destID \in Waiting$ **do** ▷ Regularly send out probes

8: send FORWARDPROBE($self, destID, self.seq$) to $self$ ▷ $\mathfrak{I}$

**Algorithm 23** BUILD-SUPERLINE: LINEARIZE($v$)

```
1:  if id(v) < id(self) then
2:      if Left ≠ ∅ then
3:          x ← argmax{id(x') |x' ∈ Left};
4:          if id(v) > id(x) then
5:              Left ← Left ∪ {v}                                   ▷ 𝔊
6:          else
7:              y_1 ← argmin{id(y') |y' ∈ Left and id(y') > id(v)}
8:              if y_1 exists then
9:                  send LINEARIZE(v) to y_1                        ▷ 𝔇
10:             y_2 ← argmax{id(y') |y' ∈ Left and id(y') < id(v)}
11:             if y_2 exists then
12:                 send LINEARIZE(v) to y_2                        ▷ 𝔇
13:     else
14:         Left ← Left ∪ {v};                                      ▷ 𝔊
15: else                                  ▷ Analogous to id(v) < id(self)
```

The GREEDY-SEARCH protocol works similarly to SEARCH+: i.e., whenever a node $u$ initiates a new search request, the INITIATENEWSEARCH($destID$) action of $u$ is called (see Algorithm 18). In this action node $u$ creates and stores a new SEARCH($u, destID$) message and starts to periodically initiate FORWARDPROBE($u, destID, self.seq$) messages that it sends to itself (see Part 2 of Algorithm 22). The major advantage of GREEDY-SEARCH is the fact that the set of nodes $Next$ is not required anymore, since we exploit the non-line edges of the super-line topology. We keep the sequence number counter $seq$ of SEARCH+ and use them in the same way. In the following, assume $id(u) < destID$ (the other case is analogous). Whenever a FORWARDPROBE($u, destID, seq$) message is received by a node $w$ with $id(w) \neq destID$ (see Algorithm 19), $w$ sends the message to the rightmost neighbor whose ID is still smaller than $destID$: i.e., the message is greedily forwarded to node that is closest to the desired destination ID without surpassing it. If $w$ has no right neighbor, it answers with a PROBEFAIL($destID, seq$) message. If a FORWARDPROBE($u, destID, seq$) message arrives at a node $v$ with $id(v) = destID$, it directly responds with a PROBESUCCESS($destID, seq, v$) message to $u$. As soon as $u$ receives a response, it acts accordingly. In case a FORWARDPROBE($u, destID, seq$) message is successfully answered by a

ProbeFail($destID, seq$) message (see Algorithm 21), it drops the corresponding Search($u, destID$) messages completely. Otherwise, Search($u, destID$) requests waiting at $u$ are directly delivered to $v$ (see Algorithm 20).

Mimicking the Search+ protocol, we save Search($u, destID$) message in batches at the initiating nodes: i.e., whenever a new Search($u, destID$) message is created by $u$ and there are already Search($u, destID$) messages that wait for an answer to a ForwardProbe($u, destID$) message, the new Search($u, destID$) message waits together with these previous requests. These batched messages are aborted or delivered as soon as a ProbeFail($destID$) message or a ProbeSuccess($destID, v$) message arrives at $u$.

**Algorithm 24** Greedy-Search: InitiateNewSearch($destID$)

```
1: create new message m = Search(self, destID);
2: if WaitingFor[destID] = ∅ then
3:     WaitingFor[destID] ← {}
4:     self.seq ← self.seq + 1
5:     seq[destID] ← self.seq
                                   ▷ Store m in WaitingFor
6: WaitingFor[destID] ← WaitingFor[destID] ∪ {m}
```

### 9.2.2. Build-SuperLine Stabilizes to the Super-Line Lopology

In this subsection we prove the following theorem.

**Theorem 9.21.** Build-SuperLine *stabilizes to the super-line topology.*

We intentionally omit to prove that our protocol maintains weak connectivity: i.e., starting from any initial state in which $G$ is weakly connected, $G$ is always weakly connected. The explicit edges are never delegated and connectivity is maintained trivially. Moreover, the implicit edges are delegated as in Build-Line+, for which we have already shown the maintenance of weak connectivity.

First we prove that Build-SuperLine fulfills the convergence property. For a node $v$ let $succ(v)$ denote the node in $Right(v)$ that has the smallest $id$ (i.e., it is the closest current successor). Analogously, we define $pred(v)$ to be the node in $Left(v)$ with highest $id$. Furthermore, the potential of a node $\phi(v)$ is $\phi(v) := |id(succ(v)) - id(v)| + |id(v) - id(pred(v))|$. In case $succ(v)/pred(v)$ does not exist (since $Right(v)/Left(v)$ are empty) we replace $id(succ(v))/id(pred(v))$

**Algorithm 25** GREEDY-SEARCH: FORWARDPROBE($source, destID, seq$)

```
1:  if destID = id(self) then
2:      send ProbeSuccess(destID, seq, self) to source                ▷ 𝔍
3:      send Linearize(source) to self                                ▷ 𝔇
4:  else
5:      if destID < id(self) then
6:          if Left = ∅ then
7:              send ProbeFail(destID, seq) to source
8:              send Linearize(source) to self                        ▷ 𝔇
9:          else
10:             y ← argmin{id(y') | y' ∈ Left and id(y') ≥ destID}
11:             send ForwardProbe(source, destID, seq) to y           ▷ 𝔇
12:             send Linearize(source) to self                        ▷ 𝔇
13:     else
14:         if Right = ∅ then
15:             send ProbeFail(destID, seq) to source
16:             send Linearize(source) to self                        ▷ 𝔇
17:         else
18:             y ← argmax{id(y') | y' ∈ Right and id(y') ≤ destID}
19:             send ForwardProbe(source, destID, seq) to y           ▷ 𝔇
20:             send Linearize(source) to self                        ▷ 𝔇
```

by an *id* value that is bigger/smaller than the maximal/minimal *id* in the network. The potential of the network $G_E = (V, E_E)$ in state s (or short $G_E(s)$) is simply the sum over all node potentials: i.e., $\phi(G_E(s)) = \sum_{v \in V} \phi(v)$. Note that the potential is minimal if $G_E$ contains the line topology as a subgraph. This is even correct when the IDs of nodes are not consecutively numbered. For convenience we use $\phi^*$ to denote the minimal possible value of $\phi(G_E)$

**Lemma 9.22.** *If a computation of* BUILD-SUPERLINE *starts from a state where $G_E$ is weakly connected, the computation contains a state such that $G_E$*

**Algorithm 26** GREEDY-SEARCH: PROBESUCCESS($destID, seq, dest$)

```
1:  if seq ≥ seq[destID] then      ▷ The message belongs to currently stored
    search requests to dest.
2:      send all m ∈ WaitingFor[destID] to dest
3:      WaitingFor[destID] ← ∅
4:  send Linearize(dest) to self                                      ▷ 𝔇
```

**Algorithm 27** GREEDY-SEARCH: PROBEFAIL($destID, seq$)

1: **if** $seq \geq seq[destID]$ **then** ▷ The message belongs to currently stored search requests to *dest*.
2: $WaitingFor[destID] \leftarrow \emptyset$

*is a supergraph of the line topology.*

*Proof.* To show that BUILD-SUPERLINE always converges to a topology that contains the line topology as a subgraph, we simply need to show that for any state $s$ if $\phi(G_E(s)) \neq \phi^*$ then $\phi(G_E)$ decreases eventually.

Assume for contradiction that there is a state $s_1$ such that for all $s' > s_1 : \phi(G_E(s_1)) = \phi(G_E(s')) \neq \phi^*$. Consequently, the line topology is not a subgraph of $G_E(s_1)$. We can subdivide $G_E(s_1)$ into maximal components $C_1, C_2, ..., C_k$ with the property that inside each component all line-subgraph edges have been established. Note that the numbering of the components is consecutive: i.e., $C_1$ contains the node with minimal $id$, $C_2$ contains the node with minimal $id$ that is not in the line component of $C_1$, and so on. Note that in case no line edges are established, each component consists of only a single node.

Consider two of these components $C_i, C_j, i < j$ with the properties that (i) $\exists (u, v) \in E_e : u \in C_i, v \in C_j$ or vice versa and (ii) $|j-i|$ is minimal. Informally stated, we chose the two closest components that are connected by at least one explicit edge. Without loss of generality assume that $u \in C_i, v \in C_j$. If there are multiple explicit edges between $C_i$ and $C_j$ that fulfill this property we pick the unique edge with the property that $id(u)$ is maximal and $id(u) - id(v)$ is minimal.

In case $v = succ(u)$, $u$ introduces itself to $v$ in the timeout action; otherwise it introduces a node $w$ from $Right(u)$ to $v$ such that $id(w) < id(v)$ with $id(w)$ being maximal with that property. In either case, $v$ eventually receives this LINEARIZE() message. If $v$ makes this edge explicit (i.e., the reference is stored in $Left(u)$) we are done, since $\phi(v)$ decreases, which is a contradiction to the assumption that the potential remains constant. Now consider the case that the reference of $u/w$ is forwarded to another node $x_1 \in Left(v)$. If $x_1$ stores the reference and makes the edge explicit, then analogously to $v$ storing the edge we end up with a contradiction. Consequently, consider the *delegation*

*path* of the message $(x_1 = v, x_2, ..., x_h)$: i.e., the nodes the message visits before it either becomes explicit or merges with an existing edge. Note that $id(x_1) > id(x_2) > ... > id(x_h)$. Moreover, all of the nodes on the delegation path have to be in component $C_j$ due to the way we picked the edge $(u, v)$: i.e., if a node $x_i$ in the delegation path is not in $C_j$ but in some other component $C_k$, our initial choices for $C_i, C_j$ and $u, v$ would have been different. Therefore, $x_h$ is also in $C_j$ and the edge $(x_h, u)$ becomes explicit in some state $s^* > s_1$. This leads to a decrease of $\phi(x_h)$ (and also of $\phi(G_E(s^*))$), contradicting our initial assumption.

Finally, consider the case that there is no explicit edge $(u, v)$ that connects two components $C_i, C_j$: i.e., all connected components are connected by implicit edges There are two cases: (i) there is at least one implicit edge connecting two components that is induced by a LINEARIZE() message and (ii) all implicit edges that connect two components are induced by FORWARDPROBE() and PROBESUCCESS() messages. In the first case, we can apply our above approach: i.e., we simply follow the delegation path of the LINEARIZE() message in order to get a contradiction. In the second case, we have to handle the two message types separately. In case a PROBESUCCESS$(destID, seq, dest)$ message in the channel of some node $u$ connects two components $C_i, C_j$, it holds that $dest \in C_i$ and $u \in C_j$ or vice versa. Once the message is received by $u$, it sends a LINEARIZE$(dest)$ message to itself and we can again use our delegation path argument to show a contradiction. If a FORWARDPROBE$(source, destID, seq)$ message in the channel of some node $u$ connects two components $C_i, C_j$, it holds that $source \in C_i$ and $u \in C_j$ or vice versa. Once the message is received by $u$, it sends a LINEARIZE$(source)$ message to itself: i.e., the delegation path argument is again applicable, thus, proving the theorem. □

The closure property follows trivially from the proof of Lemma 9.22, since explicit edges are never delegated.

**Corollary 9.23.** *If a computation of* BUILD-SUPERLINE *starts from a state where $G_E$ is a supergraph of the line topology, then in every state of the computation $G_E$ is a supergraph of the line topology.*

The proof of Theorem 9.2 directly follows from Lemma 9.22 and Corollary 9.23.

### 9.2.3. Build-SuperLine Satisfies Monotonic Searchability

The following theorem towers over this subsection and is our major result:

**Theorem 9.24.** *The* Build-SuperLine *protocol admissibly satisfies monotonic searchability according to* Greedy-Search.

In order to prove Theorem 9.24, we define some preliminary notations. A directed path of explicit edges $(u = v_1, v_2, \dots, v_k = w)$ between two nodes $u, w$ is a *greedy path* if (i) the node identifiers on the path are monotonically increasing or decreasing and (ii) for all nodes $v_i$ it holds that there does not exist a node $x \in Right(v_i)/Left(v_i)$ such that $id(x)$ is in $[id(v_{i+1}), id(w)]$. Informally speaking a path is a greedy path if each edge $(v_i, v_{i+1})$ minimizes the distance to the final node from the local view of $v_i$. Similarly to the $R(v)$ notion of the last section, we define $R_G(v)$ as the set of all nodes $x$ with $id(v) < id(x)$ for which there is a greedy path from $v$ to $x$. $L_G(v)$ is defined correspondingly. Analogously, we also reuse the notion $R_G(v, ID) := \{x \in R_G(v) | id(x) \leq ID\}$ (analogously for $L_G(v, ID)$).

Similarly to Subsection 9.1.3 we first define the following message invariants that have to hold in an admissible state:

**Invariant 1** If there is a ForwardProbe$(source, destID, seq)$ message in $u.Ch$ $id(source) < destID$ (respectively $destID < id(source)$) , then

(a) $R_G(u) \subseteq R_G(source)$ (or $L_G(u) \subseteq L_G(source)$).

(b) if $v$ with $id(v) = destID$ exists and $v \notin R_G(u, destID)$ (respectively $v \notin L_G(u, destID)$) then for every state which this Invariant was true with $source.seq[destID] < seq$, $v \notin R_G(source, destID)$ (or analogously $v \notin L_G(source, destID)$).

**Invariant 2** If there is a ProbeSuccess$(destID, seq, dest)$ message in $u.Ch$, then $id(dest) = destID$ and $dest \in R_G(u)$ if $destID > id(u)$ (or $dest \in L_G(u)$ if $destID < id(u)$).

**Invariant 3** If there is a ProbeFail$(destID, seq)$ message in $u.Ch$, then either there is no node with the ID $destID$, or for every state with $u.seq[destID] < seq$, there is no node $v$ with $id(v) = destID$ in $R_G(u)$ (and $v \notin L_G(u)$).

**Invariant 4** If there is a SEARCH$(v, destID)$ message in $u.Ch$, then $id(u) = destID$ and $u \in R_G(v)$ if $id(v) < destID$ (or $u \in L_G(v)$ if $destID < id(v)$).

These four invariants correspond to the Invariants 2 to 6 of Subsection 9.1.3 in which an intuitive description of all invariants is presented. Next we show that every computation contains a suffix that consists solely of admissible states. The following Lemma 9.25 is a first step to show our desired result.

**Lemma 9.25.** *If a computation of* BUILD-SUPERLINE *contains an admissible state, then all subsequent states are admissible.*

In order to prove Lemma 9.25, we need the following corollary and lemmas.

**Corollary 9.26.** *If for two nodes $x, v$ it holds that $x \in R_G(v)$ in some state $s$ (respectively $x \in L_G(v)$), then in every subsequent state, $x \in R_G(v)$ ($x \in L_G(v)$).*

The corollary trivially holds since each node always keeps its explicit edges and adds an edge to $Right(w)$ ($Left(w)$) if the identifier of the received node is smaller (bigger) than the smallest (biggest) identifier in $Right(w)$ ($Left(w)$): i.e., we only add nodes to $R_G(v)$ and $L_G(v)$.

We now consider the first invariant.

**Lemma 9.27.** *If a computation of* BUILD-SUPERLINE *contains a state such that the first invariant holds, then the first invariant holds in all subsequent states.*

*Proof.* Assume for contradiction that there is a state $s_1$ in which the first invariant holds and in the (direct) subsequent state $s_2$ it does not hold anymore. Note that by Corollary 9.26 and the fact that sequence numbers are monotonically increasing (according to the protocol), one can easily show that only the sending of a new FORWARDPROBE() message is able to invalidate Invariant 1. Without loss of generality we only consider the case, where $id(source) < destID$.

Assume a node $x$ sends a FORWARDPROBE$(source, destID, seq)$ message to a node $y$. This can happen in two cases: Either in the TIMEOUT action of $x$, or if $x$ receives another FORWARDPROBE$(source, destID, seq)$ message

and executes the corresponding action. In the first case, $x$ sends the message to itself and it is easy to see that part a) of the first invariant is fulfilled. In the second case, we know that the message that triggered the action fulfilled the first invariant: i.e., $R_G(x) \subseteq R_G(source)$. Moreover, $y \in Right(x)$ which yields that $R_G(y) \subseteq R_G(source)$. Thus Invariant 1a) still holds afterwards.

For the second part of the first invariant, assume that $v \in R_G(x, destID)$ in $s_1$ (otherwise, Invariant 1b) trivially holds). Again, we distinguish between the two cases that the message was either sent in the TIMEOUT action or in the FORWARDPROBE$(source, dest, seq)$ action. In the former case, notice that $R_G(x, destID) = R_G(source, destID)$ and due to Corollary 9.26 Invariant 1b) still holds in $s_2$. In the latter case, $x \in R_G(source)$ due to Invariant 1a) and $y \in Right(x)$ by definition. Consequently, it is not possible that $v \notin R_G(y, destID)$.

Thus, the first invariant still holds in $s_2$. □

Let us now consider the first three invariants.

**Lemma 9.28.** *If a computation of* BUILD-SUPERLINE *contains a state such that the first three invariants hold, then the first three invariants hold in all subsequent states.*

*Proof.* Assume for contradiction that there is a state $s_1$ in which the first three invariants hold and in the (direct) subsequent state $s_2$ one of the first three invariants does not hold. Note that by Lemma 9.27 the first invariant cannot be violated in $s_2$. Furthermore, by Corollary 9.26 and the fact that according to the protocol sequence numbers are monotonically increasing, one can check that the sole reason why Invariant 2 or 3 are invalidated is that a new PROBESUCCESS() or PROBEFAIL() message is sent. Without loss of generality we assume $id(u) < destID$.

First, we consider PROBESUCCESS() messages. Let $x$ be the node that sends a PROBESUCCESS$(destID, seq, dest)$ message to a node $y$ in $s_1$. According to the protocol, this happens in the FORWARDPROBE() action only when a FORWARDPROBE$(source, destID, seq)$ message arrives at $x$ such that $id(x) = destID$ and $y = source$. Since $s_1$ is an admissible state, case a) of the first invariant holds: i.e., $dest \in R_G(y)$.

For the PROBEFAIL() messages, let again $x$ be the node that sends the PROBEFAIL$(destID, seq)$ message to a node $y$. This happens only in a

ForwardProbe() action in case a ForwardProbe($source, destID, seq$) message arrives at $x$ such that $id(x) \neq destID$, $y = source$ and there is no node $z$ in $Right(x)$ with $id(z) \leq destID$. If no node with the ID $destID$ exists, we are done. Otherwise, we have that $v \notin R_G(u)$ with $id(v) = destID$. By case b) of the first invariant, this implies the claim.

Consequently, Invariants 2 and 3 still have to hold in $s_2$, which is a contradiction to the initial assumption and the first three invariants have to hold in $s_2$, too. □

Now we can plug together Lemma 9.27 and 9.28.

*Proof of Lemma 9.25.* Assume for contradiction there exists an admissible state $s_1$ such that in the (direct) subsequent state, $s_2$ is not admissible. Note that by Lemma 9.28, none of the first three invariants can be violated in $s_2$. Furthermore, by Lemma 9.14 the sole reason why Invariant 4 can be invalidated is that a new Search() message is sent. Without loss of generality we only consider the case $id(u) < destID$.

Assume a node $x$ sends a Search($v, destID$) message to a node $y$. According to the protocol, $x$ receives a ProbeSuccess($destID, seq, u$) in $s_1$, for which Invariant 2 holds: i.e., $id(u) = destID$, and $u \in R_G(x)$. Accordingly, the fourth invariant holds. Therefore, all invariants have to hold in $s_2$, too. □

It remains to show that each computation contains an admissible state.

**Lemma 9.29.** *In every computation of* Build-SuperLine *there is an admissible state.*

*Proof.* According to Theorem 9.21, each computation contains a state $s_1$ such that in the suffix starting in $s_1$ the line is a subgraph of $G_E$. We show that in this suffix every ForwardProbe($source, destID, seq$) violating the first invariant vanishes eventually. Without loss of generality we consider only ForwardProbe($source, destID, seq$) messages with the property that $id(source) < destID$. First, notice that any ForwardProbe() message initiated in a Timeout action by a node $x$ cannot violate the first invariant. This is obviously true for case a) and holds for case b) by Corollary 9.26.

Moreover, note that any existing ForwardProbe() message $m$ can trigger the creation of at most one new ForwardProbe() message $m'$ when $m$

is received by a node $x$. If $m$ does not violate the first invariant, $m'$ also does not violate it (for reasons similar to those in the proof of Lemma 9.27). Consequently, we just need to show that every ForwardProbe() message that violates the first invariant can only cause a finite number of ForwardProbe() messages that violate the first invariant (which is eventually received and thus disappears). Since there is only a finite number of nodes, this implies that all ForwardProbe() messages violating Invariant 1 eventually disappear.

Now, consider a state of the computation $s_2$ in which the first invariant holds. Note that by Lemma 9.27, they hold for all subsequent states. The sole action in which a new ProbeSuccess() or ProbeFail() message can be sent is in the ForwardProbe() action of a node. Such an action requires the receipt of a ForwardProbe($source, destID, seq$) message $m$ for which, by definition of $s_2$, the first invariant holds. Note that according to the protocol $m$ can only trigger a ProbeSuccess($destID, seq, dest$) message $m'$ that is sent to a node $x$, if $id(u) = destID$ (i.e., $dest = u$) and $x = source$. By Invariant 1a), $u \in R_G(source)$, implying $dest \in R_G(x)$: i.e., the second invariant holds regarding $m'$. Similarly, a ProbeFail($destID, seq, dest$) message $m''$ to a node $x$ can be caused by $m$ only if $id(u) < destID$ and there is no node in $Right(u)$ with an ID lower than destID, implying that $v \notin R_G(source, destID)$ with $id(v) = destID$. By Invariant 1b), for every admissible state with $source.seq[destID] < seq$, $v \notin R_G(source, destID)$: i.e., the third invariant holds regarding $m''$. Consequently, all ProbeSuccess() and ProbeFail() messages created in the suffix starting in $s_2$ do not violate admissibility. Moreover, by the fair message receipt assumption the computation suffix starting in $s_2$ contains a state $s_3$ such that all ProbeSuccess() and ProbeFail() messages that were in the incoming channel of nodes in $s_2$ have been received. Consequently, for all ProbeSuccess() and ProbeFail() messages the second and third invariant holds. By Lemma 9.28, they hold for all subsequent states, too.

Consider the suffix starting in state $s_3$. Notice that a Search($v, destID$) message is only sent to a node $u$ in the ProbeSuccess($destID, seq, u$) action of $v$, which requires the receipt of a ProbeSuccess($destID, seq, u$) message for which, by definition of $s_3$, the second invariant holds. This implies, $destID = id(u)$ and $u \in R_G(v)$, yielding Invariant 4 for the new message. Thus, after all Search() messages that were in the incoming channel of nodes in $s_3$ are received, the computation contains a state $s_4$ such that all invariants hold: i.e.,

$s_4$ is an admissible state. □

Note that Lemma 9.25 and Lemma 9.29 imply the following corollary:

**Corollary 9.30.** *Every computation of* Build-SuperLine *contains a suffix in which every state is admissible.*

For the rest of this subsection, we assume that every computation starts in an admissible state, since we want to show that monotonic searchability is satisfied in admissible computation suffixes. Furthermore, without loss of generality, we again only consider Search$(u, destID)$ messages with $id(u) < destID$.

Before we can prove Theorem 9.24, we need to prove the following lemma.

**Lemma 9.31.** *For any* ForwardProbe$(v, destID, seq)$ *message* $m$ *with* $id(u) < destID$ *in* $u.Ch$*, it holds that if a node* $w$ *with* $id(w) = destID$ *exists and* $w \in R_G(u)$*, then the computation contains a state such that a message* $m' =$ ForwardProbe$(v, destID, seq)$ *is in* $w.Ch$*.*

*Proof.* Consider the unique greedy path induced by the fact that $w \in R_G(u)$. By the definition of a greedy path and the way new nodes are added to $Right(w)/Left(w)$ in Build-SuperLine, the path persists throughout the computation. Moreover, Greedy-Search is designed in such a way that it forwards $m$ along the greedy path. Due to the fair message receipt assumption ForwardProbe() messages cannot be stalled forever: i.e., each node on the path receives a ForwardProbe() that corresponds to $m$. This results in a state in which a ForwardProbe$(v, destID, seq)$ is in $w.Ch$, which is received eventually. □

We can now prove the main theorem of this subsection.

*Proof of Theorem 9.24.* Let $m, m'$ be two Search$(u, destID)$ messages initiated by node $u$ in admissible states, such that $m$ was initiated before $m'$. Assume for contradiction that $m$ is delivered successfully, whereas $m'$ is not. Let $v$ be the node with $id(v) = destID$. Note that if $m'$ is added to the set $WaitingFor[destID]$ when $m$ is already in the set, then the protocol handles both messages identically: i.e., both are successfully delivered, which contradicts our assumption. Therefore, $m'$ has to be added to $WaitingFor[destID]$

when $m \notin WaitingFor[destID]$, which implies $u.seq[destID]$ has increased since the successful delivery of $m$.

Since we assume that $m'$ is not delivered successfully, two cases can occur: (i) a ProbeFail$(dest, seq)$ message eventually arrives at $u$ with $seq \geq u.seq[destID]$, or (ii) no ProbeSuccess$(destID, seq, dest)$ message with $seq \geq u.seq[destID]$ and $id(dest) = destID$ ever arrives at $u$. We consider both cases individually. In the first case, by the third invariant, $v \notin R_G(u)$ has to hold even though $m$ was already successfully delivered, contradicting Corollary 9.26. In the second case, note that ForwardProbe$(u, destID, seq)$ messages are regularly initiated by $u$ in the Timeout action with $seq \geq u.seq[destID]$ (since $u.seq$ is monotonically increasing). Again, due to the successful delivery of $m$, the fourth invariant and Lemma 9.26, $v \in R_G(u)$ when $m'$ was initiated, and therefore, by Lemma 9.31, a ForwardProbe$(u, destID, seq)$ message with $seq \geq u.seq[destID]$ is eventually in $v.Ch$. Node $v$ answers with a ProbeSuccess$(destID, seq, v)$ message, causing $m'$ to be sent to $v$. This contradicts the assumption that $m'$ is not successfully delivered. □

This concludes our analysis of monotonic searchability for the super-line topology.

### 9.2.4. An Extension with Short Routing Paths

An interesting side effect of the super-line protocol is that it should have a sublinear diameter with high probability. This follows from the basic fact that the topology can have a linear diameter only if the initial graph is a line or close to the line: i.e., there are almost no short-cutting edges that could decrease the diameter drastically. It would be highly desirable to show that Build-SuperLine stabilizes to a topology that has a polylogarithmic diameter with high probability. However, since the final topology of a computation heavily depends on the initial graph and the activation order of nodes, this problem seems hardly tractable in our setting. Instead, we present an extension of Build-SuperLine that uses ideas developed in the area of small-world networks, especially the formidably written chapter of the book *Networks, Crowds, and Markets: Reasoning about a Highly Connected World* [EK10] that builds up on the work of Jon Kleinberg in [Kle00].

In the one-dimensional setting, a small-world network is envisioned in the following way in [EK10]: Each node has an identifier in the interval $[0, 1]$. We then arrange the nodes on a one dimensional ring according to their identifier. Each node is connected by directed edges to its immediately preceding neighbor and succeeding neighbor. This includes an edge between the node with highest ID to the node with lowest ID and vice versa. Additionally, each node $v$ also has a single directed edge to some other node, which is called a *shortcut*. The probability that $v$ links to some node $w$ is $\frac{1}{Z}d(v, w)^{-1}$, where $Z$ is a normalizing constant and $d(v, w)$ is the distance between $v$ and $w$ on the ring. Note that $\frac{1}{Z}d(v, w)^{-1}$ can be lower bounded by $\frac{1}{2\log n}d(v, w)^{-1}$ as it is shown in [EK10]. The overall structure of the network is a ring that is augmented with random shortcut edges. For this kind of network [EK10] shows that a simple myopic greedy search protocol takes $\mathcal{O}(\log^2 n)$ hops in expectation.

We want to embed the structure of this small-world network into our Build-SuperLine protocol. Consequently, we assume that the IDs of our nodes are in $[0, 1]$ as well. Analogous to the consistent hashing approach [Kar+97], we assume that the nodes are assigned to IDs in $[0, 1]$ in a pseudorandom manner. Thus, we can assume that the nodes are distributed uniformly at random over the interval $[0, 1]$. In order to emulate the shortcut edge of a node, we allow each node to create a *virtual* node. The ID of this virtual node is chosen similarly to the way the shortcuts are created in the original protocol: i.e., for a *real* node $v$ and its virtual node $v_{virtual}$ the probability that the ID of the virtual node has a distance $d$ to $id(v)$ is proportional to $\frac{1}{Z}d(v, v_{virtual})^{-1}$. The IDs of nodes will not be variables in our self-stabilizing protocol: i.e., they do not change throughout a computation. This approach is inspired by the self-stabilizing protocols of [RSS11; Jac+] which also use pseudorandom IDs in $[0, 1]$ and virtual nodes. In contrast to these approaches, in our protocol a real node will always have an edge to its virtual node and vice versa. This virtual edge is treated like an explicit edge: i.e., it can be used to route messages along it. With these changes in place, we use the already established Build-SuperLine protocol to stabilize the system to the super-line topology with $2n$ nodes. Note that, contrary to the original small-world network, we do not build a ring among the nodes: i.e., there is not necessarily an explicit edge between the node with the highest and the lowest ID. In addition, we use the Greedy-Search protocol for searching which can now make use of the shortcuts established by

the virtual nodes. In fact, Greedy-Search is similar to the myopic greedy search protocol described by Kleinberg, with the exception that we disallow Greedy-Search to perform a hop that overshoots the target ID.

It remains to show that we achieved our goal of building a protocol such that the final topology yields short routing paths for search messages. The proof of the following theorem is a reevaluation of the proof in [EK10] for our topology.

**Theorem 9.32.** *The above described* Build-SuperLine *protocol yields a topology such that* Greedy-Search *needs* $\mathcal{O}(\log^2 n)$ *hops in expectation.*

*Proof.* Let $s$ be a randomly chosen start node and $t$ be a random target. Throughout the proof we assume that we search for an ID of a node that actually exists in the network. This is not a severe restriction since the routing paths for search messages that target non-existing IDs terminate at some existing node, for which our statement will hold as well. Without loss of generality let $id(s) < id(t)$.

Let $X$ denote the number of hops required by Greedy-Search to reach $t$. For two nodes $u, v$ with $id(u) < id(v)$, we say $u$ has a *distance* of $k$ to $v$, if there are $k$ many nodes whose ID is between $id(u)$ and $id(v)$. We subdivide the path from $s$ to $t$ into phases and the message is in phase $i$ if it is at a node whose distance to the target is in between $2^i$ and $2^{i+1}$. As a consequence, there are at most $\log(2n) = \log(n) + 1$ phases and we can express $X$ by the sum of the number of hops taken in each phase: i.e., $X = X_1 + X_2 + \cdots + X_{\log(n)+1}$ and, due to linearity of expectation, $E[X] = E[X_1] + E[X_2] + \cdots + E[X_{\log(n)+1}]$. Thus, all it remains to show is that the expected value of each $X_i$ is at most proportional to $\log n$.

We fix a phase $j$ of the path and a node $v$ who currently holds the message and whose distance $d$ to the target is between $2^j$ and $2^{j+1}$. Naturally, phase $j$ ends once the message reaches a node whose distance to $t$ is below $2^j$. Let $V_{d/2}$ denote the set of nodes which have a distance of at most $\frac{d}{2}$ from $t$ and whose ID is in between $id(v)$ and $id(t)$. There are $\frac{d}{2} + 1$ nodes in $V_{d/2}$ and each node $w$ in $V_{d/2}$ has a probability of at least

$$\frac{1}{2\log n} d(v,w)^{-1} \geq \frac{1}{2\log n} \cdot \frac{1}{d} = \frac{1}{2d\log n}$$

of being the virtual node of $v$. Therefore, the probability that one of them is

the virtual node of $v$ is at least $\frac{d}{2} \cdot \frac{1}{2d \log n} = \frac{1}{4 \log n}$. Consequently, in each step phase $j$ has a probability of $\frac{1}{4 \log n}$ of ending. Conversely, the probability that phase $j$ runs for at least $k$ steps (i.e., the message performs $k$ hops in it) is at most $(1 - \frac{1}{4 \log n})^{k-1}$.

We are now able to give an upper bound for $E[X_j]$, which can be expressed by the formula $E[X_j] = Pr[X_j \geq 1] + Pr[X_j \geq 2] + Pr[X_j \geq 3] + \cdots$. We have just shown that $Pr[X_j \geq k] \leq (1 - \frac{1}{4 \log n})^{k-1}$. Thus, we can conclude that $E[X_j] = 1 + (1 - \frac{1}{4 \log n})^1 + (1 - \frac{1}{4 \log n})^2 + \cdots$. This geometric sum converges to $4 \log n$: i.e., $E[X_j] \leq 4 \log n$. Consequently, $E[X]$ can be expressed as the sum of $\log(n) + 1$ terms $E[X_i]$, each of which is upper bounded by $4 \log n$, which is the desired statement. □

As Dietzfelbinger and Woelfel [DW14] have shown, the chosen probability for the long range links that only depends on $d(v, w)^{-1}$ is optimal for greedy routing: i.e., greedy routing does not perform asymptotically better for any other uniform and isotropic augmenting distribution. This completes our formal analysis of the super-line.

## 9.3. Comparing the Strict Line and the Super-Line

We want to conclude our analysis on monotonic searchability by comparing our protocols for the strict line and for the super-line as well as their respective search protocols. In order to do so we implemented the protocols in a simulator and conducted experiments. We want to emphasize that this section is not meant as a full-fledged experimental analysis of both scenarios, instead we simply want to stress a few interesting properties that stood out to us. Most importantly, we want to highlight that the Build-SuperLine protocol for the strict line is not only simpler than the Build-Line+ protocol for the strict line, but also more efficient in various ways. We are going to compare the two protocols in terms of certain network properties (i.e., degree growth and node distances), time and space complexity of self-stabilization and the number of successful searches during stabilization.

In Subsection 9.3.1 we explain the setup of our experiments. Subsection 9.3.2 focuses on the comparison of network properties. Subsection 9.3.3 concentrates on the time and space complexity of self stabilization (in terms of stabilization

time and required messages until the network is stable). In Subsection 9.3.4 we briefly evaluate the impact of searching on the stabilization time. Finally, Subsection 9.3.5 considers the search messages during stabilization in terms of successfulness.

### 9.3.1. Experiment Setup

All experiments were performed in a specifically designed simulator written in Java that ran on a Windows 10 computer with an AMD 6100 six-core processor running at 3.30 GHz and 8 GB RAM. Initial graphs for each simulation were random graphs with an average node degree of 2 and with the additional constraint of being weakly connected. In order to allow for a comprehensible comparison between the two protocols (and the corresponding topologies), both protocols were executed on the same initial graph topology for each simulation run. In case search messages were required for an experiment, both protocols had to handle the same rate of search messages in a certain time interval (i.e., 200 milliseconds).

For most experiments we focused on the scalability of our approaches, so the varied parameter is the network size: i.e., the number of nodes in the network. If possible, the following network sizes were investigated: $50, 100, 250, 500, 1000$, 2000 and 4000. In scenarios where search messages are initiated, the maximum network size was reduced to 250 nodes. This was due to increased memory consumption of search messages which afflicted the performance of the simulator. For every single parameter value we conducted 50 experiments, each with its own initial graph, and took the average of the result.

In order to allow for a reasonable implementation of our protocols we made the following changes/additions:

(a) If a node has performed a Timeout action it is forced to sleep for a time between 50 and 100 milliseconds. This is used to avoid deadlocks: i.e. to enforce a behavior that is similar to weakly fair action execution, since there is no 100% fairness for threads in Java.

(b) The Timeout action of a node is triggered when there is no message in the channel at the moment, i.e., nodes prefer handling messages to executing the Timeout action.

(c) If a node receives a message from its incoming channel it inspects the whole channel to determine whether the message is contained multiple times in the channel. If that is the case, it receives the message and its duplicates at once (i.e., they are all removed from the channel). Search messages (and their probes) are an exception to that rule. This algorithmic trick drastically reduces the required memory for each node and facilitates simulations for network sizes over 1000.

(d) There is a central monitoring node in the system which periodically (i.e., every 200 milliseconds) checks whether a legitimate state has been reached. Consequently, the measured stabilization time is in tolerance of 200ms.

### 9.3.2. Network Properties

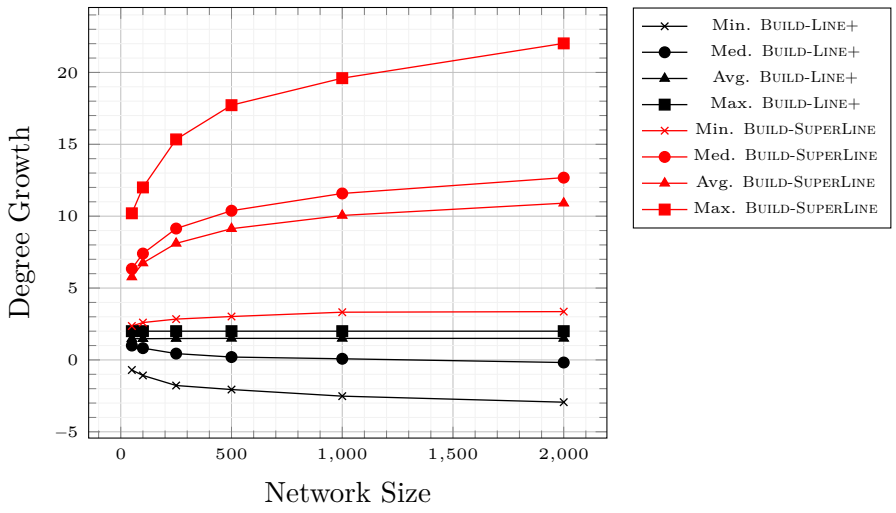



Figure 9.1.: Degree Growth Comparison

A natural consequence of the Build-SuperLine protocol is that the degree of nodes grows until the line is fully stabilized, since nodes never delegate edges. The degree of a node can also increase throughout the execution of Build-Line+ for a finite amount of time. However, once the topology is fully stabilized every node has a degree of at most two. One notion to capture the development of degrees is the *degree growth*. For a node $u$ the degree

growth is the maximum degree of that node in the computation minus its initial degree. In Figure 9.1 we compare the degree growth of Build-Line+ and Build-SuperLine. We investigate the minimum, the maximum, the average and the median degree growth for varying network sizes.

As one can easily see in the graphs, the degree growth of Build-Line+ is almost constant (average and max. degree growth): i.e., almost independent of the network size. This is not too surprising since nodes try to get rid of superfluous edges every time the timeout action is executed. More interestingly, the minimum node degree is negative and declines with an increasing number of nodes. This is an artifact of how initial graphs are created: i.e., with an increasing number of nodes it is more likely to get a node that has a high initial degree and, thus, can have a negative degree growth, since the initial degree is the maximum degree.

For Build-SuperLine the plots look vastly different, since nodes never delegate edges. Consequently, the degree growth has to be positive. The minimum degree growth is almost constant. Much more importantly, the plots of the average, median and maximum node degree look similar to a shifted logarithmic curve. This is indeed an interesting insight, since one can easily construct instances in which Build-Line+ has a linear degree growth for some nodes. We conjecture that the super-line has a logarithmic degree growth in expectation.

Another interesting parameter for an overlay network is the pairwise distance between nodes. Figure 9.2 contains the graphs for Build-Line+ and Figure 9.3 for Build-SuperLine. In both figures we omit the graph for minimum distance, since it would always be 1 (because both protocols stabilize to a topology that contains the line as a subtopology). One can clearly see in Figure 9.2 that for the strict line the maximal distance between nodes (i.e., the *diameter* of the network) is always linear and that even the average and median distances behave linearly. Since the final topology of Build-Line+ is the strict line, this result was to be expected.

In comparison, Build-SuperLine is much more efficient. As we can see in Figure 9.3 the maximal distance between nodes is close to logarithmic. In fact, the average and median distance between nodes seems to hit a ceiling at the value of 4. As a consequence, search requests can be routed quickly in the constructed topology.

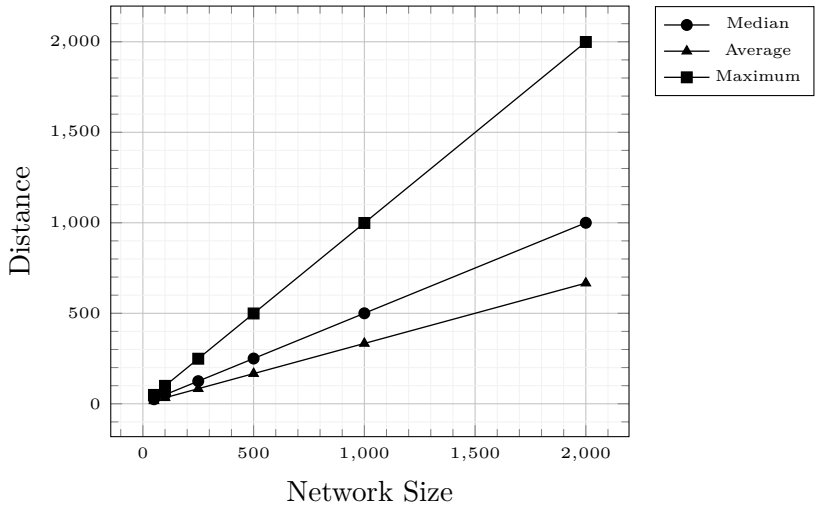



Figure 9.2.: Distances Between Nodes (Build-Line+)

To summarize, one can conclude that Build-Line+ behaves as one would predict, the degree growth is very low while the distances between nodes are high. On the other hand Build-SuperLine constructs a topology with a very low average node distance and even a low diameter. This advantage is traded off by a degree growth that grows logarithmically with network size.

### 9.3.3. Self-Stabilization (without Searching)

We now focus on the time and space complexity of the self-stabilization process. For our evaluation of time complexity we use milliseconds as the unit for measuring time. We are aware that the value itself is not very meaningful, since Java is not an efficient programming language for experimental evaluations and the value is heavily dependent on the machine used for the experiments. However, since we merely want to compare two protocols (and not the efficiency of one protocol), we believe that the choice is apt.

As we can see in Figure 9.4 the stabilization time of Build-Line+ is always longer than the one of Build-SuperLine. Even in small network sizes (i.e., up to 500 nodes) Build-Line+ is roughly five times slower than Build-SuperLine. This ratio increases for bigger network sizes. In fact, Build-Line+ is so slow that we had to cancel all simulations for network sizes greater than 2000 nodes. We can find one possible justification for this

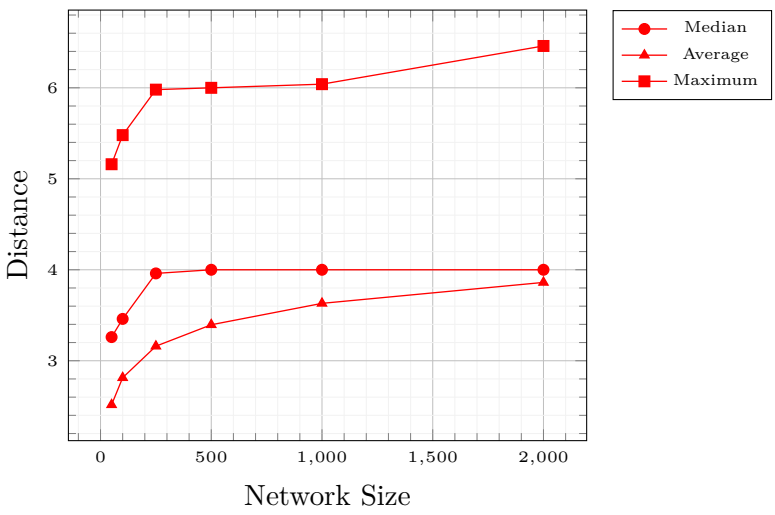



Figure 9.3.: Distances Between Nodes (Build-SuperLine)

observable effect in Figure 9.5. However, before we discuss the required number of messages for stabilization, we want to emphasize that the stabilization time for both protocols seems to be superlinear: i.e., at least quadratic.

The graphs for required messages until stabilization follow a similar pattern to the graphs for stabilization time (see Figure 9.5). Build-Line+ requires $3-6$ times more messages than Build-SuperLine. This is due to the fact that Build-Line+ requires at least three messages to delegate a single edge whereas Build-SuperLine simply never delegates. In fact, our simulations indicate that the number of messages is one of the main influencing factors for stabilization time. As the number of nodes increase, the nodes have to handle a bigger number of messages in their channel: i.e., it takes longer to process messages. However, it needs to be stated that the observed effect for large network sizes might be affected by the limited memory of the computer. It is possible that the influence of the number of messages on the stabilization time is much lower if memory is not an issue.

### 9.3.4. The Influence of Searching

In this subsection we want to evaluate whether search messages slow down the stabilization process. We tested various search message rates and measured their impact on the stabilization time (see Figure 9.6). The message rate

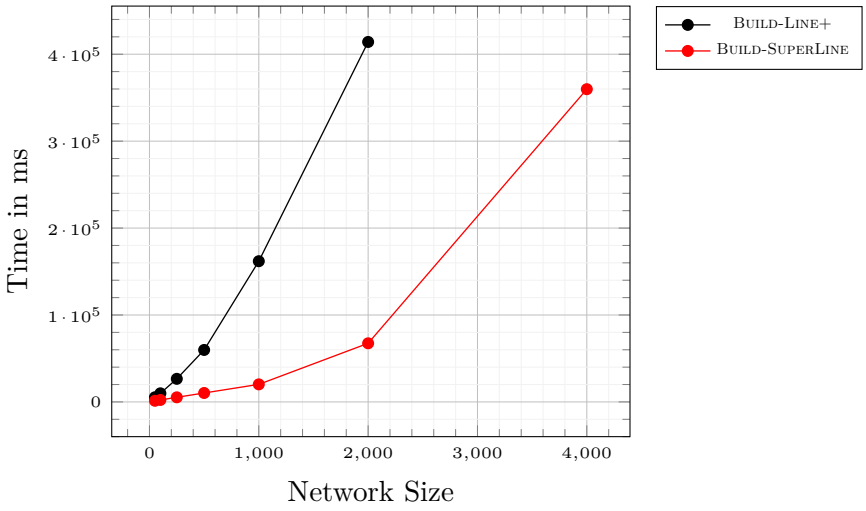



Figure 9.4.: Stabilization Time Comparison

indicates how many search requests are started per $200ms$ of the simulation. For these requests the source node is randomly selected from all nodes and $destID$ is also randomly selected from all IDs except the ID of the source node, so every search message is independently created. This is done by a central instance that has an overview over all nodes. After such a batch of search messages is created, the messages are distributed to the respective source nodes. The simulations were done only to a maximum network size of 250 nodes, since the local memory of a node gets flooded by messages in larger network sizes (i.e., nodes require a huge local memory). As we can see by the graphs, the message rate influences Build-SuperLine differently than Build-Line+. For Build-Line+ we can clearly see that the more searches are initiated, the more time stabilization requires. However, the impact is not severe. Even for 250 nodes the stabilization time for $\frac{20 \text{ Searches}}{200ms}$ is only 13% higher than the one without any searches. This slow down is most likely due to the message size of the probe message in Search+. The $Next$ set can increase to a size that is linear in the number of nodes and thus requires more computer memory than any other message. As stated before, this increased memory overhead can slow down the stabilizing process significantly.

For Build-SuperLine we can observe the inverted effect: i.e., the simulations without any searches take longer to stabilize than the one with search

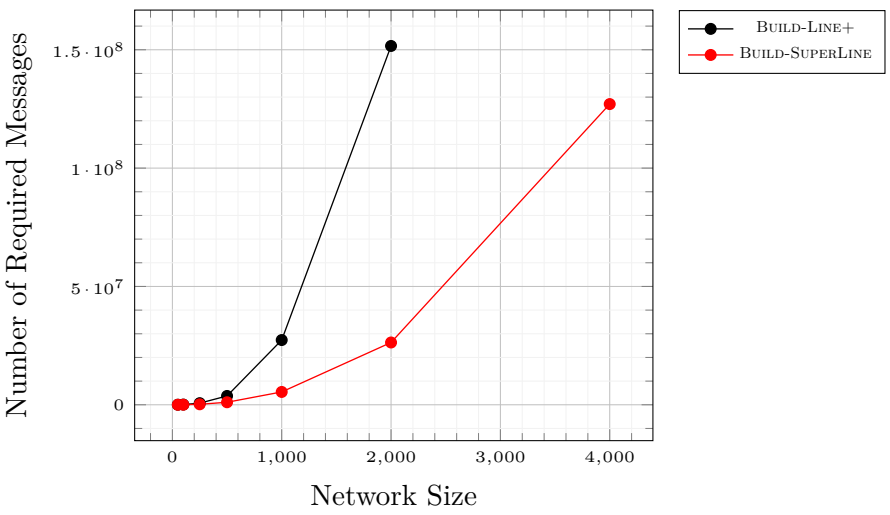



Figure 9.5.: Required Messages Comparison

messages. In fact, the higher the search rate is, the lower the stabilization time becomes. We have no reasonable explanation from an algorithmic point of view why search messages have a positive effect on the stabilization time. Our predicted effect is that search messages of Greedy-Search have a lower influence on Build-SuperLine than Search+ messages on Build-Line+. The most reasonable explanation for the observed effect we can offer is the following. Since Greedy-Search is a very lightweight protocol it incurs almost no additional overhead. Moreover, whenever Greedy-Search succeeds or fails it creates a new implicit edge from the current node to the source of the message. This effectively creates new edges in the topology that would not be available without searching. Moreover, these additional edges do not incur the overhead as they would do in Build-Line+, since Build-SuperLine does not have to delegate them.

### 9.3.5. Searchability

We want to conclude this section by comparing Build-Line+ and Build-SuperLine in terms of searchability. Since search requests are always answered correctly in the stable topology, we are only interested in the search requests that are initiated while stabilization is still in progress. Additionally, we are not concerned with search requests for IDs that do not belong to nodes in the

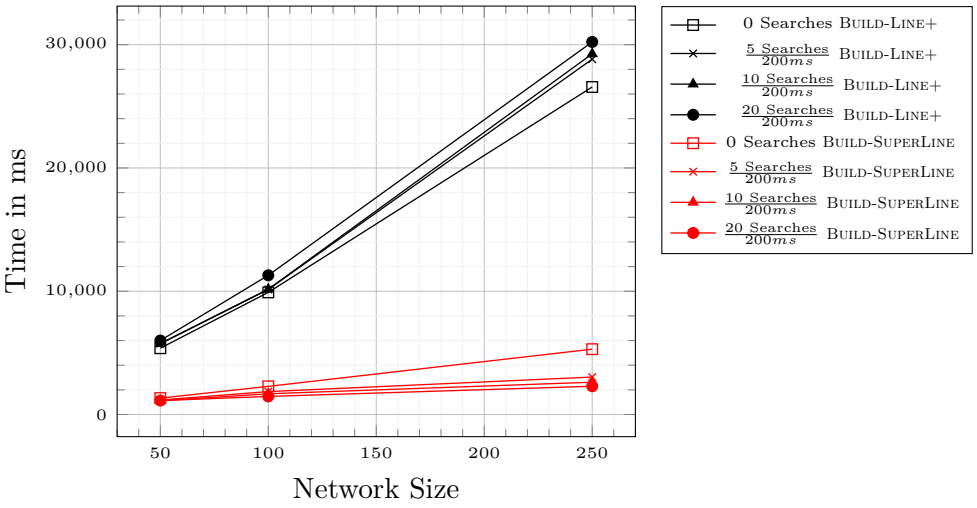



Figure 9.6.: Influence of Search-Rates on Stabilization Time

network, since they are always bound to fail. We picked a message rate of $\frac{5 \text{ Searches}}{200ms}$. The results of the experiments are presented in Figure 9.7. In general, Build-Line+ (together with Search+) and Build-SuperLine (together with Greedy-Search) perform extraordinarily well: both achieve more than 92% of successfully answered searches. This percentage value increases with network size. This effect is most likely caused by the causality between network size and stabilization time: i.e., a longer stabilization time implies a longer time frame to route search requests (which is helpful especially for Search+) and, additionally, more initiated searches, which in turn can be answered successfully in already stable parts of the network.

Interestingly, Build-Line+ outperforms Build-SuperLine in every instance. This seems to be contradictory to the basic principle that Build-SuperLine never delegates edges. One might expect that Build-SuperLine has more successful searches since the number of edges only increases in the topology. Here, the differences between the search protocols make a huge difference. Search+ visits every possible node on path between its source and the target (i.e., at every node the probe is forwarded to the node closest to the source), whereas Greedy-Search greedily selects nodes on the path. As a consequence, Search+ successfully finds the target if there is a directed path of explicit edges from source to target. However, for Greedy-Search

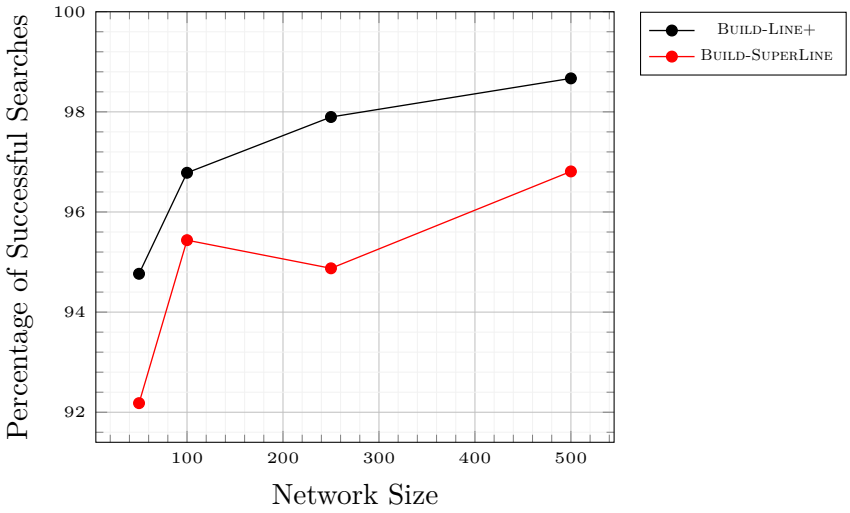



Figure 9.7.: Successful Search Comparison with $\frac{5 \text{ Searches}}{200ms}$

this statement does not hold, especially in the early stage of the stabilization process: i.e., even though there is a path of explicit edges, Greedy-Search does not find the target since it does not pick the path. We conjecture that a combination of Build-SuperLine with Search+ would yield the highest ratio of successful searches. Of course, this would be traded-off by a large message size for search messages since each node in the super-line topology has potentially a lot of neighbors that are included in the *Next* set of the Search+ messages.

# CHAPTER 10

## Conclusion of Part II

> *We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology.*
>
> Carl Sagan; Astronomer and Popular Science Writer

Similarly to Chapter 6, w conclude the second part of this thesis (and thus also the thesis as a whole) by subsuming the technical results of this part. Throughout the second part we focused on the problem of monotonic searchability for self-stabilizing overlay topologies. In Chapter 8 we introduced the specific problem statement and identified restrictions concerning the admissibility of messages in the system. Additionally, we introduced and investigated four primitives for overlay edge manipulation which can be used to show the preservation of weak connectivity in our protocols. Moreover, they possess many interesting properties (especially their universality) which might make them of independent interest even outside the self-stabilization community.

In Chapter 9 we presented our main technical results of this part of the thesis: a self-stabilizing protocol for the strict line topology and a self-stabilizing protocol for the super-line topology, both of which satisfy monotonic searchability.

The main difference between the two topologies is the way edges that are not part of the line are handled. The protocol for the strict line aims to get rid of them, whereas the protocol for the super-line simply keeps them in place. Both protocols work provably correctly: i.e., both stabilize to the desired topology and maintain monotonic searchability during the stabilization process. For the super-line, we also introduced a modification that allows for provably short routing paths in expectation. Finally, we compared the performance of both topologies and their respective protocols in experiments.

The remainder of this chapter is dedicated to further results and future work. In Section 10.1 we explain some of our results concerning monotonic searchability, which are not part of this thesis. These findings show that one can combine the universality of the primitives with the notion of monotonic searchability to achieve a more generic approach. Section 10.2 considers different directions for future results in the field.

## 10.1. Further Results

In this section we want to emphasize one additional result of ours concerning monotonic searchability. Additionally, we will briefly highlight a result that combines a protocol that maintains monotonic searchability with a protocol for self-stabilizing node departures. Both results are not included in the technical parts of this thesis. However, they are closely related and thus might give the reader a more thorough understanding of the topic as a whole.

The closest result to the work presented in this thesis is our **universal framework** for monotonic searchability [SSS16]. This framework takes an existing self-stabilizing overlay protocol and transforms it into a protocol that guarantees monotonic searchability. This transformation heavily relies on the primitives presented in Section 8.3. The approach has some specific requirements that the original protocol has to fulfill in order to be applicable. Informally speaking, the original protocol should monotonically converge to its desired topology (i.e., a node always keeps an edge that is part of the desired topology and edges that are not part of the topology are removed over time). Moreover, the action of nodes have to be deterministic and a reference of a node is always routed along a fixed path in the topology. Finally, all implicit edges have to merge eventually with explicit edges in legitimate

states. Most of these requirements are natural for existing self-stabilizing protocols. Nevertheless, they present a restriction concerning the universality of the framework. It has to be stated, that this framework drastically slows down the stabilization time, since a single use of the delegation primitive is substituted by a more complex operation. Additionally, the generic routing protocol that the framework imposes is very inefficient in terms of time and space complexity. Similar to our approach of Section 9.1 it uses a node set to route to the target node: i.e., every possible node on the path is visited and the routing paths as well as the message sizes can be $\mathcal{O}(n)$.

In [SSS15] we established a protocol that combines the Build-Line+ protocol of Section 9.1 with a protocol that solves the **finite departure problem**. In the finite departure problem, nodes want to leave the network while maintaining connectivity of the network. This is in itself a challenging task in a self-stabilization scenario: i.e., it has been shown that in general the departure problem cannot be solved without using an oracle that provides non-local information to the nodes (see [For+14]). By combining the monotonic searchability techniques with the protocol to solve the finite departure problem, we achieve a self-stabilizing protocol that: (i) makes sure that nodes which want to leave the network can do so without endangering connectivity, (ii) builds the line topology, and (iii) maintains monotonic searchability for the line topology.

## 10.2. Future Work

As already mentioned in the last section there already exists a universal protocol for monotonic searchability. However, the topic of monotonic searchability is far from being exhausted and there are various possibilities for future research.

First of all, it would be very interesting to formally study the impact of monotonic searchability maintenance on the standard complexity measures of distributed protocols: i.e., runtime and memory consumption. So far the focus has been on the feasibility of monotonic searchability. An efficiency evaluation has been out of scope. In case such an analysis is done and once the existing approaches have been thoroughly analyzed, one could even try to improve our protocols concerning these measures in a second step.

Moreover, the universal framework for monotonic searchability suffers from several disadvantages: e.g, the already mentioned long routing paths for search

messages. Therefore, it is still highly desirable to study monotonic searchability maintaining protocols for topologies that allow for short routing paths, e.g., De-Bruijn graphs, skip graphs or other hypercubic graphs.

Finally, one could also envision taking the general idea of monotonic searchability and use it to monotonically maintain other overlay properties. One example for this idea is a self-stabilizing protocol that monotonically tracks the maximum eccentricity of vertices: i.e., each node keeps track of the greatest shortest path distance to all other nodes. Once a node has a correct value for its maximum eccentricity, the value stays correct while the topology is still stabilizing.

# Bibliography


[Adl94] L. M. Adleman. "Molecular Computation of Solutions to Combinatorial Problems". In: *Science* 266.11 (1994), pp. 1021–1024. DOI: 10.1126/science.7973651.

[AGM13] C. Agathangelou, C. Georgiou, and M. Mavronicolas. "A distributed algorithm for gathering many fat mobile robots in the plane". In: *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. 2013, pp. 250–259. DOI: 10.1145/2484239.2484266.

[AK08] Z. Abel and S. D. Kominers. "Pushing Hypercubes Around". In: *CoRR* abs/0802.3414 (2008).

[Ang+05] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. "Fast construction of overlay networks". In: *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. 2005, pp. 145–154. DOI: 10.1145/1073970.1073991.

[Ang+06] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. "Computation in networks of passively mobile finite-state sensors". In: *Distributed Computing* 18.4 (2006), pp. 235–253. DOI: 10.1007/s00446-005-0138-3.

[App11] V. Appleton. *Tom Swift and His Electric Rifle; or, Daring Adventures in Elephant Land*. Grosset and Dunlap, 1911.

[AR10] D. Arbuckle and A. A. G. Requicha. "Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations". In: *Auton. Robots* 28.2 (2010), pp. 197–211. DOI: 10.1007/s10514-009-9162-7.

[AS07] J. Aspnes and G. Shah. "Skip graphs". In: *ACM Transactions on Algorithms* 3.4 (2007), p. 37. DOI: 10.1145/1290672.1290674.

[AW07] J. Aspnes and Y. Wu. "O(logn)-Time Overlay Network Construction from Graphs with Out-Degree 1". In: *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*. 2007, pp. 286–300. DOI: 10.1007/978-3-540-77096-1_21.

[Ber89] T. Berners-Lee. *Information Management: A Proposal*. http://www.w3.org/History/1989/proposal.html. 1989.

[BGP13] A. Berns, S. Ghosh, and S. V. Pemmaraju. "Building self-stabilizing overlay networks with the transitive closure framework". In: *Theoretical Computer Science* 512 (2013), pp. 2–14. DOI: 10.1016/j.tcs.2013.02.021.

[Blá+12] L. Blázovics, K. Csorba, B. Forstner, and H. Charaf. "Target Tracking and Surrounding with Swarm Robots". In: *IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS 2012, Novi Sad, Serbia, April 11-13, 2012*. 2012, pp. 135–141. DOI: 10.1109/ECBS.2012.41.

[BLF12] L. Blázovics, T. Lukovszki, and B. Forstner. "Target Surrounding Solution for Swarm Robots". In: *Information and Communication Technologies - 18th EUNICE/ IFIP WG 6.2, 6.6 International Conference, EUNICE 2012, Budapest, Hungary, August 29-31, 2012. Proceedings*. 2012, pp. 251–262. DOI: 10.1007/978-3-642-32808-4_23.

[BMV12] V. Bonifaci, K. Mehlhorn, and G. Varma. "Physarum can compute shortest paths". In: *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*. 2012, pp. 233–240.

[Bon+96] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. "On the Computational Power of DNA". In: *Discrete Applied Mathematics* 71.1-3 (1996), pp. 79–94. DOI: 10.1016/S0166-218X(96)00058-3.

[Bra+13] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1 (2013), pp. 1–41. DOI: 10.1007/s11721-012-0075-2.

[BS00] J. Brzezinski and M. Szychowiak. "Self-Stabilization in Distributed Systems - a Short Survey". In: *Foundations of Computing and Decision Sciences* 25.1 (2000), pp. 3–22.

[BS13] J. C. Barca and Y. A. Sekercioglu. "Swarm robotics reviewed". In: *Robotica* 31.3 (2013), pp. 345–359. DOI: 10.1017/S026357471200032X.

[Bui+07] A. Bui, A. K. Datta, F. Petit, and V. Villain. "Snap-stabilization and PIF in tree networks". In: *Distributed Computing* 20.1 (2007), pp. 3–19. DOI: 10.1007/s00446-007-0030-4.

[But+04] Z. J. Butler, K. Kotay, D. Rus, and K. Tomita. "Generic Decentralized Control for Lattice-Based Self-Reconfigurable Robots". In: *International Journal of Robotics Research* 23.9 (2004), pp. 919–937. DOI: 10.1177/0278364904044409.

[Can+16] S. Cannon, J. J. Daymude, D. Randall, and A. W. Richa. "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, Chicago, IL, USA, July 25-28, 2016*. 2016, pp. 279–288. DOI: 10.1145/2933057.2933107.

[Cas+11] C. E. Castro, F. Kilchherr, D.-N. Kim, E. L. Shiao, T. Wauer, P. Wortmann, M. Bathe, and H. Dietz. "A primer to scaffolded DNA origami". In: *Nature Methods* 8 (2011), pp. 221–229. DOI: 10.1038/nmeth.1570.

[CD06] A. Chavoya and Y. Duthen. "Using a genetic algorithm to evolve cellular automata for 2D/3D computational development". In: *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*. 2006, pp. 231–232. DOI: 10.1145/1143997.1144036.

[CF05] C. Cramer and T. Fuhrmann. *Self-stabilizing ring networks on connected graphs*. Technical Report. Institut fuer Telematik (TM, 2005.

[CFL08] A. Chaintreau, P. Fraigniaud, and E. Lebhar. "Networks Become Navigable as Nodes Move and Forget". In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*. 2008, pp. 133–144. DOI: 10.1007/978-3-540-70575-8_12.

[Cha+16] A. R. Chandrasekaran, N. Anderson, M. Kizer, K. Halvorsen, and X. Wang. "Beyond the Fold: Emerging Biological Applications of DNA Origami". In: *ChemBioChem* 17.12 (2016), pp. 1081–1089. DOI: 10.1002/cbic.201600038.

[Cha09] B. Chazelle. "Natural algorithms". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*. 2009, pp. 422–431.

[Che+14] H. Chen, D. Doty, D. Holden, C. Thachuk, D. Woods, and C. Yang. "Fast Algorithmic Self-assembly of Simple Shapes Using Random Agitation". In: *DNA Computing and Molecular Programming - 20th International Conference, DNA 20, Kyoto, Japan, September 22-26, 2014. Proceedings*. 2014, pp. 20–36. DOI: 10.1007/978-3-319-11295-4_2.

[Chi94] G. S. Chirikjian. "Kinematics of a Metamorphic Robotic System". In: *Proceedings of the 1994 International Conference on Robotics and Automation, San Diego, CA, USA, May 1994*. 1994, pp. 449–455. DOI: 10.1109/ROBOT.1994.351256.

[Cie+12] M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. "Distributed Computing by Mobile Robots: Gathering". In: *SIAM Journal on Computing* 41.4 (2012), pp. 829–879. DOI: 10.1137/100796534.

[Cla45] A. C. Clarke. "Extra-Terrestrial Relays". In: *Wireless World* (Oct. 1945), pp. 305–308.

[Clé+08] J. Clément, T. Hérault, S. Messika, and O. Peres. "On the Complexity of a Self-Stabilizing Spanning Tree Algorithm for Large Scale Systems". In: *14th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2008, 15-17 December 2008, Taipei, Taiwan*. 2008, pp. 48–55. DOI: 10.1109/PRDC.2008.36.

[CXW15] M. Chen, D. Xin, and D. Woods. "Parallel computation using active self-assembly". In: *Natural Computing* 14.2 (2015), pp. 225–250. DOI: 10.1007/s11047-014-9432-y.

[Das+10] S. Das, P. Flocchini, N. Santoro, and M. Yamashita. "On the computational power of oblivious robots: forming a series of geometric patterns". In: *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. 2010, pp. 267–276. DOI: 10.1145/1835698.1835761.

[Day+17] J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Leader Election with High Probability for Self-Organizing Programmable Matter". In: *CoRR* (2017).

[Day+ar] J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "On the Runtime of Universal Coating for Programmable Matter". In: *Natural Computing* (to appear).

[DD05] A. Deutsch and S. Dormann. *Cellular Automaton Modeling of Biological Pattern Formation*. 1st ed. Birkhäuser Basel, 2005. DOI: 10.1007/b138451.

[Del+10] S. Delaët, S. Devismes, M. Nesterenko, and S. Tixeuil. "Snap-stabilization in message-passing systems". In: *Journal of Parallel and Distributed Computing* 70.12 (2010), pp. 1220–1230. DOI: 10.1016/j.jpdc.2010.04.002.

[Dem+11] E. D. Demaine, M. J. Patitz, R. T. Schweller, and S. M. Summers. "Self-Assembly of Arbitrary Shapes Using RNAse Enzymes: Meeting the Kolmogorov Bound with Small Scale Factor (extended abstract)". In: *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*. 2011, pp. 201–212. DOI: 10.4230/LIPIcs.STACS.2011.201.

[Der+14] Z. Derakhshandeh, S. Dolev, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Brief announcement: amoebot - a new model for programmable matter". In: *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*. 2014, pp. 220–222. DOI: 10.1145/2612669.2612712.

[Der+15a] Z. Derakhshandeh, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems". In: *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication, NANOCOM' 15, Boston, MA, USA, September 21-22, 2015*. 2015, 21:1–21:2. DOI: 10.1145/2800795.2800829.

[Der+15b] Z. Derakhshandeh, R. Gmyr, T. Strothmann, R. A. Bazzi, A. W. Richa, and C. Scheideler. "Leader Election and Shape Formation with Self-organizing Programmable Matter". In: *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*. 2015, pp. 117–132. DOI: 10.1007/978-3-319-21999-8_8.

[Der+16a] Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa, C. Scheideler, and T. Strothmann. "On the Runtime of Universal Coating for Programmable Matter". In: *DNA Computing and Molecular Programming - 22nd International Conference, DNA 22, Munich, Germany, September 4-8, 2016, Proceedings*. 2016, pp. 148–164. DOI: 10.1007/978-3-319-43994-5_10.

[Der+16b] Z. Derakhshandeh, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Universal Shape Formation for Programmable Matter". In: *Proceedings of the 28th ACM Symposium on Par-*

*allelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 2016, pp. 289–299. DOI: 10.1145/2935764.2935784.

[Der+17] Z. Derakhshandeh, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Universal Coating for Programmable Matter". In: *Theoretical Computer Science* 671 (2017), pp. 56–68. DOI: 10.1016/j.tcs.2016.02.039.

[DH97] S. Dolev and T. Herman. "Superstabilizing Protocols for Dynamic Distributed Systems". In: *Chicago Journal of Theoretical Computer Science* 1997 (1997).

[Dij74] E. W. Dijkstra. "Self-stabilizing Systems in Spite of Distributed Control". In: *Communications of the ACM* 17.11 (1974), pp. 643–644. DOI: 10.1145/361179.361202.

[DK02] M. J. Daley and L. Kari. "DNA Computing: Models and Implementations". In: *Comments on Theoretical Biology* 7 (2002), pp. 177–198. DOI: 10.1080/08948550290022123.

[DK08] S. Dolev and R. I. Kat. "HyperTree for self-stabilizing peer-to-peer systems". In: *Distributed Computing* 20.5 (2008), pp. 375–388. DOI: 10.1007/s00446-007-0038-9.

[Dol00] S. Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN: 0-262-04178-2.

[Dot12] D. Doty. "Theory of algorithmic self-assembly". In: *Communications of the ACM* 55.12 (2012), pp. 78–88. DOI: 10.1145/2380656.2380675.

[DP06] A. Dumitrescu and J. Pach. "Pushing Squares Around". In: *Graphs and Combinatorics* 22.1 (2006), pp. 37–50. DOI: 10.1007/s00373-005-0640-1.

[DT11] S. Dubois and S. Tixeuil. "A Taxonomy of Daemons in Self-stabilization". In: *ArXiV* abs/1110.0334 (2011).

[DT13] S. Dolev and N. Tzachar. "Spanders: Distributed spanning expanders". In: *Science of Computer Programming* 78.5 (2013), pp. 544–555. DOI: 10.1016/j.scico.2012.10.001.

[DW14] M. Dietzfelbinger and P. Woelfel. "Tight Lower Bounds for Greedy Routing in Higher-Dimensional Small-World Grids". In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. 2014, pp. 816–829. DOI: 10.1137/1.9781611973402.60.

[EK10] D. A. Easley and J. M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge University Press, 2010. ISBN: 978-0-521-19533-1.

[Flo+08] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. "Arbitrary pattern formation by asynchronous, anonymous, oblivious robots". In: *Theoretical Computer Science* 407.1-3 (2008), pp. 412–447. DOI: 10.1016/j.tcs.2008.07.026.

[Flo+13] P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. "Computing Without Communicating: Ring Exploration by Asynchronous Oblivious Robots". In: *Algorithmica* 65.3 (2013), pp. 562–583. DOI: 10.1007/s00453-011-9611-5.

[For+14] D. Foreback, A. Koutsopoulos, M. Nesterenko, C. Scheideler, and T. Strothmann. "On Stabilizing Departures in Overlay Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*. 2014, pp. 48–62. DOI: 10.1007/978-3-319-11764-5_4.

[Fuk+88] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss. "Self Organizing Robots Based on Cell Structures - CEBOT". In: *Proceedings of IROS '88*. 1988, pp. 145–150. DOI: 10.1109/IROS.1988.592421.

[Gal+14] D. Gall, R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. "A Note on the Parallel Runtime of Self-Stabilizing Graph Linearization". In: *Theoretical Computer Science* 55.1 (2014), pp. 110–135. DOI: 10.1007/s00224-013-9504-x.

[GK10] N. Guellati and H. Kheddouci. "A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs". In: *Journal of Parallel and Distributed Computing* 70.4 (2010), pp. 406–415. DOI: 10.1016/j.jpdc.2009.11.006.

[Gmy+17] R. Gmyr, I. Kostitsyna, F. Kuhn, C. Scheideler, and T. Strothmann. "Forming Tile Shapes with a Single Robot". Presented at the 33st European Workshop on Computational Geometry (EuroCG 2017). Apr. 2017.

[Hér+06] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. "Brief Announcement: Self-stabilizing Spanning Tree Algorithm for Large Scale Systems". In: *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*. 2006, pp. 574–575. DOI: 10.1007/978-3-540-49823-0_44.

[Hur+15] F. Hurtado, E. Molina, S. Ramaswami, and V. S. Adinolfi. "Distributed reconfiguration of 2D lattice-based modular robotic systems". In: *Autonomous Robots* 38.4 (2015), pp. 383–413. DOI: 10.1007/s10514-015-9421-8.

[IHM02] K. Imai, T. Hori, and K. Morita. "Self-Reproduction in Three-Dimensional Reversible Cellular Space". In: *Artificial Life* 8.2 (2002), pp. 155–174. DOI: 10.1162/106454602320184220.

[Ima+03] K. Imai, Y. Kasai, Y. Sonoyama, C. Iwamoto, and K. Morita. "Self-reproduction and shape formation in two and three dimensional cellular automata with conservative constraints". In: *Proceedings of the Eighth International Symposium on Artificial Life and Robotics*. 2003, pp. 377–380.

[Jac+] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. "SKIP$^+$: A Self-Stabilizing Skip Graph". In: *Journal of the ACM* 61.6 (), 36:1–3. DOI: 10.1145/2629695.

[Jac+12] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid. "Towards higher-dimensional topological self-stabilization: A distributed algorithm for Delaunay graphs". In: *Theoretical Computer Science* 457 (2012), pp. 137–148. DOI: 10.1016/j.tcs.2012.07.029.

[JM14] C. Johnen and F. Mekhaldi. "Self-stabilizing with service guarantee construction of 1-hop weight-based bounded size clusters". In: *Journal of Parallel and Distributed Computing* 74.1 (2014), pp. 1900–1913. DOI: 10.1016/j.jpdc.2013.09.004.

[Kar+97] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[KB14] G. P. Kumar and S. Berman. "Statistical analysis of stochastic multi-robot boundary coverage". In: *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*. 2014, pp. 74–81. DOI: 10.1109/ICRA.2014.6906592.

[Ker12] S. Kernbach. *Handbook of Collective Robotics – Fundamentals and Challanges*. Vol. 1. Pan Stanford Publishing, 2012.

[KK07] S. Kamei and H. Kakugawa. "A Self-Stabilizing Distributed Approximation Algorithm for the Minimum Connected Dominating Set". In: *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370464.

[KK10] A. Kuzuya and M. Komiyama. "DNA origami: Fold, stick, and beyond". In: *Nanoscale* 2 (3 2010), pp. 309–321. DOI: 10.1039/B9NR00246D.

[KKS12] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "A Self-Stabilization Process for Small-World Networks". In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. 2012, pp. 1261–1271. DOI: 10.1109/IPDPS.2012.115.

[KKS14] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "Re-Chord: A Self-stabilizing Chord Overlay Network". In: *Theoretical Computer Science* 55.3 (2014), pp. 591–612. DOI: 10.1007/s00224-012-9431-2.

[Kle00] J. M. Kleinberg. "The small-world phenomenon: an algorithmic perspective". In: *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*. 2000, pp. 163–170. DOI: 10.1145/335305.335325.

[Kou16] A. Koutsopoulos. "Dynamics and efficiency in topological self-stabilization". PhD thesis. University of Paderborn, 2016. DOI: urn:nbn:de:hbz:466:2-24163.

[KSS15] A. Koutsopoulos, C. Scheideler, and T. Strothmann. "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*. 2015, pp. 201–216. DOI: 10.1007/978-3-319-21741-3_14.

[KSS16] A. Koutsopoulos, C. Scheideler, and T. Strothmann. "Towards a Universal Approach for the Finite Departure Problem in Overlay Networks". In: *Information and Computation* (2016). DOI: 10.1016/j.ic.2016.12.006.

[Mcl08] J. D. Mclurkin IV. "Analysis and Implementation of Distributed Algorithms for Multi-Robot Systems". PhD thesis. Massachusetts Institute of Technology, 2008.

[Mic15] O. Michail. "Terminating Distributed Construction of Shapes and Patterns in a Fair Solution of Automata". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*. 2015, pp. 37–46. DOI: 10.1145/2767386.2767402.

[MS16] O. Michail and P. G. Spirakis. "Simple and efficient local codes for distributed stable network construction". In: *Distributed Computing* 29.3 (2016), pp. 207–237. DOI: 10.1007/s00446-015-0257-4.

[Nan+10] J. Nangreave, D. Han, Y. Liu, and H. Yan. "DNA origami: a history and current perspective". In: *Current Opinion in Chemical Biology* 14.5 (2010). Nanotechnology and Miniaturization/Mechanisms, pp. 608–615. DOI: 10.1016/j.cbpa.2010.06.182.

[Neu66] J. von Neumann. *Theory of Self-Reproducing Automata*. Ed. by A. W. Burks. University of Illinois Press, 1966.

[NM12] I. Navarro and F. Matía. "An introduction to swarm robotics". In: *ISRN Robotics*. Hindawi Publishing Corporation, 2012, p. 10. DOI: 10.5402/2013/608164.

[NNS13] R. M. Nor, M. Nesterenko, and C. Scheideler. "Corona: A stabilizing deterministic message-passing skip list". In: *Theoretical Computer Science* 512 (2013), pp. 119–129. DOI: 10.1016/j.tcs.2012.08.029.

[ORS07] M. Onus, A. W. Richa, and C. Scheideler. "Linearization: Locally Self-Stabilizing Sorting in Graphs". In: *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. 2007. DOI: 10.1137/1.9781611972870.10.

[Pat14] M. J. Patitz. "An introduction to tile-based self-assembly and a survey of recent results". In: *Natural Computing* 13.2 (2014), pp. 195–224. DOI: 10.1007/s11047-013-9379-4.

[Pav+13] T. P. Pavlic, S. Wilson, G. P. Kumar, and S. Berman. "An Enzyme-Inspired Approach to Stochastic Allocation of Robotic Swarms Around Boundaries". In: *Robotics Research - The 16th International Symposium ISRR, 16-19 December 2013, Singapore*. 2013, pp. 631–647. DOI: 10.1007/978-3-319-28872-7_36.

[Pis97] N. Pisanti. *A Survey on DNA Computing*. Tech. rep. University of Pisa, 1997.

[Pug90] W. Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676. DOI: 10.1145/78973.78977.

[RCN14] M. Rubenstein, A. Cornejo, and R. Nagpal. "Programmable self-assembly in a thousand-robot swarm". In: *Science* 345.6198 (2014), pp. 795–799. DOI: 10.1126/science.1254295.

[Rot06] P. W. K. Rothemund. "Folding DNA to create nanoscale shapes and patterns". In: *Nature* 440.7082 (2006), pp. 297–302. DOI: 10.1038/nature04586.

[RS10] M. Rubenstein and W. Shen. "Automatic scalable size selection for the shape of a distributed robotic collective". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*. 2010, pp. 508–513. DOI: 10.1109/IROS.2010.5650906.

[RSS11] A. W. Richa, C. Scheideler, and P. Stevens. "Self-Stabilizing De Bruijn Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. 2011, pp. 416–430. DOI: 10.1007/978-3-642-24550-3_31.

[Sch11] J. L. Schiff. *Cellular automata: a discrete view of the world*. John Wiley & Sons, 2011.

[SR05] A. Shaker and D. S. Reeves. "Self-Stabilizing Structured Ring Topology P2P Systems". In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August - 2 September 2005, Konstanz, Germany*. 2005, pp. 39–46. DOI: 10.1109/P2P.2005.34.

[SSS15] C. Scheideler, A. Setzer, and T. Strothmann. "Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures". In: *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*. 2015, 24:1–24:17. DOI: 10.4230/LIPIcs.OPODIS.2015.24.

[SSS16] C. Scheideler, A. Setzer, and T. Strothmann. "Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 2016, pp. 71–84. DOI: 10.1007/978-3-662-53426-7_6.

[Str15] T. Strothmann. "The Impact of Communication Patterns on Distributed Self-Adjusting Binary Search Trees". In: *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*. 2015, pp. 175–186. DOI: 10.1007/978-3-319-15612-5_16.

[Str16] T. Strothmann. "The Impact of Communication Patterns on Distributed Self-Adjusting Binary Search Tree". In: *Journal of Graph Algorithms and Applications* 20.1 (2016), pp. 79–100. DOI: 10.7155/jgaa.00385.

[SW07] D. Soloveichik and E. Winfree. "Complexity of Self-Assembled Shapes". In: *SIAM Journal on Computing* 36.6 (2007), pp. 1544–1569. DOI: 10.1137/S0097539704446712.

[SW15] N. Schiefer and E. Winfree. "Universal Computation and Optimal Construction in the Chemical Reaction Network-Controlled Tile Assembly Model". In: *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*. 2015, pp. 34–54. DOI: 10.1007/978-3-319-21999-8_3.

[TAB16] N. Telecommunications, I. Administration, and U. C. Bureau. *Most popular online activities of adult internet users in the United States as of July 2015*. Statista. Accessed 04.04.2017. Mar. 2016. URL: https://www.statista.com/statistics/183910/internet-activities-of-us-users/.

[Tan17] Y. Tan. "A Survey on Swarm Robotics". In: *Nature-Inspired Computing: Concepts, Methodologies, Tools, and Applications*. Ed. by I. R. M. Association. IGI Global, 2017. Chap. 36, pp. 956–998. DOI: 10.4018/978-1-5225-0788-8.ch036.

[TKN07] A. Tero, R. Kobayashi, and T. Nakagaki. "A mathematical model for adaptive transport network in path finding by true slime mold". In: *Journal of Theoretical Biology* 244.4 (2007), pp. 553–564. DOI: 10.1016/j.jtbi.2006.07.015.

[Ver70] J. Verne. *Vingt mille lieues sous les mers*. Pierre-Jules Hetzel, 1870.

[WB08] J. Watada and R. B. A. Bakar. "DNA Computing and Its Applications". In: *Eighth International Conference on Intelligent Systems Design and Applications, ISDA 2008, 26-28 November 2008, Kaohsiung, Taiwan, 3 Volumes*. 2008, pp. 288–294. DOI: 10.1109/ISDA.2008.362.

[Wil+14] S. Wilson, T. P. Pavlic, G. P. Kumar, A. Buffin, S. C. Pratt, and S. Berman. "Design of ant-inspired stochastic control policies for collective transport by robotic swarms". In: *Swarm Intelligence* 8.4 (2014), pp. 303–327. DOI: 10.1007/s11721-014-0100-8.

[Win+98] E. Winfree., F. Liu, L. A. Wenzler, and N. C. Seeman. "Design and self-assembly of two-dimensional DNA crystals". In: *Nature* 394.6693 (1998), pp. 539–544. DOI: 10.1038/28998.

[Win98] E. Winfree. "Algorithmic self-assembly of DNA". PhD thesis. California Institute of Technology, 1998.

[Wol02] S. Wolfram. *A New Kind of Science*. Wolfram Media Inc., 2002. ISBN: 1-57955-008-8.

[Wol86] S. Wolfram. *Theory and Applications of Cellular Automata*. Ed. by S. Wolfram. World Scientific, 1986.

[Woo+13] D. Woods, H. Chen, S. Goodfriend, N. Dabby, E. Winfree, and P. Yin. "Active self-assembly of algorithmic shapes and patterns in polylogarithmic time". In: *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*. 2013, pp. 353–354. DOI: 10.1145/2422436.2422476.

[Woo13] D. Woods. "Intrinsic universality and the computational power of self-assembly". In: *Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013.* 2013, pp. 16–22. DOI: 10.4204/EPTCS.128.5.

[WP94] S. Wolfram and N. H. Packard. "Two-Dimensional Cellular Automata". In: *Cellular Automata and Complexity: Collected Papers*. Ed. by S. Wolfram. Avalon Publishing, 1994. Chap. 6, pp. 211–249.

[WWA04] J. E. Walter, J. L. Welch, and N. M. Amato. "Distributed reconfiguration of metamorphic robot chains". In: *Distributed Computing* 17.2 (2004), pp. 171–189. DOI: 10.1007/s00446-003-0103-y.

[Yim+07] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. "Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]". In: *IEEE Robotics & Automation Magazine* 14.1 (2007), pp. 43–52. DOI: 10.1109/MRA.2007.339623.

[YT10] Y. Yamauchi and S. Tixeuil. "Monotonic Stabilization". In: *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*. 2010, pp. 475–490. DOI: 10.1007/978-3-642-17653-1_34.

[YY05] X.-S. Yang and Y.-N. Young. "Cellular Automata, PDEs, and Pattern Formation". In: *Handbook of Bioinspired Algorithms and Applications*. Ed. by S. Olariu and A. Y. Zomaya. 1st ed. Chapman & Hall/CRC Press, 2005. Chap. 18, pp. 271–282. DOI: 10.1201/9781420035063.ch18.