Distributed Algorithms for Overlay Networks and Programmable Matter

Dissertation

In partial fulfillment of the requirements for the academic degree
Doctor rerum naturalium (Dr. rer. nat.)

Faculty of Computer Science,
Electrical Engineering and Mathematics
Department of Computer Science
Research Group Theory of Distributed Systems

Robert Gmyr

PADERBORN
UNIVERSITY

Reviewers: Christian Scheideler, Paderborn University
Friedhelm Meyer auf der Heide, Paderborn University
Andréa W. Richa, Arizona State University

Abstract

This dissertation consists of two parts that are dedicated to the study of distributed algorithms for overlay networks and programmable matter.

The first part revolves around the topics of robustness against attacks, recovery from faults, and monitoring network properties in the context of overlay networks. More specifically, we introduce network protocols that maintain the connectivity of an overlay network under massive adversarial churn or denial-of-service attacks, we present a self-stabilizing algorithm for the construction of metric graphs, and we initiate the study of hybrid networks by investigating the problem of continuously monitoring properties of an externally-controlled network with the help of an overlay network.

In the second part we investigate the algorithmic foundations of programmable matter. Programmable matter refers to a substance that can change its shape or other physical properties in a programmable fashion. We envision programmable matter consisting of simple computational devices that are able to self-organize in order to achieve a collective goal without any central control or external intervention. We present efficient algorithms for the fundamental problems of leader election and shape formation for programmable matter.

Zusammenfassung

Diese zweiteilige Dissertation widmet sich der Entwicklung und Analyse verteilter Algorithmen für Overlay-Netzwerke und programmierbare Materie.

Der erste Teil besteht aus drei Gruppen von Resultaten, welche sich jeweils auf die Themen der Robustheit, der Fehlertoleranz und der Überwachung von Netzwerken konzentrieren: Zunächst stellen wir Netzwerk-Protokolle vor, welche den Zusammenhang eines Netzwerks unter massivem gegnerischen Churn oder Denial-of-Service-Attacken aufrecht erhalten. Anschließend präsentieren wir einen selbststabilisierenden Algorithmus zur Konstruktion metrischer Graphen. Zuletzt führen wir das Konzept hybrider Netzwerke ein und betrachten eine Reihe von Problemen, in denen Eigenschaften eines dynamischen Netzwerks mit Hilfe eines Overlay-Netzwerks überwacht werden sollen.

Im zweiten Teil untersuchen wir die algorithmischen Grundlagen programmierbarer Materie. Programmierbare Materie bezeichnet eine Substanz, die ihre Form oder andere physikalische Eigenschaften auf programmierbare Art und Weise verändern kann. Wir betrachten programmierbare Materie, die aus einer Vielzahl gleichartiger einfacher Einheiten besteht. Die Einheiten verfolgen selbstorganisierend ein gemeinsames Ziel. Dabei unterliegen sie keiner zentralen Kontrolle, sondern agieren vollständig verteilt. Wir stellen effiziente Algorithmen für das Leader-Election-Problem und das Shape-Formation-Problem im Kontext programmierbarer Materie vor.

Acknowledgments

First and foremost I would like to thank my advisor Christian Scheideler for introducing me to a wealth of fascinating topics in distributed computing and for guiding me on the way towards this thesis. I would also like to thank Andréa Richa and Friedhelm Meyer auf der Heide for serving as reviewers for my thesis and for supporting me at various stages throughout my studies.

I thank all of my co-authors for countless interesting and fruitful discussions. I was tremendously fortunate to work with such creative, smart, and friendly people. Special thanks go to my co-author and friend Kristian Hinnenthal for proofreading every single page of this thesis I put in front of him. Furthermore, I want to thank all members, both past and current, of the Theory of Distributed Systems group at Paderborn University for making my time at the office so enjoyable. I will surely miss our daily lunch discussions on any subject imaginable.

Finally, I want to thank my family and friends for their support and patience over these past months. Ganz besonders möchte ich meinen Eltern danken. Ohne eure Unterstützung auf meinem gesamten Bildungsweg hätte es diese Dissertation nicht gegeben.

Thank you.

Contents

Chapter 1

Introduction

The theory of distributed computing is a diverse field of research that thrives on the investigation of a wide variety of models for distributed systems. It is only fitting, then, that this thesis revolves around two vastly different types of systems and models. The structure of this thesis reflects this fact in that it is subdivided into two independent parts.

In the first part we present distributed algorithms for *overlay networks*. The distinguishing feature of an overlay network is that it is *reconfigurable*, i.e., the topology of the network is not static but can be controlled by the network protocol. Large-scale and highly decentralized overlay networks have become increasingly popular since the rise of *peer-to-peer* systems. Well-known examples of peer-to-peer systems in practice are the file-sharing protocol *BitTorrent* and the digital currency *Bitcoin*, which relies on a distributed ledger called the *blockchain* to keep track of the transactions performed by its users. We investigate the topics of *robustness* and *recovery from faults* in the context of overlay networks, both of which are of critical importance when it comes to maintaining the availability of a distributed system or service. Beyond our results on pure overlay networks, we also initiate the study of *hybrid networks*, which are networks that combine an overlay network with an externally-controlled dynamic network.

The second part of this thesis is dedicated to the study of the algorithmic foundations of *programmable matter*. Programmable matter refers to a substance that can change its shape or other physical properties in a programmable fashion. We envision programmable matter consisting of simple computational devices that are able to self-organize in order to achieve a collective goal without any central control or external intervention. While some of the basic ideas behind programmable matter are already more than two decades old, rigorous algorithmic research on the subject is still quite sparse. Our goal is to contribute to the theoretical foundations of programmable matter by presenting efficient algorithms for fundamental problems in this domain.

Thesis Overview

The first part of this thesis consists of three chapters that revolve around the topics of robustness against attacks, recovery from faults, and monitoring network properties in the context of overlay networks. More specifically, Chapter 2 introduces network protocols that maintain the connectivity of an overlay network under massive adversarial churn or denial-of-service attacks, Chapter 3 presents a self-stabilizing algorithm for the construction of metric graphs, and Chapter 4 initiates the study of hybrid networks by investigating the problem of continuously monitoring properties of an externally-controlled network with the help of an overlay network. The chapters of this part are mostly self-contained and can be read in any order.

In the second part of this thesis we investigate fundamental problems in the area of programmable matter. Specifically, we consider the leader election problem in Chapter 5 and the shape formation problem in Chapter 6. In contrast to the first part, the chapters of the second part are not self-contained. For readers that are interested in our results on shape formation but want to skip the details of the leader election algorithm, we recommend to read Chapter 5 up to Section 5.3 before turning to Chapter 6.

In the following, we give a brief summary of the results presented in each chapter and list the publications on which the chapters are based.

Chapter 2: Churn- and DoS-Resistant Overlay Networks We begin the first part of this thesis by presenting two algorithms that *maintain the connectivity* of a network under *adversarial attacks*. The first algorithm organizes the nodes into an expander graph and maintains connectivity under *adversarial churn* by an omniscient adversary with a constant churn rate, i.e., the network remains connected even when nodes join and leave at a rapid rate. The second algorithm uses a network that is based on the hypercube and maintains connectivity under adversarial *denial-of-service attacks* (or *DoS-attacks*). A node under a DoS-attack is blocked from communicating with any other node. In a network of n nodes, the adversary is allowed to block $(1-\varepsilon) \cdot n$ nodes for any constant $0 < \varepsilon \leq 1$, and it can change the set of blocked nodes in every round. We assume the adversary to be *t-late* in that it only has access to topological information that is at least $t = \Theta(\log \log n)$ rounds old. Both algorithms gain their robustness by randomizing the topology of the overlay network at regular intervals. The algorithms use networks of polylogarithmic degree and require polylogarithmic communication work at each node in every round.

Underlying the algorithms is a novel node sampling technique for regular expander graphs and hypercubes that allows each node to sample a logarithmic number of nodes uniformly at random in $O(\log \log n)$ communication rounds. This technique is specific to overlay networks and its optimal running time represents an exponential improvement over known techniques. The results presented in this chapter have a wide range of applications, for example in the area of scalable and robust peer-to-peer systems. The chapter is based on the following publication.

> M. Drees, R. Gmyr, and C. Scheideler. “Churn- and DoS-resistant Overlay Networks Based on Network Reconfiguration”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, see [DGS16].

Chapter 3: Self-Stabilizing Metric Graphs While the goal of Chapter 2 is to *maintain* the topology of an overlay network (or at least its connectivity), the objective of Chapter 3 is to *recover* the topology from a corrupted state. Specifically, this chapter introduces an algorithm for the *self-stabilizing* construction of the graph corresponding to a given *metric* specified via a distance oracle. The graph corresponding to a metric (or just *metric graph*) is the unique minimal undirected graph such that for any pair of nodes the length of a shortest path between the nodes corresponds to the distance between the nodes according to the metric. To the best of our knowledge, our algorithm is the first self-stabilizing algorithm for the construction of general metric graphs. The algorithm works for both synchronous and asynchronous activations of the nodes. In the synchronous case, the algorithm constructs the metric graph in linear time and it guarantees that after stabilization the memory overhead and the number of messages sent and received per round at every node drop to a constant within a linear number of rounds. The chapter is based on the following publication and a corresponding journal article [GLS17] that will appear in Theory of Computing Systems.

> R. Gmyr, J. Lefèvre, and C. Scheideler. “Self-stabilizing Metric Graphs”. In: *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, see [GLS16].

Chapter 4: Hybrid Network Monitoring In the last chapter of the first part we broaden our focus beyond pure overlay networks by investigating a class of networks that we refer to as *hybrid networks*. In a hybrid network a set of nodes is connected by an *external network* and an *internal network*. While the external network cannot be controlled by the network algorithm and might be exposed to continuous change, the internal network is an overlay network that is fully under the control of the network algorithm. As an example, consider a set of wireless devices with access to the cell phone infrastructure that are dispersed over a limited area such as a city center. The devices can form a wireless ad-hoc network using their WiFi capabilities to establish an external network. The topology of this external network depends on the position of the devices and cannot be controlled by the network algorithm. At the same time, the devices can use the cell phone infrastructure to form an overlay network, which is an internal network that can be modified by the network algorithm.

We investigate the problem of continuously monitoring properties of the external network with the help of the internal network. We present scalable distributed algorithms that efficiently monitor the *number of edges*, the *average node degree*, the *clustering coefficient*, the *bipartiteness*, and the *weight of a minimum spanning tree.* Their performance bounds demonstrate that monitoring the external network with the help of an internal network can be done much more efficiently than by just using the external network. The chapter is based on the following publication.

> R. Gmyr, K. Hinnenthal, C. Scheideler, and C. Sohler. "Distributed Monitoring of Network Properties: The Power of Hybrid Networks". In: *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*, see [Gmy+17].

Chapter 5: Leader Election for Programmable Matter In Chapter 5, we leave the domain of overlay networks and shift our attention to the algorithmic foundations of programmable matter, which is the topic of the second part of this thesis. Our investigation of programmable matter is based on the *amoebot model*, which facilitates rigorous algorithmic research on programmable matter in the Euclidean plane. In the amoebot model, programmable matter consists of a uniform set of simple computational units called *particles* that can move and bond to other particles and that use their bonds to exchange information. The particles act asynchronously and achieve locomotion by expanding and contracting, which resembles the amoeboid movement performed by certain biological cells. To achieve a collective goal, the particles have to self-organize in a distributed fashion without any central control.

The topic of Chapter 5 is the *leader election problem*, which requires a set of particles to select one particle as its unique leader. Leader election is a central and classic problem in distributed computing. Many problems such as the consensus problem (all particles have to agree on some output value) can easily be solved once a leader has been elected. We present a local-control algorithm that elects a leader in $O(n)$ asynchronous rounds with high probability, where n is the number of particles. Our algorithm relies only on local information (e.g., particles do not have unique identifiers, they do not know n, and they do not have a global coordinate system) and requires only a constant-size memory at each particle.

As we mentioned above, the individual chapters of the second part of this thesis are not independent but rather form a single coherent unit. Accordingly, some important aspects are covered by only one of the two chapters. Specifically, Chapter 5 contains the definition of the amoebot model and gives an overview of the related work, whereas Chapter 6 provides an outlook of potential directions for future research on the subject of programmable matter.

Chapter 5 is based on the following publications.

> 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: *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming (DNA)*, see [Der+15b].

> J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. “Improved Leader Election for Self-Organizing Programmable Matter”. In: *Proceedings of the 13th International Symposium on Algorithms and Experiments for Wireless Networks (ALGOSENSORS).* To appear, see [Day+17].

Chapter 6: Shape Formation with Programmable Matter The final chapter of this thesis considers the problem of *shape formation* with programmable matter. On the basis of the amoebot model, we present an algorithm that allows the particles to construct a large class of shapes composed of a constant number of unit-size equilateral triangles that are arranged on a grid. We assume that the particles are well-initialized in that they initially form a triangle and the memory of each particle holds a representation of the desired shape. Under these assumptions, the algorithm constructs a scaled version of the given shape that includes all particles. The construction of the shape takes $O(\sqrt{n})$ rounds, which is optimal in the sense that for any shape deviating from the initial triangle, any shape formation algorithm requires $\Omega(\sqrt{n})$ rounds in the worst case. As in Chapter 5, the algorithm relies exclusively on local information and requires only a constant-size memory per particle. The chapter is based on the following publication.

> 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 Parallelism in Algorithms and Architectures (SPAA)*, see [Der+16b].

Part I

Overlay Networks

Chapter 2

Churn- and DoS-Resistant Overlay Networks

Large-scale and highly decentralized overlay networks have become increasingly popular since the rise of peer-to-peer systems and social networks. Issues such as *churn* (the membership rapidly changes over time) and *adversarial attacks* pose significant challenges for these networks, so a considerable amount of work has been invested in recent years to find effective ways of protecting overlay networks against these influences. A standard approach is to continuously refresh the topology of an overlay network so that defects in the network caused by churn or attacks are repaired. Early attempts in practice go back to solutions such as JXTA (see, e.g., [Gon01; Tra+03]), where at the core the peers continuously exchange random neighbors with their neighbors in an attempt to keep the network well-connected. In theory, it is already known that there are simple rules for randomly switching edges in a local fashion so that eventually a random graph emerges (e.g., [MS06]). However, the running time bounds shown so far for these rules are fairly large, and a rigorous analysis showing that such a rule would be sufficient to get to a random graph in polylogarithmically many parallel rounds seems to be out of reach at this time, even if there is no churn. An alternative approach is to keep the nodes in some hypercubic topology and to perform load balancing under churn in order to make sure that all places of the hypercubic topology are kept well-occupied by the nodes (e.g., [KSW05]). However, here the best result known so far is a solution that can just tolerate a logarithmic churn per round.

Currently, the most promising approach to rigorously handle high churn is to use *random walks* to continuously reorganize the network (see, e.g., [Aug+15]), which is also the approach we use in this chapter. However, instead of just using random walks in a standard fashion, which would take $\Omega(\log n / \log\log n)$ communication rounds in graphs of polylogarithmic degree in order to sample nodes uniformly at random, we combine random walks with *pointer jumping*, which *exponentially* improves the running time needed for random walks to sample nodes uniformly at random. Pointer jumping (i.e., a node introduces its neighbors to its neighbors) is a well-known technique in the area of parallel computing [JáJ92], but to our great surprise, it seems that it has never been combined with random walks so far. We refer to the technique of combining random walks with pointer jumping to sample nodes from a network as *rapid node sampling*. We present two rapid node sampling algorithms in this chapter, one for hypercubes and one for regular expander graphs. Both algorithms allow each node in a network to sample a logarithmic number of nodes uniformly at random in $O(\log \log n)$ communication rounds.

On the basis of rapid node sampling, we develop algorithms that maintain the connectivity of a network under heavy churn and *denial-of-service attacks* (or *DoS-attacks*). Our first algorithm organizes the nodes of a network into an expander and maintains connectivity under adversarial churn by an omniscient adversary with constant churn rate. An important assumption underlying this result is that a node that is prescribed to leave the network by the adversary does not have to leave immediately, but can remain in the network for another $O(\log \log n)$ rounds. Our second algorithm uses a network that is based on the hypercube and maintains connectivity under adversarial DoS-attacks. A node under a DoS-attack is blocked from communicating with any other node. The adversary is allowed to block $(1 - \varepsilon) \cdot n$ nodes for any constant $0 < \varepsilon \leq 1$, and it can change the set of blocked nodes in every round. For this result, we assume the adversary to be t-late in that it only has access to topological information that is at least $t = \Theta(\log \log n)$ rounds old. Both of our algorithms are based upon the idea of switching the topology of an overlay network to a new topology that is independent of the old topology at regular intervals. The algorithms use networks of polylogarithmic degree and require polylogarithmic communication work at each node in every round.

Underlying Publication This chapter is based on the following publication.

> M. Drees, R. Gmyr, and C. Scheideler. “Churn- and DoS-resistant Overlay Networks Based on Network Reconfiguration”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, see [DGS16].

Outline We begin with an overview of the related literature in Section 2.1. In Section 2.2 we define the network model and the exact nature of the attacks we consider. We then introduce some fundamental concepts such as the network topologies underlying our algorithms in Section 2.3. The three sections following Section 2.3 constitute the main part of this chapter. In Section 2.4 we present the rapid node sampling algorithms for overlay networks that form the foundation of our technical contribution, in Section 2.5 we present an algorithm that maintains the connectivity of a network under adversarial churn, and in Section 2.6 we present an algorithm that maintains the connectivity of a network under adversarial DoS-attacks. We conclude this chapter in Section 2.7 by discussing possible directions for future research.

2.1 Related Work

One of the central ideas in this chapter is to switch the topology of an overlay network to a new topology that is independent of the old topology at regular intervals. Various ways of achieving this have already been studied in the literature. One way is to use random local edge-switching rules in order to eventually obtain a random graph (see, e.g., [Fed+06; MS06; CDH09]). However, as we stated in the introduction, the running time bounds shown so far for these rules are fairly large, and a rigorous analysis showing that such a rule would be sufficient to get to a random graph in polylogarithmically many parallel rounds seems to be out of reach at this time, even without churn.

Another approach is to use routing or sorting with the goal of randomly reordering the nodes in an overlay network. To illustrate this, consider the skip graph, which is known to be an expander with high probability [AW09]. Suppose that the skip graph is formed by nodes with labels that are chosen uniformly at random from the interval $[0, 1]$. Then the formation of a new, independent skip graph can essentially be reduced to a routing problem in the old skip graph where each node v sends a message to the node w closest to a randomly chosen $x \in [0, 1]$, which is supposed to be v's new label. Once the routing is complete, it is not hard to use the old skip graph to establish a skip graph on the new labels in altogether just $O(\log n)$ communication rounds. However, it is not clear how to reduce this running time below $O(\log n)$ for overlay networks of polylogarithmic degree and while just allowing a polylogarithmic communication work at each node in every round, which prohibits the application of this approach in this work.

There are various further approaches for maintaining connectivity under churn. A standard approach in practice is to use a multi-tier architecture where the older, more stable peers form the actual overlay network while the young peers just connect to one or more of the stable peers (see, e.g., [Sut+13] and the references therein). Another way is to keep the nodes in some hypercubic topology and to perform local load-balancing in order to make sure that all places of the hypercubic topology are kept well-occupied by nodes under churn (e.g., [Abr+03; KSW05]). However, just eventual recovery from adversarial churn, or adversarial churn allowing only a logarithmic number of arrivals and departures per round has been considered here. Various other solutions have been proposed in theory that can handle either stochastic churn or adversarial churn that does not cause disconnectivity in a round, either by using a high enough degree or by constraining the adversary (e.g., [AS07b; NW07; JP13]).

Also, self-healing networks (e.g., [ST08; HST12; PRT16]) have been proposed, but in these networks each insertion or deletion of a constant number of nodes is followed by some process of repairing the network. Finally, self-stabilizing overlay networks can be used to handle churn (see, e.g., [Jac+09; BGP13; For+14]), but no concrete bounds on the churn rate are known for these.

Currently, the most promising approach to handle high levels of churn is to use random walks to continuously reorganize a network, which is the approach taken in this chapter. Interestingly, this approach was also used by Augustine et al. [Aug+15] in a publication that appeared only few months before the publication underlying this chapter [DGS16]. In this impressive work, Augustine et al. present a randomized distributed protocol that maintains a constant degree expander under levels of adversarial churn that are on par with those considered here. While the goals and the abstract approach of Augustine et al. are similar to ours, there are distinct differences in the underlying assumptions. For example, Augustine et al. consider an adversary that is oblivious to the random choices made by the algorithm while we assume an omniscient adversary for churn but allow the nodes to remain in the network for an additional $O(\log \log n)$ rounds, which is not a standard assumption. Furthermore, Augustine et al. allow high levels of churn but assume that the overall number of nodes remains static. In contrast, we allow the number of nodes in the network to change within certain limits, see Section 2.5. Maybe most strikingly, the approach of Augustine et al. requires a *two-step protocol* that allows pairs of nodes to communicate without interference by the adversary (i.e, the adversary cannot churn out any nodes during two consecutive rounds). They formally prove that this protocol is necessary to achieve such high levels of churn in their model. In our model, no such protocol is required. Besides these differences in the setting, there are also significant differences in the conceptual ideas behind the respective algorithms: On an abstract level, our approach could be called *discrete* in that the topology of the network is kept mostly static over certain intervals of time and is then replaced completely by a new, independent topology while the protocol of Augustine et al. is more *continuous* in that the network changes in every round. A more detailed comparison of these works and a full analysis of the implications of the different underlying assumptions is a non-trivial task that we leave for future research. We think that further investigation might reveal that certain aspects of both approaches could be combined to reap new advantages.

Some of the solutions above not only handle churn but also limited DoS-attacks (e.g., [KSW05; AS07b; Sut+13]). Within our model, DoS-attacks are more severe than churn in a sense that if an adversary decides to block a node, it is instantly blocked without warning. If the adversary is not aware of the network topology, a standard approach to prevent disconnectivity is to randomly spread nodes in an overlay network and use redundant connections.

However, once the adversary knows the topology, nothing can be done to prevent disconnectivity unless the degree (defined as the maximum number of edges originating at or leading to a node) is higher than the maximum number of nodes that the adversary can block at the same time: If an adversary wants to isolate a node u, it simply blocks all nodes v that can either send a message to u or receive a message from u. In this chapter, we use the concept of a t-late adversary that has access to topological information that is at least t rounds old. Thereby, we consider an adversary that lies between the two extremes of having complete, up-to-date knowledge and having no knowledge at all about the topology. A similar adversary has been used in [Ahm+15], for example, albeit in a different context. Next to the approaches mentioned above, there have also been several works on protecting distributed hash tables (DHTs) against DoS-attacks from outsiders [KMR02], past insiders [AS07a], or even insiders [ES15], but for past insiders and insiders the nodes are assumed to form a clique to avoid disconnectivity problems. Finally, apart from blocking nodes also other attacks on the connectivity of an overlay network have been studied, such as Sybil-attacks [Dou02] and Eclipse-attacks [Sin+06].

Our algorithms are based on *rapid node sampling*, which combines random walks with *pointer jumping*. Pointer jumping (which is also known as *pointer doubling* or simply the *doubling technique*) is a well-known technique in the area of parallel computing [JáJ92]. It has originally been applied to rapidly contract trees by letting each node continuously introduce its current parent as the new parent to its children. In this way, a tree of depth D can be contracted in $O(\log D)$ parallel rounds to a tree of depth 1. Another application of pointer jumping is to rapidly form a clique: If in a graph of diameter D every node continuously introduces its neighbors to each other, then it just takes $O(\log D)$ communication rounds until a clique is formed. However, the communication work per round when using message passing is huge towards the end for both cases, which is one of the main technical difficulties we have to overcome to apply pointer jumping in this work.

The rapid node sampling algorithms for hybercubes and regular expander graphs presented in Section 2.4 allow each node in the network to sample a logarithmic number of nodes uniformly at random using only $O(\log \log n)$ communication rounds. These algorithms are *specific to overlay networks*, i.e., they exploit the fact that a node can send a message to any other node whose identifier it knows. For *static networks* Das Sarma et al. [SNP09; Sar+13] provide distributed algorithms for different variations of the problem of creating random walks. When considering the problem of letting each node sample a logarithmic number of nodes via random walks of length $O(\log n)$, our rapid node sampling algorithms represent an exponential improvement in running time over to the algorithm of Das Sarma et al. This suggests a fundamental difference in the difficulty of this problem in the two models.

2.2 Model and Problem Statement

We consider a node set V that can potentially change over time. The nodes are organized into an *overlay network*: Every node u has a unique *identifier* $\mathrm{id}(u)$, which is a bit string of size $O(\log n)$ where $n = |V|$. We simply write u instead of $\mathrm{id}(u)$ when it is clear from the context that we refer to the identifier of u. A node u can send a message to a node v if u stores $\mathrm{id}(v)$ in its local memory. We define the edge set of the overlay network as

$$E = \{\, (u, v) \mid u \in V \text{ stores } \mathrm{id}(v) \text{ in its local memory} \,\}.$$

We use the *synchronous message passing* model, which means that all nodes operate in synchronous *rounds*. Each round consists of three steps. In the first step, a node receives all messages sent to it in the previous round. In the second step, a node can perform any kind of local computation. In the third step, a node u can send a distinct message to each node v such that $(u, v) \in E$. We define the *communication work* of a node in a specific round as the total number of bits that it receives and sends in that round.

We investigate the problem of *maintaining connectivity* in overlay networks under *large-scale adversarial churn* and *adversarial DoS-attacks* using only polylogarithmic communication work for each node in each round. Connectivity can be interpreted as a minimum requirement for a network to be resistant against attacks. The algorithms presented in this chapter actually provide stronger guarantees, which we will discuss in the corresponding sections. In general, the algorithms aim at maintaining a certain topology for the overlay network. We assume that the edges forming this topology are *marked* to be distinguishable from other overlay edges. Let E' be the set of marked edges. The subgraph induced by E' will always be a *bidirected* graph, i.e., if $(u, v) \in E'$ then also $(v, u) \in E'$. Therefore, we can consider the graph induced by E' to be undirected for simplicity.

In the following, we define the exact nature of the attacks we consider. For *adversarial churn* we assume that an *omniscient* adversary prescribes a node set W_i for each round i. The adversary has a *churn rate* of r if

$$\frac{|W_i|}{r} \leq |W_{i+1}| \leq r \cdot |W_i|$$

for all i. For each node $u \in W_i \setminus W_{i-1}$ (i.e., a new node *joining* the network in round i) we require that it is introduced to exactly one node in $W_i \cap W_{i-1}$ (i.e., a node *staying* in the network in round $i - 1$). Futhermore, we require that altogether at most $\lceil r \rceil$ new nodes are introduced to any node in a round. Formally, if a node u is introduced to a node v then v learns $\mathrm{id}(u)$. A node u is *leaving* in round i if $u \in W_{i-1} \setminus W_i$. For simplicity, we assume that every node is prescribed to join and leave the system only once and, therefore, every identifier is used at most once. The decisions of the adversary can be based on any information about the past and current state of the network.

The network has some flexibility in adapting to the W_i's. Specifically, a node that is prescribed to leave the network is allowed to remain in the network for an additional T rounds where T should be as small as possible. So formally, if V_i is the set of nodes in the network in round i, we require

$$W_i \subseteq V_i \subseteq \bigcup_{j=i-T}^{i} W_j.$$

For *adversarial DoS-attacks* we allow an *r-bounded* adversary to block $r \cdot n$ nodes in every round. A blocked node cannot send or receive messages. Therefore, a message sent by a node u to a node v in round i is successfully delivered if and only if u is not blocked in round i and v is not blocked in round $i+1$. Messages that do not satisfy these conditions are simply dropped from the network. Note that for a protocol to make progress, it is crucial that nodes that are not blocked in round i can send messages to nodes that are not blocked in round $i+1$ for any i.

In the context of DoS-attacks, we restrict the knowledge of the adversary to topological information regarding the overlay network, i.e., the adversary can inspect the node set and the edge set. The adversary cannot inspect the internal state of nodes, the contents of messages, or even the number of messages sent along an edge. We say an adversary is *t-late* if it only has access to information that is at least t rounds old. We say that an overlay network is connected under a DoS-attack if the network restricted to its non-blocked nodes is weakly connected. Recall that in an overlay network in which the sum of the in- and out-degree of each node is bounded by d, we cannot maintain connectivity under DoS-attacks by an r-bounded 0-late adversary if $r \geq d/n$ since such an adversary can isolate individual nodes. Therefore, we need $t > 0$ to achieve non-trivial values for r in a network of small degree.

2.3 Preliminaries

Before we get to the main part of this chapter, we introduce some basic tools, network topologies, and node sampling techniques.

Chernoff Bounds We extensively use the following bounds, which are known as *Chernoff bounds*.

Lemma 2.1. *Suppose that $X_1, X_2, \ldots, X_n$ are independent binary random variables. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then it holds for all $\delta > 0$ that*

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\min\{\delta^2,\delta\}\cdot\mu/3}.$$

Furthermore, it holds for all $0 < \delta < 1$ that

$$\Pr[X \leq (1-\delta)\mu] \leq e^{-\delta^2\mu/2}.$$

Network Topologies We use two network topologies throughout this chapter, namely the well-known *hypercube* and a graph class we refer to as $\mathbb{H}$-*graphs*, following the notation used in [LS03]. A d-dimensional hypercube is an undirected, simple graph $G = (V, E)$ where $V = \{0, 1\}^d$ is the set of all d-tuples with elements from $\{0, 1\}$ and E is such that two vertices are connected if and only if they differ in exactly one coordinate.

An $\mathbb{H}$-graphs is an undirected multigraph $G = (V, E)$ with $E = \bigcup_{i=1}^{d/2} C_i$ where $d \geq 8$ is an even number and each C_i is a set of edges that form a Hamilton cycle over the nodes in V. Note that the union in the definition of the edge set E is a multiset union. Even though an $\mathbb{H}$-graph is an undirected graph, we assume that each C_i has an orientation, i.e., a node u stores the identifier of its predecessor and its successor in C_i. By definition, an $\mathbb{H}$-graph is a connected d-regular multigraph that can have parallel edges but no loops.

Node Sampling Our algorithms heavily rely on the ability of the nodes to efficiently sample nodes (almost) uniformly at random from a network. In the d-dimensional hypercube we can achieve uniform node sampling using the following well-known random walk technique: A node u in the hypercube creates a token containing $\mathrm{id}(u)$ that traverses the graph for d rounds. Let $v = (b_1, b_2, \ldots, b_d)$ be the node that holds the token at the beginning of round i. The node v flips a fair coin. If the coin comes up tails, v keeps the token. Otherwise, v forwards the token to the neighboring node

$$n_i(v) = (b_1, \ldots, b_{i-1}, 1 - b_i, b_{i+1}, \ldots, b_d).$$

The node w that holds the token at the end of the random walk sends $\mathrm{id}(w)$ to u. It is not hard to see that w is chosen uniformly at random from V. Therefore, a node in a d-dimensional hypercube can perform uniform node sampling in $d = O(\log n)$ rounds.

We can also use random walks to perform node sampling on randomly generated $\mathbb{H}$-graphs. To generate an $\mathbb{H}$-graph G, we choose $d/2$ *directed* Hamiltonian cycles independently and uniformly at random from the set of all directed Hamiltonian cycles over n nodes. The graph G is formed by interpreting each of these cycles as being *undirected* and taking the union of the cycles. As we discussed above, we assume that the nodes know the direction of the cycles for technical reasons. Consider the simple random walk over G, i.e., at a node u the random walk chooses an edge incident to u uniformly at random and then moves along that edge to another node. Intuitively, the graph G is an *expander graph.* Therefore, the simple random walk on G quickly converges towards the stationary distribution of the corresponding Markov chain. Since G is a regular graph, the stationary distribution is the uniform distribution over V. As a consequence, *short* random walks are sufficient to sample nodes from V according to a distribution that is close to uniform.

Formally, we have the following arguments, which resemble the arguments given in [LS03]. The randomly chosen graph G satisfies the following theorem, which was shown by Friedman [Fri08].

Theorem 2.2 (Friedman [Fri08])**.** *For any fixed real $\epsilon > 0$ there is a constant c such that*

$$\Pr\left[\, \forall i > 1 : \; |\lambda_i| \le 2\sqrt{d-1} + \epsilon \,\right] \ge 1 - \frac{c}{n^{\tau}},$$

where λ_i are the eigenvalues of the adjacency matrix of G in descending order and $\tau = \lceil \sqrt{d-1} \rceil - 1$.

We say an event occurs *with high probability* (abbreviated as *w.h.p.*) if it occurs with probability at least $1 - n^{-c}$ for a given constant $c \ge 1$. Theorem 2.2 implies the following corollary.

Corollary 2.3. *For d and n sufficiently large we have $|\lambda_i| \le 2\sqrt{d}$ for all $i > 1$, w.h.p.*

Furthermore, we have the following lemma, which is a consequence of Theorem 5.1 in [Lov93].

Lemma 2.4. *Let G be a d-regular multigraph with $|\lambda_i| \le 2\sqrt{d}$ for all $i > 1$ and let $t \in \mathbb{N}$ such that $t \ge 2\alpha \log_{d/4} n$ for some constant $\alpha \ge 1$. Consider the simple random walk of length t starting at a node u. Then for any node $v \in V$, we have*

$$\left| \Pr\left[\textit{random walk ends at } v\right] - \frac{1}{n} \right| \le n^{-\alpha}.$$

Proof. According to Theorem 5.1 in [Lov93], we have

$$\begin{aligned} \left| \Pr[\text{random walk ends at } v] - \frac{1}{n} \right| &\le \left(\frac{2\sqrt{d}}{d} \right)^{2\alpha \log_{d/4} n} \\ &= n^{2\alpha \log_{d/4}(2/\sqrt{d})} \\ &= n^{\alpha \log_{d/4}(4/d)} \\ &= n^{-\alpha}. \end{aligned}$$

□

We refer to the probability distribution given in Lemma 2.4 as *almost uniform.* Corollary 2.3 and Lemma 2.4 imply that for n and d sufficiently large, a node in a random $\mathbb{H}$-graph can perform almost uniform node sampling in $O(\log n)$ communication rounds, w.h.p.

2.4 Rapid Node Sampling

In this section, we show how pointer jumping can be used to achieve an exponential speed-up over the node sampling techniques presented in the previous section. Our goal is to let each node in a network sample $\log n$ nodes independently and (almost) uniformly at random from the set of all nodes in the network. The *rapid node sampling* algorithms presented in this section achieve this in $O(\log\log n)$ rounds for both $\mathbb{H}$-graphs and hypercubes. The core idea underlying these algorithms is to iteratively combine shorter random walks to create longer random walks. This strategy intuitively corresponds to performing pointer jumping along a random walk that was generated in a distributed fashion. The rapid node sampling algorithms introduced in this section form the foundation for the churn- and DoS-resistant networks presented in the following sections.

Before we present the algorithms, we show that a running time of $O(\log\log n)$ rounds is asymptotically optimal.

Lemma 2.5. *Consider an overlay network that is structured according to an undirected graph $G = (V, E)$ of diameter D. An algorithm that allows every node u to sample a node from V (almost) uniformly at random requires $\Omega(\log D)$ rounds.*

Proof. Let $u, v \in V$ be such that the shortest path between the nodes has length D. Consider the following algorithm, which can be interpreted as an extreme application of pointer jumping: In each round every node mutually introduces all its neighbors by sending all identifiers it stores to all nodes it knows. This algorithm requires $\Omega(\log D)$ rounds to introduce v to u. Clearly, no algorithm can introduce v to u faster than this algorithm. Hence, for any algorithm that samples a node from V and stores it at u in $o(\log D)$ rounds, we must have $\Pr[u \text{ chooses } v] = 0$. Therefore, no algorithm that uses $o(\log D)$ rounds can perform (almost) uniform node sampling over V. □

Both $\mathbb{H}$-graphs and hypercubes are constant-degree graphs, which implies that their diameter is $\Omega(\log n)$. Hence, Lemma 2.5 implies that a running time of $O(\log\log n)$ rounds is indeed asymptotically optimal.

Throughout this section we assume that all nodes know n.

2.4.1 H-Graphs

In the rapid node sampling algorithms, the nodes collaborate to construct random walks of length $\Theta(\log n)$ in $O(\log\log n)$ rounds. For $\mathbb{H}$-graphs, each node locally executes Algorithm 2.1. The algorithm is divided into four phases. Phase 1 is executed only once. The remaining phases are executed in a loop. The execution of a phase in this algorithm corresponds to one round. The variable T in Line 5 determines the length of the generated random walks. To generate random walks of length at least $2\alpha \log_{d/4} n$, we choose

$$T = \lceil \log(2\alpha \log_{d/4} n) \rceil = \log \log n + O(1).$$

Each node stores a multiset M of identifiers. During the execution of the algorithm, M is repeatedly emptied and then filled to a certain size. Specifically, the size of M before the first iteration of the loop in Line 5 is defined as m_0 and the size of M after the i-th iteration of the loop is defined as m_i. After the execution of the algorithm, M contains identifiers of nodes that were chosen almost uniformly at random from the set of nodes in the network.

Algorithm 2.1 Rapid node sampling in $\mathbb{H}$-graphs

```
    Phase 1:
 1: $M \leftarrow \emptyset$
 2: for $j \leftarrow 1$ to $m_0$ do
 3:     choose neighbor $v$ in $\mathbb{H}$-graph uniformly at random
 4:     $M \leftarrow M \cup \{v\}$
 5: for $i \leftarrow 1$ to $T$ do
        Phase 2:
 6:     for $j \leftarrow 1$ to $m_i$ do
 7:         choose and remove $v \in M$ uniformly at random
 8:         send request to $v$
        Phase 3:
 9:     for each request received from a node $v$ do
10:         choose and remove $w \in M$ uniformly at random
11:         send response $w$ to $v$
        Phase 4:
12:     $M \leftarrow$ set of received responses
```

The correctness of the algorithm depends on the choice of the values m_i: An inappropriate choice might lead to M being empty when an element should be extracted in Line 7 or 10. We say the algorithm *succeeds for node u in iteration i* if during the i-th iteration of the loop in Line 5 in the execution of the algorithm by u, the set M is never empty when an element should be extracted. We say the algorithm *succeeds* if it succeeds for all nodes and for all iterations. Before we turn to the choice of the m_i, we show that the algorithm is correct under the assumption that it succeeds. Specifically, we show that after the execution of the algorithm, M contains identifiers of nodes that were chosen by following independent random walks of length at least $2\alpha \log_{d/4} n$.

Lemma 2.6. *Consider a node u. After the i-th iteration of the loop in Line 5, M only contains identifiers of nodes that were chosen by following independent simple random walks of length 2^i that started at u.*

Proof. We show the lemma by induction on i. Consider a node u. For $i = 0$ (i.e., before the first iteration) M contains nodes that were chosen independently and uniformly at random from the neighbors of u in the $\mathbb{H}$-graph. Therefore, the statement of the lemma holds in this case. Now suppose that the statement holds after iteration i. We consider iteration $i + 1$ and show that the statement also holds after this iteration. In Phase 2, u sends requests to nodes that were chosen by following simple random walks of length 2^i starting at u according to the induction hypothesis. A node v that receives such a request replies in Phase 3 with an identifier of a node that was chosen by following a simple random walk of length 2^i starting at v. In Phase 4, u sets M to the set of received responses. Each identifier received as a response in Phase 4 belongs to a node that was chosen by following the concatenation of two simple random walks of length 2^i. Such a concatenation corresponds to a simple random walk of length 2^{i+1}. Since the algorithm never reuses an element of M, the constructed random walks are independent. Therefore, the statement holds after iteration $i + 1$. □

In Lemma 2.8 we establish a choice for the m_i such that the algorithm succeeds, w.h.p., and after the algorithm terminates, M contains at least $\log n$ identifiers. The proof of Lemma 2.8 requires the following auxiliary lemma.

Lemma 2.7. *Consider a d-regular undirected multigraph $G = (V, E)$. Suppose that each node in V creates $k \in \mathbb{N}$ tokens. The tokens traverse G in synchronous rounds according to independent simple random walks. For $v \in V$ and $t \in \mathbb{N}$ let X be the number of tokens at node v after t rounds. Then X is a sum of independent binary random variables and $E[X] = k$.*

Proof. Let $v \in V$ and $t, k \in \mathbb{N}$. For a node u let the binary variable $X_{u,i}$ be such that $X_{u,i} = 1$ if and only if the i-th token created by u is at v after t steps. We have $\mathrm{E}[X_{u,i}] = p_u d^{-t}$ where p_u is the number of distinct (not necessarily simple) paths of length exactly t from u to v. Let $X_u = \sum_{i=1}^{k} X_{u,i}$. Then

$$\mathrm{E}[X_u] = \sum_{i=1}^{k} \mathrm{E}[X_{u,i}] = k p_u d^{-t}.$$

Finally, let $X = \sum_u X_u$. By definition, X is a sum of independent binary random variables. Furthermore, we have

$$\mathrm{E}[X] = \sum_u \mathrm{E}[X_u] = k d^{-t} \cdot \sum_u p_u = k.$$

□

Lemma 2.8. *For any* $0 < \varepsilon \leq 1$ *there is a constant* $c \geq 1$ *such that with*

$$m_i = (2+\varepsilon)^{T-i} c \log n$$

the algorithm succeeds, w.h.p.

Proof. Consider the i-th iteration of the loop in Line 5 for some $i \geq 1$. Let X denote the number of requests received by a node u in Phase 3. It is not hard to see that the algorithm succeeds for u in iteration i if $m_i + X \leq m_{i-1}$. By the definition of m_i we have

$$\Pr[m_i + X > m_{i-1}] \leq \Pr[m_i + X \geq (2+\varepsilon)m_i] = \Pr[X \geq (1+\varepsilon)m_i].$$

According to Lemma 2.6 we can apply Lemma 2.7 to deduce that X is a sum of independent binary random variables and $\mathrm{E}[X] = m_i$. Therefore, we can apply Chernoff bounds to get

$$\Pr[X \geq (1+\varepsilon)m_i] \leq e^{-\varepsilon^2 m_i/3} \leq e^{-\varepsilon^2 c \log n/3},$$

where the second inequality holds because $m_i \geq c \log n$ by definition. The lemma follows by applying the union bound over all nodes and all iterations. □

The following theorem concludes our analysis.

Theorem 2.9. *For any* $0 < \varepsilon \leq 1$ *the algorithm lets each node sample at least* $\log n$ *nodes almost uniformly at random from the set of all nodes in the network in* $O(\log\log n)$ *rounds, w.h.p. The communication work for every node in every round is* $O(\log^{2+\log(2+\varepsilon)} n)$.

Proof. According to Lemma 2.8, the algorithm succeeds, w.h.p., and the number of sampled nodes at each node after termination is $m_T = c \log n$ for some $c \geq 1$. By Lemma 2.6 and the arguments given in Section 2.3, the sampled nodes are chosen almost uniformly at random from the set of all nodes in the network. The upper bound on the running time follows from our choice of T. Finally, since $m_i \leq m_0$ for all i, the number of identifiers sent and received by a node in every round is at most

$$O(m_0) = O\left((2+\varepsilon)^T \log n\right) = O\left(\log^{1+\log(2+\varepsilon)} n\right).$$

According to our assumptions, each identifier is a bit string of length $O(\log n)$. Therefore, the given bound on the communication work holds. □

Note that the above analysis does not use any properties of $\mathbb{H}$-graphs aside from their regularity and their expansion. Therefore, the proposed algorithm works for arbitrary regular graphs with appropriate expansion properties.

2.4.2 Hypercubes

Conceptually, rapid node sampling on hypercubes works analogously to rapid node sampling on $\mathbb{H}$-graphs in that the algorithm combines short random walks to form longer random walks. However, the algorithm for hypercubes and its analysis are technically more involved. Consider a d-dimensional hypercube. Note that by definition $d = \log n$. Each node of the hypercube executes Algorithm 2.2. We refer to the node executing the algorithm as u in the pseudocode.

Algorithm 2.2 Rapid node sampling in hypercubes

```
Phase 1:
for $j \leftarrow 1$ to $\log n$ do
    $M_j \leftarrow \emptyset$
    for $k \leftarrow 1$ to $m_0$ do
        flip a fair coin
        if coin flip comes up heads then
            $M_j \leftarrow M_j \cup \{n_j(u)\}$
        else
            $M_j \leftarrow M_j \cup \{u\}$
for $i \leftarrow 1$ to $\lceil \log \log n \rceil$ do
    Phase 2:
    for $j \leftarrow 1$ to $\log n$ with step-size $2^i$ do
        if $j + 2^{i-1} - 1 \geq \log n$ then
            break
        for $k \leftarrow 1$ to $m_i$ do
            choose and remove $v \in M_j$ uniformly at random
            send request $(u, j)$ to $v$
    Phase 3:
    for each received request $(v, j)$ do
        $j' \leftarrow j + 2^{i-1}$
        choose and remove $w \in M_{j'}$ uniformly at random
        send response $(w, j)$ to $v$
    Phase 4:
    for $j \leftarrow 1$ to $\log n$ with step-size $2^i$ do
        if $j + 2^{i-1} - 1 \geq \log n$ then
            break
        $M_j \leftarrow \emptyset$
    for each received response $(v, j)$ do
        $M_j \leftarrow M_j \cup \{v\}$
```

As in the previous section, the correctness of the algorithm depends on the choice of the values m_i. We assume for now that the m_i are large enough so that whenever the algorithm attempts to take an element from a set M_j, it succeeds. The following lemma characterizes the core idea behind the algorithm. Under the given assumption, the lemma implies that once the algorithm terminates, the set M_1 at any node contains only identifiers of nodes that were chosen uniformly at random from the hypercube.

Lemma 2.10. *Consider a node u. After the i-th iteration of the loop in Line 9 of the algorithm the following statement holds. For any j such that $1 \leq j \leq \log n$ and $j \equiv 1 \pmod{2^i}$, and any node $v \in M_j$, the coordinates*

$$j, \ldots, \min\{j + 2^i - 1, \log n\}$$

of v were chosen independently and uniformly at random while the remaining coordinates of v are identical to the corresponding coordinates of u.

Proof. We show the lemma by induction on i. For $i = 0$ (i.e., before the first iteration) it is not hard to check that Phase 1 of the algorithm initializes the sets M_j of the nodes in such a way that the statement holds for all nodes. So suppose that the statement holds for all nodes after iteration $i - 1$. We consider a node u and an iteration i and show that the statement holds for u after iteration i.

Let j be such that $1 \leq j \leq \log n$ and $j \equiv 1 \pmod{2^i}$. Consider a set M_j. By the induction hypothesis, at the beginning of the iteration each node $v \in M_j$ is such that the coordinates

$$j, \ldots, \min\{j + 2^{i-1} - 1, \log n\}$$

were chosen independently and uniformly at random while the remaining coordinates of v coincide with the corresponding coordinates of u. We distinguish two cases.

If $j + 2^{i-1} - 1 \geq \log n$ then the algorithm does not send out requests for the nodes in M_j in Phase 2 and it does not replace the nodes in M_j in Phase 4. Since $j + 2^{i-1} - 1 \geq \log n$ implies $j + 2^i - 1 \geq \log n$, the statement of the lemma is satisfied after iteration i in this case.

If $j + 2^{i-1} - 1 < \log n$ then the algorithm sends requests to m_i nodes taken from M_j during Phase 2. Let $v \in M_j$ be a node to which u sends a request. In Phase 3, v replies to this request with a node w chosen uniformly at random from the set $M_{j'}$ in the memory of v where $j' = j + 2^{i-1}$. Note that $j' \leq \log n$ and, therefore, the set $M_{j'}$ exists. By the induction hypothesis, the coordinates

$$j', \ldots, \min\{j' + 2^{i-1} - 1, \log n\}$$

of w were chosen independently and uniformly at random while the remaining coordinates correspond to v. The interval of randomly chosen coordinates of w equals the interval

$$j + 2^{i-1}, \ldots, \min\{j + 2^i - 1, \log n\}.$$

When combining this argument with the argument concerning the coordinates of v given above, we get that in relation to u the coordinates

$$j, \ldots, \min\{j + 2^i - 1, \log n\}$$

of w were chosen independently and uniformly at random while the remaining coordinates of w coincide with the corresponding coordinates of u. In Phase 4, u removes all elements from M_j and then fills M_j with nodes that satisfy the property given for the node w above. Thereby, also in this case the statement of the lemma holds, which completes the induction. □

We now turn to the choice of the values m_i. Similarly to the previous section, we say the algorithm *succeeds for node u and index j in iteration i* if during the i-th iteration of the loop in Line 9 in the execution of the algorithm by u, the set M_j is never empty when an element should be extracted. We say the algorithm *succeeds* if it succeeds for all nodes, all j, and all iterations. We have the following lemma.

Lemma 2.11. *For $0 < \varepsilon \leq 1$ there is a constant $c \geq 1$ such that with*

$$m_i = (2 + \varepsilon)^{\lceil \log \log n \rceil - i} c \log n$$

the algorithm succeeds, w.h.p.

Proof. We first establish a simple observation. Consider iteration i of the loop in Line 9 during the execution by a node u, and let j be such that $1 \leq j \leq \log n$. If the algorithm removes a node from M_j in Phase 2 then we must have $j \equiv 1 \pmod{2^i}$. If the algorithm removes a node from M_j in Phase 3 then we must have $j \equiv 2^{i-1} + 1 \pmod{2^i}$. Since $1 \not\equiv 2^{i-1} + 1 \pmod{2^i}$, the algorithm cannot remove nodes from M_j in both Phase 2 and Phase 3 during the same iteration.

To prove the lemma, we show the following statement by induction on i: In iteration i we have that for every node u and every index j the algorithm succeeds, and at the end of the iteration it holds $|M_j| \geq m_i$ for all j. For $i = 0$ (i.e., before the first iteration) the statement holds since no nodes are taken from the sets M_j before the algorithm enters the loop in Line 9, and Phase 1 adds exactly m_0 elements to each M_j. So suppose the statement holds for iteration $i - 1$. We show that, w.h.p., the statement also holds for iteration i.

Consider a node u and an index j. We distinguish three cases. For the first case suppose that u does not remove any nodes from M_j in iteration i. In this case, the algorithm cannot fail, which shows the first part of the statement. Furthermore, the induction hypothesis implies that $|M_j| \geq m_{i-1}$ at the end of iteration $i-1$. Since $m_{i-1} \geq m_i$, also the second part of the statement holds.

For the second case suppose that u takes a node from M_j in Phase 2 of iteration i. According to the observation given at the beginning of the proof, this implies that the algorithm does not take any nodes from M_j in Phase 3 of the current iteration. By the induction hypothesis, we have $|M_j| \geq m_{i-1}$ at the end of iteration $i-1$. During Phase 2 of the algorithm, u removes m_i nodes from M_j and sends a request to each of these nodes. Since $m_{i-1} \geq m_i$, the algorithm succeeds. As we will argue in the third case below, all requests of u will be answered. In Phase 4, u discards the remaining elements of M_j and then fills the set with the m_i nodes received as responses to the previously sent requests. Therefore, the second part of the statement holds.

For the third and final case suppose that u removes a node from M_j in Phase 3 of iteration i. According to the observation given at the beginning of the proof, this implies that the algorithm does not remove any nodes from M_j in Phase 2 of the current iteration. Let X be the number of requests received by u such that the reply to the request should contain a node from M_j. By the induction hypothesis, we have $|M_j| \geq m_{i-1}$ at the end of iteration $i-1$. Therefore, if $X + m_i \leq m_{i-1}$ then the algorithm succeeds and at least m_i nodes remain in M_j. By the definition of m_i we have

$$\Pr[X + m_i > m_{i-1}] \leq \Pr[X + m_i \geq (2+\varepsilon)m_i] = \Pr[X \geq (1+\varepsilon)m_i].$$

As we argued in the previous case, the algorithm succeeds for nodes that take elements from M_j in Phase 2 of iteration i. Together with Lemma 2.10 and arguments analogous to those given in Lemma 2.7, this implies that X is a sum of independent binary random variables such that $\mathrm{E}[X] = m_i$. Therefore, we can apply Chernoff bounds to get

$$\Pr[X \geq (1+\varepsilon)m_i] \leq e^{-\varepsilon^2 m_i/3} \leq e^{-\varepsilon^2 c \log n/3},$$

where the second inequality holds because $m_i \geq c \log n$ by definition.

Overall, the inductive statement holds for u and j, w.h.p. Applying the union bound shows that the statement holds for all nodes and all j, w.h.p., which concludes the induction. Applying the union bound another time over all iterations shows the lemma. □

Theorem 2.12. *For any $0 < \varepsilon \leq 1$ the algorithm lets each node sample at least $\log n$ nodes uniformly at random from the set of all nodes in the network in $O(\log \log n)$ rounds, w.h.p. The communication work for every node in every round is $O(\log^{3+\log(2+\varepsilon)} n)$.*

Proof. According to Lemma 2.11, the algorithm succeeds, w.h.p., and the number of sampled nodes at each node after termination is $m_{\lceil \log\log n \rceil} = c \log n$ for some $c \geq 1$. By Lemma 2.10 and the arguments given in Section 2.3, the sampled nodes are chosen uniformly at random from the set of all nodes in the network. The upper bound on the running time follows from the fact that the algorithm executes $\lceil \log\log n \rceil$ iterations of the loop in Line 9. Finally, since $m_i \leq m_0$ for all i, the number of identifiers sent and received by a node in every round is at most

$$O(m_0 \log n) = O\left((2+\varepsilon)^{\lceil \log\log n \rceil} \log^2 n\right) = O\left(\log^{2+\log(2+\varepsilon)} n\right).$$

According to our assumptions, each identifier is a bit string of length $O(\log n)$. Therefore, the given bound on the communication work holds. □

2.5 Adversarial Churn

The rapid node sampling algorithms introduced in the previous sections will be our primary tools throughout the remainder of this chapter. In this section, we use rapid node sampling to quickly update an $\mathbb{H}$-graph in order to include joining nodes and exclude leaving nodes. Thereby, we get a network that is resistant against adversarial churn.

Intuitively, the main idea behind our approach is to keep the network topology fixed over certain intervals of time and keep track of which nodes want to join and leave during these intervals. At the end of an interval, joining nodes are included into the $\mathbb{H}$-graph and leaving nodes are excluded. We achieve this by executing an instance of Algorithm 2.3 for each of the $d/2$ Hamiltonian cycles in the $\mathbb{H}$-graph. In contrast to Section 2.4.1, in which we implicitly treated the Hamiltonian cycles as being *undirected*, we explicitly interpret the cycles as being *directed* throughout this section. Executed locally by each node, Algorithm 2.3 transforms an existing Hamiltonian cycle into a new Hamiltonian cycle that is chosen close to uniformly at random from the set of all possible directed Hamiltonian cycles over the nodes staying in the network. By iterating the algorithm, we get a network that constantly adapts to the node sets prescribed by the adversary.

As in the previous sections, the algorithm is divided into phases. However, in contrast to the previous sections, a phase of Algorithm 2.3 can take multiple communication rounds. We require the nodes to have some knowledge about the size of the network. Specifically, we need estimates on $\log\log n$ and $\log n$ as both are required for the rapid node sampling algorithm for $\mathbb{H}$-graphs. Since polynomial changes in n only cause additive changes in $\log\log n$ and for all realistic values of n, $\log\log n$ is very small, we assume for simplicity that the nodes know an upper bound k on $\log\log n$ that is precise up to some additive deviation of at most some constant c, i.e., $k - c \leq \log\log n \leq k$. Accordingly, we can use 2^k as an estimate for $\log n$ that is precise up to a constant multiplicative factor. Furthermore, we assume that the number of nodes in the network as prescribed by the adversary never falls below a threshold N. All statements said to hold *with high probability* in this section do so with respect to N.

Algorithm 2.3 Update a single Hamiltonian cycle of an $\mathbb{H}$-graph

```
Phase 1:
$U \leftarrow$ set of nodes for which this node is responsible
for each $u \in U$ do
    choose node $v \in V$ via rapid node sampling in $\mathbb{H}$-graph
    send id($u$) to $v$
Phase 2:
$U \leftarrow$ set of received identifiers
$m \leftarrow |U|$
if $U \neq \emptyset$ then
    $(u_1, u_2, \ldots, u_m) \leftarrow$ permutation of $U$ chosen uniformly at random
Phase 3:
if $U \neq \emptyset$ then
    send $u_1$ to closest active predecessor
    send $u_m$ to closest active successor
    receive $u_0$ from closest active predecessor
    receive $u_{m+1}$ from closest active successor
Phase 4:
for $i = 1$ to $m$ do
    send $(u_{i-1}, u_{i+1})$ to $u_i$
receive $(v, w)$
predecessor $\leftarrow v$
successor $\leftarrow w$
```

We now describe in detail how Algorithm 2.3 is used to update the network. We say an $\mathbb{H}$-graph G is *suitable* if the eigenvalues λ_i of the adjacency matrix of G are such that $|\lambda_i| \leq 2\sqrt{d}$ for all $i > 1$. Otherwise, we say G is *unsuitable*. We assume that initially the nodes in W_1 are organized into a suitable $\mathbb{H}$-graph. We divide time into intervals of $\Theta(\log \log n)$ rounds. Consider three consecutive intervals I_1, I_2, and I_3. During I_1, the nodes keep track of the changes to the node set prescribed by the adversary. During I_2, Algorithm 2.3 is used to construct new Hamiltonian cycles that reflect the changes prescribed in I_1. The construction of these new Hamiltonian cycles is based on the current Hamiltonian cycles in the $\mathbb{H}$-graph. Finally, at the beginning of the first round of I_3, the nodes switch to the new Hamiltonian cycles and, thereby, form a new suitable $\mathbb{H}$-graph. This process is executed iteratively in an interleaved fashion, i.e., while the cycles are restructured in I_2, the nodes keep track of further changes to the network, which are then incorporated during I_3, and so on.

A node u that is prescribed to *join* the network during I_1 is introduced to a node v that is already present in the network. We define a node to be *available* if it is part of the $\mathbb{H}$-graph during I_2. If v is available then it keeps $\mathrm{id}(u)$ in its local memory and sends its own identifier to u. If v is not available then it must have joined during I_1. The node v then forwards $\mathrm{id}(u)$ to an available node w and sends $\mathrm{id}(w)$ to u. It holds inductively that v must know an available node.

We say an available node w is *responsible* for the nodes whose identifier it learns during I_1. Furthermore, w is responsible for itself if it is not prescribed to leave the network during I_1. Since each interval consists of $O(\log\log n)$ rounds and in each round at most $\lceil r \rceil \leq r+1$ nodes can join each node in the network (where r is the churn rate), the number of nodes for which w is responsible is at most

$$(r+2)^{c\log\log n} = \log^{c\log(r+2)} n$$

for some constant c. For future reference, we define $\ell(n) = \log^{c\log(r+2)} n$.

For a node u that is prescribed to *leave* the network during I_1 there are three cases. First, if u is part of the $\mathbb{H}$-graph then it stores the fact that it is supposed to leave the network in its local memory, but it remains in the network for now. It participates in the execution of the algorithm during I_2 to integrate nodes for which it is responsible into the network. Note that since u is not staying in the network, it is not responsible for itself. The node u leaves the network in the first round of I_3. Second, if u is currently not part of the $\mathbb{H}$-graph but is scheduled to become part of the $\mathbb{H}$-graph at the beginning of I_2, then u also remains in the network. It becomes part of the $\mathbb{H}$-graph at the beginning of I_2, participates in the algorithm during I_2 while not being responsible for itself, and leaves the network at the beginning of I_3. Finally, if u is not part of the $\mathbb{H}$-graph and it is not scheduled to become part the $\mathbb{H}$-graph at the beginning of I_2, then u must have joined the network in I_1. In this case, u informs the node w responsible for u that it is leaving and then immediately leaves the network. The node w removes u from the set of nodes for which it is responsible. Recall that according to the definition given in Section 2.2, a leaving node is allowed to remain in the network for an additional T rounds, where T should be as small as possible. For all three cases we have $T = O(\log\log n)$.

Consider the execution of Algorithm 2.3 during I_2 for a specific Hamiltonian cycle. In Phase 1, a node sends the identifiers of the nodes for which it is responsible to randomly chosen nodes in the $\mathbb{H}$-graph. To generate a sufficient number of random nodes, the algorithm executes a polylogarithmic number of instances of the rapid node sampling algorithm for $\mathbb{H}$-graphs in parallel. We say a node is *active* if at least one identifier is sent to it during Phase 1. Otherwise, a node is *inactive*. In Phase 2, an active node locally arranges the identifiers it received into a random permutation. In Phases 3 and 4, the active nodes connect these permutations along the direction of the old Hamiltonian cycle to form a new Hamiltonian cycle. Thereby, the changes prescribed by the adversary during I_1 are reflected in the $\mathbb{H}$-graph at the beginning of I_3. Note that implementing Phase 3 of the algorithm in an efficient way requires some additional considerations. We will come back to this point when we analyze the running time of the algorithm.

The set of nodes in the $\mathbb{H}$-graph at the beginning of interval I_3 corresponds to the prescribed node set W_i where i is the last round of I_1. Similarly, the set of nodes in the $\mathbb{H}$-graph at the beginning of interval I_2 corresponds to the prescribed node set W_j where j is the last round of the interval before I_1. Let $n = |W_j|$ and $m = |W_i|$. According to our assumptions, we have $m \geq N$ and $n \geq N$. Hence, to show that a statement holds w.h.p. with regard to N, it is sufficient to show that it holds w.h.p. with regard to n or m. Furthermore, our model for churn implies the following relationship between n and m

$$n/\ell(n) \leq n/r^{c \log \log n} \leq m \leq n \cdot r^{c \log \log n} \leq n \cdot \ell(n).$$

With these observations in place, we are now ready to begin our analysis of the algorithm.

First, we show that the running time of the algorithm is indeed $O(\log \log n)$. Phase 1 takes $O(\log \log n)$ rounds according to Theorem 2.9, and Phase 2 and 4 require only a constant number of rounds. The critical part of the algorithm is Phase 3 in which we have to bridge the distance between two active nodes on a cycle. We define an *inactive segment* to be a sequence of consecutive inactive nodes on a cycle. Consider the execution of Algorithm 2.3 on a specific Hamiltonian cycle. We have the following lemma.

Lemma 2.13. *There is a constant c such that each inactive segment has length at most $2c \log n \cdot \ell(n)$, w.h.p.*

Proof. Consider a node u. According to Lemma 2.4, the probability that a specific random walk used in Phase 1 chooses u among the n nodes of the $\mathbb{H}$-graph is at least

$$\frac{1}{n} - \frac{1}{n^\alpha} \geq \frac{1}{2n},$$

where the inequality holds for $n \geq 2$ and $\alpha \geq 2$. Now consider a segment s of $2c \log n \cdot \ell(n)$ consecutive nodes on a cycle of n nodes. The probability that a specific random walk chooses a node in s is at least

$$\frac{2c \log n \cdot \ell(n)}{2n}.$$

Hence, for m independent random walks we have

$$\Pr[s \text{ inactive}] \leq \left(1 - \frac{c \log n \cdot \ell(n)}{n}\right)^m .$$

Since $m \geq n/\ell(n)$, we have

$$\left(1 - \frac{c \log n \cdot \ell(n)}{n}\right)^m \leq \left(1 - \frac{c \log n \cdot \ell(n)}{n}\right)^{n/\ell(n)} \leq e^{-c \log n},$$

where the last inequality uses the well-known inequality $(1 - 1/x)^x \leq 1/e$ with $x = n/(c \log n \cdot \ell(n))$. Applying the union bound over the n possible empty segments of length $2c \log n \cdot \ell(n)$ in the Hamiltonian cycle implies that there is no such empty segment, w.h.p. Therefore, there is also no longer empty segment, w.h.p. □

Applying the union bound over all $d/2$ Hamiltonian cycles implies that Lemma 2.13 holds for all Hamiltonian cycles in the $\mathbb{H}$-graph, w.h.p. Thereby, the distance between two active nodes on a cycle is at most polylogarithmic. Using pointer jumping, this distance can be bridged in $O(\log \log n)$ rounds. Thereby, the overall running time of the algorithm is indeed $O(\log \log n)$.

Next, we turn to the communication work induced by the algorithm. The parallel executions of the rapid node sampling algorithm require polylogarithmic communication work according to Theorem 2.9. To bound the communication work in the remaining parts of the algorithm, we have to bound the number of messages sent to a node in Phase 1. Consider the execution of Algorithm 2.3 on a specific Hamiltonian cycle. We have the following lemma.

Lemma 2.14. *There is a constant c such that at most $c \log n \cdot \ell(n)$ many messages are sent to each node in Phase 1, w.h.p.*

Proof. Consider a node u. According to Lemma 2.4, the probability that a specific random walk used in Phase 1 chooses u among the n nodes of the $\mathbb{H}$-graph is at most

$$\frac{1}{n} + \frac{1}{n^\alpha} \leq \frac{2}{n},$$

where the inequality holds for $\alpha \geq 1$. Let X be the number of messages sent to u in Phase 1. It is not hard to see that X is a sum of independent binary random variables such that

$$\mathrm{E}[X] \leq \frac{2m}{n} \leq 2\ell(n),$$

where the inequality holds since $m \leq n \cdot \ell(n)$. Define

$$\beta = \frac{2\ell(n)}{\mathrm{E}[X]}.$$

Note that $\beta \geq 1$. Hence, we can apply Chernoff bounds to get

$$\begin{aligned}\Pr[\, X \geq (1 + c\beta \log n) \cdot \mathrm{E}[X]\,] &\leq e^{-c\beta \log n \cdot \mathrm{E}[X]/3} \\ &= e^{-2c \log n \cdot \ell(n)/3}.\end{aligned}$$

Thereby, w.h.p. we have

$$\begin{aligned}X &< (1 + c\beta \log n) \cdot \mathrm{E}[X] \\ &\leq 2\ell(n) + 2c \log n \cdot \ell(n) \\ &\leq c' \log n \cdot \ell(n)\end{aligned}$$

for a constant c'. Applying the union bounds over all nodes in the $\mathbb{H}$-graph shows the lemma. □

By applying the union bound over all $d/2$ Hamiltonian cycles, it follows that the overall number of messages sent to a node in Phase 1 is polylogarithmic, w.h.p. It is not hard to see that thereby the overall communication work of the algorithm is polylogarithmic for each node.

Finally, we establish a series of lemmas to show that applying Algorithm 2.3 to each Hamiltonian cycle in a suitable $\mathbb{H}$-graph results in a new $\mathbb{H}$-graph that is suitable, w.h.p. We begin with a lemma that allows us to bound the probability of each outcome of the random experiment collectively performed by the nodes in Phase 1 of the algorithm. Such an outcome is an assignment of the m nodes that should form the new $\mathbb{H}$-graph to the n nodes in the current $\mathbb{H}$-graph.

Lemma 2.15. *Let G be a d-regular multigraph with $|\lambda_i| \leq 2\sqrt{d}$ for all $i > 1$. For any $\alpha \geq 1$ and any k such that $k \leq n^{\alpha-1}$ consider k independent simple random walks of length $t \geq 2\alpha \log_{d/4} n$. For any sequence of nodes $v_1, v_2, \ldots, v_k$ it holds*

$$\Pr\left[\,\forall i : \textit{random walk } i \textit{ ends at } v_i\,\right] \leq \left(1 + 2kn^{1-\alpha}\right) \frac{1}{n^k}.$$

Proof. Lemma 2.4 implies

$$\Pr\left[\,\forall i : \text{random walk } i \text{ ends at } v_i\,\right] \leq \left(\frac{1}{n} + \frac{1}{n^\alpha}\right)^k.$$

By applying the binomial theorem we get

$$\begin{aligned}\left(\frac{1}{n} + \frac{1}{n^\alpha}\right)^k &= \sum_{i=0}^{k} \binom{k}{i} \left(\frac{1}{n}\right)^{k-i} \left(\frac{1}{n^\alpha}\right)^i \\ &= \sum_{i=0}^{k} \binom{k}{i} \left(\frac{1}{n}\right)^{k+(\alpha-1)i}.\end{aligned}$$

To establish an upper bound on this sum, we show the following relationship between the $(i+1)$-th and the i-th term of the sum for $i \geq 1$:

$$\begin{aligned} && \binom{k}{i+1}\left(\frac{1}{n}\right)^{k+(\alpha-1)(i+1)} &\leq \frac{1}{2}\binom{k}{i}\left(\frac{1}{n}\right)^{k+(\alpha-1)i} \qquad (2.1)\\ \iff && \binom{k}{i+1}\left(\frac{1}{n}\right)^{\alpha-1} &\leq \frac{1}{2}\binom{k}{i} \\ \iff && \frac{k-i}{i+1} &\leq \frac{1}{2}n^{\alpha-1} \\ \iff && k &\leq \frac{1}{2}n^{\alpha-1}(i+1)+i. \end{aligned}$$

Note that the last inequality is satisfied since $k \leq n^{\alpha-1}$ and $i \geq 1$ according to our assumptions. Equation 2.1 implies

$$\begin{aligned} \sum_{i=0}^{k}\binom{k}{i}\left(\frac{1}{n}\right)^{k+(\alpha-1)i} &\leq \left(\frac{1}{n}\right)^{k} + \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{i-1} k\left(\frac{1}{n}\right)^{k+\alpha-1} \\ &= \left(\frac{1}{n}\right)^{k} + \left(\frac{1}{n}\right)^{k} kn^{1-\alpha}\cdot\sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{i} \\ &\leq \left(1+2kn^{1-\alpha}\right)\frac{1}{n^k}. \end{aligned}$$

□

In the remainder of this section, we assume that $\alpha \geq 3$. For n sufficiently large, this assumption implies

$$m \leq n \cdot \ell(n) \leq n^2 \leq n^{\alpha-1}.$$

Therefore, we can apply Lemma 2.15 with $k = m$ to bound the probability of an outcome of Phase 1 of the algorithm.

Consider a cycle of the $\mathbb{H}$-graph before the execution of Algorithm 2.3. Let $u_1, u_2, \ldots, u_n$ be the nodes of the cycle where we pick u_1 arbitrarily and u_{i+1} is the successor of u_i for all i such that $1 \leq i \leq n-1$. We define

$$X = \left\{ (x_1, x_2, \ldots, x_n) \mid \forall i : x_i \in \mathbb{Z}_{\geq 0} \text{ and } \sum_i x_i = m \right\}.$$

For $x \in X$ with $x = (x_1, x_2, \ldots, x_n)$ let E_x be the event that in Phase 1 of the algorithm, x_i messages are sent to node u_i. Let E_C be the event that Algorithm 2.3 generates the directed cycle C. We have the following lemma.

Lemma 2.16. *For a cycle C and an $x \in X$ with $x = (x_1, x_2, \dots, x_n)$ we have*

$$\Pr[E_C \cap E_x] \le \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m} \cdot \frac{1}{x_1! \cdot x_2! \cdot \dots \cdot x_n!}.$$

Proof. The cycle constructed by the algorithm is determined by the outcome of Phase 1 and 2 of the algorithm. Let Ω_1 be the set of possible outcomes of Phase 1 and let Ω_2 be the set of possible outcomes of Phase 2 when considering all nodes. As we argued above, an outcome in Ω_1 is an assignment of the m nodes that should form the new $\mathbb{H}$-graph to the n nodes in the current $\mathbb{H}$-graph. An outcome in Ω_2 determines the random permutations generated at the nodes in the current $\mathbb{H}$-graph. Note that Ω_1 can be interpreted in isolation as a sample space of a probability space while Ω_2 cannot directly be interpreted in this way because the exact nature of the random experiment collectively performed by the nodes in Phase 2 depends on the outcome of Phase 1. The overall random experiment performed by the nodes over both Phase 1 and 2 produces an outcome from the sample space $\Omega_1 \times \Omega_2$.

Define

$$X^* = \{\, (x_1, x_2, \dots, x_n) \in X \mid \exists i : x_i = m \,\}.$$

We distinguish two cases. First, suppose $x \in X \setminus X^*$. Let Ω_1' be the set of outcomes $\omega_1 \in \Omega_1$ for which there is an outcome $\omega_2 \in \Omega_2$ such that $(\omega_1, \omega_2) \in E_C \cap E_x$. We have $|\Omega_1'| = m$. To see this, consider some index j such that $x_j > 0$, and let $V(u_j)$ be the set of identifiers sent to u_j in Phase 1. For ω_1 to lie in Ω_1', the set $V(u_j)$ must consist of x_j consecutive nodes from C. Hence, there are m possible choices for the set $V(u_j)$. Since x is fixed, each of these choices directly implies the assignment of the remaining nodes in C to the nodes in the current cycle. Therefore, there are exactly m outcomes in Ω_1'. Let $E_1 = \Omega_1' \times \Omega_2$. Lemma 2.15 implies

$$\Pr[E_1] \le m \cdot \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{1}{n^m}.$$

For each outcome in $\omega_1 \in \Omega_1'$ there is exactly one outcome $\omega_2 \in \Omega_2$ such that $(\omega_1, \omega_2) \in E_C \cap E_x$: Each node u_i that receives at least one identifier in Phase 1 has to pick one specific permutation among the $x_i!$ possible permutations of the x_i identifiers u_i received. Therefore, we have

$$\Pr[E_C \cap E_x \mid E_1] = \frac{1}{x_1! \cdot x_2! \cdot \dots \cdot x_n!}.$$

Overall, we have

$$\begin{aligned}\Pr[E_C \cap E_x] &= \Pr[(E_C \cap E_x) \cap E_1] \\ &= \Pr[E_1] \cdot \Pr[E_C \cap E_x \mid E_1] \\ &\le \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m} \cdot \frac{1}{x_1! \cdot x_2! \cdot \dots \cdot x_n!}.\end{aligned}$$

Now suppose $x \in X^*$. In this case, there is exactly one outcome $\omega_1 \in \Omega_1$ for which there exists an outcome $\omega_2 \in \Omega_2$ such that $(\omega_1, \omega_2) \in E_C \cap E_x$. Specifically, ω_1 must be such that the node u_i with $x_i = m$ receives all m identifiers sent in Phase 1. Let $E_1 = \{\omega_1\} \times \Omega_2$. According to Lemma 2.15 we have

$$\Pr[E_1] \leq \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{1}{n^m}.$$

Additionally to the occurrence of ω_1, we need the outcome ω_2 of Phase 2 to be such that the node u_i chooses one of the m permutations that results in C among the $m!$ permutations of the m identifiers u_i received. Hence, we have

$$\Pr[E_C \cap E_x \mid E_1] = \frac{m}{m!}.$$

Overall, we have

$$\begin{aligned}
\Pr[E_C \cap E_x] &= \Pr[(E_C \cap E_x) \cap E_1] \\
&= \Pr[E_1] \cdot \Pr[E_C \cap E_x \mid E_1] \\
&\leq \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{1}{n^m} \cdot \frac{m}{m!} \\
&= \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m} \cdot \frac{1}{x_1! \cdot x_2! \cdot \dots \cdot x_n!},
\end{aligned}$$

where the last equality follows from the definition of X^*. □

We can now bound the probability that the algorithm generates a specific directed cycle.

Lemma 2.17. *For a cycle C it holds*

$$\Pr\left[E_C\right] \leq \left(1 + 2mn^{1-\alpha}\right) \frac{1}{(m-1)!}.$$

Proof. By Lemma 2.16 and the law of total probability we have

$$\begin{aligned}
\Pr[E_C] &= \sum_{x \in X} \Pr[E_C \cap E_x] \\
&\leq \sum_{x \in X} \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m} \cdot \frac{1}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \\
&= \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m \cdot m!} \cdot \sum_{x \in X} \frac{m!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \\
&= \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m \cdot m!} \cdot \sum_{x \in X} \binom{m}{x_1, x_2, \dots, x_n} \\
&= \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{m}{n^m \cdot m!} \cdot n^m \\
&= \left(1 + 2mn^{1-\alpha}\right) \cdot \frac{1}{(m-1)!},
\end{aligned}$$

where the penultimate line follows from the multinomial theorem and the definition of X. □

Finally, we can show that the algorithm generates a suitable $\mathbb{H}$-graph.

Lemma 2.18. *The execution of Algorithm 2.3 on each Hamiltonian cycle of a suitable $\mathbb{H}$-graph results in a suitable $\mathbb{H}$-graph, w.h.p.*

Proof. An $\mathbb{H}$-graph is a combination of $d/2$ Hamiltonian cycles. According to Corollary 2.3, an $\mathbb{H}$-graph generated by choosing $d/2$ directed Hamiltonian cycles over m nodes independently and uniformly at random is suitable w.h.p. for d and m sufficiently large. Let K be the set of tuples $(C_1, C_2, \ldots, C_{d/2})$, where each C_i is a directed Hamiltonian cycle over m nodes, such that the $\mathbb{H}$-graph corresponding to the combination of the cycles C_i is *unsuitable*. Corollary 2.3 implies that for directed cycles chosen uniformly at random, we have

$$\Pr[G \text{ unsuitable}] = \sum_{(C_1,C_2,\ldots,C_{d/2})\in K} \left(\frac{1}{(m-1)!}\right)^{d/2} \leq m^{-c} \tag{2.2}$$

for a given constant c. According to Lemma 2.17, Algorithm 2.3 chooses the cycles such that

$$\begin{aligned}\Pr[G \text{ unsuitable}] &\leq \sum_{(C_1,C_2,\ldots,C_{d/2})\in K} \left(\left(1+2mn^{1-\alpha}\right)\cdot\frac{1}{(m-1)!}\right)^{d/2}\\ &\leq \left(1+2mn^{1-\alpha}\right)^{d/2}\cdot \sum_{(C_1,C_2,\ldots,C_{d/2})\in K}\left(\frac{1}{(m-1)!}\right)^{d/2}\\ &\leq \left(1+2mn^{1-\alpha}\right)^{d/2}\cdot m^{-c},\end{aligned}$$

where the last inequality follows by the inequality given in Equation 2.2. For n sufficiently large, we have

$$m \leq n\cdot \ell(n) \leq n^2.$$

Hence, for $\alpha \geq 3$ we have

$$\left(1+2mn^{1-\alpha}\right)^{d/2} \leq 3^{d/2},$$

which implies

$$\Pr[G \text{ unsuitable}] \leq 3^{d/2}\cdot m^{-c}.$$

Since d is a constant, the lemma holds for m sufficiently large. □

Note that as long as the algorithm succeeds to update the $\mathbb{H}$-graph into a new suitable $\mathbb{H}$-graph, the overall network remains weakly connected. The above arguments concerning the running time and the communication work induced by the algorithm together with an inductive application of Lemma 2.18 imply the following theorem.

Theorem 2.19. *For any constant r the algorithm maintains the connectivity of the network under adversarial churn with a churn rate of r for a polynomial number of rounds in N. The communication work for every node in every round is polylogarithmic.*

As we argued throughout this section, the algorithm actually guarantees stronger properties than just weak connectivity. Most notably, the network always contains an $\mathbb{H}$-graph over a large subset of the nodes as a subgraph. At any point in time, the nodes in this $\mathbb{H}$-graph correspond to a node set W_i prescribed by the adversary at most $O(\log\log n)$ rounds ago. Furthermore, the degree of a node in the overlay network remains polylogarithmic at all times.

2.6 Adversarial DoS-Attacks

Next we present an overlay network based on the hypercube that is resistant against constantly changing DoS-attacks. We assume that the node set is static. For a given ε such that $0 < \varepsilon \leq 1$ we consider a $(1-\varepsilon)$-bounded t-late adversary where $t = \Theta(\log\log n)$. We organize the nodes of the network into groups of logarithmic size and let each group simulate one node of a hypercube. Specifically, if we are given a network of n nodes, we use a d-dimensional hypercube where d is the largest integer such that $2^d \leq n/(c\log n)$ for some constant c that is chosen in the analysis. To distinguish the nodes of the hypercube from the physical nodes in the network, we refer to them as *supernodes*. We denote the number of supernodes in the hypercube as $N = 2^d$. For a supernode u we use $R(u)$ to refer to the group of nodes simulating u. Each physical node is part of exactly one group. We say that two groups are neighbors if their corresponding supernodes are neighbors. The topology of the overlay network is derived from the hypercube as follows: The nodes in a group are connected to a clique, and nodes of neighboring groups form a complete bipartite graph.

It is not hard to see that as long as the adversary does not block an entire group, the network induced by the non-blocked nodes remains strongly connected. Therefore, our goal is to make it difficult for the adversary to block all nodes in a group. To achieve this goal, we randomly reassign the physical nodes to the groups every $\Theta(\log\log n)$ rounds so that a t-late adversary for $t = \Theta(\log\log n)$ never knows which nodes currently form a group. We assume that initially the network is such that each node chose its group independently and uniformly at random and that the nodes are connected as described above. We reassign the nodes to the groups in the following way.

First, the groups simulate the rapid node sampling algorithm on the hypercube. We denote the state of the rapid node sampling algorithm for a supernode u as $s(u)$. The state $s(u)$ contains all variables used by the algorithm and a counter that represents the current iteration and phase. It further contains the identifiers of all nodes in $R(u)$ and the identifiers of all nodes in $R(v)$ for every supernode v stored by u. The simulation proceeds in *steps*. One step corresponds to one round in the execution of the rapid node sampling algorithm on the hypercube. The execution of a step takes two rounds. At the beginning, all nodes in $R(u)$ are assumed to know the initial state of u. A step for a supernode u is performed as follows.

- *Simulation round*: First, all non-blocked nodes in $R(u)$ receive messages containing the current state $s(u)$ of the supernode. The nodes adopt the state given in these messages. Furthermore, the nodes receive the messages that u is supposed to receive at the beginning of the round of the rapid node sampling algorithm corresponding to the current step. Naturally, the message reception is skipped in the step corresponding to the first round of the rapid node sampling algorithm.

  After receiving all messages, each non-blocked node x simulates the local computation of u and sends a message m_x to all nodes in $R(u)$ containing the new state of u from the viewpoint of x, including all messages that u is supposed to send out in that round. Note that the messages m_x might differ between nodes in a group because of different random decisions.

- *Synchronization round*: Every non-blocked node x in $R(u)$ first receives all messages sent in the simulation round and chooses among these the message m_y of the node y of lowest identifier. Note that y is unique and it is the same node for all non-blocked nodes in $R(u)$. The node x adopts the new state $s(u)$ as given in m_y and for each message m that u is supposed to send to some supernode v, x sends m to all nodes in $R(v)$. Then, x sends the state $s(u)$ to all nodes in $R(u)$.

We perform the given simulation of the rapid node sampling algorithm k times in parallel for a constant $k \in \mathbb{N}$ determined in the analysis. It follows from Theorem 2.12 that at the end of the simulation, the collective states of a supernode u over all k instances contain at least $k \log n$ supernodes (represented by their groups) that were chosen uniformly at random from the hypercube. Suppose that for every supernode u we have $|R(u)| \leq k \log n$. Let $R(u) = \{x_1, x_2, \ldots, x_\ell\}$ where we assume $\mathrm{id}(x_i) < \mathrm{id}(x_j)$ if $i < j$, and let $v_1, v_2, \ldots, v_\ell$ be the first ℓ supernodes that were chosen. We reorganize the physical nodes into groups using four additional rounds.

- First, every non-blocked node in $R(u)$ assigns x_i to $R(v_i)$ for every i by sending messages to all nodes in $R(v_i)$ informing them about x_i.
- Then, every non-blocked node in $R(u)$ collects the nodes sent to it in the previous round to form the set $R'(u)$, which is the new group representing u. It then sends $R'(u)$ to all nodes in $R(u)$ and to all nodes in $R(v)$ where v is a neighbor of u.
- After collecting $R'(u)$ and $R'(v)$ for all neighbors v of u, each non-blocked node in $R(u)$ sends $R'(u)$ and $R'(v)$ for all neighbors v of u to all nodes in $R'(u)$.
- Finally, each non-blocked node x collects $R'(u')$ of the new supernode u' that was assigned to x and also $R'(v)$ for all neighbors v of u'.

Note that in the last round, a non-blocked node x that was newly assigned to a supernode u learns the identifiers of the nodes in $R(u)$ and the identifiers of the nodes in $R(v)$ for every neighbor v of u.

Overall, it is not hard to see that if $|R(u)| \leq k \log n$ for every supernode u and each group contains at least one non-blocked node in every round then the algorithm can correctly simulate the random node sampling algorithm and reassign the nodes to the groups. The simulation of the node sampling algorithm takes at most twice as many rounds as the original algorithm, and the reassignment of the nodes to the groups takes a constant number of rounds. Thereby, it follows from Theorem 2.12 that the algorithm takes $O(\log \log n)$ rounds to reorganize the groups such that each node is assigned to a group independently and uniformly at random.

Let t be the exact number of rounds required by the algorithm. We constantly reorganize the groups every t rounds by iteratively executing the algorithm. Thereby, a $2t$-late adversary has no knowledge pertaining to the execution of the algorithm that led to the current composition of the groups. As a consequence, the assignment of nodes to groups at any given time is completely random from the perspective of the adversary. This observation represents the main insight required for the following analysis of the algorithm.

First, we establish bounds on the size of a group. The following lemma holds both before the first execution of the algorithm and after an execution of the algorithm.

Lemma 2.20. *For any δ such that $0 < \delta < 1$ and every supernode u we have w.h.p. that*

$$(1-\delta)\frac{n}{N} < |R(u)| < (1+\delta)\frac{n}{N}.$$

Proof. Initially, the network is such that each node chose its group independently and uniformly at random by assumption. For a subsequent reorganization of the groups by the algorithm, Theorem 2.12 implies that each node is assigned to a group independently and uniformly at random. Therefore, we have for each supernode u that $\mathrm{E}[\,|R(u)|\,] = n/N$. Recall that we choose N to be the largest power of 2 such that $N \leq n/(c \log n)$. We use this inequality together with Chernoff bounds to get

$$\Pr\left[|R(u)| \leq (1-\delta)\frac{n}{N}\right] \leq e^{-\frac{\delta^2 n}{2N}} \leq e^{-\delta^2 c \log n/2}$$

and

$$\Pr\left[|R(u)| \geq (1+\delta)\frac{n}{N}\right] \leq e^{-\frac{\delta^2 n}{3N}} \leq e^{-\delta^2 c \log n/3}.$$

Therefore, the group size $|R(u)|$ is bounded as claimed, w.h.p. Applying the union bound over all supernodes implies the lemma. □

Recall that we defined t as the number of rounds required by the algorithm to reorganize the groups. We now show that a $2t$-late adversary cannot block an entire group.

Lemma 2.21. *For any ε such that $0 < \varepsilon \leq 1$, a $(1-\varepsilon)$-bounded $2t$-late adversary does not block all nodes of a group in any round during a single execution of the algorithm, w.h.p.*

Proof. If $\varepsilon = 1$ then the adversary cannot block any nodes and the lemma holds trivially. So suppose $\varepsilon < 1$. For each of the $t = \Theta(\log\log n)$ rounds of the execution of the algorithm, the adversary can block a subset of the nodes of size $(1-\varepsilon)n$. As we argued above, the assignment of the nodes to the groups is random from the perspective of the adversary. Thereby, also the assignment of the *blocked* nodes to the groups is random. Consider one round of the execution of the algorithm and let u be a supernode. Define X_i to be a binary random variable such that $X_i = 1$ if and only if the i-th blocked node is in group $R(u)$. We have $\mathrm{E}[X_i] = 1/N$. Let X be the number of blocked nodes in $R(u)$. Then

$$X = \sum_{i=1}^{(1-\varepsilon)n} X_i,$$

which implies

$$\mathrm{E}[X] = \sum_{i=1}^{(1-\varepsilon)n} \mathrm{E}[X_i] = (1-\varepsilon)\frac{n}{N}.$$

Define

$$\delta = \frac{\varepsilon}{2(1-\varepsilon)}$$

so that

$$\Pr\left[X \geq \left(1-\frac{\varepsilon}{2}\right)\frac{n}{N}\right] = \Pr\left[X \geq (1+\delta)(1-\varepsilon)\frac{n}{N}\right].$$

Note that $\delta > 0$. Therefore, we can apply Chernoff bounds to get

$$\begin{aligned}\Pr\left[X \geq (1+\delta)(1-\varepsilon)\frac{n}{N}\right] &= e^{-\min\{\delta^2,\delta\}\cdot(1-\varepsilon)n/(3N)} \\ &\leq e^{-\min\{\delta^2,\delta\}\cdot(1-\varepsilon)c\log n/3},\end{aligned}$$

where the inequality holds because $N \leq n/(c \log n)$. Since both δ and ε are positive constants, this equation implies that we can choose the constant c such that, w.h.p, it holds

$$X < \left(1 - \frac{\varepsilon}{2}\right)\frac{n}{N}.$$

According to Lemma 2.20, we have that for any $0 < \delta' < 1$ it holds, w.h.p., that

$$|R(u)| > (1-\delta')\frac{n}{N}.$$

Therefore, the number of non-blocked nodes in $R(u)$ is

$$\begin{aligned}|R(u)| - X &> \left[(1-\delta') - \left(1 - \frac{\varepsilon}{2}\right)\right]\frac{n}{N} \\ &= \left(\frac{\varepsilon}{2} - \delta'\right)\frac{n}{N} \\ &\geq \left(\frac{\varepsilon}{2} - \delta'\right)c\log n,\end{aligned}$$

where the first inequality holds w.h.p. and the last inequality again follows because $N \leq n/(c\log n)$. Hence, if we choose $\delta' < \varepsilon/2$, we get $|R(u)| - X > 0$ so that the adversary does not block all nodes in $R(u)$, w.h.p. Applying the union bound over all nodes and all $t = O(\log\log n)$ rounds of the execution of the algorithm shows that the adversary does not block an entire group in any round of the execution, w.h.p. □

We conclude our analysis with the following theorem.

Theorem 2.22. *For any ε such that $0 < \varepsilon \leq 1$ the algorithm maintains the connectivity of the network under DoS-attacks by a $(1-\varepsilon)$-bounded t-late adversary where $t = \Theta(\log\log n)$ for a polynomial number of rounds. The communication work for every node in every round is polylogarithmic.*

Proof. For the algorithm to work correctly, we have to choose the constant c according to the applications of Chernoff bounds throughout this section. Our choice of N implies that $N > n/(2c\log n)$. Together with Lemma 2.20, this implies that $|R(u)| < 4c\log n$ for every supernode u, w.h.p. Therefore, we can choose $k = 4c$. With the choices of these parameters in place, the theorem follows by inductively applying Lemmas 2.20 and 2.21. The bound on the communication work follows from the upper bound on $|R(u)|$ given above and Theorem 2.12. □

The presented algorithm guarantees stronger properties than just weak connectivity among the non-blocked nodes. Specifically, the non-blocked nodes actually form a *strongly* connected graph in every round. Furthermore, the algorithm allows the nodes in the network to simulate a hypercube in a DoS-resistant manner. We can execute any given algorithm on this hypercube through simulation in a way similar to the simulation of the rapid node sampling algorithm described above. Thereby, the presented approach can be used to make the execution of a distributed algorithm resistant against DoS-attacks at the cost of a constant-factor slow-down and increased communication work.

2.7 Outlook

The idea of constantly reorganizing an overlay network with the help of random walks in order to make the network robust against adversarial attacks seems to be quite powerful. While this work and the work of Augustine et al. [Aug+15] represent a first foray into leveraging this idea, both works leave ample room for future research in this direction. As a direct extension of the results presented in this chapter, one can envision designing networks that are robust against attacks beyond adversarial churn and DoS-attacks such as Eclipse-attacks [Sin+06], for example. Also a combination of different attacks could be considered: While we investigated churn and DoS-attacks separately in this chapter, the publication underlying this work [DGS16] already presented a first result on algorithms that can endure both kinds of attacks at the same time. One could also investigate the application of the presented techniques to related problems in the general area of overlay networks. For example, it is conceivable to use these techniques to improve current constructions of DoS-resistant DHTs [ESS14] or to make anonymous routing systems such as Tor [DMS04] robust against DoS-attacks. Finally, as we already stated in Section 2.1, it would be interesting to further investigate the relationship between this work and the work of Augustine et al. [Aug+15].

One of the fundamental operations used by the algorithms presented in this chapter is to sample nodes uniformly at random from a network by performing random walks. The rapid node sampling algorithms presented in Section 2.4 achieve this task in a running time that is exponentially faster than standard random walks. Since sampling nodes from a network is a widely used operation in network algorithms, this improvement in running time could translate to other works on overlay networks. The low running time of the rapid node sampling algorithms stems from their use of pointer jumping. We think that this technique from parallel computing could inspire further, new algorithms in the area of overlay networks. In fact, we present another application of pointer jumping in overlay networks in Chapter 4 of this thesis.

Chapter 3

Self-Stabilizing Metric Graphs

The results of the previous chapter and similar results from the literature demonstrate that it is possible to design algorithms for overlay networks that maintain a certain topology even under massive adversarial attacks. Still, the question remains, what should be done if attacks or faults are so severe that a given algorithm cannot endure them. In this case it would be desirable to have an algorithm that eventually recovers the network topology from the state left behind by a transient fault or a serious attack without outside intervention. This is the fundamental idea behind *topological self-stabilization*.

For an algorithm to construct a topology in a self-stabilizing manner, it has to guarantee that when starting from an arbitrary network state in which all nodes form a single weakly-connected component, the network eventually returns to the desired topology (*convergence*) and then remains in the desired topology (*closure*). Thereby, the network eventually recovers from any fault that leaves the network weakly connected as long as no further faults occur.

There is a plethora of work on the self-stabilizing construction of specific network topologies from simple structures such as line graphs and rings to more complex topologies such as De Bruijn graphs and Skip graphs. Next to these topology-specific results, there has also been some work on generic approaches for the self-stabilizing construction of overlay networks like the Transitive Closure Framework by Berns et al. [BGP13], which can construct a wide array of topologies that are locally checkable.

In this chapter, we present a new algorithm in the domain of topological self-stabilization: Our algorithm constructs the graph corresponding to a given metric specified via a distance oracle. The graph corresponding to a metric (or just *metric graph*) is the unique minimal undirected graph such that for any pair of nodes the length of a shortest path between the nodes corresponds to the distance between the nodes according to the metric. Since every undirected graph corresponds to a metric, our algorithm can be seen as a universal approach to self-stabilizing graph construction for situations in which the graph can be efficiently encoded as a metric. To the best of our knowledge, our algorithm is the first self-stabilizing algorithm for the construction of general metric graphs.

Our algorithm works for both synchronous and asynchronous activations of the nodes. By modifying the algorithm slightly for the synchronous case, we achieve a linear running time for the construction of the metric graph. Furthermore, the modified version of the algorithm guarantees that after stabilization in the synchronous case, the memory overhead and the number of messages sent and received per round at every node drop to a constant within a linear number of rounds. At the core of our algorithm lies a technique that uses a directed cycle covering all nodes to guide the construction of the metric graph. We think that this underlying technique could potentially have applications beyond the construction of metric graphs and, therefore, could be of independent interest.

Underlying Publications This chapter is based on the following publication and a corresponding journal article [GLS17] that will appear in Theory of Computing Systems.

> R. Gmyr, J. Lefèvre, and C. Scheideler. "Self-stabilizing Metric Graphs". In: *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, see [GLS16].

Outline We begin in Section 3.1 with a brief overview of the related work in the area of topological self-stabilization. We then present our model in Section 3.2 and formally define the problem of constructing a metric graph in a self-stabilizing manner in Section 3.3. In Section 3.4 we describe an algorithm that solves this problem under both synchronous and asynchronous executions. We then analyze the algorithm in Section 3.5. Finally, we conclude the chapter in Section 3.6 with some remarks on further applications of the general ideas underlying this work.

3.1 Related Work

The concept of *self-stabilization* was first introduced in the seminal work of Dijsktra [Dij74]. Since then, a rich body of research on self-stabilizing algorithms has been established. In the area of *topological self-stabilization* there is an abundance of work on the construction of specific network topologies. For example, there are algorithms for the self-stabilizing construction of line graphs [ORS07], rings [CF05; SR05], spanning trees [AK93; Gär03], De Bruijn graphs [RSS11], Chord graphs [KKS14], and Skip graphs [Jac+09; CNS12].

Besides these topology-specific approaches, Berns et al. [BGP13] introduced the Transitive Closure Framework, which is a general framework for the self-stabilizing construction of *locally-checkable* network topologies. Locally-checkable topologies have the property that when the network is not in the desired topology, there always exists a node that can locally detect this fact. Such a node can then initiate the recovery of the desired topology. Unfortunately, a metric graph is in general not locally checkable, so we cannot apply the Transitive Closure Framework to solve the problem at hand.

For *certain* metrics there are algorithms in the literature that can be used for the construction of graphs that *approximate* that metric. For example, the two-dimensional Euclidean metric can be approximated using *Delaunay triangulations*: Consider a finite set of points in the Euclidean plane together with the Delaunay triangulation of these points. The length of a shortest path in the Delaunay triangulation between any two points is at most 2.42 times the Euclidean distance between the two points, see [KG89; KG92]. Constructing the graph corresponding to the Delaunay triangulation of a set of points in a self-stabilizing manner can be achieved using the algorithm of Jacob et al. [Jac+12]. While this example demonstrates that there are, in fact, algorithms that offer an *approximate* solution for *certain* metrics, we are not aware of any previous algorithm for the construction of the *exact* metric graph of an *arbitrary* metric.

3.2 Model

We represent an *overlay network* as a directed graph $G = (V, E)$ and define $n = |V|$. Each node u has a unique *identifier* $u.\mathrm{id} \in [0, 1)$. Identifiers are immutable and cannot be corrupted. We simply write u instead of $u.\mathrm{id}$ when it is clear from the context that we refer to the identifier of u. The *edge set* E is defined as

$$E = \{\, (u, v) \mid u \in V \text{ stores } v.\mathrm{id} \text{ in its local memory} \,\}.$$

We use a variant of the standard message-passing model that allows for a clean presentation of both the algorithm and the proofs. The computation proceeds in *rounds*. In each round a subset of the nodes is activated. An activated node u can execute an arbitrary local computation and send a message to each node v such that $(u, v) \in E$. Messages are received at the end of a round. All nodes (including the nodes that are not activated) receive messages and update their local memories accordingly at the end of a round. In the presented algorithm, received messages only cause simple changes on local variables; a received message never triggers new messages to be sent immediately. In their computation, the nodes can use the variables stored in their local memory and a *distance oracle* that provides access to a prescribed metric d. Given two identifiers u and v, the oracle returns the distance $d(u, v)$ between the corresponding nodes.

We consider both *synchronous* and *asynchronous* executions. In a synchronous execution, each node is activated in every round. In an asynchronous execution, in each round some non-empty subset of the nodes is activated. We assume that the activation in an asynchronous execution is *fair* in that, starting at any round, each node is activated eventually. We formally specify an algorithm as a set of *rules*. A rule has the form

$$\langle \text{label} \rangle : \langle \text{guard} \rangle \longrightarrow \langle \text{commands} \rangle.$$

A $\langle$label$\rangle$ constitutes the unique name of a rule, a $\langle$guard$\rangle$ is a Boolean predicate over variables of the node, and $\langle$commands$\rangle$ consists of a sequence of instructions. Upon activation, a node sequentially checks the guards of all rules in the given order. If a guard is satisfied, the corresponding commands are executed. Additionally, we allow a node to execute rules whenever it receives a message. A rule can be executed multiple times during an activation of a node.

The *configuration* of the network c_i in round i consists of the contents of the local memories of the nodes at the beginning of round i. Since the messages sent during a round are received and processed at the end of that round, there are no messages in transit between rounds. Consequently, we do not have to consider messages in the definition of a configuration. Let C be the set of all configurations in which G is weakly connected and in which no node stores an invalid identifier. An algorithm is *self-stabilizing* with respect to a set of *legal configurations* $L \subseteq C$ if starting from any configuration $c \in C$ the computation eventually reaches a configuration in L (*convergence*) and then stays in L (*closure*).

3.3 Problem Statement

We assume that the network contains a set of directed *marked edges*. We define D to be the set of undirected edges induced by the directed marked edges, i.e., D contains an edge $\{u, v\}$ whenever the network contains the directed marked edges (u, v) and (v, u). We say D is *valid* if for every directed marked edge (u, v), the edge (v, u) is also marked. Let $d : V \times V \to \mathbb{R}^+$ be a finite metric and let $\overline{D}$ be the set of undirected edges of the graph corresponding to the metric d. Starting from any configuration in which $G = (V, E)$ is weakly connected and in which no node stores any invalid identifiers, the network has to reorganize such that eventually D is valid and $D = \overline{D}$.

To formally define $\overline{D}$, we introduce some notation. Let $(u_0, u_1, \ldots, u_k)$ be a sequence of nodes. We write $(u_0, u_1, \ldots, u_k) \in \overline{D}^*$ whenever the sequence of nodes forms a path with respect to $\overline{D}$, i.e., when $\{u_i, u_{i+1}\} \in \overline{D}$ for all $0 \leq i \leq k - 1$. Furthermore, we define

$$d(u_0, u_1, \ldots, u_k) = \sum_{i=0}^{k-1} d(u_i, u_{i+1}).$$

We formally require $\overline{D}$ to satisfy the following two conditions.

1. *Metric Graph*: For all $u, v \in V$ it holds

$$\exists (u, u_1, \dots, u_k, v)_{k \geq 0} \in \overline{D}^* : d(u, u_1, \dots, u_k, v) = d(u, v).$$

2. *Minimality*: For all $u, v \in V$ it holds

$$\Big[\exists (u, u_1, \dots, u_k, v)_{k \geq 1} \in \overline{D}^* : d(u, u_1, \dots, u_k, v) = d(u, v) \Big] \Rightarrow \{u, v\} \notin \overline{D}.$$

The first condition means that for any pair of nodes in the graph there exists a path between the nodes whose length corresponds to the distance between the nodes according to the metric. We refer to this condition as the *Metric Graph* condition. The second condition implies that the graph has to be the smallest for the inclusion. Accordingly, we refer to the second condition as the *Minimality* condition. Note that the graph defined by these conditions corresponds to the classical definition of a metric graph where the weights of the edges of the graph are implicitly defined by the pairwise distances given by the metric. It is not hard to prove that such a graph exists and is unique. We define the set L of legal configurations as the set of configurations such that D is valid and $D = \overline{D}$.

While the above definition is close to the intuition of what it means for a graph to represent a metric, the equivalent condition given in Lemma 3.1 below will prove more useful.

Lemma 3.1. *The Metric Graph condition and the Minimality condition together are equivalent to the following condition. For all $u, v \in V$ it holds*

$$\{u, v\} \notin \overline{D} \iff \exists w \in V \setminus \{u, v\} : \{u, w\} \in \overline{D} \wedge d(u, w, v) = d(u, v).$$

Proof. We first assume that the Metric Graph condition and the Minimality condition hold and show that the condition given in the lemma holds. Let $u, v \in V$. Suppose that $\{u, v\} \notin \overline{D}$. The Metric Graph condition implies the existence of a path $(u, u_1, \dots, u_k, v)_{k \geq 0} \in \overline{D}^*$ such that $d(u, u_1, \dots, u_k, v) = d(u, v)$. So we have $u_1 \in V \setminus \{u, v\}$, $\{u, u_1\} \in \overline{D}$, and $d(u, u_1, v) = d(u, v)$. Therefore, the condition given in the lemma holds. Now suppose $\{u, v\} \in \overline{D}$. The Minimality condition implies that for any path $(u, u_1, \dots, u_k, v)_{k \geq 1} \in \overline{D}^*$ we have $d(u, u_1, \dots, u_k, v) > d(u, v)$. So we have $d(u, u_1, v) > d(u, v)$ for all $u_1 \in V \setminus \{u, v\}$. Therefore, again the condition given in the lemma holds.

We now assume that the condition given in the lemma holds. We first show that the Metric Graph condition holds, i.e., for any pair of nodes u, v there is a path of length $d(u, v)$ between u and v. If $\{u, v\} \in \overline{D}$ then this trivially holds. If $\{u, v\} \notin \overline{D}$ then there is a node $w \in V \setminus \{u, v\}$ such that $\{u, w\} \in \overline{D}$ and $d(u, w, v) = d(u, v)$. We have $d(u, v) = d(u, w, v) = d(u, w) + d(w, v) > d(w, v)$. Therefore, the Metric Graph condition holds by an inductive argument. Finally, we show that the Minimality condition holds. So suppose there is a path $(u, u_1, \ldots, u_k, v)_{k \geq 1} \in \overline{D}^*$ such that $d(u, u_1, \ldots, u_k, v) = d(u, v)$. This implies in particular that there is a node $u_1 \in V \setminus \{u, v\}$ such that $\{u, u_1\} \in \overline{D}$ and $d(u, u_1, v) = d(u, v)$. It follows that $\{u, v\} \notin \overline{D}$ and, therefore, the Minimality condition holds. □

3.4 Algorithm

The algorithm consists of three parts. The first part organizes the nodes into a directed cycle in which the nodes are ordered by increasing identifier. The rules for this part are specified in Algorithm 3.1. It uses the Pure Linearization algorithm presented in [ORS07] to form a sorted list and transforms this sorted list into a directed cycle by establishing a directed edge from the node with the highest identifier to the node with the lowest identifier. The approach for constructing this cycle-edge resembles the approach presented in [KKS14].

The second part of the algorithm serves as a tool for the construction of the metric graph. The rules of this part are given in Algorithm 3.2. Intuitively, the second part of the algorithm does the following. Suppose the construction of the directed cycle is complete. Every node u maintains a pointer (i.e., a variable containing an identifier) u.test to some node in V. The specified rules cause the test-pointers of all nodes to traverse the cycle in a common direction.

Note that in Algorithm 3.2 some rules and some lines within the commands of certain rules are marked with a star. These rules are only executed in the synchronous case; in the asynchronous case the respective rules and lines are simply ignored. Furthermore, the second part of the algorithm contains two variables that are only used in the synchronous case, namely u.last and u.round. These additional rules and variables are used to achieve a linear running time in the synchronous case and to bound the communication work and memory requirement at each node after stabilization. Finally, consider the first line in the commands of the ReceiveResponse rule. Here, the statement $x \in [a, b]$ means that node x lies between the nodes a and b in the directed cycle. Formally, we have $x \in [a, b]$ if and only if $(a < x < b)$ or $(b < a < x)$ or $(x < b < a)$.

The third and final part of the algorithm is responsible for the actual construction of the metric graph. Its rules are given in Algorithm 3.3. In this part, a node u uses the pointer u.test to check whether the edge $(u, u.\text{test})$ should be marked. Additionally, every node checks for each of its outgoing edges in the current state of the metric graph whether it can be removed from the graph.

It is important to note that since the algorithm is self-stabilizing, all three parts have to be executed at the same time. Specifically, in the overall algorithm an activated node checks the guards of the rules sequentially for all three parts

Algorithm 3.1 Directed cycle construction

The rules are applied by a node u.

Establish an edge that closes the sorted list to a directed cycle:

```
InitCycle:      $u.N_\ell = \emptyset$       $\longrightarrow$  if $u.\text{cycle} \neq \text{null} \wedge u.\text{cycle} \neq u$
                                          $u.N \leftarrow u.N \cup \{u.\text{cycle}\}$
                                      $u.\text{cycle} \leftarrow u$

ForwardCycle:   $u.\text{cycle} \neq \text{null} \wedge$  $\longrightarrow$  for $v \in u.N_r$
                $u.N_r \neq \emptyset$                    $v.\text{sendCycle}(u.\text{cycle})$
```

Organize the nodes into a sorted list, see [ORS07]:

```
LeftLinearization:   $u.N_\ell \neq \emptyset$  $\longrightarrow$  $\text{pred} \leftarrow \max(u.N_\ell)$
                                       assume $u.N_\ell = \{w_1, \ldots, w_k\}$,
                                       where $w_i < w_j$ if $i < j$
                                       for $1 \leq i \leq k-1$
                                           $w_i.N \leftarrow w_i.N \cup \{w_{i+1}\}$
                                           $w_{i+1}.N \leftarrow w_{i+1}.N \cup \{w_i\}$
                                       $u.N \leftarrow u.N_r \cup \{\text{pred}\}$
                                       $\text{pred}.N \leftarrow \text{pred}.N \cup \{u\}$

RightLinearization:  $u.N_r \neq \emptyset$  $\longrightarrow$  $\text{succ} \leftarrow \min(u.N_r)$
                                       assume $u.N_r = \{w_1, \ldots, w_k\}$,
                                       where $w_i < w_j$ if $i < j$
                                       for $1 \leq i \leq k-1$
                                           $w_i.N \leftarrow w_i.N \cup \{w_{i+1}\}$
                                           $w_{i+1}.N \leftarrow w_{i+1}.N \cup \{w_i\}$
                                       $u.N \leftarrow u.N_\ell \cup \{\text{succ}\}$
                                       $\text{succ}.N \leftarrow \text{succ}.N \cup \{u\}$
```

The following rule is applied for each new cycle-value v that u receives at the end of a round:

```
ReceiveCycle:  true  $\longrightarrow$  if $u.\text{cycle} \neq \text{null} \wedge u.\text{cycle} \neq v$
                           $u.N \leftarrow u.N \cup \{u.\text{cycle}\}$
                       $u.\text{cycle} \leftarrow v$
```

Algorithm 3.2 Cycle traversal

The rules are applied by a node u. Rules and lines marked with a star are only executed in the synchronous case.

Get to an admissible state:

```
Reset: true $\longrightarrow$ $u$.Req.removeDuplicates()
                   if $u$.test = null
                       $u$.test $\leftarrow u$
                   * if $u$.last $\neq$ null
                       * $u.N \leftarrow u.N \cup \{u.\text{last}\}$
                       * $u$.last $\leftarrow$ null
```

Update the round counter:

```
* UpdateCounter: true $\longrightarrow$ if $u$.round = 2
                                 $u.N \leftarrow u.N \cup u.M \cup \{u.\text{test}\}$
                             $u$.round $\leftarrow \min\{\, u.\text{round} + 1,\ 2\,\}$
```

Move the test-pointers:

```
Request:  true                                   $\longrightarrow$ $u$.test.Req.enqueue($u$)

Respond:  $u.\text{Req} \neq \emptyset \wedge$               $\longrightarrow$ $v \leftarrow u$.Req.dequeue()
          $(u.N_r \neq \emptyset \vee u.\text{cycle} \neq \text{null})$        * $u$.last $\leftarrow v$
                                                     if $u.N_r = \emptyset$
                                                         succ $\leftarrow u$.cycle
                                                     else
                                                         succ $\leftarrow \min(u.N_r)$
                                                     $v$.sendResponse(succ)
```

The following rule is applied when u receives a response succ from a node v:

```
ReceiveResponse: true $\longrightarrow$ if $\exists w \in u.M \cup \{u\} : w \in [u.\text{test}, \text{succ}]$
                                 $u.N \leftarrow u.N \cup u.M \cup \{u.\text{test}\}$
                             if $v = u$.test
                                 $u$.test $\leftarrow$ succ
                                 * $u$.round $\leftarrow 0$
                             else
                                 $u.N \leftarrow u.N \cup \{v\}$
```

The following rule is applied after all messages have been received:

```
* ProcessLast: $u$.last $\neq$ null $\longrightarrow$ $u$.Req.remove($u$.last)
                                      $u$.last $\leftarrow$ null
```

Algorithm 3.3 Metric graph construction

The rules are applied by a node u.

The following rule removes superfluous edges from the metric graph and is executed repeatedly until its guard becomes false:

$$\begin{array}{lll} \textbf{Delegate:} & \exists v, w \in u.M : & \longrightarrow \; u.M \leftarrow u.M \setminus \{w\} \\ & v \neq w \wedge d(u,v,w) = d(u,w) & \; v.M \leftarrow v.M \cup \{w\} \\ & & \; u.N \leftarrow u.N \cup \{w\} \end{array}$$

The following rule is applied after every execution of the ReceiveResponse rule in the second part of the algorithm:

$$\begin{array}{lll} \textbf{Add:} & u.\text{test} \notin u.M \cup \{u\} \wedge & \longrightarrow \; u.M \leftarrow u.M \cup \{u.\text{test}\} \\ & \forall v \in u.M : d(u,v,u.\text{test}) > d(u,u.\text{test}) & \end{array}$$

in the specified order. A rule for receiving messages is executed once for each message of the corresponding type at the end of every round. The remaining rules are executed only once per activation with the exception of the rules in the third part, which are executed as specified in Algorithm 3.3.

The algorithm associates the following variables with each node u: The variable $u.\text{id} \in [0,1)$ stores the identifier of u. The set of identifiers $u.N$ represents the neighbors of u in the sorted list, see [ORS07]. We use $u.N_\ell$ to refer to the set of *left neighbors* of u, i.e., the subset of identifiers in $u.N$ that are strictly smaller than u.id. Similarly, we use $u.N_r$ to refer to the set of *right neighbors* of u, i.e., the subset of identifiers in $u.N$ that are strictly larger than u.id. The variable u.cycle is used to close the sorted list to a cycle by connecting the node with the highest identifier to the node with the lowest identifier. Once the construction of the cycle is complete, u.cycle contains the identifier of the node with the lowest identifier for every node u. The variable u.test specifies an outgoing edge that has to be tested in the metric graph construction. The queue u.Req contains all nodes that requested to be informed about the successor of u in the directed cycle. We assume that a value is stored at most once in this queue (i.e., if an already-present value is enqueued, the queue is not modified). The variable u.last stores the last node to which u sent a response. This variable is only used in the synchronous case. We elaborate more on the role of this variable in the last paragraph of this section. Similarly, the variable $u.\text{round} \in \{0,1,2\}$ is only used in the synchronous case. This variable is used to keep track of the number of rounds that have passed since the value of u.test changed. Finally, $u.M$ is a set of identifiers that defines the marked edges originating at u. Formally, the edge (u,v) is marked if and only if $v \in u.M$. Thereby, the sets $u.M$ encode the current state of the metric graph under construction in a distributed manner.

We define the set of undirected marked edges as

$$D = \{ \{u, v\} \mid u \in v.M \wedge v \in u.M \}.$$

Note that throughout the algorithm some messages are sent and received *explicitly*, e.g., by the rules ForwardCycle and ReceiveCycle. However, some messages are sent and received *implicitly*: For example, in the last line of the LeftLinearization rule the node u adds its own identifier to the set $u.\text{pred}.N$. This assignment actually means that u sends a message containing its identifier to $u.\text{pred}$ so that when this message is received at the end of the round, $u.\text{pred}$ adds u to $u.\text{pred}.N$. We assume that implicit messages are received before explicit messages.

Finally, the variable $u.\text{last}$ has a special role. The variable is used to correctly handle the case in which a node v sends a request to a node u in the same round as u responds to a previous request by v: If u responds to v in some round i, it stores the identifier of v in the variable $u.\text{last}$. After receiving the requests at the end of round i, u removes $u.\text{last}$ from $u.\text{Req}$ in the ProcessLast rule and thereby ignores the superfluous request sent by v. The variable $u.\text{last}$ is then set to null so that only one request is ignored. This mechanism is required to achieve a linear running time in a synchronous execution. The variable $u.\text{last}$ and the mechanism described above are not used in asynchronous executions.

3.5 Analysis

We now turn to the analysis of the algorithm. This section is structured as follows. We first show that the algorithm constructs the graph corresponding to the given metric in a self-stabilizing manner in both the synchronous and the asynchronous case. For this we show in Section 3.5.1 that the first part of the algorithm organizes the nodes into a directed cycle. In Section 3.5.2 we show that once the cycle has been constructed, the second part of the algorithm causes the test-pointers to traverse the cycle. On the basis of this result we then show in Section 3.5.3 that the third part of the algorithm constructs the metric graph. In the remaining two sections, we provide a more detailed analysis of the algorithm for the synchronous case. Specifically, we analyze the running time of the algorithm in Section 3.5.4 and present results concerning the behavior of the algorithm after stabilization in Section 3.5.5.

3.5.1 Directed Cycle Construction

Define $G_N = (V, E_N)$ where $E_N = \{ (u, v) \mid u \in V \text{ and } v \in u.N \}$. Consider a weakly-connected component C of G_N. We assume that $C = \{u_1, u_2, \dots, u_k\}$ such that $u_i < u_j$ if $i < j$. We say C *contains a sorted list* if $u_i \in u_{i+1}.N$ and $u_{i+1} \in u_i.N$ for all i such that $1 \leq i \leq k-1$. We say C *contains a directed cycle* if it contains a sorted list and $u_i.\text{cycle} = u_1$ for all i such that $1 \leq i \leq k$.

The goal in this section is to show that eventually all nodes form a single weakly-connected component in G_N that contains a directed cycle.

Lemma 3.2. *The nodes in a weakly-connected component of G_N remain weakly connected.*

Proof. Note that the only rules that can remove elements from $u.N$ are the LeftLinearization rule and the RightLinearization rule. A node u that executes these rules organizes its neighborhood in G_N into a bidirected sorted list. So if $(u, v) \in E_N$ at the beginning of round i then there is a directed path from u to v in G_N at the beginning of round $i + 1$. □

Lemma 3.3. *Let C be a weakly-connected component of G_N. Suppose that C never merges with another weakly-connected component. Then C eventually contains a directed cycle. Once this condition is true, it remains true indefinitely.*

Proof. If we consider only the LeftLinearization rule and the RightLinearization rule, it follows from the analysis of the Pure Linearization algorithm presented in [ORS07] that C eventually contains a sorted list and that this sorted list is maintained. The remaining rules of the algorithm might add new edges to G_N. Since C never merges with another component, we must have that if some rule adds an identifier v to $u.N$ for some $u \in C$, then $v \in C$. Adding such an identifier to $u.N$ does not hinder the construction of the sorted list.

It remains to show that C eventually contains a directed cycle. Suppose that C already contains a sorted list. We show by complete induction that eventually u_i.cycle $= u_1$ for $1 \leq i \leq k$ and from that point on, the values of the variables u_i.cycle do not change anymore. The node u_1 has the lowest identifier among the nodes in C. Therefore, we have $u_1.N_\ell = \emptyset$ at all times. This implies that u_1 eventually executes the InitCycle rule, upon which it sets u_1.cycle to u_1. Furthermore, since u_1 is the node with the lowest identifier in C, it can never receive a new cycle-value through the ReceiveCycle rule. Thereby, the value of u_1.cycle remains unchanged from that point on.

Now suppose that u_j.cycle $= u_1$ for all j such that $1 \leq j < i$ and the values of the variables u_j.cycle do not change anymore. We show that eventually u_i.cycle $= u_1$ and from that point on, the value of u_i.cycle does not change anymore. Since C contains a sorted list, we have $u_i \in u_{i-1}.N_r$. Eventually, u_{i-1} is activated and executes the ForwardCycle rule. Thereby, u_{i-1} sends u_1 to u_i as a new cycle-value. Upon receiving the message, u_i sets u_i.cycle to u_1. Since u_i can only receive new cycle-values from nodes u_j with $1 \leq j < i$ and for all of these nodes it holds u_i.cycle $= u_1$, the value of u_i.cycle does not change from that point on. □

Lemma 3.4. *Eventually, all nodes in V form a single weakly-connected component in G_N.*

Proof. We first consider the asynchronous case. Suppose for the sake of contradiction that the algorithm eventually forms $k \geq 2$ weakly-connected components in G_N such that none of these components ever merge. Recall that initially the graph G was weakly connected according to our model. Therefore, there must be a component C_1 and a node $u \in C_1$ such that u initially stored the identifier of a node v in a component $C_2 \neq C_1$. We distinguish five cases depending on which variable in the local memory of u initially contained the identifier of v and show that each case leads to a contradiction.

1. $v \in u.N$: In this case Lemma 3.2 implies $v \in C_1$, which is a contradiction.

2. $v = u.\text{cycle}$: By Lemma 3.3, C_1 eventually contains a directed cycle. Once this is the case, we have $u.\text{cycle} \neq v$ by definition. Hence, the initial value of $u.\text{cycle}$ must have been replaced at some point. The only rules that can set the value of $u.\text{cycle}$ are InitCycle and ReceiveCycle. Both rules add the replaced value to $u.N$. This implies that $v \in C_1$, which is a contradiction.

3. $v = u.\text{test}$: Note that the only rules that can change the value of $u.\text{test}$ are Reset and ReceiveResponse. The Reset rule changes the value of $u.\text{test}$ only if it is null, which is never the case since the ReceiveResponse rule cannot set $u.\text{test}$ to null. Thereby, only the ReceiveResponse rule changes the value of $u.\text{test}$.

   We first show by induction that $u.\text{test} \in C_2$ for all rounds. The statement holds initially by definition. So suppose the value of $u.\text{test}$ changes from $w \in C_2$ to w' through an application of the ReceiveResponse rule. For the value of $u.\text{test}$ to change, w must have sent w' to u as a response in an execution of the Respond rule. Therefore, we must have $w' \in w.N$ or $w' = w.\text{cycle}$ at the time w sends its response. If $w' \in w.N$ then $w' \in C_2$ by Lemma 3.2. If $w' = w.\text{cycle}$ and $w.\text{cycle} \notin C_2$ then we can apply the arguments from Case 2 above to show that eventually C_2 merges with another component, which is a contradiction. So overall we must have $w' \in C_2$, which concludes the induction.

   By Lemma 3.3, C_2 eventually contains a directed cycle and this cycle is maintained indefinitely. For a node w let $w.\text{succ}$ be the successor of w in the cycle, i.e., $w.\text{succ}$ is the node with the lowest identifier among the nodes in C_2 that have a higher identifier than w or, if there is no such node, $w.\text{succ}$ is the node with the lowest identifier in C_2. When the construction of the cycle is complete, we still have $u.\text{test} \in C_2$ by the inductive argument above. From that point on, we have the following property: If u.test $= w$ then eventually u.test $= w$.succ. To see this, first note that u only sets a new value for u.test if it receives a response from w in the ReceiveResponse rule. It remains to show that w will eventually send such a response and that this response is in fact w.succ. Since the execution is fair, u is eventually activated and enqueues its identifier to w.Req due to the Request rule. Once w has been activated a sufficient number of times, it dequeues u from w.Req and sends a response to u during an application of the Respond rule. Is it not hard to see that since C_2 contains a sorted cycle, w responds with w.succ.

Finally, since $u \notin C_2$ there is a node $z \in C_2$ such that $u \in [z, z.\text{succ}]$. By the arguments above and since C_2 is finite, u eventually receives the response z.succ from z while u.test $= z$ in an activation of the ReceiveResponse rule. Therefore, u detects that $u \in [z, z.\text{succ}]$ and adds z to $u.N$. Thereby, C_1 and C_2 merge, which is a contradiction.

4. $v \in u$.Req: In this case, u eventually sends a response w to v due to the Respond rule. Consider the execution of the ReceiveResponse rule by v in which this response is received. If at this point v.test $\neq u$ then v adds u to $v.N$, which is a contradiction. So we must have v.test $= u$. Therefore, v sets v.test to w. By the arguments given at the beginning of Case 3, we must have $w \in C_1$. Thereby, we can apply the entire argument from Case 3 with v taking on the role of u and w taking on the role of v to get a contradiction.

5. $v \in u.M$: If v is removed from $u.M$ at some point due to an activation of the Delegate rule then v is added to $u.N$ and thus C_1 and C_2 merge, which is a contradiction. Since no other rule can remove v from $u.M$, v remains in $u.M$ indefinitely. Eventually, every weakly-connected component of G_N contains a directed cycle. Consider the value of u.test once this point has been reached. After one activation of u we have u.test $\neq$ null according to the Reset rule. Since no rule can set u.test to null, this condition remains satisfied. Therefore, we have u.test $\in C$ for some weakly-connected component C of G_N. Note that by the arguments given in Case 3, u.test traverses the cycle contained in C, i.e., if u.test $= w$ then eventually u.test $= w$.succ. If $C \neq C_1$ then the arguments given in Case 3 imply a contradiction. So consider the case that $C = C_1$. Since $v \notin C_1$, there is a node $z \in C_1$ such that $v \in [z, z.\text{succ}]$. Because C_1 is finite, u eventually receives the response z.succ from z while u.test $= z$ in an activation of the ReceiveResponse rule. Therefore, u detects that $v \in [z, z.\text{succ}]$ and adds all nodes from $u.M$ to $u.N$. Since $v \in u.M$ this means that C_1 and C_2 merge, which is a contradiction.

Finally, we turn to the synchronous case. All statements above still hold.

However, one has to consider the possibility that the ProcessLast rule removes an identifier from a queue. It is not hard to check that this does not invalidate the given arguments. Additionally, we have to consider the case that $v = u.\text{last}$ initially. In this case, the Reset rule adds v to $u.N$, which is a contradiction. □

Combining Lemmas 3.3 and 3.4 gives us the following corollary.

Corollary 3.5. *Eventually, all nodes form a single weakly-connected component in G_N that contains a directed cycle. Once this condition is true, it remains true indefinitely.*

3.5.2 Movement of the Test-Pointers

In this section, we investigate the sequence of values the variable $u.\text{test}$ takes on after the construction of the directed cycle has been completed. Intuitively, a test-pointer traverses the directed cycle one node after the other. As a consequence, the variable $u.\text{test}$ eventually takes on the value v for every $v \in V$. Note that this statement holds at any point in time. Thereby, $u.\text{test}$ takes on any value $v \in V$ infinitely often. Formally, we have the following lemma.

Lemma 3.6. *For all nodes $u, v \in V$ it holds that eventually u attempts to execute the Add rule while $u.test = v$.*

Proof. By Corollary 3.5, eventually all nodes form a single weakly-connected component in G_N that contains a directed cycle. We consider the execution of the algorithm from that point on. After the first activation of u, we have $u.\text{test} \neq \text{null}$ according to the Reset rule. Since no rule can set $u.\text{test}$ to null, this condition remains true. The node u eventually adds its identifier to $u.\text{test.Req}$ in an execution of the Request rule. After a sufficient number of activations, the node $u.\text{test}$ eventually dequeues the identifier of u from $u.\text{test.Req}$ and sends its successor in the cycle as a response to u in an execution of the Respond rule. Upon receiving the response in the ReceiveResponse rule, u sets $u.\text{test}$ to the successor of $u.\text{test}$. Then u immediately attempts to execute Add rule. Since the cycle is finite, the lemma holds by induction.

Note that in the synchronous case, the identifier of u can be removed from $u.\text{test.Req}$ by the ProcessLast rule. However, this can happen only if $u.\text{test}$ already sent a response to u so that this case does not invalidate the lemma. □

3.5.3 Metric Graph Construction

We now show that the overall algorithm constructs the metric graph corresponding to the given metric in a self-stabilizing manner. We begin with some basic properties concerning the Add rule and the Delegate rule from the third part of the algorithm. We refer to a directed edge (u, v) as a *final edge* if the corresponding undirected edge $\{u, v\}$ lies in $\overline{D}$, i.e., if it is part of the graph corresponding to the given metric.

Lemma 3.7. *The Delegate rule never removes a final edge.*

Proof. Suppose that the Delegate rule removes a final edge (u, w) from the metric graph, i.e., it removes w from $u.M$. For this to happen, the guard of the Delegate rule must be satisfied. Therefore, there must be a node $v \in u.M$ such that $d(u, v, w) = d(u, w)$. According to the Metric Graph condition given in Section 3.3, there is a node t such that $\{u, t\} \in \overline{D}$ and $d(u, v) = d(u, t, v)$ where we might have $t = v$. Therefore, we have

$$d(u, w) = d(u, v, w) = d(u, t, v, w) \geq d(u, t, w),$$

which implies $d(u, w) = d(u, t, w)$. It follows from Lemma 3.1 that (u, w) is not a final edge, which is a contradiction. □

Lemma 3.8. *The length of a shortest directed path from a node u to a node v in the metric graph under construction can only decrease.*

Proof. Recall that the current state of the metric graph is induced by the sets $u.M$. The only rules that can change $u.M$ are the Add rule and the Delegate rule. The addition of an edge by the Add rule can only create shorter paths. When the Delegate rule removes an edge (u, v), it creates an alternative path of equal length from u to v. □

Lemma 3.9. *The Add rule never adds an edge that was previously removed by the Delegate rule.*

Proof. Suppose that a node u has delegated the edge (u, w) to a node v. This implies that $d(u, v, w) = d(u, w)$. Since $v \in u.M$, we know by Lemma 3.8 that there will always be a path between u and v of length at most $d(u, v)$. This means there will always be a node $t \in u.M$ such that $d(u, t, v) = d(u, v)$ where we might have $t = v$. Therefore, we have

$$d(u, w) = d(u, v, w) = d(u, t, v, w) \geq d(u, t, w),$$

which implies $d(u, t, w) = d(u, w)$. Accordingly, the predicate

$$\exists t \in u.M : d(u, t, w) = d(u, w)$$

is always true. Therefore, the guard of the Add rule is never satisfied for the edge (u, w). □

We are now ready to tackle the analysis of the overall algorithm.

Lemma 3.10. *Eventually, the set $u.M$ does not change anymore for every node $u \in V$.*

Proof. The only rules that can change the set $u.M$ of a node u are the Add rule and the Delegate rule. According to Lemma 3.9, the Add rule can add each edge at most once and, therefore, it eventually stops adding edges to the graph. It remains to show that the Delegate rule eventually stops delegating edges. By Lemma 3.7, a final edge is never delegated. So consider a non-final edge. The delegation of such an edge removes the edge from the graph and adds an edge to the graph that has a strictly smaller weight according to the metric. Since the metric is finite, this implies that the delegation of edges eventually ceases. □

Theorem 3.11. *The algorithm is self-stabilizing with respect to the set L defined in Section 3.3.*

Proof. Recall that

$$D = \{\, \{u, v\} \mid u \in v.M \wedge v \in u.M \,\}$$

and we say D is valid if $v \in u.M$ implies $u \in v.M$. On the basis of the statement concerning $\overline{D}$ given in Lemma 3.1 it is not hard to see that if $D = \overline{D}$ and D is valid then the guards of the Delegate rule and the Add rule cannot be satisfied and, thereby, the metric graph does not change. Therefore, the algorithm satisfies the closure-condition.

It remains to show that the algorithm satisfies the convergence-condition. According to Lemma 3.6 and the arguments given in Lemma 3.10, we have that, eventually, the guards of the Delegate rule and the Add rule are false for every node u and for any $t = u.\text{test}$. Therefore, for all nodes $u, t \in V$ we have

$$\neg\exists v, w \in u.M : v \neq w \wedge d(u, v, w) = d(u, w) \tag{3.1}$$

and

$$\neg\,(\, t \notin u.M \cup \{u\} \wedge \forall v \in u.M : d(u, v, t) > d(u, t)\,)\,. \tag{3.2}$$

Consider a node $v \notin u.M$ such that $v \neq u$. Equation 3.2 implies

$$v \in u.M \vee \exists w \in u.M : d(u, w, v) \leq d(u, v).$$

By definition, we have $v \notin u.M$. Therefore, we can use the triangle inequality to deduce that

$$\exists w \in u.M : d(u, w, v) = d(u, v).$$

Hence, we have that for all nodes $u, v \in V$ it holds

$$v \notin u.M \implies \exists w \in u.M \setminus \{u, v\} : d(u, w, v) = d(u, v).$$

Now consider a node $v \in u.M$ such that $v \neq u$. Equation 3.1 implies

$$\neg\exists w \in u.M : v \neq w \wedge d(u, w, v) = d(u, v).$$

Hence, we have that for all nodes $u, v \in V$ it holds

$$v \in u.M \implies \neg\exists w \in u.M \setminus \{u, v\} : d(u, w, v) = d(u, v).$$

In summary, we have that for all nodes $u, v \in V$ it holds

$$v \notin u.M \iff \exists w \in u.M \setminus \{u, v\} : d(u, w, v) = d(u, v).$$

Note that this equation corresponds to the condition given in Lemma 3.1. Thereby, the constructed metric graph contains the directed edge (u, v) if and only if $\{u, v\} \in \overline{D}$. This implies that $D = \overline{D}$ and that D is valid. □

3.5.4 Running Time

We now retrace our steps so far and provide a more thorough analysis of the synchronous case that incorporates the running time of the algorithm. Recall that G_N is the directed graph induced by the identifiers stored in the sets $u.N$. Our first goal is to show that after a linear number of rounds, all nodes in V form a single weakly-connected component in G_N.

Lemma 3.12. *Let $V = \{u_1, \ldots, u_n\}$ where $u_i < u_j$ if $i < j$. For a node u_i it holds that at the beginning of every round $j > i$ we have u_i.cycle $\neq$ null and the nodes u_i and u_i.cycle lie in the same weakly-connected component in G_N.*

Proof. We show the statement by complete induction. Since u_1 has the lowest identifier, we have $u_1.N_\ell = \emptyset$ at all times. Therefore, u_1 executes the InitCycle rule in every round and sets u_1.cycle to its own identifier. Because u_1 has the lowest identifier, it can never receive a cycle-value from another node. Therefore, we have u_1.cycle $= u_1$ at the beginning of each round $j > 1$.

Now suppose that the statement holds for the nodes $u_1, u_2, \ldots, u_{i-1}$. We show that it also holds for u_i. First suppose that u_i does not receive a new cycle-value in round i. Then the induction hypothesis implies that $u_i \notin u_k.N$ at the beginning of round i for all k such that $1 \leq k < i$. Therefore, we must have $u_i.N_\ell = \emptyset$ at the beginning of round $i - 1$. This implies that u_i must have executed the InitCycle rule in round $i - 1$. Furthermore, we must have $u_i \notin u_k.N$ at the beginning of round $i-1$ for all k such that $1 \leq k < i$ according to Lemma 3.2. This implies that u_i does not receive a new cycle-value in round $i - 1$. Therefore, we have u_i.cycle $= u_i$ at the beginning of round i. Since u_i does not receive a new cycle-value in round i, we still have u_i.cycle $= u_i$ at the beginning of round $i + 1$. So in this case, u_i and u_i.cycle indeed lie in the same weakly-connected component at the beginning of round $i + 1$.

If u_i does receive a new cycle-value v in round i then this value must have been sent by a node u_k with $k < i$ at the beginning of round i. By the induction hypothesis, v and u_k lie in the same weakly-connected component in G_N. Since u_k sent the cycle-value to u_i, we must have $u_i \in u_k.N$ at the beginning of round i. Therefore, u_i and u_k lie in the same weakly-connected component in G_N at that time. By transitivity, also u_i and v lie in the same weakly-connected component. According to Lemma 3.2, the nodes remain in the same weakly-connected component. Therefore, again u_i and u_i.cycle lie in the same weakly-connected component at the beginning of round $i + 1$.

If u receives a new cycle-value v in some round $j > i$ then, by arguments that are analogous to those given above, u and v must lie in the same weakly-connected component. □

Lemma 3.12 implies the following corollary.

Corollary 3.13. *Consider a round $i > n$ and a node u. At the beginning of round i we have $u.cycle \neq null$ and the nodes u and $u.cycle$ lie in the same weakly-connected component in G_N.*

The following lemma is analogous to Lemma 3.4.

Lemma 3.14. *After $O(n)$ rounds, all nodes in V form a single weakly-connected component in G_N.*

Proof. Suppose for the sake of contradiction that at the beginning of round $4n + 1$ the graph G_N contains $k \geq 2$ weakly-connected components $C_1, \ldots, C_k$. We consider the identifiers stored by the nodes at the beginning of round $n + 1$. Note that the nodes in V form a single weakly-connected component in G at the beginning of each round because otherwise we would have a contradiction to Lemma 3.4. Therefore, there must be a component C_i and a node $u \in C_i$ such that at the beginning of round $n + 1$, u stores the identifier of a node v in a component C_j such that $i \neq j$. We distinguish six cases depending on which variable in the local memory of u contains the identifier of v.

1. $v \in u.N$: In this case Lemma 3.2 implies $v \in C_i$, which is a contradiction.

2. $v = u$.cycle: Corollary 3.13 implies that u and u.cycle $= v$ lie in the same weakly-connected component in G_N, which is a contradiction.

3. $v = u$.test: First, note that for the value of u.test to change, v must send a response to u in an application of the Respond rule. The value of u.test is then set to the identifier sent along with the response. This identifier either belongs to a node from $v.N$ or to the node v.cycle. In the former case the new value of u.test is a node from C_j by definition and in the latter case the new value is from C_j according to Corollary 3.13. This argument holds inductively so that u.test $\in C_j$ at the beginning of each round k where $n + 1 \leq k \leq 3n$. Within these rounds there cannot be two consecutive rounds such that u does not receive a response

from u.test, because if this were the case, u would add $u.\text{test} \in C_j$ to $u.N$ in the following round due to the UpdateCounter rule, which is a contradiction. So during this interval of time, u receives a response from u.test at least every second round. Since the interval consists of $2n$ rounds and $|C_j| < n$ by definition, u.test must assume the same value twice during this interval. Since $u \notin C_j$, this implies that at some point during the interval the condition of the if-statement in the first line of the ReceiveResponse rule must be satisfied. As a consequence, u adds u.test to $u.N$, which is a contradiction.

4. $v \in u.\text{Req}$: Note that according to Corollary 3.13, $u.\text{cycle} \neq \text{null}$. Therefore, the guard of the Respond rule is satisfied for u as long as $u.\text{Req} \neq \emptyset$. Since u.Req can contain at most n elements, u sends a response to v within n rounds. When this response is received we must have $v.\text{test} = u$ because otherwise v would add u to $v.N$, which is a contradiction. From this point on, we can apply the arguments from Case 3 above with the roles of u and v interchanged. Note that in applying these arguments, the time interval under consideration has to be shifted back by at most n rounds. Since we consider the weakly-connected components at the beginning of round $4n + 1$, this does not cause any problems.

5. $v \in u.M$: The only rule that can remove v from $u.M$ is the Delegate rule. However, when this rule removes v from $u.M$, it adds v to $u.N$. Therefore, v is not removed from $u.M$ up to round $4n + 1$. According to the arguments given in Case 3, we have that u.test stays within the same weakly-connected component. If $u.\text{test} \in C_j$ then, by the arguments given in Case 3, within $2n$ rounds the condition of the if-statement in the first line of the ReceiveResponse rule is satisfied because $u \notin C_j$. Therefore, all nodes in $u.M$ are added to $u.N$, which is a contradiction. If $u.\text{test} \in C_k \neq C_j$ then the value of u.test must change at least every second round because otherwise $v \in u.M$ would be added to $u.N$. Therefore, within $2n$ rounds the condition of the if-statement in the first line of the ReceiveResponse rule is satisfied because $v \in u.M$ but $v \notin C_k$. Again, this implies that all nodes in $u.M$ are added to $u.N$, which is a contradiction.

6. $v = u.\text{last}$: At the beginning of each round $r > 1$ we have $u.\text{last} = \text{null}$ due to the ProcessLast rule, which is a contradiction.

□

We now show that the first part of the algorithm organizes all nodes into a directed cycle in linear time.

Lemma 3.15. *After $O(n)$ rounds, G_N contains a directed cycle.*

Proof. Lemma 3.14 together with the analysis of the Pure Linearization algorithm given in [ORS07] imply that after $O(n)$ rounds the network contains a sorted list. Using the arguments given in the proof of Lemma 3.3 while keeping in mind that in a synchronous execution each node is activated in every round, it is not hard to see that, once the sorted list has been established, it takes $O(n)$ rounds until the network contains a directed cycle. □

Next we take a closer look at the second part of the algorithm. Our goal is to characterize the way the test-pointers traverse the cycle in the synchronous case. For a node $u \in V$, we define

$$T_u(i) = \{\, v \in V \mid v.\text{test} = u \text{ at the beginning of round } i \,\}.$$

We first provide a series of lemmas that together show that eventually $|T_u| \leq 2$ for every node u. From this we can then deduce that each test-pointer traverses the entire cycle in linear time.

Lemma 3.16. *At the beginning of round $i > 2n$ we have $u.Req \subseteq T_u(i)$ for every node u.*

Proof. Consider a node $v \in u.$Req at the beginning of round $i > 2n$. We show that $v.$test $= u$ at the beginning of round i. For this, we first show that there is a round $k \leq 2n$ such that v was added to $u.$Req in round k. If there is no such round k then we must have $v \in u.$Req at the beginning of round 1. Recall that, by definition, each node can occur at most once in $u.$Req. Therefore, $u.$Req can contain at most n elements at the beginning of round 1. Furthermore, we have $u.$cycle $\neq$ null for each round $i' > n$, according to Corollary 3.13. Therefore, if $u.$Req $\neq \emptyset$ at the beginning of a round i' then u dequeues one element from $u.$Req due to the Respond rule. It follows that v must have been removed from $u.$Req in some round $j \leq 2n$. Hence, for v to be in $u.$Req at the beginning of round $i > 2n$, there must be a round k with $j \leq k < i$ such that v was added to $u.$Req in round k, which is a contradiction. So, indeed, there is a round $k \leq 2n$ such that v was added to $u.$Req in round k.

We choose k to be as large as possible, i.e., k is the last round before round i in which v is added to $u.$Req. Recall that next to the Respond rule also the ProcessLast rule can remove an element from $u.$Req. In fact, this rule can remove an element in the same round as it was added. By our choice of k, we know that the ProcessLast rule does not remove v from $u.$Req in round k and neither of the two rules removes v from $u.$Req in any round ℓ with $k < \ell < i$.

The only rule that can add a node to $u.$Req is the Request rule. For this rule to add v to $u.$Req in round k, we must have $v.$test $= u$ at the beginning of round k. The value of $v.$test changes only when v receives a response from

u that is sent in the Respond rule. Suppose that v receives such a response in round k. Then u must have set u.last to v in round k. Consequently, u would have removed v from u.Req in the ProcessLast rule at the end of round k, which is a contradiction. So suppose that v receives a response from u in some round ℓ such that $k < \ell < i$. Thereby, u must have dequeued v from u.Req in the Respond rule in round ℓ, which is again a contradiction. So overall we still have v.test $= u$ at the beginning of round i. □

Suppose that the network contains a directed cycle. We use u.succ to refer to the successor of a node u in the directed cycle and we use u.pred to refer to the predecessor of u. The following lemma shows that when we observe the variable v.test of a node v at the beginning of each round, its value never skips a node in the directed cycle.

Lemma 3.17. *There is a round $i_0 = O(n)$ such that for every round $i \geq i_0$ and every node v the following statement holds: If $v.test = u$ at the beginning of round i then at the beginning of round $i + 1$ we have either $v.test = u$ or $v.test = u.succ$.*

Proof. We define i_0 to be the smallest round such that $i_0 > 2n$ and the network contains a directed cycle at the beginning of round i_0. According to Lemma 3.15, we have $i_0 = O(n)$. Consider a node v at the beginning of round $i \geq i_0$ and let $u = v$.test. If v does not receive a response from u in round i then v.test $= u$ at the beginning of round $i + 1$ and the lemma holds. So assume that v does receive a response from u in round i. Lemma 3.16 implies that if v.test $\neq w$ for a node w then $v \notin w$.Req. It follows that u is the only node that stores the identifier of v in its queue u.Req at the beginning of round i. Therefore, v cannot receive a response from u.succ in round i, which implies v.test $= u$.succ at the beginning of round $i + 1$. □

We are now ready to show that after a linear number of rounds we have $|T_u| \leq 2$ for every node u.

Lemma 3.18. *There is a round $i_0 = O(n)$ such that for every round $i \geq i_0$ and every node u it holds $|T_u(i)| \leq 2$.*

Proof. We define i_0' to be the smallest round such that $i_0' > 2n$ and the network contains a directed cycle at the beginning of round i_0'. We start with two simple observations that hold for every node u. First, at most one new node joins T_u in every round $i \geq i_0'$. This observation holds because u.pred responds to at most one node per round. Second, in each round i such that $i \geq i_0' + 1$ and $|T_u(i)| \geq 2$, u responds to a request. This is observation holds because for u not to respond to a request, the queue u.Req must be empty at the beginning of round i. However, if at least two nodes point at u at the beginning of round

i, at least one of these nodes must have already pointed at u at the beginning of round $i-1$, according to our first observation. Therefore, this node is added to u.Req at the end of round $i-1$. Together, these two observations imply that if $|T_u(i)| \leq 2$ for some round $i \geq i'_0+1$ then $|T_u(j)| \leq 2$ for all $j \geq i$.

It remains to show that there is a round $i_0 = O(n)$ such that $i_0 \geq i'_0+1$ and $|T_u(i_0)| \leq 2$. Suppose that $|T_u(i'_0+1)| > 2$ for some node u. Lemma 3.17 implies that for every node $v \in T_u(i'_0+1)$ it holds that when v is removed from T_u it takes at least n rounds until the test-pointer of v traversed the entire cycle and comes back to u. Therefore, during the next $n-1$ rounds starting at round i'_0+1, at most $n-|T_u(i'_0+1)|$ nodes join T_u. This implies that among these $n-1$ rounds there are at at least

$$(n-1)-(n-|T_u(i'_0+1)|) = |T_u(i'_0+1)|-1$$

rounds in which no node joins T_u. In each of these rounds as long as $|T_u| \geq 2$ the value of $|T_u|$ decreases by 1, according to our second observation. Hence, we have $|T_u(i_0)| \leq 2$ for $i_0 = i'_0+n+1$. According to Lemma 3.15, we have $i'_0 = O(n)$ so that also $i_0 = O(n)$. □

Next, we show that the test-pointers completely traverse the directed cycle within $O(n)$ rounds.

Lemma 3.19. *There is a round $i_0 = O(n)$ such that for every round $i \geq i_0$ and every node v the following statement holds: If $v.test = u$ at the beginning of round i then $v.test = u.succ$ at the beginning of round $i+1$ or at the beginning of round $i+2$.*

Proof. We choose i_0 as specified in the proof of Lemma 3.18 and we consider a round $i \geq i_0$. Consider a node v and let $u = v$.test at the beginning of round i. According to Lemma 3.17, the variable v.test cannot skip the value u.succ. To proof the lemma it remains to show that u responds to v in round i or round $i+1$. First suppose that $v \in u$.Req at the beginning of round i. Lemma 3.16 and Lemma 3.18 together imply that $|u.\text{Req}| \leq 2$. Therefore, u responds to v in round i or round $i+1$.

Now suppose that $v \notin u$.Req at the beginning of round i. We know by Lemma 3.18 that $|T_u(i)| \in \{1,2\}$. If $|T_u(i)| = 1$ then u.Req is empty at the beginning of round i and v is the only node that is added to u.Req in round i. Therefore, u sends a response to v in round $i+1$. If $|T_u(i)| = 2$ then there is another node w pointing at u at the beginning of round i. Since $v \notin u$.Req, v must have set v.test to u at the end of round $i-1$. As we observed in the proof of Lemma 3.18, at most one node can join T_u in every round. This implies that w must have joined T_u in an earlier round. Therefore, w must be in u.Req at the beginning of round i. This implies that u dequeues w from u.Req in round i and sends a response to w. Both v and w send a request to u in round i so that both identifiers are added to u.Req. However, the identifier of w is removed from u.Req by the ProcessLast rule at the end of round i since u responded to w. Thereby, v is the only node in u.Req at the beginning of round $i+1$, which implies that u responds to v in that round. □

Lemma 3.19 implies the following corollary.

Corollary 3.20. *For all nodes $u, v \in V$ it holds that within $O(n)$ rounds u attempts to execute the Add rule while $u.test = v$.*

We now turn to the analysis of the third part of the algorithm, which is responsible for the construction of the metric graph.

Lemma 3.21. *After $O(n)$ rounds, the metric graph contains all final edges.*

Proof. Consider a final edge (u, v). The Minimality condition given in Section 3.3 implies that for all nodes $w \in V \setminus \{u, v\}$ it holds $d(u, w, v) > d(u, v)$. Furthermore, by Corollary 3.20 we have that within $O(n)$ rounds u attempts to execute the Add rule while u.test $= v$. If $v \notin u.M$ at this point then the guard of the Add rule is satisfied so that u adds v to $u.M$. According to Lemma 3.7, a final edge is never removed. □

The following lemma is analogous to Lemma 3.10.

Lemma 3.22. *After $O(n)$ rounds, the set $u.M$ does not change anymore for every node $u \in V$.*

Proof. According to the Lemma 3.21, the metric graph contains all final edges after $O(n)$ rounds. From that point on, the Delegate rule is the only rule that can modify the sets $u.M$. The Delegate rule transforms a non-final edge (u, v) into a path consisting exclusively of final edges. Each delegation adds one node to this path. To bound the number of rounds until no further delegation occurs, we bound the lengths of a longest constructed path where length refers to the number of nodes on the path.

Consider a longest path $(w_0 = u, w_1, \dots, w_k = v)$. Note that k must be finite according to Lemma 3.10. By the definition of the Delegate rule, we have

$$d(u, v) = d(w_0, w_k) = d(w_0, w_1, \dots, w_k).$$

Suppose there are indices $i < j$ such that $w_i = w_j$. By definition, a final edge cannot be a loop. Therefore, we have $i < j - 1$ and $w_i \neq w_{i+1}$. This implies

$$\begin{aligned} d(w_0, w_k) &= d(w_0, \dots, w_i) + d(w_i, w_{i+1}, \dots, w_j) + d(w_j, \dots, w_k) \\ &\geq d(w_0, w_k) + d(w_i, w_{i+1}, w_j), \end{aligned}$$

where $d(w_i, w_{i+1}, w_j) > 0$, which is a contradiction. Therefore, each node can occur at most once on the path, which implies that the length of the path is bounded by n. This implies that the maximum number of sequential delegations is bounded by n and, therefore, after $O(n)$ rounds the sets $u.M$ do not change anymore. □

Lemma 3.22 together with the arguments given in the proof of Theorem 3.11 imply the following corollary.

Corollary 3.23. *In the synchronous case, the algorithm constructs the graph corresponding to the given metric in a self-stabilizing manner in $O(n)$ rounds.*

3.5.5 After Stabilization

In this section, we analyze the behavior of the algorithm after stabilization in the synchronous case. Specifically, we show that the number of messages sent and received by a node in a round becomes constant and the memory overhead required at each node also becomes constant. These properties do not necessarily hold immediately after stabilization of the metric graph since even when the metric graph is stable, the directed cycle used for its construction might still change. Rather, the properties hold once both the metric graph and the directed cycle have stabilized. We show this to be the case after an overall linear number of rounds.

We say the network contains a *minimal* directed cycle if G_N contains a directed cycle and the set $u.N$ of a node u only contains the immediate predecessor and successor of u in the sorted list. We have the following lemma.

Lemma 3.24. *After $O(n)$ rounds of a synchronous execution, the network contains a minimal directed cycle and this condition remains true.*

Proof. According to Lemma 3.15, the network contains a directed cycle after $O(n)$ rounds. Consider a node u. We show that $u.N$ contains only the immediate predecessor and successor of u in the sorted list after an additional $O(n)$ rounds. Note that once the network contains a directed cycle, the value of u.cycle does not change anymore. Therefore, the rules InitCycle and ReceiveCycle do not add any more nodes to $u.N$. After the first round, the Reset rule does not add any more nodes to $u.N$. According to Lemma 3.19, after $O(n)$ rounds the value of the variable u.test changes at least every second round. Thereby, the UpdateCounter rule does not add any more nodes to $u.N$. Once the network contains a directed cycle, the second line of the ReceiveResponse rule does not add any more nodes to $u.N$. By Lemma 3.16, after $O(n)$ rounds the last line of the ReceiveResponse rule does not add any more nodes to $u.N$. Finally, after $O(n)$ rounds the set $u.M$ does not change anymore according to Lemma 3.22. Thereby, the Add rule and the Delegate rule do not add any more nodes to $u.N$.

Overall, after $O(n)$ rounds the only rules that modify the set $u.N$ are the LeftLinearization rule and the RightLinearization rule. It follows from the analysis of the Pure Linearization algorithm presented in [ORS07] that after an additional $O(n)$ rounds, $u.N$ only contains the immediate predecessor and successor of u in the sorted list, and this condition remains true. □

Concerning the communication work induced at each node, we have the following theorem.

Theorem 3.25. *After $O(n)$ rounds of a synchronous execution, each node sends and receives a constant number of messages per round.*

Proof. According to Lemma 3.24, the network contains a minimal directed cycle after $O(n)$ rounds. Once this condition holds, the first part of the algorithm causes only constant communication work per node. In the second part of the algorithm, a node sends at most one request and one response per round. Furthermore, after $O(n)$ rounds each node receives at most $|T_u| \leq 2$ requests according to Lemma 3.18 and at most one response according to Lemma 3.16. Finally, the third part of the algorithm does not incur any communication work after the stabilization of the metric graph. □

Let $\deg(u)$ be the degree of a node u in the metric graph. For the memory overhead at a node, we have the following theorem.

Theorem 3.26. *After $O(n)$ rounds of a synchronous execution, each node u stores at most $deg(u) + 7$ identifiers of other nodes in its memory.*

Proof. Consider a node u. Once the metric graph is stable, the set $u.M$ contains $\deg(u)$ identifiers. If the network contains a minimal directed cycle, the set $u.N$ contains at most two identifiers. The queue $u.\mathrm{Req}$ contains at most two identifiers by Lemma 3.16 and Lemma 3.18. Finally, u stores one identifier in each of the variables $u.\mathrm{test}$, $u.\mathrm{cycle}$, and $u.\mathrm{last}$. □

3.6 Outlook

One of the main ideas underlying the algorithm presented in this chapter is that each node maintains a test-pointer that eventually traverses a directed cycle covering all nodes. Thereby, each node eventually learns of every other node in the network. It might be possible to apply this idea to problems beyond the self-stabilizing construction of metric graphs: The presence of the test-pointers allows each node to check the validity of the topology using information outside of its local neighborhood. Thereby, a node might be able to detect a fault in a topology that is not locally checkable. The node could then initiate a process to recover the topology.

Finally, it could be of interest to modify the algorithm presented in this chapter to construct a graph that *approximates* the given metric instead of the graph that corresponds to the metric. The motivation behind this approach is that a graph approximating a given metric can have a much lower degree than the exact metric graph. While we provided a rough sketch of this approach in the publication underlying this work (see [GLS16]), a formal investigation of its limits and benefits is still open.

Chapter 4

Hybrid Network Monitoring

In this chapter we propose a new model for the study of distributed algorithms for *hybrid networks*. In a hybrid network a set of nodes is connected by an *external network* and an *internal network*. While the external network cannot be controlled by the network algorithm and might be exposed to continuous change, the internal network is an overlay network that is fully under the control of the network algorithm. By considering a combination of an overlay network and a dynamic external network, this chapter slightly broadens the focus of this part of the thesis in comparison to the previous two chapters.

Hybrid networks can be found both at a physical and a logical level. Consider, for instance, a set of wireless devices with access to the cell phone infrastructure that are dispersed over a limited area such as a city center. The devices can communicate using the cell phone infrastructure. However, this is typically associated with a certain cost to the users. The devices can also form a wireless ad-hoc network by establishing local WiFi connections with each other. While communication via such a network is free, it can have several disadvantages: The network might experience large message delays and there might be temporary network partitions.

For certain tasks it might be possible to design protocols that mostly rely on the wireless ad-hoc network for communication while sending small amounts of data via the cell phone infrastructure when necessary. Using the wireless ad-hoc network for most of the communication drastically reduces the cost for the users when compared to relying exclusively on the cell phone infrastructure. At the same time, allowing a protocol to sparingly use the cell phone infrastructure might result in solutions that are much faster and more reliable than using the wireless ad-hoc network alone. Thereby, combining the two modes of communication can give us the best of both worlds.

On a logical level, one can envision a peer-to-peer network formed by friendship links in a social network. Communicating just via these friendship links has the advantage that all interactions are trusted. However, due to the irregular structure of the social network it might be hard to perform certain tasks such as monitoring properties of the network or finding participants efficiently. Therefore, it might be useful to have a network of untrusted links on top of the social network in order to expedite such tasks.

A common theme in both of these examples is that there are two communication modes that significantly differ in the amount of control the algorithm has over the network topology, and in both examples control comes at a certain price such as financial cost, acceptance, reliability, or integrity. There is already a large body of literature on network algorithms for the case of static or dynamic networks whose topology is not under the control of the algorithm. On the other hand, there also exists an abundance of work in which the topology is fully under the control of the network algorithm, like in peer-to-peer systems. However, we are not aware of any rigorous results concerning the *combination* of such networks into a hybrid network.

We initiate the study of hybrid networks by considering the problem of *network monitoring*: In a monitoring problem, a specific node called the *monitor* has to continuously observe some property of the external network such as the number of edges or the weight of a minimum spanning tree. A monitoring algorithm can use the internal network to update the value of the property over time as the external network changes. We present scalable distributed algorithms for several monitoring problems, see Table 4.1 for an overview of our results. Since the delays given in Table 4.1 trivially increase to $\Omega(n)$ in the worst case when just using an external network, our results demonstrate that with the help of hybrid networks, monitoring can be done exponentially faster in comparison to just having an external network.

Underlying Publication This chapter is based on the following publication.

> R. Gmyr, K. Hinnenthal, C. Scheideler, and C. Sohler. “Distributed Monitoring of Network Properties: The Power of Hybrid Networks”. In: *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*, see [Gmy+17].

Outline After we survey the related literature in Section 4.1, we begin the technical part of this chapter in Section 4.2 by introducing the hybrid network model and the concept of network monitoring problems. In Section 4.3 we present an algorithm that constructs a low-diameter tree overlay from a given graph. This algorithm is used as a fundamental tool throughout the remainder of this chapter. In the next three sections we describe and analyze several network monitoring algorithms ordered by increasing algorithmic complexity: We begin in Section 4.4 with a simple approach for monitoring problems that can be solved by aggregating values stored at the nodes. We demonstrate the

Monitoring Problem	Setup Time	Delay
Number of Edges	$O(\log^2 n)$	$O\left(\frac{\log n}{\log\log n}\right)$
Average Node Degree	$O(\log^2 n)$	$O\left(\frac{\log n}{\log\log n}\right)$
Clustering Coefficient	$O(\log^2 n)$	$O\left(\frac{\log n}{\log\log n}\right)$
Bipartiteness	0	$O(\log^2 n)$
Exact MST Weight	0	$O(\log^2 n)$
Approximate MST Weight	$O(\log^2 n)$	$O\left(\frac{1}{\varepsilon} \cdot \log^2\left(\frac{W}{\varepsilon}\right) + \frac{\log n}{\log\log n}\right)$

Table 4.1: Results presented in this chapter. The variable W is the maximum edge weight in the external network. The exact algorithm for monitoring the MST weight requires W to be at most polylogarithmic in n. The approximation algorithm requires W to be at most polynomial in n and it computes the MST weight M up to an additive term of $\pm\varepsilon M$.

usefulness of this approach by applying it to monitor the number of edges, the average node degree, and the clustering coefficient of the external network. In Section 4.5 we present an algorithm that monitors whether the external network forms a bipartite graph. Finally, in Section 4.6 we consider the problem of monitoring the weight of a minimum spanning tree of the external network. We present both an exact algorithm for this problem and a more efficient approximation algorithm. We close the chapter in Section 4.7 with a brief outlook.

4.1 Related Work

In the networking community the term *hybrid network* has been used to refer to networks containing equipment from multiple vendors, networks using different networking technologies or standards, and networks incorporating both peer-to-peer and client-server approaches. These topics are not related to our work, so we do not consider them.

There is a large body of literature on *overlay networks*, especially in the context of peer-to-peer systems. Whereas most of the proposed overlay networks do not take the underlying network into consideration, there is also a number of proposals for so-called locality-aware overlays with prominent examples such as Tapestry [Zha+04] and Pastry [RD01]. However, these constructions are only concerned about adapting or optimizing the overlay to the underlying network and do not aim at monitoring properties of the underlying network with the help of the overlay.

In the *dynamic graph* model introduced by Kuhn et al. [KLO10] a set of nodes is connected by a network that lies completely under the control of an adversary in that the adversary can change the edge set in every round. The authors investigate whether the network can perform certain tasks despite the presence of such a strong adversary. Specifically, the authors focus on the counting and the token dissemination problem. Further research in this direction has been conducted in [HK11] and [Dut+13], for example. Abshoff and Meyer auf der Heide use a similar model to study the continuous aggregation problem in dynamic networks [AM14]. For a more general overview of models and algorithms for dynamic networks see [APR16]. Some of the problems investigated in this line of work are related to the monitoring problems we consider in this chapter. This especially holds for the work of Abshoff and Meyer auf der Heide [AM14]. However, the models used in this line of work differ significantly from our hybrid network model since in our model the nodes do not only communicate via an externally controlled network—they also have access to an overlay network that is controlled by the network algorithm.

Another related model is the *congested clique* model, which recently received a considerable amount of attention (see, e.g., [Lot+05; Len13; DKO14; Cen+15; Heg+15]). As the name suggests, the nodes in this model are connected to a clique, they communicate via synchronous message-passing, and each edge in the clique is limited to carry a certain number of bits per round (typically $O(\log n)$ bits). An example of a problem in the congested clique model would be to compute the minimum spanning tree of a graph that is specified by marking a subset of the edges in the clique. This example reveals that the congested clique model and the hybrid network model are similar in that both models consider two different networks or graphs: The clique in the congested clique model can be seen as an extreme example of an overlay network in the hybrid network model; and the graph induced by the marked edges in the congested clique model can be interpreted as an example of an external graph in the hybrid network model. However, aside from this high-level similarity the two models share few commonalities in both their technical properties and their goals. One of the most striking differences between the models is that in the hybrid network model a node is restricted to send and receive at most a polylogarithmic number of bits per round while this number is superlinear in the congested clique model. As a consequence, we cannot easily adapt algorithms from the congested clique model to our model.

Some of our algorithms apply techniques particularly known from the field of parallel computing. For example, we make extensive use of *pointer jumping* [JáJ92], a technique that is often used in algorithms for parallel random-access machines (or *PRAMs*) and that already played a major role in Chapter 2 of this thesis. The algorithm presented in Section 4.3 uses pointer jumping in a way that is similar to its application in [AV84; TV85], e.g., and the algorithm shares some further similarities with the algorithm presented in [JM95]. There is an abundance of parallel algorithms for computing minimum spanning trees on PRAMs, the best of which achieve a running time of $O(\log n)$ (see [HZ01] for an overview). However, we are not aware of any distributed implementation of such an algorithm with a running time of $o(\log^2 n)$ that does not cause high node congestion. Therefore, we cannot directly apply these algorithms to improve the results on monitoring the weight of a minimum spanning tree we present in Section 4.6.

The algorithm presented in Section 4.3 transforms a given graph into a rooted tree of constant degree and depth $O(\log n)$. This algorithm is frequently used as a subroutine by the monitoring algorithms presented in this chapter. Angluin et al. [Ang+05] proposed a similar algorithm for overlay networks that achieves the same result and that even works in an asynchronous setting. However, the algorithm of Angluin et al. is randomized while our algorithm is deterministic. Since the remaining parts of our monitoring algorithms are also deterministic, using our new algorithm results in fully deterministic monitoring algorithms. Even more importantly, our algorithm can also be used for the efficient construction of a spanning tree of a given graph, which is another subroutine we use in our monitoring algorithms. This cannot directly be achieved using the algorithm of Angluin et al.

In the algorithms presented in Section 4.4 the monitor continuously collects data from the nodes of the network by performing *aggregation*. There is a large amount of work on aggregation in the context of sensor networks, but research in this area has focused on monitoring environmental properties or the state of systems (such as bridges or airplanes) or facilities (such as warehouses) and not properties of graphs. Distributed aggregation has also been studied extensively for conventional, static networks (see, e.g., [AW04; KLW07; KLS08] or [Loc09] for a comprehensive overview), but not for hybrid forms as considered here.

In Section 4.6 we consider the problem of monitoring the weight of a minimum spanning tree (or *MST*). The problem of computing an MST in a distributed manner is well studied (see, e.g., [PR99; Elk04; Elk06; PRS16]), but the algorithms in this area do not consider overlay network techniques. The problem of computing only the *weight* of an MST instead of the MST itself has been studied in the field of sequential sublinear algorithms. This line of research was initiated by Chazelle et al. [CRT05] and continued in [Czu+05; CS09; BKM14]. Our algorithms for monitoring the weight of an MST apply the ideas of Chazelle et al. [CRT05] in a distributed context and also incorporate some ideas from [CS09].

4.2 Model and Problem Statement

We consider hybrid networks with a *static* node set and a *dynamic* edge set. Time proceeds in *synchronous rounds*, and for each round i we are given a set of undirected edges E_i. The *external network* in round i is represented by the undirected graph $G_i = (V, E_i)$. An algorithm has no control over the edges in E_i. However, it can establish additional *overlay edges* to form an *internal network* or *overlay network*: Each node u has a unique *identifier* $\mathrm{id}(u)$ which is a bit string of length $O(\log n)$ where $n = |V|$. Let $D_i(u)$ be the set of identifiers stored by a node u at the beginning of round i. We define the set of overlay edges in round i as $D_i = \{\, (u, v) \mid u \in V \text{ and } v \in D_i(u) \,\}$. A node has immediate access to the identifiers of its neighbors in G_i and can store such an identifier for future reference. In round i, a node u can send a distinct message to each node v such that $\{u, v\} \in E_i$ or $(u, v) \in D_i$. A message sent in round i arrives at the beginning of round $i + 1$. The local memory and computation of the nodes is unbounded. However, a node can send and receive at most a polylogarithmic number of bits in each round.

We investigate *monitoring problems*. In these problems, a designated node s that we call the *monitor node* or simply *monitor* has to continuously observe a property of the external network such as the number of edges or the weight of a minimum spanning tree. Formally, a property p is a function from the set of undirected graphs into some set of property values. Since the external network G_i is dynamic, the property value $p(G_i)$ can change from round to round. We say an algorithm *monitors a property* p with *setup time* i_0 and *delay* δ if for all rounds $i \geq i_0$ the monitor node outputs the property value $p(G_i)$ by round $i + \delta$. We refer to the first i_0 rounds in the execution of a monitoring algorithm as the *setup phase* and to the remaining rounds as the *monitoring phase*. Initially, the set of overlay edges D_0 is empty. An algorithm can use the setup phase to construct an initial internal network that supports the computation of the property value. It can continue to adapt the internal network during the monitoring phase. We assume that the degree of G_i is polylogarithmic for all i and that the graph G_0 is connected. Beyond this, we make no assumptions about the evolution of the edge set E_i.

4.3 Setup Phase

All monitoring algorithms presented in this chapter that rely on a dedicated setup phase use a common algorithm for the construction of the initial overlay network. This algorithm organizes the nodes into a tree T of polylogarithmic degree and depth $O(\log n / \log \log n)$ that is rooted at the monitor. In this section, we first present a more general algorithm that we refer to as the *Overlay Construction Algorithm*. This algorithm shares some similarities with an algorithm of Angluin et al. [Ang+05]. The algorithm is frequently used as a subroutine throughout the remainder of this chapter. On the basis of this algorithm, we describe at the end of the section how the desired tree T can be constructed. For simplicity, we assume that every node knows the total number of nodes n. The algorithms can be modified to remove this assumption.

For a given bidirected connected graph G of polylogarithmic degree, the Overlay Construction Algorithm arranges the nodes of G into a tree of constant degree and depth $O(\log n)$. On a high level, the algorithm works as follows. It operates on *supernodes* which are groups of nodes that act in coordination. Let the identifier of a supernode be the highest identifier of the nodes it contains. Define two supernodes u, v to be adjacent if there are nodes x, y that are adjacent in G such that $x \in u$ and $y \in v$. Initially, each node forms a supernode on its own. The algorithm alternately executes a *grouping step* and a *merging step*. In the grouping step, each supernode u determines the neighboring supernode v with the highest identifier. If $\mathrm{id}(v) > \mathrm{id}(u)$ then u sends a *merge request* to v. Consider the graph whose node set is the set of all supernodes and that contains a directed edge (u, v) if u sent a merge request to v. Since each supernode sends at most one merge request and a merge request is always sent from a supernode with a lower identifier to a supernode with a higher identifier, this graph is a forest. During the merging step, each tree of this forest is merged into a new supernode. Before we describe how this high-level algorithm can be implemented by the nodes, we analyze the number of iterations of consecutive grouping and merging steps required until only a single supernode remains.

Lemma 4.1. *After $O(\log n)$ iterations only a single supernode remains.*

Proof. Consider a supernode u at the beginning of a grouping step. We show that u merges with another supernode within at most two iterations. If u merges with another supernode in the current iteration, the claim holds. So suppose u does not merge with another supernode. This implies that u has a higher identifier than all of its neighbors. Consider any neighbor v of u. Since $\mathrm{id}(v) < \mathrm{id}(u)$ and v does not merge with u, the supernode v must merge with a neighbor w such that $\mathrm{id}(w) > \mathrm{id}(u)$. Therefore, at the beginning of the next grouping step, u has a neighbor with an identifier that is larger than $\mathrm{id}(u)$. This implies that u merges with another supernode in that iteration. Since every supernode merges after at most two iterations, it takes at most $O(\log n)$ iterations until only a single supernode remains. □

We now describe how the nodes can locally implement the high-level behavior of the supernodes described above. At the beginning of every grouping step, the following *invariant* holds: Each supernode is internally organized as a tree of constant degree and depth $O(\log n)$ that is rooted at the node with the highest identifier, and each node in such a tree knows the identifier of the root, which corresponds to the identifier of the supernode. The nodes cooperatively simulate the behavior of their respective supernodes during the grouping step as follows. Consider a supernode u and the corresponding internal tree T_u. First, every node of the supernode u sends $\mathrm{id}(u)$ along every incident edge in the original graph G. Thereby, every node learns the identifiers of its neighboring supernodes. Then, the nodes of u use a convergecast along T_u to determine the identifier of the supernode v with the highest identifier among the neighbors of u. This convergecast also collects the identifier of the node x with the highest identifier in u that is adjacent to a node in v. Once this convergecast is complete, the root of T_u knows both $\mathrm{id}(v)$ and $\mathrm{id}(x)$. If $\mathrm{id}(v) > \mathrm{id}(u)$ then the root of u sends a message to x. Upon receiving this message, x sends a merge request to a neighboring node in v and sends a broadcast through T_u to establish itself as the new root of T_u. All nodes in G wait until $O(\log n)$ rounds have passed before they proceed to the merging step. Thereby, the above operations can be completed in all supernodes and all nodes start the merging step at the same time.

At the beginning of every merging step, we have the following situation. Consider the graph consisting of the internal trees of all supernodes together with all edges along which a merge request has been sent. This graph is a forest and the trees of this forest form the new supernodes resulting from the merging step. Therefore, the nodes of each new supernode v are already arranged into a tree T_v. Furthermore, each supernode v contains exactly one former root node that did not instruct a node to send a merge request. It is not hard to see that this node has the highest identifier in v and, therefore, it becomes the root of T_v. In its current state, T_v can have up to polylogarithmic degree and linear depth. To restore the invariants required at the beginning of a grouping step, we have to transform T_v into a tree of constant degree and depth $O(\log n)$. Furthermore, we have to make sure that all nodes in v know the identifier $\mathrm{id}(v)$ of the root of T_v.

First, we reorganize T_v into a *child-sibling tree*. For this, each inner node y arranges its children into a path sorted by increasing identifier and keeps only the child with the lowest identifier. Each former child of y changes its parent to be its predecessor on the path and stores its successor as a *sibling*. In the resulting child-sibling tree, each node stores at most three identifiers: a parent, a sibling, and a child. By interpreting the sibling of a node as a second child, we get a binary tree. This transformation of T_v into a binary tree takes $O(1)$ rounds. Note that the transformation potentially increases the depth of T_v. However, this does not pose a problem for our algorithm.

On the basis of this binary tree, we construct a *ring* of *virtual nodes* in the following way. Consider the depth-first traversal of the tree that visits the children of each node in order of increasing identifier. A node occurs at most three times in this traversal. Let each node act as a distinct virtual node for each such occurrence and let $k \leq 3n$ be the resulting number of virtual nodes. A node can locally determine the predecessor and successor of its virtual nodes according to the traversal. Therefore, the nodes can connect their virtual nodes into a ring in $O(1)$ rounds.

Next, we use pointer jumping to quickly add *chords* (i.e., shortcut edges) to the ring. The virtual nodes execute the following protocol for $\lfloor \log n \rfloor + 1$ rounds. Each virtual node y learns two identifiers ℓ_t and r_t in each round t of this protocol. Let ℓ_0 and r_0 be the predecessor and successor of y in the ring. In round t, y sends ℓ_t to r_t and vice versa. At the beginning of round $t+1$, y receives one identifier from ℓ_t and r_t, respectively. It sets ℓ_{t+1} to the identifier received from ℓ_t and r_{t+1} to the identifier received from r_t. It then proceeds to the next round of the protocol. In every round of this protocol each virtual node adds a new chord to the ring by introducing its latest neighbors to each other. The distance between these neighbors with respect to the ring doubles from round to round up to the point where the distance exceeds the number of virtual nodes k. On the basis of this observation, it is not hard to see that after the specified number of rounds, the diameter of the graph has reduced to $O(\log n)$ while the degree has grown to $O(\log n)$. Once the protocol terminates, the root of v initiates a broadcast from one of its virtual nodes followed by a convergecast to determine the number of virtual nodes k.

Finally, we use the chords to construct a binary tree of depth $O(\log n)$ that spans all virtual nodes. For this, one of the virtual nodes of the root of v initiates a broadcast by sending a message to its neighbors $\ell_{t'}$ and $r_{t'}$ where $t' = \lfloor \log k \rfloor - 1$. Note that $t' \leq \lfloor \log n \rfloor + 1$ and, therefore, these neighbors exist. A virtual node that receives this broadcast for the first time after t steps forwards it to $\ell_{t'}$ and $r_{t'}$ where $t' = \max\{\lfloor \log k \rfloor - t - 1, 0\}$. It is not hard to see that the binary tree induced by the edges used in this broadcast has depth $O(\log n)$ and contains all nodes of the ring. At this point, the nodes discard all overlay edges constructed so far and only keep the edges of the binary tree. We then merge the virtual nodes back together such that each node adopts the edges of its virtual nodes. This results in a graph of constant degree and diameter $O(\log n)$. Note that this graph is not necessarily a tree. To construct a tree that satisfies the invariants for the grouping step, the root of v sends another broadcast through the resulting graph to construct a breadth-first search tree that has constant degree and diameter $O(\log n)$. This broadcast also informs all nodes in v about $\mathrm{id}(v)$. The operations described above take $O(\log n)$ rounds overall. As before, all nodes in G wait for $O(\log n)$ rounds to pass so that they enter the next grouping step at the same time.

Once only a single supernode u remains, its internal tree T_u covers all nodes of G and has the desired properties, i.e., T_u has constant degree and depth $O(\log n)$. A supernode is the last remaining supernode if and only if during the grouping step no node reports the identifier of a neighboring supernode.

Therefore, the root of a supernode can determine whether its supernode is the last remaining supernode. Since the algorithm runs for $O(\log n)$ iterations according to Lemma 4.1 and each iteration takes $O(\log n)$ rounds, we have the following theorem.

Theorem 4.2. *Given any connected bidirected graph G of n nodes and polylogarithmic degree, the Overlay Construction Algorithm constructs a constant degree tree of depth $O(\log n)$ that contains all nodes of G and that is rooted at the node with the highest identifier. The algorithm takes $O(\log^2 n)$ rounds.*

Theorem 4.2 implies the following corollary.

Corollary 4.3. *Consider a bidirected graph G of n nodes and polylogarithmic degree. For each connected component C of G, the Overlay Construction Algorithm constructs a constant degree tree of depth $O(\log |C|)$ that contains all nodes of C and that is rooted at the node with the highest identifier in C. The algorithm takes $O(\log^2 |C|)$ rounds in each component and $O(\log^2 n)$ rounds overall.*

Finally, note that the algorithm sends merge requests only along edges of G. This gives rise to the following observation.

Observation 4.4. *For a connected bidirected graph G, the edges along which the Overlay Construction Algorithm sends the merge requests induce a spanning tree of G.*

Observation 4.4 implies that by letting the nodes locally mark the edges that carry a merge request, the Overlay Construction Algorithm can be used for the distributed construction of a spanning tree of G in $O(\log^2 n)$ time. While this observation is not immediately relevant for the setup phase, it will become useful in Section 4.5.

On the basis of the Overlay Construction Algorithm, it is straight-forward to achieve the goal of the setup phase, which is to organize the nodes into a tree T of polylogarithmic degree and depth $O(\log n / \log\log n)$ that is rooted at the monitor s. At the beginning of the setup phase, each node u creates an overlay edge (u, v) for each edge $\{u, v\} \in E_0$ by copying the identifiers of its neighbors in the external network G_0 to its local memory. This effectively creates a bidirected overlay network that corresponds to the undirected graph G_0. Note that G_0 is connected by assumption. Therefore, we can use the Overlay Construction Algorithm to construct a tree of constant degree and depth $O(\log n)$ that contains all nodes in the network. Once the algorithm terminates, s broadcasts a message through the resulting tree to establish itself as the new root. This does not increase the asymptotic depth of the tree. We then decrease the depth to $O(\log n / \log\log n)$ as follows. Each node

u broadcasts its identifier down the tree up to a distance of $\lceil \log \log n \rceil$. Once this broadcast is complete, a node that received the broadcast of u establishes an edge to u. It is not hard to see that this creates a graph of at most polylogarithmic degree and diameter $O(\log n/\log \log n)$. Finally, s sends a broadcast through this graph to create a breadth-first search tree that has the desired properties. We have the following theorem.

Theorem 4.5. *A setup time of $O(\log^2 n)$ rounds is sufficient to organize the nodes of the network into a tree T of polylogarithmic degree and depth $O(\log n/\log \log n)$.*

Unless otherwise stated, we assume in the following sections that the setup phase is executed as described above.

4.4 Three Simple Monitoring Problems

In order to introduce some basic concepts that underlie all monitoring algorithms presented throughout this chapter, we first consider three simple monitoring problems. Specifically, we show how to monitor the number of edges, the average node degree, and the clustering coefficient of the external network by using *aggregation* on the tree T constructed during the setup phase.

Consider the problem of monitoring the number of edges. We first present an algorithm that efficiently determines the number of edges in a static graph and then show how this algorithm can be used to continuously monitor the number of edges in the external network. It is well known that the number of edges in a graph is $|E| = 1/2 \cdot \sum_{u \in V} \deg(u)$ where $\deg(u)$ is the degree of a node u. Therefore, we can compute $|E|$ by aggregating the sum of all node degrees along the tree T in the following way. In the first round, each leaf node u sends $\deg(u)$ to its parent. Once an inner node u has received a value x_j from each of its children, it sends $\deg(u) + \sum_j x_j$ to its parent. After $O(\log n/\log \log n)$ rounds, the monitor s has received a value from each of its children. It can use these values together with its own degree to compute $|E|$ as described above.

To continuously monitor $|E_i|$ for every $i \geq i_0$, we execute multiple instances of the above algorithm in a *pipelined* fashion: In each round $i \geq i_0$ a new instance of the algorithm is started. The instances run in parallel and do not interact with each other. At the beginning of a round i, each node copies the identifiers of its neighbors in G_i to its local memory. This creates a copy of the graph G_i in form of an overlay network on which the corresponding instance of the algorithm can operate. The copy is discarded once the instance terminates. The messages sent by an instance of the algorithm are labeled with the round number in which the instance was started so that received messages can be correctly assigned. Note that the number of bits a node sends and receives per round in one instance of the algorithm is polylogarithmic. Since the running time of the algorithm is also polylogarithmic, the number of bits a node sends and receives in the pipelined execution is polylogarithmic as well. The running time of the algorithm becomes the delay of the pipelined execution. Therefore, we have following theorem.

Theorem 4.6. *The number of edges can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

In the remainder of this chapter, we only present algorithms that compute the value of a network property for a static graph and implicitly assume that the respective algorithm is executed in a pipelined fashion to solve the monitoring problem under consideration.

On the basis of the ideas of the algorithm above, it is easy to solve a number of monitoring problems that can be reduced to aggregation. For example, one can monitor the average node degree by letting s multiply the result of the given algorithm with $2/n$ before outputting it. This gives us the following corollary.

Corollary 4.7. *The average node degree can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

As a final example, we consider the *clustering coefficient* of a network [WS98]. Intuitively, the clustering coefficient reflects the relative number of triangles in a graph G. It is particularly relevant in the context of biological and social networks (see, e.g., [New01; NSW01; NWS02; WS98]). Formally, the clustering coefficient of a node u is defined as

$$C(u) = \frac{2 \cdot |\{\, v, w \in N(u) \mid \{v, w\} \in E \,\}|}{\deg(u) \cdot (\deg(u) - 1)},$$

where $N(u)$ is the set of neighbors of u. The clustering coefficient of a graph G is defined as $C(G) = 1/n \cdot \sum_{u \in V} C(u)$. Each node u can compute $C(u)$ in constant time by communicating with its neighbors. Therefore, $C(G)$ can be computed by aggregating the sum of all $C(u)$ along T and dividing the result by n at the monitor. This implies the following theorem.

Theorem 4.8. *The clustering coefficient can be monitored with setup time $O(\log^2 n)$ and delay $O(\log n / \log\log n)$.*

4.5 Bipartiteness

In this section, we consider the problem of monitoring whether the external network forms a *bipartite* graph. For now, we assume that the external network is connected in each round. As we will argue towards the end of this section, it is straight-forward to remove this assumption. Our algorithm is based on the following well-known approach (see, e.g., [KT13]): Given a connected graph

G, compute a rooted spanning tree of G and assign a color from $\{0, 1\}$ to each node u that corresponds to the parity of the depth of u in the spanning tree. We say an edge in G is *valid* if it connects nodes of different colors and it is *invalid* otherwise. G is bipartite if and only if all edges in G are valid. It remains to show how this approach can be implemented efficiently in our framework.

According to Observation 4.4, we can use the Overlay Construction Algorithm to mark the edges of a spanning tree S of G. Note that S is not rooted at any particular node. Therefore, two nodes that are connected by an edge in S cannot locally decide which of them is the parent of the other. From a global perspective, we consider the monitor s to be the root of S. Each node has to determine the parity of its depth in S to determine its color. Since S might have linear depth, a simple broadcast from s does not constitute an efficient solution to this problem. Instead, we use pointer jumping along a depth-first traversal of S to determine the colors of the nodes. We define the traversal of S as follows. The traversal starts at s and moves to the neighbor of s in S with the lowest identifier. For a node u let $u_0, u_1, \ldots, u_{\deg(u)-1}$ be the neighbors of u in S arranged by increasing identifier. When the traversal reaches u from a node u_i, it continues to node $u_{(i+1) \bmod \deg(u)}$. The traversal finishes when it reaches s from the neighbor of s in S with the highest identifier. Define the *traversal distance* $d(u)$ to be the number of steps required to reach u for the first time in this traversal. We will show in Lemma 4.9 that the parity of $d(u)$ equals the parity of the depth of u in S. For now, we focus on computing $d(u)$ efficiently for all nodes.

Each node u simulates one virtual node for each occurrence of u in the traversal, and the nodes connect these virtual nodes into a ring. More specifically, each node u simulates virtual nodes $v_0, v_1, \ldots, v_{\deg(u)-1}$ such that v_i is the predecessor of a virtual node of u_i and the successor of a virtual node of $u_{(i-1) \bmod \deg(u)}$ in the ring. Since S is a tree, the ring consists of $2(n-1)$ virtual nodes. Setting up the virtual nodes takes $O(1)$ rounds.

We use pointer jumping to add chords to the ring following the same protocol we used during the merging step of the Overlay Construction Algorithm. We define ℓ_0 to be the successor of a virtual node and r_0 to be the predecessor of a virtual node. We execute the protocol for $t = \lfloor \log(2(n-1)) \rfloor$ rounds so that each node constructs chords ℓ_i and r_i for each $1 \leq i \leq t$. Each chord ℓ_i and r_i bridges a distance of exactly 2^i along the ring. The chords allow us to efficiently compute the values $d(u)$ in the following way. Let v^* be the virtual node simulated by s that precedes a virtual node of the neighbor of s with the lowest identifier (i.e., $v^* = v_0$ from the local perspective of s). The virtual node v^* stores the value 0 and initiates a broadcast by sending a message with value 2^i along each chord ℓ_i for $0 \leq i \leq t$. Consider a virtual node that receives a broadcast message and that does not yet store a value. Let x be the value associated with the received message. The virtual node stores x and sends a message containing the value $x + 2^i$ along each chord ℓ_i for $0 \leq i \leq t$. It is not hard to see that this broadcast reaches all virtual nodes within $O(\log n)$ rounds, and the value stored at a virtual node v after the broadcast finishes corresponds to the length of the path from v^* to v along the ring. Therefore, each node u can determine its traversal distance $d(u)$ by taking the minimum of the values stored at its virtual nodes. Once all nodes have computed their traversal distances in this way, each node u determines whether it is incident to an invalid edge by checking for each neighbor w in G whether $d(u) \equiv d(w)$ mod 2. Then, the nodes use a convergecast to inform s whether there is an invalid edge. If so, s outputs that G is not bipartite. Otherwise, s outputs that G is bipartite.

To establish the correctness of the algorithm, it remains to show the following lemma.

Lemma 4.9. *For each node u, the parity of the depth of u equals the parity of $d(u)$.*

Proof. Recall that the tree is traversed in a depth-first order. Therefore, the traversal takes an even number of steps between any two visits of the same node. Let P be the shortest path from s to u in S. The length of P equals the depth of u. The traversal follows P but it takes a detour whenever it explores a branch outside of P. By the argument above, each such detour has even length. Therefore, the parity of the depth of u equals the parity of $d(u)$. □

Note that the delay of the algorithm is dominated by the Overlay Construction Algorithm, which takes $O(\log^2 n)$ rounds to construct the spanning tree S according to Theorem 4.2. Therefore, we can eliminate the setup phase by constructing the tree T during the monitoring phase, which does not asymptotically increase the delay. If the external graph is disconnected we can perform the above algorithm on each connected component in parallel and aggregate the individual results along the tree T. This implies the following theorem.

Theorem 4.10. *Bipartiteness can be monitored with setup time* 0 *and delay* $O(\log^2 n)$.

4.6 Minimum Spanning Tree

We now turn to the problem of monitoring the *weight* of a *minimum spanning tree* (or *MST*). As before we assume that the edge set can change from round to round. However, in this section we require the external network to be connected in every round. Additionally, we associate weights with the edges that can change every round. In Section 4.6.1 we present an algorithm that monitors the exact MST weight and in Section 4.6.2 we present an algorithm that monitors an approximation of the MST weight with an improved delay. Both algorithms are based on a sequential approximation algorithm by Chazelle et al. [CRT05].

4.6.1 Exact MST Weight

The main idea behind the algorithm is to reduce the computation of the weight of an MST in a graph G to counting the number of connected components in certain subgraphs of G. This idea was first introduced by Chazelle et al. [CRT05]. We assume that the edge weights are taken from the set $\{1, 2, \dots, W\}$ for some $W \in \mathbb{N}$ that is bounded polylogarithmically in n. Define the *threshold graph* $G^{(\ell)}$ to be the subgraph of G consisting of all edges with weight at most ℓ, and define $c^{(\ell)}$ to be the number of connected components in $G^{(\ell)}$. The MST weight M can be computed from the values $c^{(\ell)}$ as shown in the following lemma.

Lemma 4.11 (Chazelle et al. [CRT05])**.** *In a graph with edge weights from* $\{1, 2, \dots, W\}$*, the MST weight is*

$$M = n - W + \sum_{i=1}^{W-1} c^{(i)}.$$

On the basis of Lemma 4.11, the monitor can compute the MST weight as follows. Consider the threshold graph $G^{(\ell)}$ for some $\ell \in \{1, 2, \dots, W-1\}$. According to Corollary 4.3, executing the Overlay Construction Algorithm on $G^{(\ell)}$ creates an overlay network in which each connected component of $G^{(\ell)}$ is spanned by a rooted tree of overlay edges. Each node knows whether it is a root of one of these trees. Therefore, we can determine $c^{(\ell)}$ by counting the number of roots, which can easily be achieved using aggregation along the tree T constructed during the setup phase. By iterating this process, the monitor learns the value $c^{(\ell)}$ for each $\ell \in \{1, 2, \dots, W-1\}$. It then uses the equation given in Lemma 4.11 to compute the MST weight. Since T is only used after the algorithm already ran for $O(\log^2 n)$ rounds, we can construct T during the monitoring phase and skip the setup phase. Furthermore, we can reduce the delay by computing up to $\log^2 n$ different $c^{(i)}$'s in parallel. This implies the following theorem.

Theorem 4.12. *For edge weights from* $\{1, 2, \dots, W\}$ *where* $W \in \mathbb{N}$ *is bounded polylogarithmically in* n*, the MST weight can be monitored with setup time* 0 *and delay* $O(W + \log^2 n)$*.*

Since by our assumption W is at most polylogarithmic in n, we can also compute all $W - 1$ values of the $c^{(i)}$'s in parallel at the cost of increased communication work per round. This gives us the following corollary.

Corollary 4.13. *For edge weights from* $\{1, 2, \dots, W\}$ *where* $W \in \mathbb{N}$ *is bounded polylogarithmically in* n*, the MST weight can be monitored with setup time* 0 *and delay* $O(\log^2 n)$*.*

4.6.2 Approximate MST Weight

Next, we present an algorithm in which the maximum edge weight W contributes only polylogarithmically instead of linearly to the delay of the algorithm. This improvement comes at the cost of a small approximation error. The algorithm is less restrictive in that it allows the edge weights to be real numbers from the interval $[1, W]$ for some $W \in \mathbb{R}$ that is bounded polynomially in n. The algorithm is based on the same general idea as the algorithm from the previous section but additionally incorporates ideas from the work of Czumaj and Sohler [CS09].

First, each node rounds up the edge weight of each incident edge to a power of $(1+\varepsilon)$ for a given ε with $0 < \varepsilon \leq 1$. In the resulting graph G', each edge weight is of the form $(1+\varepsilon)^i$ where $0 \leq i \leq \log_{1+\varepsilon} W$. Let M' be the MST weight in G', and let $c^{(\ell)}$ be the number of components in the threshold graph $G^{(\ell)}$ of G' as defined in the previous section. We have the following lemma, which is analogous to Lemma 4.11.

Lemma 4.14 (Czumaj and Sohler [CS09]). *In a graph with edge weights of the form $(1+\varepsilon)^i$ for $0 \leq i \leq \log_{1+\varepsilon} W$, the MST weight is*

$$M' = n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W - 1} (1+\varepsilon)^i \cdot c^{((1+\varepsilon)^i)}.$$

On the basis of Lemma 4.14, we can compute M' by determining the number of connected components $c^{((1+\varepsilon)^i)}$ in $\log_{1+\varepsilon} W$ many threshold graphs of G'. While this already implies an improvement over the algorithm from the previous section, we can further reduce the delay by ignoring large components in the threshold graphs.

Consider some threshold graph $G^{((1+\varepsilon)^i)}$. We execute the Overlay Construction Algorithm as in the previous section but we stop its execution after $O(\log^2(2W/\varepsilon))$ rounds. By Corollary 4.3, the algorithm is guaranteed to finish its computation in each connected component of size at most $2W/\varepsilon$. In larger connected components, the algorithm may finish but is not guaranteed to do so. It is not hard to modify the algorithm such that all nodes of a connected component know whether the algorithm finished its computation for that connected component. This allows us to ignore root nodes in connected components for which the algorithm did not finish. Thereby, the algorithm establishes a unique root node for each connected component of size at most $2W/\varepsilon$ while in each larger connected component either a unique root node is established or no root is established. Let $\hat{c}^{((1+\varepsilon)^i)}$ be the number of root nodes established in this way. The nodes determine the value of $\hat{c}^{((1+\varepsilon)^i)}$ using aggregation along the tree T constructed in the setup phase. We iteratively execute this process for each i such that $0 \leq i < \log_{1+\varepsilon} W$. After the Overlay Construction Algorithm finishes for an iteration, the nodes start the aggregation for counting the number of roots and, at the same time, start the next execution of the Overlay Construction Algorithm for the following iteration. Thereby, we slightly interleave consecutive iterations, which reduces the overall delay. After the monitor has learned the values $\hat{c}^{((1+\varepsilon)^i)}$, it computes and outputs

$$\hat{M} = n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \cdot \hat{c}^{((1+\varepsilon)^i)}.$$

We have the following theorem.

Theorem 4.15. *For edge weights from $[1, W]$ where $W \in \mathbb{R}$ is bounded polynomially in n, the MST weight M can be monitored up to an additive term of $\pm\varepsilon M$ for any $0 < \varepsilon \leq 1$ with setup time $O(\log^2 n)$ and delay*

$$O\left(\frac{\log W}{\varepsilon} \cdot \log^2\left(\frac{W}{\varepsilon}\right) + \frac{\log n}{\log\log n}\right).$$

Proof. We first show the approximation factor. Rounding up the edge weights to a power of $(1+\varepsilon)$ increases the MST weight by a factor of at most $(1+\varepsilon)$. Therefore, we have

$$M \leq M' \leq (1+\varepsilon) \cdot M.$$

When computing the values $\hat{c}^{((1+\varepsilon)^i)}$, the algorithm potentially ignores all connected components of size larger than $2W/\varepsilon$. In each threshold graph there are at most $\varepsilon n/(2W)$ such connected components. The algorithm cannot overestimate the number of connected components in a threshold graph. Therefore, we have

$$c^{((1+\varepsilon)^i)} - \frac{\varepsilon n}{2W} \leq \hat{c}^{((1+\varepsilon)^i)} \leq c^{((1+\varepsilon)^i)}.$$

For the upper bound on the output $\hat{M}$ of the algorithm, the equations above together with the definitions of M' and $\hat{M}$ imply

$$\hat{M} \leq M' \leq (1+\varepsilon) \cdot M.$$

For the lower bound on $\hat{M}$, we have

$$\begin{aligned}
\hat{M} &= n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \cdot \hat{c}^{((1+\varepsilon)^i)} \\
&\geq n - W + \varepsilon \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \cdot \left(c^{((1+\varepsilon)^i)} - \frac{\varepsilon n}{2W} \right) \\
&= M' - \frac{\varepsilon^2 n}{2W} \cdot \sum_{i=0}^{\log_{1+\varepsilon} W-1} (1+\varepsilon)^i \\
&\geq M' - \frac{\varepsilon}{2} \cdot n \\
&\geq M - \frac{\varepsilon}{2} \cdot n \\
&\geq M - \frac{\varepsilon}{2} \cdot 2M \\
&\geq (1-\varepsilon) \cdot M,
\end{aligned}$$

where we assume $n \geq 2$ so that $n \leq M+1 \leq 2M$ for the penultimate inequality.

We now turn to the delay of the algorithm. The algorithm iteratively computes the values $\hat{c}^{((1+\varepsilon)^i)}$ for $\log_{1+\varepsilon} W = O(\log(W)/\varepsilon)$ threshold graphs. In each of these iterations the modified Overlay Construction Algorithm is executed for $O(\log^2(W/\varepsilon))$ rounds. After the Overlay Construction Algorithm finishes in the last iteration, the nodes have to wait for the final aggregation to complete. This takes an additional $O(\log n/\log\log n)$ rounds. □

Finally, since W is bounded polynomially in n, we can execute $\log W$ iterations of the algorithm in parallel, which gives us the following corollary.

Corollary 4.16. *For edge weights from $[1, W]$ where $W \in \mathbb{R}$ is bounded polynomially in n, the MST weight M can be monitored up to an additive term of $\pm\varepsilon M$ for any $0 < \varepsilon \leq 1$ with setup time $O(\log^2 n)$ and delay*

$$O\left(\frac{1}{\varepsilon} \cdot \log^2\left(\frac{W}{\varepsilon}\right) + \frac{\log n}{\log\log n}\right).$$

4.7 Outlook

We only considered a small number of monitoring problems for hybrid networks. There is an abundance of classic problems in the literature that can be newly investigated under this framework. Furthermore, by focusing exclusively on monitoring problems we only scratched the surface of possible new research avenues opened up by considering hybrid networks. For example, our algorithms only communicate via the internal network and use the external network merely as an input to the monitoring algorithms. As we stated in the introduction of this chapter, there are scenarios in which it is natural to assume that communication via the external network is much cheaper than communication via the internal network. Accordingly, one could extend the model by associating different costs to messages sent over edges of the internal and the external network. In such a variant of the model it would be interesting to investigate the trade-off between the communication cost and the running time of protocols for various problems, also outside of the domain of network monitoring.

Part II

Programmable Matter

Chapter 5

Leader Election for Programmable Matter

Imagine a substance that can change its shape or other physical properties in a programmable fashion on the basis of user input or sensing of its environment. Such a substance would have a wide range of applications: For example, it could be used in engineering to monitor environmental and structural conditions on bridges or on the inside of nuclear reactors, possibly repairing small fissures autonomously. In medicine it could be used within our bodies to detect and coat an area where internal bleeding occurs, eliminating the need of immediate surgery. It could self-assemble into a stent opening a previously blocked vessel, or it could identify and isolate tumor cells without external intervention. Thereby, this substance could achieve tasks that are time-consuming, costly, dangerous, or even completely impossible for humans today.

A substance that can be programmed to change its physical properties is commonly referred to as *programmable matter*. While programmable matter might sound like science fiction, first small steps towards its realization have already been made. For example, in modular self-reconfiguring robotic systems (see, e.g., [Chi94; Yim+07]) a collection of basic robotic modules cooperates to achieve tasks such as the construction of shapes or enveloping objects. The progressing minification of mechatronic components might one day lead to robots that are so small that a collection of robots appears to the naked eye as a continuous piece of matter. Another example stems from the field of synthetic biology in which researchers have made progress towards programming biological cells. Currently, cells can only be programmed to perform simple computations (see, e.g., [Ben+01; Fri+09]). However, one could imagine to also control cell movement in a programmable fashion, which would pave the way for programmable matter consisting of a myriad of interacting cells.

While the physical implementation of programmable matter certainly is an enormous undertaking, making programmable matter useful also requires *efficient algorithms* to control the entities that form the programmable matter. We aim to develop such algorithms on the basis of the *amoebot model*, which facilitates rigorous algorithmic research on programmable matter in the Euclidean plane. In the amoebot model, programmable matter consists of a uniform set of simple computational units called *particles* that can move and bond to other particles and that use their bonds to exchange information. The particles act asynchronously and achieve locomotion by expanding and contracting, which resembles the amoeboid movement performed by certain biological cells (see, e.g., [AE07]). To achieve a collective goal, the particles have to self-organize in a distributed fashion without any central control.

Our goal in this part of the thesis is to investigate fundamental problems for programmable matter on the basis of the amoebot model. We begin our investigation in this chapter by considering the *leader election problem*, which requires a set of particles to select one particle as its unique leader. Leader election is a central and classic problem in distributed computing. Many problems like the *consensus problem* (all particles have to agree on some output value) can easily be solved once a leader has been elected. Leader election also allows for *symmetry breaking* in distributed systems, which in general is a necessary prerequisite to solve problems such as shape formation. We will further elaborate on this connection between leader election and shape formation at the beginning of the following chapter.

We present a local-control algorithm that solves the leader election problem in the amoebot model in $O(n)$ asynchronous rounds with high probability, where n is the number of particles. Our algorithm relies only on local information (e.g., particles do not have unique identifiers, they do not know n, and they do not have a global coordinate system), and it requires only a constant-size memory at each particle.

Underlying Publications The results presented in this chapter are based on two publications, the first of which is the following.

> 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: *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming (DNA)*, see [Der+15b].

In this work we laid the foundation for leader election in the amoebot model by presenting a leader election algorithm that achieves a running time that is linear in the number of particles *on expectation*. Due to the complexity of the algorithm, both its description and its analysis relied on simplifying assumptions such as global control and synchronous particle activation. We improved upon this result in the following publication.

> J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Improved Leader Election for Self-Organizing Programmable Matter". In: *Proceedings of the 13th International Symposium on Algorithms and Experiments for Wireless Networks (ALGOSENSORS)*. To appear, see [Day+17].

In this work we presented a leader election algorithm that is both simpler and more efficient than the previous algorithm. The algorithm achieves a linear running time *with high probability* instead of on expectation. Its simplicity allowed us to devise full local-control protocols and to remove any simplifying assumptions from the analysis.

While the algorithm and the analysis presented in chapter are based on the latter of the two publications, many of the ideas underlying the algorithm go back to the first publication. Section 5.6, which discusses variants of the leader election problem, is based on both works. Specifically, the positive results are based on the newer publication and the negative result stems from the earlier publication.

Outline We begin in Section 5.1 by introducing the amoebot model, which serves as the foundation of our investigation of algorithms for programmable matter. We then give an overview of the related literature in Section 5.2. To provide a consistent picture, this section also covers the related work concerning shape formation, which is the subject of the following chapter. In Section 5.3 we formally define the leader election problem under the amoebot model. We then turn to the main part of this chapter, which consists of three sections: First, we present our leader election algorithm in Section 5.4. Then, we analyze the algorithm in Section 5.5. Finally, we expand on the topic of leader election for programmable matter by considering some variants of the leader election problem and investigating their feasibility in Section 5.6. We close this chapter in Section 5.7 with a discussion of further variants of the leader election problem that could be considered in future research.

5.1 The Amoebot Model

In the amoebot model, programmable matter consists of simple computational units that we call *particles*. The particles occupy the nodes of the *infinite triangular grid graph* G_{ET} that is embedded into the Euclidean plane as depicted in Figure 5.1. A particle occupies either a single node or a pair of adjacent nodes in G_{ET}, 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*.

Particles move through *expansion* and *contraction*: If a particle occupies one node, it can *expand* to an unoccupied adjacent node to then occupy two nodes. If a particle occupies two nodes, it can *contract* to one of these nodes to then

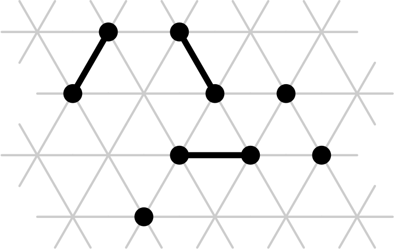

Figure 5.1: Example of a set of particles on the underlying infinite triangular grid graph G_{ET}. The figure shows a section of G_{ET} as a gray grid. A contracted particle is depicted as a black circle. An expanded particle is depicted as two black circles connected by a black line.

occupy a single node. We refer to a particle occupying one node as a *contracted* particle and we refer to a particle that occupies two nodes as an *expanded* particle, see Figure 5.1. Two particles can move in coordination by combining an expansion and a contraction in what we call a *handover*. There are two variants of a handover: First, a contracted particle p can *push* a neighboring expanded particle q and expand into a node previously occupied by q, forcing q to contract. Second, an expanded particle p can *pull* a neighboring contracted particle q to a node v occupied by p, causing q to expand into v and allowing p to contract. In both cases the handover effectively allows a pair of particles to move without disconnecting from each other.

For an expanded particle, we refer to one of the nodes occupied by the particle as its *head* and call the other node its *tail*. For a contracted particle, the single node occupied by the particle is both its head and its tail. When a contracted particle expands to a node, that node becomes the head of the particle while the other node becomes its tail. The head and the tail of a particle each have six *ports*, one for each incident edge. The ports are *labeled* in the following way: A particle considers one of the six directions in the grid formed by G_{ET} to be its *local north direction*. The local north direction of a particle does not change even when the particle moves. Each particle has its own local north direction that can differ from the local north direction of other particles. Hence, the particles do not have a common compass. They do however have *common chirality*, i.e., all particles have a common notion of clockwise rotation in the Euclidean plane. A particle labels the ports of its head (resp. its tail) with the numbers 0 to 5 in clockwise order starting at the port that points in its local north direction.

Neighboring particles can communicate through facing ports. When two particles p and q are connected by an edge, p knows the label of q's port that is incident to the edge. Furthermore, p knows whether q's port belongs to the head or tail of q. Each particle has a *local memory* that can be accessed by any neighboring particle. Particles exchange information with their neighbors by simply writing into their memory. A particle always knows whether it is contracted or expanded, and in the latter case it also knows which head port points towards its tail. This information is also available to neighboring particles. Each particle has access to a local source of random bits. The particles are *anonymous*, i.e., they do not have unique identifiers. Furthermore, they do not possess any global knowledge such as the overall number of particles.

We refer to a set of particles as a *particle system.* The computation of a particle system progresses through a sequence of *atomic particle activations*, i.e., only one particle is active at a time. Whenever a particle is activated, it can perform an arbitrary computation involving its local memory, the memories of its neighbors, and random bits, and it can perform at most one movement, i.e., an expansion, a contraction, a pull, or a push. We measure time in terms of *asynchronous rounds*, where we define a round to be complete after each particle has been activated at least once. We assume the activation of the particles to be *fair* in that at all times each particle is activated eventually.

Throughout this part of the thesis we make two assumptions that are not necessarily part of the amoebot model but that lie at the core of our investigation of programmable matter. First, we assume that the memory of each particle is of *constant size.* Thereby, the computational power of a particle is restricted to that of a probabilistic finite-state machine. We make this assumption to guarantee that our algorithms can, in principle, scale to an arbitrary number of particles without memory constraints becoming an issue. Second, we require the particle system to form a *connected structure* at all times, i.e., the subgraph of G_{ET} induced by the occupied nodes has to be connected. The reason for this requirement is that in a physical realization of programmable matter, disconnected parts of a particle system might separate irreversibly from the particle system by falling off or drifting away.

The leader election algorithm presented in this chapter does not move the particles. Therefore, large parts of the amoebot model remain unused for now. However, we will make use of the complete model in the subsequent chapter on shape formation with programmable matter.

5.2 Related Work

While some of the basic ideas behind programmable matter are already more than two decades old [TM91], rigorous algorithmic research that *explicitly* deals with this subject is still quite sparse. There is, however, a plethora of work on models and systems that are *implicitly* related to programmable matter in varying degrees. Many of these approaches share some underlying concepts with our work in the amoebot model, but overall they differ significantly either in their assumptions or in their goals. One of our primary objectives in this section is to provide the reader with a general understanding of the context of our work by giving a non-exhaustive overview of some of the related models and the problems investigated therein.[1] To limit the scope of this section, we only give a brief description of the main features of each model and mostly focus on the problems of leader election and shape formation, which are the problems investigated in this part of the thesis. While some variant of shape formation has been considered in most of these models, the leader election problem has generally received less attention. Of course, leader election is one of the classic problems of distributed computing and has been investigated under various models outside the context of programmable matter. However, we are not aware of any leader election algorithm that fits the very restrictive setting of the amoebot model.

Before we begin our discussion of related models and systems, we give a comprehensive overview of the work in the amoebot model. We first presented a preliminary version of the amoebot model at the First Workshop on Biological Distributed Algorithms and later published this version of the model in [Der+14]. Since then, we presented several results revolving around the problems of leader election, shape formation, and coating. For the *leader election* problem, which is the topic of this chapter, we introduced a first algorithm in [Der+15b] and later presented an improved and simplified algorithm in [Day+17]. A comparison of these results can be found in the description of the underlying publications at the beginning of this chapter. In the context of *shape formation*, which is the topic of the following chapter, we presented a simple approach for building a line in [Der+15b], extended this approach to other simple shapes in [Der+15a], and finally presented a more general and efficient approach to shape formation in [Der+16b]. The latter publication forms the basis of the following chapter. A comparison to the earlier results can be found at the beginning of that chapter. Finally, we also considered the *coating* problem. In this problem a particle system is connected to a static object represented by a connected set of nodes in the underlying graph G_{ET}. The goal is to reorganize the particles to form layers around the given object, i.e., the particle system has to *coat* the object. We presented an algorithm for this problem in [Der+17] and showed that the algorithm has a linear running time in [Der+16a]. Our results on coating are not part of this thesis.

Recently, also other authors have adopted the amoebot model to devise algorithms for programmable matter: Cannon et al. [Can+16] presented an algorithm for the *compression* problem in which the particles have to gather as tightly together as possible. This work introduces an interesting new aspect to the study of the amoebot model by using an algorithm that is based on a

[1] The overview of the related models and systems given in this section is largely based on the publications underlying this part of the thesis [Der+15b; Der+16b; Day+17].

stochastic process and analyzing this algorithm using Markov chain techniques. A similar approach is used in [Arr+17] to address the *shortcut bridging* problem in which a set of particles has to optimize the shape of a bridge between two given objects to balance the competing factors of path length and bridge cost. Finally, Di Luna et al. [Di +17] very recently presented new results on shape formation that expand considerably on our previous results.

We now turn our attention to other models and systems that are related to programmable matter and our work. One can broadly distinguish between passive and active systems. In *passive* systems the computational entities either do not have any intelligence at all (but just move and bond on the basis of their structural properties or due to chemical interactions with the environment), or they have limited computational capabilities but cannot control their movements. In *active* systems, on the other hand, the entities can control the way they act and move in order to solve a specific task. The amoebot model falls squarely into the category of active systems.

Prominent examples of *passive* systems are DNA computing, tile-based self-assembly, and population protocols. The field of *DNA computing* was initiated by Adleman [Adl94], who demonstrated that DNA molecules can be used to solve an instance of the directed Hamiltonian path problem. More general results on the computational power of DNA followed soon after (see, e.g., [Bon+96; OR99] and the references therein). Aside from using biological matter as a medium for computation, there are also approaches to form shapes out of DNA. One such approach is *DNA origami* (see, e.g., [Rot06]) in which DNA is folded to form nanoscale shapes and patters. A more extensive overview of DNA computing can be found in the survey of Daley and Kari [DK02] and in the chapter on DNA computing in [GDT14].

The basic idea behind *self-assembly* is that a collection of simple entities autonomously assembles into a more complex structure without external intervention.[2] This process is based on local interactions between the entities that typically follow simple rules. The most popular model for *tile-based* self-assembly, which uses tiles as its basic entities, is the *Tile Assembly Model* (or *TAM*) introduced by Erik Winfree in his seminal PhD thesis [Win98]. This model was inspired by the groundbreaking results on DNA computing at the time and comes in two variants: The *abstract Tile Assembly Model* (or *aTAM*) is a high-level model that ignores the possibility of errors and provides a framework for theoretical research on tile-based self-assembly. In contrast, the *kinetic Tile Assembly Model* (or *kTAM*) also takes several technical aspects of DNA-based self-assembly into account. The tiles in the aTAM are two-dimensional unit squares that cannot be rotated or flipped. Each edge of a tile has a *color*, and each color has an integer *strength*. The self-assembly

[2] This paragraph is based on a survey by Patitz, see [Pat14].

process starts with an initial assembly, which usually consists of a single tile, and progresses by sequentially attaching additional tiles in a non-deterministic fashion. A tile can be attached to the assembly at a certain position if its edge colors match the edge colors of the adjacent tiles and the sum of the strengths of the matching edge colors reaches a certain threshold called the *temperature* of the system. The aTAM naturally lends itself for the construction of shapes and patterns, but it can also be used to perform universal computations. There is an abundance of results on both of these subjects in the literature. We refer to the excellent surveys on tile-based self-assembly [Dot12; Woo13; Pat14] for an extensive overview of the many models and results in this area.

In the area of distributed computing, *population protocols* [Ang+06] have been used to explore the computational power of simple mobile agents. This model allows a set of finite-state agents to perform a computation by engaging in a sequence of pairwise interactions. Two interacting agents update their state by jointly applying a transition function. The agents in this model are passive in that they have no control over the interaction sequence. There are several publications that investigate the leader election problem under different variants of population protocols (see, e.g, [DS15; AG15; Das+17] and the references therein). Recently, population protocols have also been used for the construction of network topologies [MS16] and geometric shapes [Mic15], which is made possible by allowing two interacting agents to form or release a bond.

In contrast to the passive systems described above, the computational entities in *active* systems are more powerful in that they can both perform computations and control their movements. Notable examples of active systems are robotic and natural swarms, self-reconfigurable robotic systems, and the nubot model.

Models for *swarm robotics* and *natural swarms* usually assume that there is a collection of autonomous agents that can freely move in a given space and that have limited sensing and communication capabilities. The work in this area follows a variety of goals such as gathering (e.g., [AGM13; Cie+12]), spreading (e.g., [Hsi+02; CP08]), and mimicking the collective behavior of natural systems to understand the global effects of local behavior (e.g., [Cha09; Bha+13]). There are also several results on shape formation (e.g., [Flo+08; AR10; Das+10; RCN14]). Of special note from a practical point of view is the work of Rubenstein et al. [RCN14], which demonstrates that programmable self-assembly of complex two-dimensional shapes with hundreds or thousands of simple robots called *kilobots* is possible in practice today. Surveys of recent results in swarm robotics can be found in [McL08; Ker13].

A *self-reconfigurable robot* is able to deliberately change its shape by rearranging its parts in order to adapt to new circumstances, perform new tasks, or recover from damage. *Modular* self-reconfigurable robots are formed from modules (i.e., robotic building blocks), of which there are often only few different types.[3] A particularly relevant variant of self-reconfigurable robots are *metamorphic robotic systems* [Chi94], which only use a single type of module and aim at solving problems like shape formation and enveloping objects. In this regard, these systems bear some resemblance to our amoebot model. Another relevant line of work is the *Claytronics* project (see, e.g., [GCM05]), which explicitly combines the idea of modular self-reconfigurable robots with the concept of programmable matter by envisioning programmable matter consisting of myriads of simple robotic modules called *Claytronics atoms* or *catoms*. While the research on modular self-reconfigurable robots has a strong focus on the development of hardware, there has also been a number of algorithmic advances (e.g., [But+04; WWA04]). For a survey of modular self-reconfigurable robotic systems see [Yim+07].

The *nubot* model by Woods et al. [Woo+13] aims to provide a theoretical framework for studying the complexity of self-assembled structures with *active* molecular components. Thereby, it departs from previous models for self-assembly that were based exclusively on *passive* components such as the Tile Assembly Model described above. The nubot model considers a two-dimensional grid of *monomers*. Each monomer has an internal state. Adjacent monomers can interact in a pairwise fashion according to a set of rules. Two interacting monomers can jointly update their states. They can also be subject to a movement rule, which is locally applied but causes *global* movement in the system. Finally, the model allows monomers to appear and disappear. The latter two features of the nubot model are specifically tailored to the biological setting considered by the authors. Naturally, one of the problems investigated under the nubot model is the construction of two-dimensional shapes (e.g., [Woo+13; Che+14]). Furthermore, the possibility of simulating classical computational models such as Boolean circuits and Turing machines has been considered (e.g., [CXW15]).

Finally, we turn to two systems that do not readily fit into the broad categories of passive and active systems, namely cellular automata and slime molds. The concept of *cellular automata* is already more than half a century old and goes back to John von Neumann and Stanislaw Ulam.[4] A cellular automaton consists of a lattice of cells. Each cell has one of finitely-many states. The cells update their state in synchronous steps according to a local transition function that takes into account a cell's current state and the current state of its neighbors. There is a rich body of work using a multitude of different variants of cellular automata, which we cannot do justice in these few lines. The problem of forming shapes or patterns has been studied extensively in this area, and many of the models and systems described above are inherently

[3] The beginning of this paragraph is based on [Yim+07].

[4] This paragraph is based on [Sch08].

related to this model of computation. For an introduction to the subject of cellular automata see, e.g., [Sch08].

Our last (and maybe most peculiar) example of a system related to programmable matter is the slime mold *physarum polycephalum*. Experiments indicate that this amoeba-like organism is capable of solving shortest path problems in a maze [NYT00]. Thereby, this organism can be interpreted as an example of simple computational matter that occurs in nature. Bonifaci et al. [BMV12] investigated the behavior of physarum polycephalum from a theoretical point of view and showed that under an abstract mathematical model the shape of the organism indeed converges to a shortest path. The behavior of physarum polycephalum already served as an inspiration for distributed algorithms in the domain of wireless sensor networks [Li+10]. In the future it might inspire algorithms for programmable matter.

5.3 Problem Statement

We consider the classic problem of *leader election*. We define an algorithm to solve the leader election problem if for any given particle system, eventually a single particle *irreversibly* declares itself the *leader* (e.g., by setting a dedicated bit in its memory) and no other particle ever declares itself to be the leader. We define the running time of a leader election algorithm to be the number of rounds until a leader is established. Note that we do not require the algorithm to terminate for particles other than the leader. We assume the particle system to be well-initialized in that the memory of every particle is empty. Furthermore, we assume for simplicity that all particles are contracted.

We investigate several variants of the leader election problem in Section 5.6. Among these variants, we consider the case that the algorithm has to terminate for all particles and the case that the particle system contains expanded particles.

5.4 Leader Election Algorithm

Before we describe the leader election algorithm in detail, we give a short high-level overview. The algorithm consists of six *phases*. The 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. Furthermore, a particle can be involved in the execution of multiple phases at the same time. The first phase is the *boundary setup* phase (see Section 5.4.1). In this phase, each particle locally checks whether it is part of a *boundary* of the particle system, see Figure 5.2. Only the particles on a boundary participate in the leader election. Particles occupying a common boundary organize themselves into a directed cycle. The remaining phases operate on each boundary independently. In the *segment setup* phase (see Section 5.4.2), a boundary is subdivided into *segments*: Each particle flips a fair coin. Particles that flip heads become

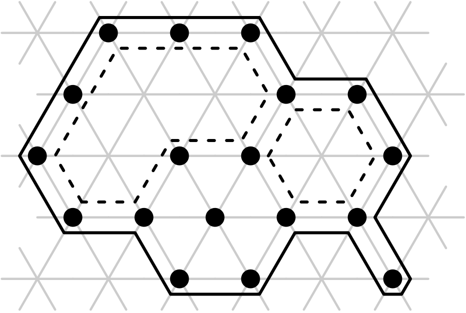

Figure 5.2: Boundaries of a particle system. The solid line represents the unique outer boundary and the dashed lines represent the inner boundaries of the particle system.

candidates and compete for leadership, whereas particles that flip tails become *non-candidates* and assist the candidates in their competition. A segment consists of a candidate and all subsequent non-candidates along the boundary up to the next candidate. The *identifier setup* phase (see Section 5.4.3) assigns a random identifier to each candidate. The identifier of a candidate is stored in a distributed manner among the particles in its segment. In the *identifier comparison* phase (see Section 5.4.4), the candidates compete for leadership by comparing their identifiers using a token passing scheme. When a candidate sees an identifier that is higher than its own identifier, it revokes its candidacy. Whenever a candidate sees its own identifier, the *solitude verification* phase (see Section 5.4.5) is triggered. In this phase, a candidate checks whether it is the last remaining candidate on the boundary. When a candidate determines that it is indeed the last remaining candidate on its boundary, it initiates the *boundary identification* phase (see Section 5.4.6) to determine whether it occupies the unique *outer boundary* of the particle system, see Figure 5.2. If so, it becomes the leader. Otherwise, it revokes its candidacy. The following detailed description of the algorithm is divided into six sections—one for each phase of the algorithm.

5.4.1 Boundary Setup

The boundary setup phase organizes the particle system into a set of *boundaries*, see Figure 5.2. Let A be the set of nodes in G_{ET} that are occupied by particles. According to our assumptions, the subgraph $G_{\mathrm{ET}}|_A$ of G_{ET} induced by A is connected. Let $G_{\mathrm{ET}} = (V, E)$. Consider the graph $G_{\mathrm{ET}}|_{V \setminus A}$ induced by the unoccupied nodes in G_{ET}. We call a connected component R of $G_{\mathrm{ET}}|_{V \setminus A}$ an *empty region*. Let $N(R)$ be the neighborhood of an empty region R in G_{ET}, that is

$$N(R) = \{\, u \in V \mid u \notin R \text{ and } \exists v \in R : \{u, v\} \in E \,\}.$$

Note that, by definition, all nodes in $N(R)$ are occupied by particles. We refer to $N(R)$ as the *boundary* of the particle system corresponding to R. Since $G_{\mathrm{ET}}|_A$ is a finite graph, exactly one empty region has infinite size while the remaining empty regions have finite size. We define the boundary corresponding to the infinite empty region to be the unique *outer boundary* and refer to a boundary that corresponds to a finite empty region as an *inner boundary.*

For each boundary of the particle system, we organize the particles occupying that boundary into a directed cycle. Upon its first activation, a particle instantly determines its place in these cycles using only local information. Figure 5.3 shows all possible neighborhoods of a particle (up to rotation) and the corresponding results of the boundary setup phase. To produce the depicted results, a particle p proceeds as follows. First, p checks for the two special cases shown in the top-most part of the figure. If p has no neighbors, it must be the only particle in the particle system since the particle system is connected. Thus, it immediately declares itself the leader and terminates. If all neighboring nodes of p are occupied, p is not part of any boundary and terminates without participating in the leader election process any further.

If these special cases do not apply then p has at least one occupied node and one unoccupied node in its neighborhood. Interpret the neighborhood of p as a directed ring of six nodes that is oriented clockwise around p. Consider a maximal sequences of unoccupied nodes $(v_1, v_2, \ldots, v_k)$ in this ring. Such a sequence is part of some empty region and, hence, corresponds to a boundary that includes p. Let v_0 be the node before v_1 and let v_{k+1} be the node after v_k in the ring. Note that we might have $v_0 = v_{k+1}$. By definition, v_0 and v_{k+1} are occupied. Particle p implicitly arranges itself as part of a directed cycle spanning the aforementioned boundary by considering the particle occupying v_0 to be its *predecessor* and the particle occupying v_{k+1} to be its *successor* on that boundary. It does this individually for each maximal sequence of unoccupied nodes in its neighborhood to produce the results shown in Figure 5.3.

The remaining phases of the leader election algorithm operate exclusively on boundaries. Furthermore, they are executed on each boundary independently, i.e., the executions within different boundaries do not interact with each other. Note that a particle can have up to three maximal sequences of unoccupied nodes in its neighborhood, see Figure 5.3. As a consequence, a particle can be part of up to three distinct boundaries. However, a particle cannot locally decide whether two distinct sequences of unoccupied nodes belong to two distinct empty regions or to the same empty region. To guarantee that the executions on distinct boundaries are isolated, we let the particles treat each sequence of unoccupied nodes as a distinct empty region. For each such sequence, a particle executes an independent instance of the same algorithm that encompasses the remaining five phases of the leader election algorithm. We say a particle acts as a number of distinct *agents*—one for each maximal sequence of unoccupied

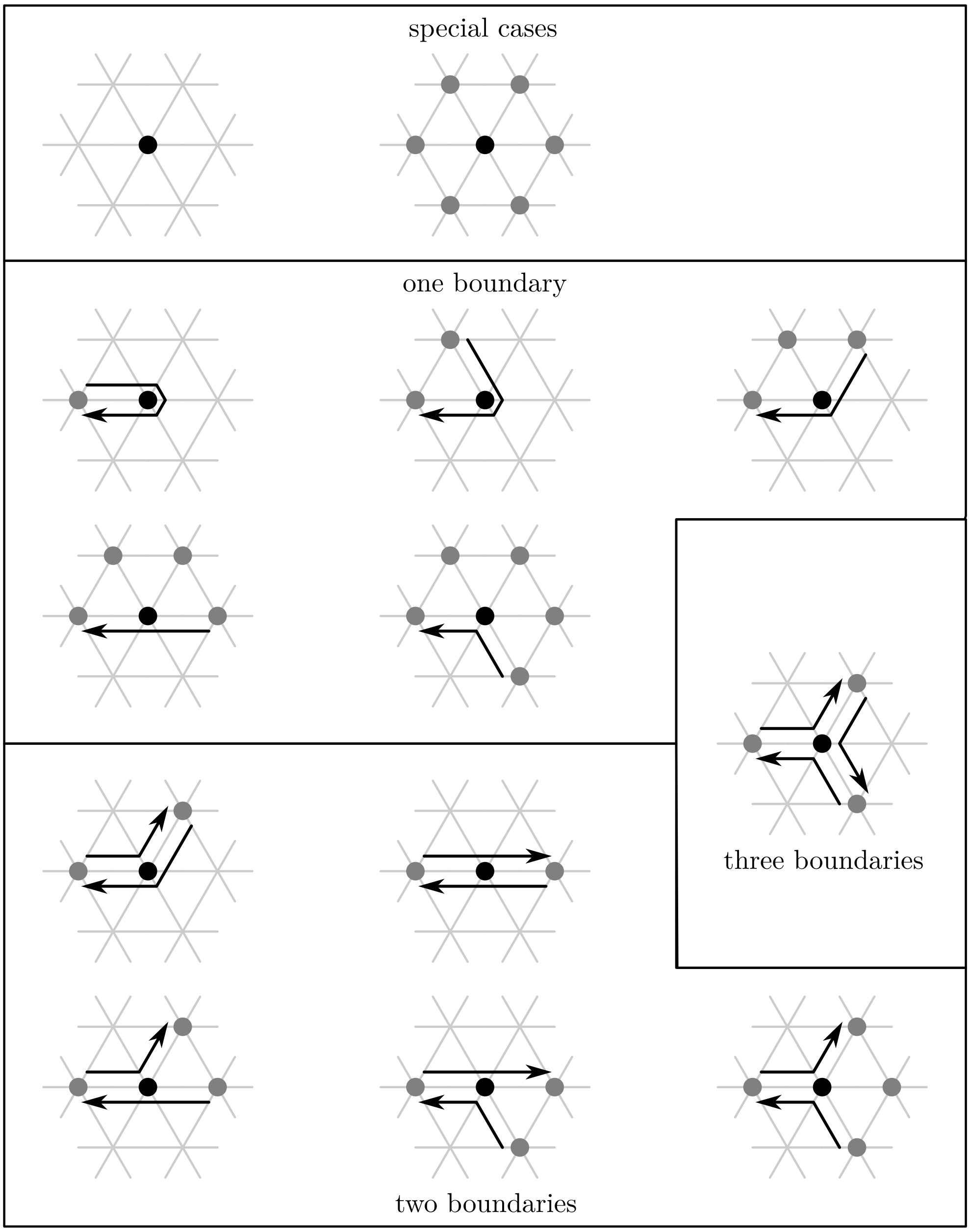

Figure 5.3: Results of the boundary setup phase (up to rotation). In the special cases shown at the top of the figure the particle is not part of any boundary. The rest of the figure is organized according to the number of boundaries a particle is part of from its local perspective. For each boundary, the figure shows an arrow going from the predecessor to the successor of the particle on that boundary.

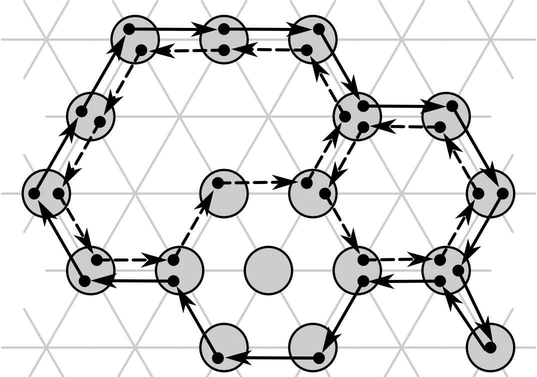

Figure 5.4: Organization of agents into directed cycles. Note that the particle system shown in this figure is the same as the one shown in Figure 5.2. Particles are depicted as gray circles and the agents of a particle are depicted as black dots inside of the corresponding circle. After the boundary setup phase, the agents form disjoint cycles that span the boundaries of the particle system. The solid arrows represent the unique outer boundary and the dashed arrows represent the two inner boundaries.

nodes in its neighborhood. Whenever a particle is activated, it sequentially executes the independent instances of the algorithm for each of its agents in an arbitrary order, i.e., whenever a particle is activated, also its agents are activated. Each agent is assigned the predecessor and successor that was determined by the particle for the corresponding sequence of unoccupied nodes. This effectively connects the set of all agents into disjoint cycles spanning the boundaries of the particle system, see Figure 5.4. We refer to the predecessor and successor of an agent a as a.pred and a.succ, respectively. Note that as a consequence of this approach, a particle can occur up to three times on the same boundary as different agents. While we can ignore this property for most of the remaining phases, we have to carefully handle it in the solitude verification phase described in Section 5.4.5.

5.4.2 Segment Setup

This phase and all subsequent phases operate on each boundary independently. Therefore, we only consider a single boundary for the remainder of the algorithm description. The goal of the segment setup phase is to divide the boundary into disjoint *segments*. Each agent flips a fair coin. The agents for which the coin flip comes up heads become *candidates* and the agents for which the coin flip comes up tails become *non-candidates*. In the following phases, candidates compete for leadership while non-candidates assist the candidates in their competition. A segment is a maximal sequence of agents $(a_1, a_2, \ldots, a_k)$ such that a_1 is a candidate, a_i is a non-candidate for $i > 1$, and $a_i = a_{i-1}.\mathrm{succ}$ for $i > 1$. Note that the maximality condition implies that the successor of a_k is a candidate. We refer to the segment starting at a candidate c as $c.\mathrm{seg}$ and call it the segment of c. In the following phases, each candidate uses its segment as a distributed memory.

5.4.3 Identifier Setup

After the segments have been set up, each candidate generates a random *identifier* by assigning a random digit to each agent in its segment. The candidates use these identifiers in the next phase to engage in a competition with the goal of eliminating all but one candidate on the boundary. Note that the term identifier is slightly misleading in that two distinct candidates can have the same identifier (in fact, this will happen quite frequently). Nevertheless, we hope that the reader agrees that the way these values are used makes this term an appropriate choice.

To generate a random identifier, a candidate c sends a *token* along its segment in the direction of the cycle spanning the boundary. A token is simply a constant-size piece of information that is passed from one particle (or agent) to the next by writing it to the memory of a neighboring particle. Throughout the leader election algorithm, a particle holds at most a constant number of tokens at all times, so the constant-size memory of the particles is sufficient to implement a token-passing mechanism. While the token traverses the segment, it assigns a value chosen uniformly at random from $[0, r-1]$ to each visited agent where r is a constant that is fixed in the analysis. The identifier generated in this way is a number with radix r consisting of $|c.\mathrm{seg}|$ digits where c holds the most significant digit and the last agent of $c.\mathrm{seg}$ holds the least significant digit. We refer to the identifier of a candidate c as $c.\mathrm{id}$. The competition in the next phase of the algorithm is based on comparing identifiers. When comparing identifiers of different lengths, we define the shorter identifier to be lower than the longer identifier.

After generating its random identifier, each candidate creates a copy of its identifier that is stored in *reversed digit order* in its segment. This step is required as a preparation for the next phase. To achieve this, we use a single token that moves back and forth along the segment and copies one digit at a time. More specifically, we reuse the token described above that generated the random identifier. Once this token reaches the end of the segment, it starts copying the identifier: It reads the digit of the last agent of the segment and moves to the beginning of the segment. There, it stores a copy of that digit in the candidate c. It then reads the digit of c and moves back to the end of the segment where it stores a copy of that digit in the last agent of the segment. It proceeds in a similar way with the second and the second to last agent and so on until the identifier is completely copied. Afterwards, the token moves back to c to inform the candidate that the identifier setup is complete.

Note that for ease of presentation we deliberately opted for a simple algorithm over a fast algorithm for creating a reversed copy of the identifier. As we will show in Section 5.5.2, the running time of this simple algorithm is dominated by the running time of the next phase so that the overall asymptotic running time of the leader election algorithm does not suffer.

5.4.4 Identifier Comparison

During the identifier comparison phase the agents use their identifiers to compete with each other. Each candidate compares its own identifier with the identifier of every other candidate on the boundary. A candidate whose identifier is not the highest identifier withdraws its candidacy, whereas a candidate with the highest identifier eventually progresses to the solitude verification phase, which is described in the next section. To achieve the comparison, the non-reversed copies of the identifiers remain stored in their respective segments while the reversed copies move backwards along the boundary as a sequence of tokens. More specifically, a *digit token* is created for each digit of a reversed identifier. A digit token created by the last agent of a segment is marked as a *delimiter token*. Once created, the digit tokens traverse the boundary against the direction of the cycle spanning it. Each agent is allowed to hold at most two tokens at a time, which gives the tokens some space to move along the boundary. The tokens are not allowed to overtake each other, so whenever an agent stores two tokens, it keeps track of the order they were received in and forwards them accordingly. An agent forwards at most one token per activation. Furthermore, an agent can receive a token only after it created its own digit token. We define the *token sequence* of a candidate c as the sequence of digit tokens created by the agents in c.seg. Note that, according to the rules for forwarding tokens, the token sequences remain separated and the tokens within a token sequence maintain their relative order along the boundary.

Whenever a token sequence traverses a segment c.seg of a candidate c, the agents in c.seg cooperate with the tokens of the token sequence to compare the identifier c.id with the identifier stored in the token sequence. This comparison has three possible outcomes: (i) the token sequence is longer than c.seg or the lengths are equal and the token sequence stores an identifier that is strictly greater than c.id, (ii) the token sequence is shorter than c.seg or the lengths are equal and the token sequence stores an identifier that is strictly smaller than c.id, or (iii) the lengths are equal and the identifiers are equal. In the first case, c does not have the highest identifier and withdraws its candidacy. In the second case, c might be a candidate with the highest identifier and therefore remains a candidate. Finally, in the third case c initiates the solitude verification phase, which is then executed in parallel to the identifier comparison phase. Note that solitude verification might be triggered quite frequently, especially for candidates with short segments. We describe how to deal with this fact in the next section.

We now describe the token passing scheme for the identifier comparison on the basis of the example of a candidate c and the next candidate c' along the boundary at the beginning of the identifier comparison phase. Both agents and tokens can be either *active* or *inactive*. Agents are initially active while tokens are initially inactive. When an active agent receives an active token, both become inactive and we say the agent and the token *match*. Since the tokens in the token sequence of c are initially inactive, they are forwarded by the agents of c.seg without matching. Whenever a token is forwarded by a candidate into a new segment, the token becomes active. Therefore, the tokens in the token sequence of c' are active when they enter c.seg. When an agent matches with a token, it compares its digit of the *non-reversed* identifier with the digit stored in the token and keeps the result of the comparison for future reference. Note that since the digits of c'.id are stored in *reversed order* in the token sequence of c', the agent holding the least significant digit of c.id matches with the token holding the least significant digit of c'.id, the agent holding the second to least significant digit of c.id matches with the token holding the second to least significant digit of c'.id, and so on.

The lengths of the identifiers are compared as follows. Recall that the digit token generated by the last agent of a segment is marked as a delimiter token. If the delimiter token of c' matches with c, the token sequence of c' has the same length as c.seg. If the delimiter token of c' matches with another agent of c.seg (and is therefore already inactive when it reaches c), the token sequence of c' is shorter than c.seg. Finally, if a non-delimiter token of c' matches with c, the token sequence of c' is longer than c.seg. Consequently, c can distinguish these three cases once it receives the delimiter token of c'.

If the token sequence of c' has the same length as c.seg, c has to compare its identifier with the identifier stored in the token sequence of c'. As described above, the digits of the respective identifiers are compared in the correct order and the results of the comparisons are stored in a distributed way by the agents of c.seg. When the delimiter token of c' traverses c.seg, it keeps track of the comparison result of the most significant digit for which the identifiers differ. It can do so because during its traversal it sees the consecutive digit-wise comparisons going from the least significant digit to the most significant digit. Once c receives the delimiter token of c', c can use the information stored within the token to decide whether the identifiers are equal or, if not, which identifier is greater.

It remains to describe how the agents and tokens are prepared for subsequent comparisons. Specifically, we have to define when inactive agents and tokens become active again and when the comparison results stored in the agents are deleted. As described above, an inactive token becomes active when it is forwarded by a candidate into a new segment. The remaining tasks are the responsibility of the delimiter tokens: When an agent receives a delimiter token, it executes the computations described above and then deletes its comparison result and becomes active again.

Finally, note that a candidate that withdrew its candidacy still takes part in the identifier comparison phase to a certain extent: Since the agents in its segment still match with incoming tokens, the candidate has to keep activating these tokens when it forwards them to the segment of the preceding candidate. However, candidates that withdrew their candidacy will never progress to solitude verification and are treated as non-candidates in the solitude verification phase.

5.4.5 Solitude Verification

The goal of the solitude verification phase is for a candidate c to check whether it is the last remaining candidate on its boundary. Solitude verification is triggered during the identifier comparison phase whenever a candidate detects equality between its own identifier and the identifier of a token sequence that traversed its segment. Note that such a token sequence can either be the token sequence created by c itself or the token sequence created by some other candidate that chose the same random identifier. Once the solitude verification phase has been started, it runs in parallel to the identifier comparison phase and does not interfere with it.

A candidate c can check whether it is the last remaining candidate by determining whether the next candidate in the direction of the cycle is again c or some other candidate. To achieve this, the solitude verification phase has to span not only c.seg but also all subsequent segments of former candidates that already withdrew their candidacy during the identifier comparison phase. We refer to the union of these segments as the *extended segment* of c. The basic idea of the algorithm is the following: We treat the edges that connect the agents on the boundary as vectors in the two-dimensional Euclidean plane. For c to be the last remaining candidate, the candidate at the end of the extended segment of c must occupy the same node as c. This is the case if and only if the vectors corresponding to the edges along the extended segment of c together with the vector going from the last agent of the extended segment to the next candidate sum up to the zero vector. To determine whether this is the case, c locally defines a two-dimensional coordinate system (e.g., the coordinate system depicted in Figure 5.6) and uses a token passing scheme to determine whether the x- and y-coordinates of the vectors respectively sum up to zero. We only describe the token passing scheme for the x-coordinate. The token passing scheme for the y-coordinate works analogously. The two token passing schemes are executed in parallel.

First, c sends an *activation token* along its extended segment to the next candidate. Whenever the token moves *right* (i.e., in the *positive* direction of the local x-axis defined by c), it creates a *positive token* that is sent back along the boundary towards c. Whenever the token moves *left* (i.e., in the *negative* direction of the local x-axis), it creates a *negative token* that is also sent back towards c. The positive and negative tokens move independently of each other. However, a token of either type cannot overtake another token of the same type. Each agent can hold at most two tokens of each type. The tokens do not move beyond c.

Consider a positive token t. Eventually, t reaches an agent a such that the agents from c to a.pred all hold two positive tokens. At this point, t cannot move any closer to c and we say t has *settled*. Each token holds a bit that specifies whether it has settled. The bit is initially false. It is set to true if the token reaches c or if the token is held by an agent a such that a.pred already holds two settled positive tokens. The negative tokens use the same mechanism to detect whether they are settled. Observe that once all tokens of a specific type have settled, they form a consecutive sequence whose length corresponds to the number of tokens.

After the activation token completely traversed the extended segment of c and reached the next candidate, it moves back towards c while staying behind the positive and negative tokens. When it first encounters an agent a in front of it that holds a settled token, it moves to a and waits until all tokens at a are either forwarded or have settled. At this point, the observation from the last paragraph implies the following property: The total number of positive tokens equals the total number of negative tokens if and only if the number of settled positive tokens at a equals the number of settled negative tokens at a. Therefore, the activation token can locally decide whether the x-coordinate of the aforementioned sum of vectors is zero or not. The activation token then moves back to c and reports the result. On its way to c it deletes all positive and negative tokens it encounters. Once c has received the results for both the x- and the y-coordinate, it knows whether the vectors induced by its extended segment sum up to the zero vector.

Using the given token passing scheme, a candidate c can decide whether the next candidate along the boundary occupies the same node in G_{ET} as c. However, this is not sufficient to decide whether c is the last remaining candidate on the boundary. As mentioned in Section 5.4.1, a particle can occur up to three times as different agents on the same boundary. Therefore, there can be distinct agents on the same boundary that occupy the same node of G_{ET}. If an extended segment reaches from one of these agents to another, the vectors induced by the extended segment sum up to the zero vector even though there are at least two agents left on the boundary. To handle this case, each particle assigns a locally unique *agent identifier* from $\{1, 2, 3\}$ to each of its agents in an arbitrary way. When the activation token reaches the end of the extended segment, it reads the agent identifier of the candidate at the end of the extended segment and carries this information back to c. It is not hard to see that c is the last remaining candidate on the boundary if and only if the vectors sum up to the zero vector and the agent identifier stored in the activation token equals the agent identifier of c.

Finally, we have to address the interaction between the solitude verification phase and the identifier comparison phase. As already mentioned in the previous section, the solitude verification phase can be triggered quite frequently. Therefore, it can happen that the solitude verification phase is triggered for a candidate c while c is still performing a previously triggered execution of the solitude verification phase. In this case, c simply continues with the already ongoing execution and ignores the request for another execution. Furthermore, c might be eliminated by the identifier comparison phase while it is performing solitude verification. In this case, c waits for the ongoing solitude verification to finish and only then withdraws its candidacy.

5.4.6 Boundary Identification

Once a candidate c determines that it is the only remaining candidate on its boundary, it initiates the boundary identification phase to check whether it lies on the unique outer boundary of the particle system or on an inner boundary. If it lies on the outer boundary, the particle responsible for c declares itself the leader. Otherwise, c revokes its candidacy. To achieve this, we make use of the observation that the outer boundary is oriented clockwise while an inner boundary is oriented counter-clockwise, see Figure 5.4. This observation is a direct consequence of the way the predecessor and successor of an agent are defined in Section 5.4.1.

The candidate can distinguish between the two cases using the following simple token passing scheme: It sends a token along the boundary that sums up the angles of the turns it takes according to Figure 5.5. When the token returns to the candidate, the absolute value $|\alpha|$ of the value α stored in the token represents the external angle of the polygon induced by the boundary. It is well known that the external angle of a polygon in the Euclidean plane is $|\alpha| = 360°$. Note that the rules given in Figure 5.5 take into account the orientation (i.e., clockwise versus counter-clockwise) of the traversal. Since the outer boundary is oriented clockwise and an inner boundary is oriented counter-clockwise, we have $\alpha = 360°$ for the outer boundary and $\alpha = -360°$ for an inner boundary. The token can encode α as an integer k such that $\alpha = k \cdot 60°$. To distinguish the two possible final values of k it is sufficient to store k modulo 5 so that we have $k = 1$ for the outer boundary and $k = 4$ for an inner boundary. Therefore, the token only needs three bits of memory.

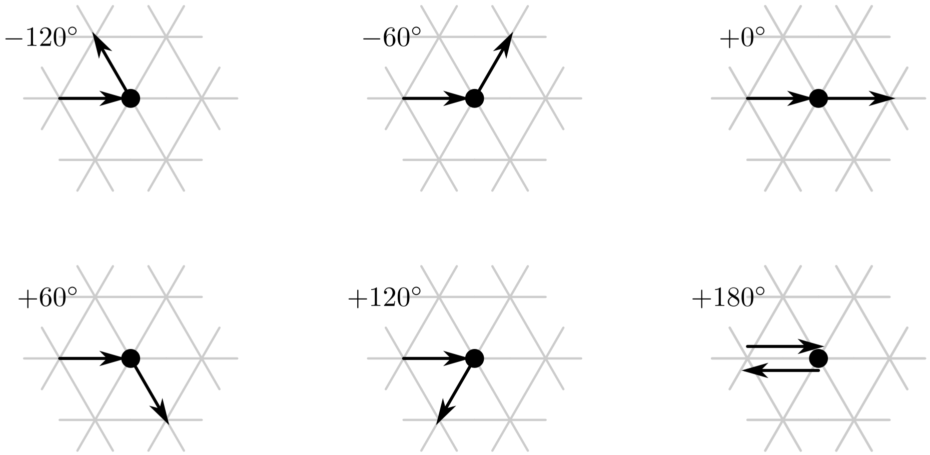

Figure 5.5: Angles along a traversal of a boundary. For the depicted agent a the incoming arrow represents the vector from a.pred to a and the the outgoing arrow represents the vector from a to a.succ. Note that only the angle between these vectors is relevant; the absolute global direction of the vectors cannot be detected by the agent a since the particles do not possess a global compass.

5.5 Analysis

We now turn to the analysis of the leader election algorithm. We first show the correctness of the algorithm in Section 5.5.1 and then analyze its running time in Section 5.5.2.

5.5.1 Correctness

To show the correctness of the algorithm we have to prove that eventually a single particle irreversibly declares itself to be the leader of the particle system and no other particle ever declares itself to be the leader. Note that an agent on an inner boundary can never cause its particle to become the leader: Even if the algorithm reaches the point at which there is exactly one candidate c on some inner boundary, c will withdraw its candidacy in the boundary identification phase. Therefore, we can focus exclusively on the behavior of the algorithm on the unique outer boundary.

Let n be the number of particles in the particle system. Recall that we say an event occurs *with high probability* (abbreviated as *w.h.p.*) if it occurs with probability at least $1 - n^{-c}$ for a given constant $c \geq 1$. We first show a series of lemmas to establish that, with high probability, there is a unique candidate that has an identifier that is strictly greater than the identifier of every other candidate. Define L to be the length of the outer boundary, i.e., the number of agents in the cycle spanning the boundary. We have the following bound on L.

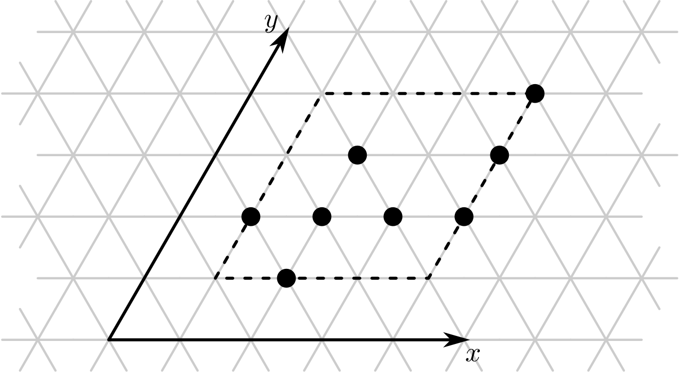

Figure 5.6: Coordinate system and bounding box (dashed) of a particle system.

Lemma 5.1. *$L \geq \sqrt{n}$.*

Proof. We define a coordinate system by picking an arbitrary node of G_{ET} as the origin and orienting the x- and y-axis as depicted in Figure 5.6. Consider the *axis-aligned bounding box* of the particle system, see Figure 5.6. We define the length of a side of this bounding box as the number of nodes it spans. Since the particle system is connected and contains n particles, one of the sides of the bounding box has to be of length at least $\sqrt{n}$. Assume for now that the top and bottom side of the bounding box have the longest length. Define a *column* of the particle system to be a maximal set of particles with the same x-coordinate. The particle with the greatest y-coordinate in each column is part of the outer boundary. Since the number of columns equals the length of the top side, there are at least $\sqrt{n}$ agents on the outer boundary. The case that the left and right side of the bounding box have the longest length is analogous. □

The next lemma gives us a lower bound on the length of the longest segment, which is equal to the number of digits in the longest identifier.

Lemma 5.2. *For n sufficiently large, there is a segment of length at least $\lfloor 0.25 \log n \rfloor$, w.h.p.*

Proof. Pick an arbitrary agent a_1 on the boundary and let $a = (a_1, a_2, \ldots, a_L)$ where $a_i = a_{i-1}.\text{succ}$ for $2 \leq i \leq L$. Define $b = (b_1, b_2, \ldots, b_L)$ where $b_i = 1$ if the coin flip performed by a_i in the identifier setup phase came up heads and $b_i = 0$ otherwise. We first show that b contains a subsequence of length at least $m = \lfloor 0.25 \log n \rfloor$ such that all elements of the subsequence are 0. For this, we divide b into consecutive subsequences of length m. According to Lemma 5.1, we get k subsequences where

$$k = \lfloor L/m \rfloor \geq \lfloor \sqrt{n}/m \rfloor \geq \lfloor 4\sqrt{n}/\log n \rfloor.$$

For k to be well defined we assume $n \geq 16$ so that $m > 0$. We define $b^{(i)} = (b_{(i-1)m+1}, \ldots, b_{im})$ to be the i-th subsequence where $1 \leq i \leq k$. Let E_i be the event that all elements of $b^{(i)}$ are 0. We have

$$\Pr[E_i] = 1/2^m \geq n^{-1/4}.$$

Since the events E_i are independent, we have

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

Note that $\lfloor x \rfloor > x/4$ for any $x \geq 4/3$. Since $4\sqrt{n}/\log n \geq 4/3$ for $n \geq 16$, this implies $\lfloor 4\sqrt{n}/\log n \rfloor > \sqrt{n}/\log n$. Therefore, we have

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

Applying the well-known inequality $(1 - 1/x)^x \leq 1/e$ with $x = n^{1/4}$ yields

$$\Pr\left[\bigcap_{i=1}^{k} \overline{E_i}\right] \leq e^{-n^{1/4}/\log n}.$$

So for n sufficiently large, the entries of one of the subsequences $b^{(i)}$ are all 0, w.h.p. This implies that b contains a subsequence of at least m consecutive entries that are 0.

It remains to show that at least one element of b is 1 and, therefore, there is at least one candidate on the outer boundary. This holds with probability $1 - 2^{-L} \geq 1 - 2^{-\sqrt{n}}$ according to Lemma 5.1. So overall, for n sufficiently large there is a segment of size at least $\lfloor 0.25 \log n \rfloor$, w.h.p. □

We can now show that there is a unique candidate that has an identifier that is strictly greater than the identifier of every other candidate.

Lemma 5.3. *For n sufficiently large, there is a candidate c^* such that $c^*.id > c.id$ for every candidate $c \neq c^*$, w.h.p.*

Proof. Let C be the set of candidates, let M be the set of candidates with maximal segment length, and let c^* be some candidate with the highest identifier. Since in a comparison between two identifiers of different length the shorter identifier is defined to be lower than the longer identifier, we must have $c^* \in M$ and $c^*.\text{id} > c.\text{id}$ for all $c \in C \setminus M$. It remains to show that $c^*.\text{id} > c.\text{id}$ for all $c \in M \setminus \{c^*\}$. This is the case if the identifier of c^* is unique.

By definition, the identifiers of the candidates in M all consist of the same number of digits. By Lemma 5.2, this number of digits is at least $\lfloor 0.25 \log n \rfloor$, w.h.p. Each digit is chosen independently and uniformly at random from the interval $[0, r-1]$ for a constant r of our choice. Therefore, for any candidate $c \in M \setminus \{c^*\}$ the probability that $c.\mathrm{id} = c^*.\mathrm{id}$ is at most $r^{-\lfloor 0.25 \log n \rfloor} \leq r^{1-0.25 \log n}$. Applying the union bound shows that the probability that there exists a candidate $c \in M \setminus \{c^*\}$ such that $c.\mathrm{id} = c^*.\mathrm{id}$ is at most

$$(|M| - 1) \cdot r^{1-0.25 \log n} = (|M| - 1) \cdot r \cdot n^{-0.25 \log r} \leq 3r \cdot n^{1-0.25 \log r},$$

where the second inequality holds because $|M| \leq 3n$ by definition. Choosing r sufficiently large and requiring $n \geq r$ shows the lemma. □

On the basis of Lemma 5.3, we can now prove the correctness of the algorithm.

Theorem 5.4. *The algorithm solves the leader election problem, w.h.p.*

Proof. We have to show that eventually a single particle irreversibly declares itself to be the leader of the particle system and no other particle ever declares itself to be the leader. As described at the beginning of this section, an agent on an inner boundary of the particle system cannot cause its particle to become the leader. So consider the agents on the outer boundary. Once every particle has finished the boundary setup phase, every agent has finished the segment setup phase, and every candidate has finished the identifier setup phase, we have that w.h.p. there is a unique candidate c^* that has the highest identifier on the outer boundary according to Lemma 5.3. Since c^* has the highest identifier, it does not withdraw its candidacy during the identifier comparison phase. In contrast, every other candidate $c \neq c^*$ eventually withdraws its candidacy because the token sequence of c^* eventually traverses $c.\mathrm{seg}$. Therefore, an agent $c \neq c^*$ cannot cause its particle to become the leader. Once c^* is the last remaining candidate on the outer boundary, it eventually triggers the solitude verification phase because the token sequence of c^* eventually traverses $c^*.\mathrm{seg}$ while c^* is not already performing solitude verification. After verifying that it is the last remaining candidate, c^* executes the boundary identification phase and eventually determines that it lies on the outer boundary. It then instructs its particle to declare itself the leader of the particle system. □

5.5.2 Running Time

Recall that the running time of an algorithm for leader election is defined as the number of rounds until a leader is established. Since the given algorithm always establishes a leader on the unique outer boundary, we can limit our attention to that boundary.

The first two phases of the algorithm, namely the boundary setup phase and the segment setup phase, consist entirely of computations based on local neighborhood information. Therefore, these phases can be completed instantly by a particle upon its first activation. Since each particle is activated at least once in every round, the first two phases of the algorithm are complete for all particles after a single round. When an agent becomes a candidate, it initiates the identifier setup phase. We have the following lemma.

Lemma 5.5. *For a segment of length ℓ the identifier setup phase takes $O(\ell^2)$ rounds.*

Proof. In the identifier setup phase a token that is created by the candidate first traverses the segment to establish the random digits of the identifier. Once the token reaches the end of the segment, it creates a copy of the identifier that is stored in reversed order in the segment. For this, the token moves in an alternating fashion back and forth through the segment. Finally, once the copying is complete, the token moves back to the candidate. It is not hard to see that the overall number of steps taken by the token is $O(\ell^2)$.

After the first round, the segment is established and therefore the token can move without being stalled. In each further round, the token can take at least one step along its trajectory. Hence, the identifier setup phase takes $O(\ell^2)$ rounds. □

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

Lemma 5.6. *The length of a segment on the outer boundary is $O(\log n)$, w.h.p.*

Proof. For $n \geq 2$ and any constant real $c \geq 1$, the probability that an agent becomes a candidate with a segment of length at least $c \log n$ is at most $1/2^{c \log n} = n^{-c}$. Since there are n particles in the particle system and each particle corresponds to at most three agents, there are at most $3n$ agents on the outer boundary. Applying the union bound shows that the probability that there is a segment of length at least $c \log n$ is at most $3n^{1-c}$. Therefore, there is no segment of length at least $c \log n$, w.h.p. □

Combining the previous two lemmas gives us the following corollary.

Corollary 5.7. *The identifier setup phase is complete for all candidates on the outer boundary after $O(\log^2 n)$ rounds, w.h.p.*

After the identifiers have been generated, they are compared in the identifier comparison phase. In this phase a set of digit tokens, one for each agent on the boundary, traverses the boundary against the direction of the cycle spanning it. Each agent can store at most two tokens. The tokens are not allowed to overtake each other, so agents maintain the order of the tokens when forwarding them. Note that a token is never delayed unless it is blocked by tokens in front of it. Therefore, an agent a forwards a token whenever a.pred can hold an additional token. Finally, an agent forwards at most one token per activation.

We define the number of *steps* a token took as the number of times it was forwarded from one agent to the next since its creation. Let T be the earliest round such that at the beginning of the round every agent on the outer boundary has created its digit token. We have the following lemma.

Lemma 5.8. *At the beginning of round $T + i$ for $i \geq 0$, each digit token on the outer boundary took at least i steps.*

Proof. We establish a lower bound on the number of steps a token took by comparing the *asynchronous execution* of the token passing scheme with a *synchronous execution* in which the tokens move in lockstep. For the synchronous execution, we assume that each token is initially stored at the agent that created it. We refer to this point in time in the synchronous execution as round 0. The tokens move in lockstep along the boundary so that in every round each agent stores exactly one token. For a token t let $s_i(t)$ be the number of steps t took by the beginning of round i of the synchronous execution. By definition, we have $s_i(t) = i$. Similarly, let $a_i(t)$ be the number of steps t took at the beginning of round $T + i$ of the asynchronous execution. We show by induction on i that $a_i(t) \geq s_i(t)$ for all i and for every token t.

The statement holds for $i = 0$ by definition. Suppose that the statement holds for some $i \geq 0$ and consider a token t. We show $a_{i+1}(t) \geq s_{i+1}(t)$. If $a_i(t) > s_i(t)$ then

$$a_{i+1}(t) \geq a_i(t) \geq s_i(t) + 1 = s_{i+1}(t),$$

and, therefore, the statement holds. So assume $a_i(t) = s_i(t)$. To prove $a_{i+1}(t) \geq s_{i+1}(t)$, we have to show that t is forwarded at least once in round $T+i$ of the asynchronous execution. Recall that in each round of the asynchronous execution, each agent is activated at least once. Therefore, also the agent holding t at the beginning of the round is activated at least once. Since the tokens do not overtake each other, their relative order along the boundary (considering both a token's position on the boundary as well as which token is forwarded first if an agent holds two tokens) is well defined and remains unchanged in both the synchronous and the asynchronous execution. Let t' be the next token from t in the direction of the traversal along the boundary (i.e., against the direction of the cycle spanning the boundary). In the synchronous execution t' started one agent ahead of t. Furthermore, the induction hypothesis together with our assumption that $a_i(t) = s_i(t)$ implies

$$a_i(t') \geq s_i(t') = s_i(t) = a_i(t).$$

Therefore, t' is at least one agent ahead of t in the asynchronous execution. By an analogous argument one can see that the token t'' following t' is at least two agents ahead of t in the asynchronous execution. Since the tokens preserve their order, there are no other tokens in between t, t', and t''. Therefore, the agent to which t should be forwarded in round $T + i$ of the asynchronous execution holds at most one token, namely t'. Furthermore, even if the agent holding t at the beginning of round $T + i$ of the asynchronous execution holds an additional token, t is forwarded first because the order of the tokens is preserved. Therefore, t is indeed forwarded at least once in round $T + i$ of the asynchronous execution. □

Consider the proof of Lemma 5.8 above. On a conceptual level, we showed that the asynchronous execution of a token passing scheme *dominates* a corresponding synchronous execution in terms of a suitable measure of progress. We refer to this kind of argument as a *domination argument.* Domination arguments play a crucial role in this part of the thesis as they are our main tool for analyzing the running time of algorithms in the amoebot model.

The following lemma establishes an upper bound on the running time of the solitude verification phase. Its proof is another example of a domination argument.

Lemma 5.9. *For an extended segment of length ℓ the solitude verification phase takes $O(\ell)$ rounds.*

Proof. The token passing scheme of the solitude verification phase is executed independently for the x- and the y-coordinate. We consider one of these executions and show that it takes $O(\ell)$ rounds. First, the activation token moves through the extended segment and creates positive and negative tokens. Since the activation token moves through the extended segment unhinderedly, this traversal takes $O(\ell)$ rounds. Then, the activation token moves back towards the candidate but now stays behind the tokens it created. The two types of tokens created by the activation token move back towards the candidate independently of each other. However, two tokens of the same type cannot overtake each other. Once all tokens of both types have settled, the activation token can move back to the candidate unhinderedly, which takes another $O(\ell)$ rounds.

It remains to determine the number of rounds until all tokens have settled. We use a domination argument that is very similar to the one used in the proof of Lemma 5.8. Consider a single token type. To simplify our notation, we define round 0 of the asynchronous execution to be the earliest round such that at the beginning of the round the activation token already created all tokens of the considered type. We compare the asynchronous execution with the following synchronous execution: In round 0 of the synchronous execution, each token is stored at the agent that created it. The tokens then move in lockstep towards the candidate. Apart from the movement of the tokens, the synchronous execution works the same way as the asynchronous execution, i.e., each agent can store two tokens and the tokens move as close to the candidate as possible. We assign the numbers $1, 2, \ldots, \ell$ to the agents of the extended segment starting with 1 at the candidate. For a token t let $s_i(t)$ be the number assigned to the agent that holds t at the beginning of round i of the synchronous execution. Similarly, let $a_i(t)$ be the number of the agent that holds t at the beginning of round i of the asynchronous execution. We show by induction on i that $a_i(t) \leq s_i(t)$ for all i and for every token t.

The statement holds for $i = 0$ by definition. Suppose that the statement holds for some round $i \geq 0$ and consider a token t. We show $a_{i+1}(t) \leq s_{i+1}(t)$. If $a_i(t) < s_i(t)$ then

$$a_{i+1}(t) \leq a_i(t) \leq s_i(t) - 1 \leq s_{i+1}(t),$$

and, therefore, the statement holds. So assume $a_i(t) = s_i(t)$. We need two observations: First, since the tokens do not overtake each other, their order along the extended segment is well defined and remains unchanged in both the synchronous and the asynchronous execution. Second, when an agent holds two tokens in the synchronous execution, both tokens must have settled.

Let t' be the next token from t towards the candidate. If there is no such token, the statement holds since t can move unhinderedly in the asynchronous execution. Otherwise, we have $s_i(t') \leq s_i(t)$ by our first observation. We distinguish three cases and show that in each case $a_{i+1}(t) \leq s_{i+1}(t)$.

1. If $s_i(t') = s_i(t)$ then, by our second observation, both t and t' have settled. Therefore, t is never forwarded again, neither in the synchronous execution nor in the asynchronous execution. This implies

$$a_{i+1}(t) = a_i(t) = s_i(t) = s_{i+1}(t).$$

2. If $s_i(t') \leq s_i(t) - 2$ then the agent ahead of t holds no token in the synchronous execution because there is no token between t and t' by our first observation. By our assumption that $a_i(t) = s_i(t)$ together with the induction hypothesis, the agent ahead of t also holds no token in the asynchronous execution. Therefore, t is forwarded at least once in the asynchronous execution and we have

$$a_{i+1}(t) \leq a_i(t) - 1 = s_i(t) - 1 = s_{i+1}(t).$$

3. If $s_i(t') = s_i(t) - 1$ then we have to distinguish two subcases. If the agent $s_i(t')$ holds two tokens in the synchronous execution then t' has settled and hence also t has settled. Therefore, the statement holds by the same argument as in the first case above.

So suppose that $s_i(t')$ only holds t' in the synchronous execution. Let t'' be the next token from t' towards the candidate. If there is no such token, then the agent ahead of t holds at most one token in the asynchronous execution, namely t'. Otherwise, t'' is at least two agents ahead of t in the synchronous execution and, by the induction hypothesis, also in the asynchronous execution. So again, the agent ahead of t holds either no token or only t' in the asynchronous execution. Therefore, t is forwarded at least once in the asynchronous execution and the statement holds by the same argument as in the second case above.

It is not hard to see that the synchronous execution moves all tokens to their final positions in $O(\ell)$ rounds. By the domination argument given above, the asynchronous execution requires $O(\ell)$ rounds to do the same. Once all tokens have settled, it takes $O(\ell)$ additional rounds until each token sets the bit showing that it has settled to true. □

The boundary identification phase is executed only when a candidate determines that it is the last remaining candidate on the boundary. The following lemma provides an upper bound for the running time of this phase. Recall that L is defined as the number of agents on the outer boundary.

Lemma 5.10. *The boundary identification phase on the outer boundary takes $O(L)$ rounds.*

Proof. The token calculating the angle completely traverses the outer boundary. Since the boundary has length L and the token is forwarded at least once in every round, this takes $O(L)$ rounds. □

Finally, we can show the main theorem of this chapter.

Theorem 5.11. *The algorithm solves the leader election problem in $O(L)$ rounds, w.h.p.*

Proof. After the first round the boundaries, the candidates, and the segments have been established. By Corollary 5.7, all candidates on the outer boundary complete the identifier setup phase within $O(\log^2 n)$ rounds. So according to Lemma 5.3, a unique candidate c^* with the highest identifier on the outer boundary has been established at that point, w.h.p. We show that c^* instructs its particle to become the leader after $O(L)$ rounds.

At the beginning of round $T = O(\log^2 n)$ all digit tokens have been created. According to Lemma 5.8, after an additional L rounds, every digit token has completed a pass along the outer boundary. At this point, the token sequence of c^* has traversed the segment of every other candidate and, therefore, every candidate except for c^* either already withdrew its candidacy or is flagged to withdraw its candidacy after it completes its current execution of the solitude verification phase. Since the maximum length of an extended segment of a candidate on the outer boundary is L, the candidates that still execute the solitude verification phase withdraw their candidacy after $O(L)$ additional rounds according to Lemma 5.9. Once c^* is the only remaining candidate on the outer boundary, it has to start its final execution of the solitude verification phase. For this, it might have to finish an already running execution of the solitude verification phase that might fail because it was started too early. In this case, it takes $O(L)$ rounds for the solitude verification phase to fail. After $O(L)$ additional rounds, the token sequence of c^* traverses c^*.seg again and thus triggers the final execution of the solitude verification phase, which takes another $O(L)$ rounds. When c^* determines that it is the last remaining candidate, it executes the boundary identification phase, which takes $O(L)$ rounds according to Lemma 5.10. So after overall $O(L)$ rounds, c^* determines that it is the only remaining candidate on the outer boundary and instructs its particle to become the leader. □

Theorem 5.11 specifies the running time of the leader election algorithm in terms of the number of *agents* on the outer boundary. Let k be the number of *particles* on the outer boundary. Since each particle on the outer boundary corresponds to at most three agents on the outer boundary, we have the following corollary.

Corollary 5.12. *The algorithm solves the leader election problem in $O(k)$ rounds, w.h.p.*

Depending on the application it might be desirable to specify the running time of the algorithm in terms of the number of particles n in the entire particle system. Clearly, the number of particles on the outer boundary is at most n. This implies the following corollary.

Corollary 5.13. *The algorithm solves the leader election problem in $O(n)$ rounds, w.h.p.*

Note that in some cases the bound given in Corollary 5.13 is quite loose compared to the bound given in Corollary 5.12 because the number of particles on the outer boundary of a particle system can be much lower than n. For example, a solid square of n particles (shaped like the bounding box depicted in Figure 5.6) only has $k = O(\sqrt{n})$ particles on its outer boundary. However, in general we can have $k = \Theta(n)$, which can easily be seen in the example of a particle system forming a straight line.

5.6 Variants of the Leader Election Problem

In this section, we consider several variants of the leader election problem. We present three positive results: We show how a leader can be elected when the particle system contains expanded particles, we demonstrate how the leader election algorithm can be extended such that its execution terminates for all particles, and we present a variant of the algorithm that not only elects a leader in $O(L)$ rounds with high probability, but also eventually elects a leader almost surely. The algorithms for these variants can be combined into a single algorithm that satisfies all of the above properties. We close this section with a negative result concerning the generalization of the leader election problem to arbitrary graphs under the absence of geometric information.

5.6.1 Expanded Particles

It is straight-forward to extend the leader election algorithm to allow the particle system to contain expanded particles: An expanded particle p simply simulates two distinct contracted particles, one for each node occupied by p. Whenever p is activated, it simulates the activations of the corresponding contracted particles one after another in an arbitrary order. This effectively reduces the problem of leader election with expanded particles to leader election without expanded particles.

Since every expanded particle is activated at least once in every round, also every simulated particle is activated at least once in every round. Recall that the analysis presented in Section 5.5 holds for any fair activation order. Therefore, the analysis remains valid despite the fact that two simulated particles of an expanded particle are always activated immediately after one another. This implies that the statement of Theorem 5.11 also holds for particle systems containing expanded particles. Since the number of simulated particles is at most twice the number of particles, the statements of Corollary 5.12 and Corollary 5.13 also remain valid.

5.6.2 Termination for All Particles

In the definition of the leader election problem given in Section 5.3, the leader is the only particle for which the algorithm must terminate. Any non-leader particle is allowed to execute the algorithm indefinitely. The algorithm presented in Section 5.4 can in fact experience infinite loops for a subset of the agents in certain situations. For example, consider a particle system with an empty region R of size 1. With constant probability, all six agents on the inner boundary corresponding to R become candidates and get the same one-bit identifier. The identifier comparison phase of the leader election algorithm will never eliminate any of the six candidates in this situation. Therefore, the candidates are stuck in an infinite loop.

Depending on the application, it might be desirable to have a leader election algorithm that is guaranteed to terminate for *all* particles. This can be achieved using the following extension of the algorithm presented in Section 5.4: After the leader has been elected, it broadcasts a *termination message* through the particle system. A particle receiving this message forwards it to each of its neighbors and then terminates its execution of the leader election algorithm. Let A be the set of occupied nodes in G_{ET} and let $G_{\mathrm{ET}}|_A$ be the subgraph of G_{ET} induced by A. Clearly, the running time of the broadcast is linear in the diameter D of the graph $G_{\mathrm{ET}}|_A$. Therefore, after $O(L + D)$ rounds, a leader has emerged and the execution of the leader election algorithm has terminated for all particles in the particle system. We summarize this result in the following theorem.

Theorem 5.14. *The algorithm solves the leader election problem and terminates for all particles in $O(L + D)$ rounds, w.h.p.*

Note that the parameters L and D are in general independent of each other. For example, it is not hard to construct particle systems such that either $L = \Omega(n)$ and $D = O(\sqrt{n})$, or $L = O(\sqrt{n})$ and $D = \Omega(n)$. However, L and D are both clearly in $O(n)$, which implies the following corollary.

Corollary 5.15. *The algorithm solves the leader election problem and terminates for all particles in $O(n)$ rounds, w.h.p.*

5.6.3 Almost-Sure Leader Election

The leader election algorithm presented in Section 5.4 elects a leader with high probability. Accordingly, there is a small probability that the algorithm fails to elect a leader. For example, every agent could decide to become a non-candidate during the segment setup phase so that the algorithm effectively comes to a halt without electing a leader. In this section, we describe how the leader election algorithm can be extended such that it *almost surely* elects a leader eventually (i.e., it elects a leader with probability 1 in the limit as the number of rounds approaches infinity) and it still elects a leader in $O(L)$ rounds with high probability.

The main idea behind the extension is to run a second leader election algorithm in parallel to the algorithm presented in Section 5.4. The second algorithm sets up the boundaries as described in Section 5.4.1. Each agent is initially a candidate, and the candidates alternate between the following two phases: In the first phase, a candidate flips a coin and sends the result along its boundary to both its preceding and its succeeding candidate. A candidate withdraws its candidacy if its coin flip came up tails and the coin flips of both its predecessor and its successor came up heads. Note that this competition locally synchronizes competing candidates.

The second phase of the algorithm corresponds to the solitude verification phase described in Section 5.4.5. Once a candidate determines that it is the last remaining candidate on its boundary, it executes the boundary identification phase described in Section 5.4.6. If the candidate lies on an inner boundary, it withdraws its candidacy. If it lies on the outer boundary, it sends a token along the boundary that stops the execution of the original algorithm in the particles on the outer boundary and, at the same time, checks whether the original algorithm already established a leader. If there already is a leader, the candidate withdraws its candidacy. Otherwise, it declares itself the leader.

We have the following theorem.

Theorem 5.16. *The algorithm elects a leader in $O(L)$ rounds with high probability, and it eventually elects a leader almost surely.*

Proof. Consider the execution of the second algorithm on the outer boundary. As long as there is more than one candidate on the boundary, the first phase of the algorithm reduces the number of candidates with a probability that is lower bounded by a constant. Furthermore, the last remaining candidate on the boundary competes with itself and will therefore never withdraw its candidacy during the first phase of the algorithm. Therefore, it holds almost surely that eventually only a single candidate remains on the outer boundary.

The last remaining candidate on the outer boundary eventually passes the solitude verification phase and the boundary identification phase. It then interacts with the original leader election algorithm by sending a token along the outer boundary. This interaction produces one of three results: First, if the original algorithm already established a leader, the second algorithm does not produce a leader. Second, if the original algorithm established a unique candidate with the highest identifier on the outer boundary but the particle corresponding to that candidate is reached by the aforementioned token before it claims leadership, only the second algorithm establishes a leader. Finally, if the original algorithm fails to establish a unique candidate with the highest identifier on the outer boundary, the second algorithm establishes a leader.

With high probability either the first or the second case holds according to Theorem 5.4. In the first case, the leader is established by the original algorithm in $O(L)$ rounds by Theorem 5.11. In the second case, the second algorithm stops the execution of the original algorithm before it establishes a leader. Since the original algorithm establishes a leader in $O(L)$ rounds, the aforementioned token must have been created in $O(L)$ rounds. The token requires at most L rounds to traverse the boundary and, therefore, the second algorithm establishes a leader in $O(L)$ rounds. Finally, in the third case the second algorithm almost surely establishes a leader eventually. □

5.6.4 General Graphs

In the amoebot model, the particles occupy the nodes of the infinite triangular grid graph G_{ET}. The graph G_{ET} is embedded in the Euclidean plane in a specific way, which provides the particles with geometric information. Specifically, the particles can measure distance because each adjacent pair of nodes is separated by a unit distance, and they can measure angles since each face of G_{ET} is an equilateral triangle and the particles know the rotational order of neighboring nodes through their port labels. The leader election algorithm presented in Section 5.4 explicitly uses this geometric information in the boundary setup phase, the solitude verification phase, and the boundary identification phase.

A natural question is whether there is an algorithm that achieves leader election on any given graph G without geometric information. To formally investigate this question, we assume that we are given a set of contracted particles occupying a connected subset of the nodes of a graph G of constant degree d, and the ports of the particles are labeled arbitrarily with numbers from 1 to d. This variant of the amoebot model was presented in [Der+15b] as the *general amoebot model*.

Let G be a ring of n nodes and let each node be occupied by a contracted particle. To solve the leader election algorithm, eventually a single particle has to irreversibly declare itself the leader and no other particle may ever declare itself to be the leader. Since every node is occupied by a contracted particle, the particles cannot move. Therefore, the particles have to elect a leader using communication only. Note that the problem of electing a leader in the particle system under these conditions is equivalent to the problem of electing a leader in an undirected ring of anonymous nodes where each node is a probabilistic finite automaton.

The problem of electing a leader in a ring of anonymous nodes was already investigated by Itai and Rodeh [IR90]. We say a leader election algorithm fails if it either does not establish a leader or it establishes more than one leader. The results of Itai and Rodeh imply that for every $\rho < 1$, the error probability of an algorithm that solves the leader election problem on rings of arbitrary size is greater than ρ, i.e., the probability that the algorithm fails cannot be bounded away from 1. Therefore, the leader election problem is infeasible on arbitrary graphs without geometric information.

5.7 Outlook

We already investigated several variants of leader election under the amoebot model in the previous section. Still, many intriguing questions in this area remain open. For example, our algorithm elects a leader in $O(L)$ rounds where L is the number of agents on the outer boundary. As we discussed in Section 5.6.2, the parameter L is in general independent from the diameter D of the particle system. It would be interesting to investigate whether there is an algorithm that elects a leader in $O(D)$ time. If so, one could try to combine this algorithm with the algorithm presented in this chapter to achieve a running time of $O(\min\{D, L\})$.

We showed in Section 5.6.4 that there is no algorithm that solves the leader election problem in general graphs without geometric information. However, it seems reasonable to adapt the presented algorithm to other specific graphs such as the infinite square grid graph and its natural embedding in the Euclidean plane in which each node lies at integer coordinates. For this, it would be necessary to modify the boundary setup phase, the solitude verification phase, and the boundary identification phase since these phases explicitly use the geometric information induced by the embedding of the graph. The remaining phases of the leader election algorithm only rely on the boundaries established in the boundary setup phase and should therefore work without any modification.

Finally, it would also be interesting to see how the feasibility and the complexity of leader election changes when the amount of geometric information provided to the particles is varied. For example, we conjecture that if all nodes have the same local north direction (i.e., the particle system has a global compass), leader election can be solved *deterministically* using a variant of the presented algorithm. Furthermore, one could investigate whether the assumption of common chirality is actually necessary for leader election.

Chapter 6

Shape Formation with Programmable Matter

We continue our investigation of the algorithmic foundations of programmable matter in this chapter by considering the *shape formation problem.* As its name suggests, this problem requires the particles of a particle system to reorganize into a specific shape such as a line or a triangle.

There is an interesting connection between the shape formation problem and the leader election problem we investigated in the previous chapter. We first presented the ideas behind this connection in [Der+15b], albeit in a slightly different context. Consider a ring of six expanded particles in which the local north directions of the particles are chosen in such a way that the particle system looks identical from the local point of view of each individual particle. To clarify this intuitive description, note that such a ring actually has a hexagonal shape due to the underlying graph G_{ET} and its embedding in the plane. Suppose that the particles have to reorganize into a straight line. To solve this problem, some particle in the ring has to contract since no other movement is possible. At the same time, an algorithm has to make sure that the particles on the opposite side of the ring do not contract since this would subdivide the particle system into multiple connected components. Therefore, an algorithm has to *break the symmetry* of the initially given ring in order to solve the shape formation problem in this case. This can, of course, be achieved by *electing a leader* and designating it as the first particle to contract.

On the other hand, suppose that we had an algorithm for the construction of a straight line. Such an algorithm would allow for a very simple approach to leader election: Once the line has been constructed, the two particles at the ends of the line engage in a competition using random coin flips that are exchanged by sending tokens along the line. The particle that wins the competition becomes the leader of the particle system. Therefore, the leader election problem can essentially be *reduced* to the problem of forming a line.

We introduced a first approach to shape formation in the amoebot model in [Der+15b] and later extended this approach to form more shapes in [Der+15a]. The framework presented in this work allows for the construction of simple shapes such as lines, triangles, and hexagons. The underlying idea is to construct a shape by sequentially adding new particles to the end of a construction path that originates at a given leader particle. For the example of a line, the construction path is also a line. For a hexagon, the construction path spirals outwards from the leader particle. The particles that are not part of the shape are organized into a spanning forest: Particles adjacent to the shape form the roots, and the remaining particles form the actual trees. To move the particles towards the end of the construction path, the roots traverse the boundary of the shape in a common direction while pulling the particles in their respective trees behind them. This basic approach to shape formation has two major disadvantages. First, it can only construct simple shapes that are amenable to being built using a construction path. Furthermore, due to the sequential way in which the approach extends the shape, it generally requires $\Omega(n)$ rounds to construct a shape consisting of n particles, even if the shape could in principle be formed more efficiently.

In this chapter, we present a more advanced approach to shape formation that alleviates both of these shortcomings. Our approach can be used to construct a large class of shapes. Specifically, we consider shapes composed of a constant number of equilateral triangles of unit size that are arranged on a grid. The approach can construct any shape that is *sequentially constructible*, i.e., any shape that can be formed by sequentially adding triangles to the outside of the shape under construction. We assume that the initial particle system is well-initialized in that the particles form a triangle and the memory of each particle only holds a representation of the desired shape. Under these assumptions, our approach forms a scaled representation of the given shape that includes all particles in $O(\sqrt{n})$ rounds. We show that this bound is optimal in the sense that for every shape deviating from the initial triangle, any algorithm requires $\Omega(\sqrt{n})$ rounds to construct the shape in the worst case.

Underlying Publication This chapter is based on the following publication.

> 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 Parallelism in Algorithms and Architectures (SPAA)*, see [Der+16b].

Unfortunately, we had to slightly weaken the main result in this chapter in comparison to the underlying publication to compensate for an oversight in the original analysis. Specifically, we had to restrict the shape formation algorithm to shapes that are sequentially constructible instead of arbitrary shapes.

Outline We already introduced the amoebot model and gave an overview of the related literature, including the work pertinent to shape formation, in the previous chapter. Therefore, we can turn directly to the technical part. We begin in Section 6.1 by formally specifying the shape formation problem under consideration. In Section 6.2 we introduce a number of movement primitives that allow us to move large sets of particles in an efficient manner. These primitives are used extensively throughout the remainder of this chapter. We then describe an algorithm that transforms the initially given triangle into an intermediate structure in Section 6.3, which simplifies the shape formation process. We finally present the actual shape formation algorithm in Section 6.4. Each of these algorithms is accompanied by a corresponding analysis in the respective section. We close this chapter in Section 6.5 by discussing some directions for potential future research on shape formation and, more generally, on the algorithmic foundations of programmable matter overall.

6.1 Problem Statement

In the *shape formation problem*, a particle system has to reconfigure into a given shape. Consider a set S of faces in the planar embedding of G_{ET}. Note that S corresponds to a set of nodes in the dual graph of G_{ET}. We say two triangular faces of G_{ET} are *connected* if they share a side, which is the case if and only if the corresponding nodes in the dual graph are connected. We say S is connected if the subgraph of the dual graph induced by S is connected. We define a *shape* to be a connected, constant-size set of faces in G_{ET}.

Let $s = |S|$. We say S is *sequentially constructible* if there is a permutation $(a_1, a_2, \ldots, a_s)$ of the faces in S such that for each prefix $(a_1, a_2, \ldots, a_i)$ of the permutation the shape corresponding to that prefix is connected, and each triangular face a_i has a side that lies on the outer boundary of the shape induced by the prefix $(a_1, a_2, \ldots, a_i)$. Intuitively, this means that it must be possible to construct S by sequentially adding faces to the outside of the shape in a way such that each intermediate shape created during the construction is connected. Figure 6.1 shows a shape that is not sequentially constructible, and Figure 6.2 shows a shape that is sequentially constructible.

Consider a transformation that consists of a translation, a rotation by a multiple of 60°, and an isotropic scaling. Let S' be the set of triangles resulting from applying this transformation to each face of a shape S. We require the transformation to be such that the vertices of the triangles in S' coincide with nodes of G_{ET}. Let $V(S')$ be the set of nodes in G_{ET} that lie on a vertex, on an edge, or on the inside of a triangle from S'. We call $V(S')$ a *representation* of the shape S. Figure 6.2 illustrates this definition by an example.

For the shape formation problem, we assume that the particles system initially forms a triangle consisting entirely of contracted particles. The local memory of every particle initially only contains a binary representation of a

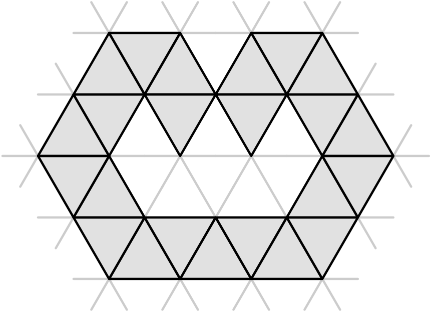

Figure 6.1: A shape that is not sequentially constructible. For each permutation of the faces in this shape either there is an intermediate shape that is not connected or at some point a face is added to the inside of an intermediate shape.

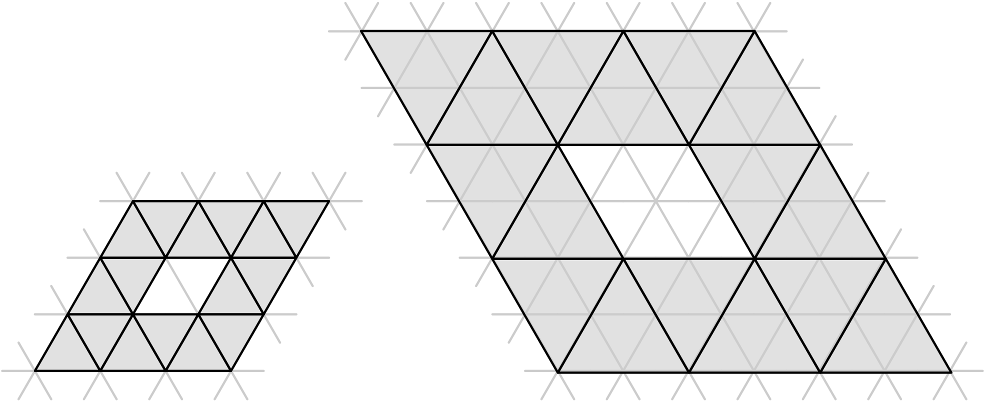

Figure 6.2: Two representations of a shape consisting of 16 faces. We will often represent sets of particles as geometric shapes. In this example, each position inside a triangle, on a side of a triangle, and on a vertex of a triangle is occupied by a particle. In the left representation the triangles coincide with the faces of G_{ET}. Note that in this representation the hole in the middle of the shape is actually closed and therefore the representation is a solid quadrilateral. In the right representation each triangle consists of four faces of G_{ET} and the shape is rotated counter-clockwise by 120° compared to the left representation.

sequentially constructible shape S. For ease of presentation, we assume that the total number of particles n is a triangular number and, therefore, the particles initially form a perfect triangle. We briefly discuss in Section 6.5 how this assumption can be lifted. An algorithm solves the shape formation problem if it terminates for all particles and once all particles have terminated, the set of occupied nodes corresponds to some representation of S. Note that we allow the representation to contain expanded particles.

6.2 Movement Primitives

The shape formation algorithm reconfigures the particle system using a set of primitives for moving connected sets of particles. The simplest of these primitives moves a *chain* of particles along a *path*: Consider a simple directed path P of length ℓ in G_{ET}. Let the first m nodes of P be occupied by contracted particles for some $m \leq \ell$. The goal is to let the particles *traverse* P: i.e., the particles should move only through the nodes in P, they have to stay connected at all times, and in the end the particles should occupy the last m nodes of P while being contracted. This is achieved by letting the particles move as a connected chain using handovers.

Every particle in the chain moves forward according to the following *greedy movement strategy*: The front-most particle in the chain expands along P whenever it is contracted. The last particle contracts whenever it is expanded. A contracted particle that is not the front-most particle pushes the particle in front of it if possible. Finally, an expanded particle that is not the last particle pulls the particle behind it if possible. We assume that the front-most particle of the chain can locally deduce P, i.e., it knows in which direction to expand and when to stop. Note that the given movement strategy never changes the order of the particles in the chain and keeps the chain connected.

The following lemma bounds the number of rounds required for a particle chain to traverse a path P. The proof of the lemma is based on a domination argument, the concept of which we introduced in the previous chapter. Note that in the previous chapter we used domination arguments to reason about the movement of *tokens*, whereas the proof of the following lemma uses a domination argument to reason about the movement of *particles*.

Lemma 6.1. *A chain of particles of size $m \leq \ell$ can traverse a path P of length ℓ in $O(\ell)$ rounds.*

Proof. We compare the asynchronous execution of the greedy movement strategy with a synchronous execution in which all particles move at the same time and each particle is allowed to be part of at most one movement, i.e., one expansion, one contraction, or one handover. Let $P = (v_1, v_2, \ldots, v_\ell)$. The particles initially occupy the nodes v_1 to v_m. For a particle p we write $h_i^s(p) = j$ if the head of p occupies the node v_j at the beginning of round i

of the synchronous execution, and we write $t_i^s(p) = j$ if the tail of p occupies the node v_j at the beginning of round i of the synchronous execution. We define $h_i^a(p)$ and $t_i^a(p)$ analogously for the asynchronous execution. We show by induction on the number of rounds that $h_i^a(p) \geq h_i^s(p)$ and $t_i^a(p) \geq t_i^s(p)$ for every particle p and every round i.

The statement holds by definition for $i = 0$. Suppose that the statement holds for round i and consider a particle p. We show $h_{i+1}^a(p) \geq h_{i+1}^s(p)$ and $t_{i+1}^a(p) \geq t_{i+1}^s(p)$. If $h_i^a(p) > h_i^s(p)$ or $t_i^a(p) > t_i^s(p)$ then the statement holds. So assume $h_i^a(p) = h_i^s(p)$ and $t_i^a(p) = t_i^s(p)$. We distinguish the following five cases for the movement of p in round i of the synchronous execution.

1. If p does not move then certainly the statement holds.

2. If p expands into an empty node then p must be the front-most particle of the chain so that p also expands in the asynchronous execution.

3. If p contracts without pulling another particle then p must be the last particle of the chain so that p also contracts in the asynchronous execution.

4. If p performs a handover with the particle p' in front of it then p must be contracted and p' must be expanded at the beginning of round i of the synchronous execution. By the induction hypothesis, p' must also be expanded at the beginning of round i of the asynchronous execution. Therefore, once p or p' is activated in round i of the asynchronous execution, the particles perform a handover.

5. If p performs a handover with the particle p' behind it then p must be expanded and p' must be contracted at the beginning of round i of the synchronous execution. By the induction hypothesis, p' must also be contracted at the beginning of round i of the asynchronous execution. Therefore, once p or p' is activated in round i of the asynchronous execution, the particles perform a handover.

Since the statement holds in all cases, the induction is complete.

It is not hard to see that the synchronous execution of the greedy movement strategy lets the particle chain traverse P in $O(\ell)$ rounds. Therefore, the domination argument above implies the lemma. □

The remaining movement primitives operate on a set of contracted particles that form a triangle in G_{ET}. We define four operations for such triangles, namely *expansion*, *contraction*, *rotation*, and *shift*. An example of the first three of these operations is shown in Figure 6.3. The expansion of a triangle results in a geometric shape we refer to as an *expanded triangle*. An expanded triangle is a quadrilateral consisting of two triangles of the same size as the

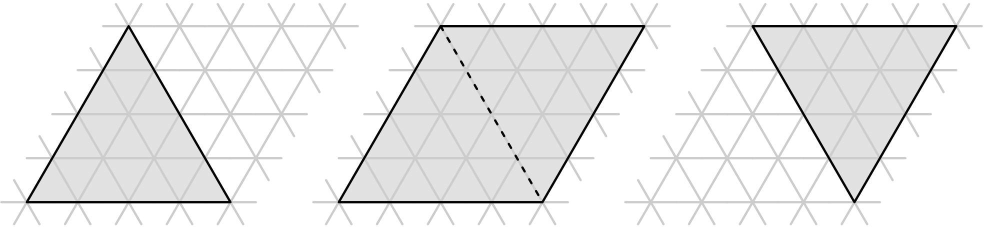

Figure 6.3: Expansion, contraction, and rotation of a triangle. A triangle (left) can expand to form an expanded triangle (middle). An expanded triangle can contract back to a triangle (right). Concatenating an expansion and a contraction effectively rotates a triangle by 60° around one of its vertices.

original triangle that share a common side. The contraction of a triangle transforms an expanded triangle back into a triangle. An expansion can be combined with a contraction to rotate a triangle around one of its vertices by 60°. Finally, a shift of a triangle moves all of its particles in a common direction by one node.

The movement of a triangle is controlled by a designated particle occupying one of its vertices. We refer to this particle as the *coordinator* of the triangle. We assume that the particles on the boundary of a triangle are organized into a cycle oriented counter-clockwise around the triangle. Each particle on the boundary knows the direction of its preceding and its succeeding particle in this cycle.

A triangle coordinator c can perform a counter-clockwise expansion of its triangle in the following way. Let T be the set of nodes occupied by the triangle before the expansion. Consider the side of the triangle that is incident to c and that occurs first when traversing the boundary of the triangle in counter-clockwise direction. Let L_1 be the set of nodes corresponding to this side of the triangle, see Figure 6.4. Let L_2 be the set of nodes corresponding to the other side of the triangle that is incident to c. Finally, let L_3 be the set of nodes we get by rotating L_2 counter-clockwise around c by 60°. Define a *row* of the triangle as a maximal subset of T that forms a straight line in G_{ET} that is parallel to the side of the triangle opposite of c.

The coordinator uses the following token passing scheme to control the expansion. First, c sends an *activation token* along L_1. Once a particle receives the activation token, it forwards the token along L_1 and sends a *row token* along its row. A particle p that receives a row token forwards it to the next particle p' along its row. From that point on, p considers p' to be the particle in front of it in a particle chain and p' considers p to be the particle behind it in the chain. In each row there is a unique particle that initially occupies L_2.

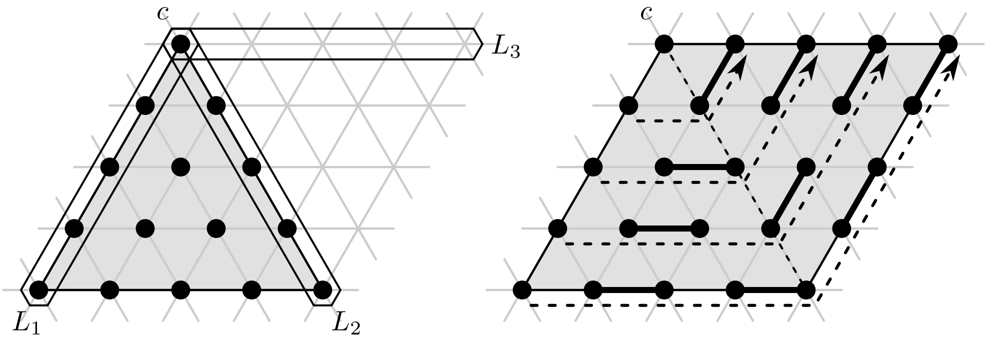

Figure 6.4: Example of a counter-clockwise triangle expansion. The coordinator c is located at the top vertex of the triangle. The rows of the triangle expand independently of each other by moving as particle chains along the paths shown as dashed arrows in the right part of the figure. Once every particle not occupying a node in L_1 is expanded, the triangle expansion is complete.

These particles are the front-most particles of the particle chains. Once such a particle receives a row token, it starts moving as depicted in the right part of Figure 6.4. Thereby, the particles in each individual row move as a connected particle chain. The particles occupying L_1 are part of these particle chains but they are forced to stay at their position. Thus, according to the rules of the greedy movement strategy, eventually all particles are expanded except for the particles occupying L_1. At this point, the particles form an expanded triangle.

To determine when the expansion of the triangle is complete, the coordinator c proceeds as follows: When c sends out the activation token, it also creates a *validation token*. A particle holding the validation token tries to forward it along L_3, i.e., it forwards the token if there is a particle occupying the next position in L_3 and it keeps the token if there is no such particle. It is not hard to see that the validation token can only completely traverse L_3 when the triangle expansion is complete. Once the validation token reaches the last node of L_3, it is sent back along L_3 to c. Note that the particle p occupying the last node of L_3 in the expanded triangle can easily be distinguished from the remaining particles occupying nodes in L_3 since p was originally a vertex of the triangle. When the validation token returns to c, the coordinator knows that the triangle expansion is complete.

On the basis of the ideas presented in the previous paragraphs, it is not hard to design local-control algorithms for the remaining movement primitives. We have the following lemma.

Lemma 6.2. *A triangle of side length ℓ can be expanded, contracted, rotated, and shifted in $O(\ell)$ rounds.*

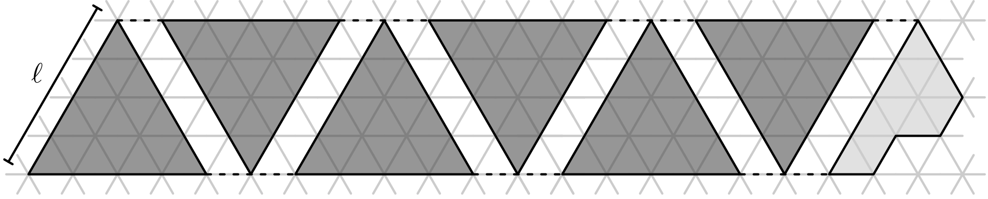

Figure 6.5: Intermediate structure.

Proof. We prove the statement of the lemma for the case of triangle expansion. First, note that the activation token and the row tokens only move through static particles. The activation token traverses the side of the triangle corresponding to L_1, which consists of ℓ particles. Since the token is forwarded every round, this traversal takes at most $O(\ell)$ rounds. Once the activation token has completed its traversal, it takes another $O(\ell)$ rounds until every row token reaches a particle occupying a node in L_2. At this point, the particles of each row have been organized into a particle chain. The particle chains move independently of each other. The movement of a particle chain stops once all its particles are expanded except for the particle occupying a node in L_1. Note that the path along which a particle chain moves has length $O(\ell)$. By arguments analogous to those presented in the proof of Lemma 6.1, the movement of a particle chain takes $O(\ell)$ rounds. Once all particle chains have completed their movement, the validation token can traverse L_3 unhinderedly. Since this token traverses $O(\ell)$ static particles, it takes another $O(\ell)$ rounds until it returns to the coordinator. Taking the sum of all of these terms gives us a total running time of $O(\ell)$ rounds. □

Consider a triangle consisting of m particles. According to Lemma 6.2, such a triangle can be expanded, contracted, rotated, and shifted in $O(\sqrt{m})$ rounds. This allows us to move large groups of particles efficiently, which is crucial for the performance of the shape formation algorithm.

6.3 Intermediate Structure

The goal of this section is to reconfigure the particle system from the initially given triangle into an intermediate structure that simplifies the actual shape formation process. The intermediate structure we aim for is shown in Figure 6.5. It consists of Δ equilateral triangles of side length ℓ (shown in dark gray) that are arranged in a straight line, and a remainder of particles (light gray) that is too small to form an additional triangle. All particles in this structure are contracted. Note that the algorithm we present in this section actually creates a structure that can be slightly worse in the sense that it might leave a larger remainder of particles. We will come back to this at the end of the section; for now we assume that we can build the ideal structure.

Given a certain number of particles, the number of triangles Δ in the intermediate structure is completely determined by the choice of the side length ℓ. Recall that $s = |S|$ is the number of faces in the given shape. For the intermediate structure to be useful, we want to choose ℓ such that

$$\frac{3}{4}s + 1 \leq \Delta \leq s - 3.$$

The reason for these bounds on Δ will become apparent in the following section. Let L be the side length of the initial triangle. We choose

$$\ell = \left\lceil \frac{L}{\lfloor c \cdot \sqrt{s} \rfloor} \right\rceil$$

for a constant $c < 1$. The following lemma establishes a value of c such that our choice of ℓ is appropriate.

Lemma 6.3. *For $c = 50/51$, $s \geq 153$, and $L \geq 99 \cdot \sqrt{s}$, we have*

$$\frac{3}{4}s + 1 \leq \Delta \leq s - 3.$$

Proof. The number of particles $t(k)$ in a triangle of side length k is

$$t(k) = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}.$$

Therefore, we have

$$\Delta = \left\lfloor \frac{t(L)}{t(\ell)} \right\rfloor = \left\lfloor \frac{L(L+1)}{\ell(\ell+1)} \right\rfloor.$$

Let $m = \lfloor c \cdot \sqrt{s} \rfloor$ such that $\ell = \lceil L/m \rceil$. This implies

$$L/m \leq \ell \leq L/m + 1. \tag{6.1}$$

We first establish the upper bound on Δ. We have

$$\Delta = \left\lfloor \frac{L(L+1)}{\ell(\ell+1)} \right\rfloor \leq \frac{L(L+1)}{\ell(\ell+1)} \leq \frac{L(L+1)}{\ell^2}.$$

Applying the left inequality of Equation 6.1 gives us

$$\Delta \leq \frac{L(L+1)}{L^2} \cdot m^2 = \frac{L+1}{L} \cdot \lfloor c \cdot \sqrt{s} \rfloor^2 \leq \frac{L+1}{L} c^2 s.$$

Since $L \geq 50$ we have $(L+1)/L \leq c^{-1}$, which implies

$$\Delta \leq cs \leq s - 3,$$

where the last inequality holds since $s \geq 153$.

We now turn to the lower bound on Δ. We have

$$\Delta = \left\lfloor \frac{L(L+1)}{\ell(\ell+1)} \right\rfloor \geq \frac{L(L+1)}{\ell(\ell+1)} - 1 \geq \left(\frac{L}{\ell+1}\right)^2 - 1.$$

Applying the right inequality of Equation 6.1 gives us

$$\begin{aligned}\Delta &\geq \left(\frac{L}{L/m+2}\right)^2 - 1\\ &= \left(\frac{L}{L+2m}\right)^2 \cdot m^2 - 1\\ &= \left(\frac{L}{L+2m}\right)^2 \cdot \lfloor c \cdot \sqrt{s} \rfloor^2 - 1\\ &\geq \left(\frac{L}{L+2m}\right)^2 \cdot (c \cdot \sqrt{s} - 1)^2 - 1,\end{aligned}$$

where the last inequality holds because $c \cdot \sqrt{s} \geq 1$. Simple arithmetic shows that since $L \geq 99 \cdot \sqrt{s}$ we have

$$\left(\frac{L}{L+2m}\right)^2 \geq c^2,$$

which implies

$$\Delta \geq \left(c^2 \cdot \sqrt{s} - c\right)^2 - 1 \geq \frac{3}{4}s + 1,$$

where the last inequality holds because $s \geq 130$. □

We now describe how the initial triangle is reorganized into the intermediate structure. First, we use the algorithm presented in the previous chapter to perform leader election. This establishes a unique leader particle on the outer boundary of the initial triangle. Once the leader has been established, it initiates a broadcast throughout the particle system to stop the execution of the leader election algorithm as described in Section 5.6.2 of the previous chapter. The leader particle then sends a token along the boundary to transfer its leadership to a particle at a vertex of the initial triangle. Since the triangle has a circumference of $O(\sqrt{n})$, the leader election takes $O(\sqrt{n})$ rounds with high probability according to Corollary 5.12. Transferring the leadership to a vertex of the triangle also takes $O(\sqrt{n})$ rounds.

Next, the particle system determines the value of ℓ. Note that, in general, a single particle cannot store $\ell = \Omega(\sqrt{n})$ since the memory of a particle has constant size. Therefore, we have to store ℓ in a distributed fashion over multiple particles. To determine ℓ, we use the following token passing scheme: The leader sends a *counter token* along an adjacent side of the initial triangle.

This token stores a counter that counts the number of steps the token takes modulo $\lfloor c \cdot \sqrt{s} \rfloor$. Since c and s are constants, also $\lfloor c \cdot \sqrt{s} \rfloor$ is a constant so that the counter only requires constant memory. The counter is initialized with the value 0 and is incremented whenever the token is forwarded. When a particle holds the counter token while the counter is 0, the particle creates a *marker token.* Note that thereby also the leader creates a marker token. Once the counter token has traversed the side of the triangle, it is consumed by the particle occupying the vertex at the end of the side.

Upon consuming the counter token, the particle at the end of the side creates a *terminal token.* Both the marker tokens and the terminal token travel back towards the leader. Each particle is allowed to store either one marker token or the terminal token. An exception to this rule is the particle that creates the terminal token: If this particle creates both a marker token and a terminal token, it can temporarily store both tokens but has to pass the marker token before the terminal token. Since the marker tokens all move towards the leader, they eventually occupy a segment of consecutive particles starting at the leader. To detect when this state is reached, each marker token stores a bit that is initially false. The bit becomes true if the marker token is held by the leader or if the marker token has another marker token in front of it that has its bit set to true. Once the terminal token has a marker token with its value set to true in front of it, it knows that all marker tokens are done moving. The terminal token then moves through the segment of marker tokens back to the leader to inform it that the token passing scheme has terminated. Note that we allow the particles to hold both a marker token and the terminal token for this traversal of the segment. After the token passing scheme has terminated, the length of the segment of marker tokens corresponds to the value of ℓ. We prove this fact in the following lemma.

Lemma 6.4. *The segment of marker tokens created by the token passing scheme has length ℓ.*

Proof. Consider the set of particles that create marker tokens. These particles divide the side of the triangle into disjoint segments. Let $m = \lfloor c \cdot \sqrt{s} \rfloor$. If L is divisible by m then all segments are of length exactly m. Otherwise, the last segment is shorter than m. In both cases, the number of segments is $\lceil L/m \rceil$. Since the number of segments corresponds to the number of marker tokens, the lemma holds. □

Next, we analyze the running time of the token passing scheme. Note that the presented token passing scheme resembles the token passing scheme used in the solitude verification phase of the leader election algorithm, see Section 5.4.5. However, the two schemes differ in the initial placement of the tokens and the number of tokens a particle (or agent) can hold. We have the following lemma.

Lemma 6.5. *The token passing scheme to determine ℓ takes $O(\sqrt{n})$ rounds.*

Proof. The counter token traverses the side of the triangle in $O(\sqrt{n})$ rounds. When its traversal is complete, all marker tokens and the terminal token have been created and move towards the leader. Once the marker tokens occupy a segment of consecutive particles, it takes $O(\sqrt{n})$ rounds for the bit in every marker token to become true. The terminal token reaches the end of the segment, detects that the marker tokens are done moving, and moves back to the leader in an additional $O(\sqrt{n})$ rounds.

It remains to determine the time from the creation of the last marker token until the marker tokens occupy a segment of consecutive particles. We use a domination argument that is very similar to the argument used in the proof of Lemma 5.9. We define round 0 of the asynchronous execution to be the earliest round such that at the beginning of the round all marker tokens have been created. We compare the asynchronous execution with the following synchronous execution: In round 0 of the synchronous execution, each token is stored at the particle that created it. The tokens then move in lockstep towards the leader. We assign the numbers $1, 2, \ldots, L$ to the particles of the side of the triangle starting with 1 at the leader. For a marker token t let $s_i(t)$ be the number assigned to the particle that holds t at the beginning of round i of the synchronous execution. Let $a_i(t)$ be defined analogously for the asynchronous execution. We show by induction on i that $a_i(t) \leq s_i(t)$ for all i and for every marker token t.

The statement holds for $i = 0$ by definition. Suppose the statement holds for round i. Consider a marker token t. We show $a_{i+1}(t) \leq s_{i+1}(t)$. If $a_i(t) < s_i(t)$ then certainly $a_{i+1}(t) \leq s_{i+1}(t)$. So assume $a_i(t) = s_i(t)$. Consider the next token t' from t towards the leader. If $s_i(t') < s_i(t) - 1$ then by the induction hypothesis we have

$$a_i(t') \leq s_i(t') < s_i(t) - 1 = a_i(t) - 1$$

so that t does not have a token in front of it at the beginning of round i in the asynchronous execution. Therefore, t is forwarded in round i of the asynchronous execution when the particle holding t is activated, which implies $a_{i+1}(t) \leq s_{i+1}(t)$.

Now suppose $s_i(t') = s_i(t) - 1$. Note that $\lfloor c \cdot \sqrt{s} \rfloor \geq 2$ by our choice of c and the lower bound on s established in Lemma 6.3. Therefore, at the beginning of round 0 of the synchronous execution the marker tokens are separated by at least one particle that does not hold a token. This implies that when two tokens occupy neighboring particles in the synchronous execution, both tokens must be at their final position. Hence, t and t' must be at their final position so that neither token is forwarded in round i in the synchronous execution. This implies $a_{i+1}(t) \leq s_{i+1}(t)$ and completes the induction.

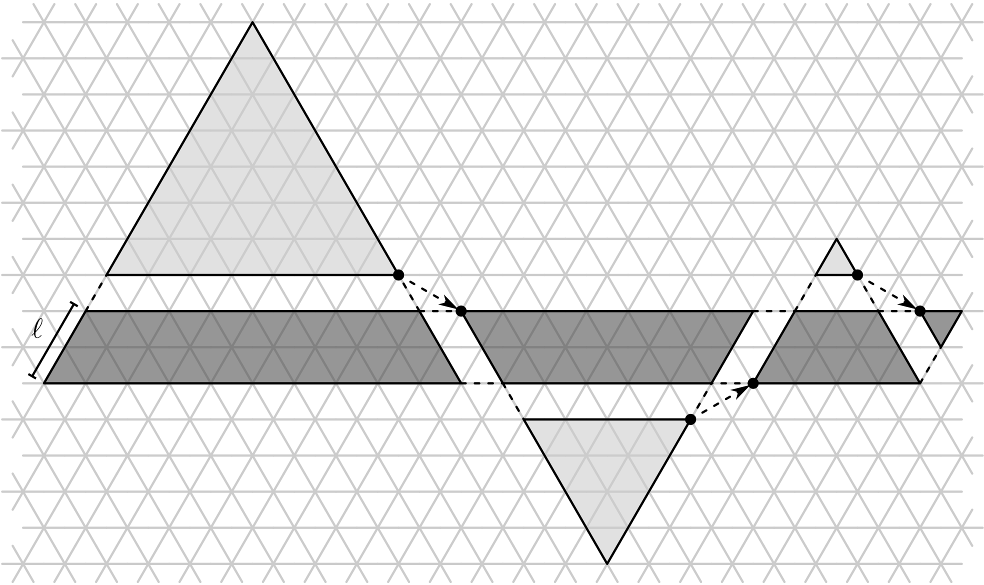

Figure 6.6: Construction of the intermediate structure. The parts forming the intermediate structure are shown in dark gray. The triangles that are shifted and rotated are shown in light gray. For each dashed arrow, the circles at the origin and at the tip of the arrow mark the same particle before and after the respective triangle is moved. As an example, the left-most triangle is rotated clockwise by 60° around the vertex at the origin of the corresponding arrow, then it is shifted once to the right and once to the bottom-right, and finally it is rotated clockwise by 120° around the vertex at the tip of the arrow.

It is not hard to see that the synchronous execution moves all tokens to their final position in $O(\sqrt{n})$ rounds. By the domination argument above, the asynchronous execution also requires $O(\sqrt{n})$ rounds. □

Once ℓ has been established, the intermediate structure is formed using a recursive process that is coordinated by the leader. The process reconfigures the particle system using triangle shifts and triangle rotations as illustrated in Figure 6.6. It starts with the initial triangle. First, it splits the triangle into a smaller triangle and an isosceles trapezoid with legs of length ℓ. The isosceles trapezoid becomes the first part of the intermediate structure. Without loss of generality, we assume that the leader initially occupies the top vertex of the largest triangle shown in Figure 6.6. To determine the length of the legs of the isosceles trapezoid, the leader sends a token that transfers the length ℓ from the top of the triangle to the bottom by traversing the path shown in Figure 6.7.

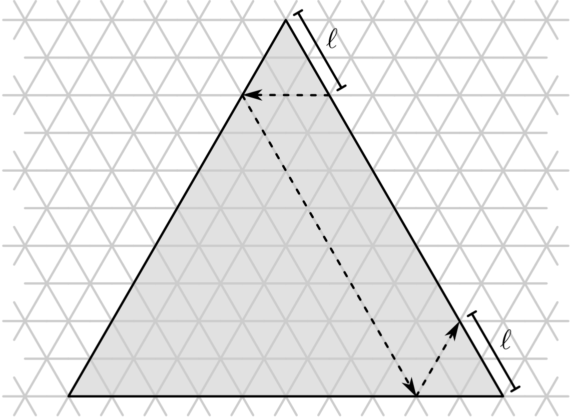

Figure 6.7: Transferring the length ℓ from the top of a triangle to the bottom.

Note that traversing this path can be achieved using only local information. To separate the smaller triangle from the isosceles trapezoid, the token traverses the circumference of the triangle and establishes a counter-clockwise cycle in the particles that designates the set of particles as a triangle. Then the token establishes a coordinator for the new triangle at its bottom right vertex.

To form the next part of the intermediate structure, the new triangle is rotated, shifted twice, and rotated again as shown in Figure 6.6. Once the movement of the triangle is complete, the coordinator gives up its role and sends a token to the leader. Upon receiving this token, the leader sends a broadcast through the triangle that removes the cycle from the boundary. The process then recurs on the new triangle. Note that the orientation of the rotations and the direction of the shifts changes with every recurrence. Also the paths taken by the tokens used in the process change slightly from one recurrence to the next. The process ends after moving the first triangle that has a side length less or equal to ℓ.

The following lemma provides an upper bound on the number of rounds required to build the intermediate structure.

Lemma 6.6. *The presented algorithm builds the intermediate structure in $O(\sqrt{n})$ rounds.*

Proof. The algorithm terminates after $O(L/\ell) = O(1)$ recursions. In each recursion the tokens coordinating the construction process traverse an overall distance of $O(\sqrt{n})$, which takes $O(\sqrt{n})$ rounds. The newly established triangle is rotated and shifted a constant number of times, which takes $O(\sqrt{n})$ rounds according to Lemma 6.2. □

The final part of the algorithm subdivides the intermediate structure into triangles that are arranged as shown in Figure 6.5 by organizing the particles

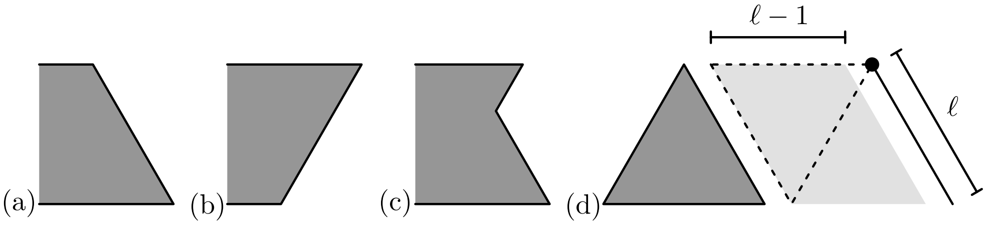

Figure 6.8: End of the intermediate structure.

on the sides of each triangle into a cycle. This is achieved by letting a single token traverse the sides of all triangles, which the token can do using only local information. During the traversal, the token counts the number of triangles in the intermediate structure. After the traversal, it reports this number to the leader. Since there is a constant number of triangles in the intermediate structure and the sides of each triangle have length $\ell = O(\sqrt{n})$, the traversal takes $O(\sqrt{n})$ rounds.

As we mentioned above, the given algorithm builds a structure that is slightly different from the ideal intermediate structure shown in Figure 6.5. Roughly speaking, the end of the structure is not necessarily perfectly suited for the purpose of dividing the structure into triangles of side length ℓ. Consequently, the number of triangles Δ' in the structure constructed by the algorithm might be smaller than the number of triangles Δ in an ideal intermediate structure. The following lemma characterizes the properties of the intermediate structure constructed by the algorithm.

Lemma 6.7. *The algorithm builds an intermediate structure containing Δ' triangles where $\Delta - 1 \leq \Delta' \leq \Delta$.*

Proof. We have $\Delta' \leq \Delta$ by definition. To show $\Delta' \geq \Delta - 1$ we need the following observation: The end of the structure built by the algorithm takes on one of the three shapes depicted in the first three parts of Figure 6.8. If the last triangle moved by the algorithm has a side length greater or equal than $\ell - 1$, the shape in (a) or (b) is created. Otherwise, the end of the structure conceptually resembles the shape shown in (c).

The remainder of the proof argues about Part (d) of Figure 6.8. Without loss of generality, let the last triangle in the intermediate structure be oriented as shown in dark gray. The potential next triangle, shown as a dashed outline, cannot be formed. Therefore, the vertex of that triangle marked by a black dot must be unoccupied. This holds because if it were occupied then all nodes of the dashed triangle would be occupied, no matter which of the three possible shapes the end of the structure has. Since the node marked by the circle is unoccupied, also the nodes on the bold line must be unoccupied and every node to the right of that line must be unoccupied. Therefore, all particles of the intermediate structure that are not part of a triangle have to lie in the light gray area. This area contains $\ell(\ell-1)$ nodes. Since a triangle of side length ℓ contains $\ell(\ell+1)/2$ nodes, the particles that are not part of any triangle could form at most one additional triangle, which shows the lemma. □

We summarize the results of this section in the following theorem.

Theorem 6.8. *For $s \geq 153$ and $L \geq 99 \cdot \sqrt{s}$ the algorithm builds an intermediate structure consisting of Δ' triangles where*

$$\frac{3}{4}s \leq \Delta' \leq s-3.$$

in $O(\sqrt{n})$ rounds, w.h.p.

Proof. The bound on Δ' follows from Lemma 6.3 and Lemma 6.7. For the running time bound we simply add up the bounds presented throughout this section for the individual steps of the algorithm: Establishing a leader at a vertex of the initial triangle takes $O(\sqrt{n})$ rounds with high probability. The token passing scheme for determining the value of ℓ takes $O(\sqrt{n})$ rounds according to Lemma 6.5. Building the intermediate structure takes $O(\sqrt{n})$ rounds according to Lemma 6.6. Finally, subdividing the intermediate structure into triangles takes another $O(\sqrt{n})$ rounds. Therefore, the overall running time of the algorithm is $O(\sqrt{n})$. □

6.4 Shape Formation Algorithm

We now turn to the shape formation algorithm, which transforms the intermediate structure into a representation of the desired shape. In this section, we use the term *triangle* exclusively to refer to a set of particles that forms a triangle, and we use the term *face* to refer to a triangular face of the given shape S. From a high-level perspective, the shape formation algorithm can be interpreted as a sequential algorithm that is globally coordinated by the leader. On a low level, however, the algorithm implicitly parallelizes the movement of large sets of particles by applying the movement primitives presented in Section 6.2. This parallelization is crucial for the running time of the algorithm.

The main idea behind the algorithm is to construct the shape one face at time by sequentially adding triangles from the intermediate structure to the outside of the shape under construction. Before we describe how this can be achieved, we point out a number of difficulties that the algorithm has to overcome in order to solve the shape formation problem.

First, note that the faces of a shape have overlapping edges, while the edges of triangles cannot overlap because each node in G_{ET} can only be occupied by a single particle. As a consequence, building a representation of a shape

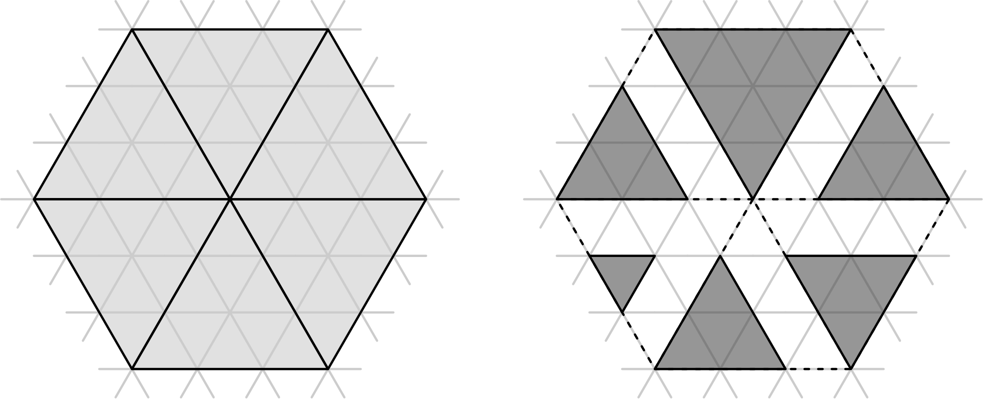

Figure 6.9: Realization of a representation. The left part shows a representation of a shape with six faces. The right part shows a possible realization of this representation. Note that the realization requires triangles of three different side lengths.

generally requires triangles of three different side lengths, namely ℓ, $\ell - 1$, and $\ell - 2$, see Figure 6.9. Therefore, the algorithm must remove one or two rows of particles from some of the triangles, and the removed particles must be incorporated back into the structure. There are two additional special cases concerning the placement of triangles in a representation. We will address these cases once we have established a more detailed picture of the shape formation algorithm.

Second, recall that in general there is a set of particles at the end of the intermediate structure that does not form a triangle, see Figure 6.8. We refer to this set of particles as the *remainder* of the intermediate structure. To include all particles of the particle system into the representation of the shape, the remainder has to be incorporated along with the particles that form the triangles of the intermediate structure.

Finally, the number of triangles Δ' in the intermediate structure is lower than the number of faces s in the given shape S according to Theorem 6.8. To make up for the missing triangles, the algorithm has to use some expanded triangles to form pairs of neighboring faces in the representation.

6.4.1 Simplified Algorithm

For ease of presentation, we ignore most of the above difficulties for now. So suppose that the intermediate structure does not contain a remainder, it consists of exactly s triangles, and the particles removed from the triangles do not have to be incorporated into the representation. The algorithm constructs a representation of the shape in which the sides of the faces have length ℓ, i.e., the sides cover ℓ nodes in G_{ET}. Since the shape S is sequentially constructible, there is a permutation $(a_1, a_2, \ldots, a_s)$ of the faces in S such that for each prefix $(a_1, a_2, \ldots, a_i)$ of the permutation the shape corresponding to that prefix is connected, and each triangular face a_i has a side that lies on the outer boundary of the shape induced by the prefix $(a_1, a_2, \ldots, a_i)$. Since s is a constant, the leader of the particle system can locally compute such a permutation.

The algorithm adds triangles to the shape according to the order of the faces in the computed permutation. Since each prefix of the permutation forms a connected shape, the shape remains connected during its construction. Furthermore, since each newly added face has a side on the outer boundary of the shape induced by the corresponding prefix, the shape can always be extended by adding a new triangle to the outer boundary of the shape.

Before the algorithm starts the construction of the shape, it maps the triangles in the intermediate structure to the faces of the shape. It then prunes the triangles in the intermediate structure to the correct size by removing one or two rows from a triangle. The removed rows are moved out of the way using particle chain movement. The algorithm then sequentially moves the triangles from the intermediate structure along the outer boundary of the shape to their respective goal positions using triangle rotations and shifts. Whenever the intermediate structure or the removed rows interferes with the placement of the next triangle, they are moved out of the way.

The intermediate structure contains a constant number of triangles and each of these triangles has a side length of $O(\sqrt{n})$. The algorithm can place a triangle using a constant number of applications of the movement primitives for triangles. So according to Lemma 6.2, moving the triangles takes $O(\sqrt{n})$ rounds. The number of rows removed from the triangles is constant and each row has length $O(\sqrt{n})$. The algorithm moves a row a constant number of times along a distance of $O(\sqrt{n})$. So according to Lemma 6.1, moving the pruned rows requires $O(\sqrt{n})$ rounds. Overall, the algorithm requires $O(\sqrt{n})$ rounds to construct the shape under these simplified conditions.

6.4.2 Full Algorithm

We now show how to extend the simplified algorithm to handle the difficulties described at the beginning of this section. First, we address the two special cases concerning the placement of triangles we briefly mentioned above. As depicted in Figure 6.10, when placing a triangle of side length $\ell - 1$, a vertex of the triangle might already be present in the current shape. In this case, we prune this vertex from the triangle while it is still part of the intermediate structure. We then split the triangle into a smaller triangle of side length $\ell - 2$ and a particle chain of length $\ell - 2$. These two parts are sequentially moved to their designated position once the corresponding face of the shape has to be constructed.

The second special case occurs when a triangle of side length $\ell - 1$ has to

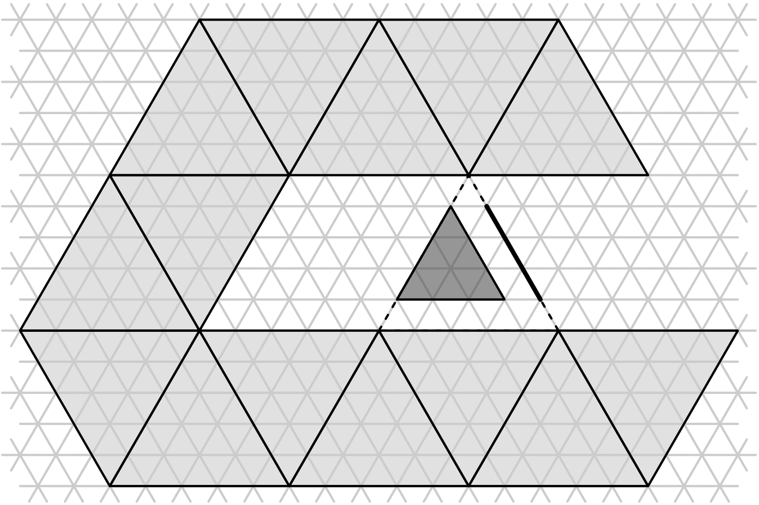

Figure 6.10: Placing a triangle of side length $\ell - 1$ with a vertex already present in the shape.

be moved through an already constructed part of the shape that can only accommodate triangles of side length $\ell - 2$ (e.g., a shape formed like a bottle, with a narrow tunnel and larger inner area). In this case, we split the triangle of side length $\ell - 1$ into a smaller triangle of side length $\ell - 2$ and a particle chain of length $\ell - 1$. As in the first case, we sequentially move these parts to their respective position once the construction reaches the corresponding face.

Our solution to the remaining difficulties requires us to modify the given shape S. Specifically, we define S' to be the shape resulting from subdividing each face in S into four faces, each forming an equilateral triangle. It is not hard to see that a representation of S' is also a representation of S. We can construct S' by sequentially placing groups of four triangles that correspond to the faces in S. This shows that if S is constructible also S' is constructible.

The intermediate structure can have a remainder of particles at its end that does not form a triangle of side length ℓ but that has to be incorporated into the representation of the shape. Since the structure of the remainder might be such that it cannot easily be moved using the movement primitives described in Section 6.2, our goal is to incorporate the remainder without moving it. To this end, the algorithm rotates and translates the shape S such that the first two faces of S (according to the computed permutation of the faces) are aligned with the remainder at the end of the intermediate structure as depicted in Figure 6.11. The algorithm then constructs a representation of the two faces that incorporates the remainder as shown in the figure.

Let $s' = 4s$ be the number of faces in S'. Constructing the first two faces of

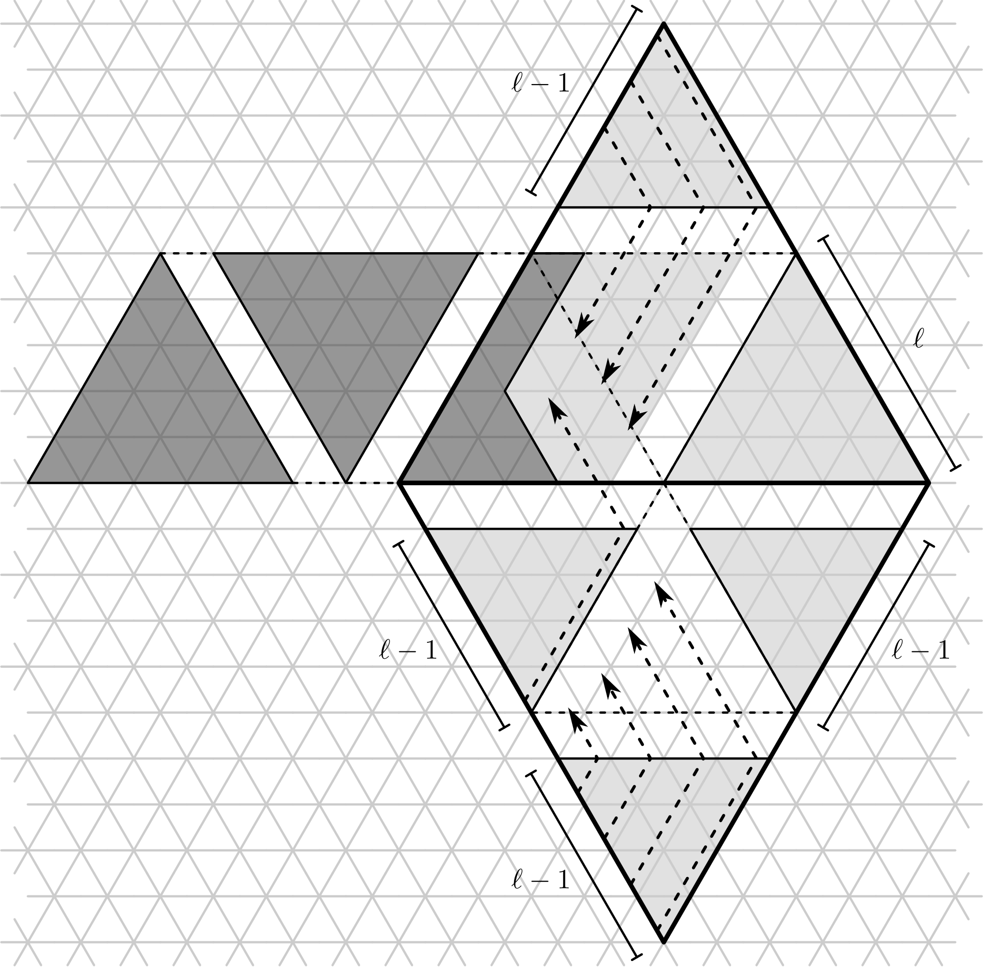

Figure 6.11: Incorporating the remainder into the first two faces (bold outline) of S. The end of the intermediate structure is shown in dark gray. The triangles placed around the remainder are shown in light gray. The area containing the remainder is also shown in light gray. The unoccupied nodes in this area are filled by placing triangles of side length $\ell - 1$ above and below the remainder and expanding them as indicated by the dashed arrows. This expansion can require *all* particles of a triangles to expand, i.e., also the particles occupying the line L_1 in Figure 6.4 including the coordinator might have to expand. The rest of the two faces is constructed from a triangle of side length ℓ and two triangles of side length $\ell - 1$, one of which is expanded as indicated. Note that, depending on the shape of the remainder, it can be necessary to temporarily place triangles around this construction to guarantee connectivity.

S in the way depicted in Figure 6.11 uses five triangles to represent eight faces of S'. According to Theorem 6.8, the number of triangles in the intermediate structure (which we now construct on the basis of the value s' instead of s) satisfies $\Delta' \leq s' - 3$. This implies that the construction presented in Figure 6.11 cannot cause a surplus of triangles.

Next, we address the problem of incorporating the pruned rows and vertices back into the structure. Consider the representation of the first two faces of S shown in Figure 6.11. The bottom face shown in the figure contains an expanded triangle. We incorporate the pruned particles into the representation by selectively contracting rows of this expanded triangle and moving the pruned particles into the space opened up in this way using particle chain movement. Since the number of pruned particles is bounded by $O(\sqrt{n})$ and the expanded triangle can make space for an additional $\Omega(n)$ particles, all pruned particles can be accommodated.

Finally, it remains to show how S' can be constructed when the number of triangles in the intermediate structure is lower than $s' - 3$. Consider a face in S other than the initial two faces. The face corresponds to four faces in S'. By choosing a pair of neighboring faces among these four faces and representing it by an expanded triangle, we can construct all four faces using only three triangles. Constructing the first two faces in S using five triangles as shown in Figure 6.11 and constructing each of the remaining $s - 2$ faces using three triangles as described above implies that overall we require δ triangles to construct S' where

$$\delta = 3(s-2) + 5 = 3\left(\frac{s'}{4} - 2\right) + 5 = \frac{3}{4} \cdot s' - 1 \leq \frac{3}{4} \cdot s'.$$

According to Theorem 6.8, the intermediate structure contains Δ' triangles where

$$\Delta' \geq \frac{3}{4} \cdot s' \geq \delta.$$

Therefore, the number of triangles in the intermediate structure is sufficient to construct S'. By selectively representing faces in S using either three or four triangles, the algorithm can construct the shape using exactly Δ' triangles.

Note that constructing S' instead of S does not increase the asymptotic running time of the algorithm. Furthermore, it is not hard to see that the techniques for overcoming the various difficulties do not invalidate the arguments about the running time we gave for the simplified algorithm. Combining these arguments with the statement of Theorem 6.8 implies the following theorem.

Theorem 6.9. *The algorithm solves the shape formation problem in $O(\sqrt{n})$ rounds, w.h.p.*

We conclude this section with a theorem that shows that the running time of the presented algorithm is optimal.

Theorem 6.10. *Let S be any shape except for a triangle. Any algorithm requires $\Omega(\sqrt{n})$ rounds to construct a representation of S.*

Proof. Note that an algorithm cannot control the activation order of the particles. Thus, every particle moves only a distance of at most $O(1)$ per round in the worst case. If S is not a triangle then there exists a particle that has to be moved by distance of $\Omega(\sqrt{n})$ in order to form a representation of S. Therefore, any algorithm needs at least $\Omega(\sqrt{n})$ rounds to solve the shape formation problem for S in the worst case. □

6.5 Outlook

There are several ways in which the shape formation algorithm presented in this chapter could be extended. As we already mentioned in Section 6.1, it is not hard to remove the assumption that the number of particles in the given particle system is a triangular number: Suppose that the particles are initially organized into a triangle such that the last row of the triangle is not necessarily completely filled with particles. Note that there are at most $O(\sqrt{n})$ particles in the last row and the leader can determine via a token passing scheme whether the last row is complete. While the intermediate structure is built, the incomplete row is moved out of the way using particle chain movement such that it does not interfere with the construction process. During the construction of the representation of the given shape, we move the incomplete row together with the intermediate structure when necessary. We incorporate the incomplete row into the representation in the same way we incorporated the pruned rows and vertices, i.e., we move it into the area filled by an expanded triangle that was used to construct one of the first two faces of the shape. During the entire algorithm, the incomplete row has to move a distance of $O(\sqrt{n})$ so that the given bound on the running time remains intact.

Beyond this simple extension of the algorithm, it would be interesting to investigate whether a similar running time can be achieved when the initial distribution of the particles is not a triangle. Certainly, if the initial distribution has a diameter of $\Omega(n)$, shape formation takes $\Omega(n)$ rounds since the representation of a shape consisting of a constant number of equilateral triangles has a diameter of $O(\sqrt{n})$. Therefore, it seems reasonable to restrict the diameter of the initial particle distribution to $O(\sqrt{n})$ and to ask whether starting from such a distribution, a given shape can be constructed in $O(\sqrt{n})$ rounds.

Another promising direction for future research is the construction of more complex shapes. For example, one could consider shapes consisting of a non-constant number of triangles. The description of such a shape could be spread across multiple particles. More generally, one could encode a shape as a Turing-machine program, which is an approach used in other related models (see, e.g., [Mic15]). In fact, Di Luna et al. already mentioned such an approach for the amoebot model in a recent brief announcement [Di +17]. A conference version of this work was accepted at the 21st International Conference on Principles of Distributed Systems and should appear soon.

Finally, it would be desirable to restore the original result of the publication underlying this chapter by lifting the requirement that the given shape has to be sequentially constructible. We conjecture that subdividing the triangular faces of a shape that is not sequentially constructible into smaller triangles might result in a new shape that is sequentially constructible. However, we leave the investigation of this conjecture for future work.

From a broader perspective that goes beyond the contents of this thesis, there is a plethora of problems that can be investigated under the amoebot model. As we discussed in Section 5.2, we already presented an algorithm for the *coating* problem in which a given object has to be covered by even layers of particles (see [Der+17; Der+16a]). Other authors considered problems such as *compression* [Can+16] and *shortcut bridging* [Arr+17]. In future research, one could investigate problems such as the construction of the *convex hull* of a given object or the *collective transport* problem from swarm robotics in which a set of particles has to collaborate in order to transport a given object to another location. A particularly interesting aspect that has not yet been investigated under the amoebot model is *exploration*. Consider, for example, the following simple exploration problem: We are given two static objects that are separated by a certain distance and a particle system that is initially connected to only one of these objects. The particle system has to explore the surrounding area without disconnecting from its object in order to find the second object as quickly as possible.

When looking at the subject of programmable matter from an even broader perspective, it becomes apparent that there are numerous algorithmic challenges that have to be overcome before programmable matter can become a reality. One of the most obvious challenges is the fact that physical space is *three-dimensional* while the amoebot model only considers two dimensions. The problem of extending the amoebot model to three dimensions raises a number of questions. For example, one would have to determine a suitable underlying graph to replace the equilateral triangular graph. A possible approach to this problem is to ask what three-dimensional shape a physical particle should have. Naturally, the cube would be an obvious candidate. However, the *rhombic dodecahedron*, which is also a space-filling polyhedron, might be a better choice since it is more closely related to the geometry underlying the two-dimensional amoebot model. Aside from finding a suitable model, one would have to investigate in what capacity the existing algorithms can be generalized to three dimensions or whether entirely new algorithms are necessary.

Another important issue one has to address in order to bring programmable matter closer to reality is recovery from *faults*. In systems consisting of thousands or even millions of tiny computational entities, faults will most likely be unavoidable. The potentially enormous scale of these systems presumably prohibits dealing with faults manually, i.e., by outside human intervention. We therefore need models and algorithms that inherently consider the possibility of unresponsive, faulty, or even byzantine particles and automatically deal with these problems when they arise. Faulty particles could be detected, deactivated, and extracted from the particle system by other particles. Alternatively, one could envision having a certain degree of redundancy in a particle system that allows it to operate even when a certain fraction of the particles is unresponsive.

Finally, it would be highly desirable to have an ongoing dialog between researchers working on the algorithmic foundations of programmable matter and those working on its physical implementation in the form of robotic and biological systems. Results and advancements in the practical field should influence the models used by theoreticians. In turn, theoretical results might inspire new approaches and solutions for the physical implementation of programmable matter. One aspect of programmable matter for which interdisciplinary research is of crucial importance is the choice of the *locomotion primitive*: The amoebot model achieves locomotion by allowing particles to expand and contract, which is a very *local* movement. It might be reasonable to implement *non-local* movements by allowing particles to carry or otherwise move other particles within reasonable limits. This might then allow for much more efficient algorithms for problems such as shape formation. Contrasting the algorithmic power of locomotion primitives with their practical feasibility is a task that requires contributions from both theoretical and applied research alike.

Bibliography

[Abr+03] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov. “A Generic Scheme for Building Overlay Networks in Adversarial Scenarios”. In: *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*. Nice, France, 2003. DOI: `10.1109/IPDPS.2003.1213125`.

[Adl94] L. M. Adleman. “Molecular Computation of Solutions to Combinatorial Problems”. In: *Science* 266.5187 (1994), pp. 1021–1024. DOI: `10.1126/science.7973651`.

[AE07] R. Ananthakrishnan and A. Ehrlicher. “The Forces Behind Cell Movement”. In: *International Journal of Biological Sciences* 3.5 (2007), pp. 303–317. DOI: `10.7150/ijbs.3.303`.

[AG15] D. Alistarh and R. Gelashvili. “Polylogarithmic-Time Leader Election in Population Protocols”. In: *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP)*. Kyoto, Japan, 2015, pp. 479–491. DOI: `10.1007/978-3-662-47666-6_38`.

[AGM13] C. Agathangelou, C. Georgiou, and M. Mavronicolas. “A Distributed Algorithm for Gathering Many Fat Mobile Robots in the Plane”. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. Montreal, Canada, 2013, pp. 250–259. DOI: `10.1145/2484239.2484266`.

[Ahm+15] M. Ahmadi, A. Ghodselahi, F. Kuhn, and A. R. Molla. “The Cost of Global Broadcast in Dynamic Radio Networks”. In: *Proceedings of the 19th International Conference on Principles of Distributed Systems, (OPODIS)*. Rennes, France, 2015, pp. 7:1–7:17. DOI: `10.4230/LIPIcs.OPODIS.2015.7`.

[AK93] S. Aggarwal and S. Kutten. “Time Optimal Self-Stabilizing Spanning Tree Algorithms”. In: *Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Bombay, India, 1993, pp. 400–410. DOI: `10.1007/3-540-57529-4_72`.

[AM14] S. Abshoff and F. Meyer auf der Heide. "Continuous Aggregation in Dynamic Ad-Hoc Networks". In: *Proceedings of the 21st International Colloquium on Structural Information and Communication Complexity (SIROCCO).* Takayama, Japan, 2014, pp. 194–209. DOI: 10.1007/978-3-319-09620-9_16.

[Ang+05] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. "Fast Construction of Overlay Networks". In: *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 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.

[APR16] J. Augustine, G. Pandurangan, and P. Robinson. "Distributed Algorithmic Foundations of Dynamic Networks". In: *SIGACT News* 47.1 (2016), pp. 69–98. DOI: 10.1145/2902945.2902959.

[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: *Autonomous Robots* 28.2 (2010), pp. 197–211. DOI: 10.1007/s10514-009-9162-7.

[Arr+17] M. A. Arroyo, S. Cannon, J. J. Daymude, D. Randall, and A. W. Richa. "A Stochastic Approach to Shortcut Bridging in Programmable Matter". In: *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming (DNA).* Austin, Texas, USA, 2017, pp. 122–138. DOI: 10.1007/978-3-319-66799-7_9.

[AS07a] B. Awerbuch and C. Scheideler. "A Denial-of-Service Resistant DHT". In: *Proceedings of the 21st International Symposium on Distributed Computing (DISC).* Lemesos, Cyprus, 2007, pp. 33–47. DOI: 10.1007/978-3-540-75142-7_6.

[AS07b] B. Awerbuch and C. Scheideler. "Towards Scalable and Robust Overlay Networks". In: *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS).* Bellevue, Washington, USA, 2007.

[Aug+15] J. Augustine, G. Pandurangan, P. Robinson, S. T. Roche, and E. Upfal. "Enabling Robust and Efficient Distributed Computation in Dynamic Peer-to-Peer Networks". In: *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS).*

Berkeley, California, USA, 2015, pp. 350–369. DOI: 10.1109/FOCS.2015.29.

[AV84] M. J. Atallah and U. Vishkin. “Finding Euler Tours in Parallel”. In: *Journal of Computer and System Sciences* 29.3 (1984), pp. 330–337. DOI: 10.1016/0022-0000(84)90003-5.

[AW04] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. 2nd Edition. John Wiley and Sons, 2004.

[AW09] J. Aspnes and U. Wieder. “The expansion and mixing time of skip graphs with applications”. In: *Distributed Computing* 21.6 (2009), pp. 385–393. DOI: 10.1007/s00446-008-0071-3.

[Ben+01] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. “Programmable and autonomous computing machine made of biomolecules”. In: *Nature* 414 (2001), pp. 430–434. DOI: 10.1038/35106533.

[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.

[Bha+13] A. Bhattacharyya, M. Braverman, B. Chazelle, and H. L. Nguyen. “On the Convergence of the Hegselmann-Krause System”. In: *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS)*. Berkeley, California, USA, 2013, pp. 61–66. DOI: 10.1145/2422436.2422446.

[BKM14] P. Berenbrink, B. Krayenhoff, and F. Mallmann-Trenn. “Estimating the number of connected components in sublinear time”. In: *Information Processing Letters* 114.11 (2014), pp. 639–642. DOI: 10.1016/j.ipl.2014.05.008.

[BMV12] V. Bonifaci, K. Mehlhorn, and G. Varma. “Physarum Can Compute Shortest Paths”. In: *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Kyoto, Japan, 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.

[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)*. Chicago, Illinois, USA, 2016, pp. 279–288. DOI: 10.1145/2933057.2933107.

[CDH09] C. Cooper, M. E. Dyer, and A. J. Handley. “The Flip Markov Chain and a Randomising P2P Protocol”. In: *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Calgary, Canada, 2009, pp. 141–150. DOI: 10.1145/1582716.1582742.

[Cen+15] K. Censor-Hillel, P. Kaski, J. H. Korhonen, C. Lenzen, A. Paz, and J. Suomela. “Algebraic Methods in the Congested Clique”. In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*. Donostia-San Sebastián, Spain, 2015, pp. 143–152. DOI: 10.1145/2767386.2767414.

[CF05] C. Cramer and T. Fuhrmann. *Self-Stabilizing Ring Networks on Connected Graphs*. Tech. rep. University of Karlsruhe, 2005.

[Cha09] B. Chazelle. “Natural Algorithms”. In: *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New York City, New York, USA, 2009, pp. 422–431.

[Che+14] H.-L. Chen, D. Doty, D. Holden, C. Thachuk, D. Woods, and C.-T. Yang. “Fast Algorithmic Self-assembly of Simple Shapes Using Random Agitation”. In: *Proceedings of the 20th International Conference on DNA Computing and Molecular Programming (DNA)*. Kyoto, Japan, 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 IEEE International Conference on Robotics and Automation (ICRA)*. San Diego, California, USA, 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.

[CNS12] T. Clouser, M. Nesterenko, and C. Scheideler. “Tiara: A self-stabilizing deterministic skip list and skip graph”. In: *Theoretical Computer Science* 428 (2012), pp. 18–35. DOI: 10.1016/j.tcs.2011.12.079.

[CP08] R. Cohen and D. Peleg. "Local spreading algorithms for autonomous robot systems". In: *Theoretical Computer Science* 399.1-2 (2008), pp. 71–82. DOI: `10.1016/j.tcs.2008.02.007`.

[CRT05] B. Chazelle, R. Rubinfeld, and L. Trevisan. "Approximating the Minimum Spanning Tree Weight in Sublinear Time". In: *SIAM Journal on Computing* 34.6 (2005), pp. 1370–1379. DOI: `10.1137/S0097539702403244`.

[CS09] A. Czumaj and C. Sohler. "Estimating the Weight of Metric Minimum Spanning Trees in Sublinear Time". In: *SIAM Journal on Computing* 39.3 (2009), pp. 904–922. DOI: `10.1137/060672121`.

[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`.

[Czu+05] A. Czumaj, F. Ergün, L. Fortnow, A. Magen, I. Newman, R. Rubinfeld, and C. Sohler. "Approximating the Weight of the Euclidean Minimum Spanning Tree in Sublinear Time". In: *SIAM Journal on Computing* 35.1 (2005), pp. 91–109. DOI: `10.1137/S0097539703435297`.

[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).* Zürich, Switzerland, 2010, pp. 267–276. DOI: `10.1145/1835698.1835761`.

[Das+17] S. Das, G. A. Di Luna, P. Flocchini, N. Santoro, and G. Viglietta. "Mediated Population Protocols: Leader Election and Applications". In: *Proceedings of the 14th Annual Conference on Theory and Applications of Models of Computation (TAMC).* Bern, Switzerland, 2017, pp. 172–186. DOI: `10.1007/978-3-319-55911-7_13`.

[Day+17] J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. "Improved Leader Election for Self-Organizing Programmable Matter". In: *Proceedings of the 13th International Symposium on Algorithms and Experiments for Wireless Networks (ALGOSENSORS).* To appear. Vienna, Austria, 2017.

[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: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*

(SPAA). Prague, Czech Republic, 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 2nd Annual International Conference on Nanoscale Computing and Communication (NANOCOM)*. Boston, Massachusetts, USA, 2015, pp. 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: *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming (DNA)*. Boston and Cambridge, Massachusetts, USA, 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: *Proceedings of the 22nd International Conference on DNA Computing and Molecular Programming (DNA)*. Munich, Germany, 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 Parallelism in Algorithms and Architectures (SPAA)*. Asilomar State Beach/Pacific Grove, California, USA, 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.

[DGS16] M. Drees, R. Gmyr, and C. Scheideler. “Churn- and DoS-resistant Overlay Networks Based on Network Reconfiguration”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Asilomar State Beach/Pacific Grove, California, USA, 2016, pp. 417–427. DOI: 10.1145/2935764.2935783.

[Di +17] G. A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. “Brief Announcement: Shape Formation by Programmable Particles”. In: *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*. Vienna, Austria, 2017, pp. 48:1–48:3. DOI: 10.4230/LIPIcs.DISC.2017.48.

[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.

[DKO14] A. Drucker, F. Kuhn, and R. Oshman. “On the Power of the Congested Clique Model”. In: *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC).* Paris, France, 2014, pp. 367–376. DOI: `10.1145/2611462.2611493`.

[DMS04] R. Dingledine, N. Mathewson, and P. F. Syverson. “Tor: The Second-Generation Onion Router”. In: *Proceedings of the 13th USENIX Security Symposium (SSYM).* San Diego, California, USA, 2004, pp. 303–320.

[Dot12] D. Doty. “Theory of Algorithmic Self-Assembly”. In: *Communications of the ACM* 55.12 (2012), pp. 78–88. DOI: `10.1145/2380656.2380675`.

[Dou02] J. R. Douceur. “The Sybil Attack”. In: *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS).* Cambridge, Massachusetts, USA, 2002, pp. 251–260. DOI: `10.1007/3-540-45748-8_24`.

[DS15] D. Doty and D. Soloveichik. “Stable Leader Election in Population Protocols Requires Linear Time”. In: *Proceedings of the 29th International Symposium on Distributed Computing (DISC).* Tokyo, Japan, 2015, pp. 602–616. DOI: `10.1007/978-3-662-48653-5_40`.

[Dut+13] C. Dutta, G. Pandurangan, R. Rajaraman, Z. Sun, and E. Viola. “On the Complexity of Information Spreading in Dynamic Networks”. In: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* New Orleans, Louisiana, USA, 2013, pp. 717–736. DOI: `10.1137/1.9781611973105.52`.

[Elk04] M. Elkin. “Unconditional Lower Bounds on the Time-Approximation Tradeoffs for the Distributed Minimum Spanning Tree Problem”. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC).* Chicago, Illinois, USA, 2004, pp. 331–340. DOI: `10.1145/1007352.1007407`.

[Elk06] M. Elkin. “A faster distributed protocol for constructing a minimum spanning tree”. In: *Journal of Computer and System Sciences* 72.8 (2006), pp. 1282–1308. DOI: `10.1016/j.jcss.2006.07.002`.

[ES15] M. Eikel and C. Scheideler. “IRIS: A Robust Information System Against Insider DoS Attacks”. In: *ACM Transactions on Parallel Computing* 2.3 (2015), pp. 18:1–18:33. DOI: 10.1145/2809806.

[ESS14] M. Eikel, C. Scheideler, and A. Setzer. “RoBuSt: A Crash-Failure-Resistant Distributed Storage System”. In: *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*. Cortina d’Ampezzo, Italy, 2014, pp. 107–122. DOI: 10.1007/978-3-319-14472-6_8.

[Fed+06] T. Feder, A. Guetz, M. Mihail, and A. Saberi. “A Local Switch Markov Chain on Given Degree Graphs with Application in Connectivity of Peer-to-Peer Networks”. In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Berkeley, California, USA, 2006, pp. 69–76. DOI: 10.1109/FOCS.2006.5.

[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.

[For+14] D. Foreback, A. Koutsopoulos, M. Nesterenko, C. Scheideler, and T. Strothmann. “On Stabilizing Departures in Overlay Networks”. In: *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Paderborn, Germany, 2014, pp. 48–62. DOI: 10.1007/978-3-319-11764-5_4.

[Fri+09] A. E. Friedland, T. K. Lu, X. Wang, D. Shi, G. Church, and J. J. Collins. “Synthetic Gene Networks That Count”. In: *Science* 324.5931 (2009), pp. 1199–1202. DOI: 10.1126/science.1172005.

[Fri08] J. Friedman. “A Proof of Alon’s Second Eigenvalue Conjecture and Related Problems”. In: *Memoirs of the AMS* 195.910 (2008). DOI: 10.1090/memo/0910.

[Gär03] F. C. Gärtner. *A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms*. Tech. rep. Swiss Federal Institute of Technology (EPFL), 2003.

[GCM05] S. C. Goldstein, J. Campbell, and T. C. Mowry. “Programmable Matter”. In: *IEEE Computer* 38.6 (2005), pp. 99–101. DOI: 10.1109/MC.2005.198.

[GDT14] T. F. Gonzalez, J. Diaz-Herrera, and A. Tucker, eds. *Computing Handbook, Third Edition: Computer Science and Software Engineering*. CRC Press, 2014.

[GLS16] R. Gmyr, J. Lefèvre, and C. Scheideler. “Self-stabilizing Metric Graphs”. In: *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Lyon, France, 2016, pp. 248–262. DOI: `10.1007/978-3-319-49259-9_20`.

[GLS17] R. Gmyr, J. Lefèvre, and C. Scheideler. “Self-stabilizing Metric Graphs”. In: *Theory of Computing Systems* (2017). To appear.

[Gmy+17] R. Gmyr, K. Hinnenthal, C. Scheideler, and C. Sohler. “Distributed Monitoring of Network Properties: The Power of Hybrid Networks”. In: *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*. Warsaw, Poland, 2017, pp. 137:1–137:15. DOI: `10.4230/LIPIcs.ICALP.2017.137`.

[Gon01] L. Gong. “Industry Report: JXTA: A Network Programming Environment”. In: *IEEE Internet Computing* 5.3 (2001), pp. 88–95. DOI: `10.1109/4236.935182`.

[Heg+15] J. W. Hegeman, G. Pandurangan, S. V. Pemmaraju, V. B. Sardeshmukh, and M. Scquizzato. “Toward Optimal Bounds in the Congested Clique: Graph Connectivity and MST”. In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed (PODC)*. Donostia-San Sebastián, Spain, 2015, pp. 91–100. DOI: `10.1145/2767386.2767434`.

[HK11] B. Haeupler and D. R. Karger. “Faster Information Dissemination in Dynamic Networks via Network Coding”. In: *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. San Jose, California, USA, 2011, pp. 381–390. DOI: `10.1145/1993806.1993885`.

[Hsi+02] T.-R. Hsiang, E. M. Arkin, M. A. Bender, S. P. Fekete, and J. S. B. Mitchell. “Algorithms for Rapidly Dispersing Robot Swarms in Unknown Environments”. In: *Algorithmic Foundations of Robotics V, Selected Contributions of the Fifth International Workshop on the Algorithmic Foundations of Robotics (WAFR)*. Nice, France, 2002, pp. 77–94. DOI: `10.1007/978-3-540-45058-0_6`.

[HST12] T. P. Hayes, J. Saia, and A. Trehan. “The Forgiving Graph: a distributed data structure for low stretch under adversarial attack”. In: *Distributed Computing* 25.4 (2012), pp. 261–278. DOI: `10.1007/s00446-012-0160-1`.

[HZ01] S. Halperin and U. Zwick. “Optimal Randomized EREW PRAM Algorithms for Finding Spanning Forests”. In: *Journal of Algorithms* 39.1 (2001), pp. 1–46. DOI: `10.1006/jagm.2000.1146`.

[IR90] A. Itai and M. Rodeh. "Symmetry Breaking in Distributed Networks". In: *Information and Computation* 88.1 (1990), pp. 60–87. DOI: `10.1016/0890-5401(90)90004-2`.

[Jac+09] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid, and H. Täubig. "A distributed polylogarithmic time algorithm for self-stabilizing skip graphs". In: *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*. Calgary, Canada, 2009, pp. 131–140. DOI: `10.1145/1582716.1582741`.

[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`.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[JM95] D. B. Johnson and P. T. Metaxas. "A Parallel Algorithm for Computing Minimum Spanning Trees". In: *Journal of Algorithms* 19.3 (1995), pp. 383–401. DOI: `10.1006/jagm.1995.1043`.

[JP13] T. Jacobs and G. Pandurangan. "Stochastic Analysis of a Churn-Tolerant Structured Peer-to-Peer Scheme". In: *Peer-to-Peer Networking and Applications* 6.1 (2013), pp. 1–14. DOI: `10.1007/s12083-012-0124-z`.

[Ker13] S. Kernbach, ed. *Handbook of Collective Robotics: Fundamentals and Challanges*. Pan Stanford Publishing, 2013.

[KG89] J. M. Keil and C. A. Gutwin. "The Delaunay triangulation closely approximates the complete Euclidean graph". In: *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*. Ottawa, Canada, 1989, pp. 47–56. DOI: `10.1007/3-540-51542-9_6`.

[KG92] J. M. Keil and C. A. Gutwin. "Classes of Graphs Which Approximate the Complete Euclidean Graph". In: *Discrete & Computational Geometry* 7 (1992), pp. 13–28. DOI: `10.1007/BF02187821`.

[KKS14] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. "Re-Chord: A Self-stabilizing Chord Overlay Network". In: *Theory of Computing Systems* 55.3 (2014), pp. 591–612. DOI: `10.1007/s00224-012-9431-2`.

[KLO10] F. Kuhn, N. A. Lynch, and R. Oshman. "Distributed Computation in Dynamic Networks". In: *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*. Cambridge, Massachusetts, USA, 2010, pp. 513–522. DOI: `10.1145/1806689.1806760`.

[KLS08] F. Kuhn, T. Locher, and S. Schmid. “Distributed Computation of the Mode”. In: *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC)*. Toronto, Canada, 2008, pp. 15–24. DOI: 10.1145/1400751.1400756.

[KLW07] F. Kuhn, T. Locher, and R. Wattenhofer. “Tight Bounds for Distributed Selection”. In: *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. San Diego, California, USA, 2007, pp. 145–153. DOI: 10.1145/1248377.1248401.

[KMR02] A. D. Keromytis, V. Misra, and D. Rubenstein. “SOS: Secure Overlay Services”. In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. Pittsburgh, Pennsylvania, USA, 2002, pp. 61–72. DOI: 10.1145/633025.633032.

[KSW05] F. Kuhn, S. Schmid, and R. Wattenhofer. “A Self-repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn”. In: *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*. Ithaca, New York, USA, 2005, pp. 13–23. DOI: 10.1007/11558989_2.

[KT13] J. Kleinberg and E. Tardos. *Algorithm Design: Pearson New International Edition*. Pearson Education Limited, 2013.

[Len13] C. Lenzen. “Optimal Deterministic Routing and Sorting on the Congested Clique”. In: *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*. Montreal, Canada, 2013, pp. 42–50. DOI: 10.1145/2484239.2501983.

[Li+10] K. Li, K. Thomas, C. E. Torres, L. F. Rossi, and C.-C. Shen. “Slime Mold Inspired Path Formation Protocol for Wireless Sensor Networks”. In: *Proceedings of the 7th International Conference on Swarm Intelligence (ANTS)*. Brussels, Belgium, 2010, pp. 299–311. DOI: 10.1007/978-3-642-15461-4_26.

[Loc09] T. Locher. “Foundations of Aggregation and Synchronization in Distributed Systems”. PhD thesis. 2009.

[Lot+05] Z. Lotker, B. Patt-Shamir, E. Pavlov, and D. Peleg. “Minimum-Weight Spanning Tree Construction in $O(\log\log n)$ Communication Rounds”. In: *SIAM Journal on Computing* 35.1 (2005), pp. 120–131. DOI: 10.1137/S0097539704441848.

[Lov93] L. Lovász. “Random Walks on Graphs: A Survey”. In: *Combinatorics, Paul Erdős is Eighty*. Ed. by D. Miklós, V. T. Sós, and T. Szőnyi. Vol. 2. János Bolyai Mathematical Society, 1993.

[LS03] C. Law and K.-Y. Siu. "Distributed Construction of Random Expander Networks". In: *Proceedings of the 22nd IEEE International Conference on Computer Communications (INFOCOM)*. San Franciso, California, USA, 2003, pp. 2133–2143. DOI: 10.1109/INFCOM.2003.1209234.

[McL08] J. D. McLurkin. "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)*. Donostia-San Sebastián, Spain, 2015, pp. 37–46. DOI: 10.1145/2767386.2767402.

[MS06] P. Mahlmann and C. Schindelhauer. "Distributed Random Digraph Transformations for Peer-to-Peer Networks". In: *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Cambridge, Massachusetts, USA, 2006, pp. 308–317. DOI: 10.1145/1148109.1148162.

[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.

[New01] M. E. J. Newman. "The structure of scientific collaboration networks". In: *Proceedings of the National Academy of Sciences* 98.2 (2001), pp. 404–409. DOI: 10.1073/pnas.98.2.404.

[NSW01] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. "Random graphs with arbitrary degree distributions and their applications". In: *Physical Review E* 64 (2 2001), pp. 026118-1–026118-17. DOI: 10.1103/PhysRevE.64.026118.

[NW07] M. Naor and U. Wieder. "Novel Architectures for P2P Applications: The Continuous-Discrete Approach". In: *ACM Transactions on Algorithms* 3.3 (2007). DOI: 10.1145/1273340.1273350.

[NWS02] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. "Random graph models of social networks". In: *Proceedings of the National Academy of Sciences* 99 (2002), pp. 2566–2572. DOI: 10.1073/pnas.012582999.

[NYT00] T. Nakagaki, H. Yamada, and Á. Tóth. "Maze-solving by an amoeboid organism". In: *Nature* 407 (2000), p. 470. DOI: 10.1038/35035159.

[OR99] M. Ogihara and A. Ray. “Simulating Boolean Circuits on a DNA Computer”. In: *Algorithmica* 25.2-3 (1999), pp. 239–250. DOI: 10.1007/PL00008276.

[ORS07] M. Onus, A. W. Richa, and C. Scheideler. “Linearization: Locally Self-Stabilizing Sorting in Graphs”. In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. New Orleans, Louisiana, USA, 2007, pp. 99–108. 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.

[PR99] D. Peleg and V. Rubinovich. “A Near-Tight Lower Bound on the Time Complexity of Distributed MST Construction”. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*. New York City, New York, USA, 1999, pp. 253–261. DOI: 10.1109/SFFCS.1999.814597.

[PRS16] G. Pandurangan, P. Robinson, and M. Scquizzato. “Fast Distributed Algorithms for Connectivity and MST in Large Graphs”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Asilomar State Beach/Pacific Grove, California, USA, 2016, pp. 429–438. DOI: 10.1145/2935764.2935785.

[PRT16] G. Pandurangan, P. Robinson, and A. Trehan. “DEX: self-healing expanders”. In: *Distributed Computing* 29.3 (2016), pp. 163–185. DOI: 10.1007/s00446-015-0258-3.

[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.

[RD01] A. I. T. Rowstron and P. Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Heidelberg, Germany, 2001, pp. 329–350. DOI: 10.1007/3-540-45518-3_18.

[Rot06] P. W. K. Rothemund. “Folding DNA to create nanoscale shapes and patterns”. In: *Nature* 440 (2006), pp. 297–302. DOI: 10.1038/nature04586.

[RSS11] A. W. Richa, C. Scheideler, and P. Stevens. "Self-Stabilizing De Bruijn Networks". In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Grenoble, France, 2011, pp. 416–430. DOI: 10.1007/978-3-642-24550-3_31.

[Sar+13] A. Das Sarma, D. Nanongkai, G. Pandurangan, and P. Tetali. "Distributed Random Walks". In: *Journal of the ACM* 60.1 (2013), pp. 2:1–2:31. DOI: 10.1145/2432622.2432624.

[Sch08] J. L. Schiff. *Cellular Automata: A Discrete View of the World.* John Wiley & Sons, 2008.

[Sin+06] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. "Eclipse Attacks on Overlay Networks: Threats and Defenses". In: *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM)*. Barcelona, Spain, 2006. DOI: 10.1109/INFOCOM.2006.231.

[SNP09] A. Das Sarma, D. Nanongkai, and G. Pandurangan. "Fast Distributed Random Walks". In: *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Calgary, Canada, 2009, pp. 161–170. DOI: 10.1145/1582716.1582745.

[SR05] A. Shaker and D. S. Reeves. "Self-Stabilizing Structured Ring Topology P2P Systems". In: *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P)*. Konstanz, Germany, 2005, pp. 39–46. DOI: 10.1109/P2P.2005.34.

[ST08] J. Saia and A. Trehan. "Picking up the Pieces: Self-Healing in Reconfigurable Networks". In: *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. Miami, Florida, USA, 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536326.

[Sut+13] K. Suto, H. Nishiyama, N. Kato, T. Nakachi, T. Fujii, and A. Takahara. "THUP: A P2P Network Robust to Churn and DoS Attack Based on Bimodal Degree Distribution". In: *IEEE Journal on Selected Areas in Communications* 31.9-Supplement (2013), pp. 247–256. DOI: 10.1109/JSAC.2013.SUP.0513022.

[TM91] T. Toffoli and N. Margolus. "Programmable Matter: Concepts and Realization". In: *Physica D: Nonlinear Phenomena* 47.1 (1991), pp. 263–272. DOI: 10.1016/0167-2789(91)90296-L.

[Tra+03] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.-C. Hugly, E. Pouyoul, and B. Yeager. *Project JXTA 2.0 Super-Peer Virtual Network.* 2003.

[TV85] R. E. Tarjan and U. Vishkin. "An Efficient Parallel Biconnectivity Algorithm". In: *SIAM Journal on Computing* 14.4 (1985), pp. 862–874. DOI: 10.1137/0214061.

[Win98] E. Winfree. "Algorithmic Self-Assembly of DNA". PhD thesis. 1998.

[Woo+13] D. Woods, H.-L. Chen, S. Goodfriend, N. Dabby, E. Winfree, and P. Yin. "Active Self-Assembly of Algorithmic Shapes and Patterns in Polylogarithmic Time". In: *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS).* Berkeley, California, USA, 2013, pp. 353–354. DOI: 10.1145/2422436.2422476.

[Woo13] D. Woods. "Intrinsic universality and the computational power of self-assembly". In: *Proceedings of the 6th Conference on Machines, Computations and Universality (MCU).* Zürich, Switzerland, 2013, pp. 16–22. DOI: 10.4204/EPTCS.128.5.

[WS98] D. J. Watts and S. H. Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393 (1998), pp. 440–442. DOI: 10.1038/30918.

[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.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. "Modular Self-Reconfigurable Robot Systems". In: *IEEE Robotics and Automation Magazine* 14.1 (2007), pp. 43–52. DOI: 10.1109/MRA.2007.339623.

[Zha+04] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784.